Usporedba React i Angular radnih okvira za razvoj web aplikacija

Dumičić, Nino

Undergraduate thesis / Završni rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka, Faculty of Engineering / Sveučilište u Rijeci, Tehnički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:190:680894

Rights / Prava: Attribution 4.0 International/Imenovanje 4.0 međunarodna

Download date / Datum preuzimanja: **2024-05-17**

Repository / Repozitorij:

Repository of the University of Rijeka, Faculty of Engineering

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Preddiplomski sveučilišni studij računarstva

Završni rad

Usporedba React i Angular radnih okvira za razvoj web aplikacija

Rijeka, srpanj 2022.

Nino Dumičić
0069085361

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Preddiplomski sveučilišni studij računarstva

Završni rad

Usporedba React i Angular radnih okvira za razvoj web aplikacija

Mentor: doc. dr. sc. Marko Gulić

Rijeka, srpanj 2022.

Nino Dumičić
0069085361

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
POVJERENSTVO ZA ZAVRŠNE ISPITE

Rijeka, 7. ožujka 2022.

Zavod: **Zavod za računarstvo**
Predmet: **Razvoj web aplikacija**
Grana: **2.09.06 programsko inženjerstvo**

ZADATAK ZA ZAVRŠNI RAD

Pristupnik: **Nino Dumičić (0069085361)**
Studij: Preddiplomski sveučilišni studij računarstva

Zadatak: **Usporedba React i Angular radnih okvira za razvoj web aplikacija / The comparison of React and Angular frameworks for web application development**

Opis zadatka:

Treba napraviti detaljnu usporedbu React i Angular radnih okvira klijentske strane koji se koriste za razvoj web aplikacija. Usporedba će se izvršiti na način da se razviju dvije identične RESTful jednostranične aplikacije (Single Page Application, SPA) pomoću zadanih radnih okvira koje će u sebi sadržavati osnovne CRUD (create, read, update, delete) operacije i autentifikaciju. Treba analizirati mogućnosti, arhitekturu i načine razvoja web aplikacija pomoću ova dva radna okvira. Za razvoj poslužiteljskog dijela web aplikacije treba koristiti Spring Boot radni okvir uz proizvoljno odabran sustav za upravljanje bazama podataka. Također, treba koristiti paket Spring Security za realizaciju autentifikacije s klijentskom stranom (JWT token).

Rad mora biti napisan prema Uputama za pisanje diplomskih / završnih radova koje su objavljene na mrežnim stranicama studija.

Zadatak uručen pristupniku: 21. ožujka 2022.

Mentor:

Doc. dr. sc. Marko Gulić

Predsjednik povjerenstva za završni ispit:

Prof. dr. sc. Kristijan Lenac

Izjava o samostalnoj izradi rada

Izjavljujem da sam samostalno izradio ovaj rad.

Rijeka, srpanj 2022.

_____
Ime Prezime

Zahvala

Zahvaljujem se majci i ocu, prijateljima, mentoru, curi i Bogu na podršci tijekom pisanja ovoga rada i korisnim raspravama i savjetima. Zahvaljujem obitelji na podršci tijekom studiranja.

Sadržaj

1 Uvod **1**

2 Opis tehnologija **3**

2.1 TypeScript . . . 3
2.2 Tailwind . . . 6
2.3 Bootstrap . . . 7
2.4 Jednostranična web aplikacija . . . 7
2.5 JWT . . . 8
2.6 Spring Boot . . . 11
2.7 React . . . 13
2.8 Angular . . . 15

3 Opis aplikacije **18**

3.1 Stranica za prijavu . . . 18
3.2 Pregled računa šefa . . . 20
3.2.1 Početna stranica šefa . . . 20
3.2.2 Stvaranje novog zadatka . . . 22
3.2.3 Stranica s tablicom radnika . . . 23
3.2.4 Stvaranje novog radnika . . . 24
3.3 Pregled računa zaposlenika . . . 25
3.3.1 Početna stranica zaposlenika . . . 26
3.3.2 Stranica sa zaposlenikovim zadacima . . . 26

4 Usporedba React i Angular radnih okvira **28**

4.1 Programski jezik . . . 28
4.2 Renderiranje . . . 29

4.3 Komponente 32
4.4 Upravljanje stanjem podataka (eng. Data state management) 37
4.5 Forme 42
4.6 Zahtjevi na API 45
4.7 Proces učenja radnog okvira 48

5 Zaključak **50**

Bibliografija **52**

Pojmovnik **53**

Sažetak **54**

Popis slika

2.1 Primjer validnog JWT tokena iz aplikacije. 10

3.1 Izgled početne stranice za prijavu. 19
3.2 Primjeri grešaka prilikom prijave. 19
3.3 Gumbovi sa šefovog zaglavlja. 20
3.4 Izgled tablice s zadacima. 20
3.5 Lista završenih i nezavršenih podzadataka. 22
3.6 Izgled skočnog prozora za kreiranje zadataka. 23
3.7 Izgled tablice s radnicima. 24
3.8 Izgled skočnog prozora za kreiranje radnika. 25
3.9 Gumbovi sa zaposlenikovog zaglavlja. 26
3.10 Izgled tablice s nedodijeljenim radnicima. 27
3.11 Izgled tablice sa zaposlenikovim zadacima. 27

4.1 Primjeri tumačenja komponenti. 31

Popis isječaka koda

2.1 JavaScript neprovjeravanje grešaka 3
2.2 Primjer TypeScript sučelja 4
2.3 Primjer korištenja TypeScript sučelja u funkciji 5
2.4 Primjer stvaranja gumba s Tailwind CSS-om 6
2.5 Primjer Spring Bean definicije 11
2.6 Primjer Spring kontrolor klase 12
2.7 Primjer uslužne klase Spring Boota 12
2.8 Spring Boot primjer Repozitorija 13
2.9 TSX primjer 14
2.10 Primjer NgClass direktive 15
2.11 Primjer ngModel direktive 16
2.12 Primjer ngIf direktive 16
2.13 Primjer ngFor direktive 16

4.1 Primjer TSX-a u aplikaciji 29
4.2 Primjer korijenskog čvora Reacta 29
4.3 Primjer index.html datotke iz Angular aplikacije 30
4.4 Primjer definiranja TS klase komponente Angulara 32
4.5 Primjer definicije React komponente 33
4.6 Korištenje svojstva (eng. props) u Reactu 33
4.7 Korištenje anotacije Input u Angularu 34
4.8 Slanje podataka djetetu u Angularu 34
4.9 Korištenje varijable iz TS klase u HTML šabloni 35
4.10 Primjer anotacije Output u Angularu 35
4.11 Primjer korištenja EventEmitter klase u Angularu 35
4.12 Primjer slanja podataka od djeteta roditelju u Reactu 36

4.13 Primjer funkcije kad se klikne gumb u Reactu 36
4.14 Primjer korištenja useState kuke 37
4.15 Ažuriranje stanja u useState kuki 38
4.16 Primjer stvaranja React konteksta 38
4.17 Primjer korištenja useContext kuke 39
4.18 Primjer stvaranja korijenskog reduktora 39
4.19 Primjer stvaranja Redux skladišta 40
4.20 Primjer korištenja Reduxovih kuka 40
4.21 Primjer Angularove Behavior Subject klase 41
4.22 Primjer koda s HTML šablone Angular komponente 43
4.23 Primjer korištenja klase FormGroup 44
4.24 Primjer Formika i Yupa u React formi 45
4.25 Primjer korištenja fetch funkcije 46
4.26 Primjer odgovora na zahtjev u Reactu 46
4.27 Primjer zahtjeva koristeći Angularov HTTP API 47
4.28 Primjer korištenja RxJs subscribe funkcije 48

Poglavlje 1

Uvod

Cilj završnog rada bilo je napraviti detaljnu analizu i usporedbu Angular i React radnih okvira u izradi web aplikacija. Analiza je napravljena kroz izradu dvije identične RESTful jednostranične aplikacije (eng. Single Page Application, skraćeno SPA) u oba radna okvira, koristeći Spring Boot radni okvir za poslužiteljski dio aplikacije koji mora sadržavati osnovne *CRUD* (create, read, update, delete) operacije i autentifikaciju.

Za autentifikaciju korišten je Spring Security paket s kojim se fokusiralo na JWT autentifikaciju s klijentske strane. Potrebno je analizirati mogućnosti, arhitekturu i načine razvoja web aplikacija pomoću ova dva radna okvira, te doći do osobnog zaključka i mišljenja o oba okvira uočavajući prednosti i nedostatke jednog i drugog okvira tijekom obavljanja određenog zadatka unutar aplikacije.

Fokus je bio na slanju zahtjeva poslužitelju, kako funkcionira upravljanje stanja kroz aplikaciju, kako se šalju podaci između komponenta, koliko je lagano naučiti koristiti radni okvir. Nadalje, analiziralo se koliko dodatnih paketa treba instalirati za pojedine stvari, kako funkcionira dodavanje elemenata u Document Object Model (DOM), na koji način se pozivaju funkcije. Provjeravale su se razlike u *routeru*, koje su bile komplikacije u razvijanju i u kojem radnom okviru je lakše otklanjanje pogrešaka. Uz to promatralo se za koji radni okvir je lakše pronaći informacije te se uspoređivalo Reactove komponente te modularnost s Angularovom velikom bazom ugrađenih biblioteka s kojim postoji potpuni radni okvir. Naposlijetku, analiziralo se

Reactovo korištenje JSX-a i miješanje koda za programiranje sučelja i logike renderiranja i usporedilo s Angularovom MVC arhitekturom i odvojenim kodom za sučelje i renderiranje.

Aplikacija na kojoj je odrađeno testiranje napravljena s ciljem testiranja što više mogućnosti koje nude oba radna okvira. Stoga je tema aplikacije *Ticketing* sustav koji omogućuje firmi vođenje svojih zaposlenika i zadataka koje je potrebno odraditi. Postoje dva računa, račun šefa koji može stvarati nove zadatke, dodavati i otpuštati zaposlenike. Također ima pregled svih zadataka i zaposlenika. Račun zaposlenika odnosno radnika ima pregled slobodnih zadataka na koje se može dodijeliti te pregled svojih zadataka na kojima je dodijeljen te koje može završiti u tom vremenskom okviru čime šef vidi da je zadatak završen na vrijeme. Svaki zadatak također ima neodređeni broj podzadataka koje dodaje šef kada kreira zadatke. Osim toga, dodaje i kratki opis zadataka i datum do kojeg mora biti gotov. Zaposlenik može predati zadatak tek kada završi sve podzadatke.

Putem svih tih funkcionalnosti aplikacije omogućeno je testiranje oba radna okvira i njihovih specifičnih značajki. Za pomoćne alate u stvaranju aplikacije korišten je Intellij IDEA, razvojno okruženje koje je savršeno optimizirano za Java programski jezik te za JavaScript i TypeScript koji su korišteni u aplikaciji. Također, korišten je Bootstrap CSS radni okvir koje uvelike ubrzava proces dizajniranja stranice s već napravljenim komponentama koje se samo ubace i urede po želji. Za uređivanje stranice korišten je dodatno i Tailwind CSS radni okvir koji pomaže pri uređivanju komponenti i ima konfigurirane CSS komponente koje se ubace *inline* u HTML kod. Nadalje, za razne ikone koje se nalaze po stranicama aplikacije korištena je Hero Icons biblioteka ikona. Tip ove aplikacije je naravno jednostranična aplikacija s obzirom da se uspoređuju Angular i React radni okviri koji su specijalizirani za izradu takvog tipa aplikacije.

Što se tiče poslužiteljske strane aplikacije, kao što je već prije spomenuto, korišten je Spring Boot radni okvir koji sadrži sve dijelove za stvaranje jednostavne aplikacije koja ne treba posebni poslužitelj već se samo pokrene i radi. Spring Boot u sebi već sadrži H2 bazu podataka koja se nalazi u memoriji aplikacije. Za sigurnost je korištena Spring Security biblioteka koja u sebi sadrži alate za konfiguraciju autentifikacije i autorizacije korisnika pomoću Auth0 JWT tokena.

Poglavlje 2

Opis tehnologija

2.1 TypeScript

Kako bi se razumio TypeScript moraju se znati osnove JavaScripta, jer on je upravo razlog potrebe za TypeScriptom. U svojim početcima JavaScript (JS) je bio namijenjen jednostavnim komandama i skriptama koje bi se pokretale na statičnim web stranicama. S vremenom, doduše, počeo se koristiti sve više za sve kompleksnije operacije i od pomoćnog alata na web stranicama prešao je u glavni programski jezik cijelog internetskog web sučelja. Iako se s vremenom JS optimizirao i dodana je mogućnost Application programming interface (API), u suštini je ostala arhitektura jezika namijenjenog za jednostavne skripte [1].

Jedan od glavnih problema je manjak statičnog provjeravanja koda, odnosno JS ne javlja potencijalne greške prije kompilacije koda čime se programeru znatno smanjuje sposobnost rješavanja grešaka u trenutku pisanja [1]. Na primjer u isječku koda 2.1 u objektu *kvadrat* varijabla se zove *visina* dok ju se u *površina* konstanti

```
1 const kvadrat = { sirina: 5, visina: 5 };
2 const povrsina = kvadrat.sirina * kvadrat.visna;
```

Isječak koda 2.1 JavaScript neprovjeravanje grešaka

pozove *visna*. U gornjem slučaju JavaScript ne bi javio da postoji greška. Ovaj

problem TypeScript rješava tako što uvodi tipove i provjerava ako postoje greške s obzirom na operacije koje se odvijaju nad tim vrijednostima. Takvo provjeravanje zove se statično provjeravanje tipova. Na primjer, ako se istu stvar ako napravi u TS-u javit će se ova greška : "*Property 'visna' does not exist on type ' sirina: number; visina: number; '. Did you mean 'visina'?*".

Međutim, TypeScript (TS) koristi sva pravila JS-a, uključujući sintaksu i ponašanje tijekom izvođenja programa, čime se svaki JS program može i izvršiti u TS-u, odnosno sav TS kod se prevodi u JS čime ponašanje nakon komplikacije bude jednako u oba jezika [1].

Glavna značajka TS-a su tipovi koji su nalik klasama u Javi, no nisu striktni u svojoj uporabi kao što su klase u objektno orijentiranom programiranju, već se podaci mogu fleksibilno slati bez da budu u predodređenoj klasi. Ovo se u kodu koristi kao *interface* odnosno sučelje u kojem se definira koje podatke bi to sučelje trebalo imati i kojeg su tipa ti podaci [1]. Na primjer u projektu je često korišten isječak koda 2.2.

```
1     export interface User {
2     id: number;
3     email : string;
4     name: string;
5     token: string;
6     role : Role;
7 }
```

Isječak koda 2.2 Primjer TypeScript sučelja

U ovom isječku koda postoji sučelje *User* koje je korišteno u svrhu prijave korisnika u sustav kako bi se sa servera dobili točni podaci koji se onda spremaju u varijablu tipa *User*. Ovo olakšava upotrebu te varijable jer upozori programera ako koristi neke vrijednosti koje ne postoje u tom sučelju. U isječku koda 2.3 se vidi korištenje sučelja *User*.

Može se vidjeti da konstanta *onSubmit* prima varijablu *values* koja ima *UserLogin* sučelje, te tu varijablu šalje u *employeeLogin* funkciju. Nakon što ta funkcija izvrši

```
const onSubmit = async (values: UserLogin) => {
        let response = await api.employeeLogin(values);
        localStorage.setItem("currentUser",
        JSON.stringify(response))
        if(response.role === 'WORKER') {
            history.push("/worker");
        } else if(response.role === 'BOSS') {
            history.push("/boss/home");
        }
        window.location.reload();
};
...
export async function employeeLogin
    (data : UserLogin) : Promise<User> {
...
return response.json();
}
```

Isječak koda 2.3 Primjer korištenja TypeScript sučelja u funkciji

sve naredbe ona vraća *response.json()* kojem je dodijeljeno sučelje *Promise<User>*. Potrebno je omotati *User* u *Promise* jer je ovo asinkrona funkcija koja čeka podatke sa servera, stoga treba nastaviti s izvršavanjem dok se čeka na odgovor. Nakon što dobije odgovora funkcija *employeeLogin()* vraća vrijednost sa sučeljem *User*. Kad ta vrijednost dođe nazad u konstantu *onSubmit* iz nje se koristi varijabla *role* kako bi se znalo koju ulogu ima korisnik koji se trenutno prijavio i sustav i preko toga bi ga poslalo na početnu stranicu šefa ili zaposlenika funkcijom *history.push()*.

TS je korišten uz React i Angular radne okvire kako bi olakšao razvijanje i smanjila sklonost pogreškama. Angular po zadanom načinu rada koristi TS, zato ga nije bilo potrebno dodati već se s njime automatski i radi, no u React se dodaje preko npm komande *npm install –save typescript|*

Svaka JavaScript biblioteka koja se dodaje kao zavisnost u aplikaciju ima i posebnu TypeScript verziju koja kad se dodaje preko npm komande često kao prefix

ima *@types* čime se naglašava da je prilagođena TS-u.

2.2 Tailwind

Tailwind je zamjena za tradicionalne načine uređivanja teksta preko Cascading Style Sheets (CSS)-a. Pisanje klasa za uređivanje Hypertext Markup Language (HTML) datoteka dugo traje i globalno je što znači da izmjena za neki dio stranice često izmjeni druge stvari koje mogu imati nepredvidljive posljedice. Ako se to želi izbjeći moraju se pisati dodatne postavke, što uzrokuje sve veće *.css* datoteke. Tailwind je stoga lokalan, te ga se piše zasebno u *className* dijelu oznake u Reactu, a u *class* kad koristimo Angular [2].

Tailwind čini skup već gotovih CSS klasa koje je pripremio Tailwind tim, one omogućuju konzistentan stil kroz aplikaciju bez mnogo rada i konfiguracije [2]. Na isječku koda 2.4 je gumb dizajniran s ovom tehnologijom koji je često korišten u aplikaciji.

```
<button
    value={value}
    onClick={onClick}
    className="group relative w-full flex
    justify-center py-2 px-10
    border border-transparent
    text-sm font-medium rounded-md text-white bg-blue-600
    hover:bg-blue-700 focus:outline-none focus:ring-2
    focus:ring-offset-2 focus:ring-blue-500"
>
...
</button>}
```

Isječak koda 2.4 Primjer stvaranja gumba s Tailwind CSS-om

Gumb je napisan u HTML-u, koristeći React radni okvir, no Tailwind se može zamijetiti u *className* dijelu oznake *<button>*. Kroz predodređene postavke, jed-

nostavno je složiti gumb modernog izgleda, a pošto je napisan u Reactu, jednostavno ga je koristiti na više mjesta.

Kao što se vidi koristeći Tailwindove postavke, vrlo lako je definirati granice i debljinu gumba s *px* i *py* klasama. Nadalje, definirano je da su rubovi zaokruženi jednostavno koristeći *rounded* postavku. Jednom kad se nauči kako je nazvana koja konfiguracija, uređivanje HTML-a postane vrlo jednostavno i brzo.

Problem kod Tailwinda je ružan HTML kod prepun Tailwindovih klasi unutar datoteka. Time je ponekad teško čitati datoteke, ali to je subjektivno i ovisi o programeru što mu je bitnije za projekt. Aplikacija za ovaj projekt je mala i nije trebala posebne CSS postavke, stoga je Tailwind uvelike olakšao razvoj i pobrzao uređivanje.

2.3 Bootstrap

Bootstrap je CSS radni okvir koji sadrži znatan broj komponenti za web sučelje, napravljen s ciljem da ubrza razvijanje web stranica ubacivanjem često korištenih komponenti u postojeći CSS i dodajući HTML oznake koje se mogu ubaciti u dizajn stranice. Omogućuje i korištenje JavaScript komponenti, poput skočnih prozora, navigacijskih padajućih izbornika itd. Pošto je projekt usporedba Reacta i Angulara, Bootstrap također ima svoje prilagođene verzije za ove radne okvire, a to su *Reactstrap* za React i *ng-bootstrap* za Angular.

2.4 Jednostranična web aplikacija

Jednostranična aplikacija (eng. Single-page application, skraćeno SPA) implementacija je web-aplikacije koja učitava samo jedan web-dokument, a zatim ažurira sadržaj tijela tog pojedinačnog dokumenta putem JavaScript API-ja kao što su *XMLHttpRequest* i *Fetch* kada se treba prikazati različit sadržaj. Prednosti takvog tipa aplikacije je učitavanje resursa (HTML, CSS i skripte) samo jednom tijekom života aplikacije, a između klijenta i poslužitelja šalju se samo podaci. Nije više potrebno

pisati kod za renderiranje stranica na poslužitelju, već se može početi samo učitavanjem datoteke na pretraživač.

Ovakve stranice mogu sve podatke koji su im potrebni spremiti u lokalni spremnik pretraživača čime mogu raditi bez internetske veze. Time se omogućuje korisnicima korištenje stranica bez potrebe učitavanja dijelova kojih su već u spremniku s poslužitelja.

Iako imaju mnogo prednosti, SPA-ovi imaju i znatne nedostatke, kao što je otežavanje optimizacije pretraživača (eng. Search Engine Optimization, skraćeno SEO) zato što jednostranične web aplikacije učitavaju hiperveze tek kada se uđe u dio stranice u kojem se te veze nalaze. Zbog toga *Googleova* optimizacija ne može dostići istražiti cijelu stranicu i time pohvatati sve veze koje su potrebne za optimizaciju.

Nadalje, potrebno je više truda za održavanje stanja što će biti objašnjeno kod implementacije u React i Angular radnim okvirima. Također, sigurnost kod *Cross Site Scripting* napada je slaba, čime su hakeri u mogućnosti ubacivati skripte u web aplikacije drugih korisnika.

2.5 JWT

JSON Web Token (JWT) je otvoreni standard koji definira kompaktan i samostalan način za siguran prijenos informacija između strana kao JSON objekt. Može se poslati putem URL-a, putem POST parametra ili unutar HTTP zaglavlja i brzo se prenosi. Sadrži sve potrebne informacije o entitetu kako bi se izbjeglo više od jednog upita na bazu podataka. Primatelj JWT-a također ne treba pozvati poslužitelja da bi potvrdio token. Još jedna prednost JWT-a je što koristi JSON, a parseri JSON-a su česti u programskim jezicima, jer omogućuju direktno mapiranje na objekte, što znatno olakšava posao programera[3].

Glavna primjena JWT tokena su [3]:

- Autentifikacija: Kada se korisnik uspješno prijavi koristeći svoje vjerodajnice, vraća se ID token. Prema specifikacijama OpenID Connect (OIDC), ID token je uvijek JWT.

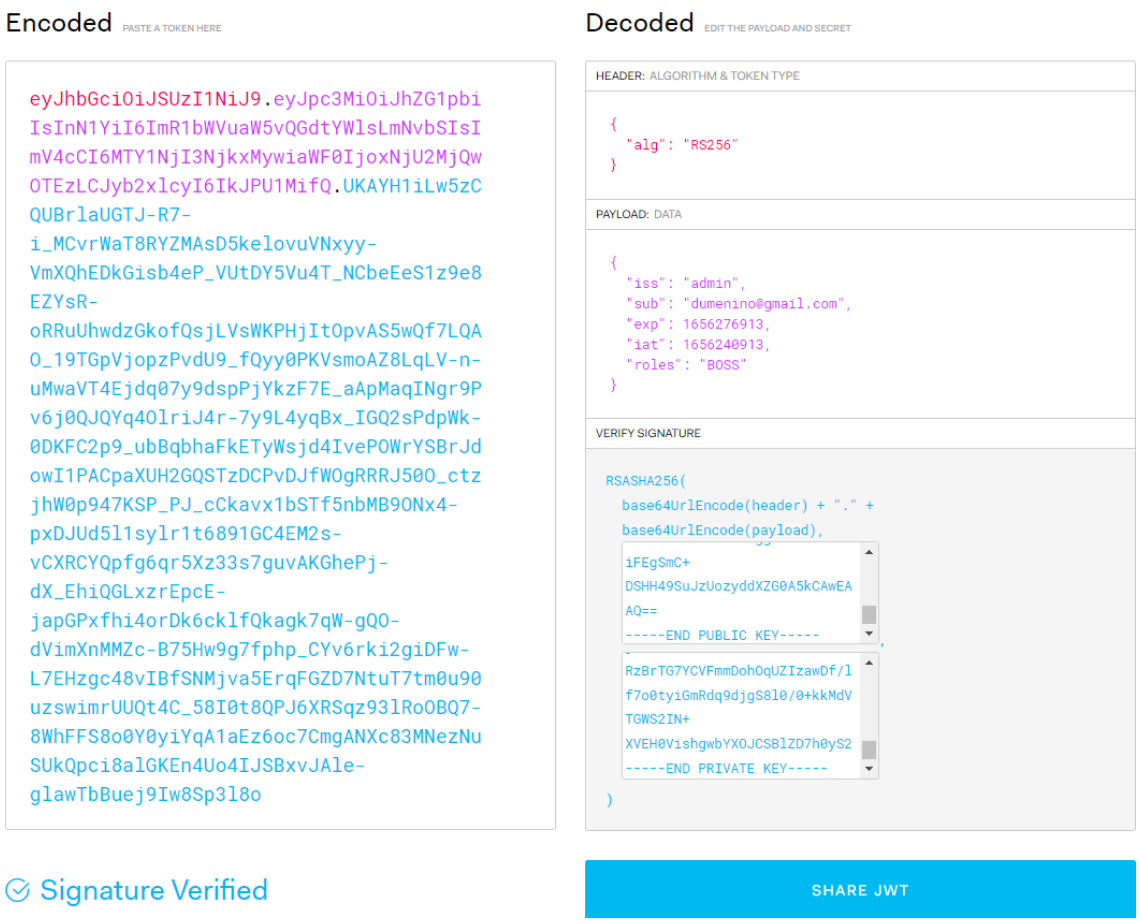
- Autorizacija: Nakon što je korisnik uspješno prijavljen, aplikacija može zatražiti pristup rutama, uslugama ili resursima (npr. API-ji) u ime tog korisnika. Da bi to učinio, u svakom zahtjevu mora biti poslan Token pristupa, koji može biti u obliku JWT-a. Jedinstvena prijava (eng. Single sign-on, skraćeno SSO) naširoko koristi JWT zbog malih troškova formata i njegove sposobnosti da se jednostavno koristi u različitim domenama.
- Razmjena informacija: JWT-ovi su dobar način sigurnog prijenosa informacija između strana jer se mogu potpisati, što znači da korisnik može biti siguran da su pošiljatelji oni za koje kažu da jesu. Osim toga, struktura JWT-a omogućuje vam da provjerite da sadržaj nije mijenjan.

Općenito, JWT-ovi se mogu potpisati korištenjem tajne (s HMAC algoritmom) ili para javnih/privatnih ključeva pomoću RSA. Kada se tokeni potpisuju pomoću parova javnih/privatnih ključeva, potpis također potvrđuje da je samo strana koja drži privatni ključ ona koja ga je potpisala [3].

Prije nego što se primljeni JWT iskoristi, treba ga ispravno potvrditi korištenjem njegovog potpisa. Uspješno provjereni token znači samo da informacije sadržane u tokenu nisu mijenjane od strane bilo koga drugog. To ne znači da drugi nisu mogli vidjeti sadržaj koji je pohranjen u običnom tekstu. Zbog toga osjetljive informacije nikada ne bi trebale biti pohranjivane unutar JWT-a i trebali bi biti poduzeti drugi koraci kako bi se osiguralo da JWT-ovi ne budu presretnuti, kao što je slanje JWT-ova samo preko HTTPS-a, sljedeća najbolja praksa je korištenje samo sigurnih i ažuriranih biblioteka za validaciju tokena [3].

Na slici 2.1 vidi se primjer JWT tokena kojeg je generirala aplikacija nakon prijave korisnika [4]. Crvenom bojom je naglašen algoritam koji je korišten u potpisivanju tokena, a to je RS256. Razlog zbog kojeg je on korišten u aplikaciji je taj što on omogućuju verifikaciju da sadržaj tokena nije mijenjan, a to se radi kombinacijom privatnog i javnog ključa.

Javni ključ služi za dekodiranje tokena i šalje se korisniku, a privatni ključ služi za enkodiranje na serverskoj strani. Oni se stvore u paru, te tako i funkcioniraju za verifikaciju. Nakon toga, se nalaze podaci naglašeni ljubičastom bojom, oni ne smiju biti povjerljivi, već samo služe za autentifikaciju ubuduće.



Slika 2.1 Primjer validnog JWT tokena iz aplikacije.

U podacima se nalazi *iss* odnosno *issuer*, izdavač tokena kao i *sub* ili subjekt za kojeg je namijenjen token. Također, nalazi se i *exp* koji predstavlja datum isticanja validnosti tokena, dok je *iat* datum koji označava kad je token napravljen. Posljednji podatak je uloga koju ima korisnik koji se ulogirao, a to je u ovom slučaju *BOSS*, uloga koja ima ovlasti admina u aplikaciji.

Na dnu se nalazi potpis. Kao što je prije navedeno koristi se RS256 algoritam, te je stoga u prvoj kućici upisan javni ključ, a u drugoj privatni, čime sustav na stranici `jwt.io` kvačicom potvrđuje da je token validan.

Implementacija samog sustava autentifikacije i autorizacije će se objasniti u po-

glavlju koje će se baviti *Spring Boot* radnim okvirom, jer je tamo odrađena gotovo sva logika.

2.6 Spring Boot

Spring Boot je platforma za Java programere za razvoj samostalne i proizvodne Spring aplikacije koja se može jednostavno pokrenuti. Može se započeti s minimalnim konfiguracijama bez potrebe za cijelom postavkom Spring konfiguracije.

Najbitnije stavke Spring Boota su: Inverzija kontrolnog (IoC) spremnika, koji pruža dosljedna sredstva za konfiguriranje i upravljanje Java objektima. Kontejner je odgovoran za upravljanje životnim ciklusom objekata određenih objekata: stvaranje ovih objekata, pozivanje njihovih metoda inicijalizacije i konfiguriranje ovih objekata povezujući ih. U suštini on kontrolira sve klase koje se definiraju kao Beanovi, te ih drži u spremniku od kuda se one dohvaćaju kad zatrebaju. Klasa se definira Bean anotacijom te onda bude spremljena u spremnik. Na isječku koda 2.5 se vidi primjer *Bean* klase.

```
1 @RestController
2 @RequestMapping("/auth/")
3 @AllArgsConstructor
4 public class AuthController {
5     private final AuthenticationManager authenticationManager;
```

Isječak koda 2.5 Primjer Spring Bean definicije

Vidi se anotacija `{@RestController` koja označava *Bean* te kontrolni sloj aplikacije, koji će kasnije biti pojašnjen. Tom anotacijom IoC spremnik zna da ovu klasu treba spremiti za daljnje korištenje. Kada se želi koristiti *Bean* klasa ona se definira kao *private* i *final* te se mora zvati u konstruktoru, a to se odradi s `@AllArgsConstructor` anotacijom koja sve definirane vrijednosti zove u konstruktor.

Okvir za pristup podacima (eng. *Spring Data framework*), omogućuje izbjega-

vanje SQL-a i obavljanje komunikacije s bazom preko okvira poput JPA i Hibernate pišući samo Java kod. Na ovom projektu korišten je JPA, koji omogućuje automatsko zatvaranje i otvaranje transakcijskih prozora s bazom i ima vrlo intuitivan način korištenja s anotacijama i repozitorijskim sučeljem.

Spring Boot projekti su razdijeljeni na 3 sloja, koji odrađuju cjelokupni posao od primanja zahtjeva, odrađivanja poslovne logike, obrađivanja podataka, spremanja u bazu podataka, te vraćanja podataka koje je zahtjev zatražio.

Kontrolor (eng. Controller) upravlja REST sučeljem koje spaja serversku i klijentsku stranu te rješava zahtjeve i odgovara na njih, on prosljeđuje ono što primi s klijentske strane i daje Uslužnom sloju. Primjer kontrolora vidi se na isječku koda 2.6.

```
1 @RestController
2 @AllArgsConstructor
3 @RequestMapping("/ticket/")
4 public class TicketController {
```

Isječak koda 2.6 Primjer Spring kontrolor klase

Na kodu iznad se vidi anotacija *RestController* koja označava da je ta klasa kontrolor, dok anotacija *RequestMapping* govori da je put do ovog kontrolera *root* adresa poslužitelja s dodanim */ticket/*.

Uslužni sloj nadalje odrađuje poslovnu logiku i radi operacije nad podacima, te na kraju ako želi imati interakciju s bazom podataka ih prosljeđuje Repozitoriju. Primjer definicije Uslužnog sloja se vidi na isječku koda 2.7. Definicija Uslužnog

```
1 @Service
2 @AllArgsConstructor
3 public class TicketService {
```

Isječak koda 2.7 Primjer uslužne klase Spring Boota

sloja vidi seanotacijom *@Service* koja je čisto estetska jer zapravo ništa ne mijenja no naglašava programerima što je namjena ove klase.

Repozitoriju je posao da sprema, dohvaća i briše podatke iz baze te s njom komunicira. Definicija se vidi na isječku 2.8 Isječak koda 2.8 prikazuje anotaciju Repository

```
@Repository
public interface TicketRepository
    extends JpaRepository<Ticket, Long> {
    List<Ticket> findAllByEmployee(Employee employee);
    List<Ticket> findAllByEmployeeIsNull();
}
```

Isječak koda 2.8 Spring Boot primjer Repozitorija

koja daje Springu do znanja da je ovo sučelje repozitorij, a nasljeđivanjem sučelja JpaRepository repozitoriju daje moćne JPA alate koji omogućuju komunikaciju s bazom preko Javinog koda.

2.7 React

React je besplatna JavaScript biblioteka otvorenog koda za građenje korisničkih sučelja temeljenih na posebnim JSX React komponentama. Napravljena je od strane Mete (prije Facebooka) i glavna svrha je gradnja jednostraničnih aplikacija. Bavi se upravljanjem stanjem i prikazivanjem tog stanja u DOMu, tako da stvaranje React aplikacija obično zahtijeva korištenje dodatnih knjižnica za usmjeravanje, kao i određene funkcionalnosti na strani klijenta [5].

Glavna svojstva Reacta su: komponente, one su entiteti koji grade Reactov kod. Njihova poanta je da se mogu ponovno koristiti na više mjesta a podaci između komponenata se dijele pomoću svojstava (eng. props). Oni su nepromjenjiva svojstva koja se mogu poslati između roditeljske i dječje komponente, poput funkcija i varijabli[5].

Reactove komponente se pišu u JSX-u, no u aplikaciji se pišu u TypeScript verziji TSX-u. TSX se koristi jer vizija Reacta je da sva logika renderiranja i sučelja vrlo povezana te stoga je bolje da se piše u istim datotekama te da je HTML izravno

podložan manipuljaciom JS funkcijama. Na primjer dopušta stvari kao na ovom isječku koda 2.9

```
const ticketDescription = ticket.description;
const element = <h1>Description: {ticketDescription}</h1>;
```

Isječak koda 2.9 TSX primjer

Drugo svojstvo Reacta su Kuke (eng. Hooks), funkcije koje dopuštaju programerima da se "priključe" na React stanje i značajke životnog ciklusa iz funkcijskih komponenti [5]. Iako su kuke Javascript funkcije, imaju 2 zasebna pravila a to su:

- Kuke se mogu zvati samo na najvišoj razini. Ne smiju se pozivati unutar petlji, uvjeta ili ugniježđenih funkcija. Prativši ovo pravilo osigurava se da su kuke pozvane u istom redu svaki put kada se komponenta rerenderira, a to omogućuje da se očuva stanje kuka između više poziva kuka[5].
- Kuke se smiju zvati samo iz Reactovih funkcijskih komponenti. Ne smiju se zvati iz uobičajenih JavaScript funkcija i vlastitih kuka koje programer može sam napraviti. Ovo pravilo treba pratiti jer se osigurava da je sva logika stanja u komponenti jasno vidljiva iz njenog izvornog koda[5].

Dvije kuke koje se najviše koriste su *useEffect* i *useState*. *useEffect* kuka funkcionira tako da se pokrene kad se React komponenta montira (eng. mount), što znači da se ova kuka pokrene nakon svakog renderiranja, ona se najčešće koristi kada trebamo učitati podatke u stranicu svaki put kad se ta stranica renderira [5].

useState kuka se koristi za očuvanje stanja između renderiranja. Ona vraća par, trenutno stanje i funkciju za ažuriranje tog stanja. Ova kuka je korisna kada treba očuvati neke podatke između renderiranja, na primjer podatke u tablici [5].

Treće značajno svojstvo je Virtualni DOM, što je preslika DOM-a koja se mijenja kad god se promijene stanja. Kada su stanja gotova s mijenjanjem onda se stvari izmjene u pravom DOM-u. On postoji jer mijenjanje pravog DOM-a je sporo i skupo dok je mijenjanje Virtualnog DOM-a jeftino i brzo. Stoga se prvo u Virtualnom DOM-u rade promjene, a kad se završe, onda se izmjene i dijelovi pravog DOM-a [5].

2.8 Angular

Angular je besplatan TypeScript radni okvir otvorenog koda temeljen na komponentama koje se sastoje od 4 dijela, a to su: HTML šablona koja definira što će se renderirati na stranici, TypeScript klasa koja definira logiku i ponašanje stranice, CSS birač koji definira kako će se komponenta koristiti u šabloni i opcionalno CSS datoteka sa stilovima. Angular je razvio Google 2016. godine i podupiran je od strane tog tima i ostalih nezavisnih razvijača [6].

Glavna svojstva su komponente, šablone, direktive i injekcija zavisnosti. Komponente su kao što je prije opisao temeljni dijelovi svake Angular aplikacije. Ono što se vidi na ekranu je napisano u HTML šabloni, no u tu šablonu se može ubaciti podaci koji se nalaze u TypeScript klasi za tu komponentu. U toj klasi također se može pomoću događaja životnog ciklusa manipulirati što će se kada dogoditi, npr sa *ngOnInit* kukom se može odraditi neku logiku na stranici kad se ona tek krene stvarati [6].

Direktive su pomoćne klase posebne za Angular koje omogućuju dodatno manipuliranje HTML elementima, atributima, postavkama i komponentama. Tri najčešće atributske direktive su:

- *NgClass* je direktiva koja miče ili dodaje set CSS klasa s obzirom na uvjet koji se postavi[6], npr. na isječku koda 2.10.

```
1    <div [ngClass]="isWorker ? 'worker' : 'boss'">...</div>
2
```

Isječak koda 2.10 Primjer NgClass direktive

- *ngStyle* omogućuje da se postavi stil na element tako što se svojstvo koje je definirano u TS klasi poveže na *ngStyle*.
- *ngModel* je direktiva koja omogućuje dvostrano povezivanje atributa s TS klase s poljem za unos u HTML šabloni [6], na primjer kao u sljedećem isječku koda 2.11

```
<label>Email</label> <input
    type="email" class="form-control"
    formControlName="email"
    [(ngModel)]="loginData.email" required >

```

Isječak koda 2.11 Primjer ngModel direktive

[6]. Postoje još tri bitne strukturalne direktive koje su zaslužne za HTML raspored, tako što manipuliraju elemente na razne načine [6], a to su

- NgIf koji postavlja uvjet. Ukoliko je uvjet istinit element se renderira, inače ne [6]. Na isječku koda 2.12 može se vidjeti i provjeravanje *boolean* varijable *isBoss*. Ako je uvjet istinit onda se renderira kod unutar *<td>* oznake.

```
<td ngIf='isBoss'>
    ...
</td>

```

Isječak koda 2.12 Primjer ngIf direktive

- NgFor je direktiva koja imitira *for* petlju. Na primjer, može poslužiti i za renderiranje liste podataka, a primjer direktive može se vidjet na isječku koda 2.13 [6].

```
<th *ngFor="let ticket of tickets">
    <td>{{ticket.description}}</td>
</th>

```

Isječak koda 2.13 Primjer ngFor direktive

- NgSwitch je set direktiva za prikazivanje jednog elementa od mogućih više s obzirom na neki uvjet. Dakle, ngSwitch radi isto kao *switch* operater u JavaScriptu ili C-u [6].

Posljednje svojstvo je injekcija zavisnosti koja omogućuje da se uslužne klase ubace u klase komponenta. Uslužne klase služe za logiku koja ima višekratnu upotrebu i može se koristiti na više komponenta, čime smanjuje dupliciran kod [6].

Poglavlje 3

Opis aplikacije

3.1 Stranica za prijavu

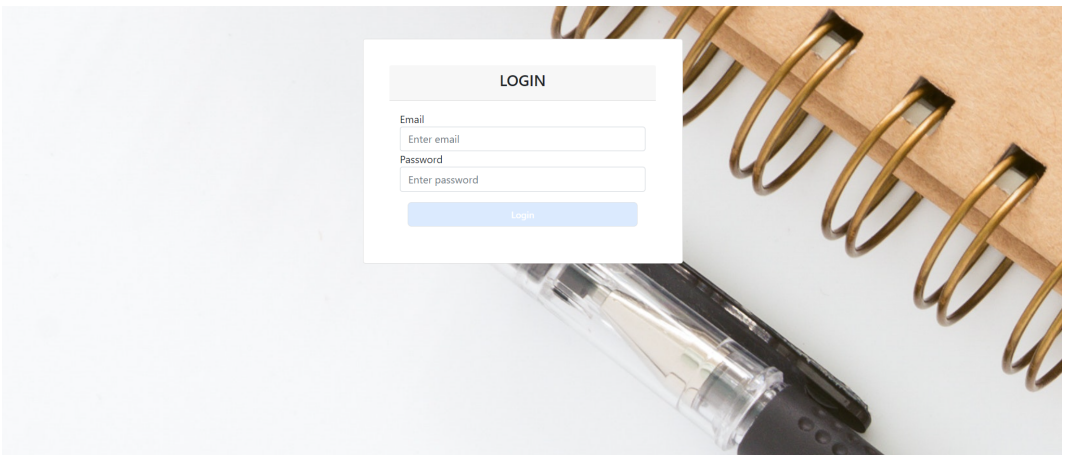
Aplikacija služi vođenju zadataka u firmi, te provjeri koje zadatke trenutno rade zaposlenici kao i do kad ih trebaju napraviti. Osmišljena je da ima 2 vrste profila, profil šefa i radnika. Šef će moći dodavati radnike i zadatke, a radnik će odabrati koji zadatak želi odraditi i kad označiti ga kad završi. Svaki zadatak u sebi ima neodređeni broj podzadataka, koji olakšavaju radniku da podijeli posao na manje dijelove, i završava ih jedan po jedan, te na kraju završi cijeli krovni zadatak. Šef može otpustiti radnika i tako ga maknuti iz sustava.

Ulaskom na početnu stranicu web aplikacije korisniku je prikazan ekran za prijavu prikazan na slici 3.1.

Na ekranu je jasno što se traži od korisnika, a to je njegov e-mail te njegova lozinka. U slučaju šefa, on je već otpočetka ubačen u bazu podataka, a radniku je dodijeljena šifra i službeni e-mail kad ga šef doda u sustav.

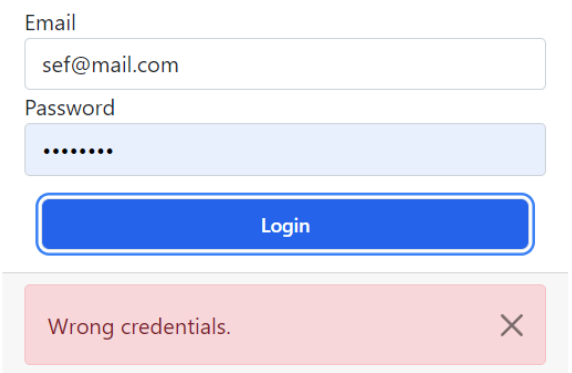
Ako se upišu ispravni podaci za šefa, sustav će prebaciti osobu na početni ekran za šefa, a ako se upišu ispravni podaci za radnika, prebacit će korisnika na početni ekran radnika.

Korisnik može upisati neispravne podatke te se pokušati u prijaviti sustav, no time će forma javiti da on ima krive podatke kao što je prikazano na slici 3.2a.

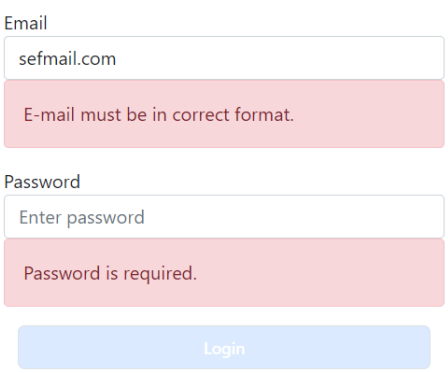


Slika 3.1 Izgled početne stranice za prijavu.

Na slici 3.2b vidi se primjer neuspjele validacije podataka gdje e-mail mora biti u ispravnom formatu npr. *sef@gmail.com* te oba polja moraju biti ispunjena. Također, na slici 3.2b se vidi da polje za lozinku nije popunjeno. Kada se to polje popuni i e-mail bude u ispravnom formatu, gumb za prijavu neće više biti onemogućen za stisnuti te će se moći prijaviti u sustav.



(a) Primjer neuspjele prijave.



(b) Primjer neuspjele validacije.

Slika 3.2 Primjeri grešaka prilikom prijave.

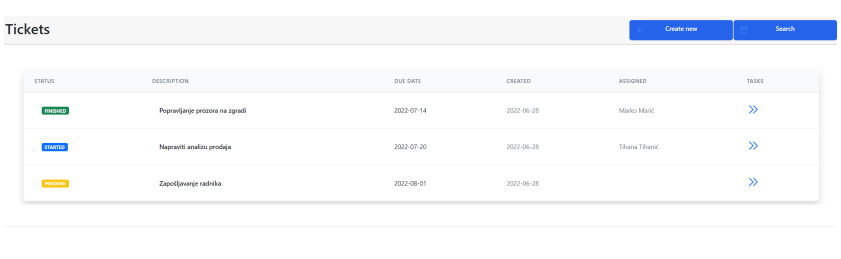
3.2 Pregled računa šefa

3.2.1 Početna stranica šefa

Uspješno se prijavivši, šef će biti prebačen automatski na početnu stranicu za njegov profil. Na ovoj stranici prvo će biti primijećeno zaglavlje ekrana od kojeg gumbove vidimo na slici 3.3. Nalaze se 3 opcije, prva je gumb s e-mailom prijavljenog šefa. Ako se pritisne taj gumb, navigacija odvede korisnika na početnu stranicu na kojoj se nalazi trenutno. Drugi gumb odvede subjekta na stranicu gdje se vide svi zaposlenici tvrtke, a ona će kasnije biti detaljnije objašnjena. Posljednji gumb s lokotom te tekstom *logout*, trenutnog korisnika odjavi iz sustava te ekran prebaci na ekran za prijavu na slici 3.1, čime se korisnik osigurava da mu netko ne može dirati stvari po profilu. No, ako se zaboravi odjaviti, sustav automatski odjavi korisnika automatski nakon određenog vremena.

Slika 3.3 Gumbovi sa šefovog zaglavlja.

Kad se pogledom spusti dolje, korisnik će primijetiti glavni dio početnog ekrana, a to je tablica sa svim trenutnim zadacima u sustavu prikazana na slici 3.4.



Slika 3.4 Izgled tablice s zadacima.

Tablica automatski preko poslužitelja pretraži sustav za svim zadacima te ih prikaže u tablici. Pregledavajući detaljnije tablicu s lijeva na desno, primjećuje se prvi stupac koji označava status određenog zadatka.

Spuštajući se po stupcu, vidi se da svaki od zadataka trenutno ima različiti status, a prvi u redu je zelene boje u kojem se nalazi tekst *FINISHED*, signalizirajući da je ovaj zadatak potpuno završen. Ispod tog status nalazi se status pobojan plavom bojom s tekstom *WORKING* čime se da zaključiti da ovaj zadatak je u procesu rješavanja. Posljednji status pobojan je žutom bojom s tekstom *PENDING* što znači da ovaj zadatak još nije nikome dodijeljen te se čeka da ga neki zaposlenik uzme i krene raditi.

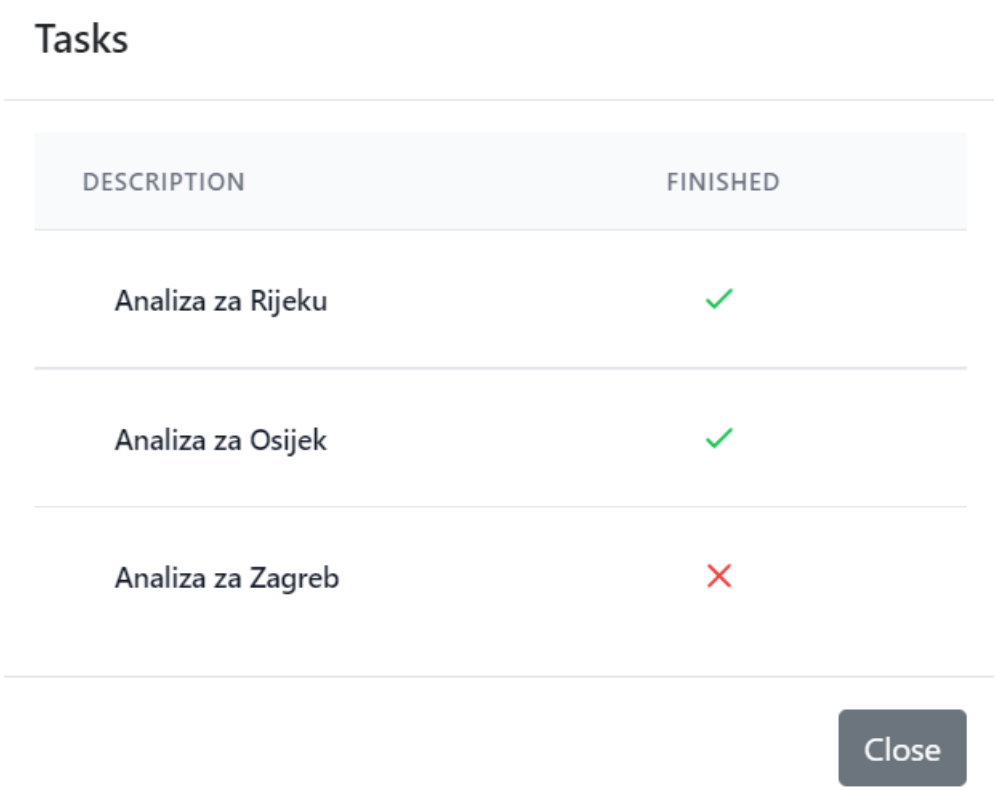
Sljedeći stupac se naziva Opis (eng. Description) koji označava skraćeni opis ovog zadatka, kako bi zaposlenik mogao dobiti generalnu ideju o čemu se zadatak radi i da za njega može pogledati koje podzadatke ima.

Nadalje, treći stupac označava datum do kojeg zaposlenik mora završiti odabrani zadatak, a odmah nakon tog, vidi se i datum na kojem je šef napravio zadatak. Stoga zaposlenik može dobiti ideju o vremenskom okviru u kojemu mora završiti zadatak.

Pri kraju se nalazi stupac *Assigned* koji naglašava kojem zaposleniku je dodijeljen koji zadatak. Prva dva imaju dodijeljene osobe, Marka Marića i Tihanu Tihanić, no treći nema nikoga, što znači da taj zadatak još nije krenut s rješavanjem što se može i vidjeti sa statusom *PENDING*.

Na samom kraju nalazi se stupac *Tasks* s gumbovima u obliku strelica, koji označava detaljniji pregled što se točno treba raditi u zadatku, ako zaposleniku nije dovoljan kratki opis. Pritisnuvši na jednu od strelica korisniku će se otvori skočni prozor s listom podzadataka, od reda s kojeg je stisnuta strelica, kao što je prikazano na slici 3.5.

Tablica s podzadacima sadrži samo 2 stupca, a to je opis i status zadatka. Šef ne može ništa raditi na ovom prozorčiću već samo provjeravati napredak zaposlenika u njihovom nastojanju da završe cjelokupni zadatak na vrijeme. Kada je gotov s provjeravanjem može pretisnuti gumb *Close* čime se skočni prozor zatvori i ekran s tablicom zadataka je ponovno u fokusu.



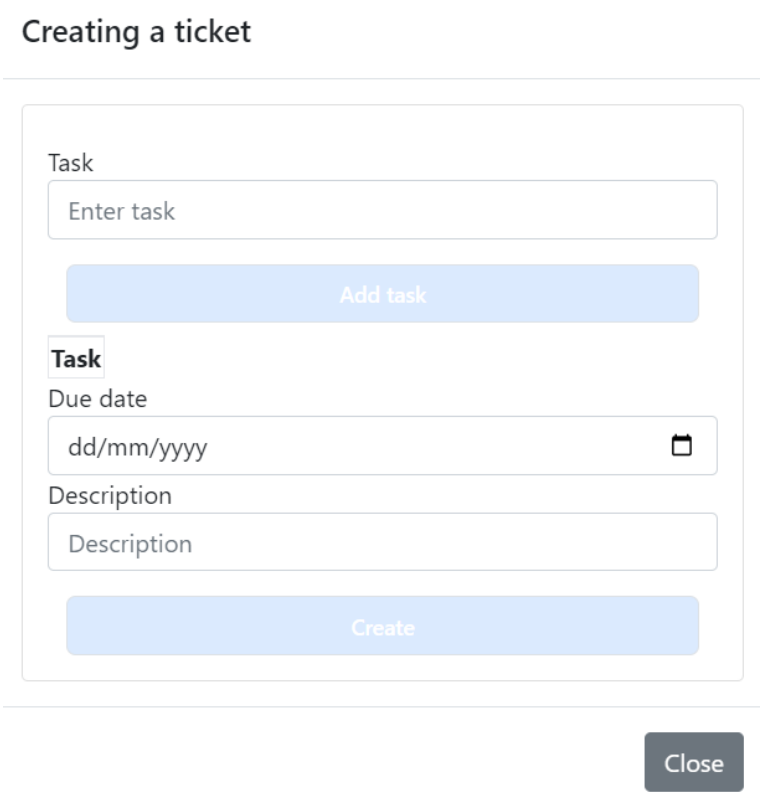
Slika 3.5 Lista završenih i nezavršenih podzadataka.

3.2.2 Stvaranje novog zadatka

Na slici 3.4 u gornjem desnom se vide dva gumba, prvi je *Search* s ikonicom arhivske kutije, koji pretraži bazu podataka za svim zadacima, te ih prikaže u tablici. Drugi gumb koji se zove *Create new* s ikonicom plusa otvara novi skočni prozor prikazan na slici 3.6.

Ovaj prozor nudi broj polja za unos s kojima se kreira novi zadatak. Prvo prozor traži dodavanje podzadataka upisivanjem opisa za određeni podzadatak, te se time dodaju u malu tablicu ispod samog polja, tako da korisnik zna koje je podzadatke dodao, i ima listu svih kojih planira ubaciti u zadatak. Ispod toga je polje koji zahtjeva da se upiše datum do kojeg zadatak mora biti predan, te na posljetku kratki opis cijelog zadatka. Kada se ispune sva polja prozor će dopustiti da se

pretisne gumb *Create* čime se zadatak ubaci u sustav. Kada je ubačen, zadatku se automatski postavi *PENDING* status čime ga zaposlenici mogu vidjeti i dodijeliti ga sebi. Ukoliko se korisnik predomisli, pritisne sivi gumb *Close* i zatvori skočni prozor.



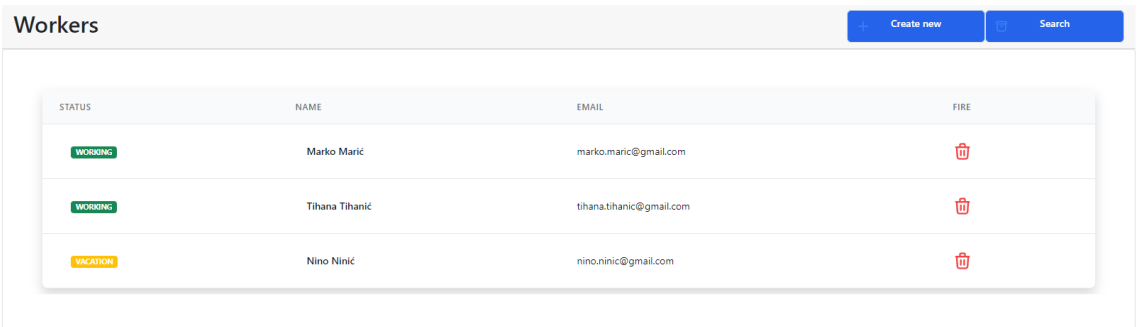
Slika 3.6 Izgled skočnog prozora za kreiranje zadataka.

3.2.3 Stranica s tablicom radnika

Nakon toga, ako šef želi vidjeti listu svojih radnika ili dodati novog u sustav, treba pogledati zaglavlje stranice te vidjeti gumb *Workers* sa slike 3.3. Kada pritisne na njega, aplikacija će ga odvesti na stranicu s radnicima prikazanu na slici 3.7. Ova stranica sadrži jednako zaglavlje kao i stranica s listom zadataka, no sadržaj je drugačiji.

Prvo što je uočljivo je tablica s listom trenutnih zaposlenika tvrtke. Ako se gleda s lijeva na desno uoči se stupac sa statusima. Prva dva reda sadrže status u zelenom *WORKING*, što znači da su oni trenutno dostupni i da rade normalno radno vrijeme, kao što se može vidjet na slici 3.4 gdje su dodijeljeni na zadatke. U trećem redu vidi se status pobojan žutom bojom *VACATION* koji naglašava da je ta osoba na godišnjem odmoru. Također, na slici 3.4 može se uočiti da zaposlenik Nino Ninić nije dodijeljen ni na jedan zadatak, jer je on na godišnjem odmoru.

U drugom stupcu se vide imena zaposlenika, koja se mogu prepoznati sa stupca *ASSIGNED* na slici 3.4, dok se u sljedećem stupcu vidi njihova e-adresa. Naposljetku se nalazi crveni koš za smeće, a naziv stupca je *FIRE* odnosno ovo je gumb za otpuštanje zaposlenika. Kad bi ga korisnik pritisnuo, izbrisao bi zaposlenika iz sustava i maknuo ga sa svih zadataka, te vratio zadatak u stanje *PENDING*, odnosno stanje čekanja.

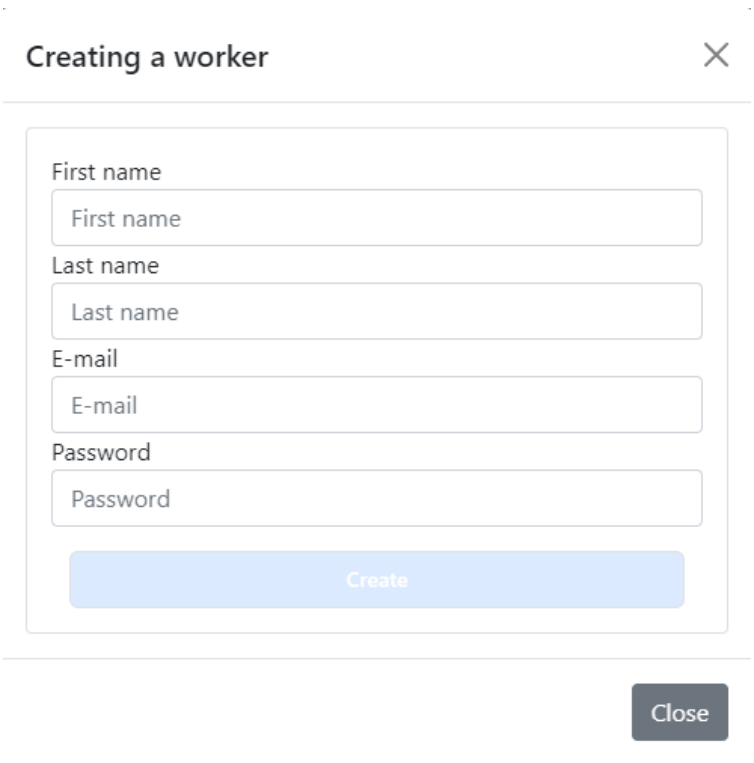


Slika 3.7 Izgled tablice s radnicima.

3.2.4 Stvaranje novog radnika

U gornjem desnom kutu na slici 3.7 se vide 2 gumba ista kao i na slici 3.4, no ova dva gumba umjesto za zadatke svoje funkcije obavljaju za radnike. Gumb *Search* pretraži bazu podataka za svim radnicima te ih prikaže u tablici, a gumb *Create new* otvori skočni prozor prikazan na slici 3.8. Cilj ovog prozora je omogućiti šefu da ubaci u sustav radnike iz njegove tvrtke kako bi mogli započeti sa svojim radom u sustavu.

Prva dva polja traže ime i prezime radnika, u trećem se traži e-adresa zaposlenika s kojom će se prijavljivati u sustav, a posljednje polje je šifra s kojom će se prijaviti. Nakon što su sva polja ispunjena, plavi gumb *Create* će prestati biti onemogućen te će pritiskom na njega radnik biti dodan u sustav. Ako šef ipak ne želi dodati nikog, može pritisnuti sivi gumb *Close* te izaći iz skočnog prozora i ponovno imati u fokusu tablicu sa svim radnicima i zaglavlje.



Slika 3.8 Izgled skočnog prozora za kreiranje radnika.

Na kraju rada šef će pritisnuti gumb *Logout* u zaglavlju i izaći iz sustava.

3.3 Pregled računa zaposlenika

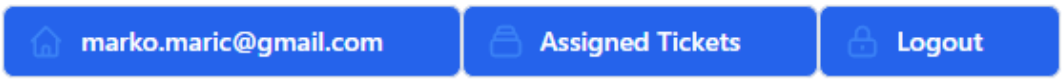
Kad se odjavi iz računa šefa, korisnik se ponovno nalazi na stranici za prijavu na slici 3.1, no ovog puta korisnik će se prijaviti s podacima zaposlenika, a ne šefa čime

će biti prebačen na početnu stranicu zaposlenika.

3.3.1 Početna stranica zaposlenika

Uspješno se prijavivši, zaposlenik će biti prebačen automatski na početnu stranicu za njegov profil. Na ovoj stranici prvo će biti primijećeno zaglavlje ekrana čiji se gumbovi vide na slici 3.3.

Nalaze se 3 opcije, prva je gumb s e-mailom prijavljenog zaposlenika. Ako se pritisne na njega, navigacija odvede subjekta na početnu stranicu na kojoj je trenutno. Drugi gumb odvede zaposlenika na stranicu gdje vidi sve zadatke koji su mu trenutno dodijeljeni, a posljednji gumb s lokotom te tekstom *logout* trenutnog korisnika odjavi iz sustava te ekran prebaci na ekran za prijavu na slici 3.1.



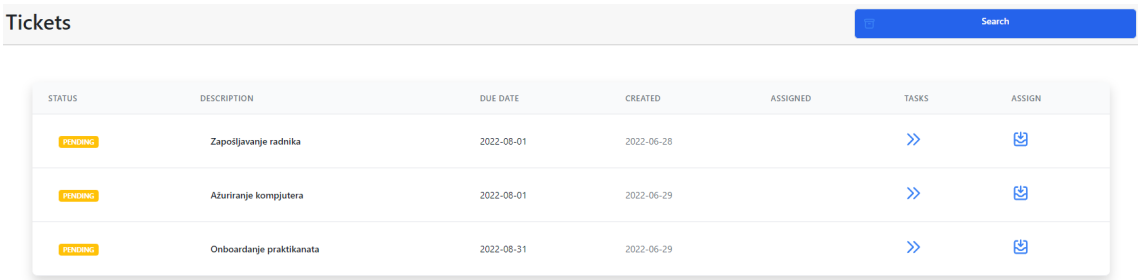
Slika 3.9 Gumbovi sa zaposlenikovog zaglavlja.

Ispod zaglavlja zaposlenik će primijetiti tablicu s dostupnim zadacima koje sebi može dodijeliti. U ovoj tablici svi statusi će biti žuto pobojani *PENDING*, jer se tu nalaze samo slobodni zadaci, odnosno oni koji čekaju na dodjeljivanje. Svi stupci su isti kao i na ekranu sa slike 3.4, osim zadnjeg.

Zadnji stupac naziva se *Assign* te ako se pritisne na gumb od nekog reda, zaposlenik će sebi dodijeliti taj zadatak, te taj zadatak više neće biti vidljiv na ovoj listi. Recimo da se korisnik dodijelio na zadatak s imenom "Onboardanje praktikanata", sustav će automatski ponovno pretražiti sustav za svim nedodijeljenim zadacima i maknuti onaj na koji se dodijelio korisnik.

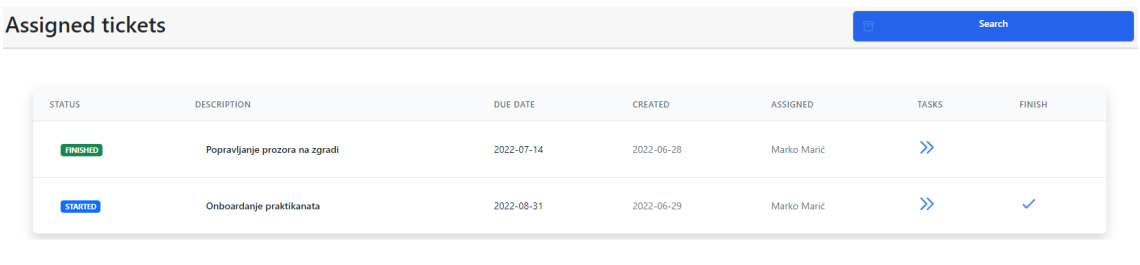
3.3.2 Stranica sa zaposlenikovim zadacima

Nakon što si je zaposlenik dodijelio zadatak, može preko zaglavlja na slici 3.9 pritisnuti gumb *Assigned Tickets* te se prebaci na ekran s njegovim zadacima prikazan



Slika 3.10 Izgled tablice s nedodijeljenim radnicima.

na slici 3.11. Ovdje se vide njegovi zadaci, jedan koji je zaposlenik već riješio sa zelenim statuom *FINISHED* te drugi zadatak na koji se zaposlenik upravo dodijelio u prošlom prozoru s plavim statusom *STARTED*.



Slika 3.11 Izgled tablice sa zaposlenikovim zadacima.

Ova tablica je dosta slična kao prijašnje dvije sa zadacima no razlikuju se dva zadnja stupca. Kada se pritisne na strelicu koja otvori skočni prozor za podzadatke, zaposlenik će vidjeti isti prozor kao na slici 3.5, no ako pritisne na zelenu kvačicu ili crveni X korisnik će promijeniti status završenosti podzadatka. Tek nakon što su svi podzadaci zatvoreni, moći će se završiti kompletni zadatak. Kada je završio sve podzadatke zaposlenik može pritisnuti plavu kvačicu u zadnjem stupcu tablice na slici 3.10, te će time završiti taj zadatak i status će biti prebačen u zeleni *FINISHED*.

Nakon završetka rada zaposlenik se može odjaviti od sustava preko *Logout* gumba u zaglavlju.

Poglavlje 4

Usporedba React i Angular radnih okvira

4.1 Programski jezik

Oba jezika su nastali na verzijama odnosno modifikacijama JavaScript programskog jezika.

Angular koristi TypeScript, koji je opisan u poglavlju 2.1. Odabir ovog jezika prisiljava programera da piše bolji kod koji se lakše provjeri za greške i da bude dobro održavan u budućnosti i jasniji za čitanje. TypeScript u praksi se može vidjeti na isječcima koda 2.2 i 2.3

React se s druge strane drži JavaScripta uz sintaktičko produženje nazvano JavaScript XML (JSX) koji omogućuje fleksibilno korištenje JavaScripta za definiranje što će se i kada renderirati. Na primjer, to se može vidjeti i u kodnom isječku 4.1.

Vidi se da se koristi JavaScript kod tako što se konstanta *content* provjerava te ako je ona jednaka *stringu home* onda će se renderirati React komponenta `<BossTickets/>`. To je nešto što nije moguće odraditi u ovakvom formatu bez JSX-a. Bilo kakav JS kod se može ubaciti između HTML-a ako ga ubacimo u vitičaste zagrade.

Iako je JSX koristan svejedno mu fali statičko provjeravanje tipova opisano u poglavlju 2.1, no pošto je React vrlo modularan i fleksibilan napravljena je TS verzija

```
...
<>
  <BossHeader/>
  {content === "home" && <BossTickets/>}
  {content === "workers" && <BossWorkers/>}
  ...
</>
```

Isječak koda 4.1 Primjer TSX-a u aplikaciji

JSX-a nazvana TSX koja radi identično samo omogućuje korištenje TS sintakse i sad je kod u React aplikaciji pisan upravo u TSX datotekama.

4.2 Renderiranje

React funkcionira tako da ima korijenski čvor negdje u HTML datoteci i sve što se nalazi unutar tog čvora će biti vođeno od strane Reactovog Virtualnog DOM-a. Početak React projekta i stvaranje korijenskog čvora u kodu izgleda kao na isječku 4.2. Programeri koji su koristili druge radne okvire kao što je Angular su navikli na

```
import React from 'react';
import { createRoot } from 'react-dom/client';
import { Provider } from 'react-redux';
...
const container = document.getElementById('root')!;
const root = createRoot(container);
root.render(
...
```

Isječak koda 4.2 Primjer korijenskog čvora Reacta

klasični `index.html` koji su sebi sadrži HTML-ove oznake *html*, *head* i *body* koji su izvor za sve u projektu, no u Reactu je to sve odrađeno komandom `createRoot()`

koja u sebi prima HTML element s identifikacijom *root* i vrati Reactov element u kojem se razvija cijela React aplikacija. `root.render()` komanda naglašava da se tu počinje renderirati aplikacija.

Za svaki DOM postoji virtualni DOM koji ima ista svojstva kao obični DOM samo što nema moć za mijenjanjem onoga što je na ekranu. Pošto virtualni DOM ništa ne crta na ekran njega se može brzo i efikasno mijenjati, može se to zamisliti kao shematski plan koji još nije izvršen u praksi. On funkcionira tako da kada se izmjeni neki TSX element, on se cijeli ažurira, nakon toga usporedi samog sebe sa starijom verzijom da sazna koje stvari su izmijenjene, i onda u stvarnom DOM-u ažurira samo stvari koje su izmijenjene, čime se vrijeme puno skrati, jer nije potrebno ponovno crtati cijeli ekran već samo dijelove koje su izmijenjeni [5].

S druge strane Angular ima klasični korijenski čvor `index.html` u kojem se mogu prepoznati svi dijelovi klasične HTML datoteke. Oni su oznake `<html>`, `<head>` i `<body>`. U `<body>` oznaci se nalazi početak aplikacije.

Nasuprot Reactu, sve što se nalazi u projektu je automatski dio Angular radnog okvira, dok u Reactu samo ono što se nalazi pod korijenskim čvorom [5]. Zbog tog se u React radnom okviru može imati više zasebnih React aplikacija koje svaka imaju svoj zasebni korijenski čvor [5]. U Angularu to nije slučaj, manje je fleksibilan i nije kompatibilan s drugim opsežnijim radnim okvirima [6].

Izgled `index.html` u Angular aplikaciji se vidi na isječku koda 4.3.

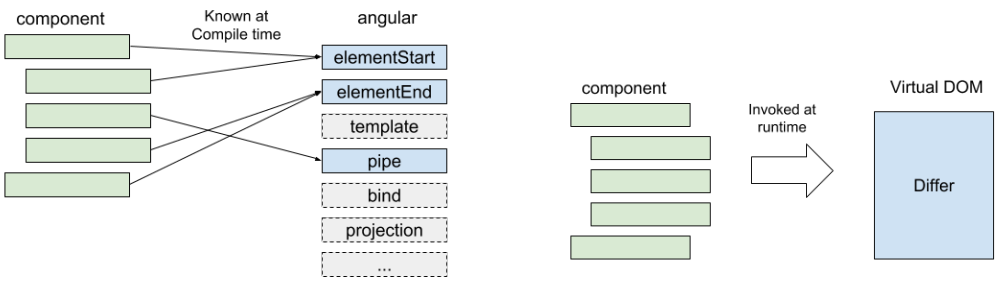
```
1 <!doctype html>
2 <html lang="en">
3 <head>
4 ...
5 </head>
6 <body>
7   <app-root></app-root>
8 </body>
9 </html>
```

Isječak koda 4.3 Primjer index.html datotke iz Angular aplikacije

`app-root` zapravo je prvi dio koda aplikacije i on se poziva u `body` oznaci (eng. tag).

Nakon što je React tim predstavio virtualni DOM, Google se odlučio za svoje rješenje, a to je Inkrementalni DOM. On funkcionira na skroz drugačiji način, tako da sve komponente budu sastavljene u niz uputa [7]. Te upute rade DOM stabla i ažuriraju ih na mjestu kada se podaci promijene. Razlog zašto nisu koristili virtualni DOM je taj što je cilj bila optimizacija za mobilne uređaje što znači da su veličina aplikacije i otisak memorije jako bitne stavke [7].

Angularov DOM to postiže tako da radni okvir ne tumači komponentu, već komponenta referencira instrukcije, a ako se neka instrukcija ne referencira nigdje, ona neće biti korištena, a to se sve zna u vremenu sastavljanja koda čime se onda nekorištene instrukcije mogu izbaciti iz paketa i smanjiti veličinu [7]. S druge strane virtualni DOM ima potrebu kompletnog tumača jer se ne zna koji dijelovi će biti potrebi za vrijeme sastavljanja, što znači da se cijeli paket mora staviti u pretraživač [7]. Vizualni primjer virtualnog i inkrementalnog DOM-a se vidi na slici 4.1.



(a) Primjer inkrementalnog DOM-a. (b) Primjer virtualnog DOM-a.

Slika 4.1 Primjeri tumačenja komponenti.

Inkrementalni DOM štedi mnogo radne memorije tako da alocira memoriju samo kad se DOM čvorovi dodaju ili uklone. Također, veličina alokacije je proporcionalna promjeni, pošto većina poziva na renderiranje ne mijenjaju ništa ili vrlo malo. Posljedica toga je u velika štednja u memoriji [7].

Nasuprot tome virtualni DOM kreira stablo od nule svaki put kad se rerenderira

element, no to stablo ne crta ništa kao što je prije navedeno, pa ne zauzima toliko memorije, no svejedno više [7].

4.3 Komponente

Kao što je navedeno u poglavlju 2.8, Angularove komponente su podijeljene u 3 do 4 dijela, od kojih će fokus biti na HTML šabloni i TS klasi, jer korištenjem Tailwind CSS-a, eliminirana je potreba za CSS dijelom komponente.

Cijela komponenta, odnosno njezini dijelovi su definirani u TS klasi te se anotacijom `@Component` iznad definicije klase ona bude označena kao komponenta i time ju Angular računa kao takvu i daje joj posebna svojstva. Definiciju se može vidjeti na isječku koda 4.4.

```
@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {
```

Isječak koda 4.4 Primjer definiranja TS klase komponente Angulara

Vidi se anotacija te u njoj postoje dodatne postavke, kao što je *selector* koji označava kako će izgledati HTML oznaka za ovu komponentu. Nadalje, sljedeća postavka je *templateUrl* koja definira lokaciju HTML šablone, a posljednja je *styleUrls* koja pokazuje na lokaciju CSS datoteke za stil.

Nasuprot tome, Reactove komponente, da bi radile, moraju biti pod definiranom korijenskim čvorom te sadrže samo jedan dio, a to je TSX datoteka u kojoj je sva logika renderiranja i HTML sučelje. Za stvaranje komponente nije potrebna nikakva posebna sintaksa već dovoljna jednostavna TypeScript funkcija [5] poput ove u isječku 4.5. Dovoljno je da ta funkcija vraća Reactov element, a možemo vidjeti da vraća `<div>` što znači da je ona zaista validna komponenta. Kada

```
export default function Header({actions}: HeaderProps) {
    return (
        <div>
            ...
        </div>
    );
}
```

Isječak koda 4.5 Primjer definicije React komponente

bi se htjela pozvati unutar nekog drugog React elementa, ona bi izgledala ovako `<Header actions={headerActions}/>`

Header prima svojstva koja su *HeaderProps* sučelje, a u tom sučelju se nalazi *actions* vrijednost koju onda koristi komponenta *Header*. Svojstva (eng. props) su način na koji se u Reactu podaci od roditeljske komponente šalju u dječju,a jedno bitno pravilo je da se ne smiju mijenjati u dječjoj komponenti, odnosno oni su samo za čitanje (eng. read-only). U isječku 4.6 se vidi korištenje vrijednosti *actions* koje je poslano u komponentu *Header*.

```
...
{
    actions?.map((action) => (
            <NavbarText>
                <AppButton key={action.key}title={action.title
   }
                ={action.onClick} icon={action.icon}/>
            </NavbarText>
      ))
}
...
```

Isječak koda 4.6 Korištenje svojstva (eng. props) u Reactu

actions zapravo lista sučelja *HeaderAction*, te ta lista iterira i stvara elemente `<NavbarText>` u kojem se nalazi element `<AppButton>` koji je ručno napravljen za aplikaciju i prima svojstva *key*, *title*, *onClick* i *icon* koja se nalaze u *HeaderAction* sučelju.

Angularov način slanja podataka roditelja djetetu ne koristi svojstva već anotaciju `@Input` . Na isječku 4.7 je prikazano korištenje anotacije za primanje vrijednosti `ticketId` u komponenti *TaskModalComponent*.

```
@Component({
  selector: 'task-modal',
  templateUrl: './task-modal.component.html',
  styleUrls: ['./task-modal.component.css']
})
export class TaskModalComponent implements OnInit {
    ...
  @Input() ticketId: any;
  ...
}
```

Isječak koda 4.7 Korištenje anotacije Input u Angularu

Varijabla koja se anotira s *@Input* se šalje djetetu od roditeljske komponente putem koda 4.8.

```
    <task-modal [ticketId]="currentId"></task-modal>
```

Isječak koda 4.8 Slanje podataka djetetu u Angularu

Kad bi se ta varijabla koristila u HTML šabloni pozvala bi se korištenjem vitičastih zagrada kao u isječku koda 4.9. Onda bi automatski bila postavljena na koju god vrijednost dodijeljenu od roditeljske komponente.

Nadalje, slanje podataka s djeteta roditelju malo je kompliciranije a radi se tako da se koristi anotacija `@Output` s kojom se anotira varijabla koja se želi slati te se

```
<div>id: {{ticketId}}</div>
```

Isječak koda 4.9 Korištenje varijable iz TS klase u HTML šabloni

varijabla postavi kao `EventEmitter` klasa koja omogućuje da se podaci emitiraju roditelju [6]. U kodu se to vidi na isječku 4.10.

```
@Output() isTicketFinished = new EventEmitter<boolean>();

finishTicket(value: boolean) {
    this.isTicketFinished.emit(value);
}
```

Isječak koda 4.10 Primjer anotacije Output u Angularu

Putem funkcije `finishTicket()` emitira se vrijednost koji je primiljena i time će roditeljska komponenta to pokupiti. Kako bi ovo funkcioniralo u roditeljskoj HTML šabloni također kada se zove dječja komponenta mora se naglasiti da se koristi događaj (eng. event) koji će emitirati podatke, a to se vidi u isječku 4.11.

```
<task-modal (isTicketFinished)="closeModal($event)"
```

Isječak koda 4.11 Primjer korištenja EventEmitter klase u Angularu

U funkciji `closeModal()` bit će primljen *boolean* koji je poslala dječja komponenta.

React ponovno ima jednostavniji način slanja podataka s djeteta roditelju, a radi to tako da proslijedi funkciju kroz svojstva (eng. props) dječjoj komponenti te dijete onda može pozvati tu funkciju i u njoj poslati podatke koji onda stignu u roditeljskoj komponenti te s njima može manipulirati kako želi. U roditeljskoj komponenti to izgleda kao na isječku 4.12.

U dječju komponentu `<TicketTable>` šalje se funkcija `onTasks()` koja prima vrijednost *e* bilo kojeg tipa, u dijelu `setTicketId(e.currentTarget.value);` se

```
export default function BossTickets() {
    ...
    const onTasks = (e : any) => {
        setShowTasks(true);
        setTicketId(e.currentTarget.value);
    }
    return(
        ...
        <TicketTable tickets={tickets} onClick={onTasks} />
        ...
    )
```

Isječak koda 4.12 Primjer slanja podataka od djeteta roditelju u Reactu

vidi da je *e* zapravo događaj od gumba, iz kojeg se onda izvuče vrijednost koja mu je pridodana, a u dječjoj komponenti se i vidi taj gumb na isječku 4.13.

```
    <button value={ticket.id} onClick={onClick}>
        ...
    </button>
```

Isječak koda 4.13 Primjer funkcije kad se klikne gumb u Reactu

Vidi se da je *value* postavljen na *ticket.id*, te da se funkcijom `onClick()` šalje događaj da je gumb pritisnut čime šalje sve o tom događaju uključujući i vrijednost postavljenu u *value* koju se onda koristi u roditeljskoj komponenti. Naravno, ovo se ne mora raditi preko događaja gumba, već preko obične funkcije u kojoj se proslijedi vrijednost, no ovo je primjer iz rađene aplikacije.

Komponente oba radna okvira odrađuju svoj posao efikasno. Iako je Reactov način jednostavan i fleksibilan može uzrokovati mnogim pogreškama koje su se i događale tijekom razvoja aplikacije. Kada se nauči TSX postane vrlo intuitivan i kreiranje komponenata se pretvori u jednostavan zadatak bez nepotrebnog koda i sve je jednostavno za spajati i dijeliti podatke. S druge strane Angular je vrlo

eksplicitan te se sve definira s anotacijama i odredi se gdje će što ići i gdje će se primati, što zna biti naporno, jer se piše šablonski kod, ali uzrokuje manje grešaka i jasno je. Podjela komponenata na više dijelova također smanjuje miješanje koda i rezultira čišćim kodom.

4.4 Upravljanje stanjem podataka (eng. Data state management)

Upravljanje stanjem način je za stvaranje komunikacije i dijeljenja podataka među komponentama. Stvara konkretnu podatkovnu strukturu koja predstavlja stanje aplikacije koje se može čitati i pisati. U najjednostavnijoj definiciji, Stanje je JavaScript objekt koji predstavlja dio komponente koji se može mijenjati na temelju rezultirajuće radnje korisnika. Također se može reći da su stanja jednostavno memorija komponente [8].

U Reactu se upravljanje sa stanjem može odraditi na više načina. Postoji bezbroj biblioteka koje pružaju svoja rješenja za upravljanje stanjem, no u ovom radu fokus će biti na 2 rješenja koja su korištena u aplikaciji, a to su Redux biblioteka i Reactove kuke koje već i same dolaze s Reactom.

Postoje 3 kuke za upravljanje stanjem a to su *useState*, *useReducer* i *useContext*. useState i useReducer se koriste za lokalno upravljanje stanjem podataka, dok je useContext za globalno [8]. *useState* i *useReducer* se zovu unutar funkcijskih komponenti te omogućuju da se stanje nekog podatka ne izgubi između renderiranja. Kuka useState vraća par: trenutno stanje i funkciju koja omogućuje da se ažurira to stanje [8], primjer tog se vidi na isječku 4.14.

```
1     const [tasks, setTasks] = useState<Task[]>([]);
```

Isječak koda 4.14 Primjer korištenja useState kuke

tasks je trenutno stanje, a *setTasks* je funkcija s kojom se ažurira to stanje. U kuki *useState* definira se da je podatak tipa polje *Task* tipova, te u zagradama se

definira da je inicijalno stanje prazno polje. Ažuriranje tog stanje se može vidjeti na isječku 4.15.

```
1    const response = api.getTicketTasks(ticketId);
2    response.then((newTasks) => setTasks(newTasks))
```

Isječak koda 4.15 Ažuriranje stanja u useState kuki

Dobije se odgovor s poslužitelja te se nakon toga on koristeći funkciju *setTasks* postavi u trenutno stanje.

useReducer funkcionira vrlo slično, no, umjesto da prima samo inicijalno stanje, on prima reduktorsku funkciju i inicijalno stanje, te vraća novo stanje. S tom funkcijom puno je lakše manipulirati kompleksnim objektima, jer se u reduktorskoj funkciji može primiti vrijednost s kojom se odredi što će biti novo stanje.

useContext je posljednja kuka za upravljanje stanjem te je drugačija od prijašnje dvije, jer se može koristiti kroz više komponenata bez da se šalje kroz svojstva svake komponente. Ona funkcionira tako da se napravi u nekoj roditeljskoj komponenti, ili ako se želi da bude globalna onda u korijenskoj, te joj se da neka vrijednost koju kasnije preko kuke bilo koja komponenta ispod može iskoristiti. U kodu to izgleda kao na kodnom isječku 4.16. Vidi se korištenje funkcije `createContext()` koja stvara

```
1    const UserContext = createContext();
2    ...
3    return(
4    <UserContext.Provider value={api.getCurrentUser()}>
5        ...
6    </UserContext.Provider>
7    )
```

Isječak koda 4.16 Primjer stvaranja React konteksta

kontekst, no da se odredi gdje počinje taj kontekst mora ga se iskoristiti kao React element u HTML oznaci i dodati *Provider* što znači da ova oznaka pruža kontekst svim dječjim komponentama ispod nje.

Kada neka dječja komponenta želi koristiti kontekst mora to napraviti preko `useContext()` kuke, kao što se vidi na isječku 4.17.

```
1     const currentUser = useContext(UserContext)
```

Isječak koda 4.17 Primjer korištenja useContext kuke

Nakon toga, vrijednost koja je spremljena u kontekstu, postavlja se u lokalnu varijablu te se s njom može raditi što god. Kada se vrijednost koja je postavljena u *Provideru* promijeni, ona se aplicira na sve instance gdje se koristi kontekst.

Redux je biblioteka koja se često koristi uz React te joj je glavni cilj omogućiti jednostavno upravljanje stanjem na jednom mjestu [10]. Funkcionira na principu skladišta (eng. *store*) koje sadrži sva stanja u aplikaciji koja su stavljena u njega i ažurira ih kad je potrebno, ono u sebi sadrži reduktore (eng. *reducers*) odnosno funkcije koje ažuriraju stanje u skladištu s obzirom na akciju koju prime [10]. Za korištenje Reduxa prvo je potrebno namjestiti skladište (eng. store) i korijenski reduktor (eng. reducer) koji u sebi sadrži sve reduktore korištene u aplikaciji, to se vidi u isječku 4.18

```
1 export const rootReducer = combineReducers({
2     bossTickets,
3     bossWorkers,
4     assignedTickets,
5     workerTickets
6 });
```

Isječak koda 4.18 Primjer stvaranja korijenskog reduktora

nakon što se namjesti korijenski reduktor mora se smjestiti u skladište, a to se radi prema kodnom isječku 4.19. Nakon toga za svako mjesto gdje se koristi reduktor potrebno je napraviti krišku (eng. slice), a to je funkcija koja prihvaća početno stanje, reduktorske funkcije i ime te automatski generira tipove akcija i stvaratelje akcija koji odgovaraju danim reduktorskim funkcijama i stanju [10]. Uz to koristi se funkcija `createAsynchThunk()` koja prima tip akcije koji će služiti u kriški za generiranje

```
    const store = configureStore({
    reducer : rootReducer
});
```

Isječak koda 4.19 Primjer stvaranja Redux skladišta

tipova akcija, te funkciju povratnog poziva koja će vratiti obećanje (eng. promise) jer je asinkrona [10]. U aplikaciji se ta funkcija koristi za izradu API zahtjeva koji onda vraćaju obećanje za podatke. S obzirom na stanje zahtjeva u aplikaciji se postavlja status u reduktorskim funkcija i sprema u store, koji onda osigurava da se stranica ne učita prije nego što dođu podaci [10]. Redux u primjeni izgleda kao na isječku 4.20.

```
    const dispatch = useDispatch();

    const {tickets} = useSelector(
    (state: RootState) => state.bossTickets.result);
    const {ticketsStatus} = useSelector(
    (state: RootState) => state.bossTickets.status);

    useEffect(() => {
        dispatch(actions.getAllTickets())
    }, [dispatch]);
```

Isječak koda 4.20 Primjer korištenja Reduxovih kuka

U `useEffect()` kuki koja se pokrene kad se učita stranica, koristi se Reduxova kuka `useDispatch()` koja omogućuje da se pokrene akcija iz kriške, u ovom slučaju da se dohvate podaci o zadacima s API-ja, nakon što se pokrene zahtjev za podacima, koristi se kuka `useSelector()` koja iz skladišta dohvaća rezultate zahtjeva koji su tako spremljeni [10].

S druge strane Angular za lokalno stanje sve sprema u TS klasu svake komponente, stoga nije potrebno koristiti nikakve dodatne kuke ili funkcije već sve što

definiramo u klasi može se koristiti u HTML šabloni, te za jednostavnije prenose podataka s roditelja na dijete ili obrnuto koriste se prije navedene `@Output` i `@Input` anotacije.

Uz to koriste se servisi koje mogu koristiti više komponenti i injektibilni su. To se radi preko RxJS biblioteke koja u sebi sadrži klasu `BehaviorSubject()`,a potrebno je napraviti servis koji će u sebi sadržavati kod poput ovog u kodnom isječku 4.21.

```
1 private task$ = new BehaviorSubject<any>({});
2 constructor() {}
3
4 setTask(task: any) {
5     this.task$.next(task);
6 }
```

Isječak koda 4.21 Primjer Angularove Behavior Subject klase

BehaviorSubject klasa funkcionira tako da se na nju preplati (eng. subscribe). Nakon toga bilo kakve promjene na klasi će se automatski primijeniti na svakom mjestu koje je preplaćeno [9]. Koristeći `setTask()` funkciju, koju može pozvati bilo koja komponenta koja ima injektiran servis, može se mijenjati vrijednost varijable *task* tako što se koristi funkcija `next()` u koju se ubaci vrijednost koja treba postati novi *task* [9]. Korištenjem ovakvih servisa može se postići globalno upravljanje stanjima, ali takav način upravljanja zahtjeva kompleksno programiranje i optimizaciju, a postoje naprednije biblioteke koje rade posao za programera.

Kada nastane potreba za opsežnijim upravljanjem stanjima, za Angular se kao i za React može koristiti više vanjskih biblioteka, ali za usporedbu će biti analizirana biblioteka NgRx za upravljanje globalnim stanjem koja je inspirirana Reduxom i prilagođena za Angular.

Koristi se ista logika kao u Reduxu, sa skladištem, reduktorskim funkcijama, selektorima i akcijama. Za svako stanje s kojim se želi upravljati globalno, potrebno je napraviti reduktor, selektor, i akcije u zasebnim datotekama [11]. U datoteci s akcijama se definiraju tipovi akcija koji se zovu te onda s obzirom na tip akcije u reduktoru se pozove funkcija koja dohvaća podatke iz stanja i ažurira ih [11]. Također

je potrebno napraviti datoteku koja će držati stanja te selektorske datoteke s kojima se definira što će se pozvati iz stanja [11]. Kada se to sve napravi, u komponenti u kojoj se želi koristiti skladište ono se definira u konstruktoru. Time se može koristiti funkcije `select()` koja omogućuje preko selektora da dohvatimo podatak iz skladišta, te `dispatch` koja omogućuje da se pokrenu akcije na primjer dohvaćanje podataka s API-ja [11].

Sve u svemu React sa svojim nativnim bibliotekama ima rješenje za lokalno i globalno stanje, iako za globalno stanje može postati dosta neuredno, jer konteksti se mogu nalaziti po svuda, i nisu na jednom mjestu. Nasuprot toga Angular sa svojim vlastitim bibliotekama ima rješenje za lokalno stanje sa svojim servisima koji se mogu dijeliti među komponentama i s TS klasama koje drže podatke, a za globalno stanje mogu se koristiti servisi sa *BehaviorSubject* klasom, ali najčešće se koristi NgRx biblioteka. U Reactu zbog malo primitivnijeg načina globalnog stanja također se koristi vanjska biblioteka Redux koja efikasnije rješava problem, no sadrži mnogo šablonskog koda koji se mogao vidjeti u objašnjenju.

4.5 Forme

Nakon uspoređivanja tehničkih aspekata oba radna okvira, potrebno je i analizirati konkretne primjene koje će se često koristiti, a to su forme.

Primjer forme u Angularu se može vidjeti na sljedećim primjerima. Npr. na HTML šabloni se napiše isječak koda 4.22.

U primjeru se vidi polje za unos e-mail vrijednosti, definirano je ime forme, funkcija *onSubmit* koja će se pozvati kada se preda forma te kako će izgledati kada je validacija neuspjela. Definicija direktive *ngModel* označava da se ovo polje veže za objekt loginData i označava da se polje *email* veže za objekt loginData odnosno atribut *email* od tog objekta.

loginData vezan za sučelje *Login* i ima u sebi dvije vrijednosti *email* i *password*. Pošto su oni vezani korištenjem direktive, svaka izmjena polja e-mail na sučelju izmjeni ga i u modelu u klasi. Na taj način, ako se manipulira vrijednost u TS klasi odrazit će se i na sučelju.

```
<form (ngSubmit)="onSubmit()" #loginForm="ngForm">
    <div class="form-group">
                <label>Email</label> <input
                type="email" class="form-control"
                formControlName="email"
                [(ngModel)]="loginData.email" required
                >
                <div *ngIf="email?.invalid &&
                    (email?.dirty || email?.touched)"
                     class="alert alert-danger">

                  <div *ngIf="email?.errors?.['required']">
                    E-mail is required.
                  </div>
    </div>
    ...
</form>
```

Isječak koda 4.22 Primjer koda s HTML šablone Angular komponente

Nadalje, implementirana je i validacija koja se može vidjeti ispod `<input>` oznake gdje se provjerava ako je polje *email* dotaknuto te ako je nevaljano, u tom slučaju se pojavi crveni tekst koji objašnjava korisniku da polje nije ispunjeno, a način na koji se to provjerava se nalazi u TS klasi i to se vidi na primjeru isječka koda 4.23.

S *FormGroup* klasom se prati cijela forma i njezin status. Ako je išta u toj formi nevaljano, onda se može onemogućiti korištenje gumba s kojim se podnese forma. S *FormControl* klasom se prati pojedini dio te forme te s njom se mogu postavljati validacije, kao što se vidi s `Validators.required` validacijom koja diktira da ovo polje mora biti ispunjeno. Pošto je forma dvosmjerno spojena na varijablu *loginData* kada ju podnesemo pošalje se *loginData* kao tijelo zahtjeva.

U Reactu ima mnogo različitih biblioteka za validaciju i forme, no dvije napopularnije su Formik za forme i Yup za validaciju tih formi, i međusobno su dobro integrirani. Formik omogućava lako postavljanje inicijalni vrijednosti , te dopušta

```
loginForm: FormGroup;

ngOnInit() {
    this.loginForm = new FormGroup({
        email: new FormControl(this.loginData.email, [
            Validators.required,
        ])
    });
}
```

Isječak koda 4.23 Primjer korištenja klase FormGroup

da se ubaci Yup validacija preko *validationSchema* opcije u oznaci za Formik [12].

Primjer forme se vidi na kodnom isječku 4.24, te se vidi da se Yup objekt stvori sa `Yup.object().shape()` funkcijama te ga se napuni s validacijama za polja za unos koja postoje u formi [13]. Tako za polje *email* postoji validacija *required* koja naglašava da ako to polje nije ispunjeno validacija nije uspješna.

Nakon što se ispuni validacijska shema ona se predaje Formiku koji ju onda mapira na polja u formi, te preko TSX-a se ulazi u funkciju koja od Formika prima 3 varijable: *touched* koja za svako polje provjeri ako je taknuto, *errors* koji sadrži greške od svakog polja i *values* koji sadrži upisane vrijednosti svakog polja [12]. React komponenta `<InputField>` je napravljena u aplikaciji i prima greške. Ako je polje dotaknuto te, ako je validacija javila grešku, onda ispiše crvenim slovima grešku. Nakon što se podnese forma Formik sve vrijednosti pošalje u funkciju koja je postavljena za predaju, u slučaju aplikacije to je *onSubmit* funckija.

Uzevši u obzir oba radna okvira, može se zaključiti da su forme jednostavne u oba radna okvira i lagane za složiti i koristiti. Angularu se daje prednost jer ima uključene intuitivne forme u radnom okviru, no biblioteke koje se koriste za React su također odlično optimizirane i rade efikasno i jednostavno.

```
const ValidationSchema = Yup.object().shape({
    email: Yup.string().required("Field is required")
    ...
    })
    ...
<Formik initialValues={loginData} onSubmit={onSubmit}
    validationSchema={ValidationSchema}>
            {({touched, errors, values}) => {
                return (
                    <Form >
                        <div className="text-left ">
                            <InputField name="email"
                            touched={touched.email}
                            errors={errors.email}
                            type="email" label="E-mail"/>
                            ...
                        </div>
                    </Form>
</Formik>
```

Isječak koda 4.24 Primjer Formika i Yupa u React formi

4.6 Zahtjevi na API

Zahtjevi na API su jedan od najbitnijih dijelova web aplikacije za radni okvir koji se bavi sučeljem. Stoga, potrebno je analizirati koji od radnih okvira je bolji.

React dopušta slobodu korištenja koje god biblioteke za rađenje zahtjeva na API, stoga za što manje kompliciranja u aplikaciji je korišten *fetch*, primjer je vidljiv na isječku 4.25

Ova asinkrona funkcija prima podatke te vraća obećanje (eng. *promise*) klase *Response*. U *fetch* funkciju šalje se adresa na koju se želi poslati zahtjev, te razne dodatne postavke kao metodu slanja, tijelo u kojem se šalju podaci, te zaglavlja (eng. *headers*) u kojima se definira JWT token za autorizaciju, i tip sadržaja [14]. Ono

```
export async function createWorker(data: CreateWorker)
    : Promise<Response> {
    return await fetch("http://localhost:8888/employee/create",
    {
        credentials: "include", method: "POST",
        body: JSON.stringify(data),
        headers: {
            "Authorization": "Bearer " + token,
            "Content-Type": "application/json"
        }
    });
}
```

Isječak koda 4.25 Primjer korištenja fetch funkcije

što je vraćeno se vidi na kodnom isječku 4.26.

```
    try {
            const response = await api.createWorker(values);
            if(response.status === 200) {
                handleUpdate();
            }
        } catch (error) {
            setError(error)
        }
```

Isječak koda 4.26 Primjer odgovora na zahtjev u Reactu

Vidi se da se vraća klasa *Response* koja u sebi ima atribut status. Ako je zahtjev uspješan pokrene se funkcija `handleUpdate()`, no ako je pronađena greška u stanje komponente se spremi ta greška. Ako zahtjev vraća neku vrijednost u tijelu onda da bi se to dobilo koristeći funkciju `response.json()` koja vraća tijelo mapirano na TS objekt [14].

S druge strane Angular koristi ugrađeni HTTP API koji se koristi kao Angula-

rov servis te se injektira u konstruktor od klase neke komponente i to omogućuje korištenje. U kodnom isječku 4.27 se vidi korištenje HTTP servisa kako bi se napravio zahtjev za svim radnicima od API-ja. Funkcija vraća klasu *Observable* koja omogućuje pretplatu te time pošto je asinkrona funkcija se mogu čekati podaci.

```
getAllWorkers() : Observable<Worker[]> {
    return this.http.get("http://localhost:8888/employee/list"
    {
    withCredentials: true,
    headers: {
            "Authorization":
                "Bearer " + token
        }
    });
    }
}
```

Isječak koda 4.27 Primjer zahtjeva koristeći Angularov HTTP API

Nakon što je zahtjev uspješan, na kodnom isječku 4.28 se u povratnoj funkciji od funkcije `subscribe()` dobiju podaci koji se onda postave na atribut *workers* čime su dostupni na HTML šabloni. Ukoliko zahtjev nije uspješan unutar `subscribe()` povratničke funkcije se vrati greška koju se onda riješi. U slučaju aplikacije se doda u polje grešaka, kako bi se kasnije prikazalo na sučelju.

Sve u svemu, Reactu su dostupni mnogi vanjski alati, kao i ugrađena `fetch()` funkcija, te je sintaksa vrlo jednostavna i korištenje treba vrlo malo navikavanja.

Angularov HTTP servis za slanje zahtjeva je malo drugačiji i za njegovo učenje je potrebno nešto vremena kako bi se shvatio princip klase *Observable* te kako funkcionira pretplata na nju, no kad se to shvati, pisanje zahtjeva postane jednostavno i brzo.

```
1    const response = this.workerService.getAllWorkers();
2    response.subscribe(
3    (data) => {
4        this.workers = data;
5    },
6    (error) => {
7        this.errors.push(error);
8    }
9    )
```

Isječak koda 4.28 Primjer korištenja RxJs subscribe funkcije

4.7 Proces učenja radnog okvira

Što se tiče brzine učenja, opet treba sagledati da se React smatra bibliotekom, dok je Angular kompletni radni okvir. S obzirom na to usporedba će se svejedno baviti teškoćom rađenja kompletne jednostranične web aplikacije.

Angular na svojoj službenoj stranici ima vrlo dobru dokumentaciju što se tiče svih dijelova za rađenje aplikacije. Kroz proces učenja gradi se web aplikacija koji koristi glavna svojstva Angulara i na kraju učenik ima gotovo sučelje web aplikacije pripremljeno za API. Za tu aplikaciju ima i potpuni kod koji se može preuzeti i biti od pomoći za gradnju vlastite aplikacije.

Glavna teškoća kod učenja je na početku kada se treba razumjeti način na koji komponente rade i kako funkcionira spoj TypeScript klase i HTML šablone. Iako se na prvu čini teško, ljudima s iskustvom u OOP i TypeScriptu stvari postanu jasne relativno brzo. Sljedeći problem na koji se naiđe su direktive koje su jedinstvene za Angular i potrebno ih je proučiti i razumjeti. Na primjer dođu do koristi kod rađenja formi i vrlo su korisne u toj svrsi za dvostrano vezivanje podataka (eng. two-way data binding).

Suprotno od Angulara, React ne pruža rješenja za svaki dio web aplikacije već samo daje alate za gradnju komponenata i optimiziranost. Iz tog razloga učenje optimalnih rješenja i pravilne sintakse i načina programiranja u Reactu je znatno

teže, jer postoje mnogo različitih biblioteka i rješenja za svaki problem, stoga odabir pravih uzme dovoljno vremena. Uz to učenje tih radnih okvira isto uzme vremena.

Osim toga, Reactove kuke (eng. hooks) nemaju deskriptivna imena te često nisu skroz jasne što rade i gdje se trebaju koristiti, a službena dokumentacija ne daje dovoljno primjera i objašnjenja u jednostavnim terminima. Na primjer za forme u aplikaciji su korištene dvije zasebne biblioteke koje dobro rade uz React, a to su Formik i Yup. Formik olakšava korištenje formi, a Yup omogućuje jednostavnu validaciju i dobro je integriran s Formikom, što je i opisano u poglavlju 4.5.

Sve u svemu Angular se pokazao bolje strukturiranim za učenje i ulazak u ekosustav nego React, koji je jednostavan za započeti, no što se više kopa po bibliotekama, treba biti svjestan što se koristi i što je zapravo potrebno a što ne. Iako, u slučaju upravljanja stanja podataka, oba radna okvira za kompleksnije projekte često koriste dodatne knjižnice, React koristi Redux ili neku sličnu opciju, a Angular koristi NgRx koji je vrlo sličan Reduxu.

Poglavlje 5

Zaključak

Napravljena aplikacija uspješno je istestirala radne okvire Angular i React te prikazala njihove prednosti i mane kao i omogućila analizu njihovih svojstava. Iako se često uspoređuju, Angular i React su u svojim temeljnih konceptima vrlo drugačiji. Angular se tretira kao potpuni radni okvir, dok se React često svrstava kao programska biblioteka više nego radni okvir.

Za tu razliku ima mnogo temelja jer dok Angular nudi svoja rješenja na gotovo sve probleme na koje programer može naići u stvaranju jednostranične web aplikacije, React daje alate s kojima se može stvarati aplikaciju, no pušta programeru da odabire biblioteke s kojima želi riješiti pojedine probleme. Na primjer, Angular u sebi sadrži svoje načine za upravljanje stanjem podataka, za React se često koristi Redux knjižnica, iako je to manje izraženo u zadnje vrijeme.

Prednost Reacta je što se osnove mogu brzo naučiti te njegova arhitektura većinom počiva na JSX datotekama koje spajaju kod za sučelje i kod za logiku renderiranja, čime programer vrlo intuitivno može utjecati na sve dijelove sučelja i manipulirati dizajnom preko raznih JavaScript funkcija što većina programera web sučelja već dobro razumije. Uz to dopušta korisniku da koristi bilo koje biblioteke koje želi za razne svrhe. Stoga, može koristiti biblioteke s kojima je već upoznat i samo ih ubaciti u aplikaciju, nije potrebno učenje koda specifičnog za radni okvir. Sa svojim React komponentama omogućeno je stvaranje generičkih komponenti vrlo jednostavno preko svojstva. Svojstva su podaci koji se šalju od komponente rodi-

telja, djetetu. Koristeći ih može se slati razne podatke generičkim komponentama i tako ih prilagoditi za koju god svrhu je potrebno.

Mane Reacta su to što je sav kod od logike renderiranja i šučelja na istom mjestu, što može rezultirati ružnim kodom i velikim datotekama punim koda koje mogu izgledati zbunjujuće i nečitljive, također React ne nudi nikakva riješenja za dizajn sučelja, već programer to sam mora pronaći.

Prednost Angulara je to što u sebi sadrži sve alate koji su potrebni za stvaranje aplikacije kakvu korisnik želi. Ukoliko je programer upoznat s OOP bit će dobro naviknut na Angularovu MVC arhitekturu koja koristi OOP. Angular ima dvosmjerno povezivanje stanja sa sučeljem, što znači da ako promijenimo stanje na sučelju, model će se također jednako promijeniti i isto tako u suprotnom slučaju. Nadalje, korištenje TypeScripta je automatski uključeno, što znači da će programer, ako ne zna, morati naučiti TypeScript, no to je u svakom slučaju preporučljivo, jer TypeScript pruža mnoge prednosti za programiranje web aplikacija. MVC arhitektura je vrlo intuitivna i podjela logike renderiranja i sučelja pomaže čistom kodu i razumijevanju točno što se događa i kad, jer je sva logika renderiranja za određeni ekran na jednom mjestu.

Mane Angulara su to što zahtjeva učenje specifične sintakse i manje je prilagođen za korištenje stranih biblioteka.

Sve u svemu oba radna okvira imaju svoju svrhu i ovisi o preferencama programera o tome koji radni okvir će odabrati za svoju svrhu.

Bibliografija

[1] TypeScript dokumentacija, https://www.typescriptlang.org/docs/handbook/intro.html, 15.05.2022

[2] Tailwind dokumentacija, https://tailwindcss.com/docs/installation, 18.05.2022

[3] Auth0 i JWT, https://auth0.com/docs/secure/tokens/json-web-tokens, 24.05.2022

[4] JWT dekoder https://jwt.io/, 24.05.2022

[5] React dokumentacija, https://reactjs.org, 15.04.2022

[6] Angular dokumnetacija, https://angular.io, 01.05.2022

[7] Understanding Angular Ivy: Incremental DOM and Virtual DOM, https://blog.nrwl.io/ understanding-angular-ivy-incremental-dom-and-virtual-dom-243be844bf36?gi=707c62ef584b, 10.06.2022

[8] State Management Battle in React 2021: Hooks, Redux, and Recoil https://dev.to/workshub/ state-management-battle-in-react-2021-hooks-redux-and-recoil-2am0, 12.06.2022

[9] RxJs dokumentacija, https://rxjs.dev/, 13.06.2022

[10] Redux dokumentacija, https://redux.js.org/, 13.06.2022

[11] NgRx dokumentacija, https://ngrx.io/, 10.06.2022

[12] Formik dokumentacija, https://formik.org/docs/overview, 31.05.2022

[13] Yup dokumentacija, https://github.com/jquense/yup, 31.05.2022

[14] Fetch dokumentacija, https://web.dev/introduction-to-fetch/, 25.05.2022

Pojmovnik

API Application programming interface. 3

CSS Cascading Style Sheets. 6

DOM Document Object Model. 1

HTML Hypertext Markup Language. 6, 7

JS JavaScript. 3, 4

JSX JavaScript XML. 28

JWT JSON Web Token. 8

TS TypeScript. 4, 5

Sažetak

Cilj rada bio je napraviti detaljnu analizu i usporedbu React i Angular radnih okvira putem 7 stavki. One su: Programski jezik koji je korišten, kako funkcionira proces renderiranja, izgled i rad komponenti, upravljanje stanjem podataka, kako rade forme i zahtjevi na API te složenost postupka učenja. Radi usporedbe, napravljene su dvije identične web aplikacije svaka u svome radnom okviru, koristeći Spring Boot za poslužiteljsku stranu. Zaključak rada je da React dopušta više slobode i fleksibilnosti u korištenju tehnologija te pruža razne mogućnosti koje olakšavaju razvijanje web aplikacije, dok je Angular sveobuhvatni radni okvir koji zahtjeva korištenje ugrađenih svojstava koja su odlično integrirana i optimizirana. Na kraju sve ovisi o želji programera i preferenciji jer s tehničke strane oba radna okvira odlično funkcioniraju.

Ključne riječi **— React, Angular, Usporedba SPA radnih okvira, Jednostranična web aplikacija, Spring Boot**

Abstract

The goal of the paper was to make a detailed analysis and comparison of React and Angular frameworks through 7 items. They are: The programming language used, how the rendering process works, the appearance and operation of components, data state management, how forms and API requests work, and the complexity of the learning process. For comparison, two identical web applications were created, each in its own framework using Spring Boot for the server side. The conclusion of the work is that React allows more freedom and flexibility in the use of technology and provides various options that facilitate the development of web applications, while Angular is a comprehensive framework that requires the use of built-in properties that are well integrated and optimized. In the end, everything depends on the developer's wishes and preferences, because from the technical side, both frameworks work great.

Keywords **— React, Angular, SPA frameworks comparison, Single page application, Spring Boot**