Studentski portal za oglašavanje ponuda upotrebom Spring razvojnog okvira

Cecelja, Marko

Master's thesis / Diplomski rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Faculty of Electrical Engineering, Computer Science and Information Technology Osijek / Sveučilište Josipa Jurja Strossmayera u Osijeku, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:200:361554

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

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

Repository / Repozitorij:

Faculty of Electrical Engineering, Computer Science and Information Technology Osijek

zir.nsk.hr

Image not found or type unknown

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

STUDENTSKI PORTAL ZA OGLAŠAVANJE PONUDA UPOTREBOM SPRING RAZVOJNOG OKVIRA

Diplomski rad

Marko Cecelja

Osijek, 2022.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za diplomski ispit

Osijek, 28.06.2022.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za diplomski ispit

Ime i prezime Pristupnika:	Marko Cecelja
Studij, smjer:	Diplomski sveučilišni studij Računarstvo
Mat. br. Pristupnika, godina upisa:	D-1109R, 13.10.2020.
OIB studenta:	61140051417
Mentor:	Izv. prof. dr. sc. Ivica Lukić
Sumentor:	Izv. prof. dr. sc. Mirko Köhler
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	Izv.prof.dr.sc. Zdravko Krpić
Član Povjerenstva 1:	Izv. prof. dr. sc. Mirko Köhler
Član Povjerenstva 2:	Miljenko Švarcmajer, mag. ing. comp.
Naslov diplomskog rada:	Studentski portal za oglašavanje ponuda upotrebom Spring razvojnog okvira
Znanstvena grana diplomskog rada:	**Informacijski sustavi (zn. polje računarstvo)**
Zadatak diplomskog rada:	Aplikacija treba nuditi različite kategorije kao što su edukacija, kultura, hrana itd., unutar kojih razni oglašivači mogu kreirati ponude, događaje, promotivne kodove koje bi studenti koristili za ostvarivanje popusta, pronalaženje željene usluge, aktualnih događaja i slično. Oglašivači bi svoje ponude kreirali putem web platforme dok bi ih studenti mogli pretraživati uz pomoć android aplikacije. Studenti bi imali mogućnost kontaktirati oglašivača u vezi dodatnih pitanja te komunikaciju obavljati kroz aplikaciju. Rezervirano za: Marko Cecelja
Prijedlog ocjene pismenog dijela ispita (diplomskog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 2 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene od strane mentora:	28.06.2022.

Potvrda mentora o predaji konačne verzije rada:	*Mentor elektronički potpisao predaju konačne verzije.*
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

IZJAVA O ORIGINALNOSTI RADA

Osijek, 20.07.2022.

Ime i prezime studenta:	Marko Cecelja
Studij:	Diplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	D-1109R, 13.10.2020.
Turnitin podudaranje [%]:	5

Ovom izjavom izjavljujem da je rad pod nazivom**: Studentski portal za oglašavanje ponuda upotrebom Spring razvojnog okvira**

izrađen pod vodstvom mentora Izv. prof. dr. sc. Ivica Lukić

i sumentora Izv. prof. dr. sc. Mirko Köhler

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD *1*

1.1. Zadatak rada **1**

2. PREGLED PODRUČJA TEME *2*

3. PREGLED KORIŠENIH TEHNOLOGIJA *4*

3.1. Tehnologije na strani poslužitelja **4**

3.1.1. Java programski jezik 4

3.1.2. Spring razvojni okvir 7

3.2. Tehnologije na strani klijenta **8**

3.2.1. JavaScript 8

3.2.2. React 9

3.2.3. Bootstrap5 i Sass 10

3.3. Android tehnologije **12**

3.3.1. Kotlin programski jezik 12

3.3.2. MVVM obrazac i LiveData 14

3.4. WebSocket **15**

4. WEB APLIKACIJA *18*

4.1. Realizacija poslužitelja **18**

4.2. Realizacija klijenta **24**

4.3. Korištenje aplikacije **29**

5. ANDROID APLIKACIJA *35*

5.1. Fragmenti **38**

5.2. Recycler View **40**

5.3. Retrofit 2 ..... **42**

6. ZAKLJUČAK *44*

LITERATURA *45*

SAŽETAK *46*

ABSTRACT *47*

ŽIVOTOPIS..... 48

1. UVOD

Ideja za izradu ovoga rada nastala je iz razloga što sve veći broj organizacija nudi studentima razne ponude, organiziraju posebne događaja i pružaju svoje usluge. Međutim, zbog velikog broja informacija, često te ponude ne dospiju do studenata ili dospiju prekasno. Svrha ovog portala je pružiti mjesto gdje će studenti moći pronaći sve što bi ih moglo zanimati na jednome mjestu, a organizacijama omogućiti jedinstveni sustav oglašavanja. Prilikom registracije na web aplikaciju, pripadnici organizacije se mogu uključiti u nju unosom pristupnog kôda kojeg dobiju od vlasnika te organizacije. Postoji i opcija stvaranja nove organizacije, ukoliko organizacija ne postoji, koju zatim administrator sustava mora odobriti. Nakon što se priključe željenoj organizaciji, omogućeno im je stvaranje objava u različitim kategorijama u ime organizacije koju predstavljaju te njihova deaktivacija ukoliko više nisu važeće. Studentima je omogućeno, preko Android aplikacije, pregled svih aktivnih objava te kontaktiranje oglašivača ukoliko imaju dodatnih pitanja u vezi neke objave. Komunikacije između studenta i oglašivača vrši se direktno kroz aplikaciju.

Drugo poglavlje bavi se analizom postojećih rješenja i područja djelovanja sličnih aplikacija. Treće poglavlje daje pregled korištenih tehnologija za izradu web i android aplikacije. Pružen je kratki opis svake tehnologije i objašnjeni su osnovni koncepti. Četvrto poglavlje prikazuje funkcionalnost i primjenu web aplikacije, dok peto poglavlje prikazuje korištenje android aplikacije.

1.1. Zadatak rada

Izraditi web aplikaciju koja će omogućiti pripadnicima različitih organizacija kreiranja objava u ime svoje organizacije. Izraditi android aplikaciju kroz koju će studenti moći pregledati pojedine objave. Omogućiti *WebSocket* komunikaciju između organizacije i studenta.

2. PREGLED PODRUČJA TEME

Jedno od sličnih rješenja koje nudi studentima pregled aktualnih oglasa predstavlja Kolegio.hr web stranica. Njeno korištenje je potpuno besplatno te dostupno svim redovnim i izvanrednim studentima preddiplomskih, diplomskih i poslijediplomskih studija na svim javnim i privatnim fakultetima u Hrvatskoj [1]. Nakon registracije na ovu stranicu, potrebno je potvrditi svoj studentski status putem sveučilišne e-mail adrese i AAI sustava. Ukoliko studenti poslijediplomskih studije ne posjeduju AAI identitet, isti moraju zatražiti od fakulteta kako bi mogli koristiti usluge platforme. Studentima je omogućen pregled raznih objava prilikom kojih je moguće prikazati kôd, iskoristiv prilikom Internet kupnje. Također, moguće je određene oglase svrstati među favorite kako bi ih se lakše moglo pronaći u budućnosti. Ova stranica nudi i mogućnost pisanja članaka koje studenti mogu čitati. Članci se također mogu podijeliti i na društvenim mrežama kao što su Facebook, Instagram, Twitter, LinkedIn i Pinterest. Kolegio.hr nudi Android i iOS aplikaciju kako bi pronalaženje promotivnih kôdova bilo još jednostavnije. Na web stranici dostupna je poveznica do ovih aplikacija kako bi ih studenti mogli instalirati na svoje mobilne uređaje. Također, navode kako je popuste moguće koristiti i prilikom kupovine u poslovnicama prilikom čega je potrebno preuzeti kupon s web stranice te ga spremiti u mobilnu aplikaciju nakon čega ga je moguće njegovo predočenje na prodajnom mjestu. Nakon pronalaženja željenog kupona, studenti mogu otvoriti detaljni prikaz gdje se nalazi opis te uvjeti ponude. Stranica također implementira višejezičnost na nekoliko različitih jezika među kojima su Češki, Slovenski i Slovački. Mobilna aplikacija studentima služi i kao identifikator unutar kojeg je moguće postaviti profilnu sliku te dokazati status studenta. Aplikacija je u potpunosti sinkronizirana s web stranicom što znači da se kuponi iz web stranice mogu spremiti u mobilnu aplikaciju. Također, dostupni su filteri za kupone iskoristive putem Internet kupovine te na kupone iskoristive isključivo u poslovnica. Platforma kuponima rukuje u dvije kategorije. Kuponi za jednokratnu upotrebu poništava se nakon kupovine, dok se kuponi za višekratnu upotrebu ne poništavaju što znači da se popust može ostvariti korištenjem istog kupona. Studentima su dostupne informacije o njihovom statusu uz naznaku datuma isteka potvrde. Također, moguće je ispuniti profil s dodatnim informacijama kao što je adresa, kontakt podatak, datum rođenja i slično. Kolegio.hr studentima nudi mogućnost promjene lozinke te korisničku podršku. Korisnička podrška služi kako bi studenti mogli poslati upit ukoliko imaju problema s registracijom, potvrđivanjem svojeg statusa ili ukoliko imaju bilo kakva pitanja u vezi korištenja. Podrška se ostvaruje kroz obrazac koji se ispunjava na platformi nakon čega se odgovorna osoba studentima javlja putem e-mail poruke. Ova platforma posjeduje Facebook i Instagram profil do kojeg je

moguće doći putem web stranice te tamo poslati neki upit vezan uz korištenje samo platforme. Postoje i razna druga rješenja koja objedinjuju ponude i kupone poput eKuponi.com, međutim ova rješenja su dostupna svima, a ne samo studentima. Navedena platforma, kupone svrstava prema trgovini unutar koje ih je moguće koristiti te također nudi pretraživanje trgovine po njezinom nazivu. Ova stranica prikazuje poveznicu za studentske popusti koja zatim korisnika odvodi na Kolegio.hr platformu.

Glavna razlika između dostupnih rješenja i onog koji se razmatra kroz ovaj rad je u mogućnosti organizacije da sama stvara svoje ponude studentima te kompletno upravljanje njima. To znači da organizacije imaju direktan utjecaj na koji način će se studentima predstaviti. Također, trenutno dostupna rješenja ne nude komunikaciju organizacije i studenta putem same aplikacije već su potrebni dodatni koraci kao što je komunikacije e-mail putem. Razvijene aplikacija, osim popusta i ponuda, nudi kreiranje i raznih drugih događaja koji su organizirani isključivo za studente poput studentskih zabava, sportskih događanja i slično. Aplikacije nudi svrstavanje ovih događaja i ponuda kroz razne kategorije kako bi bile lakše i brže pronađene što postojeća rješenja ne nude. Također, organizacijama je dostupno dodavanje drugih članova kako bi kreiranje obavijesti bilo brže i efikasnije.

3. PREGLED KORIŠENIH TEHNOLOGIJA

3.1. Tehnologije na strani poslužitelja

Poslužitelj se može definirati kao računalo ili programska podrška koja pruža neku uslugu većem broju klijenata. Od klijenta prima razne informacije, obrađuje ih, pohranjuje i na kraju vraća odgovor na zatraženu uslugu. Poslužitelj služi kao posrednik između klijenta i baze podataka, a glavna svrha mu je spriječiti klijenta u direktnom upravljanju podacima kako ne bi mogao narušiti njihov integritet. Postoje razne tehnologije s kojima se može razvijati poslužiteljski dio aplikacije. Za potrebu ovoga rada korišteni su Java programski jezik uz *Spring* razvojni okvir, koji će detaljnije biti objašnjeni u nastavku.

3.1.1. Java programski jezik

Java je objektno orijentirani programski jezik visoke razine. Dizajniran je na način da se uvodi jasna razlika između grešaka pri prevođenju programskog kôda u strojni jezik, koje mogu i moraju biti otkrivene, te između grešaka nastalih za vrijeme izvođenja programa [2]. Programski kôd strukturira se od klasa iz kojih se zatim kreiraju objekti. Klasa predstavlja opis objekta, dok je objekt konkretna instance klase. Iz jedne klase se može kreirati više objekata.

Definicija klase provodi se na način da se upotrijebi ključna riječ *class* nakon koje slijedi naziv klase. Dobro definirana klasa ima naziv u obliku imenice. Java klasa, kao i klase drugih objektno orijentiranih jezika, može se sastojati od atributa, konstruktora i metoda.

Atributi predstavljaju svojstva objekata, te se pomoću njih objekt opisuje. Svakome atributu potrebno je dodijeliti pravo pristupa koji može biti privatan, javan ili zaštićen. Privatni atributi vidljivi su samo unutar klase što znači da njima mogu pristupati i upravljati samo metode koje su definirane unutar iste klase. Javnim atributima može pristupati bilo koji dio programa i mijenjati im vrijednost. To nije dobra praksa stoga se treba izbjegavati dodjeljivanje ovoga tipa pristupa. Zaštićeni atributi vidljivi su unutar klase u kojoj se definirani i u svim njenim izvedenicama što znači da će klasa koja naslijedi drugu klasu, moći pristupati i njenim zaštićenim atributima. Osim ova tri, Java uvodi u četvrto pravo pristupa koje se naziva paketno-privatnim, a označava da atributu mogu pristupiti sve klase koje se nalaze u istom paketu kao i polazna klasa. Ovo je ujedno i zadano pravo pristupa ukoliko se ne odredi drugačije.

Konstruktori predstavljaju poseban oblik metoda koji se koristi za stvaranje novih instanci objekta. Ima isti naziv kao klasa i nema povratni tip. Postoje više vrsta konstruktora koje je moguće definirati. Prazni konstruktor služi za stvaranje objekta koji nema definirane atribute. Konstruktor

s parametrima može stvoriti objekt sa svim atributima klase ili samo s određenima. Konstruktor kopije služi za kopiranje jednog objekta u drugi objekt. Za razliku od C++, Java programski jezik nema potrebu za destruktorima, koji služe za uništavanje objekta, budući da uvodi takozvani skupljač smeća (engl. *Garbage Collector*) koji se brine o uništavanju objekata i oslobađanju memorije kada se navedeni objekti više ne koriste. Ovaj pristup predstavlja veliki iskorak budući da programeri ne moraju više ručno voditi računa o zauzimanju i oslobađanju memorije.

Metode služe za obavljanje određenih akcija. To može biti dodjeljivanje vrijednosti atributima, dohvat atributa ili neke druge radnje. Dobro definirana metoda imenovana je glagolom i odgovorna je isključivo za jednu radnju koju obavlja. Metode, kao i atributi, mogu imati različita prava pristupa. Postoji više vrsta metoda:

- Bez parametara i s povratnim tipom
- Bez parametara i bez povratnog tipa
- S parametrima i s povratnim tipom
- S parametrima i bez povratnog tipa.

```
1. public class Example {
2.
3.  private String attribute;
4.
5.  public Example(String attribute) {
6.   this.attribute = attribute;
7.  }
8.
9.  public String getAttribute() {
10.    return attribute;
11.   }
12.  }
```

Primjer 3.1. Java klasa s atributom, konstruktorom i metodom

Java, kao i ostali objektno orijentirani jezici, zasniva se na tri stupa objektno orijentirane paradigme, a to su enkapsulacija, nasljeđivanje i polimorfizam.

Enkapsulacija predstavlja posebni obrazac razvoja koji nalaže da svi atributi klase moraju biti privatnog prava pristupa, a prema van se izlažu metode koje upravljaju tim atributima, dohvaćaju ih ili im postavljaju vrijednost. Takve metode se sukladno tome nazivaju *getter* i *setter*. Na primjeru 3.1. moguće je uočiti metodu koja služi za dohvat atributa navedene klase.

Nasljeđivanje je izgradnja hijerarhije između klasa. Odnosno, jedna klasa može naslijediti drugu klasu čime nasljeđuje i sve njene javne ili zaštićene atribute i metode. Objekt se može stvoriti kao konkretna ili nadređena klasa čime se omogućuje veća razina apstrakcije prilikom programiranja.

Polimorfizam predstavlja višeznačnost određenih metoda. Drugim riječima metoda s jednim nazivom može imati više oblika koji se razlikuju po broju ili tipu parametra. Ovo svojstvo se u Java programskom jeziku očituje i mogućnosti više klasa da pruže vlastitu implementaciju određene metode, njenim prepisivanjem. Najčešći primjer toga je kada svaka klasa pruži vlastitu implementaciju metode njene nadređene klase.

Java omogućuje definiranje sučelja i apstraktnih klasa. Sučelje se definira ključnom riječi *interaface*, dok se apstraktna klasa definira ključnom riječi *abstract* ispred naziva klase. Važno je napomenuti da unutar Java programskog jezika jedna klasa može naslijediti isključivo jednu klasa, a može implementirati više različitih sučelja, dok sučelje može i naslijediti više drugih sučelja.

Sučelje predstavlja poseban tip sporazuma kojeg moraju poštovati sve klase koje ga implementiraju. Odnosno, sve klase koje implementiraju sučelje, moraju i implementirati sve njegove metode. Java nudi mogućnost implementacije zadanog ponašanja metode sučelja upotrebom ključne riječi *default* prije definiranja metode. Međutim, treba biti oprezan prilikom korištenja zadanog ponašanje jer može ukazivati na grešku u dizajnu sustava. Važno je napomenuti da je unutar sučelja moguće definirati samo metode koje je potrebno implementirati, sučelje nema mogućnost definiranja atributa. Sve metode unutar sučelja imaju javno pravo pristupa.

```
1. public interface Example {
2.
3.  void doSomething();
4. }
```

Primjer 3.2. Sučelje u Java programskom jeziku

Apstraktna klasa je po strukturi slična kao i ostale klase, a razlikuje se po apstraktnim metodama koje se mogu definirati unutar njih. Za razliku od C++ programskoj jezika, Java ne zahtjeva postojanje barem jedne apstraktne metode kako bi klasa bila apstraktna. Unutar apstraktne klase se mogu definirati zajednički atributi i metode koje neke klasa trebaju naslijediti. Prilikom nasljeđivanja apstraktne klase, sve konkretne klase moraju pružiti i implementaciju svih njenih apstraktnih metoda.

```
1. public abstract class Example {
2.
3.  private String attribute;
4.
5.  public String getAttribute() {
6.   return attribute;
7.  }
8.
9.  public abstract void doSomething();
10.  }
```

Primjer 3.3. Apstraktna klasa u java programskom jeziku

Java među prvima uvodi postojanje *null* reference objekta. To je posebna vrsta reference koja može biti bilo kojeg tipa, a ono što je karakteristično za nju je da nema vrijednost. Uvođenjem ovog tipa referenci, programeri su obavezni provjeravati ima li objekt vrijednosti ili riskirati rušenje programa koje nastaje usred pokušaja rukovanja s *null* vrijednosti. Stvaratelj ove reference, Tony Hoare, nazvao ju je pogreška od milijardu dolara.

Java također uvodi i sustav anotacija. Anotacija predstavlja poseban element koji se sastoji od naziva i proizvoljnog broja parova atributa i njihovih pripadajućih vrijednosti. Njihova svrha je povezati informacije s označenim programskim elementom [2]. Ovaj pristup predstavlja veliki iskorak jer skraćuje pisanje kôda te ga čini preglednijim. Java biblioteka *Lombok*, uvodi značajan broj anotacija kao što su automatsko generiranje metoda za dohvat i postavljanje atributa, generiranje konstruktora, provjeru ispravnosti atributa i ostalih.

3.1.2. Spring razvojni okvir

Spring je razvojni okvir otvorenog kôda kojem je glavni cilj olakšati razvoj Java aplikacija. Za razliku od jednoslojnih razvojnih okvira, poput *Struts* ili *Hibernate*, glavna svrha *Spring* razvojnog okvira je pomoći programerima pri oblikovanje cijele aplikacije na dosljedan i produktivan način, objedinjujući najbolje jednoslojne razvojne okvire radi stvaranje cjelovite arhitekture [3]. Za razvoj *Spring* aplikacija, najčešće se koristi Model-Pogled-Upravitelj (engl. *Model-View-Controller*, MVC) oblikovni obrazac kojem je glavno obilježje razdvajanje aplikacije na dio koji definira vrstu i strukturu podatka, na dio koji rukuje i upravlja podacima te na dio koji služi za prikaz podataka. Iako se, uz *Spring*, može koristi *Thymeleaf* predložak za izradu pogleda na strani poslužitelja, u ovome radu pristupilo se potpunom odvajanju poslužitelja od klijenta te će tehnologije korištene za izradu klijentskog dijela aplikacije biti objašnjene u idućem poglavlju.

Nakon pisanja kôda, često je potrebno napisani kôd potkrijepiti testovima. Test predstavlja zaseban komadić kôda unutar kojeg se simulira neki zahtjev te se uspoređuje dobiveni odgovor s očekivanom vrijednosti. Moguće je pisati jedinične i integracijske testove. Za potrebu ovog rada,

pisani su integracijski testovi budući da najbolje mogu pokriti servise pisane u *Spring* razvojnom okviru. Jedinični test predstavlja testiranje jednog dijela sustava koji može raditi kao cjelina. Najčešće bi to bila metoda koja obavlja određenu operaciju. S druge strane, integracijski test predstavlja testiranje komunikacije jednog dijela sustava s ostalima. Primjer toga bio bi test koji poziva odgovarajuće programsko sučelje koje zatim poziva odgovarajuću metodu servisa. Upravljaču se šalje lažni objekt koji simulira HTTP zahtjev, a nakon što servis vrati rezultat obrade, uspoređuje se dobiveni odgovor s očekivanom vrijednosti. Očekivana vrijednost može biti konkretni rezultat obrade ili poruka o grešci. Potrebno je testirati sve očekivane slučajeve kako bi bili sigurni da će poslužitelj, u slučaju greške, vratiti odgovarajuću poruku koja se zatim može prikazati klijentu. Za automatizaciju testova korišten je *Travis CI* (engl. *Continuous Integration*) koji osigurava pokretanje svih testova kada dođe do neke promjene u izvornom kôdu. Ovaj pristup naziva se još i regresijsko testiranje koje utvrđuje utječe li promjena kôda na postojeće funkcije sustava.

3.2. Tehnologije na strani klijenta

Klijent predstavlja računalni program koji komunicira s poslužiteljem u svrhu dohvata i prikaza određenih informacija. Klijentske tehnologije su pod velikim utjecajem krajnjeg korisnika pa se iz tog razlog treba izbjegavati obavljanje poslovne logike na ovoj strani aplikacija. Glavna svrha klijenta je prikupiti informacije od korisnika, objediniti ih u smislenu cjelinu i poslati na obradu poslužitelju. Kada od poslužitelja prime odgovor, klijenti taj odgovor prikazuju korisniku kako bi bio svjestan je li neka operacija uspješno obavljena te ako nije, koji je razlog neuspjeha. Također, klijenti su zaduženi za pružanje jasnog i intuitivnog sučelja kako bi korisnicima bila jasna glavna namjena usluge kojoj pristupaju. Tehnologije korištene za razvoj klijentskog dijela aplikacije su JavaScript jezik uz *React* biblioteku, a za sam izgled korisničkog sučelja korištena je Bootstrap5 biblioteka i *Saas* predobrada vlastitih stilova.

3.2.1. JavaScript

JavaScript je skriptni jezik za razvoj dinamičnih web stranica. Prije pojava JavaScript jezika, sve stranice su bile statične što znači da je za svaku promjenu na stranici bilo potrebno napraviti novi HTML (engl. *HyperText Markup Language*) dokument i povezati ga s prethodnim. Upravo zbog toga, mogućnosti tadašnjih web stranica nisu bile velike i uglavnom su služile kao blog stranice. Čak i prije same objave ovog jezika, mnogi su ga uspoređivali s Java programskim jezikom budući da je većina osnovnih elemenata, gramatički podsjećala na Java stil. Međutim, promatrajući s klijentske strane, ovaj jezik ima totalno drugačiju primjenu. Namijenjen je da djeluje kao jezik

integriran u HTML dokument, umjesto dotadašnjih malih programa, takozvanih apleta (engl. *Applet*), koji su zauzimali fiksni, pravokutni dio stranice i koji nisu bili svjesni ničega što se može nalaziti na stranici [4]. Nadalje, navodi se kako, za razliku od Java jezika koji posjeduje raskošan vokabular, JavaScript posjeduje skroman vokabular i lakše probavljivi model programiranja. Iako mu je osnovna zadaća pisanje skripti na klijentskoj strani, u današnje vrijeme JavaScript posjeduje razvojne okvire za pisanje poslužiteljskih skripti kao što je Node.js. Još jedna prednosti JavaScript kôda je ta što ne zahtjeva posjedovanje velik razvojnih okruženja, već ga je moguće pisati u bilo kojem uređivaču teksta. Za razliku od programskih jezika, koji koriste jezične prevoditelje, takozvane kompajlere, za prevođenje cijelog kôda u strojni jezik, skriptni jezici koriste interpreter koji u realnom vremenu izvršava liniju po liniju izvornog kôda.

Postoji nekoliko načina na kojih je moguće pisati JavaScript kôd:

- Unutar *head* HTML oznake,
- Unutar *body* HTML oznake,
- Unutar vanjske datoteke.

Ako se kôd piše unutar *head* oznake, izvršava se prije samog učitavanja web stranice, za razliku od kôda napisanog unutar *body* oznake koji se izvršava kada se naleti na njega ili na kraju učitavanja ukoliko je zadnji element stranice. Kôd pisan unutar vanjske datoteke, izvršava se na mjestu gdje ga se uključuje. JavaScript kôd piše se unutar *script* oznaka.

```
1. <script>
2. console.log("Hello World!");
3. </script>
```

Primjer 3.4. JavaScript kôd za ispis na zaslon konzole

Postoje razne biblioteke koje olakšavaju pisanje JavaScript kôda kao što su *Angular* i *React*. Za potrebu ovog projekta, korištena je *React* biblioteka te će ona biti objašnjena u nastavku.

3.2.2. React

React je JavaScript biblioteka otvorenog kôda koju je razvio Meta, bivši Facebook. Glavni princip *React* biblioteke je razdvajanje pojedinih dijelova sustava na samostalne, višestruko primjenjive komponente od kojih se gradi korisničko sučelje. Svaka *React* komponenta posjeduju specifičnu funkcionalnost koju obavlja, a dizajniranju komponenti se pristupa s pogleda jedinstvenih sposobnosti koje ta komponenta posjeduje. *React* omogućuje razvoj cjelovitih blokova kôda koji odmah na prvi pogled govore na koji način će se komponenta prikazati na zaslonu korisnika. Za

kreiranje rasporeda elemenata na zaslonu, koriste se HTML oznake [5]. Glavna prednost ove biblioteke je u tome što uvodi veliku razinu fleksibilnosti i dinamičnosti. Usvaja princip jednostranične aplikacije (engl. *Single-Page Application*). Ovaj princip podrazumijeva postojanje isključivo jedne web stranice, a sama biblioteka je zadužena za pružanje načina na koji se stranice prikazuje. To znači da je zadaća *React* biblioteke pružiti sustav koji će na svaku promjenu unutarnjeg stanja, izvršiti ponovno učitavanje komponenti u ovisnosti o novom stanju. *React* komponente mogu se podijeliti na dvije skupine, a to su funkcijska i klasna komponenta. Jedna od glavnih zadaća *React* programera je mogućnost donošenja odluke u kojoj situaciji upotrijebiti koji tip komponente.

Funkcijska komponenta, kako i sam naziv kaže, predstavljena je pozivom funkcije. Odnosno, za vrijeme učitavanja komponenti, poziva se funkcija koja raspoređuje elemente unutar DOM (engl. *Document Object Model*) okvira. Ono što je karakteristično za ovaj tip komponente je to što ona nema mogućnost posjedovanja unutarnjeg stanja nego se isključivo koristi za učitavanje elemenata.

Klasna komponenta predstavljena je standardnom klasnom strukturom koja se viđa u objektno orijentiranim jezicima. Ovaj tip komponente može posjedovati unutarnje stanje na čiju se promjenu izvršava metoda koja obavlja učitavanje elemenata, *render* metoda. Ukoliko komponenta posjeduje stanje, mora posjedovati i konstruktor za postavljanje zadane vrijednosti tog stanja. Ova komponenta može posjedovati metode životnog ciklusa. To su metode koje definiraju ponašanje komponente prije nego se učita u DOM, nakon što je učitana, prije nego će se ukloniti iz DOM sučelja te ostale koje utječu na životni ciklus komponente.

Čuvanje stanja u pojedinim komponentama može vrlo brzo zakomplicirati proces razvoja budući da u velikom broju slučaja, ta stanja treba dijeliti među ostalim komponentama. Rezultat toga mogu biti velike komponente s velikim brojem atributa koje nisu baš dugoročno održivi. Rješenje za ovaj problem pruža *Redux* biblioteka koja sva stanja drži u jednom velikom, globalnom stanju, te ih po potrebi može mapirati i pružiti ostalim komponentama.

3.2.3. Bootstrap5 i Sass

Bootstrap je jedan od najpoznatijih stilskih razvojnih okvira koji omogućuje jednostavan razvoj responzivnih stranica orijentiranih prema mobilnim uređajima (engl. *Mobile-First Site*). Ovakva orijentacija proizlazi iz činjenice da sve više korisnika, web stranicama pristupa preko mobilnih uređaja. Zbog toga je potrebno dizajn prilagoditi na način da ga se prvo razvija za mobilne uređaje, a zatim za zaslone većih dimenzija. *Bootstrap* rješava ovaj problem uvodeći niz CSS (engl.

Cascading Style Sheets) klasa koje je moguće koristiti pri razvoju aplikacije. Kako bi se *Bootstrap* mogao koristiti potrebno je uključiti njegov stilski dokument, prije bilo kojih drugih vlastitih stilskih dokumenata, upotrebom *link* oznake. Mnoge *Bootstrap* komponente zahtijevaju korištenje JavaScript kôda za pravilno funkcioniranje, stoga je potrebno uključiti i *Bootstrap* skriptu upotrebom *script* oznake. S verzijom 5 uvedene su mnoge promjene koju su znatno poboljšale iskustvo pri radu s *Bootstrap* razvojnim okvirom. Primjerice, prije ove verzije, *Bootstrap* se oslanjao na *jQuery* za postizanje dinamičkih funkcionalnosti no to nije odgovaralo većini programera koji su ovaj razvojni okvir trebali koristiti s modernim JavaScript bibliotekama kao što je *React*. Zbog toga, *Bootstrap* u ovoj verziji potpuno izbacuje ovisnost o *jQuery* biblioteci. Nadalje, odbacuju podršku za Internet Explorer što omogućuje programerima dodavanje vlastitih CSS atributa za još veću fleksibilnost prilikom razvoja. Za razliku od prethodnih verzija koje su ovisile o podršci pojedinih preglednika, ova verzije omogućuje puno veću kontrolu nad obrascima koja rezultira znatno boljim dizajnom u svim preglednicima. Kao i prethodna verzija, tako i ova verzija uključuje sustav mreže za raspored elemenata na zaslonu. Međutim, ova verzija poboljšava dizajn takvog sustava, uvodeći nove CSS klase, zadržavajući pri tome postojeću arhitekturu što čini prelazak s niže verzije lakšom i ostvarivom bez previše truda. Također je ubačena i eksperimentalni CSS mrežni sustav koji bi trebao zamijeniti postojeći sustav kutija (engl. *Flexbox*). Poboljšan je i sustav navigacijske trake koji sada omogućuje veću fleksibilnost. Ubačen je i sustav plutajućih oznaka, unutar polja za unos, koji je sve više primjenjiv i u razvoju mobilnih aplikacija što smanjuje razliku između tih tehnologija i krajnjim korisnicima omogućuje jedinstveno iskustvo korištenja. Uveden je sustav rezerviranog mjesta (engl. *Placeholder*) koji poboljšava UI (engl. *User Interface*) na način da omogućuje prikaz korisnicima dijelove aplikacije dok se ona još učitava. Ova verzija je uvela i svoju biblioteku s preko 1000 ikona koje je moguće koristiti za stiliziranje aplikacije. Osim ovih, postoje mnoge druge prednosti koje nudi Bootstrap5 ali, osim toga, nudi i puno bolju dokumentaciju potkrijepljenu s raznim primjerima korištenja.

CSS sam po sebi omogućuje kreiranje dinamičkog i fleksibilnog dizajna aplikacije. Međutim, kako zahtjevi na aplikaciju postaju veći i složeniji, tako i stilovi postaju teži za održavanje. Temeljem toga, dolaze na vidjelo prednosti koje nudi *Saas* (engl. *Syntactically Awesome Style Sheets*) predobrada. *Saas* pruža mogućnosti koje ne postoje pri radu sa standardnim CSS stilovima, kao što su ugnježđivanje, nasljeđivanje i mnoge druge. *Saas* radi na način da uzme datoteke napisane vlastitom sintaksom i spremi ih kao CSS datoteke koje se zatim mogu koristiti za stiliziranje web stranice. Jedna od glavnih značajki koje uvodi su varijable unutar kojih se mogu spremati informacije koje je potrebno ponovno koristiti na više različitih mjesta kao što su boja, font ili bilo

koja druga CSS vrijednost. Ugnježđivanje omogućuje čistu i jasno sintaksu hijerarhije. Selektori se mogu ugnijezditi unutar drugih selektora kako bi simulirali vizualnu hijerarhiju koja je vidljiva unutar HTML kôda. Međutim, treba voditi računa da pretjerano ugnježđivanje može rezultirati pretjerano specifičnim CSS kôdom koji je teško održiv i predstavlja lošu praksu. *Saas* također omogućuje kreiranje djelomičnih stilova. Ovi stilovi označavaju se s donjom crtom u nazivu i signaliziraju da ih je potrebno uključiti među ostale stilove, a ne pretvoriti u zaseban CSS kôd. Kako bi se *Saas* kôd držao DRY (engl. *Don't Repeat Yourself*) principa, uvodi poseban oblik grupiranja stilova koje je zatim moguće koristiti kroz cijeli kôd. Nasljeđivanje omogućuje dijeljenje stilova između više različitih selektora uz pomoć klase čiji se stil primjenjuje samo kad ju se naslijedi. *Saas* također omogućuje i obavljanje matematičkih operacija unutar kôda što može biti korisno prilikom računanja potrebnih dimenzija i sličnog. Osim ovih, *Saas* ima mnogo drugih mogućnosti koji znatno olakšavaju pisanje CSS kôda i dizajniranje korisničkog sučelja.

```
1. .header {
2.     .navbar {
3.         background: #6ab446;
4.     }
5. }
```

Primjer 3.5. Ugnježđivanje *Saas* stilova

Na primjeru 3.5. vidljivo je kako funkcionira ugnježđivanje *Saas* stilova. Pozadinska boja, definirana unutar *navbar* selektora, primijeniti će se samo na one elemente koji na sebi imaju definiranu tu klasu i čiji je roditelj element s klasom *header*.

3.3. Android tehnologije

Android je operacijski sustav namijenjen za mobilne uređaje. Temeljen je na Linux jezgri što omogućuje veliku fleksibilnost samog sustava. U današnje vrijeme, ovaj operacijski sustav moguće je pronaći i u pametnim televizorima, automobilima i mnogim drugim električnim uređajima. Aplikacije razvijane za ovaj operacijski sustav koriste *apk* datotečni nastavak te ih je potrebno instalirati na uređaj na kojem će se koristiti. Iako su se ove aplikacije dugo razvijala u Java programskom jeziku, u današnje vrijeme sve se više programera odlučuje za Kotlin programski jezik budući da nudi rješenje za mnoge nedostataka prisutne u Java programskom jeziku te će detaljnije biti objašnjen u nastavku.

3.3.1. Kotlin programski jezik

Kotlin je objektno orijentirani programski jezik novije generacije. Temeljen je na Java programskom jeziku, a razvijen je u svrhu rješavanja mnogih problema s kojima se susreću Java

programeri kao što je *null* referenca ili pretjerano opširan kôd. Dovodi proces razvoja aplikacija na sasvim novu razinu poboljšavajući kvalitetu kôda, sigurnost i općenito performanse razvoja. Službena podrška za Android platformu predstavljena je 2017. od strane Google poduzeća ali ovaj programski jezik, postoji i puno prije toga.

Karakteriziraju ga sljedeće kvalitete:

- Sigurnost s aspekta *null* reference i promjenjivosti podataka,
- Problem se može otkriti puno brže za vrijeme razvoja u odnosu na padanje aplikacije i narušavanje kvalitete korištenja,
- Potrebno je puno manje kôda za postići osnovne funkcionalnosti što ga čini čitljivijim i razumljivijim,
- Može raditi u skladu s Java programskim jezikom što osigurava očuvanje postojećeg ekosustava biblioteka i razvojnih okvira,
- Podržava razne platforme kao što su Android, poslužitelji, desktop aplikacije, klijentski kôd za web preglednike pa čak i ugradbene sustave [10].

Kotlin kôd, za razliku od većine drugih programskih jezika, ne mora, ali može, završavati interpunkcijskim znakom točka sa zarezom. Kada se govori o varijablama, ne zahtjeva eksplicitno definiranje tipa varijable, već tu informaciju može saznati iz konteksta. Međutim, tip varijable se može pružiti dodavanjem znaka dvotočka nakon naziva varijable, nakon čega se može specificirati njezin tip. Također, potrebno je pružiti informaciju može li se varijabli promijeniti vrijednost ili je ona konstantna. Ukoliko se vrijednost može promijeniti, potrebno je prije naziva varijable upotrijebiti ključnu riječ *var*, a ukoliko ne može, ključnu riječ *val*. Kako je već prije rečeno, ovaj programski jezik pruža sigurnost pri radu s *null* referencama. To se postiže na način da varijabla ne može poprimiti ovu referencu ukoliko programer eksplicitno ne odredi suprotno dodavanjem znaka upitnik na tip varijable. Dodatnu razinu sigurnosti pružaju sigurnosni pozivi metoda koji osiguravaju da će se metoda pozvati samo u onim slučajevima kada objekt nad kojim se poziva nema *null* referencu, u suprotnom će se poziv zanemariti. Na ovaj način, izbjegava se pojavljivanje iznimke *null* reference koja je bila česta pojava u Java programskom jeziku. Također, uvodi razliku i među skupovima podataka na način da razlikuje promjenjive od nepromjenjivih skupova uvodeći niz sučelja i metoda za rukovanje ovim podacima. Prema tome, nepromjenjivim skupovima se ne mogu dodavati ili uklanjati podaci, moguće ih je samo čitati.

Definiranje metoda postiže se upotrebom ključne riječi *fun* nakon čega slijedi njezin naziv. Povratni tip metode, definira se nakon specifikacije njezinih parametara, upotrebom znaka dvotočka.

Kotlin klase, definiraju se slično kao i Java klase upotrebom ključne riječi *class*. Prednost ovoga programskoj jezika dolazi do vidjela kod definiranja atributa klasa. Naime, u Java programskom jeziku, bilo je potrebno pružiti za svaki privatni atribut odgovarajuće metode za dohvat i postavljenje vrijednosti. Ovdje to nije potrebno, već se te metode generiraju automatski prilikom prevođenja kôda što sami kôd čini puno preglednijim i čitkijim. Još jedna razlika između ova dva jezika očituje se i u definiranju konstruktora. Kako bi se konstruktor definirao u ovome programskom jeziku, potrebno je samo upotrijebiti ključnu riječ *constructor* i navesti potrebe atribute od kojih će konstruktor izgraditi objekt. Svi konstruktori, koji se definiraju na ovaj način, nazivaju se sekundarnim konstruktorima. Primarni konstruktor je onaj koji se definira u zaglavlju klase te on ne može sadržavati dodatni kôd. Zbog toga, inicijalizacija podataka se obavlja u *init* bloku kôda koji će biti pozvan prilikom stvaranja klase. Međutim, moguće je atribute definirati direktno u primarnom konstruktoru dodavanjem *var* ili *val* svojstva što znatno skraćuje pisanje kôda. Ukoliko klasa predstavlja samo skup podataka, moguće ju je označiti s *data* ključnom riječi što će pružiti metodu za ispis objekata te klase te će se oni moći uredno pretvoriti u znakovni niz, spreman za ispis.

Nasljeđivanje se ostvaruje pomoću znaka dvotočka nakon naziva klase. Važno je napomenuti da klasa koja se nasljeđuje, mora biti označena kao otvorena što se postiže s ključnom riječi *open* prilikom same definicije klase. Također, sve metode i atributi koji se nasljeđuju moraju biti označeni s istom oznakom. Prilikom nasljeđivanja, potrebno je obaviti i poziv konstruktora klase koja se nasljeđuje. Implementacija sučelja, obavlja se na isti način kao i nasljeđivanje, a razlika je u tome što sučelja ne zahtjeva poziv konstruktora, budući da ih ne posjeduje.

Osim svih prednosti navedenih u ovome poglavlju, Kotlin pruža i razne druge mogućnosti koje olakšavaju pisanje kôda i sam proces razvoja aplikacija. Neke od tih mogućnosti, biti će detaljnije objašnjene kroz iduća poglavlja.

3.3.2. MVVM obrazac i LiveData

MVVM (engl. *Model-View-ViewModel*) je oblikovni obrazac koji je postao standardnom praksom prilikom razvoj android aplikacija, a predstavlja rješenje nedostataka koji su se pojavljivali u MVC obrascu. Karakterizira ga razdvajanje korisničkog sučelja, odnosno pogleda, od poslovne logike aplikacije. Model predstavlja apstrakciju podataka, a radi u kombinaciji s pogledom modela (engl.

ViewModel) kako bi dohvaćao i spremao podatke. Pogled modela izlaže one podatke koji su bitni za pogled, a služi kao veza između modela i pogleda. Pogled modela ne smije imati nikakvu referencu na sam pogled niti smije direktno utjecati na njega. Svrha pogleda je informirati pogled modela o akcijama koje korisnik obavlja u sustavu. Ovaj sloj se pretplaćuje na pogled modela te promatra njegove unutarnje promjene na koje ažurira prikaz ali pri tome ne smije sadržavati nikakvu poslovnu logiku aplikacije. Ovaj obrazac implementira se uz pomoć *LiveData* koncepta. On predstavlja spremnik podataka čije promijene je moguće promatrati radi obavljanja određene akcije. Za razliku od običnih promatrača, ne ovisi o životnom ciklusu pojedine komponente kao što je fragment, zaslon i slično. Ovo omogućuje ažuriranje samo onih komponenti koji su u aktivnom životnom ciklusu.

Upotreba ovog koncepta omogućuje sljedeće prednosti:

- Osigurava da korisničko sučelje odgovora unutarnjem stanju podataka,
- Nema curenja memorije budući da je veza za životni ciklus komponente,
- Nema padanja aplikacije zbog neaktivnih komponenti budući da u tom stanju ne primaju događaje,
- Komponente korisničkog sučelje ne prestaju promatrati podatke već ovaj sustav omogućuje totalno upravljane time budući da je svjestan životnog ciklusa komponente,
- Kada komponenta postane neaktivna, prilikom ponovnog aktiviranje, dobiju se najnoviji podaci,
- Ukoliko se zaslon ili fragment ponovno stvori kao rezultat okretanja zaslona ili sličnog događaja, dobiju se najnoviji dostupni podaci,
- Omogućeno dijeljenje resursa.

Za implementaciju ovog ponašanja potrebno je kreirati klasu koja nasljeđuje *ViewModel* klasu. Unutar nje potrebno je stvoriti privatni atribut koji predstavlja promjenjivi podatak, te javni atribut *LiveData* sučelja koji poprima vrijednost prethodno navedenog atributa. Pogled se pretplaćuje na javni atribut te promatra njegovo unutarnje stanje. Kada dođe do promjene u privatnom atribute, odašilje se obavijest o promjenama na koju pogled može reagirati i ažurirati zaslon.

3.4. WebSocket

WebSocket predstavlja dvosmjerni oblik komunikacije između poslužitelja i klijenta. Kada je veza uspostavljena, ona ostaje otvorena sve dok ju poslužitelj ili klijent ne odluče zatvoriti. Ovaj oblik

komunikacije najveću primjenu ima unutar sustava gdje je komunikaciju potrebno voditi u stvarnom vremenu kao što su sustavi razmjene poruka.

Konfiguracija *WebSocket* komunikacije na poslužitelju ne zahtjeva puno vremena. Potrebno je definirati posrednika poruka (engl. *Message Broker*) s putanjom koja će omogućiti prijenos poruke do klijenta. Kako bi se osigurala STOMP podrška, potrebno je registrirati i krajnju točku ovog protokola. STOMP predstavlja jednostavni protokol dizajniran za asinkronu razmjenu poruka između klijenata uz neki poslužiteljski posrednik. Nastao je iz potrebe za povezivanjem raznih posrednika poruka iz skriptnih jezika [13].

```
1. @Configuration
2. @EnableWebSocketMessageBroker
3. public class WebSocketConfig implements WebSocketMessageBrokerConfigu
   rer {
4.
5.  @Override
6.  public void configureMessageBroker(MessageBrokerRegistry registry) {
7.   registry.enableSimpleBroker("/topic");
8.   registry.setApplicationDestinationPrefixes("/app");
9.  }
10.
11.   @Override
12.   public void registerStompEndpoints(StompEndpointRegistry registry)
    {
13.    registry.addEndpoint("/chat").setAllowedOriginPatterns("*");
14.    registry.addEndpoint("/chat").setAllowedOriginPatterns("*").withS
   ockJS();
15.   }
16.  }
```

Primjer 3.6. Konfiguracija posrednika poruka

Nakon ove konfiguracije, potrebno je samo još definirati model poruke te upravljač koji će omogućiti pozivanje posrednika poruka.

Na klijentskoj strani aplikacije, za svrhu ovoga rada, korištena je *react-stomp* biblioteka koje dolazi s ugrađenom komponentom *SockJs* biblioteke za razmjenu poruka. Sve što je potrebno napraviti za ostvarivanje komunikacije je definirati URL na koji će se poruka odaslati, definirati koji posrednik poruka se koristi, koja akcija će se obaviti prilikom uspostava veze, koja akcija će se obaviti prilikom prekida veze i što će se obavljati kada se primi nova poruka. Prilikom primanja nove poruke, od poslužitelja će se ponovno zatražiti dohvat svih poruka kako bi se one mogle prikazati unutar odgovarajuće komponente.

```
<SockJsClient
url={`${Environments.LOCAL}/chat`}
topics={['/topic/messages']}
onConnected={this.onConnected}
onDisconnect={console.log("Disconnected!")}
onMessage={msg => this.onMessageReceived(msg)}
debug={false}
/>
```

Primjer 3.7. Konfiguracija razmjene poruka na strani klijenta

Za kraj, potrebno je pružiti i Android konfiguraciju razmjene poruka. U tu svrhu, također se koristi STOMP protokol biblioteka koju pruža [14]. Unutar klase fragmenta, koja služi za prikaz poruka, definira se argument tipa *StompClient* koji koristi Kotlin mogućnost kasne inicijalizacije. Prilikom stvaranja pogleda, ovaj klijent se inicijalizira pružajući URL odašiljanja poruke te pozivom *connect* metode. Također, potrebno je pružiti i URL do posrednika poruke, ponašanje u slučaju dolaska do pogreške te obaviti pretplatu na klijent koji osluškuje događaj primitak nove poruke na koji reagira i obavlja određenu akciju.

```
stompClient = Stomp.over(Stomp.ConnectionProvider.OKHTTP, Environment
Enum.SOCKET.url)
        stompClient.connect()

        stompClient.topic("/topic/messages").doOnError {
            Log.e("Socket error", "${it.message}")
        }.subscribe {
            messagesViewModel.addMessage(
                gson.fromJson(it.payload, MessageDTO::class.java)
            )
          }
```

Primjer 3.8. Konfiguracija razmjene poruka u Android aplikaciji

4. WEB APLIKACIJA

Web dio aplikacije, namijenjen je raznim organizacijama koje bi htjele svoje uslugu približiti studentima. Svaki korisnik, koji se registrira putem web sučelja, poprima ulogu člana organizacije. Studentima se nije moguće prijaviti u web aplikaciju.

4.1. Realizacija poslužitelja

Izreda poslužiteljskog dijela aplikacije kreće definiranjem modela. Model predstavlja poseban oblik klase unutar kojeg se nalaze svi podaci koji ga opisuju. Objekti nastali iz tih klasa, nazivaju se jednostavnim Java objektima (engl. *Plain Old Java Object*, POJO). Za takve klase, karakteristično je da su izgrađene samo od atributa i metoda za dohvat i postavljanje željenih atributa. Unutar *Spring* razvojnog okvira, koristi se *Hibernate* za objektno-relacijsko mapiranje (engl. *Object-Relational Mapping*, ORM) ovakvih klasa u relacije unutar baze podataka.

```
1. @Entity
2. public class Model {
3.
4.  @Id
5.  @GeneratedValue(strategy = GenerationType.IDENTITY)
6.  private Long id;
7.
8.  private String name;
9. }
```

Primjer 4.1. POJO klasa modela

Primjer 4.1. prikazuje jednostavan model. Za modele je karakteristično da se moraju označiti s *Entity* anotacijom kako bi *Hibernate* za vrijeme prevođenja znao da se radi o modelu te kako bi izvršio pripadajuće mapiranje. Svaka klasa označena ovom anotacijom, zahtjeva postojanje jedinstvenog identifikatora koji će predstavljati primarni ključ u bazi podataka. Iz tog razloga je potrebno takav atribut označiti s *Id* anotacijom i dodijeliti mu odgovarajuću strategiju dodjeljivanja. Svi ostali atributi klase odgovaraju atributima relacije koja će nastati nakon mapiranja. Atribut također može biti i drugi model prilikom čega je potrebno izvršiti dodatna mapiranja, odnosno potrebno je odrediti vrstu veze između relacija.

Veza jedan na prema jedan označava da polazišnom modelu pripada isključivo jedan zapis modela koji se referencira. Ova vrsta mapiranja se postiže upotrebom *OneToOne* anotacije. Prilikom mapiranja, *Hibernate* će stvoriti strani ključ unutar relacije koja je proglašena vlasnikom veze. Vlasnik veze definira se pomoću *mappedBy* atributa unutar navedene anotacije, odnosno one

strana koja nema postavljen ovaj atribut je vlasnik. Upotreba ovog atributa karakteristična je za dvostrana mapiranja.

Veza jedan na prema više i više na prema jedan često dolaze u paru čime se također ostvaraju dvostrano mapiranje. One označavaju da jedan zapis polazišnog modela može imati više zapisa referenciranog modela. Model unutar kojeg se nalazi *ManyToOne* anotacija postaje vlasnik veze te unutar njega *Hibernate* kreira strani ključ. Važno je napomenuti da se s *OneToMany* anotacijom mogu označiti samo skupovi podataka budući da ih može biti više.

Zadnji primjer veze je više na prema više i to je najsloženiji način mapiranja budući da podrazumijeva postojanje vezne tablice. Ostvaruje se upotrebom *ManyToMany* anotacije nad skupom podataka. Nakon definiranja ove veze, potrebno je definirati veznu relaciju upotrebom *JoinTable* anotacije unutar koje se definiraju vezni stupci pomoću *JoinColumn* anotacije. Ovi stupci ujedno postaju primarni i strani ključeve ove relacije. Vezna relacija se može stvoriti i uz pomoć dodatne klase koja ju tada predstavlja. U tom slučaju nije potrebno koristiti vezu više na prema više nego se jednako ponašanje ostvaruje upotrebom kombinacija jedan na prema više i više na prema jedan veze. Ovaj pristup se najčešće koristi kada vezna relacija, osim stupaca koji označavaju identifikatore referenciranih relacija, treba sadržavati još neke dodatne atribute.

Za vrijeme kreiranje ovih veza moguće je definirati kaskadni tip koji definira ponašanje prilikom izmjene podataka, te tip dohvata koji igra veliku ulogu u performansama aplikacije. Postoji dva tipa dohvata podataka, a to su lijeni, koji je preporučeni, i žurni dohvat.

Prilikom korištenja žurnog dohvata, svi podaci se dohvaćaju odmah, neovisno o tome koriste li se ili ne. To može predstavljati problem pri velikom skupu podataka, budući da će se uvijek dohvaćati što će u krajnjem slučaju rezultirati s N + 1 problemom upita. Ovaj problem govori da će se za jedan upit na bazu podataka izvršiti još onoliko upita koliko je podataka vezano za model koji se dohvaća. Lako je uočljivo da pri velikom skupu podataka to može rezultirati i u nekoliko tisuća upita na bazu podataka što znatno usporava izvršavanje aplikacije. Rješenje ovog problema ležu u lijenom dohvatu podataka. Za ovaj tip dohvata karakteristično je da se podaci dohvaćaju samo onda kada se koriste, odnosno tek kada se pozove metode za dohvat toga atributa. Međutim, to samo po sebi neće riješiti N + 1 problem već ga samo odgoditi. Pravo rješenje problema nalazi se u kombinaciji lijenog dohvata i dodavanja pridruženog dohvata (engl. *Join Fetch*). Pridruženi dohvat označava da će se prilikom izvršavanja upita, na njega nadovezati još dio koji u istom trenutku dohvaća sve vezne podatke. Drugim riječima, željeni model i svi njegovi vezni podaci se dohvate u jednom upitu za razliku od N upita koji bi inače bili potrebni. Lijeni dohvat ovdje igra

glavnu ulogu u tome što pridruženim dohvatom treba dohvatiti samo podatke za koje smo sigurni da će se koristiti, a za ostale ostaviti lijeni budući da ni veliki upiti koji dohvaća jako puno podataka nije sretno rješenje s pogleda performansi.

```
1. @NamedEntityGraph(name = "graph_name",
2.   attributeNodes = {
3.     @NamedAttributeNode(value = "attribute")
4.   }
5. )
```

Primjer 4.2. Ostvarivanje pridruženog dohvata

Na primjeru 4.2. može se vidjeti kako *Hibernate* ostvaraju pridruženi dohvat uz pomoć grafa entiteta. *NamedEntityGraph* anotacija postavlja se iznad klase koja predstavlja entitet. Grafu je potrebno dodijeliti naziv te atribute koje je potrebno dohvatiti upotrebom *NamedAttributeNode* anotacije. Moguće je definirati i podgraf koji se koristi za specificiranje pridruženog dohvata određenih atributa koji se nalaza unutar glavnog atributa koji se dohvaća. Ovako definirani graf potrebno je ubaciti u određenu metodu dohvata podataka iz repozitorija upotrebom *EntityGraph* anotacije.

Repozitorij je Java sučelje unutar kojeg se nalaze metode koje predstavljaju upite na bazu podataka. *Spring* razvojni okvir nudi nekoliko vrsta repozitorija unutar kojih se već nalaze predefinirane metode kao što su metoda za spremanje zapisa, brisanje zapisa, dohvat pojedinog i mnoge druge. Ove metode predstavljaju CRUD (engl. *Create-Read-Update-Delete*) operacije, te se sukladno tome, jedan od repozitorija najniže razine zove *CrudRepository* kojeg zatim nasljeđuje ostale vrste repozitorija. Jedini repozitorij koji se nalazi na najnižoj razini naziva se samo *Repository* i on ne sadrži predefinirane metode. Nakon nasljeđivanja jednog od repozitorija, moguće je definirati izvedene metode upita koje imaju oblik običnih Java metoda no u pozadini se prevode u SQL (engl. *Structured Query Language*). Također, moguće je i pisanje vlastitih upita upotrebom *Query* anotacije. Ovi upiti mogu se pisati pomoću JPQL (engl. *Java Persistence Query Language*) ili izvornim SQL kôdom, upotrebom *nativeQuery* parametra unutar navede anotacije. Osim nasljeđivanja jednog od repozitorija kojeg nudi *Spring*, moguće je pisati i vlastite implementacije sučelja upotrebom *Criteria API* (engl. *Criteria Application Programming Interface*) kojeg nudi *Hibernate*.

```
1. public interface UserRepository extends JpaRepository<User, Long> {
2.
3.  @EntityGraph("user_graph")
4.  Optional<User> findUserById(Long userId);
5. }
```

Primjer 4.3. Repozitorij

Na primjeru 4.3. moguće je vidjeti sučelje repozitorija koji nasljeđuje jedan od *Spring* repozitorija. Unutar sučelja, nalazi se izvedena metoda upita kojoj je dodijeljen graf entiteta kako bi se postigao pridruženi upit.

Nakon definiranja modela, potrebno je definirati objekte za prijenos podataka (engl. *Data Transfer Object*, DTO). To su također POJO klase koje služe za reprezentaciju modela. Potreba za njima se javila budući da nije poželjno dopustiti direktno upravljanje modelima, nego se ono vrši kroz ove objekte. Njih se može serijalizirati u JSON (engl. *JavaScript Object Notation*) ili neki drugi transportni format i deserijalizirati natrag u Java objekt upotrebom *Jackson* ili neke druge biblioteke. Unutar ovih objekata, enkapsuliraju se podaci koji se razmjenjuju između različitih dijelova sustava, te se može ograničiti što se od podataka modela vraća klijentu. Uz ove objekte, potrebno je pružiti i odgovarajući sustav mapiranja modela u DTO i obrnuto. Ovo se može postići upotrebom *MapStruct* biblioteke. Ona omogućuje stvaranje apstraktne klase ili sučelja unutar kojih se definiraju metode mapiranja upotrebom odgovarajućih anotacija nakon čega se automatski stvara njihova implementacija.

```
1. @Mapper(componentModel = "spring",
2.   unmappedTargetPolicy = ReportingPolicy.IGNORE,
3.   uses = {CodeBookMapper.class})
4. public abstract class UserMapper {
5.
6.  @Mappings({
7.    @Mapping(target = "firstName", source = "entity.firstName"),
8.    @Mapping(target = "lastName", source = "entity.lastName"),
9.    @Mapping(target = "roles", source = "entity.roles"),
10.   })
11.   public abstract UserDTO userToUserDTO(User entity);
12.  }
```

Primjer 4.4. Apstraktna klasa za mapiranje

Na primjeru 4.4. vidljivo je kako je dovoljno unutar klase pružiti apstraktnu metodu i odgovarajuća pravila mapiranja kako bi mapiranje bilo uspješno implementirano. Parametar *traget* predstavlja naziv atributa unutar DTO klase, dok parametar *source* predstavlja naziv atributa modela. Također, vidljivo je kako je moguće koristiti i druge klase mapiranja upotrebom *uses* parametra.

Nakon izrade DTO klasa i sustava mapiranja, može se krenuti na izradu servisa koji će obavljati određene operacije. Ovdje dolazi do jednog vrlo važnog koncepta *Spring* razvojnog okvira, a to je inverzija kontrole (engl. *Inversion of Control*), odnosno konkretnije rečeno, ubrizgavanje ovisnosti (engl. *Dependency Injection*). Inverzija kontrole može se objasniti kao upravljanje po principu „Ne zovi me, ja ću zvati tebe“. Ovo znači da kôd razvojnog okvira poziva aplikacijski kôd, upravljajući cjelokupnim tijekom aktivnosti, umjesto da aplikacijski kôd poziva kôd razvojnog okvira. Ubrizgavanje ovisnosti opisan je kao princip u kojem se ovisnosti ubacuju u aplikacijski objekt za vrijeme izvođenja za razliku od tradicionalnog pristupa u kojem aplikacija povlači ovisnosti iz svojeg okruženja [3]. Ovo je jasno vidljivo prilikom ubrizgavanja ovisnosti u servise ili upravljače.

Ubrizgavanje ovisnosti se može vršiti na tri načina:

- Atributsko ubrizgavanje upotrebom *Autowired* anotacije,
- Ubrizgavanje kroz metodu postavljanja atributa,
- Ubrizgavanje kroz konstruktor.

Servis predstavlja sučelje koje sadrži metode za obavljanje određenih operacija. Za svaki servis je potrebno pružiti njegove implementaciju koja se proglašava servisom upotrebom *Service* anotacije. Atributi servisa mogu biti razni repozitoriji, klase za mapiranje, drugi servisi i ostalo, čije je ovisnosti potrebno ubrizgati na jedan od navedenih načina. Ukoliko metoda servisa obavlja neku transakciju nad bazom podataka, poželjno ju je označiti s *Transactional* anotacijom kako bi se moglo definirati razno ponašanje kao što je obrada iznimke.

```
1. public interface UserService {
2.
3.  UserDTO getCurrentUserInfo();
4. }
```

Primjer 4.5. Sučelje servisa

```
1. @Service
2. @RequiredArgsConstructor(onConstructor = @__(@Autowired))
3. public class UserServiceImpl implements UserService {
4.
5.  private final UserRepository userRepository;
6.
7.  private final UserMapper userMapper;
8.
9.  @Override
10.   public UserDTO getCurrentUserInfo() {
11.    User user = userRepository.findUserById(AuthorizedRequestContext.
   getCurrentUser().getId()).orElse(null);
12.
13.    return userMapper.userToUserDTO(user);
14.   }
15.  }
```

Primjer 4.6. Implementacija servisa

Kako bi servisi bili funkcionalno, potrebno je definirati odgovarajuće programsko sučelje (engl. Application Programming Interface, API) preko kojeg će se servisi pozivati. Programska sučelja definiraju se u posebnim klasama koji se nazivaju upravljači. Kako bi se klasa proglasila upravljačem, potrebnu ju je označiti s *RestController* anotacijom. Unutar upravljača pišu se metode koje se mapiraju na određeni HTTP (engl. *Hypertext Transfer Protocol*) protokol. Ove metode od klijenta primaju zahtjev, prosljeđuju ga servisu na obradu i vraćaju klijentu odgovor nakon obrade.

Mapiranje je moguće pružiti za sljedeće HTTP metode:

- GET – kada klijent zahtjeva od servera neke informacije,
- POST – zapisivanje novog zapisa u bazu podataka,
- PUT – uređivanje postojećeg zapisa u bazi podataka,
- DELETE – brisanje zapisa iz baze podataka,
- PATCH – slično kao i PUT, predstavlja uređivanje resursa.

```
1. @RestController
2. @RequestMapping("/api/users")
3. @RequiredArgsConstructor(onConstructor = @__(@Autowired))
4. public class UserController {
5.
6.  private final UserService userService;
7.
8.  @GetMapping("/current")
9.  public ResponseEntity<ResponseMessage<UserDTO>> getCurrentUserInfo()
    {
10.    return ResponseEntity.ok(new ResponseMessage<>(
11.      userService.getCurrentUserInfo())
12.    );
13.   }
14.  }
```

Primjer 4.7. Upravljač

Na primjeru 4.7. vidljivo je kako je moguće definirati osnovnu putanju API poziva, upotrebom *RequestMapping* anotacije, a specifičnu putanju HTTP metode upotrebom odgovarajuće anotacije koja predstavlja tu metodu.

Nakon definiranja željenih metoda programskog sučelja, potrebno ih je zaštiti od neovlaštenog korištenja. Za tu potrebu, *Spring* nudi već gotove klase koje je moguće naslijediti i prepisati zadana ponašanja. Svaku konfiguracijsku klasu, potrebno je označiti s *Configuration* anotacijom, a klasu koje predstavlja konfiguraciju sigurnosti i s *EnableWebSecurity*. Unutar ove klase, moguće je navoditi API pozive na koje se stavlja zaštita, definirati razni filtere, metodu raspršivanja (engl. *Hashing*) korisniče lozinke i mnoge druge sigurnosne postavke. Raspršivanje predstavlja postupak tijekom kojeg se odgovarajući zapis, upotrebom zadanog ključa, pretvori u jedinstvenu kombinaciju znakova. Ovaj proces je jednosmjeran što znači da se jednom raspršeni zapis, više ne može vratiti u originalni oblik. Upravo zbog toga, predstavlja dobar način za pohranu korisničke lozinke. Za potrebe ovog rada, korištena je *Bcrypt* metoda raspršivanja lozinke.

4.2. Realizacija klijenta

Izrada klijentskog dijela aplikacije kreće definiranjem sustava za upravljanje stanjem upotrebom *Redux* biblioteke. *Redux* se pridržava osnovnog koncepta *React* biblioteke, a to je jednosmjeran tok podataka. Ovakav tok podataka karakterizira pogled koji pokrene određenu akciju, ta akcija zatim prolazi kroz otpravnik (engl. *Dispatcher*) i spremnik (engl. *Store*) te će na kraju promijeniti pogled kada se promijeni stanje. *Redux* akcije enkapsuliraju informacije o promjeni stanja, a samo stanje se pohranjuje u spremnik. Spremnik je realiziran pomoću oblikovnog obrasca jedinac (engl. *Singleton*). Samim time je jasno kako ne može postojati više spremnika stanja. Nadalje, ne postoji jedinstveni otpremnik već koristi višestruke reduktore (engl. *Reducer*) koji preuzimaju informacije

iz akcije i sažimaju ih na novo stanje koje se pohranjuje u spremnik. Kada se stanje promjeni, pogled može na to reagirati pretplaćujući se na spremnik [6].

Akcije predstavljaju JavaScript objekte koji se sastoje od tipa akcije i rezultata. Dok je tip akcije predstavljen znakovnim nizom (engl. *String*), rezultat može biti bilo što i nije obavezan. Izvršavanje akcije se može pozvati u sloju pogleda i može biti nešto jednostavno kao što je pritisak gumba. Nakon što se akcija otpremi, prolazi kroz sve reduktore dok ne naiđe na onaj koji ju može obraditi.

```
1. const setCurrentUser = user => ({
2.     type: UserActionTypes.SET_CURRENT_USER,
3.     payload: user
4. })
```

Primjer 4.8. *Redux* akcija

Reduktor je čista funkcija što znači da će za iste ulaze, uvijek proizvesti isti izlaz. Posjeduje dva ulazna parametra, a to su trenutno stanje i akcija. Stanje predstavlja kompletno stanje iz spremnika zbog čega je potrebno, prije mijenjanja određenog dijela, raspršiti preostale dijelove. Stanje je nepromjenjivo, što znači da uvijek vraća novo stanje umjesto da mijenja postojeće.

```
1. const INITIAL_STATE = {
2.     currentUser: null
3. }
4.
5. const userReducer = (state = INITIAL_STATE, action) => {
6.     switch (action.type) {
7.         case UserActionTypes.SET_CURRENT_USER:
8.             return {
9.                 ...state,
10.                  currentUser: action.payload
11.             }
12.          default:
13.              return state;
14.      }
15.  }
```

Primjer 4.9. Reduktor

Na primjeru 4.9. jasno su vidljivi objašnjeni principi. Reduktor provjerava pomoću *switch* naredbe tip pristigle akcije. Ukoliko tu akciju može izvršiti, vraća novu verziju stanja koja se sastoji od raspršenih elemenata prošlog stanja i promijenjenog elementa za kojeg je zadužena akcija.

Spremnik, kako je već ranije rečeno, čuva jedno globalno stanje. Ne postoji više spremnika niti više stanja. Kako bi se stvorio spremnik, potrebno je pozvati *createStore* metodu koja kao parametar prima osnovni reduktor, a može primiti i izborni parametar koji predstavlja međuslojeve

(engl. *Middlewares*). Osnovni reduktor nije ništa drugo već spoj svih pojedinačnih reduktora. Ovaj spoj ujedno čini i glavno stanje aplikacije. Kreiranjem spremnika, gotova je inicijalizacija *Redux* sustava.

```
1. const middlewares = [];
2.
3. if(process.env.NODE_ENV === 'development') {
4.     middlewares.push(logger);
5. }
6.
7. const store = createStore(rootReducer, applyMiddleware(...middlewares
   ));
8.
9. const persistor = persistStore(store);
```

Primjer 4.10. Spremnik s međuslojem

Kako bi jednostranična aplikacija funkcionirala kako je zamišljeno, potrebno je definirati određene putanje (engl. *Routes*) na čiji poziv će *React* ažurirati određene dijelove aplikacije. Budući da *React*, sam po sebi, ne dolazi s ugrađenim alatom za rukovanje putanjama, postoje razne biblioteke koje to omogućuju, a za potrebu ovog rada, korištena je biblioteka *React-Router*. Ova biblioteka osluškuje URL (engl. *Uniform Resource Locator*) promjene kako bi aplikaciju održavala sinkroniziranom, ažurirajući pojedine komponente. Osim dinamičnog usmjeravanja i jednostavne navigacije, ova biblioteka nudi različite mogućnosti koje su potrebne za izradu standardne web stranice, a neke od tih mogućnosti uključuju:

- Navigaciju naprijed i natrag unutar aplikacije, čuvanje povijesti i obnavljanje stanja,
- Učitavanja odgovarajuće komponente preko određene URL putanje,
- Preusmjeravanje korisnika s jedne putanje na drugu,
- Podršku za prikaz 404 stranice kada niti jedna putanja ne odgovara URL putanji [7].

Osnovna komponenta, koja dolazi s ovom bibliotekom, naziva se *BrowserRouter*. Osnovnu komponentu aplikacije, potrebno je omotati s ovom komponentu te se na taj način proglašava korištenje putanja u aplikaciji. Ovaj pristup omogućuju rukovanje s povijesti glavne komponente, dijeleći s ostalim komponentama atribute koji se nalaze unutar programskog sučelja što rezultira s mogućnošću komponenti da uspoređuju URL putanju i na temelju toga prikazuju određeni sadržaj. Ovaj pristup ne predstavlja zapravo putanje već uvjetno prikazivanje komponenti na temelju uzorka koji se dobije iz trenutne URL putanje [7]. Kako bi se omogućio isključivo rukovanje putanjama potrebno je koristiti *Switch* komponentu s kojom je moguće omotati sve *Route* komponente. Prednost ovog načina je što se putanja, koja je zadana s *path* atributom, može usporediti kao cjelina ili samo kao dio. To se postiže upotrebom *exact* atributa koji nalaže da se

cjelokupna putanja mora podudarati s URL putanjom, dok izostavljanje ovog atributa pronalazi podudaranje čak i ako samo dio putanje odgovara definiranoj putanji. Osim ovoga bitno je još spomenuti *component* atribut unutar kojeg se zadaje komponenta koju je potrebno učitati i *render* atribut koji prima funkcijski poziv na temelju kojeg je moguće provjeriti dodatne uvjete za učitavanje komponente. Dodatne provjere su potrebne kada je neku putanju potrebno zaštiti od neovlaštenog pristupa u ovisnosti o ulozi korisnika ili nečeg drugog.

```
1. <Switch>
2. <Route exact path='/' component={HomePage} />
3. <Route exact path='/organizations' component={OrganizationsPage} />
4. <Route exact path='/organizations/:id' component={OrganizationPage} /
   >
5. </Switch>
```

Primjer 4.11. Definiranje putanja uz *React-Router*

Na primjeru 4.11. može su uočiti kako *React-Router* omogućuje ugradnju varijabli unutar putanje, kao što je jedinstveni identifikator, na temelju koje je moguće još više specificirati na koji način će se komponenta prikazivati. Ova biblioteka također nudi i upravljanje poveznicama upotrebom *Link* komponente. Ovu komponentu je moguće ugraditi bilo gdje unutar aplikacije te se može definirati koja putanja će se pozvati ukoliko korisnik odabere poveznicu upotrebom *to* atributa.

Kako bi aplikacija mogla obavljati određene funkcionalnosti, potrebno ju je povezati s poslužiteljem, odnosno potrebno je uspostaviti komunikaciju između poslužitelja i klijenta. Ovo je ostvareno upotrebom *Axios* HTTP klijenta. *Axios* omogućuje jednostavan način pozivanja asinkronih zahtjeva na poslužitelj, obradu odgovora i rukovanje iznimka. Asinkrona obrada zahtjeva postiže se upotrebom *async* ključne riječi prilikom definiranja metode. Ovo podrazumijeva da aplikacija ne mora čekati na odgovor poslužitelja kako bi nastavila s radom već se odgovor obrađuje na zasebnoj niti kada pristigne. Međutim, u većini slučajeva je ipak potrebno čekati odgovor kako bi znali koje informacije je potrebno prikazati nakon što ih poslužitelj vrati. Čekanje na odgovor je moguće postići upotrebom *await* ključne riječi prije poziva asinkrone funkcije. Sve navedeno nakon ove ključne riječi će se izvršiti tek kada aktualna nit završi sa svojom obradom. *Axios* omogućuje definiranje određene HTTP metode koju je potrebno pozvati, URL putanje do poslužitelja i putanje željene metode, zaglavlje i tijelo zahtjeva. Ukoliko poslužitelj vrati grešku, tu grešku je moguće uhvatiti s *catch* funkcijskim pozivom unutar kojeg je moguće definirati rutinu za obradu iznimke. Kako bi se ovi pozivi mogli ostvariti, na strani poslužitelja je potrebno definirati odgovarajuću CORS (engl. *Cross-Origin Resource Sharing*) politiku. To je metoda koja poslužitelju omogućuje da iz zaglavlja zahtjeva pročita odakle zahtjeva potječe i u ovisnosti o tome odluči hoće li dopustiti ili uskratiti tom klijentu uslugu. Ovo funkcionira na način

da se prije samog zahtjeva, prema serveru, pošalje zahtjev s OPTION HTTP metodom i ukoliko server vrati odgovor sa statusom 200 znači da je prepoznao klijenta te da će mu pružiti traženu uslugu te se na temelju toga može poslati i originalno željeni zahtjev.

```
1. const response = await axios({
2.         method: method,
3.         url: `${Environments.LOCAL}${path}`,
4.         data: body,
5.         headers: getHeader()
6.     }).catch(function () {
7.         return null;
8. });
```

Primjer 4.12. Uspostava *Axios* poziva

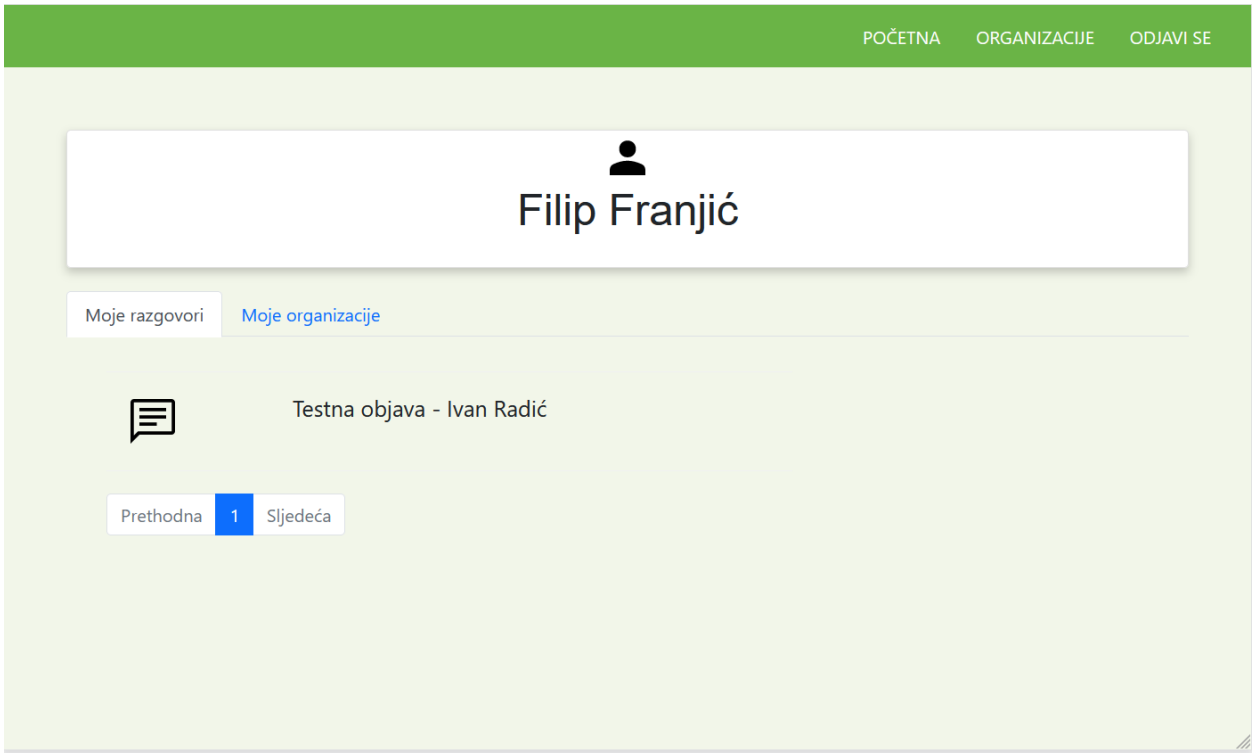
Često je u samo zaglavlju zahtjeva potrebno poslati neki identifikator po kojem će poslužitelj znati konkretno koji korisnik traži određenu uslugu, ima li pravo na nju i na temelju toga vratiti odgovarajući odgovor. Kako bi to bilo moguće, prilikom autentifikacije korisnike, token koji poslužitelj vrati sprema se među kolačiće (engl. *Cookies*) kako bi ga se moglo koristiti pri ostalim zahtjevima. Kolačići predstavljaju kratkotrajne podatke, male veličine koji su lokalno spremljeni na računalo korisnika. Oni imaju određeni životni vijek nakon koje se uništavaju i više nisu važeći. Za potrebu ovog rada, korištenja je *Universal-Cookie* biblioteka za rukovanje kolačićima. Kada korisnik zatraži određenu uslugu od poslužitelja, njegov token se dohvaća iz kolačića i postavlja u *Authorization* atribut zaglavlja. Iz ovog atributa, poslužitelj ima mogućnost rekonstruirati podatke o korisniku i provjeriti ima li pravo na traženo uslugu. Ukoliko se token ne nalazi među kolačićima, znači da niti jedan korisnik još nije prijavljen u sustav na tom računalu te ga se sukladno tome preusmjerava na stranicu za prijavu.

U ovome poglavlju, navedene su samo neke mogućnosti koje je moguće ostvariti uz *React* biblioteku no njih ima još dosta. *React* je moćan alat koji je znatno olakšao razvoj klijentske strane aplikacije uvodeći jasne koncepte koji su riješili problem sve više prisutnog špageti kôda pri razviju sa čistim JavaScript jezikom, a čak i uz upotrebu *jQuery* biblioteke. Budući da je otvorenog kôda, svatko može pridonijeti njegovom razvoju pa stoga postoje i razne druge biblioteke koje, uz *React*, značajno pojednostavljuju koncept razvoja. Neke od njih bi bile *React-Paginate* koji omogućuje straničenje (engl. *Pagination*), *React-Hot-Toast* koji omogućuje prikaz nestajućih poruke kako bi interakcija sa sustavom bila intuitivnija i mnogi drugi. Također, uz *React*, je potrebno koristiti i određeni sustav pakiranja programske podrške kao što su *npm* ili *yarn*. Ovi sustavi olakšavaju dijeljene kôda s drugim programeri kroz pakete. Paketi se sastoje od cjelokupnog kôda koji se dijeli uz JSON datoteku koja se naziva manifest i koja opisuje paket.

Unutar manifesta nalazi se popis svih korištenih biblioteka, njihove verzije, različite skripte za pokretanje projekta, a glavna svrha mu je usklađivanje svih korištenih tehnologija za sve programere kako ne bi dolazilo do sukoba između verzija.

4.3. Korištenje aplikacije

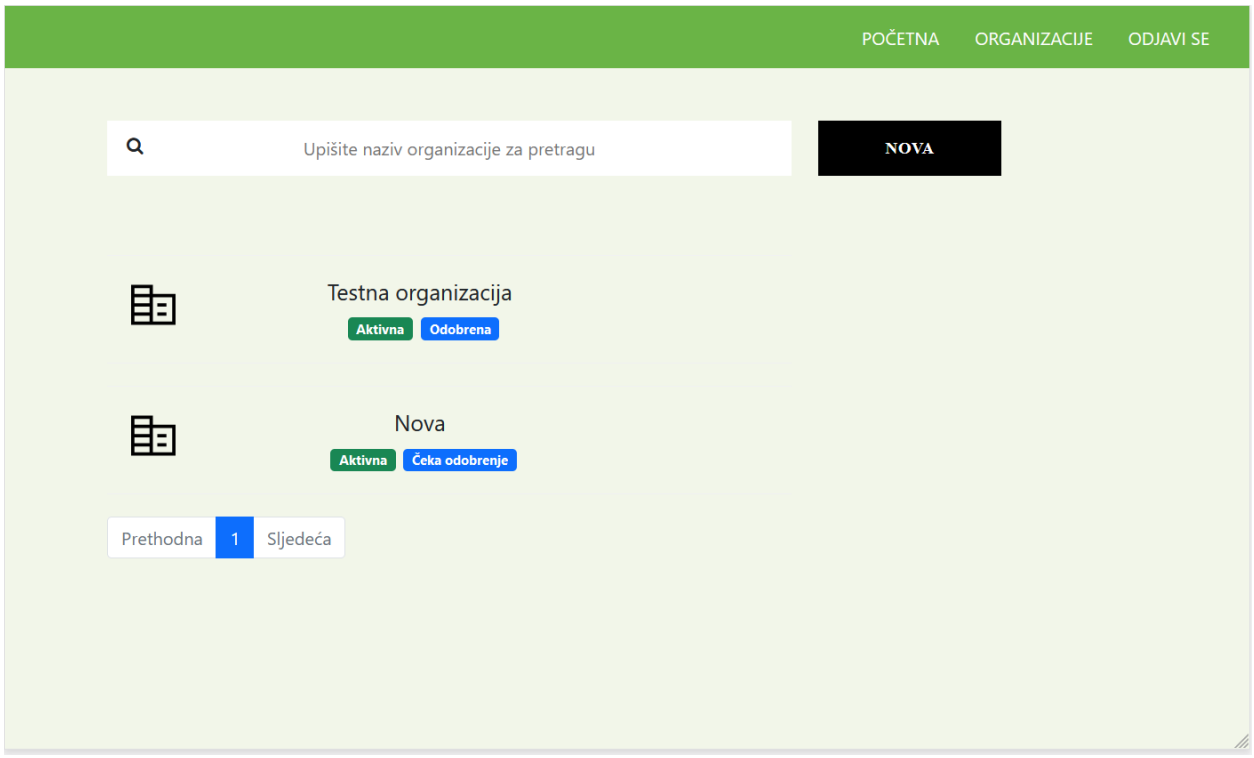
Zaglavlje aplikacije sastoji se od navigacijske trake koja nudi mogućnost prikaza početnog zaslona, popisa svih aktivnih organizacija te odjavu iz sustava. Administratoru sustavu, dostupna je i administracijska stranica. Kada se član organizacije prijavi u aplikaciju, na početnom zaslonu vidi svoje informacije te navigacijsku traku s pregledom razgovora organizacije i studenata, vezanih uz neku objavu, i popis svih organizacija u kojima je član. Radi preglednosti prikaza, na karticama s razgovorima i organizacijama, broj stavki ograničen je na četiri stavke po stranici. Radi toga, omogućena je navigacija po stranicama koja od poslužitelja zahtjeva određene stavke ovisno o poslanoj stranici.



Slika 4.1. Prikaz profila korisnika

Ukoliko korisnik nije član niti jedne organizacije, dostupna je stranica sa svim organizacijama u sustavu. Na ovoj stranici, moguće je vidjeti informacije o nazivu organizacije, statusu njene aktivnosti i statusu odobrenja. Članovima organizacije, vidljive su samo aktivne organizacije, dok su administratoru sustava vidljive i neaktivne. Omogućeno je pretraživanje organizacije po nazivu. Pretraživanje radi na način da se vraćaju organizacije, koje zadovoljavaju poslani naziv, za vrijeme

dok se naziv upisuje što rezultira preglednim i dinamičnim sustavom pretraživanja. Ukoliko korisnik na popisu ne može pronaći svoju organizaciju, postoji mogućnost registriranja nove organizacije. Nakon što ju registrira, korisnik odmah postaje njezin administrator te može pristupni kôd poslati ostalim članovima kako bi se priključili. Sve nove organizacije trebaju biti odobrene od sistemskog administratora kako bi objave bile vidljive studentima, međutim objave je moguće stvarati i dok organizacija još nije odobrena.

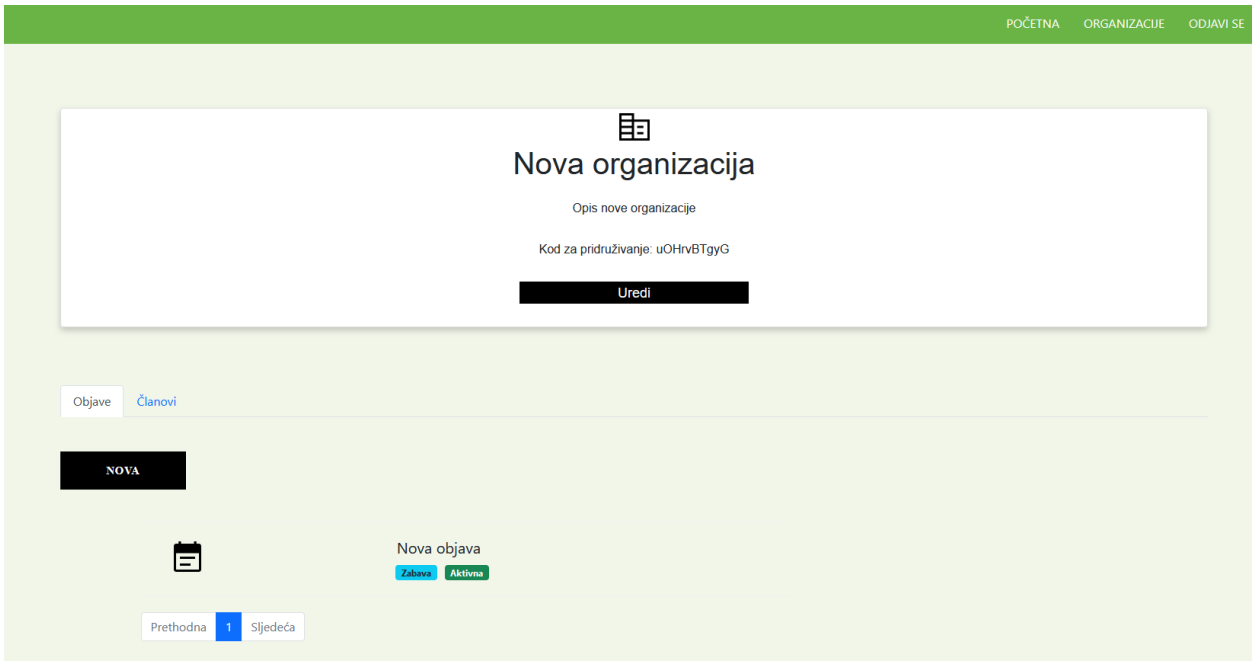


Slika 4.2. Pregled organizacija

Klikom na gumb „Nova“, na slici 4.2., otvara se modal unutar kojeg je potrebno unijeti naziv i opis organizacije kako bi ona bila registrirana u sustav. Odabirom jedne od organizacija, otvara se pregled profila organizacije.

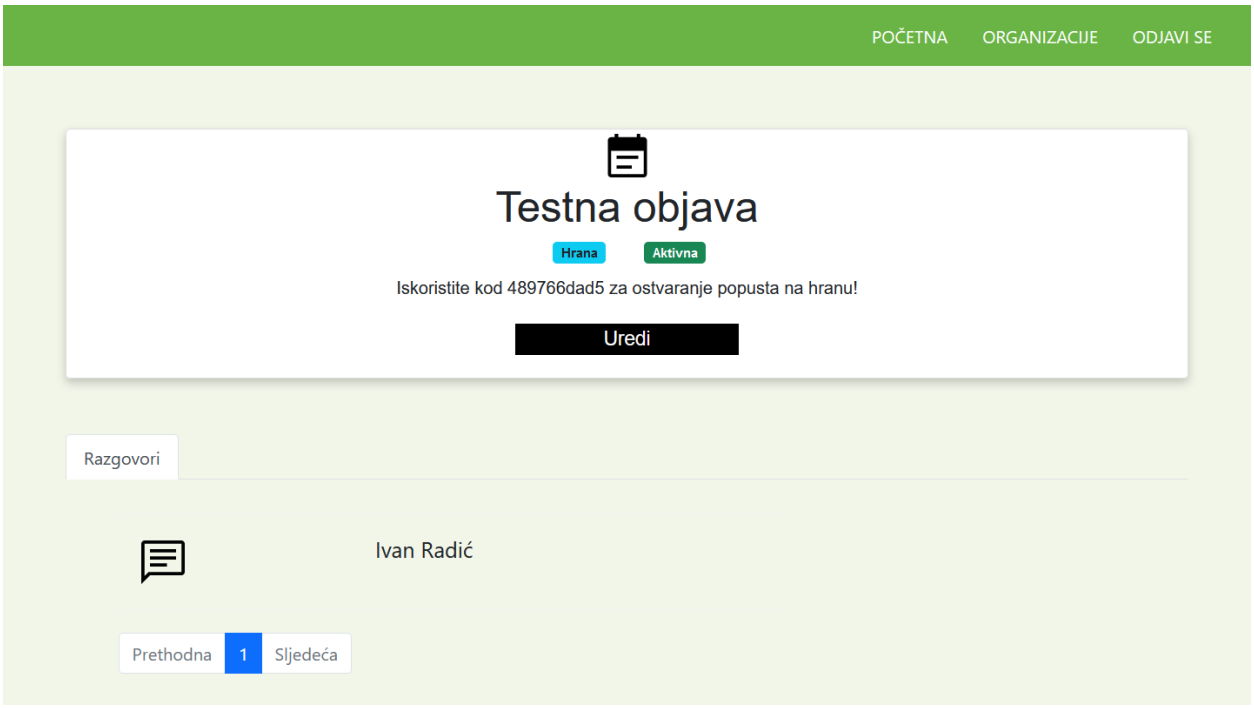
Kada korisnik otvori organizaciju, na novom zaslonu su mu dostupne informacije o toj organizaciji. Osim informacija, dostupan je i kartični prikaz unutar kojeg se nalaze objave i članovi organizacije. Ukoliko administrator organizacije pregledava organizaciju, osim osnovnih informacija, dostupan mu je i kôd za pridruživanje koji služi kako bi se i ostali članovi mogli uključiti. Kôd za pridruživanje generira se automatski od strane sustava prilikom registriranja organizacije. Na kartici „Objave“ nalaze se sve objave organizacije s informacijama o nazivi, statusu aktivnosti i kategoriji u kojoj se nalaze. Članovi organizacije ima mogućnost stvarati novi objavu klikom na odgovarajući gumb. Ovo će otvoriti modal unutar kojeg je moguće upisati naslov objave, njezin opis i odabrati kategoriju. Kategorije se nalaze u padajućem izborniku koji je

popunjen svim aktivnim kategorijama sustava. Također, kategorije je moguće pretraživati po nazivu što će filtrirati opcije unutar padajućeg izbornika. Klikom na jednu od objava otvara se njezin detaljniji prikaz. Kartica „Članovi“ sadrži pregled svih članova organizacije s njihovim imenom, prezimenom i ulogom koju imaju unutar organizacije. Klikom na gumb „Uredi“, otvara se novi modal s formom koja omogućuje uređivanje informacija o organizaciji. Unutar ovog modala, sistemski administrator, ima mogućnost i deaktivirati organizaciju. Deaktivacijom organizacije, automatski i sve njene objave postaje neaktivne te više nisu dostupne studentima za pregled. Sistemski administrator, na pregledu organizacije, također ima mogućnost i njenog odobravanja ukoliko još nije odobrena.



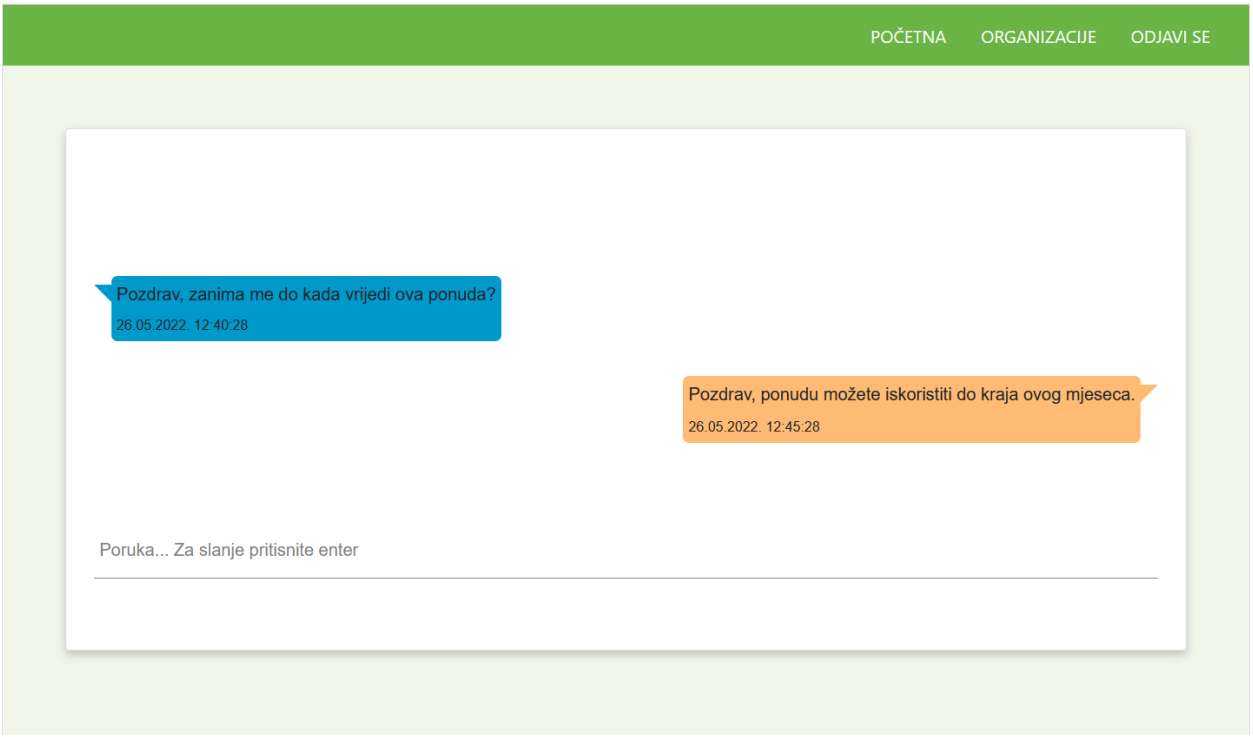
Slika 4.3. Profil organizacije

Na prikazu objave, nalaze se osnovne informacije o objavi kao što su njezin naziv, opis, kategorija i status aktivnosti. Također, ovdje se nalazi i prikaz svih razgovora između studenata i organizacije koji su vezani za tu objavu. Klikom na razgovor, otvara se pregled svih poruka unutar tog razgovora. Odabranu objavu je moguće uređivati klikom na odgovarajući gumb koji otvara novi modal za uređivanje. Unutar toga modala, moguće je izmijeniti naziv, opis i kategoriju objave. Također, unutar istog modala, moguće je deaktivirati objavu, čime ona postaje skrivena od studenata, a također ju je moguće i ponovno aktivirati.



Slika 4.4. Prikaz detalja objave

Na pregledu razgovora, nalazi se prikaz svih poruka poredan prema datumu stvaranja na način da je najnovija poruka uvijek prva prikazana. Poruke koje je poslao student, nalaze se na lijevoj strani okvira razgovora, dok se poruke organizacije nalaze na desnoj strani. Ispod same poruke, nalazi se datum i vrijeme njenog kreiranja. Budući da za prikaz poruka ne bi odgovara pregled po stranicama, korišten je pregled uz pomoć sadržaja koji se učitava po potrebi. Drugim riječima, moguće je učitavati nove poruke dok se ne dođe do kraja, a po završetku učitavanja, sve poruke su dostupne za pregled. Za implementaciju ovakvog sustava razmjena poruka, korišten je *WebSocket* što omogućuje da se poruke prikazuju u stvarnom vremenu njihovog kreiranja. Odnosno, nije potrebno osvježiti stranicu kako bi novo pristigle poruke bile prikazane korisniku.



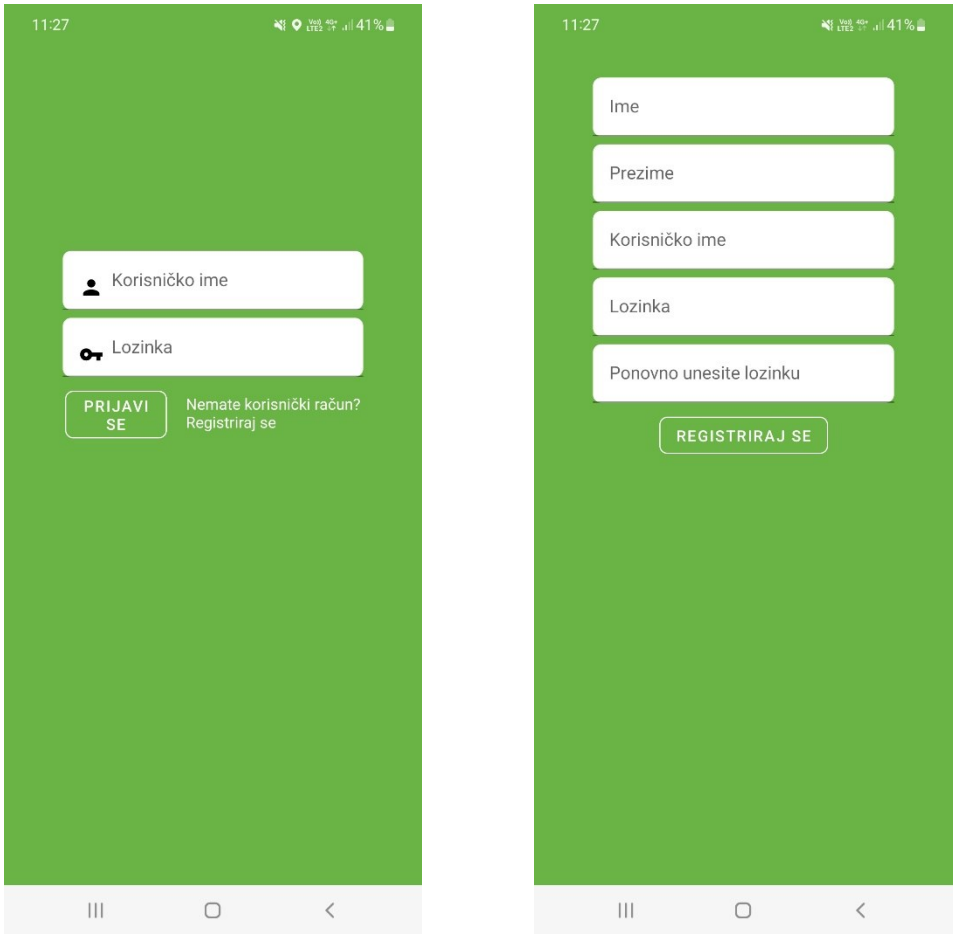
Slika 4.5. Razgovor između studenta i organizacije

Kao što je već ranije rečeno, administratoru sustava, omogućeno je uređivanje postojećih i stvaranje novih kategorija te pregled organizacija koje čekaju odobravanje unutar administracijskih alata sustava. Prilikom stvaranja nove kategorije, potrebno je unijeti njezin naziv, a prilikom uređivanja, osim naziva, moguće je odabranu kategoriju aktivirati ili deaktivirati. Deaktivacijom kategorije, deaktiviraju se i sve objave koje se nalaze unutar odabrane kategorije te one više nisu vidljive studentima. Prilikom pregleda organizacije koje čekaju odobravanje, moguće je otvoriti njihov detaljan pregled unutar kojeg se nalazi gumb za odobravanje.

Slika 4.6. Administracijski alati sustava

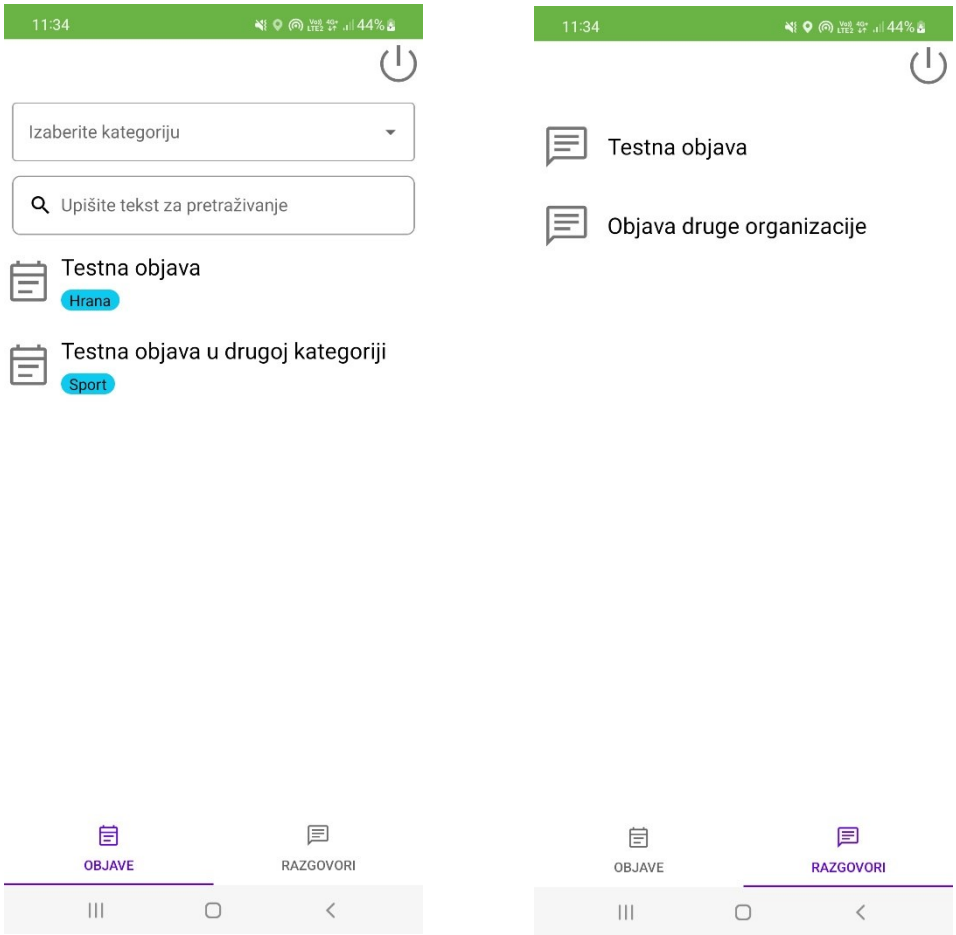
5. ANDROID APLIKACIJA

Android aplikacija namijenjena je studentima kako bi na brzi i jednostavan način mogli pregledavati objave organizacija i pronaći onu koja ih zanima. Svaki korisnik, koji se registrira putem android aplikacije, automatski poprima ulogu studenta. Aplikacije se sastoji od dva glavna zaslona, zaslon za autentifikaciju i glavni zaslon aplikacije, koji različiti sadržaj prikazuju uz pomoć fragmenata. Zaslon za autentifikaciju se sastoji od fragmenta za prijavu i registraciju te je to prvi zaslon koji se prikazuje prilikom otvaranja aplikacije. Ukoliko student već posjeduje račun, može se prijaviti u sustav upisujući svoje korisničko ime i lozinku. Ovi podaci su obavezni te ukoliko su izostavljeni, ili su uneseni pogrešno, prikazuje se odgovarajuća poruka. Ukoliko student još ne posjeduje korisnički račun, može se prebaciti na fragment za registraciju koji od studenta zahtjeva popunjavanje podataka kao što su ime, prezime, korisničko ime, lozinka i ponovljena lozinka kako bi se osigurala ispravnost unosa. Nakon prijave ili registracije, studentu se dopušta pristup sustavu uz pohranjivanje generiranog tokena u lokalnu memoriju aplikacije. Ovime se osigurava da se student ne mora ponovno prijaviti svaki put kada otvori aplikaciju.



Slika 5.1. Android sučelje za prijavu i registraciju

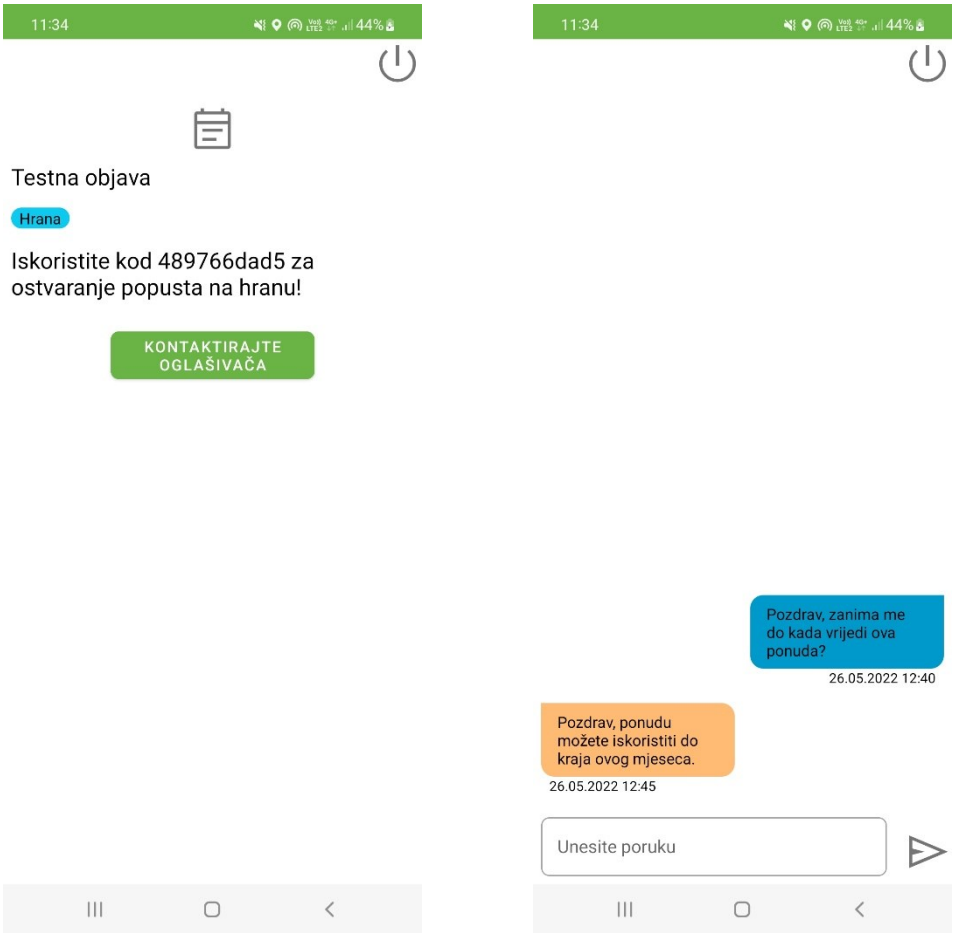
Nakon prijave, studentu se otvara glavni zaslon aplikacije koji sadrži kartični fragment unutar kojeg svaka od kartica predstavlja još jedan fragment. Prva kartica sastoji se od prikaza svih aktivnih objava. Ove objave smještene su unutar *Recycler View* elementa kako bi se poboljšalo korisničko iskustvo prolaska kroz elemente. Na ovome zaslonu, studentu su dostupne informacije o nazivu objave i kategoriji kojoj pripada. Također, studentu je omogućeno pretraživanje objava prema nazivu te filtriranje prema kategoriji. Pretraživanje po nazivu funkcionira na način da se rezultat filtrira za vrijeme upisivanja. Upisani naziv, može se obrisati jednim klikom što vraća originalni prikaz elemenata. Filtriranje po kategoriji realizirano je pomoću padajućeg izbornika unutar kojeg se nalaze sve aktivne kategorije sustava. Odabirom objave, otvara se njezin detaljni prikaz o kojem će više biti rečeno u nastavku. Druga kartica glavnog zaslona sadrži sve razgovore studenta vezane uz objave. Ovi razgovori su također smješteni unutar *Recycler View* elementa radi preglednijeg prikaza. Svaki element sastoji se od naziva objave uz koju je taj razgovor vezan. Odabirom razgovora, otvara se prikaz svih poruka.



Slika 5.2. Kartični prikaz objava i razgovora

Na slici 5.2., vidljivo je kako je aktivna kartica naglašenija što poboljšava korisničko iskustvo prilikom upotrebe aplikacije.

Detalji pojedine objave smješteni su unutar novog fragmenta koji se otvara klikom na jednu od objava. Ovaj fragment sadrži informacije o nazivu, kategoriji i opisu objave. Također, ovdje se nalazi i gumb koji omogućuje kontaktiranje oglašivača u vezi odabrane objave što otvara razgovor. Ukoliko razgovor već sadrži poruke, one se prikazuju studentu. Student može zatvoriti razgovor bez slanja poruke što neće utjecati na kreiranje novog razgovora. Tek kada student pošalje prvu poruku, razgovor se kreira te također postaje dostupan na popisu svih razgovora vidljivom na slici 5.2. Poruke razgovora, smještene su unutar obrnutog *Recycler View* što omogućuje prikaz najnovije poruke prve. Poruka, osim samog sadržaja, sadrži i vrijeme kreiranja. Poruke poslane od strane studenta, smještene su na desnoj strani zaslona, dok su poruke organizacije smještene na lijevoj strani što predstavlja klasičan raspored poruka pošiljatelja i primatelja. Kao i web dio, fragment koji sadrži razgovor studenta i organizacije, spojen je na *WebSocket* što omogućuje prikaz poruka u stvarnom vremenu kako one pristižu. Ovo omogućuje tečnu komunikaciju budući da nije potrebno osvježiti prikaz kako bi se prikazale nove poruke.



Slika 5.3. Prikaz detalja objave i razgovora studenta s organizacijom

5.1. Fragmenti

Fragment predstavlja ponovno upotrebljivi dio korisničkog sučelja. On definira i upravlja vlastitim rasporedom elemenata, ima vlastiti životni ciklus i može upravljati vlastitim događajima. Fragment ne može postojati kao samostalna cjelina već mora biti vezan za određeni zaslon ili drugi fragment [11]. Za potrebu ovog rada, uz klasični fragment, korišten je i kartični raspored fragmenata.

Kako bi se fragmenti mogli koristiti, na zaslon unutar kojeg će se prikazivati, potrebno je stvoriti okvir unutar kojeg će fragment biti smješten. Zatim je potrebno definirati XML (engl. *Extensible Markup Language*) datoteku koja će predstavljati raspored elemenata na fragmentu. Budući da fragmenti predstavljaju objekte klasa, potrebno je stvoriti i Kotlin klasu fragmenta koja nasljeđuje Fragment klasu. Iz ove klase, moguće je prepisati metodu koja će, iz definirane XML datoteke, „napuhati“ sučelje. Ukoliko je odlučeno koristiti vezivanje pogleda (engl. *View Binding*), ovo je moguće napraviti pozivanjem *inflate* metode na argumentu koji predstavlja taj pogled. Ovaj argument koristi Kotlin mogućnost kasne inicijalizacije pa se sukladno tome može inicijalizirati za vrijeme kreiranje pogleda fragmenta. Također, prilikom definicije fragmenta, moguće je koristiti *companion object*, dio Kotlin programskog jezika, koji će zatim pozvati metodu za kreiranje fragmenta. Ovaj tip objekta zamjenjuje statične metode koje su bile dostupne u Java programskom jeziku što znači da je metode definirane unutar toga bloka, moguće pozvati i bez kreiranja konkretnog objekta klase. Ovo je korisno prilikom kreiranja transakcije koja omogućuje izmjene među fragmentima, budući da nije potrebno pozivati konstruktor fragmenta već se inicijalizacija vrši pozivom statične metode. Svaka transakcije mora se završiti pozivom *commit* metode.

```
1. class ChatFragment : Fragment() {
2.
3.     private lateinit var chatFragmentBinding: FragmentChatBinding
4.
5.     override fun onCreateView(
6.         inflater: LayoutInflater, container: ViewGroup?,
7.         savedInstanceState: Bundle?
8.     ): View {
9.         chatFragmentBinding = FragmentChatBinding.inflate(
10.             inflater,
11.             container, false
12.         )
13.
14.         return chatFragmentBinding.root
15.     }
16.
17.     companion object {
18.         const val TAG = "Chats"
19.         fun create(): ChatFragment {
20.             return ChatFragment()
21.         }
22.     }
23. }
```

Primjer 5.1. Kreiranje jednostavne fragment klase

Realizacija kartičnog pregleda fragmenta zahtjeva još dodatne postavke. Unutar XML datoteke potrebno je definirati spremnik fragmenta upotrebom *ViewPager* elementa te raspored kartica upotrebom *TabLayout* elementa. Potrebno je kreirati i adapter klasu unutar koje se nalazi informacija o broju kartica te metode za kreiranje fragmenta i dodavanje na odgovarajuće mjesto među karticama.

```
1. class PageAdapter(fragmentActivity: FragmentActivity) :
2.     FragmentStateAdapter(fragmentActivity) {
3.
4.     private val fragments = ArrayList<TabFragmentEnum>()
5.
6.     override fun getItemCount(): Int = 2
7.
8.     override fun createFragment(position: Int): Fragment {
9.         return fragments[position].fragment
10.     }
11.
12.     fun addFragment(fragment: TabFragmentEnum) {
13.         fragments.add(fragment)
14.     }
15. }
```

Primjer 5.2. Adapter za kartični fragment

Ostatak inicijalizacije se obavlja slično kao i za obične fragmente uz dodatak prepisivanja metode koja nakon kreiranja pogleda dodaje sve potrebne fragmente u adapter i veže odgovarajuće kartice uz fragment.

```
1. override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
2.
3.         val pageAdapter = PageAdapter(requireActivity())
4.         pageAdapter.addFragment(TabFragmentEnum.POSTS)
5.         pageAdapter.addFragment(TabFragmentEnum.CHAT)
6.
7.         viewPager = tabFragmentBinding.viewPager
8.         viewPager.adapter = pageAdapter
9.
10.          val tabLayout = tabFragmentBinding.tabLayout
11.          TabLayoutMediator(tabLayout, viewPager) { tab, position ->
12.              TabFragmentEnum.getByPosition(position)?.let {
13.                  tab.text = it.title
14.                  tab.setIcon(it.icon)
15.              }
16.          }.attach()
17.      }
```

Primjer 5.3. Metoda koja vrši povezivanje kartica s fragmentom

5.2. Recycler View

Svrha *RecyclerView* biblioteke je pružiti efikasni prikaz velikih kolekcija podataka. Programer samo treba pružiti podatke i definirati na koji način će se svaki od podataka prikazivati, a biblioteka će ih dinamički stvoriti kada budu potrebni. Ona, kako i samo ime nalaže, reciklira pojedinačne elemente. Kada element nestane sa zaslona, ne uništava se pogled već se on ponovno koristi za nove elemente koji su pristigli na zaslon [12]. Ovaj pristup uvelike poboljšava performanse same aplikacije u odnosu na dotadašnji *ListView*. Još jedna prednost ovakvog rasporeda je ta što omogućuje i horizontalno i vertikalno pomicanje sadržaja što prije nije bilo moguće. Iako implementacija ovog elementa zahtjeva nešto više posla, dugoročno gledano taj posao se isplati uzimajući u obzir sve prednosti koje sa sobom donosi.

Implementacija ovog sustava prikaza sadržaja kreće s definiranjem XML datoteke koja predstavlja raspored unutar pojedinačnog elementa te postavljanje kompletnog *RecyclerView* elementa na zaslon ili fragment unutar kojeg ga je potrebno prikazati. Potrebno je definirati i klasu koja predstavlja odgovarajući držač pogleda (engl. *View Holder*) unutar koje je moguće postaviti prikaz za pojedini element kao i vezati određene događaje poput odabira tog elementa.

```
1. class PostViewHolder(itemView: View) :
2.     RecyclerView.ViewHolder(itemView) {
3.
4.     fun bind(post: PostDTO, postItemClickListener: PostItemClickListe
   ner) {
5.
6.         val itemBinding = ItemPostBinding.bind(itemView)
7.         itemBinding.tvPost.text = post.title
8.         itemBinding.tvCategory.text = post.category.name
9.
10.          itemView.setOnClickListener { postItemClickListener.onPostC
   licked(post) }
11.      }
12.  }
```

Primjer 5.4. Držač pogleda elementa

Nakon toga, potrebno je definirati klasu koja nasljeđuje adapter iz ove biblioteke. Unutar te klase, nalaze se podaci o elementima, njihovoj veličini te metode koje služe za stvaranje određenog držača pogleda te njegovo povezivanje s elementima. Ovakvu klasu, potrebno je postaviti kao adapter *RecyclerView* elementa definiranog u XML datoteci zaslona ili fragmenta.

```
1. class PostAdapter(
2.     posts: List<PostDTO>,
3.     private val postItemClickListener: PostItemClickListener
4. ) : RecyclerView.Adapter<PostViewHolder>() {
5.
6.     private val posts: MutableList<PostDTO> = mutableListOf()
7.
8.     override fun onCreateViewHolder(parent: ViewGroup, viewType: Int)
   : PostViewHolder {
9.         val view = LayoutInflater.from(parent.context)
10.             .inflate(R.layout.item_post, parent, false)
11.
12.         return PostViewHolder(view)
13.     }
14.
15.     override fun onBindViewHolder(holder: PostViewHolder, position:
    Int) {
16.         val post = posts[position]
17.         holder.bind(post, postItemClickListener)
18.     }
19.
20.     override fun getItemCount(): Int {
21.         return posts.size
22.     }
23.  }
```

Primjer 5.5. Jednostavni *RecyclerView* adapter

Adapter se može proširiti dodavajući metodu za osvježavanje podataka koja će zamijeniti postojeće podatke unutar elementa.

5.3. Retrofit 2

Retrofit predstavlja jednostavan REST (engl. *Representational State Transfer*) klijent koji omogućuje komunikaciju s poslužiteljem iz Android aplikacije. Ovaj REST klijent koristi oblikovni obrazac graditelja kako bi izgradio objekt koji se zatim može koristiti prilikom komunikacije. Unutar toga objekta, definira se osnovna putanja do poslužitelja te pravila koja će se koristiti prilikom serijalizacije podataka u neki od transportnih jezika kao što je JSON. Također, potrebno je definirati i maksimalno vrijeme koje će se utrošiti na čekanje odgovora od poslužitelja prije nego se zahtjev proglasi neuspješnim. Kako bi se pojednostavio poziv odgovarajućih servisa, moguće je definirati metodu koja će obavljati radnje karakteristične za sve servise, poput postavljanja autorizacijskog tokena u zaglavlje zahtjeva i slično. Nakon inicijalnih postavki klijenta, potrebno je definirati odgovarajuća sučelja koja predstavljaju odgovarajuće pozive. Unutar tih sučelja, moguće je metodi dodijeliti odgovarajuću HTTP reprezentaciju, definirati odgovarajuće parametre upita, definirati tijelo zahtjeva te na kraju povratni tip koji predstavlja odgovor poslužitelja na zahtjev.

```
1. interface PostService {
2.
3.     @GET("posts")
4.     fun getPosts(
5.         @Query("categoryId") categoryId: String?,
6.         @Query("title") title: String?
7.     ): Call<ResponseMessage<Page<PostDTO>>>
8.
9.     @GET("posts/{id}")
10.      fun getPost(
11.          @Path("id") postId: String
12.      ): Call<ResponseMessage<PostDTO>>
13.  }
```

Primjer 5.6. Sučelje *Retrofit* servisa

Na primjeru 5.6. vidljivo je kako je, uz putanju, servisa moguće pružiti odgovarajuće parametre upotrebom *Query* anotacije. Također, moguće je dinamički nadovezati varijablu na samu putanju upotrebom *Path* anotacije. Ovaj slučaj se najčešće koristi prilikom nadovezivanja jedinstvenog identifikatora elementa kojeg se od poslužitelja zahtjeva.

Nakon definiranja servisa, njegove metode moguće je pozivati unutar *Retrofit* klijenta. Uz sam poziv metode, potrebno je prepisati odgovarajuća ponašanja u slučaju uspješne i neuspješne komunikacije s poslužiteljem kako ne bi došlo do ispada aplikacije. Metode neuspješne komunikacije poziva se kada poslužitelj nije bio u mogućnosti obraditi zahtjev u definiranom vremenu obrade, ukoliko nije dopušteno spajanje na Internet i slično.

```
1. val apiCall =
2.             RestUtil.createService(
3.                 PostService::class.java,
4.                 PreferenceManager.getPreference(PreferenceEnum.TOKEN)
5.             ).getPosts(categoryFilter.value, titleFilter.value)
6.
7.         apiCall.enqueue(object : Callback<ResponseMessage<Page<PostDT
   O>>> {
8.             override fun onResponse(
9.                 call: Call<ResponseMessage<Page<PostDTO>>>,
10.                  response: Response<ResponseMessage<Page<PostDTO>>>
11.             ) {
12.             }
13.
14.             override fun onFailure(
15.                 call: Call<ResponseMessage<Page<PostDTO>>>,
16.                 t: Throwable
17.             ) {
18.             }
19.         })
```

Primjer 5.7. Poziv odgovarajućeg servisa

Važno je napomenuti da unutar manifest datoteke mora biti omogućen pristup internetu kako bi se *Retrofit* klijent mogao koristiti.

6. ZAKLJUČAK

Glavna svrha ovoga rada je pružiti studentima i organizacijama jedinstveni način na koji se mogu povezati i vršiti komunikaciju. Organizacijama je dostupna web aplikacija unutar koje mogu skupiti sve svoje pogodnosti koje žele prezentirati studentima. Studentima je na raspolaganje dana Android aplikacija unutar koje su prikupljene sve te objave kako bi ih na lakši, brži i efikasniji način mogli pretraživati budući da su objedinjene na jednome mjestu.

Prije samog prikaza funkcionalnosti razvijenih aplikacija, pružen je uvid korištenih tehnologija uz kratki opis svake. Prilikom opisivanja realizacije poslužiteljskog i klijentskog dijela, pruženi su primjeri kôda iz same aplikacije što omogućuje lakše vizualizaciju tijeka razvoja ovakvog portala. Tehnologije su podijeljene na tri grupe. Prva grupa predstavlja tehnologije na strani poslužitelja koje se sastoje od Java programskog jezika i *Spring* razvojnog okvira. Klijentska strana aplikacije razvijana je uz *React*, JavaScript, biblioteku te Bootstrap5 i *Saas* za kreiranje korisničkog sučelja. Android aplikacija razvijena je u Kotlin programskoj jeziku. Kako bi studenti i organizacije mogle nesmetano komunicirati, implementirana je komunikacije putem *WebSocket* sustava što omogućuje razmjenu poruka u stvarnom vremenu.

Ovaj rad posjeduje i nekoliko smjerova za unapređenje. Prvi takav smjer bio bi integracija s AAI sustavom kako se od studenata ne bi morala zahtijevati registracije te kako bi se mogli prijaviti sa svojim sveučilišnim računom. Nadalje, uz Android aplikaciju, može se razviti i iOS aplikacija, kako bi usluga bila dostupna i njezinim korisnicima. Organizacijama bi se mogao pružiti sustav odobravanja objava od strane organizacijskog administratora te kompleksnije upravljanje korisničkim računima. Također bi bilo korisno implementirati datotečni sustav kako bi se uz objavu mogla priložiti i neka datoteka što bi uvelike poboljšalo kvalitetu usluge.

Za kraj, potrebno je navesti kako je ovaj portal namijenjen da funkcionira kao cjelina, iako je razvijena kroz nekoliko samostalnih komponenti.

LITERATURA

[1] Kolegio.hr web stranica, https://kolegio.hr/, [Datum zadnje posjete 28.06.2022.]

[2] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, Addison-Wesley, Boston, San Francisco, New York, Toronto, Montreal, London, Munich, Paris, Madrid, Capetown, Sydney, Tokyo, Singapore, Mexico City, 2005.

[3] R. Johnson, J. Hoeller, A. Arendsen, T. Risberg, C. Sampaleanu, Professional Java Development with the Spring Framework, Wiley Publishing, Inc, Indianapolis, 2005.

[4] D. Goodman, JavaScript Bible, IDG Books Worldwide, Inc., California, 1998.

[5] G. Sidelnikov, React.js Book: Learning React JavaScript Library From Scratch, Independently published, 2017.

[6] R. Wieruch, Taming the State in React, Lean Publishing, 2017.

[7] S. Gantra, React Router Quick Start Guide, Packt Publishing Ltd., UK, 2018.

[8] Bootsrap5 dokumentacija, https://getbootstrap.com/docs/5.0/getting-started/introduction/, [Datum zadnje posjete 17.06.2022.]

[9] Saas dokumentacija, https://sass-lang.com/guide, [Datum zadnje posjete 28.06.2022.]

[10] M. Moskal, I. Wojda, Android Development with Kotlin, Packt Publishing Ltd., UK, 2017.

[11] Fragment dokumentacija, https://developer.android.com/guide/fragments, [Datum zadnje posjete 28.06.2022.]

[12] RecyclerView dokumentacija, https://developer.android.com/guide/topics/ui/layout/recyclerview, [Datum zadnje posjete 28.06.2022.]

[13] STOMP dokumentacija, https://stomp.github.io/stomp-specification-1.2.html#Abstract, [Datum zadnje posjete 28.06.2022.]

[14] Android STOMP biblioteka, https://github.com/NaikSoftware/StompProtocolAndroid, [Datum zadnje posjete 28.06.2022.]

SAŽETAK

Ovaj rad ima za svrhu stvoriti jedinstveni portal kojem je glavni cilj povezivanja studenata s organizacijama u svrhu jednostavnog i brzog pronalaženja ponuda koje zadovoljavaju njihove potrebe. Prvi dio rada u glavni fokus stavlja tehnologije korištene za realizaciju ovog portala. Uz svaku tehnologiju, pružen je detaljan opis njezinog korištena potkrijepljen primjerima iz samog rada što praćenje samog procesa razvoja čini jednostavnijim. Nakon opisa svake tehnologije, pružen je uvid u sami rezultat portala potkrijepljen slikama i uputama za korištenje. Registracijom, pripadnici organizacije, dobivaju mogućnost pridruživanja organizaciji te stvaranje objava u ime te organizacije. Studenti na raspolaganju imaju Android aplikaciju kroz koju mogu pregledavati objave i kontaktirati organizaciju ukoliko ima nekih nejasnoća oko same objave.

Ključne riječi: Android, oglašavanje, React, Spring razvojni okvir, WebSocket

ABSTRACT

Student portal for advertising offers using Spring framework

This thesis aims to create a unique portal which main purpose is to connect students and organizations and to allow quick and straightforward way of finding offers which would satisfy their needs. First part of this thesis in the focus puts used technologies. Along with each technology, this thesis provides a detailed description of its use along with examples from project which allows the easy following of development process. Furthermore, this thesis provides the result of the portal along with pictures and instructions of how to use it. Organization members, after registration, get access to their organization from which they can create posts. Students can use Android application to access these posts and to contact organization if they have any questions regarding the specific post.

Keywords: Android, advertising, React, Spring framework, WebSocket

ŽIVOTOPIS

Marko Cecelja rođen je 05. srpnja 1998. u Našicama, Republika Hrvatska. Nakon završetka osnovne škole Matije Petra Katančića u Valpovu, upisao je i završio Elektrotehničku i prometnu školu u Osijeku. Po završetku srednje škole, upisao je preddiplomski studij računarstva na fakultetu Elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. 2020. godine završava preddiplomski studij računarstva, te upisuje diplomski studij računarstva, smjer Programsko inženjerstvo na fakultetu Elektrotehnike, računarstva i informacijskih tehnologija u Osijeku. 2019. godine zapošljava se na studentskoj poziciji programskog inženjera unutar Lamaro Digital poduzeća koje je dio Sedam IT grupe.

_____
Potpis autora