Razvoj korisničkog sučelja web aplikacija uz primjenu web komponenti i okvira

Kranjec, Filip

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Filip Kranjec

RAZVOJ KORISNIČKOG SUČELJA WEB APLIKACIJA UZ PRIMJENU WEB KOMPONENTI I OKVIRA

ZAVRŠNI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Filip Kranjec

Matični broj: 45888/17–R

Studij: Informacijski sustavi

RAZVOJ KORISNIČKOG SUČELJA WEB APLIKACIJA UZ PRIMJENU WEB KOMPONENTI I OKVIRA

ZAVRŠNI RAD

Mentor:

Prof. dr. sc. Kermek Dragutin

Varaždin, rujan 2022.

Filip Kranjec

Izjava o izvornosti

Izjavljujem da je moj završni 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 rada je opis struktura web aplikacija, opis korištenih tehnologija za izradu web aplikacije. Opis uloga HTML, CSS, JavaScript te TypeScript. Razlike između JavaScript- a i TypeScript-a. Prednosti i mane TypeScripta i JavaScripta. Detaljni opis uloga web komponenti prilikom izrade web aplikacija.Opis React-a, React Router-a. Usporedba React-a sa ostalim okvirima (Angular, Vue). Praktični dio rada odnosi se na izradu web aplikacije putem okvira React koristeći TypeScript, MonogDB za izradu noSQL baze podataka te ExpressJS za izradu aplikacijskog programskog sučelja. Aplikacije koja se izrađuje jest online burza rada za sezonske radnike unutar koje je moguće pronaći posao te objaviti radno mjesto.

Ključne riječi: web; web aplikacija; korisničko sučelje; razvoj; html; css; typescript; okvir; web komponenta;

Sadržaj

1 Uvod

U ovom radu opisati će se što znači razvoj aplikacija pomoću web komponenti i okvira te detaljno opisati i objasniti što su web komponente i koje uloge imaju u razvoju web aplikacija. Također će biti pojašnjeno što su to web okviri te čemu služe i koje su razlike između onih najkorištenijih.

Nakon tumačenja web komponenti i web okvira opisati će se tehnologije koje su korištene za izradu web aplikacije, a to su za klijentsku stranu React biblioteka unutar koje će se koristiti TypeScript kako bi kod bio stabilniji, skalabilniji te čitljiviji i jasniji, te dodatne biblioteke koje pomažu u radu sa React-om kao što su React Router pomoću kojega određujemo navigaciju unutar aplikacije te autorizaciju pristupa određenim stranicama za određene korisnike te React Redux za upravljanje stanjima i podacima unutar aplikacije prilikom implementacije kompleksnijih arhitektura aplikacije. Nakon toga će se objasniti nešto malo više o poslužiteljskoj strani na kojoj će se koristiti Express te MongoDB za bazu podataka. Sa navedenom kombinacijom tehnologija moguće je napraviti veoma robusnu i skalabilnu web aplikaciju pošto tehnologije dopuštaju da klijentsku stranu, ali i poslužiteljsku stranu podijelimo na manje dijelove koje je moguće ponovno koristiti na određenim dijelovima aplikacije.

Za izradu web aplikacije potrebno je opširno znanje ne samo o tehnologijama koje se koriste nego i o nekima od principa sigurnosti i zaštite web aplikacije. Također je važno napomenuti da je dizajn jedna od najbitnijih stavki na strani klijenta pošto sučelje mora biti pristupačno te intuitivno.

2 Web aplikacije

Web aplikacije su programski sustavi koji se nalaze na udaljenom poslužitelju te im korisnici mogu pristupiti koristeći preglednike. Web aplikacija se u većini slučajeva sastoji od dva osnovna dijela, a to su: strana klijenta (engl. Front-end) i strana poslužitelja (engl. Back-end).[1]

Pod stranom klijenta smatra se onaj dio koji korisnik može vidjeti te s kojim ima interakciju. Na strani klijenta se prikazuju svi podaci koji se spremaju te obrađuju na strani poslužitelja. Za programiranje na strani klijenta najčešće se koriste HTML, CSS te skriptni jezik JavaScript.

Strana poslužitelja kod web aplikacija se također sastoji od dva dijela, a to su baza podataka i aplikacijsko sučelje koje je zapravo poveznica između baze podataka i sučelja za korisnike. Baza podataka služi za strukturirano spremanje podataka što omogućuje pamćenje korisnika te općenito izgradnju sustava unutar kojega korisnik može imati svoj osobni profil sa podacima koji su zapamćeni na udaljenom poslužitelju dok aplikacijsko sučelje koristi za dohvaćanje, kreiranje, ažuriranje te brisanje tih istih podataka točnije za izvršavanje operacija nad bazom podataka te posluživanja tih podataka klijentu ovisno o tome kada ih klijent zatraži.

2.1 Web komponente

Web komponente su samostalni dijelovi koda koje je moguće dijeliti i ponovno koristiti koji uz vizualni dio nude i mogućnosti prosljeđivanja podataka u komponentu te dodatnih opcija kojima se komponenta prilagođava točno onome što je potrebno. [2, str. 3]

Korištenjem web komponenti omogućeno je kreiranje samostalnih komponenti poput: gumbova, kalendara, satova, polja za unošenje teksta, kartica korisnik i još mnogih koje je moguće koristiti u bilo kojem dijelu aplikacije. Također postoje dodatne mogućnosti prosljeđivanja dodatnih opcija te onda na primjer kada se implementira gumb moguće je postaviti opciju odabira boje gumba na način da se proslijedi određena vrijednost unutar HTML elementa na način kako se dodjeljuju atributi svakom elementu. Na taj način je također moguće proslijediti točne podatke koje komponenta treba ispisivati kao na primjer naslov gumba.

Web komponente se temelje na četiri glavne tehnologije, a to su: sjenoviti DOM (engl. Shadow DOM), ES moduli (engl. ES Modules), HTML predlošci (engl. HTML Template) i prilagodljivi elementi (engl. Custom Elements) [2, str. 6].

Pomoću sjenovitog DOM-a se postiže enkapsulacija komponente te se komponenta štiti od okolnih dijelova aplikacije. Na primjer ako se kreira komponenta za upisivanje nove tekstualne vrijednosti za to se koristi HTML element input te ako se koristi Shadow DOM unutar alata za programere u pregledniku vidjeti će se samo oznaka HTML elementa input te njegov atribut type, dok će se unutar te komponente zapravo nalaziti dosta više elemenata.

Tehnologija prilagodljivih elemenata omogućuje kreiranje nove HTML oznake te pridruživanje određenog uzorka istom.

HTML predlošci su zapravo HTML oznake template unutar kojih se navode ostali elementi koje je potrebno pridružiti prilagodljivom elementu.

ES moduli jest tehnologija koja je promijenila način korištenja web komponenata. Dolaskom ECMAScript 6 značajki JavaScript jezika korištenje web komponenti postalo je vrlo jednostavno iz razloga što je moguće implementirati komponente tako da svaka od njih zapravo samo brine o tome što je unutra uvedeno i što će se unutar nje koristiti. Takav način rada omogućava razdvajanje koda na manje cjeline te ponovno korištenje koda jednostavnom naredbom import.

2.2 Web okviri

Web okviri su aplikacijski okviri napravljeni da bi olakšali pokretanje i izgradnju web aplikacija koristeći posebne značajke koje su drugačije od okvira do okvira. Kod web okvira postoje dvije vrste istih, a to su: web okviri za programiranje na strani poslužitelja i web okviri za programiranje na strani klijenta.[3]

Neki od najpoznatijih web okvira za programiranje na strani poslužitelja su: Django (Python), Ruby on Rails (Ruby), Express (JavaScript), Symfony (PHP). Unutar tih okvira su već implementirane neke od funkcionalnosti poput čitanja podataka sa računala, upravljanje HTTP zahtjevima, upravljanje bazom podataka i još mnogih funkcionalnosti.

Kod web okvira za programiranje na strani klijenta neki od poznatijih i korištenijih u današnje vrijeme su: Angular, Vue, Ember te React koji se čak i ne smatra okvirom nego bibliotekom jer ne pruža okolinu za razvoj kao na primjer Angular unutar kojega je moguće kreirati nove komponente, servise, module pomoću naredbi. Također unutar React-a nema nekih od već implementiranih funkcionalnosti poput navigiranja kroz aplikaciju nego je potrebno istu dodati unutar projekta kao vanjsku biblioteku.

Za razliku od sustava za upravljanje sadržajem unutar web okvira izgradnja aplikacije počinje od nule, a programer mora sam implementirati (engl. implement) sve funkcionalnosti koje su potrebne da bi web aplikacija radila.

2.3 Prikazivanje na strani poslužitelja protiv prikazivanja na strani klijenta

Najveći problem u današnje vrijeme kod izgradnje web aplikacija jest vrijeme koje je potrebno da se aplikacija prikaže te zbog vrlo velikih i zahtjevnih korisničkih sučelja vrijeme prikazivanja postaje sve duže. Kod prikazivanja web stranice postoje dva različita načina, a to su prikazivanje na strani klijenta (engl. Client side rendering) te prikazivanje na strani poslužitelja (engl. Server side rendering).

Prikazivanje na strani klijenta postao je dosta korišten način prikazivanja iz razloga što većina web okvira za izradu web aplikacija koristi način prikazivanja na strani klijenta te je zapravo prikazivanje na strani poslužitelja dodatna funkcionalnost. Na primjer Angular te React koriste prikazivanje na strani klijenta, ali moguće je instalirati dodatne biblioteke koje omogućuju prikazivanje na strani poslužitelja kao što su angular-universal te Next.js. Također kod prikazivanja na strani klijenta zahtjev se prosljeđuje unutar jedne HTML datoteke te poslužitelj prikazuje predložak te datoteke bez ikakvih podataka sve dok preglednik ne priredi sve što je potrebno da bi se podaci prikazali. Prednost takvog prikazivanja jest u tome što će početno učitavanje aplikacije možda potrajati malo duže, ali daljnje korištenje aplikacije postaje brže zato što preglednik ima cijelu aplikaciju već učitanu i spremnu za prikazivanje.[4]

Kod prikazivanja na strani poslužitelja poslužitelj učitava HTML datoteku te se stranica prikazuje tek kada je sve učitano.[4] Tu je dolazilo do problema jer bi se prilikom promjene

stranice svi podaci morali ponovno učitavati te bi to dosta usporilo rad same aplikacije. Također prednost kod prikazivanja na strani poslužitelja jest u tome što je moguće napraviti vrlo kvalitetnu optimizaciju tražilice (engl. Search Engine Optimization) što omogućuje jednostavnije pronalaženje web mjesta, dok je optimizaciju tražilice skoro nemoguće napraviti kod prikazivanja na strani klijenta.

	Prikazivanje na strani poslužitelja	Prikazivanje na strani klijenta
Optimizacija tražilice	Jednostavno implementiranje optimizacije tražilice iz razloga što sadržaj postoji prije nego što se prikaže korisniku.	Vrlo teško implementiranje optimizacije tražilice pošto sadržaj nije učitan prije nego što dođe do korisnika
Inicijalno učitavanje	Brže inicijalno učitavanje aplikacije, ali svako promjenom putanje dolazi do ponovnog učitavanja cijele stranice.	Sporije inicijalno učitavanje iz razloga što se aplikacije učitava u potpunosti te se svakom promjenom putanje samo izmjenjuje sadržaj koji se prikazuje korisniku.
Potreba za vanjskim bibliotekama	Nema potrebe za vanjskim bibliotekama	Potrebne su vanjske biblioteke

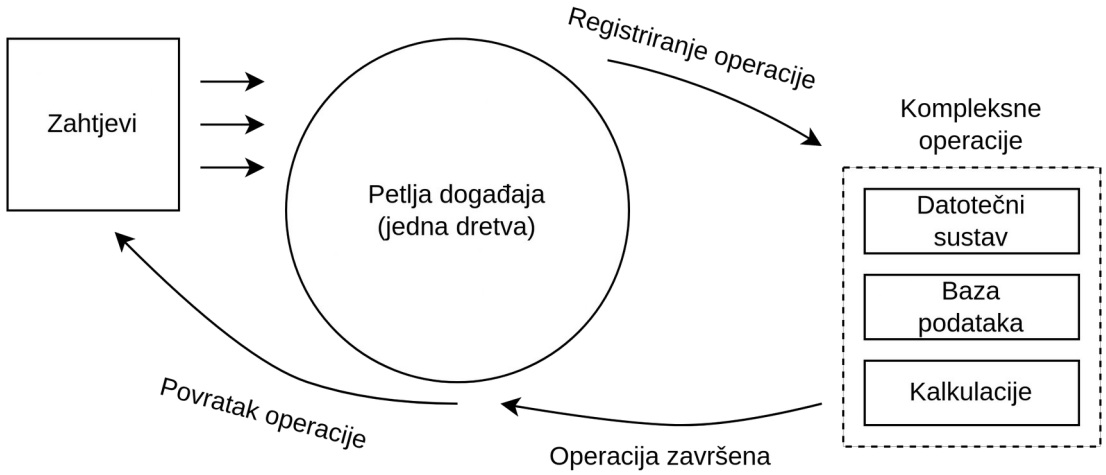
Tablica 1: Prikazivanje na strani poslužitelja protiv prikazivanja na strani klijenta (izvor: vlastita izrada)

3 Tehnologije korištene za izradu

Odabir tehnologija za izradu aplikacija je jedan od bitnijih koraka kod izrade web aplikacije. Tehnologije se dijele na tehnologije za programiranje na strani klijenta te na tehnologije za programiranje na strani poslužitelja. Danas je dostupan veliki broj web okvira i biblioteka koje omogućuju olakšanu izradu web aplikacije. Kod programiranja na strani klijenta trenutno najpopularniji okviri su Angular, Vue i React dok na strani poslužitelja postoji još mnogo više tehnologija poput PHP-a, Java-e, C#-a, Python-a i još mnogih. To su većinom tehnologije za izradu središnjeg dijela aplikacije koji se povezuje sa bazom podataka. Također je potrebno odabrati i pravu bazu podataka, a neke od njih su PostgreSQL, MySQL, MongoDB, Mnesia, Cassandra i još mnoge. Najveća razlika kod odabira baze podataka jest to da li nam treba relacijska ili ne relacijska baza podataka.

3.1 Node.js

Node.js je JavaScript platforma koja je bazirana na asinkronim događajima te koristi samo jednu dretvu. Jako je poznat baš iz razloga što koristi jednu dretvu koja procesira sve zahtjeve. Node poslužitelj radi na principu ne blokirajućih ulaza i izlaza što znači da kada dođe jedan zahtjev dretva će krenuti u izvršavanje toga zadatka te je odmah nakon spremna primiti novi zahtjev i izvršiti ga. Korištenjem neblokirajućih ulaza i izlaza uspiješno je omogućeno to da dretva više nikada ne dolazi do stanja čekanja nego uvijek procesira.[5]



Slika 1: Prikaz rada Node.js (izvor: https://codeburst.io/the-only-nodejs-introduction-youll-ever-need-d969a47ef219)

Node može služiti za kreiranje, brisanje, pregledavanje i obnavljanje datoteka na serveru, generiranje dinamičkih sadržaja za web aplikaciju te kreiranje, pregledavanje, ažuriranje, brisanje podataka iz baze podataka.

Vrlo je popularan za korištenje pošto je programeru omogućena izrada potpune aplikacije koristeći samo JavaScript na primjer ako se koristi React za stranu klijenta te Express.js u kombinaciji sa MongoDB tada cijelu aplikaciju izrađujemo koristeći samo jedan programski jezik. Također je poznat po tome što koristi V8 JavaScript pokretač (engl. engine) koji je poznat po tome da je veoma brz.Node.js također se može pohvaliti sa svojom brzinom.

Za instalaciju novih biblioteka unutar aplikacija i projekata koje pokreće Node koristi se npm (Node package manager). Npm omogućuje instalaciju, brisanje te ažuriranje svih biblioteka i njihovih verzija koje koristimo za izradu web aplikacije.

3.2 React.js

React je Javascript biblioteka koja se koristi u svrhu izrade jednostraničnih aplikacije. Napisan je od strane Facebook te njihovog inžinjera Jordan Walkea koji je 2011. godine kreirao FaxJS točnije rani prototip React-a. Kasnije je 2013. godine na konferenciji JS ConfUS prvi put je predstavljen React te je postao biblioteka otvorenog koda. [6]

Najveća prednost React-a je to što se pomoću njega izgrađuju jednostranični sustavi koji omogućuju prikazivanje novih podataka bez ponovnog učitavanja stranice. Također je jedna od prednosti što React dopušta da se kreira web aplikacije sa prikazivanjem na strani klijenta te uz pomoć Next.js što je inačica React-a moguće je kreirati web aplikacije sa prikazivanjem na strani poslužitelja.

Bitno je napomenuti da React koristi JSX, što označava JavaScript XML. JSX je sintaksa koja se koristi izričitno unutar React-a. Za razliku od nekih drugih okvira React omogućava da se cijeli kod za jednu komponentu piše unutar jednog dokumenta, dok na primjer kod Angular-a za jednu komponentu se koristi nekoliku dokumenata .[7]

Jednostavan isječak koda koji prikazuje jednostavnu komponentu unutar koje se prikazuje naslov:

```
const Naslov = () => {
      let naslov = "NASLOV";
      return (
            <h1>{naslov}</h1>
      );
};
export default Naslov;
```

Kod spajanja više komponenti u jednu kompleksniju komponentu najbolje je koristiti paradigmu roditelj dijete te se u tom slučaju podaci šalju u jednom smjeru od roditelja prema djetetu što omogućuje da se u jednom smjeru šalju podaci, a u drugom događaji koji će onda napraviti promjenu u komponenti koja je roditelj te će se sve te promjene odmah prikazati za sve komponente kojima je potrebno.

Kada se koristi paradigmu roditelj dijete tada bi prethodno prikazana komponenta Naslov bila dijete koje prima parametre, a to bi izgledalo ovako:

```
const Naslov = (props) => {
      return (
      <h1>{props.naslov}</h1>
      );
};
export default Naslov;
```

Dok bi roditelj komponenta u sebi imala varijablu naslova te bi inicijalizirala komponentu Naslov na ovaj način:

```
const Kartica = () => {
      let naslov = "NASLOV";
      return (
            <div>
            <Naslov naslov={naslov}/>
            </div>
```

```
        );
};
  export default Kartica;
```

U primjeru je naveden jedan način na koji se podaci prosljeđuju u komponentu. Postoji još nekoliko načina upravljanja stanjima u komponentama, a to su pomoću React Contexta te pomoću biblioteka kao što su MobX ili React Redux koji omogućuju da se na jednostavniji način pristupa podacima unutar svake komponente.

Jedna od najbitnijih značajki su udice (engl. React Hooks) koje omogućavaju promjenu stanja komponente tijekom izvođenja aplikacije, a jedna od najpoznatijih je UseEffect koja se ponaša više kao životni ciklus zato što se izvršava svaki put kada se komponenta inicijalizira, ali unutar te udice je moguće također proslijediti niz ovisnosti (engl. Dependency array) u koji se spremaju varijable o kojima ovisi pokretanje određenih funkcija unutar useEffect-a. To znači da ako se varijabla koja je u nizu ovisnosti promjeni da će se sve funkcije koje su zadane u useEffect-u ponovno izvršiti. Kod useEffect-a treba posebno paziti zato što je vrlo lako napraviti beskonačnu petlju. Nadalje postoji useState koji zapravo dopušta da kada se dogodi promjena stanja odmah to novo stanje prikažemo krajnjem korisniku, te imamo useMemo koji se najčešće koristi za kreiranje privremenih podataka u komponenti. Isječak koda koji opisuje korištenje useEffect-a možemo vidjeti u nastavku.

```
const Kartica = () => {
      let naslov = "";
      useEffect(()=>{
            naslov = "Naslov";
            return ()=>{}
      },[])
      return (
      <div>
            <Naslov naslov={naslov}/>
      </div>
      );
};
export default Kartica;
```

Na taj način je omogućeno da svaki puta kada se komponenta otvori varijabli naslov

dodijelimo određenu vrijednost. Nadalje dio funkcije `return ()=>{}` predstavlja funkciju koja se poziva kada se komponenta uništava. U prijašnjim verzijama React-a funkcija useEffect se zvala componentDidMount.[7]

3.2.1. React Router

React router je biblioteka koja se koristi za navigiranje kroz web aplikaciju. Trenutna verzija je V6 koja ima dosta novih funkcionalnosti uvedeno, ali one najbitnije su navigiranje direktno do neke stranice, navigiranje kroz povijest, dohvaćanje trenutnih parametara, dohvaćanje trenutne adrese i još mnogi.[8]

Najosnovniji tip navigiranja bi bilo navigiranje do određene rute na primjer ako aplikacija ima tri stranice (početna, o nama, povijest) tada će se navigirati točno do te određene rute na primjer /pocetna ili /onama ili /povijest. Nadalje postoji tip navigiranja gdje se koriste ugniježdene rute što bi značilo da na svakoj od navedenih stranica postoji još pod stranica te bi u tom slučaju rute bile nešto poput /pocetna/dodaj ili /pocetna/obrisi.

Kod React Router-a postoje mogućnosti da se kreira jednu rutu na koju aplikacija vodi ako korisnik hoće pristupiti ruti koja ne postoji, te također postoje zaštićene rute pomoću kojih se dopušta pristupanje samo određenim korisnicima te se radi dodatni sloj oko osnovne rute koji provjerava je li trenutni korisnik prijavljen te je li određeni pristupni kod valjan.

3.2.3 Usporedba React.js-a s Angular-om

Najveća razlika između React-a i Angulara jest u tome što Angular već u početku određuje otprilike kako bi se datoteke trebale organizirati u projektu dok kod React-a nije određeno kako treba biti organizirano.[9] Također je bitno napomenuti da Angular ima veliki dio biblioteka odmah ugrađen prilikom instalacije paketa pomoću npm-a. Na primjer u React-u ako je potrebno napraviti navigiranje kroz stranicu treba instalirati React Router dok kod Angulara ta biblioteka već postoji u paketu od Angulara.

Jedna od razlika jest to što Angular koristi Dvosmjerno vezanje podataka (engl. Two-way data binding), a React koristi Jednosmjerno vezanje podataka (engl. One-way data binding). Razlika je u tome što unutar Angulara komponente imaju ulaze (engl. Input) i izlaze (engl. Output) te je pomoću njih na jednostavan način moguće implementirati kontorliranu komponentu, dok se kod React-a koriste dodatne funkcije koje se prosljeđuju u komponentu kako bi rekreirale događaje koji pokreću metode iz komponente roditelja.

Poslovna logika i korisničko sučelje su kod Angular-a podjeljeni na način da se sva poslovna logika stavlja unutar servisa koje onda komponenta koristi putem injektiranja ovisnosti (engl. Dependency injection), a dizajn korisničkog sučelja se sprema unutar same komponente. Kod React-a se poslovna logika i dizajn korisničkog sučelja nalaze u istom dokumentu.

Jedan od problema kod React-a je to što je dosta teško odvojiti korisničko sučelje od poslovne logike aplikacije pošto se upravljanje stanjima događa direktno unutar komponenata dok kod Angulara postoje servisi koji se mogu injektirati unutar komponente (eng. Dependency Injection). Injekcija je trenutno podržana u React-u, ali ne potpuno.[9]

3.3 HTML5

HTML, skraćeno za HyperText Markup Language jest jezik oznaka koji se koristi za izradu web stranica razvijen od strane Tim-a Berners-Lee-a 1989. godine.

Sintaksa HTML-a je vrlo jednostavna te se većina elemenata sastoji od oznake za otvaranje i oznake za zatvaranje elementa. Sadržaj elementa se dodaje između oznaka te može biti tekst ili neki drugi element.

```
<!DOCTYPE html>
<html>
	<head>
		<title></title>
	</head>
	<body>

	</body>
</html>
```

U primjeru je prikazan kostur HTML dokumenta. Početna verzija u sebi ima elemente html, head i body. Unutar elementa head uvode se podaci o stranici te se dodaju meta podaci koji su kasnije potrebni zbog optimizacije pokretača pretraživanja. Nadalje unutar elementa body dalje se nastavlja graditi HTML stranica.

Postoje dvije vrste elemenata: blok elementi i linijski elementi. Blok elementi su elementi koji se prikazuju u novoj liniji neovisno o elementu ispred te će također sljedeći element biti u novoj liniji dok su se linijski elementi dodaju jedan pored drugog. Neki od blok elemenata su: <header>, <section>, <footer>, , <p>, <li>, <nav> dok su neki od linijskih elemenata , <input>, <label>, i još mnogi. [10]

Svaki element u HTML-u također ima atribute kojima se pobliže određuje za što se koriste. Neki od poznatijih atributa su: class, id, href, name, type, method, action.

3.4 CSS

Pomoću HTML-a bilo je moguće napraviti samo skromne stranice bez ikakvog dizajna pa je iz tog razloga bilo potrebno osmisliti proširenja pomoću kojih je moguće dodavati stilove na elemente. To je ostvareno pomoću CSS-a ili kaskadnih stilskih listova. Kroz povijest su postojale tri verzije, a danas se koristi treća pod nazivom CSS3.

CSS je jezik stilskih listova pomoću kojega se određenim elementima u HTML dokumentu dodjeljuju stilovi. Postoje tri načina za povezivanje CSS-a sa HTML-om. Prvi način jest da se unutar HTML dokumenta nadoda oznaku <style> te unutar nje zapisuju stilovi. Drugi način je da se u posebni dokument zapisuju CSS stilovi te da se taj dokument poveže sa HTML dokumentom unutar oznake <head> pomoću oznake <link>, te je treći način da se elementu koji treba stilizirati nadoda atribut style unutar kojega se zapisuju stilska pravila samo za taj element.[11] Sintaksa CSS-a izgleda ovako:

```
div {
    background: green;
}
```

Također kod CSS-a postoje selektori koji služe za povezivanje HTML elementa sa stilskom uputom. U prethodnom primjeru korišten je implicitni selektor pomoću kojega se dohvaćaju elementi prema samom nazivu elementa. Nadalje postoji eksplicitni selektor ili selektor klase pomoću kojega se dohvaća HTML element koji ima vrijednost atributa class jednaku selektoru.

```
.klasa {
    background: green;
}
```

Postoji jednoznačni selektor pomoću kojega se HTML element dohvaća putem atributa id.

```
#jednoznacno {
    background: green;
}
```

3.5 TypeScript

TypeScript je jezik koji je nadskup JavaScript-a što znači da se može koristiti sve što se koristi i u JavaScript-u, ali također i dodatne funkcionalnosti.[12] Izumio ga je Andreas Hejlsberg, izumitelj C#-a, za vrijeme dok je radio u Microsoft-u. Najbitnija dodatna funkcionalnost TypeScript-a jesu tipovi podataka koji omogućuju dodavanje tipova varijablama što nije bilo moguće napraviti unutar JavaScript-a. Dodavanjem tipova podataka unutar TypeScript-a omogućeno je skalabilnije korištenje skriptnog jezika te lakše čitanje koda od stane osobe koja ga nije napisala.

Sintaksa TypeScript-a je ista kao i kod JavaScript-a jedino što je potrebno varijablama i funkcijama dodijeliti tipove što se radi pomoću operatora dvotočka te nakon nje napisan tip koji može biti number, string, boolean, array ([]), unknown, null, undefined te any pomoću kojega varijabla može biti bilo kojeg tipa.

```
var broj: number = 1;
var niz: number[] = [1,2];
```

Tip podataka je također moguće kreirati pomoću sučelja (engl. Interface) te dodijeliti varijabli taj tip podatka.

```
Interface Sucelje {
      broj: number
      rijec: string
      niz: number[]
      drugoSucelje: DrugoSucelje
}

Interface DrugoSucelje {
      sve: any
}
```

Kod sučelja je također moguće proširivanje sučelja isto kao i kod klasa te je na taj način moguće implementirati polimorfizam.

```
Interface Sucelje {
      broj: number
```

```
            rijec: string
            niz: number[]
        }

Interface SuceljeDodatno extends Sucelje {
    nepoznato: unknown
}
```

Uz sučelja moguće je napraviti i enumeracije pomoću ključne riječi enum. Svakoj vrijednosti unutar enumeracije moguće je dodijeliti broj ili tekst kojim je tu vrijednost moguće dohvatiti.

```
    enum enumeracija {
        JEDAN = 1,
        DVA = 2,
        TRI = 3
    }
```

Unutar TypeScript-a također je moguće dodati povratni tip podatka metodama što znači da točno određujemo što nam koja metoda vraća. Tip podatka koji metoda vraća dodjeljuje se na isti način kako se dodjeljuje tip podatka varijabli. Zbog preglednosti i čitljivosti koda predlaže se da je svakoj funkciji dodijeljen povratni tip te ako funkcija ne vraća nikakvu vrijednost potrebno je staviti void tip podatka.

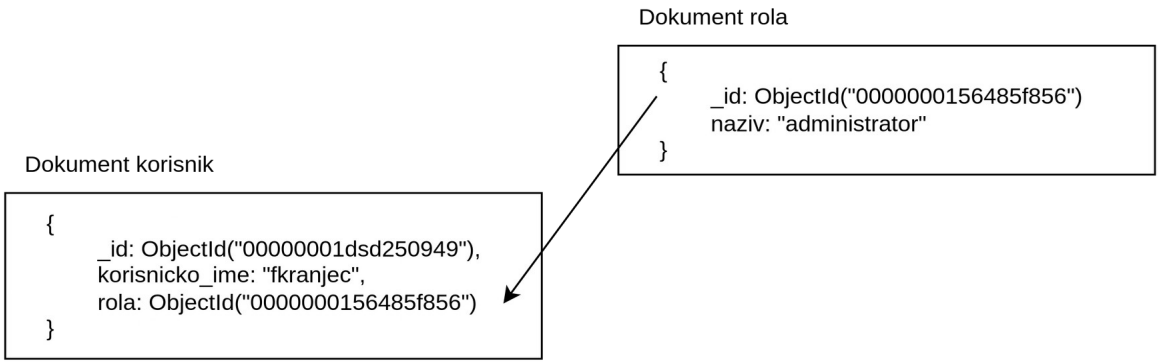
```
    function metoda(parametar: number): number {
return parametar;
    }
```

Kod metoda unutar TypeScript-a je moguće parametre postaviti da budu opcionalni što znači da parametar nije obavezan za unesti te ako nije predan kod poziva metoda njegova vrijednost je undefined. Da bi parametar bio opcionalan potrebno je staviti operator upitnik neposredno nakon naziva parametra.

3.6 MongoDB

MongoDB je baza podataka orijentirana na dokumente. To znači da umjesto tablica kao kod relacijski baza koriste se dokumenti. Unutar MongoDB-a svaka baza se sastoji od kolekcija koje sadrže dokumente. Svaki dokument može biti različit te veličina tih dokumenata smije biti različita.[13]

Struktura tih dokumenata je vrlo slična JSON objektima što programerima olakšava rad sa podacima te je također moguće cijeli programski kod za izradu baze podataka napisati u JavaScript-u što, kao što je već navedeno, programerima olakšava posao pošto je dovoljno da nauče jedan programski jezik koji im je dovoljan za napraviti potpunu web aplikaciju.



Slika 2: Prikaz strukture podataka unutar baze (izvor: vlastita izrada)

Također je važno napomenuti da unutar dokumenta nije potrebno navoditi poseban ID za svaki novi dokument nego se on automatski dodaje pod nazivom _id te ga je moguće koristiti isto kao i ID koji programeri sami dodaju unutar tablice ili dokumenta.

3.7 Express.js

Express je web okvir za Node.js koji služi za izradu Restful web servisa. Pomoću njega je moguće na vrlo jednostavan način napraviti vrlo moćan web servis. Express je u današnje vrijeme vrlo popularan pošto većina programera teži tome da se cijela web aplikacija (strana poslužitelja i strana klijenta) implementira pomoću jednog programskog jezika, JavaScript-a, što je pomoću Express-a i moguće. Iako je zapravo Node sasvim dovoljan za razvijanje na strani poslužitelja Express se koristi zato što ima neke od kompleksnijih funkcionalnosti već implementirane.[14]

Jedan osnovni primjer implementacije web servisa u Express-u bi izgledao ovako:

```
const express = require('express');
const  aplikacija = express();
const port = 3000;

aplikacija.get('/', (req) => {
     console.log("Pozdrav");
})

aplikacija.listen(port, ()=>{
     Console.log(`Slušam na portu ${port}`);
})
```

U navedenom primjeru je prikazana jedna putanja (engl. route) koja je osnovna početna ruta. Kada se aplikacija pokrene pomoću biblioteke nodemon koja omogućuje da se pokrenuta aplikacija neprestano izvršava u pozadini lokalno, prvo će se ispisati 'Slušam na portu 3000' jer se metoda listen iz okvira express poziva samo kada je aplikacija prvi put pokrenuta dok se metoda get poziva svaki put kada se pristupi ruti localhost:3000/ te će se u tom slučaju ispisati 'Pozdrav'.

3.8 GraphQL

GraphQL je jezik kojim se pišu upiti za web servis te pokretač koji te upite izvršava nad postojećim podacima.[15] Jedna od velikih prednosti GraphQL-a jest to što se prilikom implementacije web servisa putem shema cijeli web servis automatski i dokumentira te je unutar dokumentacije moguće pronaći upite, mutacije, tipove i skalare. Mutacije unutar GraphQl-a bi se mogli usporediti sa POST zahtjevima dok su upiti GET zahtjevi. Pomoću mutacija se unose podaci u bazu podatka, ažuriraju i brišu dok upiti služe izričito za dohvaćanje ili svih podataka ili točno određenih podataka.

GraphQL je također specifičan iz razloga što su svi podaci dostupni na jednoj ruti koja je u većini slučajeva /graphql dok je kod restful web servisa potrebno raditi mnogo različitih ruta te svaka dohvaća samo jednu vrstu podataka.[15]

Također se kod GraphQL-a koriste tipovi podataka tako da se osigura da se traže samo podaci koji stvarno postoje. Primjer definiranja tipa podataka izgleda ovako:

```
Type Korisnik {
      Ime: String
      Prezime: String
      Rola: Rola
}

Type Rola {
      Naziv: String
}
```

Prema navedenim tipovima upit za dohvaćanje svih podataka o korisniku bi izgledao ovako :

```
Query dohvatiKorisnike{
      Korisnik{
            ime
            prezime
            rola {
                  naziv
```

```
                }
            }
        }
```

Ako u upitu nije potrebno na primjer ime korisnika jednostavno ga se niti ne navede unutar upita te se ime korisnika ne bude niti dohvatilo.

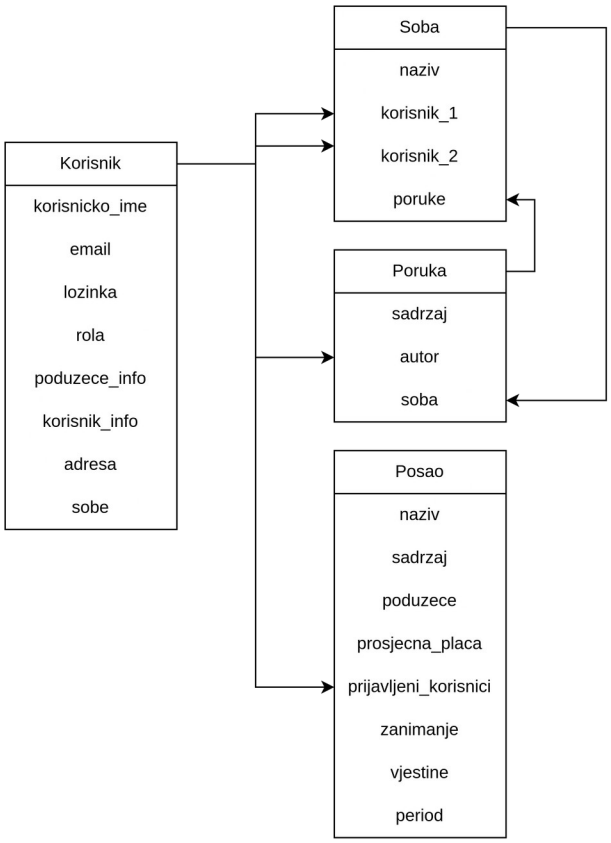
Jedna od popularnijih mogućnosti GraphQL-a su također pretplate koje omogućavaju da se na određenom dijelu na strani klijenta sluša na određene akcije koje su definirane prilikom implementacije pretplate. Tako je na primjer moguće implementirati razgovore ili notifikacije unutar aplikacije.

Svi upiti, tipovi podata, ulazni tipovi podataka, mutacije i pretplate se definiraju unutar GraphQL sheme. Da bi poslužitelj znao koji tip podataka je potrebno vratiti uz određeni upit ili koji tip argumenta prima mutacija potrebno je unutar sheme sve definirati na sljedeći način:

```
type Korisnik {
     ime: String
     prezime: String
}

type Query {
     dohvatiKorisnike : [Korisnik]
}

type Mutation {
     dodajKorisnika(ime:String, prezime: String): Korisnik
}
```

4 Izrada Aplikacije

U sljedećem dijelu je prikazana izgradnja aplikacije za zapošljavanje i objavljivanje radnih mjesta. Početna stranica koja se prikazuje kod pristupanja jest stranica na kojoj se odabire želi li se korisnik registrirati ili prijaviti ili registrirai poduzeće. To znači da nije moguće ništa pregledavati unutar aplikacije ako korisnik nije registriran. Nakon registracije i prijave korisnik ima mogućnosti pregledavati objavljene poslove te podatke o tim poslovima kao što su lokacija, plaća, početak i kraj rada te zahtjeve koji su poželjni, ali ne i nužni. Korisnik također može prilagođavati svoj profil tako što može dodati jezike koje govori, vještine koje posjeduje, prethodne poslove te edukaciju koju je prošao. Kada je korisnik prijavljen kao poduzeće tada on nema mogućnost prijavljivanja na posao, ali svejedno može pregledavati poslove, kreirati nove poslove, započeti razgovor sa korisnikom koji se prijavio na posao i pregledati sve korisnike koji su se prijavili za posao.



Slika 3: Prikaz modela baze podataka (izvor: vlastita izrada)

4.1 Izrada baze podataka

Za izradu MongoDB baze podataka korištena je biblioteka mongoose kao što je već navedeno te se unutar nje uz pomoć TypeScript-a kreiraju sheme koje se pretvaraju u modele koji zatim određuju kakva će biti struktura dokumenta. [16] Pošto se shema implementira pomoću TypeScript-a također je moguće shemi proslijediti sučelje kojemu shema odgovara što znači da svaka akcija koja se izvodi nad shemom uvijek se izvodi nad istim skupom podataka. Baza podataka koja je izrađena prilikom izrade aplikacije sastoji se od četiri sheme: User, Job, Message i Room.

```
npm install mongoose --save
```

Unutar sheme korisnik (engl. User) nalaze se podaci koji su vezani i za korisnika i za poduzeće te dodatno podaci koji su vezani samo za poduzeće i podaci koji su vezani samo za korisnika. Podaci vezani za korisnika i poduzeće su ugrađeni dokumenti (engl. Embedded documents) što znači da ta polja imaju dokument ugrađen u sebe umjesto da se radila nova shema za to polje. Također postoji i polje za vrstu korisnika koje je zapravo enumeracija te tom polju ograničavamo vrijednosti koje su u njega mogu upisati.

```
const userSchema = new Schema<IUser>({
      username: String,
      email: String,
      password: String,
      createdAt: String,
      image: String,
      registrationCode: String,
      isVerified: {
            type: Boolean,
            default: false
      },
      address: {
            city: String,
            postalCode: Number,
            street: String,
            streetNumber: String,
```

```
            state: String
        },
        userType: {
            type: String,
            enum: ['USER', 'ADMIN',       'COMPANY'],
            default: 'USER'
        },
        companyInfo: {
            companyName: String,
            numberOfEmployees: {
                type: String,
                enum: ['1 - 5', '6 - 10',
        '11 - 20', '21 - 40',                          '41 - 60',
    '61 -                                  100']
            },
            description: String,
            typeOfCompany: String,
        },
        userInfo: {
            firstName: String,
            lastName: String,
            education: [String],
            previousJobs: [String],
            languages: [String],
            skills: [String]
        },
        rooms: [
            {
                type:Schema.Types.ObjectId,
                ref: 'Room'
            }
        ]
    });
```

Iz prikazanog isječka koda može se vidjeti da unutar sheme nije potrebno navoditi id pošto se on sam dodaje na svaki dokument. Također postoji i polje za sobe unutar kojega se nalaze id-jevi od drugog dokumenta te je na taj način napravljena veza više naprema više koristeću ugrađene dokumente. Da bi se od te sheme napravio model potrebno je pozvati

funkciju model na sljedeći način:

```
export default const User = model('User', userSchema);
```

Pošto su oba dijela aplikacije izrađena koristeći TypeScript može se vidjeti da je klasi Schema moguće proslijediti generički tip podataka što znači da se tip podataka određuje za vrijeme izvođenja aplikacije, a ne za vrijeme prevođenja. Na taj način je proslijeđen tip podataka koji je definiran u istoj datoteci.

```
export interface IUser {
      id?: Types.ObjectId,
      username?: string,
      token?: string,
      email?: string,
      password?: string,
      createdAt?: string,
      image?: string,
      registrationCode?: string,
      isVerified?: boolean,
      address?: Address,
      userType?: USERTYPE | any,
      companyInfo?: CompanyInfo,
      userInfo?: UserInfo,
      rooms?: Types.ObjectId
}

enum USERTYPE {
      USER = "USER",
      COMPANY = "COMPANY",
      ADMIN = "ADMIN"
}

interface Address {
      city: string,
      postalCode: number,
      street: string,
      streetNumber: string,
      state: string
```

```
}

interface CompanyInfo {
      companyName: string,
      jobs: Jobs[]
}

interface UserInfo {
      firstName?: string,
      lastName?: string,
      education?: string[],
      previousJobs?: string[],
      languages?: string[],
      skills?: string[]
}
```

U prikazanom isječku moguće je vidjeti nekoliko sučelja od kojih se izgrađuje ono glavno sučelje za korisnika.

Nadalje shema posao (engl. Job) sastoji se od naziva posla, sadržaja koji se sastoji od naslova, teksta te podnožja. Sljedeće polje je referenca na dokument korisnik koje predstavlja poduzeće koje je objavilo radno mjesto. Također unutar iste sheme nalazi se i lista korisnika koji su se prijavili na radno mjesto te se sastoji od referenci opet na dokument korisnik. Sljedeća polja su zanimanje, lista vještina, vrijeme kreiranja, vrijeme do kada objava vrijedi, broj radnika koji je potreban te period obavljanja posla. Istom funkcijom kao i kod sheme korisnik kreira se model koji se izdvaja u modul.

```
const jobSchema = new Schema<Job>({
      name: String,
      content: {
            title: String,
            body: String,
            footer: String
      },
      company: {
            type: Schema.Types.ObjectId,
            ref: 'User'
```

```
        },
        averageSalary: {
              from: Number,
              to: Number
        },
        applied: [{
              type: Schema.Types.ObjectId,
              ref: 'User'
        }],
        occupation: String,
        skills: [String],
        timeCreated: Date,
        validTo: Date,
        workersNeeded: Number,
        period: {
              from: String,
              to: String
        }
});
```

Sljedeće dvije sheme su povezane iz razloga što služe za implementaciju razgovora unutar aplikacije. Shema poruka (engl. Message) sastoji se od sadržaja, autora poruke koji je referenca na shemu korisnik te sobe koja je referenca na shemu soba (engl. Room). Druga shema je već spomenuta shema sobe unutar koje se spremaju podaci o nazivu sobe, korisnicima koji sudjeluju u razgovoru, u svakom razgovoru može biti točno dva korisnika, te liste poruka koja sadrži reference na poruke.

```
const messageSchema = new Schema({
        content: String,
        author: {
              type: Schema.Types.ObjectId,
              ref: 'User'
        },
        room: {
              type: Schema.Types.ObjectId,
              ref: 'Room'
```

```
        }
});

const roomSchema = new Schema<IRoom>({
        name: String,
        userID1: {
                type: Schema.Types.ObjectId,
                ref: 'User'
        },
        userID2: {
                type: Schema.Types.ObjectId,
                ref: 'User'
        },
        messages: [{
                type: Schema.Types.ObjectId,
                ref: 'Message'
        }]
});
```

Svaki model koji je kreiran se izdvaja u modul te se pomoću tih modela izvršavaju CRUD operacije nad dokumentima. Najčešće korištene metode koje se nalaze unutar modela su: `find()`, `findOne()`, `findOneAndUpdate()` te `save()`.[11] Metoda `find()` služi za dohvaćanje liste dokumenata prema strukturi zadanoj u modelu, dok se pomoću metode `findOne()` dohvaća samo jedan dokument koji odgovara uvjetima koji su proslijeđeni kao parametar. Sljedeća metoda je `findOneAndUpdate()` koja je dosta slična prethodnoj metodi, ali dopušta prosljeđivanje još jednog argumenta unutar kojega se određuju vrijednosti koje će se ažurirati. Pomoću metode `save()` sprema se novi dokument u bazu.

4.2 Izrada aplikacijskog programskog sučelja

Prilikom izrade aplikacijskog programskog sučelja korišteno je nekoliko biblioteka te jedan web okvir. Web okvir je Express zato što se unutar njega nalazi veliki broj potrebnih alata za jednostavniju implementaciju određenih funkcionalnosti. Također unutar aplikacijskog programskog sučelja se koriste biblioteke koje su prikazane na slici ispod.

```
"dependencies": {
  "@graphql-tools/schema": "^9.0.1",
  "@sendgrid/mail": "^7.7.0",
  "@types/express-fileupload": "^1.4.1",
  "apollo-boost": "^0.4.9",
  "apollo-server-core": "^3.10.1",
  "apollo-server-express": "^3.10.1",
  "bcryptjs": "^2.4.3",
  "body-parser": "^1.20.0",
  "cors": "^2.8.5",
  "dotenv": "^16.0.0",
  "express": "^4.18.1",
  "express-fileupload": "^1.4.0",
  "graphql": "^15.8.0",
  "graphql-subscriptions": "^2.0.0",
  "graphql-tools": "^8.2.8",
  "graphql-upload": "^12.0.0",
  "graphql-ws": "^5.10.0",
  "jsonwebtoken": "^8.5.1",
  "mongodb": "^4.8.0",
  "mongoose": "^6.4.6",
  "multer": "^1.4.5-lts.1",
  "nodemailer": "^6.7.8",
  "pg": "^8.7.3",
  "validators": "^0.3.1",
  "ws": "^8.8.1"
},
"devDependencies": {
  "@types/bcryptjs": "^2.4.2",
  "@types/graphql-upload": "^8.0.11",
  "@types/jsonwebtoken": "^8.5.8",
  "@types/pg": "^8.6.5",
  "@types/ws": "^8.5.3",
  "ts-node": "^10.7.0",
  "tslint": "^6.1.3",
  "typescript": "^4.6.3"
}
```


Slika 4: Lista biblioteka web servisa (izvor: vlastita izrada)

Jedna od bitnijih biblioteka jest apollo-server-express iz razloga što se pomoću nje povezuju GraphQL sheme sa HTTP poslužiteljem koji je dostupan zbog web okvira Express.

Sljedeći korak je kreiranje početnog dokumenta pod nazivom index.ts unutar kojega se pokreće web servis. Potrebno je kreirati GraphQL sheme i razrješivače (engl. Resolvers) pomoću kojih se zatim instancira novi ApolloServer te se na taj način omogućuje korištenje GraphQL upita na strani klijenta. Također unutar web servisa postoje i dvije rute koje su REST strukture te služe za prenošenje slika i za verificiranje putem elektroničke pošte. Zadnji korak jest povezivanje na bazu podataka pomoću biblioteke mongoose.

```
const app = express();

const httpServer = http.createServer(app);

const wsServer = new WebSocketServer({
     server: httpServer,
     path: '/graphql'
})
```

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

const serverCleanup = useServer({ schema }, wsServer);

const server = new ApolloServer({
     schema,
     csrfPrevention: true,
     cache: 'bounded',
     context: ({ req }) => ({ req }),
})
await server.start();
app.use(fileUpload({ createParentPath: true }))
app.use(express.static('uploads'))
app.use(cors())
app.post('/upload-avatar', async (req, res) => {

})
app.get('/verify/:code', async (req, res) => {

})
server.applyMiddleware({
app,
path: '/graphql'
})
mongoose
.connect(process.env.DB_STRING, {})
.then((res: any) => {
console.log("MongoDB connected!");
})
```

Varijable typeDefs i resolvers se dobiju tako da se konstantama dodjele liste koje se sastoje od uvezenih shema i razrješivača. Unutar GraphQL shema se dokumentiraju upiti, mutacije, tipovi, skalari te pretplate (engl. Subscription). Sheme su složene tako da postoji dokument za svaki model pomoću kojega se kreira baza podataka što znači da unutar aplikacije postoje četiri GraphQL sheme. Da bi se sheme implementirale potrebni su razrješivači iz čega se može zaključiti da su također potrebna i četiri razrješivača.

Da bi se GraphQL sheme mogle implementirati potrebno je koristiti funkciju iz biblioteke

apollo-server-express pod nazivom gql te pomoću sljedeće sintakse odrediti koje su mutacije i upiti potrebni.

```
export const userTypeDefs: DocumentNode = gql`

    type Address {
          city: String
          postalCode: String
          street: String
          streetNumber: String
          state: String
    }

    type CompanyInfo {
          companyName: String
          numberOfEmployees: String
          description: String
          typeOfCompany: String
    }

    type UserInfo {
          firstName: String
          lastName: String
          education: [String]
          previousJobs: [String]
          languages: [String]
          skills: [String]

    type User {
          id: ID
          username: String
          image: String
          email: String
          token: String
          createdAt: String
          address: Address
          userType: String
          companyInfo: CompanyInfo
```

```
      userInfo: UserInfo
      rooms: [Room]
}

input LoginInput {
      username: String!
      password: String!
}

input RegisterInput {
      username: String!
      password: String!
      confirmPassword: String!
      email: String!
}

input AddressInput {
      city: String
      postalCode: String
      street: String
      streetNumber: String
      state: String
}
input CompanyInfoInput {
      companyName: String
      numberOfEmployees: String
      description: String
      typeOfCompany: String
}

input UserInfoInput {
      firstName: String
      lastName: String
      education: [String]
      previousJobs: [String]
      languages: [String]
      skills: [String]
}
```

```
input UserInput {
      username: String
      image: String
      email: String
      token: String
      createdAt: String
      address: AddressInput
      userType: String
      companyInfo: CompanyInfoInput
      userInfo: UserInfoInput
}

type Query {
      allUsers: [User]
      getUser(userId:ID!): User!
}

type Mutation {
      register(registerInput: RegisterInput!): User!
      login(username: String!, password: String!):User!
      updateUser(id:ID!, userInput: UserInput):User!
}`;
```

Iz isječka koda moguće je vidjeti da su definirane tri mutacije i dva upita dok tipova podataka ima više zato što je potrebno omogućiti što će sve biti moguće dohvatiti na strani klijenta iz određenog upita ili mutacije.

Za prikazanu shemu potrebno je implementirati razrješivače

```
const userResolvers: any = {
Query: {
     allUsers: async (_: any): Promise<IUser[]> =>   {
           try {
                 const users = await User.find();
                 return users;
           } catch (err) {
                 throw new Error(err);
           }
```

```
},
getUser: async (_: any, { userId }: any): Promise<IUser> => {
     try {
          console.log(userId)
          const user = await
     User.findById(userId);
          return user;
     } catch (err) {
          throw new Error(err);
     }
}
},
Mutation: {
     register: async(_:any): Promise<IUser> => {}
          login: async(_:any): Promise<IUser> => {}
}
}
```

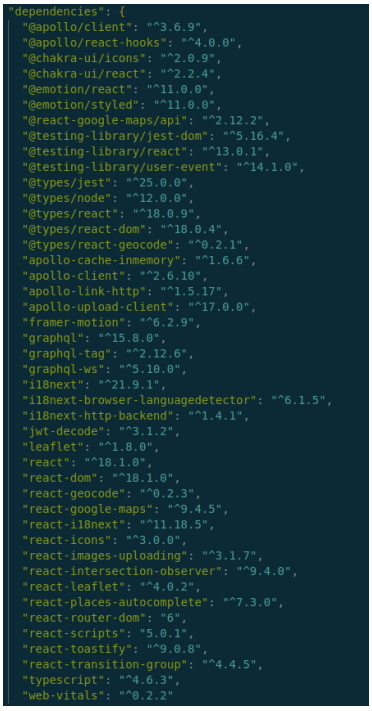
Pomoću GraphQL je osim mutacija i upita moguće napraviti i pretplate pomoću dodatne biblioteke graphql-subscription. Pomoću pretplata moguće je postaviti osluškivače koji ovisno o događajima izvršavaju funkcije. Pomoću te funkcionalnosti je na vrlo jednostavan način moguće implementirati razgovore unutar aplikacije. Sve što je potrebno je unutar mutacije pomoću koje se šalje poruka izvršiti događaj koji pretplata cijelo vrijeme osluškuje te prilikom dolaska signala na klijentsku stranu šalje podatke. Tu nastaje problem zato što se taj signal šalje na svaku poruku koja je poslana i također svaki klijent koji je aktivan ima pokrenute osluškivače što znači da će svi klijenti dobivati poruke koje se možda ne budu mogle ugraditi te će se dogoditi greška na klijentu. To se spriječava pomoću funkcije withFilter() iz biblioteke graphql-subscription koji omogućava da se filtriraju događaji te se na klijenta šalju podaci samo onim korisnicima koji sudjeluju u razgovoru.

```
message: async (_: any, input: any): Promise<IMessage>     => {
     const data = {
          content: input.content,
          author: input.author,
          room: input.room
     }

     const message = await Message.create(data);
```

```
      await message.save();
      const room = await Room.findOneAndUpdate({ id:   input.room },
{ $push: { messages: message.id    } });
      pubsub.publish(MESSAGE_SENT, { messageSent:     await
message.populate('author') });
      return message.populate('author');
}

      messageSent: {
subscribe: withFilter(
      () => pubsub.asyncIterator(MESSAGE_SENT),
      (payload: any, variables: any): any => {
            return payload.messageSent.room._id.equals(variables.room);
})}
```

4.3 Izrada web aplikacije

Web aplikacije je izrađena pomoću React-a koristeći TypeScript te još veći broj biblioteka koje omogućuju: lokalizaciju, izvršavanje GraphQL upita i mutacija, prikazivanje Google karata, navigiranje kroz aplikaciju te biblioteka za olakšani rad sa css-om.

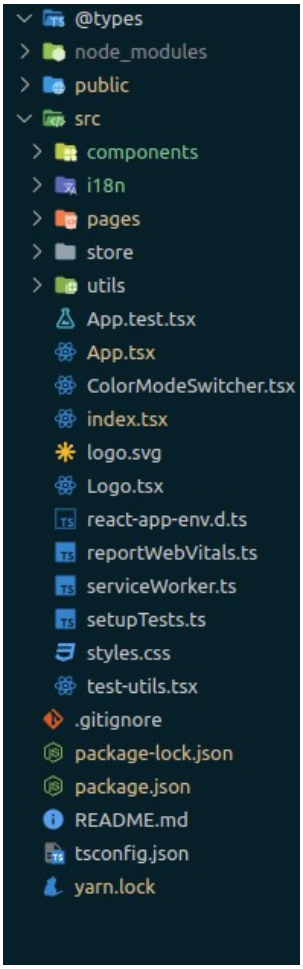


Slika 5: Lista biblioteka web aplikacije (izvor: vlastita izrada)

Da bi se kreirao kostur (engl. Boilerplate) za razvoj pomoću React-a i TypeScript-a potrebno je izvršiti naredbu:

```
npx create-react-app my-app --template=typescript
```

Ta naredba omogućuje da se sa udaljenog poslužitelja dohvate zadnje verzije potrebnih biblioteka za pokretanje React projekta te da se postavi početna struktura direktorija. Unutar React projekta ne postoji točno određena struktura direktorija koju je potrebno koristiti nego programer ima potpunu slobodu, a struktura direktorija izrađene aplikacije izgleda ovako:



Slika 6: Struktura direktorija web aplikacije (izvor: vlastita izrada)

Iz slike je moguće vidjeti direktorij src unutar kojega se nalaze sve što je potrebno za implementaciju sučelja , node_modules gdje su spremljene sve biblioteke koje se koriste u projektu, public unutar kojega je HTML dokument koji se zapravo prikazuje klijentima te se unutar njega nalazi element kojemu se onda dodjeljuje cijela aplikacija unutar index.ts dokumenta. Direktorij src podijeljen je na direktorije pages, store, utils, components te i18n. Unutar svakog direktorija se nalaze dijelovi aplikacije koji su funkcionalno povezani što znači da se na primjer unutar direktorija components nalaze sve komponente koje je moguće ponovno koristiti u bilo kojem dijelu aplikacije, unutar direktorija pages se nalaze stranice koje se sastoje od komponenti, utils direktorij sadržava

sve pomoćne dokumente poput transformacija što omogućuje bolju preglednost koda pošto su sve transformacije koje se događaju izdvojene u zasebne funkcije koje je također moguće koristiti u bilo kojem dijelu projekta.

Prva stvar koju je potrebno napraviti jest složiti navigiranje unutar aplikacije točnije koje će sve stranice postojati u aplikacije i na koji način se do njih dolazi. Da bi se to omogućilo potrebno je instalirati biblioteku react-router koja omogućuje kreiranje ruta, navigiranje po definiranim rutama te korištenje parametara. Unutar dokumenta App.tsx definiraju se rute sljedećom sintaksom:

```
export const App = () => {
     return (
          <ChakraProvider theme={theme}>
               <AuthContextProvider>
               <Routes >
          <Route path="/" element={<Navigate to="/dashboard/home"
          replace />} />
          <Route path="/login" element={<Login />} />
          <Route path="/register" element={<Register />} />
          <Route path="/register-company"
          element={<RegisterCompany />} />
          <Route path="/landing" element={<Landing />} />
          <Route path="/dashboard" element={
               <ProtectedRoute>
               <React.Suspense fallback={<Spinner></Spinner>}>
               <Dashboard />
               </React.Suspense>
               </ProtectedRoute>
          }>
          <Route index element={<React.Suspense
                    fallback={<Spinner> </Spinner>}><Home />
               </React.Suspense>} />
          <Route path="/dashboard/home" element={<React.Suspense
                    fallback={<Spinner> </Spinner>}><Home
               /></React.Suspense>} />
          <Route path="/dashboard/companies" element={<Companies
               />} />
          <Route path="/dashboard/jobs" element={<Jobs />} />
          <Route path="/dashboard/job/:id" element={<Job />} />
          <Route path="/dashboard/profile/:id" element={<Profile />} />
```

```
<Route path="/dashboard/usersPerJob"
      element={<JobsByUser />} />
</Route>
</Routes>
</AuthContextProvider>
<ToastContainer />
</ChakraProvider>
)}
```

Kao što je moguće vidjeti sa slike aplikacija ima deset različitih stranica s tim da na četiri stranice može pristupiti bilo tko dok na ostalih šest mogu pristupiti samo korisnici koji su napravili korisnički račun. Zbog bolje arhitekture upravljanja stanjima korišten je pristup gdje postoje ugnježdene rute te je tom arhitekturom jednostavnija implementacija cijele autorizacije, autentifikacije te dohvaćanja podataka. Jednostavnije je na način da se ništa niti ne dohvaća iz baze podataka sve dok korisnik ne uđe u glavni dio sučelja unutar kojega se zatim podaci učitavaju tek kada korisnik pristupi jednoj od stranica. Podaci koji se dohvaćaju na stranicama su samo podaci koje je na toj stranici potrebno prikazati te se nakon toga dalje prosljeđuju ostalim komponentama koje se nalaze na toj stranici kroz atribute.

Na slici je prikazana komponenta App.tsx unutar koje je oko aplikacije postavljen sloj za teme unutar aplikacije te sloj koji omogućava korištenje React Context-a. React Context omogućuje da se određenim podacima može pristupiti iz bilo kojeg dijela aplikacije te je u ovom slučaju korišten za autentifikaciju korisnika.

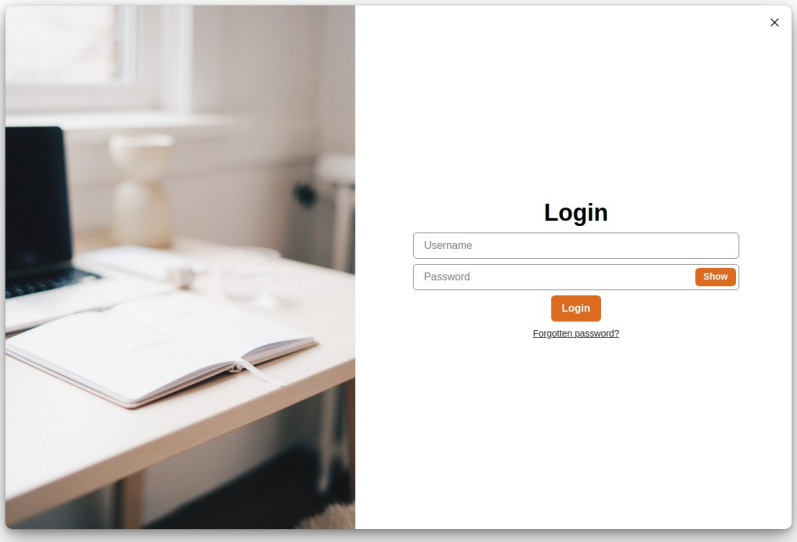
```
export const AuthContext: Context<any> = createContext({
     token: '',
     user: {},
     isLoggedIn: false,
     login: (token: string) => { },
     logout: () => { },
     register: () => { }
})

export const AuthContextProvider = (props: any): ReactElement => {
     let tok: string | null = localStorage.getItem('token');

     const [token, setToken] = useState<string | null>(tok ? tok : '')
     const [user, setUser] = useState<any>(tok ? jwt(tok) : {})
```

```
    const navigate = useNavigate();

    const userIsLogedIn = !!token;

    const login = (token: string, refreshToken: string) => {
        localStorage.setItem('token', token);
        localStorage.setItem('refreshToken', refreshToken)
        setToken(token);
        setUser(jwt(token));
        navigate("/dashboard");
    }

    const register = () => {
        navigate("/login");
    }

    const logout = () => {
        localStorage.removeItem('token')
        setToken("");
    }

    const authProvider = {
        token: token,
        user: user,
        isLoggedIn: userIsLogedIn,
        login: login,
        logout: logout,
        register: register
    }

    return <AuthContext.Provider  value={authProvider}>
            { props.children }
        </AuthContext.Provider>
    }

export const AuthConsumer = AuthContext.Consumer;
```

Sa slike je moguće vidjeti da se unutar implementacije React Context-a nalaze sve funkcije koje su potrebne za upravljanje autentifikacijom korisnika te je u svakom dijelu aplikacije moguće provjeriti postoji li korisnik ili ne te prema prethodno objašnjenoj arhitekturi za tog korisnika dohvatiti sve potrebne podatke.

4.4 Prikaz ekrana praktičnog dijela

Prva prikazana stranica jest stranica za prijavu na kojoj se nalaze samo dva polja za unos tekstualnih podataka i gumb koji izvršava mutaciju.



Slika 7: Prikaz ekrana za prijavu (izvor: vlastita izrada)

Za postavljanje varijabli korisničko ime i lozinka koristi se React udica useState koja omogućava ponovno iscrtavanje stranice svaki put kada se promjeni stanje. Također to stanje se prosljeđuje unutar mutacije te mutacija to koristi kao argumente. Za pozivanje mutacije koristi se funkcija handleLogin koja poziva mutaciju. Unutar mutacije je postavljena povratna funkcija (engl. Callback function) onCompleted koja poziva funkciju login iz autentifikacijskog konteksta te prikazuje notifikaciju o tome je li korisnik uspješno prijavljen.

```
const Login = () => {
    const { t } = useTranslation('common');
    const [user, setUser] = useState({ username: "", password: "" });
```

```
const [show, setShow] = useState(false);
const [loginUser, { loading }] = useMutation(LOGIN, {
variables: { username: user.username, password: user.password },
onCompleted: (res) => {
      const { token, refreshToken } = res.login;
      authContext.login(token, refreshToken);
      toast.success("Login Successful");
},
onError: (err) => {
      toast.error(err.message)
}
})
const authContext = React.useContext(AuthContext);

const handleLogin = () => {
      loginUser();
}
return (
<Center width='100%' bg='white' bgSize='cover' height='100vh'>
<Container boxShadow='dark-lg' p='0' bg='rgba(255,255,255, 0.6)'
maxW='1200px' h='800px' borderRadius='10px'>

<HStack height='100%' p='0px' w='100%' m='0px'>
<Container overflow='hidden' flex='0 0 50%' p='0'
borderLeftRadius='10px' h='800px' w='100%' m='0px'>
<Container
      overflow='hidden'
      p='0'
      m='0'
      position='relative'
      transform='scale(1,1)'
      transition='all 0.3s ease-out'
      _hover={{ transform: 'scale(1.1,1.1)', transition: 'all
      0.3s ease-in-out' }}
      width='100%'
      bgImage={loginImg}
      h='800px'
      bgSize='cover'
      bgPosition='center'>
```

```
</Container>
</Container>
<VStack placeContent='center' flex='1 0 50%' h='100%' p={20}
      position='relative' margin='auto 0'>
      <CloseButton position='absolute' top='10px' right='10px' />
      <Heading color='black'>{t('login')}</Heading>
      <Input
            borderColor='grey'
            focusBorderColor='black'
            placeholder='Username'
            color='black'
            _placeholder={{ color: 'grey' }}
            onChange={(e) => { setUser({ ...user, username:
            e.target.value }) }} />
      <InputGroup size='md'>
      <Input
            borderColor='gray'
            focusBorderColor='black'
            color='black'
            type={show ? 'text' : 'password'}
            _placeholder={{ color: 'grey' }}
            placeholder='Password'
            value={user.password}
            onChange={(e) => { setUser({ ...user, password:
            e.target.value }) }}
      />
      <InputRightElement width='4.5rem'>
      <Button colorScheme='orange' h='1.75rem' size='sm'
      onClick={() => setShow(!show)}>
            {show ? 'Hide' : 'Show'}
      </Button>
      </InputRightElement>
      </InputGroup>
      <Button colorScheme='orange' isLoading={loading}
      onClick={() => handleLogin()}>{t('login')}</Button>
      <Text
            _hover={{ cursor: "pointer", color: 'orange.500' }}
            onClick={() => { console.log("Change password") }}
            fontSize='sm'
```

```
                    textDecor='underline'>
                    Forgotten password?
              </Text>
        </VStack>
        </HStack>
        </Container>
        </Center>
)}
```

Na sljedećoj slici prikazana je sintaksa mutacije koja prihvaća korisničko ime i lozinku te ovisno o tome jesu li podaci ispravni vraća ili token, korisničko ime i id ili vraća pogrešku koja se zatim ispisuje kao notifikacija.

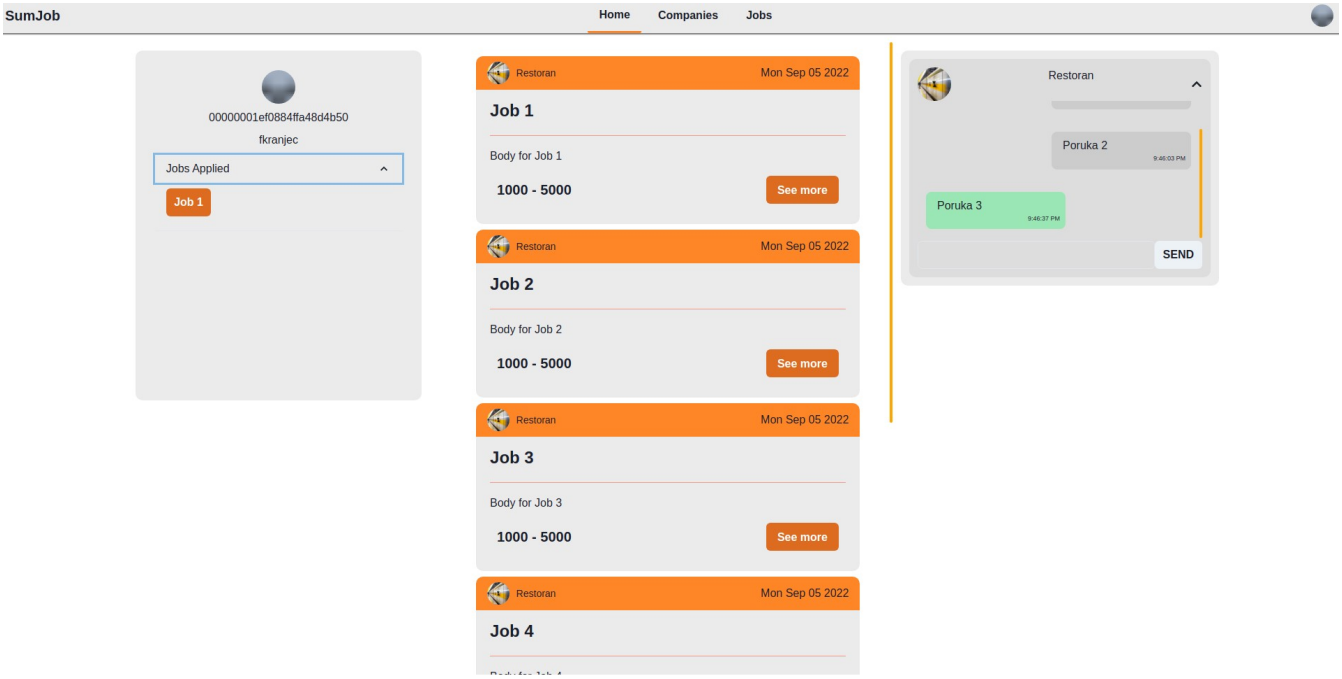
```
const LOGIN = gql`
     mutation login($username: String!, $password: String!){
           login(username: $username, password: $password) {
                 token
                 refreshToken
           }
     }
`
```

Za izvršavanje te mutacije potrebno ju je unutar web servisa implementirati. Unutar te mutacije prvo se rade provjere ispravnosti unesenih varijabli, zatim se traži u bazi postoji li korisnik sa tim korisničkim imenom te ako ne postoji klijentu se vraća pogreška. Ako korisnik postoji provjerava se je li korisnik potvrdi e-mail adresu te ako je onda se provjerava jednakost lozinke i na kraju se generira token sa osnovnim podacima od korisnika.

```
login: async (_: any, { username, password }: any, context: any):
Promise<IAuth> => {
     const { errors, valid } = validateLoginInput(username, password);
     const loginErrors: LoginErrors = errors as LoginErrors;
     if (!valid) {
```

```
            throw new UserInputError('Errors!', { loginErrors });
        }

        const user = await User.findOne({ username });
        if (!user) {
            loginErrors.general = 'User not found!';
            throw new UserInputError('User not found!',
            { loginErrors });
        }

        if (!user.isVerified) {
            loginErrors.verification = 'User not activated!';
            throw new UserInputError('User not activated!',
            { loginErrors });
        }

        const match = await bcrypt.compare(password, user.password);
        if (!match) {
            loginErrors.general = 'Wrong credentials!';
            throw new UserInputError('Wrong credentials!',
            { loginErrors });
        }

        const [token, refreshToken] = await createTokens(user);
        return {
            token,
            refreshToken
        };
    }
```

Sljedeća stranica na koju se dolazi nakon prijave jest početna stranica. Na njoj se nalazi navigacija unutar koje su četiri gumba od kojih se tri koriste za navigiranje i zadnji u obliku slike se koristi kao dodatni izbornik sa postavkama za promjenu teme, promjenu jezika, odjavu sa sustava te posjećivanje osobnog profila. Sadržaj stranice sastoji se od tri dijela. Lijevi i desni dio su statički i ne miču se dok se srednji dio pomiče. Na lijevom dijelu se nalazi kratki prikaz prijavljenog profila, u sredini su radna mjesta u obliku objava, a na desnoj strani se nalaze razgovori koji se mogu proširivati ovisno koji je potrebno koristiti.



Slika 8: Ekran početne stranice (izvor: vlastita izrada)

Za implementaciju te stranice izrađena je komponenta Layout koja omogućuje prethodni opis sadržaja u tri stupca. Svaka komponenta koja se prikazuje će se učitati do kraja i prikazati tek kada se podaci dohvate.

```
const Home: FC = () => {
    let limit = 3;
    let offset = 0;
    let changed: boolean = false;
    const [jobs, setJobs] = useState([]);

    const { loading, data, fetchMore, error } =
    useQuery<JobsResponse>(GET_JOBS, {
        variables: { offset: offset, limit: limit }, fetchPolicy:
        'no-cache',
        onCompleted(res) {
            setJobs(res.getJobs.jobs)
        },
    });

    const authContext = useOutletContext<IProfileShort>();
    return (
```

```
<Layout>
      <Layout.Left>
            <Suspense fallback={<Spinner></Spinner>}>
                  <ProfileCard id={authContext?.id}
                        userType={authContext?.userType}
                        username={authContext?.username}
                        image={authContext?.image} />
            </Suspense>
      </Layout.Left>

      <Layout.Mid>
            <VStack flexDirection='column'>
            {!loading && jobs?.length !== 0 &&
            jobs?.map((job: any) => (
                  <JobCard
                        title={job.name}
                        id={job.id}
                        key={job.id}
                        content={{ title: job?.content?.title,
                        body: job.content?.body, footer:
                        job.content?.footer }}
                        salary={{ from: job.averageSalary?.from,
                        to: job.averageSalary?.to }}
                        user={{ ...job.company }}
                        createdAt={job.timeCreated}
                  />
            ))
            }
            {!loading && jobs?.length !== 0 && (
                  <InView
                  onChange={async (inView) => {
                  if (limit >= data?.getJobs.totalCount) {
                  limit = data?.getJobs.totalCount
                  } else {
                  limit = data?.getJobs.totalCount
                  }
                  if (inView) {
                  const { data, error } = await fetchMore({
                        variables: {
```

```
                        offset: offset,
                        limit: limit
                        }
                  })
                  if (jobs.length <= data.getJobs.totalCount) {
                        setJobs([...data?.getJobs.jobs])
                  }

                  }}}
                  ></InView>
            )}
      </VStack>
      </Layout.Mid>

      <Layout.Right>
            <Suspense fallback={<Spinner></Spinner>}>
                  <Rooms id={authContext?.id} />
            </Suspense>
      </Layout.Right>
</Layout >
)}
```

Upit koji se izvršava na početnoj stranici jest dohvaćanje svih poslova, ali ima implementirano straničenje tako da se podaci učitavaju kako korisnik prolazi kroz podatke, a ne unaprijed. Upitu je potrebno proslijediti offset i limit te vraća id posla, naziv, sadržaj te poduzeće koje je kreiralo radno mjesto.

```
const GET_JOBS = gql`
      query GetJobs($offset:Int, $limit: Int) {
            getJobs(offset:$offset, limit: $limit) {
                  totalCount
                  jobs{
                        id
                        name
                        content{
                              title
                              body
                              footer
                        }
```

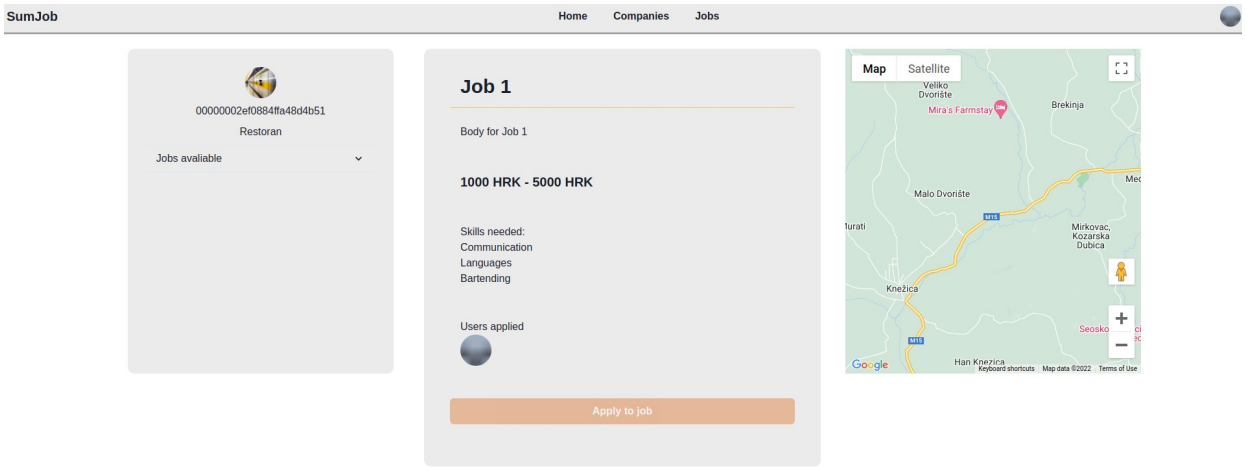
```
                averageSalary{
                        from
                        to
                }
                timeCreated
                company{
                        id
                        username
                        image
                }
        }
    }
}`;
```

Straničenje je implementirano vrlo jednostavno pošto mongoose modeli unutar sebe sadrže metode skip koja predstavlja offset i funkciju limit. Također se koristi metoda populate koja omogućava popunjavanje podataka za reference. Reference su polja u shemi koja su povezana sa drugim dokumentom.

```
getJobs: async (_: any, { offset, limit }: any, context: any):
Promise<GetJobsOutput> => {
    try {
        const user = await auth(context);
        const totalCount = await Job.count();
        const jobs = await
        Job.find().populate('company').skip( offset ).limit(limit);
        const pagination: GetJobsOutput = {
            jobs: jobs,
            totalCount: totalCount
        };
        return pagination;
    } catch (err) {
        throw new Error(err);
    }
}
```

Stranica detaljnog opisa posla izgledom je slična početnoj stranici, a to je tako zbog komponente Layout koja se i tu koristi samo što ovoga puta se na lijevoj strani prikazuje skraćeni profil poduzeća koje je kreiralo radno mjesto, u sredini se nalazi detaljni opis posla

gdje je moguće vidjeti potrebne vještine, potencijalnu plaću te korisnike koji su se prijavili na posao. Također moguće se i prijaviti na posao ako korisnik već nije prijavljen te se u tom slučaju onemogućava gumb. Sa desne strane se nalazi Google karta na kojoj je prikazana lokacija na kojoj se obavlja posao.



Slika 9: Ekran detaljni opis posla (izvor: vlastita izrada)

Na stranici detaljnog opisa se prilikom pokretanja stranice automatski izvršava upit te se na klik gumba izvršava mutacija. Upitu je potrebno proslijediti id posla koji se dobije iz parametara trenutne rute pošto se svaki posao prikazuje na svojoj ruti te taj upit vraća sve podatke o poslu koji postoje u bazi podataka za zadani model. Mutaciji je potrebno proslijediti id posla i id korisnika koji je prijavljen te se zatim izvršava ažuriranje posla i dodaje se novi prijavljeni korisnik samo ako isti već nije dodan. Također ta mutacija sadržava događaj koji šalje signal pretplati koja je zadužena za notifikacije tako da poduzeća ako su prijavljena u sustav u stvarnom vremenu mogu vidjeti kada se netko prijavi za posao te odmah poslati poruku ili kontaktirati korisnika.

```
const GET_JOB = gql`
    query getJob($jobId: ID!){
        getJob(jobId: $jobId) {
            id
            name
            skills
            averageSalary{
                from
```

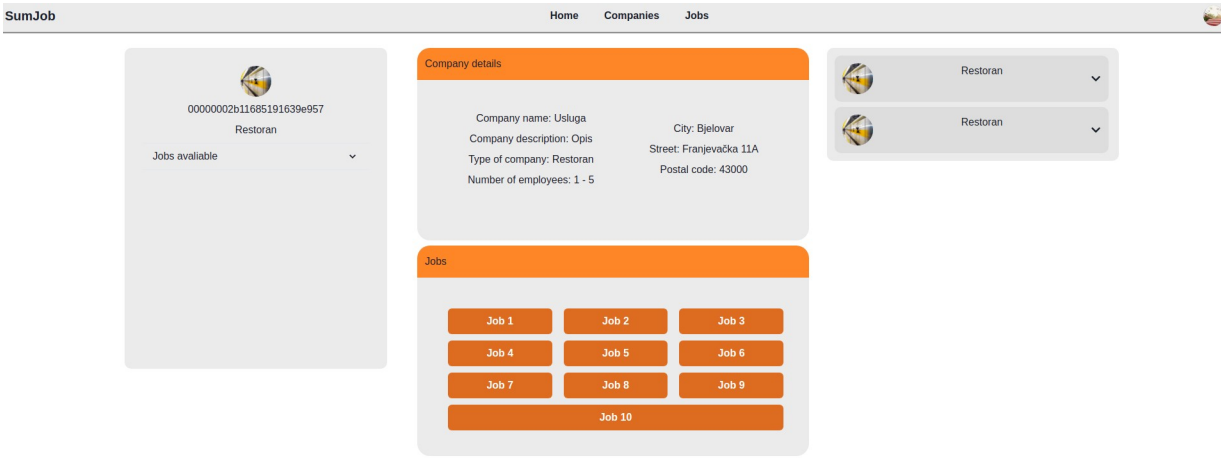
```
      to
}
applied{
      id
      image
}
content{
      title
      body
      footer
}
timeCreated
period {
      from
      to
}
company {
      id
      username
      image
      address {
            city
            postalCode
            streetNumber
            street
            state
            latlng{
                  lat
                  lng
            }
      }
      companyInfo {
            companyName
            description
            numberOfEmployees
            typeOfCompany
      }
      userType
}
```

```
        }
    }`

const APPLY_TO_JOB = gql`
    mutation ApplyToJob($jobId: ID!, $userId: ID!) {
        applyToJob(jobId: $jobId, userId: $userId) {
            jobId
            userId
        }
    }`;
```

Pregled svih poduzeća koja postoje unutar sustava moguće je pregledati na putanji poduzeća te je ista moguće filtrirati prema određenim svojstima. Unutar svake kartice poduzeća prikazani su osnovni podaci poput naziva, slike, broja zaposlenika, vrste poduzeća te opisa. Također je moguće pritisnuti gumb “Idi na profil” te vidjeti detaljnije podatke o željenom poduzeću.

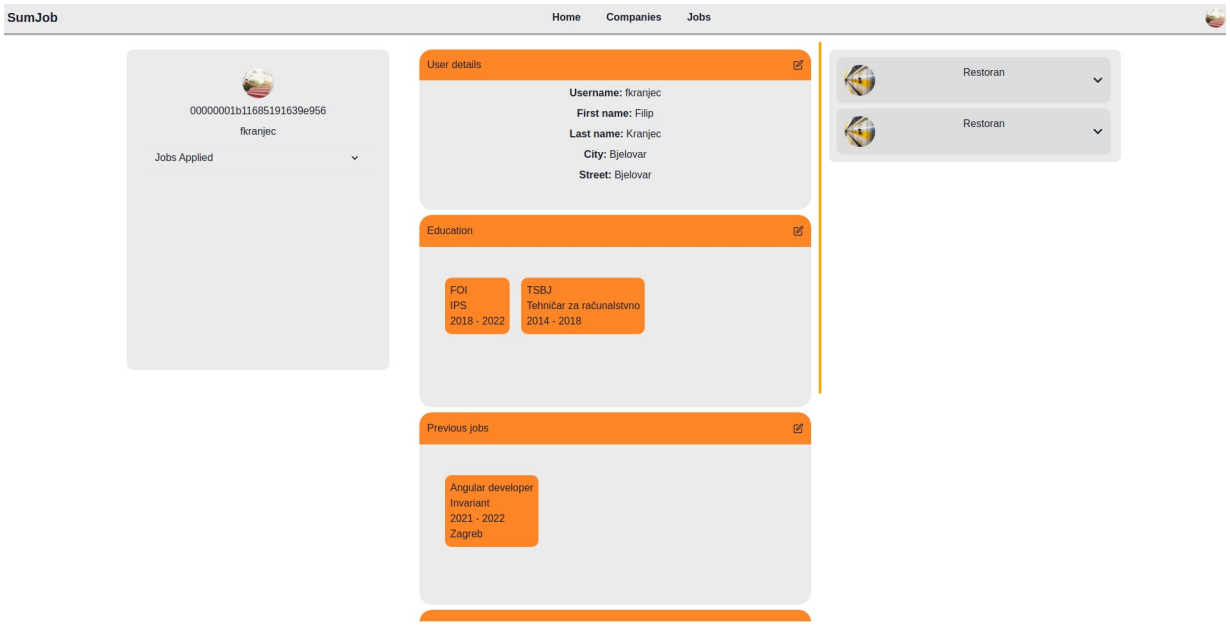
Slika 10: Ekran pretraživanja svih poduzeća (izvor: vlastita izrada)

Detaljni opis poduzeća moguće je vidjeti na stranici profila unutar koje se prema primljenom tipu korisnika odlučuje hoće li se prikazati prikaz za korisnika ili za poduzeće. Stranica se sastoji od detalja poduzeća te trenutno objavljenih poslova koje je moguće posjetiti te pregledati.



Slika 11: Ekran detaljno prikaza poduzeća (izvor: vlastita izrada)

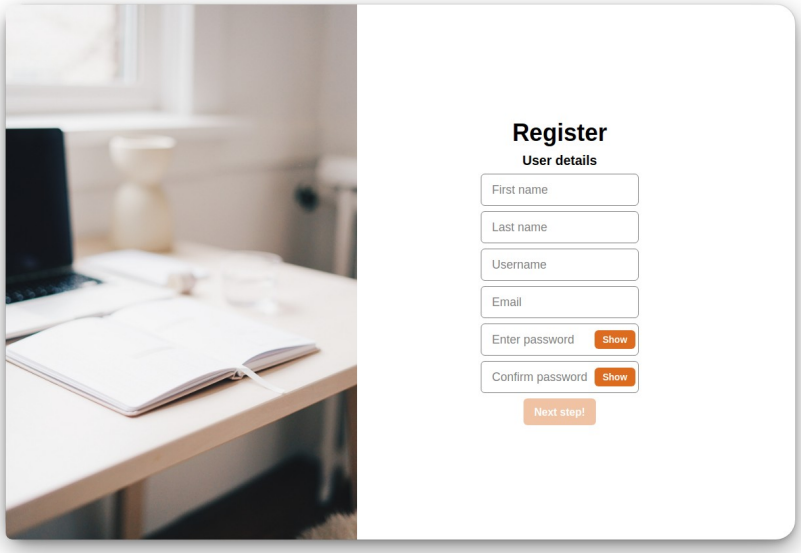
Drugi oblik prikaza već spomenute stranice profila jest prikaz korisnika koji se sastoji od svih detalja i informacija koje je korisnik naveo te je također moguće iste i ažurirati samo ako je prijavljeni korisnik na stranici svoga profila. Na ovoj stranici je moguće vidjeti podatke poput prijašnjih edukacija korisnika, prijašnjih zaposlenja, vještina te jezika.



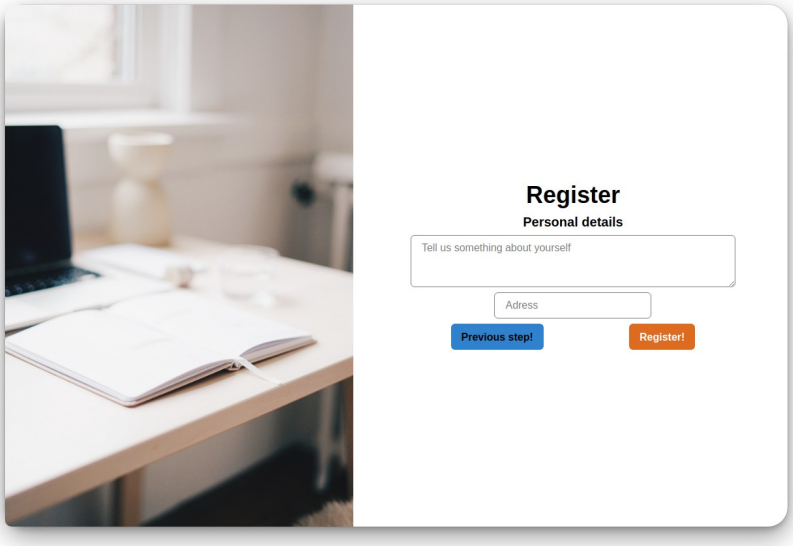
Slika 12: Ekran detaljnog prikaza korisnika (izvor: vlastita izrada)

Registracija se sastoji od dva koraka koja su podijeljena tako da korisnik popunjava

prvo podatke za prijavu, osobne podatke i lozinku, a nakon toga dodatne opcionalne podatke kao što je adresa, vještine, jezici ili prethodna zaposlenja. Prva forma je obavezna za ispunjavanje dok je drugi dio opcionalan te ga je moguće ispuniti i nakon registracije. Nakon prođenih koraka korisnik pritišće gumb za registraciju te ga aplikacija navigira na ekran za prijavu.



Slika 13: Ekran za registracija (izvor: vlastita izrada)



Slika 14: Ekran za registracija drugi korak (izvor: vlastita izrada)

Za implementiranje ekrana registracije korištena su tri dokumenta. Dokument za prvu formu, dokument za drugu formu te dokument unutar kojega se odvija glavno upravljanje stanjima i poslovna logika komponente. Prva i druga forma predstavljaju komponente koje

se koriste na način da im se unutar atributa proslijede funkcije koje se na određenu akciju pozivaju. Na taj način je odvojena poslovna logika komponente od prikaza podataka.

```
const Register: React.FunctionComponent<Register> = () => {
    const authContext = React.useContext(AuthContext);
    const [register, setRegister] = useState<Register>({
        step: 1,
        username: '',
        email: '',
        password: '',
        confirmPassword: '',
        firstName: '',
        lastName: '',
        image: '',
        languages: []
    });
    const [registerUser, { loading }] = useMutation(REGISTER, {
        update(_, result) {
            authContext.register();
            toast.success("Registration Successful");
        }
        variables: { registerInput: { username: register.username,
        password: register.password, confirmPassword:
        register.confirmPassword, email: register.email, userInfo: {
        firstName: register.firstName, lastName:
        register.lastName } } }
    });
    const handleChange = (input: string) => (e: any) => {
        setRegister({ ...register, [input]: e.target.value });
    }
    const nextStep = () => {
        setRegister({ ...register, step: register.step + 1 });
    }
    const previousStep = () => {
        setRegister({ ...register, step: register.step - 1 });
    }
    const handleRegister = () => {
        registerUser();
```

```
        }
        switch (register.step) {
              case 1: return (
              <>
                    <UserDetails handleChange={handleChange}
                    nextStep={nextStep}
                    values={register2UserDetails(register)} />
              </>
              )
              case 2: return (
              <>
                    <PersonalDetails handleChange={handleChange}
                    nextStep={nextStep} prevStep={previousStep}
                    register={handleRegister}
                    values={register2PersonalDetails(register)} />
              </>
              )
              default: return (<PersonalDetails
                    handleChange={handleChange} nextStep={nextStep}
                    prevStep={previousStep} register={handleRegister}
                    values={register2PersonalDetails(register)} />)
        }
  }
```

Mutacija za registraciju korisnika kao argument prima objekt tipa RegisterInput koji je prikazan u GraphQL shemi te se sastoji od svih podataka koje je moguće upisati za korisnika, ali nisu svi obavezni za unesti. Kao i kod prijave mutacija vraća token, korisničko ime te id korisnika ako je korisnik uspješno registriran, a ako nije onda vraća pogrešku koja se prikazuje kao notifikacija.

```
const REGISTER = gql`
     mutation register($registerInput:RegisterInput!){
           register(registerInput:$registerInput) {
                 token
                 username
                 id
           }
     } `;
```

5 Zaključak

U svrhu razvoja web aplikacije izrađena je aplikacija za klijente koristeći React web okvir te web servis koristeću Express web okvir koji ima GraphQL arhitekturu te dohvaća podatke iz MongoDB baze podataka. Prednost GraphQL arhitekture web servisa je u tome što klijentska aplikacija ima mogućnost izvršavati puno više upita i mutacija zbog brzine izvršavanja CRUD operacija nad tim podacima. Također velika prednost ove kombinacije web okvira jest u okruženju koje pružaju programerima i vremenu koje je potrebno da se aplikacija do kraja izgradi.

Prilikom implementacije arhitekture cijele web aplikacije posebna pažnja posvećena je upravljanju stanjima i provođenju stanja kroz aplikaciju iz razloga što brzina izvođenja aplikacije i pojava grešaka ovisi o tome kada će koja komponenta dobiti stanja ili kada će se koja komponenta učitati i prikazati.

Najveći dio vremena kod izgradnje aplikacije oduzelo je kreiranje upita i mutacije te implementiranje istih iz razloga što je potrebno dobro promisliti na koji način nešto spremiti u bazu ili na koji način nešto modificirati. Tako je i došlo do implementacije ažuriranja poslova i korisnika gdje se sa jednom mutacijom može ažurirati bilo koje polje koje postoji u modelu. Ta implementacija je dosta posebna zato što se ulaznom objektu brišu svi ključevi koji nemaju vrijednost dok taj objekt može imati neodređeni broj ključeva.

Trenutno je JavaScript najtraženiji te najkorišteniji programski jezik zato što je u današnje vrijeme pomoću njega moguće napraviti web aplikacije, mobilne aplikacije, web servise i stolne aplikacije, ali TypeScript je nasljednik JavaScipt-a koji ima mogućnosti raditi sa tipovima podataka, enumeracijama, sučeljima te se na taj način poboljšava kod i programeri mogu biti sigurniji da sve bude radilo kako treba.

Popis literature

[1] What is web application? [mrežno]. Dostupno: https://www.stackpath.com/edge-academy/what-is-a-web-application/ [Pokušaj pristupa 10. rujan 2022.]

[2] Ben Farrell (2018.), Web Components in Action

[3] What are web frameworks and why do you need them? [mrežno]. Dostupno: https://www.intelegain.com/what-are-web-frameworks-and-why-you-need-them/ [Pokušaj pristupa 10. rujan 2022.]

[4] Guillaume Breux, Client-side vs. Server-side vs. Pre-rendering for Web Apps [mrežno]. Dostupno: https://www.toptal.com/front-end/client-side-vs-server-side-pre-rendering [Pokušaj pristupa 10. rujan 2022.]

[5] Node [mrežno]. Dostupno: https://nodejs.org/en/docs/ [Pokušaj pristupa 17. srpanj 2022.]

[6] The History of React.js on a Timeline. [mrežno]. Dostupno: https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/ [Pokušaj pristupa 10. srpanj 2022.]

[7] React. [mrežno]. Dostupno: https://reactjs.org/ [Pokušaj pristupa 11. srpanj 2022.]

[8] React Router [mrežno]. Dostupno: https://reactrouter.com/en/main. [Pokušaj pristupa 27. kolovoz 2022.]

[9] InterviewBit, Angular Vs React: Difference Between Angular and React. [Mrežno] Dostupno: https://www.interviewbit.com/blog/angular-vs-react/#:~:text=React%20is%20a%20library%2C%20but,React%20works%20a%20bit%20faster). [Pokušaj pristupa 14. srpanj 2022.]

[10] HTML elements reference [mrežno]. Dostupno: https://developer.mozilla.org/en-US/docs/Web/HTML/Element [Pokušaj pristupa 10. rujan 2022.]

[11] CSS: Cascading Style Sheets. [mrežno]. Dostupno: https://developer.mozilla.org/en-US/docs/Web/CSS [Pokušaj pristupa 10. rujan 2022.]

[12] TypeScript. [mrežno]. Dostupno: https://www.typescriptlang.org/docs/ [Pokušaj pristupa 3. rujan 2022.]

[13] MongoDB [mrežno]. Dostupno: https://www.mongodb.com/docs/manual/ [Pokušaj pristupa 23. kolovoz 2022.]

[14] Express [mrežno]. Dostupno: https://expressjs.com/ [Pokušaj pristupa 15. srpanj 2022.]

[15] GraphQL. [mrežno]. Dostupno: https://graphql.org/ [Pokušaj pristupa 12. srpanj 2022.]

[16] Mongoose. [mrežno]. Dostupno: https://mongoosejs.com/. [Pokušaj pristupa 27. kolovoz 2022.]

Popis slika

Popis tablica