Razvoj web aplikacije u MERN Stack razvojnom okviru

Glavina, Jakov

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Jakov Glavina

RAZVOJ WEB APLIKACIJE U MERN STACK RAZVOJNOM OKVIRU

ZAVRŠNI RAD

Varaždin, 2020.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Jakov Glavina

Matični broj: 46211/17–R

Studij: Informacijski sustavi

RAZVOJ WEB APLIKACIJE U MERN STACK RAZVOJNOM OKVIRU

ZAVRŠNI RAD

Mentor:

Prof. dr. sc. Danijel Radošević

Varaždin, kolovoz 2020.

Jakov Glavina

Izjava o izvornosti

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

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

Glavni fokus ovog završnog rada je, kako je i samim naslovom specificirano, razvoj web aplikacije u MERN razvojnom okviru. U uvodu će biti napravljen uvod u web kao platformu za razvoj aplikacija te kao platformu s kojom su korisnici u interakciji. Potom će biti specificirane i opisane web tehnologije za razvoj tradicionalnih web aplikacija (statičke prirode) te će iste biti uspoređene sa web tehnologijama današnjice, tj. onima za razvoj dinamičkih web aplikacija i aplikacija u realnom vremenu (pojam aplikacije u realnom vremenu pomnije je objašnjen kasnije u radu. Nadalje, uvest ću pojam web razvojnog okvira (eng. web development stack) te opisati što sve podrazumijeva. Navest će se i primjeri nekih od najpopularnijih razvojnih okvira. Kako je među njima i MERN razvojni okvir, posebna pozornost bit će posvećena tom okviru te ću u drugom dijelu završnog rada implementirati cjelovitu web aplikaciju koristeći MERN okvir i popratne tehnologije uz dodatna pojašnjenja gdje će to biti potrebno. Na kraju, biti će dan završni komentar te iznesen popis literature i ostalih korištenih materijala u realizaciji ovog završnog rada.

Ključne riječi: web, web tehnologija, MERN, razvojni okvir, web aplikacija, node.js, React, Express, MongoDB

Sadržaj

1. Uvod

U današnjem svijetu Web (eng. World Wide Web) je jedna od najutjecajnijih računalnih platforma u svakodnevnim životima „povezanog“ svijeta. Slušajući svakodnevne razgovore ljudi, moguće je čuti više fraza i riječi koje se koriste naizmjenično da bi se opisao isti koncept. Neke od tih riječi uključuju „web“, „web aplikacija“, „Internet“ i sl. Kao neizbježni članovi društva informatičkog doba, nužno nam je znati razlikovati ove pojmove. Kada netko kaže „Internet“, u 99% slučajeva zapravo misli na Web. Internet je globalne mreža umreženih računala koja razmjenjuju informacije koristeći mnoštvo različitih protokola kojih je toliko da bi, kada bi išli pomnije opisivati svaki od njih, uvelike premašili opseg ovog rada. World Wide Web, skraćeno WWW, je podsustav većeg sustava (Interneta) čija je svrha omogućiti razmjenu i protok informacija na način da računala međusobno razmjenjuju tzv. web dokumente, koji po svojoj podrijetlu i prirodi ne moraju biti samo tekstualne datoteke, kako se to obično smatra. Naime, Web dokumenti su svi dokumenti koji se na bilo kakav način razmjenjuju preko Web-a, a to su primjerice videozapisi, audio zapisi, slike i ostali tipovi multimedijskih sadržaja.

World Wide Web svoje korijene vuče u još „davnim“ osamdesetim godina prošloga stoljeća, kada je engleski znanstvenik, imenom Tim Berners Lee, počeo s realizacijom svoje vizije koja se temeljila na mnoštvu tekstualnih dokumenata koji bi bili pisani hipertekstom te međusobno povezani hipervezama gdje bi veza u jednom dokumenta vodila na neki drugi dokument. U početku ovakav način je povezivanja računala i dokumenata je bio izvediv samo u CERN-u (eng. European Organization for Nuclear Research), no kroz godine Web je svoju primjenu ugledao na sve širem i širem području. Zadnjih petnaestak godine Web i aplikacije razvijene za Web, tj. web aplikacije svom rasprostranjenošću sve više i više potiskuju tradicionalne desktop aplikacije iz svakodnevne upotrebe nudeći jednake, a sve češće i superiorne alternative ostvarene pomoću web tehnologija.

Zbog toga što su web aplikacije sve prisutnije oko nas, nama kao informatičarima se posebno nameće dužnost da proširimo znanja na ovom području kako bismo mogli ostati ažurni, kompetentni i konkurentni u svojoj struci. Uviđajući važnost ove problematike, odlučio sam se pozabaviti jednim od fragmenata ove problematike te ga detaljnije razraditi, a to je razvoj (dinamičkih) web aplikacija u MERN web razvojnom okviru. Nakon kraćeg uvoda u osnovne koncepte web aplikacija i prezentiranja nekolicine web razvojnih okvira, pokazat će se važnost i snaga konkretno MERN razvojnog okvira.

2. Metode i tehnike rada

Za teoretski dio rada, u grafičkim prikazima koncepata koji će se objašnjavati, koristit ću besplatni alat draw.io, inače svjetski poznat alat za crtanje grafičkih prikaza i dijagrama svake vrste. Za praktični pak dio aplikacije između ostalog koristit ću HTML, CSS i JavaScript, tehnologije koje su okosnica svake moderne web aplikacije. Na njih ću „nadograditi“, tj. koristiti tehnologije koje su primarni fokus ovog rada, a koje su sadržane upravo u akronimu MERN. To su redom: **M**ongoDB, **E**xpress.js, **R**eact.js i naposljetku **N**ode.js. Slijedi kratak uvod u svaku od korištenih tehnologija.

2.1. HTML/CSS/JavaScript

Svaka web aplikacija unatrag petnaest godina koristi ove tehnologije od kojih svaka pomaže realizirati web aplikaciju na svojem „sloju“, kojih je tri. Prvi i najbitniji sloj je tzv. eng **markup** sloj na kojem se definira logička struktura i sadržaj glavnih elemenata web stranice i on se ostvaruje pomoću HTML-a (eng. HyperText Markup Language). Sljedeći je sloj prikaza/prezentacije, a definira izgled stranice (tipografija, boje, razmještaj elemenata i sl.) za koji koristimo CSS (eng. Cascading Style Sheets). Naposljetku, tu je JavaScript, skriptni jezik (znači da se ne prevodi u strojni kod) koji se izvršava u realnom vremenu u web pregledniku te koji omogućava interaktivnost između korisnika i web aplikacije. To je jezik više razine koji nije strogo tipiziran te ne implementira tradicionalnu objektno orijentiranu paradigmu, nego koristi nasljeđivanje preko tzv. prototipnih svojstava, kako navode i Mozilla.org i David Herman [1] [2].

2.2. MongoDB

Većina ljudi u informatičkoj struci upoznata je s bazama podataka, ali to su u velikoj većini slučajeva relacijske baze podataka. Pojam relacijske baze podataka označava logičku organizaciju rezultata konceptualnog modeliranja u tzv. relacije (tablice) u kojima se identificiraju primarni ključevi te vanjski ključevi kojima se ostvaruje povezanost između tablica. Sustavi koji omogućuju upravljanje relacijskim bazama podataka se nazivaju SURBP ili eng. RDBMS (**R**elational Data**b**ase **M**anagement **S**ystem). Podacima u takvim SUBP-ovima manipulira se tzv. **SQL** jezikom (eng. **S**tructured **Q**uery **L**anguage) koji omogućuje CRUD (**C**reate, **R**ead, **U**pdate, **D**elete) operacije nad setovima podataka. S druge strane, **NoSQL** SUBP-ovi su sustavi za upravljanje nerelacijskim bazama podataka, tj. onima koji ne implementiraju tradicionalni relacijski model organizacije podataka. Dok je u relacijskim

bazama struktura entiteta vrlo jasno i strogo definirana, kod NoSQL baza podataka, kako navodi Guru99 [3], to nije slučaj te podaci mogu biti strukturirani, polustrukturirani ili ne moraju uopće biti strukturirani, tj. njihova struktura je fleksibilna i nadogradiva. Također nemaju standardizirani upitni jezik.

Glavni strukturni element MongoDB-a su tzv. dokumenti. Dokumenti svojom strukturom pdosjećaju na objekte u JavaScriptu zbog toga što su strukturno u JSON (**J**ava**S**cript **O**bject **N**otation) obliku, na primjeru vidljivo u sljedećem zapisu:

```
{
      „Ime“: „Jakov“,
      „Prezime“: „Glavina“,
      „Adresa“: {
            „Ulica“: „Sajmišna“
            „Broj“: 31,
            „Mjesto“: „Prelog“
      }
}
```

Jedan dokument otprilike odgovara jednom zapisu u nekoj tablici u SQL bazama podataka, a dokument se sprema u tzv. *kolekciju* što odgovara tablici/relaciji u SQL bazama.

2.3. Node.js

Dok se MongoDB brine za perzistenciju podataka, programiranje na strani poslužitelja bit će realizirano koristeći Node.js, JavaScript okolinu u vrijeme izvršavanja (eng. runtime environment) u kojoj se kod izvršava izvan preglednika, kao i kod drugih serverskih programskih jezika (PHP, Ruby, Python, Java itd.). Iako je prva verzija Node.js-a puštena davne 2009. godine [4], ova okolina još ne uživa toliku upotrebu kao konsolidirani jezici poput PHP-a, no kako vrijeme prolazi, node.js postaje sve zreliji za široku, skalabilnu upotrebu. Jedan od glavnih aduta node.js-a je taj što omogućuje pisanje poslužiteljskog koda u JavaScript-u, a pošto se JavaScript koristi i za programiranje interaktivnog sloja aplikacije na strani klijenta, nekom programeru je ovo prilično velika „olakotna okolnost“ budući da može programirati u istom jeziku i na strani klijenta i na strani poslužitelja. „Ispod haube“ node.js je implementiran na Google-ovom V8 JavaScript engine-u otvorenog koda (koji pokreće i preglednik Google Chrome) pa je samim time osigurana pouzdanost, stabilnost i brzina izvođenja koda na strani poslužitelja.

Slika 1. Logo node.js-a (izvor: node.js.org)

2.3.1. npm

Kod Node.js-a posebno je važno napomenuti npm. Npm, kratica za **n**ode **p**ackage **m**anager, je, kako i samo ime kaže, upravljač paketima za Node.js. Kao i kod raznih Linux operacijskih sustava, njegova uloga je rukovanje (de)instalacijom novih i nadogradnjom postojećih Node.js paketa koji se mogu koristiti u web aplikaciji implementiranoj u JavaScript-u, bez obzira radi li se o strani poslužitelja ili strani klijenta. Motivacija oko ovog alata je takva da se olakša upravljanje eksternim JavaScript bibliotekama koje programeru mogu zatrebati, u smislu da nije potrebno razmišljati na tradicionalni način korištenja JavaScripta, a to je da se pazi gdje se i koje biblioteke uključuju, u koje HTML datoteke, na koje točno mjesto i kojim redoslijedom (da bi se izbjegle greške u međuovisnostima raznih biblioteka i sl.).

2.4. Express.js

Iako se cjelokupna funkcionalnost potrebna za izvođenje koda na strani poslužitelja može implementirati u „čistom“ JavaScript-u pomoću node.js, takav pristup je relativno kompliciran iz razloga što uključeni moduli node.js-a nisu toliko prilagođeni za brzi razvoj koda. Tu u igru dolazi Express.js koji je svojevrsna „nadogradnja“ tog pristupa i uvelike olakšava pisanje koda na strani poslužitelja. Prema odgovoru na pitanje postavljeno na poznatom programerskom portalu Stack Overflow, neke od mogućnosti koje Express pruža su: napredno usmjeravanje (eng. *routing*), primjerice moguće je definirati posebne upravljače događaja (eng. *event handlers*) ovisno o HTTP metodi (GET, POST, PUT, DELETE itd.), omogućuje serviranje statičkih datoteka (HTML, CSS, JavaScript), mnoštvo metoda i svojstava za objekte zahtjeva (*request* objekt) i odgovora (*response* objekt) i mnoge druge [5]. Neke od navedenih mogućnosti moguće je implementirati u „sirovom“ node.js-u, ali je način kompliciraniji nego da se jednostavno koristi Express.

Kod ispod prikazuje kreiranje poslužiteljske aplikacije u Express-u.

```
var express = require('express')
var app = express()
```

Prva linija koda zahtijeva (*require*) modul express instaliran preko npm-a i sprema ga u varijablu express. Tada varijabli app pridružujemo poziv funkcije `express()` koji u pozadini kreira Express aplikaciju spremnu za posluživanje HTTP zahtjeva pomoću koje se onda može pokrenuti poslužitelj. Za usporedbu, ovako se pokreće server u „sirovom“ node.js-u:

```
var http = require('http');
http.CreateServer(function (request, response) {
    // pozdravna poruka prilikom pokretanja servera
    response.writeHead(200, { 'Content-Type': 'text/plain' });
    response.write(„Pozdrav iz node.js HTTP servera!“);
    response.end();
}).listen(3000);
```

Ovdje je odmah uočljivo da sirova implementacija zahtijeva poviše linija koda – prvo se pozove ugrađeni *http* modul te se pozove funkcija `CreateServer` kojoj se pridružuje upravljač događaja. Njegov parametri su `request` i `response`, koji predstavljaju objekt HTTP zahtjeva i HTTP odgovora respektivno. U primjeru iznad, prilikom prvog izvršavanja ovog koda, tj. kreiranjem servera, klijentu će se poslati zaglavlje sa statusnim kodom 200 OK te definicijom tipa sadržaja. Dalje se šalje pozdravna poruka te se odgovor završava pozivanjem `.end()` metode objekta `response`.

Navedeni su samo neki od primjera svojstava i metoda koje Express ima za ponuditi, kako navodi i službena Express dokumentacija [6]:

`express.static()` – daje uputu Express-u da želimo servirati statične web dokumente. Putanja do korijenskog direktorija u kojem će se servirati dokumenti se se stavlja kao jedini parametar funkcije.

`req.ip` – svojstvo koje vraća IP adresu s koje je pristigao HTTP zahtjev.

`req.method` – svojstvo vraća tip metode kod slanja zahtjeva (GET, POST itd.)

`req.get('neki parametar zaglavlja')` – vraća vrijednost parametra zaglavlja (eng. *header*) koji je proslijeđen u zahtjevu

`res.json()` – vraća odgovor u JSON obliku.

2.5. React.js

React.js (ili React, ReactJS) je deklarativna biblioteka za izgradnju korisničkih sučelja. Stičući popularnost od svojeg izdavanja 2015. godine pa sve do danas, zahvaljujući svojoj jednostavnosti i deklarativnoj paradigmi, React se sve češće koristi na webu za opisivanje korisničkih sučelja, no osim toga koristi se i u izgradnji sučelja za mobilne aplikacije (kao zasebna biblioteka po imenu React Native). Deklarativnost biblioteke označava osobinu da je bitan izgled i finalni rezultat gradivnih elemenata sučelja, a ne i njihova implementacija „ispod haube“, tj. *način* na koji ćemo doći do prikazanih elemenata nije toliko bitan. Ovakav pristup omogućava brže razvijanje sučelja, ali i lakše rješavanje pogrešaka.

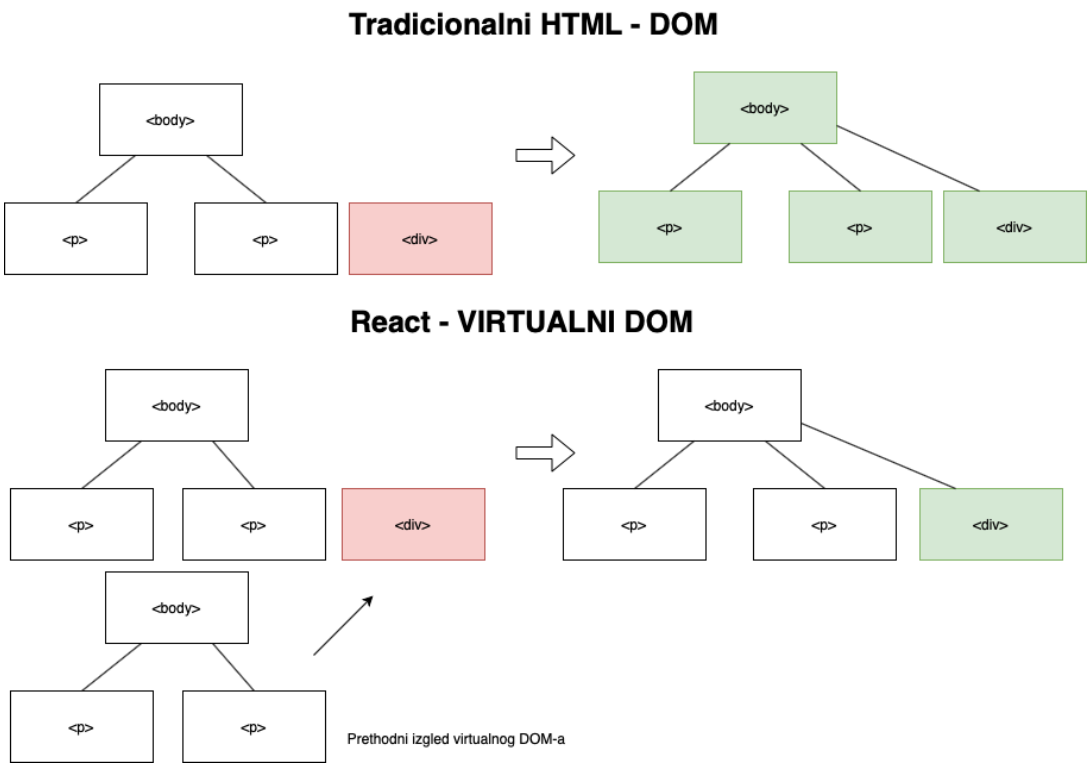
Osnovni gradivni elementi sučelja kod React-a su tzv. komponente (eng. *components*). Komponente predstavljaju male, kompaktne i ponovno iskoristive elemente koji enkapsuliraju svu svoju funkcionalnost unutar sebe. Također upravljaju svojim stanjem (eng. *state*), a logika (interaktivnost) je pisana u JavaScriptu. Slijedi primjer komponente:

```
1. const MojaKomponenta = (props) => {
2.   const ime = „Jakov“
3.   const prezime = props.prezime
4.   return (
5.           <h1>Pozdrav, {ime} {prezime}!</h1>
6.   )
7. }
8.           <MojaKomponenta prezime=“Glavina“ />
```

Iz primjera je vidljivo da su komponente osmišljene kao elementi koji imaju jednu odgovornost što im omogućava da budu ponovno iskoristivi. Prema službenoj dokumentaciji React-a, svaka React komponenta je zapravo funkcija koja prihvaća argumente (varijable, funkcije itd.) preko ugrađenog `props` objekta [7]. Primjerice, ako smo u zadnjoj liniji, kod „pozivanja“, tj. prikazivanja (eng. *render*) komponente (koristeći React DOM biblioteku) njoj pridružili atribut *prezime* s vrijednošću *Glavina*, tada će prezime interno u komponenti biti spremljeno kao svojstvo objekta `props`. Tako svi atributi koji budu proslijeđeni kod prikazivanja komponente će biti spremljeni u objekt `props`. U 3. liniji koda to prezime spremamo u lokalnu varijablu prezime da bi na kraju ispisali prilagođeni pozdrav koristeći ugrađivanje prilagođenih vrijednosti unutar vitičastih zagrada (unutar vitičastih zagrada može ići bilo kakav JavaScript ili JavaScript izraz).

Kako funkcionira prikazivanje React komponenti? Dobro je poznato da u tradicionalnom načinu rada web stranica postoji DOM (eng. *Document Object Model*), tj.

sučelje za organizaciju svih prikazanih elemenata web stranice u stablastu strukturu koja omogućuje brzo pretraživanje i manipulaciju istih. Kada napravimo neku promjenu u tradicionalnoj web stranici, stranica se treba u potpunosti osvježiti što znači da se ovježava i cjelokupni DOM. Ovisno o performansama računala i mreže, ovo je vremenski neisplativa operacija. S druge strane, React koristi **virtualni DOM** što znači da prilikom prvog renderiranja stranice, React u predmemoriji stvara početni izgled cjelokupnog DOM-a. Svaki element DOM-a ima svoj korespondirajući element u virtualnom DOM-u. Kada se bilo koji element virtualnog DOM-a ažurira ili promijeni stanje, React uspoređuje novo stanje virtualnog DOM-a sa onim prije samog ažuriranja te na temelju toga identificira element koji se promijenio i ponovno prikaže/osvježi *samo taj* element u regularnom DOM-u., štedeći resurse i vrijeme. Na slici 2. ilustrirana je ova razlika. S lijeve strane prikazano je početno stanje DOM-a (<*body*> element s dvoje djece <*p*> elemenata). Crvenom bojom je naznačen element koji dodajemo u DOM-u, a na desnoj strani slike naznačeni su elementi koji se ponovno renderiraju nakon dodavanja <*div*> elementa u DOM.



Slika 2. Tradicionalni i virtualni DOM (izvor: vlastita izrada)

2.6. WebSockets & Socket.IO

Jedan od velikih nedostataka HTTP-a kao protokola koji koristi TCP na transportnom sloju, a koji mu je dugo bio slaba točka je činjenica da je to jednosmjerni protokol koji funkcionira na način da klijent prvo šalje pakete kojima zahtijeva uspostavu veze s poslužiteljem, te kada je veza uspostavljena, šalje podatke prema poslužitelju. Isto tako, ako je zahtjev takav da zatražuje neke podatke prisutne na poslužitelju, tada poslužitelj vraća zatražene podatke klijentu. Uglavnom, cijela komunikacija se svodi na shemu da klijent nešto zatraži od poslužitelja, a poslužitelj klijentu tada taj resurs vrati. Početkom 2010-ih, točnije 2011. godine standardiziran je WebSockets protokol aplikacijskog sloja koji omogućuje potpunu dvosmjernu (eng. *full duplex*) komunikaciju između klijenta i poslužitelja, koristeći samo jednu TCP vezu. Ovo znači da sada poslužitelj napokon može klijentu neovisno slati poruke, poruke koje nisu samo puki odgovor na klijentove zahtjeve nego naprosto poruke arbitrarnog sadržaja. WebSockets protokol je programerima otvorio pregršt mogućnosti za razvoj aplikacija u realnom vremenu, a jedna od primjena (koja će i kasnije u mojoj aplikaciji biti implementirana) je tzv. real-time chat. Komuniciranje (chat) u realnom vremenu je u današnje vrijeme sveprisutno pa mi je ovo bio logičan odabir za primjer implementacije, a chat ću implementirati koristeći Socket.IO biblioteku koja *nije* implementacija WebSocket protokola sama po sebi, nego samo koristi WebSockets kao metodu transporta poruka, te primjerice ako WebSockets nije dostupan za korištenje, onda ima rezervu (eng. *fallback*) na neke druge slične protokole.

2.6.1. Osnove korištenja Socket.IO biblioteke

Slijedi prikaz osnovne Socket.IO implementacije. Svaka Socket.IO implementacija se sastoji od poslužiteljskog i klijentskog dijela.

```
1. // const server. . . varijabla unaprijed kreiranog HTTP servera
2. const io = require('socket.io')(server);
3. io.on('connection', (socket) => {
4. . . .
5. io.emit(„Pozdrav sa posluzitelja!“)
6. socket.on('sendMessage', (message) => console.log(message.message)“
7. } );
```

Na 2. liniji uključujemo Socket.IO biblioteku te ju priključujemo na prethodno kreirani `server`. Nakon toga se definira *connection* upravljač događaja na koju se kači callback funkcija čiji argument je *socket* objekt koji se kreirao kao posljedica povezivanja klijenta na poslužitelja. Na 5. liniji poslužitelj će *emitirati* poruku svim priključenim klijentima sa sadržajem

Pozdrav sa posluzitelja!. Nadalje, na liniji 6 je definirano da kada socket primi događaj *sendMessage*, da kao posljedicu tog događaja u konzoli će ispiše sadržaj primljene poruke.

Na klijentskoj strani:

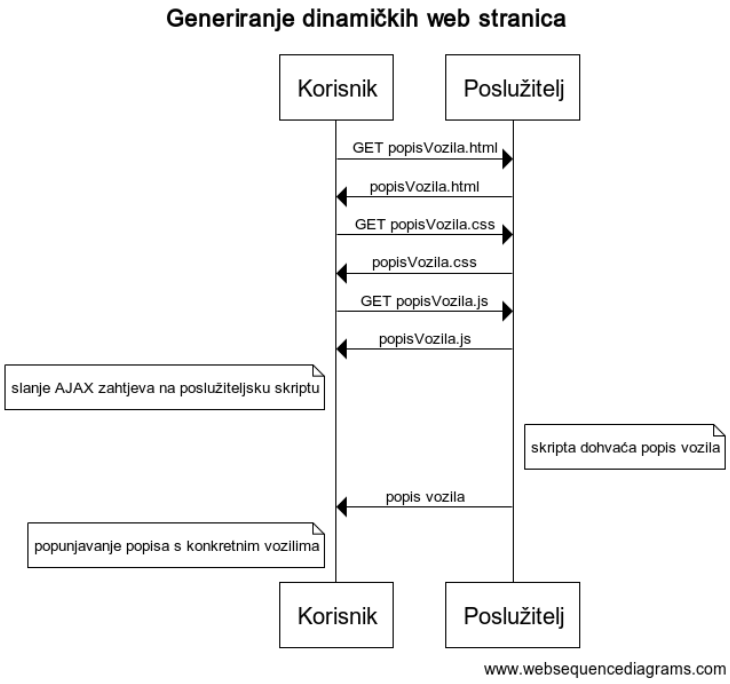
```
1. const io = require('socket.io-client')
2. const socket = io.socketIOClient(ENDPOINT) // ENDPOINT - unaprijed
   definirana adresa na kojem se vrti Socket.IO posluzitelj
3. socket.emit('sendMessage', { message: „Pozdrav sa klijenta“ })
```

Sve što treba učiniti na klijentskoj strani je uključiti relevantnu biblioteku i funkcijom `socketIOClient` kreirati Socket.IO klijent, nakon čega možemo emitirati i primati poruke pomoću tog socketa. Na taj način ostvarena je WebSocket veza prema poslužitelju i kada klijent emitira poruku na liniji 3., poslužitelj će u svojoj konzoli ispisati poruku *Pozdrav sa klijenta*.

3. Programiranje na webu – nekad i danas

Iako je Web prisutan već četrdesetak godina, dobar dio tog vremena je postojao isključivo u statičkom obliku. Ako je web statičan, to znači da je način prikazivanja i generiranja stranica predvidljiv, konzistentan i jednoznačan. To je upravo zato jer u začecima Weba nije postojala ideja o dinamičnosti web stranica pa su sve stranice bile statičke. Programer bi ih prvi put napisao i stavio na poslužitelj da bi mogle biti servirane korisnicima. Logično je da je sa implementacijske strane stvar vrlo primitivna i HTML kod (s eventualno popratnim dokumentima poput CSS uputa) bi se jednokratno dohvaćao s poslužitelja. Samim time nema interaktivnosti na stranici te ovisno o tome čemu je namijenjena, vrlo brzo može postati dosadna te odvući korisnika, koji ju pregledava, od sebe. No, kako je vrijeme prolazilo, ovaj pristup pokazao se vremenski neisplativim i umarajućim pošto je za bilo kakve promjene i ažuriranja bilo potrebno mijenjati web dokument i ponovno ga stavljati na poslužitelj.

S druge strane, početkom novog tisućljeća pojavljuju se stranice koje su izgrađene kao dinamičke. Danas je ideja kompletno statičnog weba de facto nezamisliva zbog toga što bi se izgubilo mnoštvo interaktivnosti koja je trenutno prisutna na web stranicama. Dinamičke stranice funkcioniraju na način da klijent pošalje HTTP (obično GET) zahtjev na poslužitelj zahtijevajući neki web dokument koji je najčešće osmišljen tako da se dinamički generira. Nakon pristizanja zahtjeva, poslužiteljska web skripta obrađuje taj zahtjev i dohvaća relevantne podatke, najčešće iz baze podataka. Prema statičkim dijelovima dokumenta generira se kostur stranice koji se vraća kao odgovor na zahtjev i te taj isti kostur korisnička skripta popunjuje s dohvaćenim podacima iz baze podatka. Na slici 3. pobliže je ilustriran princip generiranja dinamičkih web stranica na banalnom primjeru dohvaćanja popisa vozila.



Slika 3. Princip generiranja dinamičkih web stranica (izvor: vlastita izrada)

Na temelju opisanog primjera možemo si predočiti koja je snaga dinamičkih web stranica te zaključiti da njima postižemo daleko veći stupanj interaktivnosti nego ako bismo koristili isključivo statičke web stranice. Tema ovog rada, razvoj aplikacije u MERN razvojnom okviru, utemeljena je na principu dinamičkih web stranica gdje će se svi relevantni podaci dohvaćati sa strane korisnika i poslužitelja koji će pak dohvaćati podatke iz baze i spremati ih u bazu.

3.1. Okviri za razvijanje web aplikacija

Razvojni okviri za razvijanje web aplikacija (eng. *web development stacks*) možemo tumačiti kao setove alata i aplikacija koji olakšavaju razvojni proces određene web aplikacije. Korijene njihova nastajanja možemo naslutiti upravo u činjenici da su se s razvitkom weba razvijale i web aplikacije, a tako i njihova složenost i kompleksnost. Upravljati mnogim raznim alatima koji su programeru potrebni i na raspolaganju kao i vladanje konceptima modernih web aplikacija je bilo sve teže te je cijeli taj ekosustav bilo potrebno optimizirati. Tako su s vremenom nastali mnogi razvojni okviri (i popratne razvojne okoline, eng. *framework*) za razvijanje web aplikacija: MEAN, MERN, MEVN, LAMP, Meteor, Django, Ruby on Rails, Spring (Java) i mnogi drugi. Ovo zvuči kao nabrajanje raznih naziva bez razloga pa će ukratko biti istaknuti neki od okvira, uz njihove prednosti i nedostatke.

3.1.1. MEAN / MEVN

MEAN je kratica za **M**ongoDB, **E**xpress, ***A****ngular*, **N**ode.js, a MEVN za **M**ongoDB, **E**xpress, ***V****ue.js*, **N**ode.js. Odmah iz akronima može se naslutiti da su ova dva razvojna okvira vrlo slična MERN okviru, okviru koji je središnja tema ovog rada. Tehnologije u kojima se razlikuju su u kosim slovima, a odnose se na JavaScript razvojnu okolinu koja je korištena za razvoj web aplikacije sa strane klijenta. U MEAN-u korišten je Angular, a u MEVN-u Vue.js. Važno je napomenuti da je na strani klijenta kod MEAN-a i MEVN-a, riječ o *razvojnoj okolini (eng. framework)*, a kod MERN-a samo o *biblioteci, a ne cijeloj razvojnoj okolini.*

Angular (verzija 2.x) , originalno AngularJS (verzija 1.x), je razvojna okolina koju je razvio Google 2010. godine s ciljem da se olakša razvoj tzv. SPA (eng. *single-page app*) web aplikacija, tj. aplikacija koje se temelje na principu da se korisnik zadržava na jednoj, glavnoj stranici koja dinamički mijenja sadržaj ovisno o tome kakvu interakciju korisnik održava sa njome (nadalje SPA). Angular omogućava implementaciju MVC (Model – View – Controller) i MVVM (Model – View – Viewmodel) arhitekture na strani klijenta. Nadalje, službena dokumentacija Angular razvojnog okvira navodi da se Angular također (kao i React) temelji na deklarativnoj programskoj paradigmi umjesto klasične, imperativne programske paradigme [8].

S druge strane, Vue.js, originalno napravljena od strane bivšeg zaposlenika Google-a Evana Youa, je prilično „zelena“ razvojna okolina u odnosu na Angular s obzirom da njezinu prvu pojavu bilježimo u veljači 2014. godine. Vue.js se temelji na implementaciji isključivo MVVM arhitekture, a ono što mu je zajedničko s Angularom je to da je zamišljen tako da olakša implementaciju SPA aplikacija, a ono što je zajedničko i s Reactom je to da se osniva na kreiranju komponenata i njihovoj kompoziciji.

Glavne prednosti oba ova razvojna okvira je da se sve korištene tehnologije (izuzev MongoDB-a, no valja napomenuti da i on ima sučelje u JavaScriptu) temelje na JavaScriptu što olakšava programeru prijelazak između različitih „područja“ aplikacije. Nedostatak je pak u tome da govorimo o relativno novim okvirima (naspram nekih ostalih) pa podrška od strane programerske zajednice još nije toliko velika.

3.1.2. LAMP

LAMP je kratica za **L**inux, **A**pache, **M**ySQL, **P**HP. Ovaj razvojni okvir u fokus stavlja stranu poslužitelja i infrastrukturu koja pokreće web aplikaciju. Tako se programer može odlučiti za Linux kao operacijski sustav poslužiteljskog računala, Apache kao poslužitelj (za serviranje web stranica), MySQL kao bazu podataka i konačno PHP za izvođenje programskog koda na poslužitelju. Očita prednost ovog razvojnog okvira je činjenica da je to vrlo

„zrela“ kombinacija alata i budući da je pokazana kao pouzdana, LAMP je još uvijek jedan od popularnih izbora programera za razvojni okvir na poslužitelju (eng. *back-end stack*). Nedostatak LAMP razvojnog okvira je u tome da se od programera zahtijeva fluentnost u više prilično različitih tehnologija i alata koji nisu povezani sami po sebi. Nadalje, unatoč tome što je tako popularan, PHP i dalje ima određene nekonzistentnosti u svom dizajnu i ponašanju pa ga programeri ili vole ili, blago rečeno, ne vole.

3.1.3. Django, Ruby on Rails, Java Spring

Django je web razvojni okvir temeljen na Pythonu koji implementira MVC predložak arhitekturalnog dizajna. Prema Django-u, osnovni principi koje ovaj razvojni okvir pospješuje su „ponovna iskoristivost i nadogradivost komponenti, manje programskog koda, neovisnost elemenata i brzi razvoj produkata“ [9]. Django je u potpunosti osmišljen u Pythonu, a koriste ih mnogo poznatih imena, od kojih su neka Instagram, Mozilla i Bitbucket.

Slika 4. Logo Django razvojnog okvira

Slično kao i prethodno spomenuti Django, **Rails**, tj. **Ruby on Rails**, je razvojni okvir temeljen na programskom jeziku Ruby koji implementira MVC predložak arhitekturalnog dizajna. Neki od koncepata koji je u programerski svijet unio Rails su jednostavno kreiranje baze podataka, generiranje kostura za poglede (eng. *views*, odnosi se na stranu klijenta), a koji su inspirirali druge razvojne okvire poput Djanga, Laravel-a i Sails.js-a. Valja istaknuti da je Rails vrlo robusna, stabilna i popularna platforma za razvoj web aplikacija što je poduprto činjenicom da pokreće stranice poput Airbnb-a, GitHub-a i Scribd-a [10].

Slika 5. Logo Ruby on Rails razvojnog okvira

Spring je razvojni okvir otvorenog koda temeljen na programskom jeziku Java. Izašao 2002. godine, Spring je konsolidirao svoje mjesto kao razvojni okvir za bilo koju Java aplikaciju, a ne samo web aplikacije. Nudi bogatu kolekciju modula, npr. modul za pristup bazi podataka, modul za autentikaciju i autorizaciju, modul za pisanje jediničnih i integracijskih testova, a posebno su zanimljiva dva modula koji su na neki način svoj zasebni okvir: modul za aspektno orijentirano programiranje i modul za implementaciju aplikacije na bazi MVC uzorka arhitekturalnog dizajna. Jedan od najpoznatijih korisnika Springa je Netflix, koji, kako kaže i sam zaposlenik Netflixa Taylor Wicksell, je originalno koristio vlastite razvijene Java biblioteke i razvojne okvire, no od 2019. godine migrirali su gotovo cijelu svoju platformu na Spring [11].

Slika 6. Logo Spring razvojnog okvira

4. Implementacija aplikacije u MERN razvojnom okviru

4.1. Opis aplikacijske domene

U današnje vrijeme jedan od glavnih multimedijskih sadržaja koji se konzumira su filmovi. Iako popularnost takozvanih „streaming“ servisa kao što je Netflix raste iz dana u dan, ništa ne može nadmašiti osjećaj gledanja dugo iščekivanog filma u kinodvorani na dan premijernog prikazivanja filma. U Hrvatskoj postoji samo jedan veliki lanac kinodvorana za prikazivanje koji je u vlasništvu tvrtke CineStar. Svi ostali prikazivači filmova su, u usporedbi s njima, relativno mali i ne mogu ostvariti slični tržišni utjecaj. Jedna od stvari koja je omogućila osobitu pristupačnost CineStar filmova je mogućnost vršenja rezervacija putem njihove web stranice. Mušterije dolaze na web stranice Cinestara, izlistavaju filmove koji su dostupni za prikazivanje u tekućem tjednu, te odabiru onaj kojeg će gledati. Odabir sjedala u dvorani za prikazivanja se vrši dinamički u smislu da je u realnom vremenu prikazano koja sjedala su

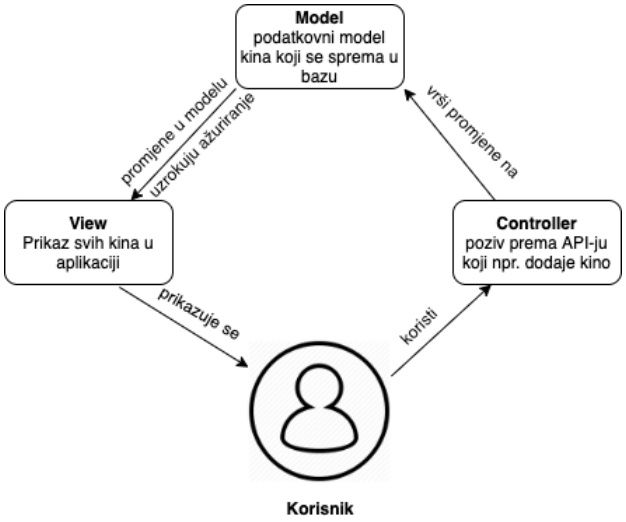
zauzeta a koja slobodna za rezervaciju. Prilikom završetka rezervacije korisnik dobiva potvrdu o uspješnoj rezervaciji te ju netom prije prikazivanja filma prikazuje blagajniku koji obrađuje tu potvrdu.

Moja aplikacija, koja je otvorenog koda, nastoji riješiti problem manjih prikazivača filmova u smislu da im omogući pristup korištenju iste, da bi na atraktivniji način privukli mušterije. Aplikacija koju ću napraviti vuče korijene u Cinestar-ovoj zamisli uz neke modifikacije. U samoj suštini temelji se na provedbi rezervacija za filmove koji su trenutno dostupni za prikazivanje. Dolaskom na glavnu stranicu neregistrirani korisnik predstavljen je sa pozdravnom porukom te navigacijskim oknom na vrhu stranice. Posjetom stranice popisa filmova pregledava filmove gdje za svaki film pišu informacije poput popisa glumaca, produkcijske kuće, sumarizirane radnje filma, ukupne ocjene i sl. Uzimajući u obzir da se prethodno prijavio, može pokrenuti proces rezervacije istog. Na stranici koja mu se otvara odabire dostupno kino u kojem se taj film prikazuje te vrijeme prikazivanja filma. Potom odabire jedno ili više sjedala koje želi rezervirati, a pritom su mu naznačena sjedala koja su slobodna, koja su već zauzeta te koja je trenutno odabrao. Nakon odabira sjedala u dvorani za prikazivanje, korisniku se otvara prozor za plaćanje rezervacije preko PayPal-a, gdje ju korisnik plaća, potvrđuje rezervaciju i na kraju prima QR kod s podacima o rezervaciji koji prikazuje zaposleniku kinodvorane uživo netom prije samog prikazivanja filma. U bilo kojem trenutku, na bilo kojoj stranici, u donjem desnom kutu stranice moguće je otvoriti prozor za komunikaciju u realnom vremenu s drugim korisnicima aplikacije, bio to administrator aplikacije ili neki drugi registrirani/neregistrirani korisnik aplikacije. Registrirani korisnik dodatno ima pregled svih svojih rezervacija i mogućnost promjene vlastitih postavki kao što su ime i prezime, e-mail i lozinka. Administrator svake tvrtke/kina, između ostalog, ima CRD operacije nad filmovima i vremenima prikazivanja koji će biti dostupni za rezerviranje, sistematizirani pregled svih rezervacija kao i generalne statistike o radu cijeloj aplikaciji.

4.2. Arhitektura aplikacije

Postoji mnogo različitih uzoraka dizajna arhitekture aplikacije, a samo neki od njih su: **M**odel-**V**iew-**A**dapter (MVA), **M**odel-**V**iew-**C**ontroller (MVC), MVP (**M**odel-**V**iew-**P**resenter), MVVM (**M**odel-**V**iew-**V**iew**M**odel) i mnogi drugi. Ukratko, uzorci dizajna arhitekture daju određene smjernice oko toga kako je aplikacija strukturirana, kako se upravlja podacima aplikacije, kako se oni mijenjaju/transformiraju i na koji način se isti prikazuju korisniku aplikacije. Za svoju aplikaciju odabrao sam MVC uzorak, što znači da ću aplikaciju podijeliti u tri posebna dijela: podatkovne **modele**, **kontrolere** koji upravljaju mojim podacima te **prikaze**

(views) koji prikazuju moje podatke. Slika 7. prikazuje pojednostavljeni način funkcioniranja aplikacije razvijene koristeći MVC uzorak dizajna.

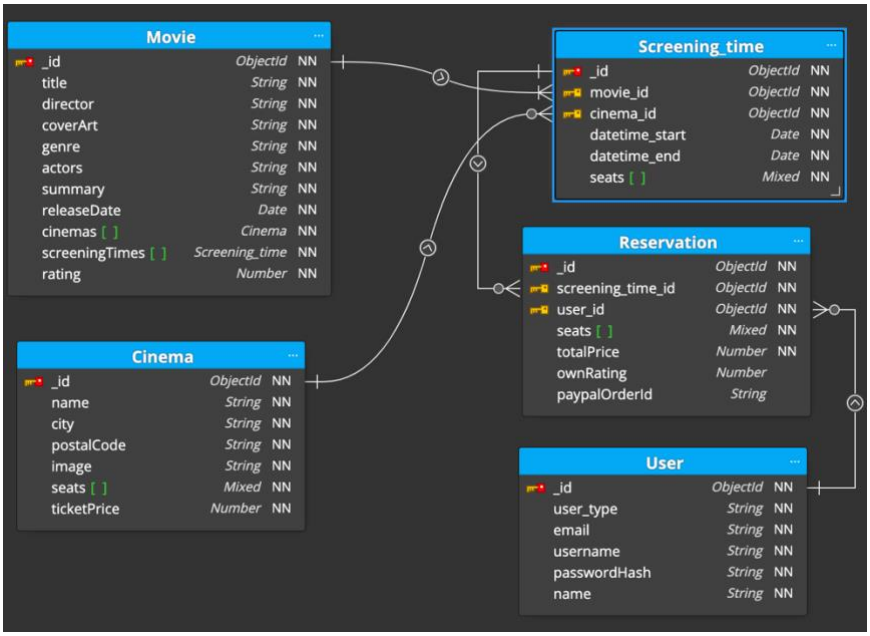


Slika 7. Osnovna zamisao MVC uzorka dizajna na primjeru moje aplikacije (izvor: vlastita izrada, prema: Model–view–controller, Wikipedia.org, Diagram of interactions within the MVC pattern)

Razlog odabira ovog uzorka arhitekturalnog dizajna aplikacije je vrlo jednostavan: MVC je vrlo popularan, pokazan i dokazan uzorak dizajna koji se pojavio još 70-ih godina prošlog stoljeća, a na njegovu rasprostranjenost ukazuje činjenica da upravo okviri koje su opisani u poglavlju 3.1.3. (Spring, Django, Rails) implementiraju Model – View – Controller uzorak dizajna.

4.3. Podatkovni model

Unatoč tome što je spomenuto da NoSQL baze podataka, pa među njima i MongoDB, ne forsiraju striktno definiranu strukturu osnovnih gradivnih elemenata baze, postoje mnogi alati i principi po kojima se može definirati „okvirna“ struktura dokumenata svake kolekcije. Zapravo je glavna prednost MongoDB-a, čija struktura nije striktno definirana, ta da na taj način omogućuje brzi i fleksibilni razvoj aplikacije dok bi kod SQL baza podataka bilo prvo potrebno definirati ERA model, te kreirati svaku od tablica i striktno se držati modela podataka kojim su one definirane. Ako želimo prakticirati slične stvari kod MongoDB-a, to je sasvim dopušteno i u tu svrhu koristio sam alat Moon Modeler. Osim za MongoDB, Moon Modeler može se koristiti za modeliranje podataka za MariaDB i GraphQL. Slika 8 prikazuje podatkovni model aplikacije.



Slika 8. Podatkovni model aplikacije (izvor: vlastita izrada)

Nadalje je opisana svake od shema dokumenata:

- **User** – u dokumente kolekcije `users` spremaju se podaci o svakom korisniku. Sadrži sljedeće atribute, tj. ključeve: *user_type* sadrži informaciju o tipu korisnika („administrator“ ili „registeredUser“), *email, username, passwordHash* – hashirana lozinka koristeći bcrypt algoritam te *name*
- **Movie** – predstavlja individualni film u koji spremamo sve bitne ključeve potrebne za prezentiranje informacija o filmu korisniku, a to su *title, director, coverArt, genre, actors, summary* i *releaseDate*. Nadalje, spremamo i reference (vanjski ključ) na sva kina i vremena prikazivanja u kojima se pojavljuje pojedini film
- **Cinema** – predstavlja individualno kino, te ima atirbute koji opisuju njegovo ime, grad, poštanski broj, sliku za prikaz i cijenu jedne ulaznice (u proizvoljnoj valuti, no mi ćemo koristiti američke dolare). Isto tako ima i polje sjedala za svako kino, čiji tip je *Mixed*, što znači da je svaki elment polja *seats* objekt proizvoljne strukture, no mi ćemo spremati objekte s dogovorenom strukturom: ID sjedala, ime sjedala te zauzetost sjedala (0 – slobodno, 1 - odabrano, 2 – zauzeto)
- **Screening_time** – predstavlja pojedino vrijeme prikazivanja koje se asocira na pojedini film (vanjski ključ *movie_id*) i pojedino kino (vanjski ključ *cinema_id*). Osim toga dokument ove sheme ima definirano vrijeme početka i završetka prikazivanja filma, kao i popis sjedala (koji se kreiranjem dokumenta prekopira iz pripadnog dokumenta kina)

- **Reservation** – predstavlja napravljenu rezervaciju koja ima ima vanjske ključeve na vrijeme prikazivanja (*screening_time_id*) i korisnika koji je napravio rezervaciju (*user_id*). Spremiti treba i sva sjedala koja je korisnik rezervirao, ukupnu cijenu rezervacije, ocjenu (filma), te ID narudžbe od PayPala.

Što se tiče kardinalnosti veza, svakom vremenu prikazivanja pripada točno jedan film i jedno kino dok svaki film i kino mogu biti sadržani u više vremena prikazivanja. Nadalje, svakoj rezervaciji odgovara točno jedno vrijeme prikazivanja i točno jedan korisnik, a svako vrijeme prikazivanja i korisnik mogu biti u više rezervacija. Valja istaknuti da svaka od shema ima primarni ključ *_id* tipa ObjectId. ObjectId je poseban identifikatorski tip podataka sadržan u specificikaciji BSON-a (Binary JSON), binarnog serijalizacijskog formata koji se koristi za pohranu dokumenata unutar MongoDB-a [12]. Prednost korištenja ObjectId kao jedinstvenog identifikatora je činjenica da zauzimaju malo memorijskog prostora, brzo se generiraju i generiraju se na način da su poredani (eng. *ordered*) te se na taj način postiže poredak kod svih sljedećih zapisa.

4.4. Postavljanje razvojnog okruženja

Prije nego možemo započeti sa implementacijom konkretne aplikacije, potrebno je postaviti naše razvojno okruženje. To između ostalog podrazumijeva uređivač teksta (ili, po želji, integrirano razvojno okruženje poput JetBrains Webstorm-a), a ja sam odabrao uvelike popularan i besplatan Visual Studio Code, dostupan na poveznici https://code.visualstudio.com/. Aplikaciju ću razvijati u dva odvojena dijela: klijentski dio (frontend) i poslužiteljski dio (backend), a cijeli proces razvijanja aplikacije će više-manje biti verzioniran uz pomoć sustava git te servisa GitHub. Frontend je dostupan na poveznici https://github.com/fobos531/cinema-frontend, a backend na poveznici https://github.com/fobos531/cinema-backend. U trenutku pisanja ovog rada oba repozitorija su postavljena na privatnu vidljivost, no po objavljivanju rada vidljivost će biti prebačena na javnu.

Za svaki od repozitorija potrebno je kreirati node.js „projekt“ pomoću node package managera. To se radi komandom `npm init` i nakon slijeda koraka se na taj način kreira `package.json` datoteka koja deskriptivno opisuje specifikacije projekta (ime, verzija, moduli koji se koriste i sl.). Slijede moduli instalirani u backend projektu i kratak opis te svrha njihove instalacije:

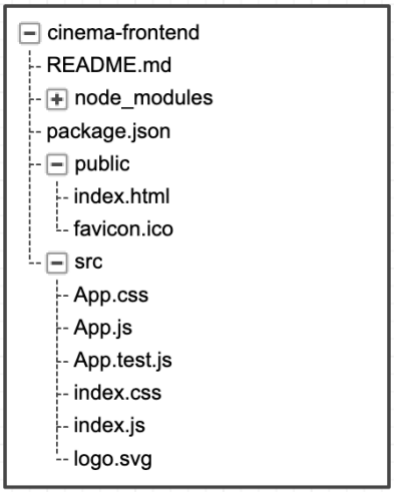
- **bcrypt –** biblioteka za stvaranje pouzdanog kriptografskog sažetka, koristi se kod spremanja lozinke u bazu

- **body-parser –** middleware za Express koji parsira dolazeće zahtjeve prema poslužitelju i tijelo zahtjeva sprema u `body` svojstvo objekta zahtjeva
- **cloudinary –** modul za ostvarivanje veze prema online Cloudinary servisu za upravljanje (uploadanje i preuzimanje) slikovnim sadržajima
- **cors –** middleware za Express koji omogućava CORS (Cross – Origin Resource Sharing) mehanizam
- **dotenv –** modul koji učitava varijable okruženja (eng. environment variable) iz lokalne .env datoteke u `process.env` objekt globalno dostupan u projektu.
- **express – najvažniji modul backenda**, okvir za razvoj web aplikacija na poslužitelju
- **generatepassword –** biblioteka za generiranje nasumičnih, sigurnih lozinki
- **imdb-api –** node.js modul za sučelje prema OMDb API-ju, tj. neslužbenom IMDb API-ju koji služi za dohvaćanje svih relevantnih podataka o filmovima
- **jsonwebtoken –** modul u kojem je implementirana JSON Web Token specifikacija i koji omogućuje autentikaciju i autorizaciju korisnika kroz cijelu aplikaciju
- **mongoose –** Object Dana Modeling (ODM) biblioteka za node.js koja je „čovjek u sredini“ za komunikaciju s MongoDB bazom podataka. Po načinu funkcioniranja slična je Entity Frameworku u .NET-u.
- **mongoose-simple-random –** plugin za mongoose sheme (eng. Schema) koji omogućava traženje nasumičnog dokumenta u nekoj kolekciji
- **mongoose-unique-validator –** plugin za mongoose sheme koji omogućuje i primijenjuje razne validacije (specificirane shemom dokumenta) na novim dokumentima koji se pokušavaju zapisati u neku kolekciju
- **multer –** middleware za Express koja upravlja zahtjevima koji su rezultat slanja formi sa enctype atributom postavljenim na multipart/form-data, a čija svrha je olakšati postavljanje slika na poslužitelj
- **nodemailer –** biblioteka za slanje email poruka preko SMTP protokola, korištena kod mogućnosti povrata lozinke
- **nodemon –** biblioteka koja ubrzava razvoj node.js aplikacija na način da ponovno pokrene aplikaciju istog trenutka dok detektira promjenu u nekom od datoteka u direktoriju projekta

Instalacija na frontend-u je uglavnom analogna onoj na backend-u, no ima nekih razlika vrijednih napominjanja. Dok se na backend-u za inicijalizaciju projekta koristi naredba `npm init`, kod frontenda, budući da zapravo kreiramo React aplikaciju, koristimo *create-react-app*.

create-react-app je snažna skripta razvijena od strane Facebook-a koja u jednoj komandi instalira sve potrebne biblioteke i popratne module o kojima te biblioteke ovise i na taj način ubrzava postavljanje razvojnog okruženja na klijentskoj strani. Konkretno, frontend projekt za našu aplikaciju kreiramo tako da pokrenemo skriptu sa `npx create-react-app cinema-frontend`.

Prilikom završetka pokretanja skripte, kreira se sljedeća struktura direktorija:



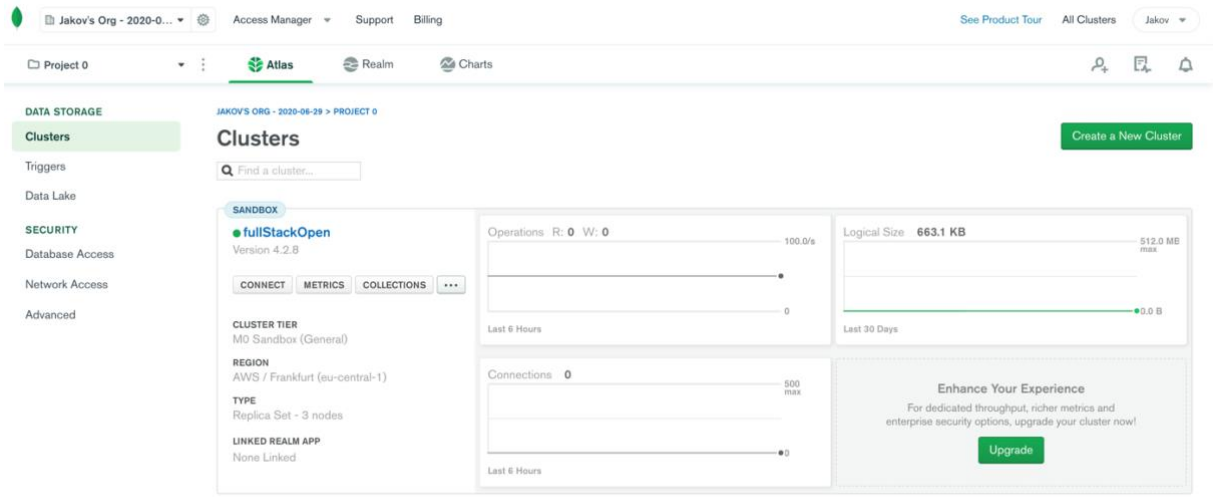
Slika 9. Struktura direktorija nakon kreiranja React aplikacije (izvor: vlastita izrada)

Možemo vidjeti da se kreiralo podosta datoteka koje omogućuju brže započinjanje s razvojem React aplikacije. Najbitnije od njih za napomenuti su: ***package.json*** – specifikacija projekta, ***public/index.html*** – sadrži „praznu“ stranicu u koju će se renderirati React sučelje te ***src/index.js*** – ulazna točka aplikacije – kada pokrenemo aplikaciju, prvo će se pokrenuti ***index.js*** datoteka i prema toma sve ostale datoteke.

4.5. Baza podataka – MongoDB Atlas

U zadnje vrijeme postoje dva glavna pristupa za postavljanje okruženja u kojem će raditi baza podataka. Jedna solucija je da se ručno preuzme sustav za upravljanje odabranom bazom podataka (npr. MySQL i phpMyAdmin), instalira na poslužiteljskom računalu, kreira se podatkovni model i u našoj aplikaciji se ostvari veza na tu bazu podataka. To je dokazani i robusni pristup postavljanju baze podataka. Ja sam se odlučio za „mlađi“ pristup, a to je korištenje jednog od mnogobrojnih dostupnih Database-as-a-Service servisa što naprosto predstavlja bazu podataka „u oblaku“. Jedan od takvih servisa je i MongoDB Atlas. MongoDB Atlas omogućuje brzu konfiguraciju MongoDB DBMS-a i jednostavno spajanje aplikacije na isti.

Prvo je potrebno otići na početnu stranicu MongoDB Atlas servisa. Nakon kreiranja korisničkog računa, sljedeći korak je kreirati novi MongoDB Cluster. Biraju se razne postavke kao što je pružatelj cloud usluga, ime klastera, razred klastera (ima i besplatnih i onih koji se plaćaju) i sl.



Slika 10. Upravljačka ploča MongoDB Atlas-a

Nakon kreiranja klastera vidimo našu glavnu upravljačku ploču pomoću koje možemo upravljati svim parametrima i podacima u bazi podataka. Zanimljivo je napomenuti da je ***Type*** našeg klastera *Replica Set – 3 nodes.* To znači da su podaci u našoj bazi podataka raspoređeni na 3 fizička servera što pomaže s raspoređivanjem opterećenja (eng. *load balancing*, u scenarijima kad se razvijaju velike i kompleksne aplikacije koje zahtijevaju veliku skalabilnost). Spajanje na ovu bazu podataka se ostvaruje korištenjem tzv. *connection string*-a koji koristimo u razvoju backenda aplikacije (kasnije u varijabli okruženja `MONGODB_URI`). Prikazan je konkretno primjer connection stringa za našu bazu podataka:

```
mongodb+srv://fobos531:<LOZINKA_OVDJE>@fullstackopen-
zieb2.mongodb.net/<IME_BAZE_PODATKA_OVDJE>?retryWrites=true&w=majority
```

Ime baze podataka za našu aplikaciju je *cinema,* a pripadajuće kolekcije će imati imena *cinemas, movies, reservations, screeningtimes* i *users.*

4.6. Razvoj aplikacije - backend

Kao osnovu svake moderne web aplikacije potrebno je kvalitetno razviti backend stranu aplikacije. Struktura direktorija backenda je sljedeća:

Slika 11. Struktura direktorija backend projekta

- **node_modules –** sadrži datoteke svih modula u projektu
- **utils –** sadrži `transporter` objekt biblioteke *nodemailer* koji služi tome da se pripreme postavke za slanje e-mail poruka sa backenda
- **controllers –** JavaScript datoteke u kojima se kreira Express `router` objekt za upravljanje zahtjevima prema unaprijed određenim rutama
- **models –** modeli podataka definirani mongoose shemama
- **middleware –** razne pomoćne funkcije koje se koriste u upravljačima zahtjeva (eng. route handler), npr. middleware za ostvarenje autentikacije i autorizacije pomoću JSON Web Tokena

4.6.1. Konfiguracija i postavljanje servera

Ponovna iskoristivost koda u Node.js okruženju

Ovdje ćemo iskoristiti priliku da objasnimo princip ostvarivanja ponovne iskoristivosti koda (eng. *code reusability*) kod Node.js-a. Node.js koristi tzv. CommonJS, prvi projekt koji je predstavljao težnju za standardizacijom i uspostavom konvencija o korištenju modula (ponovno iskoristivih blokova koda) izvan pregledničkog okruženja.

Prema CommonJS konvencijama, pojedini modul se uključuje u datoteku s funkcijom `require()`, a kao parametar se proslijeđuje ime modula. Po potrebi se na uključenom modulu odmah poziva neka funkcija, ili se uključen modul može pridružiti nekoj varijabli za kasniju manipulaciju. Nadalje, varijable, funkcije, objekti i sl. se iz datoteke izvezuju na način da se pridruže posebnom `module.exports` objektu koji je sam po sebi prisutan u svakoj JavaScript datoteci u Node.js okruženju.

```
1. const http = require('http');
2. const socketIo = require('socket.io');
3. const app = require('./app');
4.
5. const server = http.createServer(app);
6. const io = socketIo(server);
7.
```

```
8. io.on('connection', (socket) => {
9.   console.log('New client connected');
10.    socket.on('sendMsgToServer', (message) => {
11.      socket.broadcast.emit('sendMsgToClient', { sender: socket.id,
12.        message
13.      });
14.    });
15.
16.    socket.on('disconnect', () => {
17.      console.log('Client disconnected');
18.    });
19.  });
20.
21.  server.listen(process.env.port || 80, () => {
22.    console.log(`Server running on port ${process.env.PORT}`);
23.  });
```

Prikazan je osnovni programski kod potreban za pokretanje poslužitelja. U 1. liniji uključujemo ugrađeni *http* modul s funkcijama potrebnima za pokretanje poslužitelja. Uključujemo i *socket.io* biblioteku te pripremljenu i konfiguriranu aplikaciju iz *app* modula u `app` varijablu. Korištenjem `createServer()` kreira se server čiji parametar je funkcija koja osluškuje za zahtjeve prema serveru (eng. *request listener*). Budući da moja aplikacija ima mogućnost komunikacije između korisnika u realnom vremenu, inicijalizira se i socket.io sa poslužiteljske strane, koji osluškuje kada se klijenti spoje i odspoje sa poslužitelja te ispisuju odgovarajuću poruku na konzolu. Nakon inicijalizacije Socket.IO poslužitelja, `socket` objekt osluškuje za događaj *`sendMsgToServer`* koji se okida kada korisnik pošalje poruku sa klijentskog računala.

```
1. const mongoose = require('mongoose');
2. const cors = require('cors');
3. const express = require('express');
4. const bodyParser = require('body-parser');
5. const moviesRouter = require('./controllers/movies');
6. const usersRouter = require('./controllers/users');
7. const loginRouter = require('./controllers/login');
8. const cinemasRouter = require('./controllers/cinemas');
9. const screeningTimesRouter = require('./controllers/screeningTimes');
10.  const miscRouter = require('./controllers/misc');
11.  const reservationsRouter = require('./controllers/reservations');
12.  const authMiddleware = require('./middleware/authentication');
13.  const app = express();
14.  mongoose.connect(process.env.MONGODB_URI, { useNewUrlParser: true,
    useFindAndModify: false, useUnifiedTopology: true, useCreateIndex:
    true })
15.    .then(() => console.log('connected to mongodb'))
16.    .catch((error) => console.error('cannot connect to mongodb: ',
    error.message));
17.  app.use(cors());
18.  app.use(bodyParser.urlencoded({ extended: true }));
19.  app.use(express.json());
20.  app.use('/api/movies', moviesRouter);
21.  app.use('/api/users', usersRouter);
22.  app.use('/api/login', loginRouter);
```

```
23.  app.use('/api/cinemas', cinemasRouter);
24.  app.use('/api/screeningtimes', authMiddleware,
   screeningTimesRouter);
25.  app.use('/api/misc', miscRouter);
26.  app.use('/api/reservations', reservationsRouter);
27.
28.  module.exports = app;
```

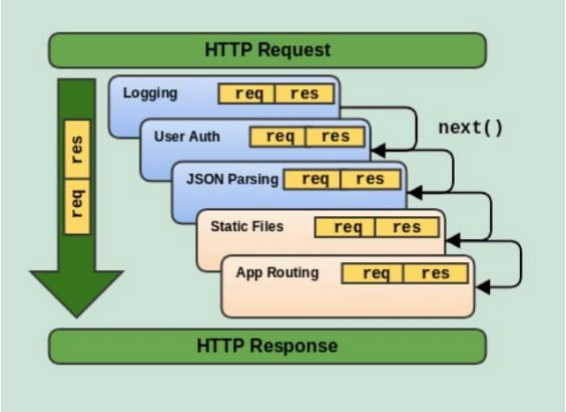
Zadnji ključni element inicijalizacije aplikacije je app.js datoteka. U linijama 1-4 uključujemo sve bitne biblioteke, a u linijama u 5-11 uključujemo sve *router* objekte u kojima su definirani upravljači zahtjeva prema poslužitelju (tj. prema onome što se zove API). Nakon kreiranja Express aplikacije i povezivanja na bazu podataka u liniji 14, pomoću `.use()` metode uključuje se sav middleware bitan za funkcioniranje aplikacije. U linijama 17-19 to je middleware s pomoćnim funkcijama za rukovanje `request` objektom dok se u linijama 20-26 definira koji *router* objekt je odgovoran za rukovanje *request-response* ciklusom ovisno o tome kojoj putanji API-ja se pristupa (prvi parametar).

Middleware funkcije

Prije nego krenemo dalje, važno je objasniti prethodno spomenute *middleware* funkcije koje Express intezivno koristi. Middleware funkcija, u kontekstu Expressa, je funkcija oblika

```
1. const middlewareFunkcija = (request, response, next) => {
2.    if (req.body.username == „Jakov“) {
3.      console.log(„Odlicno, ti si Jakov! Samo naprijed!“)
4.      next()
5.   }
6. }
```

Svaka middleware funkcija prima tri parametra, objekt zahtjeva, objekt odgovora i funkciju `next`. `next` je funkcija koja poziva sljedeću middleware funkciju definiranu u „lancu“ middleware funkcija. U prethodnom programskom primjeru, u linijima 17-26 napravili smo mnoštvo poziva `.use()` funkcije kojima smo uključili sav potreban middleware u našu Express aplikaciju. Točno onim redoslijedom poziva funkcije `use()`, pozivat će se definirane middleware funkcije. Konkretno, u našem primjeru prvo će se pozvati `cors()` funkcija, pa će se nad objektom zahtjeva izvesti `bodyParser` funkcija itd. Ovdje se nazire prava moć middleware funkcija – middleware funkcije omogućuju step-by-step manipulaciju objektima zahtjeva i odgovora na način da se npr. prvo provjerava je li korisnik prijavljen, pa se po potrebi autenticira/autorizira, pa se onda parsira JSON teret (eng. *payload*) u objektu zahtjeva da bi se na kraju servirale neke statičke datoteke korisniku. Upravo ovo prikazano je na primjeru na slici 12.



Slika 12. Ciklus middleware funkcija (izvor: Dev.to, Understanding Express middleware{A beginners guide} https://dev.to/ghvstcode/understanding-express-middleware-a-beginners-guide-g73)

4.6.2. Autentikacija na poslužiteljskoj strani

Autentikacija je ostvarena pomoću JSON Web Tokena. JSON Web Tokeni su relativno novi način prijenosa povjerljivih informacija između primatelja i pošiljatelja. Svaki JSON Web Token sastoji se od tri dijela: zaglavlja (header), podataka (payload) i potvrde potpisa. Svaki token se potpisuje pomoću privatnog *secret*-a ili para javnog i privatnog ključa. Neki od algoritama koji se koriste za enkripciju podataka su: HS256, RS384, ES512, PS256 i dr. Autentikacija na poslužiteljskoj strani ostvarena je tako da je svaki poziv prema nekoj krajnjoj točki (eng. *endpoint*) API-ja zaštićen pomoću tokena. Pogledajmo sljedeći kod:

```
1. const jwt = require('jsonwebtoken');
2. const User = require('../models/user');
3.
4. const isAuthenticated = async (req, res, next) => {
5.   try {
6.     const token = req.header('Authorization').replace('Bearer ', '');
7.     const decodedToken = jwt.verify(token, process.env.SECRET);
8.     if (!token || !decodedToken.id) {
9.       return res.status(401).json({ error: 'token missing or
   invalid' });
10.      }
11.      const user = await User.findById(decodedToken.id);
12.      req.token = token;
13.      req.user = user;
14.      req.isAuthenticated = true;
15.      next();
16.    } catch {
17.      res.sendStatus(401).send({ error: 'invalid username or
   password! ' });
18.    }
19.  };
20.
21.  module.exports = isAuthenticated;
```

Autentikacija je implementirana kao middleware koji se priključuje na gotovo svaki upravljač zahtjeva da bi se osigurala sigurnost aplikacije. `isAuthenticated` je asinkrona metoda kod koje tražimo token u `Authorization` zaglavlju zahtjeva te ga dekodiramo sa preddefiniranim `SECRET`-om. Ako odgovarajući token ne postoji, šalje se 401 Unauthorized odgovor, a u suprotnom u bazi podataka se pronađe odgovarajući korisniik, informacije o njemu se nakače na objekt zahtjeva te se poziva sljedeća middleware funkcija.

4.6.3. Implementacija podatkovnih modela

Da bi mogli rukovoditi s entitetima podatkovnog modela opisanog u poglavlju 4.2., potrebno je implementirati iste kao model u backendu da bi mogli upravljati odgovarajućim dokumentima u bazi. Prikazan je primjer implementacije modela korisnika:

```
1. const mongoose = require('mongoose');
2. const uniqueValidator = require('mongoose-unique-validator');
3.
4. const userSchema = new mongoose.Schema({
5.   user_type: {
6.     type: String,
7.     required: true,
8.     enum: ['administrator', 'registeredUser'],
9.   },
10.    email: {
11.      type: String,
12.      required: true,
13.      unique: true,
14.    },
15.    username: {
16.      type: String,
17.      required: true,
18.      minlength: 3,
19.      unique: true,
20.    },
21.    passwordHash: {
22.      type: String,
23.      required: true,
24.    },
25.    name: {
26.      type: String,
27.      required: true,
28.    },
29.  });
30.
31.  userSchema.set('toJSON', {
32.    transform: (document, returnedObject) => {
33.      returnedObject.id = returnedObject._id.toString();
34.      delete returnedObject._id;
35.      delete returnedObject.__v;
36.      delete returnedObject.passwordHash;
37.    }
38.  });
39.
40.  userSchema.plugin(uniqueValidator);
41.  module.exports = mongoose.model('User', userSchema);
```

Konstruktorom `Schema` klase u biblioteci *mongoose* definira se izgled prethodno definiranog modela korisnika. U linijama 31-38 se prilagođava ponašanje ugrađene `.toJSON()` (pomoću koje se vraća odgovarajući *dokument* iz kolekcije) funkcije svake mongoose scheme gdje se iz sigurnosnih i estetskih razloga miče lozinka te se svojstvo `_id` mijenja u `id`. Nadalje, na liniji 40 korisničkoj shemi pridružujemo dodatak za mongoose, *mongoose-unique-validator* koji smo uvezli u drugoj liniji koda. Ovaj dodatak omogućuje nam da osiguramo jedinstvenost vrijednosti pojedinih atributa u dokumentu što se specificira svojstvom `required` u objektu pojedinog atributa. Na koncu, izvozimo kompilirani mongoose ***model*** sa imenom `User`. Ime modela se definira u jednini, a prvim kreiranjem dokumenta ovog modela u bazi podataka mongoose će inteligentno kreirati kolekciju s odgovarajućim imenom u množini (*users*).

4.6.4. API (aplikacijsko programsko sučelje)

Slijedi sažet opis aplikacijskog programskog sučelja aplikacije, tj. opis *Controller* sloja u MVC arhitekturi: popis svih endpoint-a API-ja, njihova funkcija, te primjer zahtjeva i odgovora. Napomena: svaki endpoint ima prefiks */api/*.

Tablica 1. Krajnje točke API-ja aplikacije

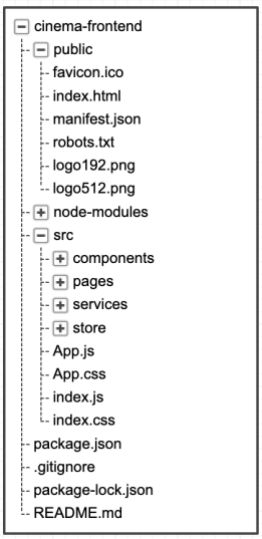
Kategorija	Metoda	Endpoint	Zahtjev	Odgovor	Funkcija
Prijava	POST	/login	Korisnički podaci s email-om i lozinkom	JWT, name i username	Autentikacija i prijava korisnika
Kino	POST	/cinemas	Podaci o kinu	-	Dodavanje kina
Kino	GET	/cinemas	-	Sva kina u JSON-obliku	Dohvat svih kina
Kino	DELETE	/cinemas/:id	ID kina u URL-u	Informacija o obrisanom kinu	Brisanje kina
Film	GET	/movies	-	Svi filmovi	Dohvat filmova
Film	GET	/movies/:id	ID filma u URL-u	Pronađen film	Dohvat određenog filma
Film	POST	/movies	Podaci o filmu	Dodani film	Dodavanje filma
Film	DELETE	/movies/:id	ID filma u URL-u	Informacija o obrisanom filmu	Brisanje filma
Vr. prikazivanja	POST	/screeningtimes/	Podaci o vr. prikazivanja	Dodano vrijeme prikazivanja	Dodavanje vr. prikazivanja

Vr. prikazivanja	GET	/screeningtimes/	-	Sva vr. prikazivanja	Dohvat vr. prikazivanja
Vr. prikazivanja	DELETE	/screeningtimes/:id	ID vr. prikazivanja	Informacija o obrisanom vr. pri.	Brisanje vr. prikazivanja
Rezervacija	POST	/reservations	Podaci o rezervaciji	Dodana rezervacija	Dodavanje rezervacija
Rezervacija	GET	/reservations/:id	ID rezervacije	Pronađena rezervacija	Dohvaćanje rezervacije
Rezervacija	PATCH	/reservations/cinema:id	ID kina	Popis sjedala u kinu	Ažuriranje statusa sjedala
Rezervacija	PATCH	/reservations/rate/:id	ID rezervacije	Ažurirana rezervacija	Ocjenjivanje filma/rezervacije
Rezervacija	GET	/reservations/	-	Sve rezervacije	Dohvat svih rezervacija
Korisnik	POST	/users/	Podaci o korisniku	Dodani korisnik	Dodavanje korisnika
Korisnik	PUT	/users/	Podaci o korisniku	Ažurirani korisnik	Ažuriranje korisnika
Korisnik	PATCH	/users/	Email korisnika	Korisnik s novom lozinkom	Resetiranje korisničke lozinke
Korisnik	GET	/users/totalUsers	-	Ukupni broj korisnika	Ukupni broj korisnika
Razno	GET	/misc/seats/:id	ID kina	Sjedala za odabrano kino	Vraćanje svih sjedala za odabrano kino
Razno	GET	/misc/ticketPrice/:id	ID kina	Cijena ulaznice za odabrano kino	Vraćanje cijene ulaznice za odabrano kino
Razno	GET	/misc/verifyToken	Token u zaglavlju	{authenticated: boolean }	Provjerava ispravnost JWT-a u zaglavlju
Razno	GET	/misc/verifyTokenAdmin	Token u zaglavlju	{authenticated: boolean }	Provjerava je li korisnik administrator

4.7. Razvoj aplikacije - frontend

Nakon što je pripremljen backend za našu aplikaciju, možemo krenuti s frontend-om. Na frontend-u koristimo pretežito React.js biblioteku, a projekt frontenda podijeljen je na sljedeći način:

- **node_modules –** sadrži datoteke svih modula u projektu
- **public –** datoteke koje se prezentiraju i serviraju korisniku
- **src –** datoteke izvornog koda u kojem se razvija React aplikacija
    - **components –** React komponente koje se koriste širom aplikacije
    - **pages –** React komponente koje su sastavni dio pojedinih podstranica
    - **services –** JavaScript datoteke koje komuniciraju s API-jem na backendu
    - **store –** JavaScript datoteke odgovorne za implementaciju perzistencije podataka u „stanju“ na stranici (implementirano u biblioteci Redux)
- **index.js i App.js –** glavna ulazna mjesta za aplikaciju.



Slika 13. Struktura direktorija frontend projekta

4.7.1. Perzistencija podataka u stanju pomoću Redux-a

U poglavlju 2.5. kao jedna od prednosti korištenja React.js biblioteke spomenuta je činjenica da React na vrlo učinkovit način „vodi evidenciju“ o tome koje komponente, tj. koji dijelovi DOM-a se nakon promjena u web stranici trebaju ponovno učitavati te da tom prilikom React ponovno učitava isključivo te elemente stranice. Komponente u Reactu također mogu imati vlastito „stanje“, tj. varijable koje u sebi čuvaju arbitrarne podatke (broj, string, objekt, boolean), a nama zapravo semantički predstavljaju podatke čije stanje se može mijenjati. Primjerice, jedan takav slučaj može biti praćenje akcija koje je korisnik napravio na stranici,

recimo je li pritisnuo neki gumb. Pogledajmo sljedeći primjer jedne vrlo jednostavne komponente:

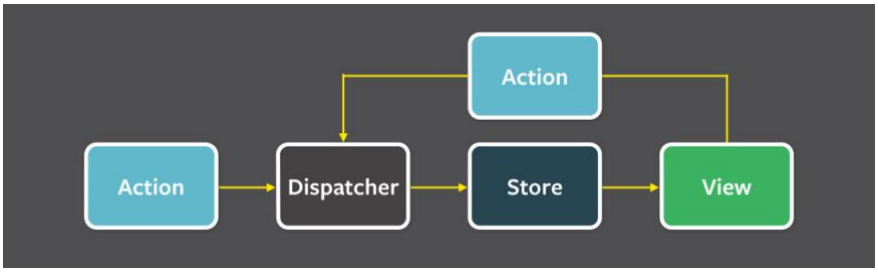
```
1. import React, { useState } from 'react'
2. const PrimjerStanja = () => {
3.   const [gumbKliknut, postaviGumbKliknut] = useState(false)
4.   if (gumbKliknut) {
5.     return <p>Gumb je kliknut!</p>
6.   }
7.   else {
8.     return (
9.       <Button onClick={() => postaviGumbKliknut(true)}>
10.         Klikni me
11.       </Button>
12.     )
13.   }
14. }
```

Napravili smo komponentu PrimjerStanja koja prikazuje Button, tj. gumb koji ima tekst *Klikni me* i koji se može kliknuti. U 3. liniji pozivamo `useState()` funkciju, i proslijeđujemo joj parametar false. Što radi `useState()` funkcija? Ova funkcija stvara tzv. *stateful* varijablu - varijablu namijenjenu za čuvanje stanja, i funkciju kojom se stvorena varijabla može ažurirati. `useState()` kao parametar prima početnu vrijednost stateful varijable što je u našem slučaju false. Povratne vrijednosti `useState()` funkcije smo stavili u varijable `gumbKliknut` i `postaviGumbKliknut` korištenjem sintakse destrukturiranja varijabli, koja je u JavaScript uvedena u ES6 reviziji jezika. Klikom na gumb okida se *onClick* događaj i onda se varijabla `gumbKliknut` postavlja na true. **Svaka promjena bilo kojeg dijela stanja, tj. bilo koje varijable stanja koja je dio neke komponente, uzrokuje ponovno renderiranje te komponente.** Na ovaj način React „zna“ da se dogodila promjena u ovoj komponenti te da sigurno treba ažurirati ovaj dio DOM-a. Kao posljedica ponovnog renderiranja ove komponente, ući ćemo u if blok na liniji 4, te će se umjesto gumba renderirati tekst *Gumb je kliknut!*. Na ovom primjeru možemo uočiti moć Reacta - React omogućuje praćenje arbitrarno definiranog stanja i nadalje, uvjetno renderiranje raznih dijelova komponente (ako je gumb kliknut, prikazuje se tekst, a u protivnom se prikazuje gumb). Nadalje, varijable stanja se mogu prenijeti komponentama djeci na korištenje tako da ih se prenese u `props` objekt. Međutim, ovdje se počinje nazirati problem. Za male aplikacije koje nisu kompleksne, ova solucija sa `useState()` je odlična i nije problem prenositi varijable stanja komponentama djeci, no što učiniti kada razvijamo aplikaciju velike složenosti koja ima mnogo različitih varijabli stanja? Upravljati aplikacijom s mnogom različitih varijabla stanja je vrlo složeno, posebno ako komponentama djeci varijable stanja prenosimo preko svojstava i na mnogo razina u dubinu. Ovaj problem rješava **Redux**.

Redux je biblioteka za manipulaciju aplikacijskim stanjem napravljena upravo sa ciljem da riješi prethodno opisan problem. Dok se u tradicionalnom načinu upravljanja stanjem sve svodi na varijable stanja rasute po raznim komponentama aplikacije, Redux implementacija spaja sve „dijelove“ stanja (logično podijeljene po funkciji) u jedinstveni „kontejner“ koji čini cjelokupno aplikacijsko stanje. Ovaj kontejner je dostupan kroz cjelokupnu aplikaciju i omogućuje organiziran i kategoriziran pristup do aplikacijskog stanja.

Slika 14. Logo biblioteke Redux (izvor: https://redux.js.org/)

Redux se u načinu funkcioniranja „ispod haube“ oslanja na svog prethodnika, arhitekturu Flux razvijenu od strane Facebooka, a Flux funkcionira ovako:



Slika 15. Princip funkcioniranja Flux-a (izvor: https://fullstackopen.com/en/part6/flux_architecture_and_redux)

Postoje 4 glavna elementa:

- View – predstavlja svaki element aplikacije koji se prezentira korisniku.
- Store – podatke koje koristi pojedini View koje će prikazati korisniku dohvaća iz prethodno spomenutog zajedničkog „kontejnera“, čiji je službeni naziv *store* (grubi prijevod na hrvatski bi bio *skladište*)
- Action – iz svakog View-a moguće je „otpremiti“ neku akciju do „otpremitelja akcije“ (eng. *dispatcher*)
- Dispatcher – element Redux-a koji akciju „doprema“ do store-a, i ovisno o tipu akcije, mijenja neki dio store-a.

Kada komponenta koja prati sadržaj nekog dijela store-a ili cijeli store, „dozna“ da se promijenio sadržaj store-a, to će uzrokovati njezino ponovno renderiranje da bi se

odrazile promjene koje su se dogodile, u slučaju da koristi neke elemente store-a za prikaz svojeg sadržaja.

Svaka akcija koja se otprema do store-a je zapravo objekt koji minimalno mora imati svojstvo koje određuje tip akcije (označena masnim slovima):

```
{
      type: „ADD_MOVIE“,
      payload: movie
}
```

Akcija prikazana gore definira da ono što se događa isporučivanjem iste je činjenica da se dodaje novi film. Način na koji će akcija utjecati na stanje u store-u definira tzv. *reducer*, koji je funkcija koja prihvaća dva argumenta: trenutno stanje i akciju, a vraća novo stanje.

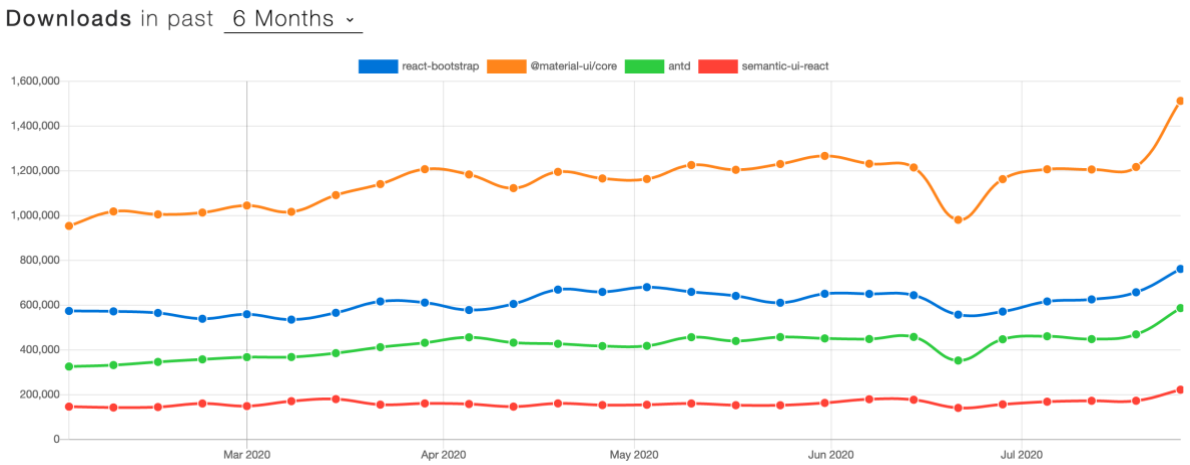
```
const initialState = { movies: [] }
const movieReducer = (state = initialState, action) => {
      state = { …state, movies: […state.movies, action.payload] }
      return state
}
```

Kod prikazan iznad će primiti isporučenu akciju, te će u objekt stanja odgovoran za upravljanje filmovima, koji sadrži polje svih filmova, dodati novi film i na kraju vratiti novo stanje.

4.8. Razvijena aplikacija

4.8.1. Dizajn

Što se tiče pitanja dizajna, tu zaista ima mnogo opcija. Naravno, dizajn može biti u potpunosti vlastiti korištenjem vlastitih stilskih uputa, no kako je rasla zrelost React-a, tako su se s vremenom pojavljivali i razvojni okviri za razvoj korisničkih sučelja u Reactu. Dosada je razvijeno mnoštvo takvih razvojnih okvira, a samo neki od njih uključuju: PrimeReact, Ant Design, Shards React, Material-UI, React Bootstrap (adaptacija vrlo poznatog Bootstrap okvira za React), Semantic UI i mnogi drugi. Slika 16 prikazuje usporedbu popularnosti pojedinih razvojnih okvira prema broju preuzimanja pripadajućih npm modula u zadnjih pola godine, prema stranici npmtrends.com:

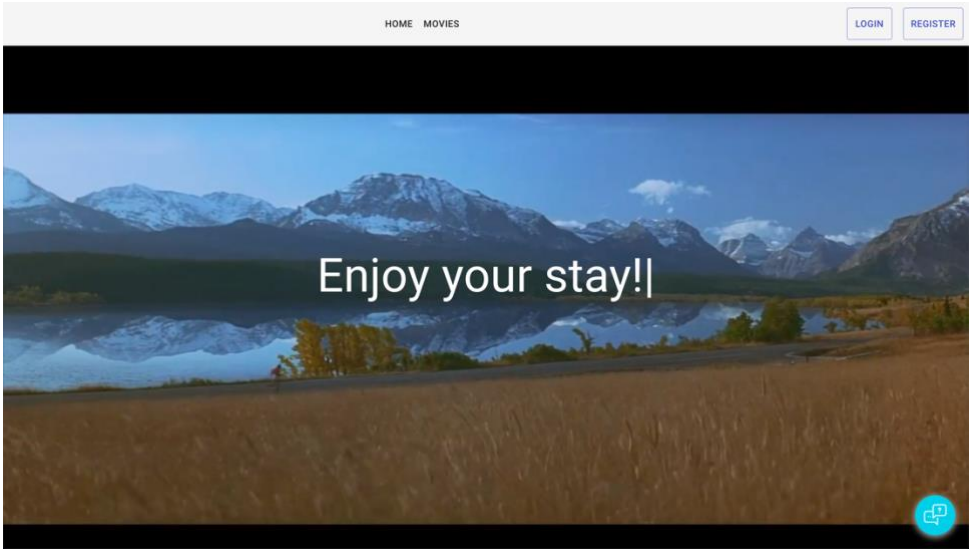


Slika 16. Popularnost pojedinih React UI razvojnih okvira (izvor: npmtrends.com)

Kao što je i vidljivo iz samog grafičkog prikaza, jedan od uvjerljivo najpopularnijih razvojnih okvira je Material-UI. Upravo iz tog razloga, a i mnoštva drugih razloga (primjerice, ima odlično podršku drugih developera) odabrao sam ovaj razvojni okvir kao okosnicu za dizajn svoje aplikacije. U aplikaciji su za dizajn uglavnom korištene pripremljene komponente dostupne u Material-UI okviru te su za neke podstranice korišteni gotovi predlošci dostupni na Material-UI stranici koji su posebno modificirani i dotjerani za potrebe ove aplikacije. U aplikaciji nije korišten tradicionalni pristup pisanju stilskih uputa, gdje bi se stilske upute pisale u zasebnim CSS datotekama, nego se iskoristio relativno novi koncept tzv. *CSS-in-JS* stilskih uputa što naprosto znači da se stilske upute pišu unutar istih datoteka kao i React komponente. Ovo je u duhu paradigme da se funkcionalnost cjelokupne aplikacije ne dijeli na strukturu (HTML), prezentaciju (CSS) i interaktivnost (JS) gdje je svaka od ovih točaka u vlastitim datotekama, nego se sve tri točke spajaju i dijele po **funkcionalnostima**. Dakle, svaka funkcionalnost ima sve „gradivne elemente“ koji joj trebaju u samoj sebi. Nadalje, unatoč tome što Material-UI podržava mobilnu responzivnost, **aplikacija je samo djelomično responzivna i kao takva je namijenjena korištenju na stolnom računalu kao mediju te nije preporučljivo koristiti ju na drugim uređajima.**

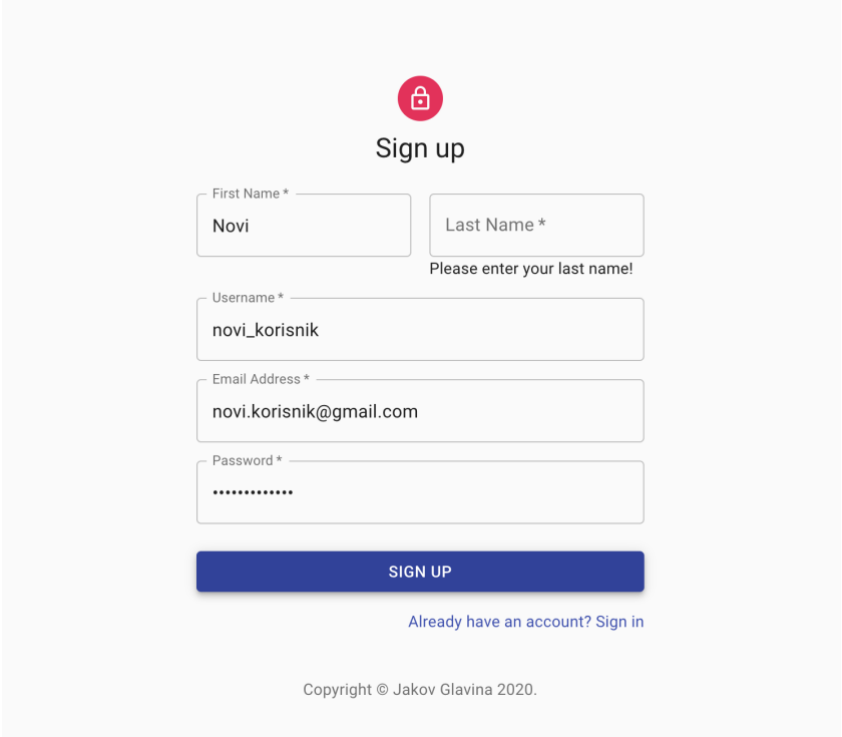
4.8.2. Neregistrirani korisnik

Na početnoj stranici korisnik se susreće sa atraktivnim prikazom pozadinskog videa filmske tematike. U gornjem dijelu stranice nalazi se navigacijsko okno, a u donjem desnom kutu je gumbić koji omogućuje chat s drugim korisnicima aplikacije.

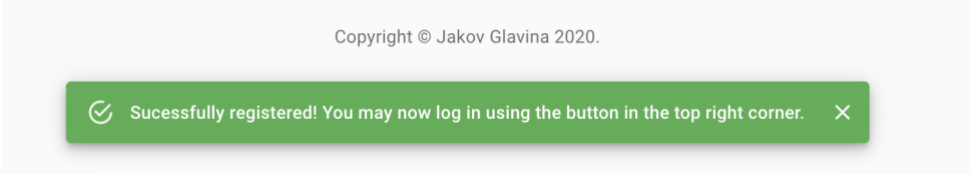


Slika 17. Početna stranica

Klikom na *Register* u gornjem desnom uglu, korisnik se može registrirati, pri čemu se dinamički validira unos u formu.



Slika 18. Registracija



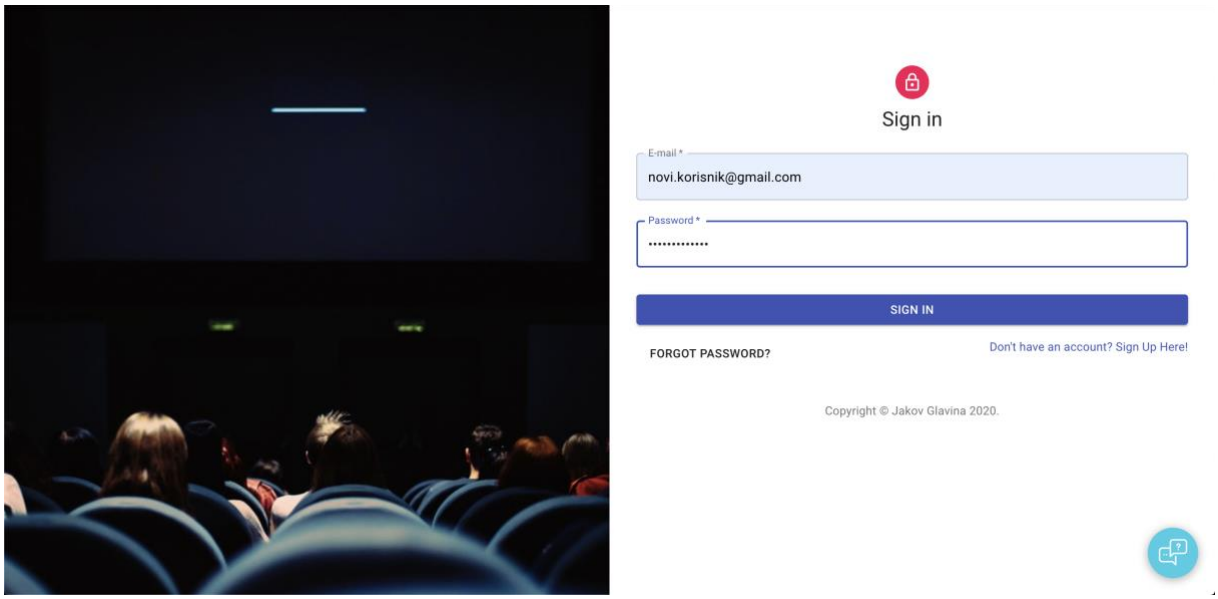
Slika 19. Povratna informacija o uspješnoj registraciji

Na korisnikove akcije aplikacija vraća prikladnu povratnu informaciju, kao na slici 18. Za implementaciju ove povratne informacije korišteni su Snackbar i Alert elementi Material UI biblioteke (biblioteka za razvoj korisničkih sučelja).

```
1. <Snackbar open={open} autoHideDuration={4000} onClose={handleClose}>
2.          <Alert onClose={handleClose} severity="success">
3.            Sucessfully registered! You may now log in using the button
   in the top right corner.
4.          </Alert>
5.        </Snackbar>
```

Komponenti Snackbar se pridružuju svojstva koja određuju hoće li biti otvoren (`open`), vrijeme potrebno da se automatski skrije (`autoHideDuration`), i upravljač događaja `onClose`.

Nakon registracije, korisnik se prijavljuje, nakon čega je prebačen ponovno na početnu stranici. Naravno, u slučaju da je korisnik zaboravio lozinku, može poslati zahtjev za resetiranjem lozinke, nakon čega će mu na e-mail korišten za registraciju biti isporučena nova lozinka.



Slika 20. Prijava

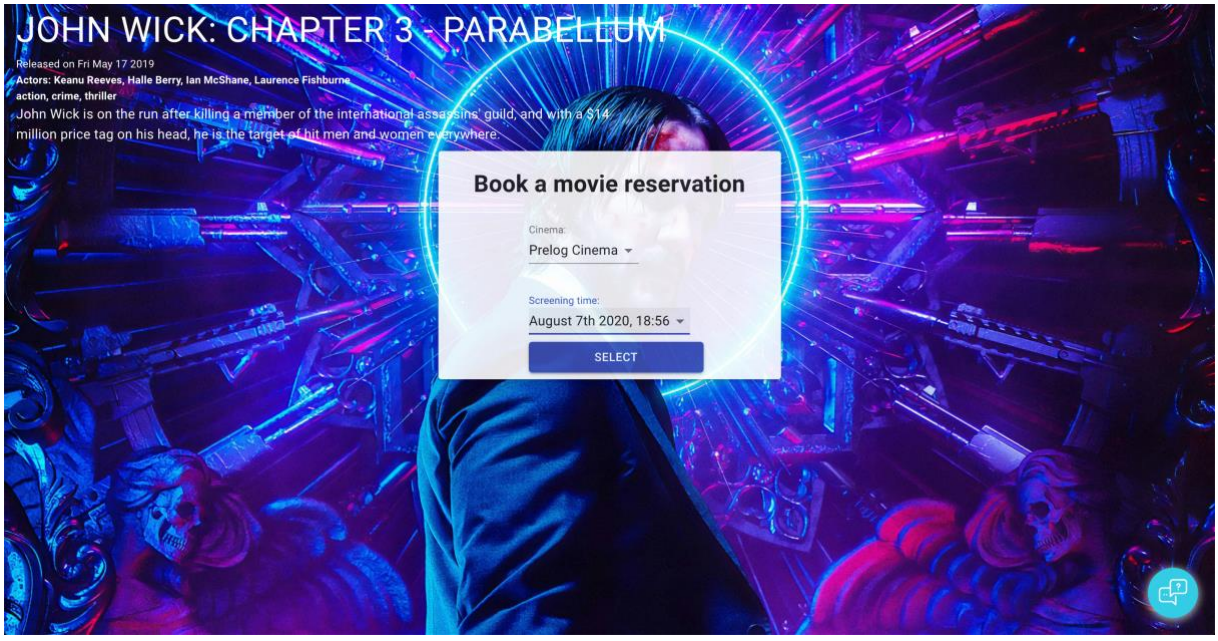
Sljedeći korisnikov korak je da otiđe na pregled filmova i odabere film koji želi gledati, tj. za kojeg želi napraviti rezervaciju. Važno je napomenuti da je za pravljenje rezervacije potrebno biti ulogiran, inače korisnik neće moći napraviti rezervaciju. Stranica za pregled filmova implementirana je na način da se dohvaćaju podaci o filmovima iz baze podataka te se na

temelju toga dinamički kreira stranica čiji atraktivni dizajn je postignut činjenicom da koristi tzv. „parallax“ efekt, koji stvar ugođaj prelijevanja sadržaja iz jednog dijela u drugu.

Slika 21. Pregled filmova

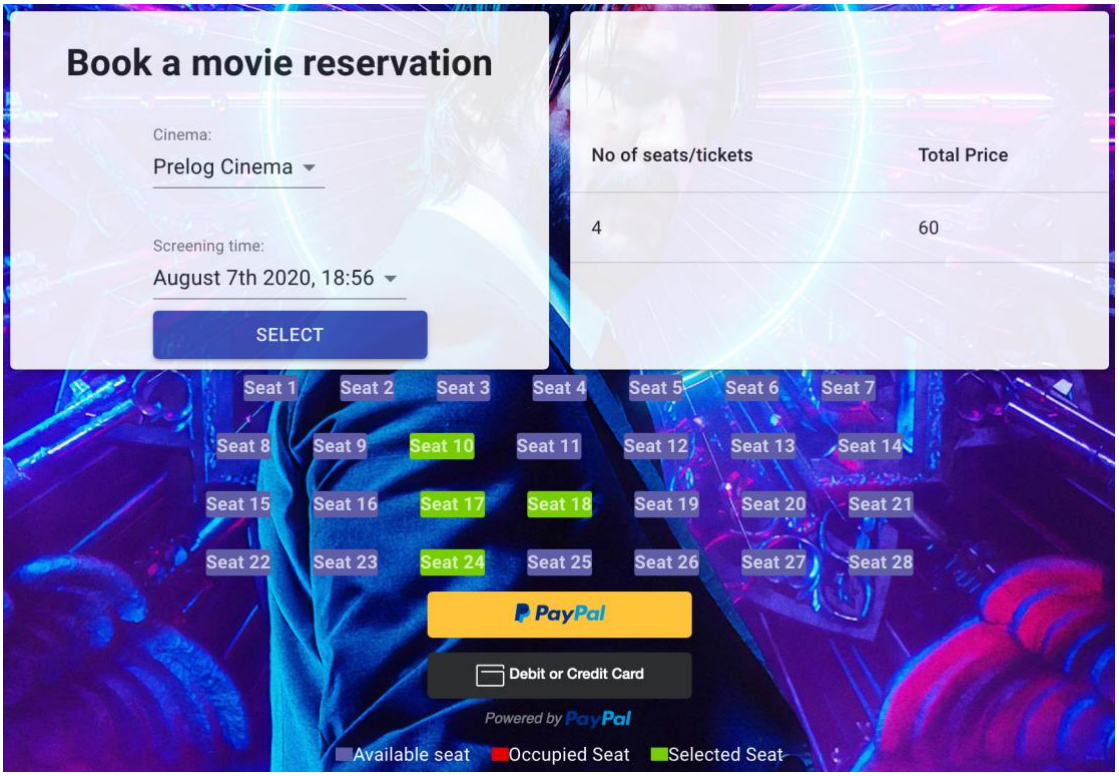
4.8.3. Registrirani korisnik

Nakon što je odabran film, istodobno uzimajući u obzir da se korisnik prijavio, možemo službeno započeti proces rezervacije. Prva stvar koju registrirani korisnik mora učiniti je odabrati kino i dostupna vremena prikazivanja za to kino:



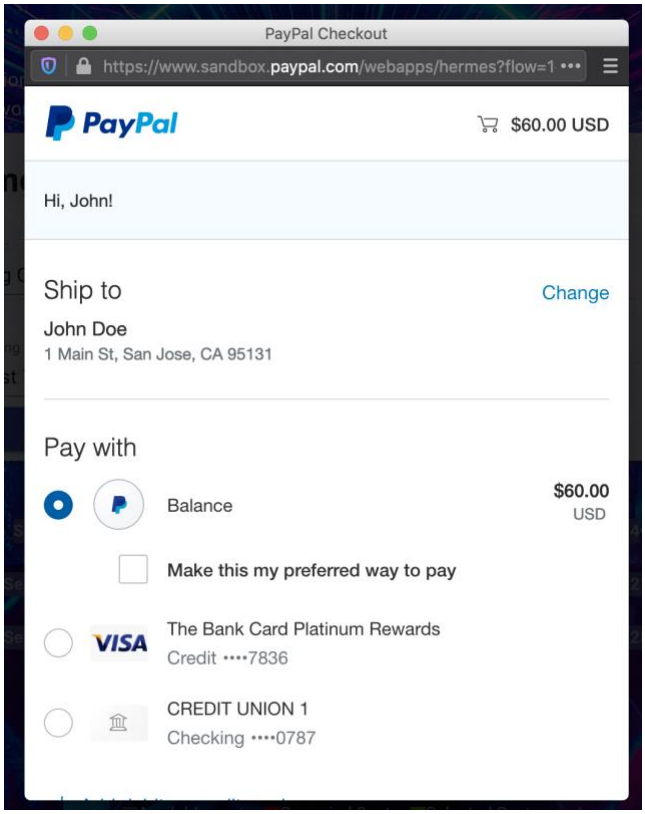
Slika 22. Odabir kina i vremena prikazivanja

Otvara se prikaz svih sjedala u kinu te korisnik može odabrati jedno ili više sjedala. Nezauzeto mjesto označeno je ljubičastom bojom, trenutno označeno mjesto zelenom bojom, a zauzeto mjesto crvenom bojom. Da bi se to olakšalo i pojasnilo korisniku, ispod pregleda sjedala nalazi se i legenda s opisom koja boja korespondira kojem tipu sjedala. S desne pak se strane dinamički prikazuje broj zauzetih mjesta te iznos koji će korisnik platiti da izvrši rezervaciju. Zatim, kada korisnik odabere sva mjesta koja želi zauzeti, ponuđene su mu dvije opcije za plaćanje – PayPal i klasična debitna/kreditna kartica. No, kako je ovaj koncept plaćanja razvijen u demonstracijske svrhe, mogućnost plaćanja koristeći debitnu/kreditnu karticu trenutno nije funkcionalna, nego svaki korisnik treba rezervaciju platiti koristeći PayPal.



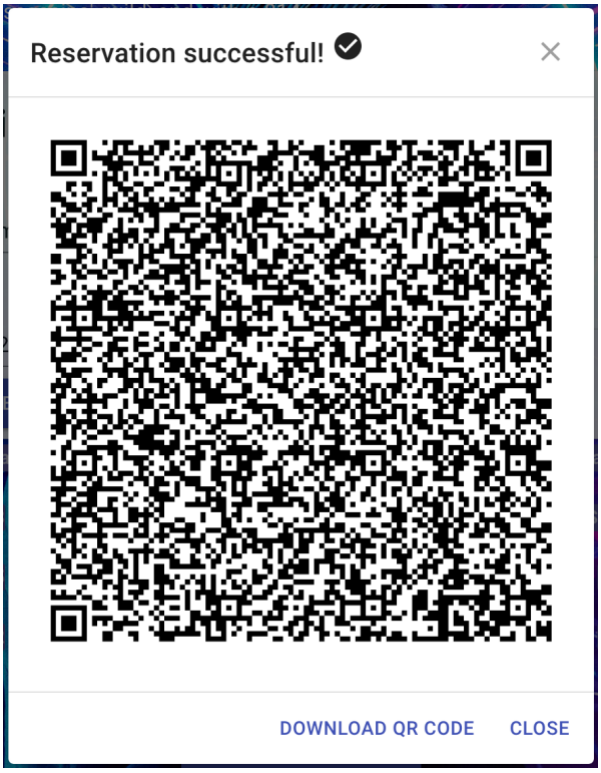
Slika 23. Odabir sjedala

Klikom na žuti gumb PayPal otvara se modal servisa PayPal u kojem korisnik unosi svoje PayPal podatke te podešava sve svoje parametre za provedbu plaćanja.



Slika 24. Podešavanja parametara plaćanja

Nakon uspješnog plaćanja korisnik treba stisnuti na gumb *FINISH RESERVATION* da bi završio svoju rezervaciju i primio QR kod kojeg može spremiti na lokalno računalo (slika 25). Ovaj QR kod služi korisniku da ga pokaže zaposleniku kina kada dođe gledati film, tako da ga zaposlenik kina može propustiti netom prije početka filma.

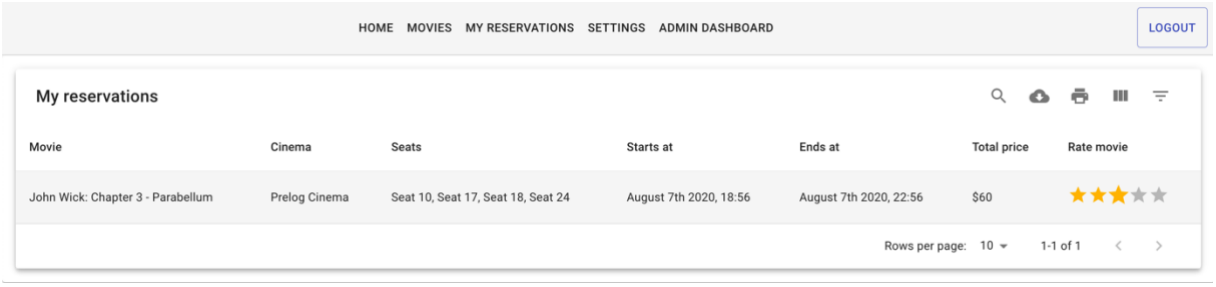


Slika 25. QR kod sa podacima o rezervaciji

Konkretno, QR kod na slici 24 skriva sljedeće podatke o rezervaciji:


```
{
  "seats": [
    {
      "id": 10,
      "seat_name": "Seat 10",
      "occupied": 2
    },
    {
      "id": 17,
      "seat_name": "Seat 17",
      "occupied": 2
    },
    {
      "id": 18,
      "seat_name": "Seat 18",
      "occupied": 2
    },
    {
      "id": 24,
      "seat_name": "Seat 24",
      "occupied": 2
    }
  ],
  "screeningTime_id": "5f244c09f9952205f6c2973c",
  "user_id": "5f23b13532bb5f04ef4138f3",
  "totalPrice": 60,
  "paypalOrderId": "3DX23063C38929543",
  "id": "5f2454fadacb1c06872b1547"
}
```

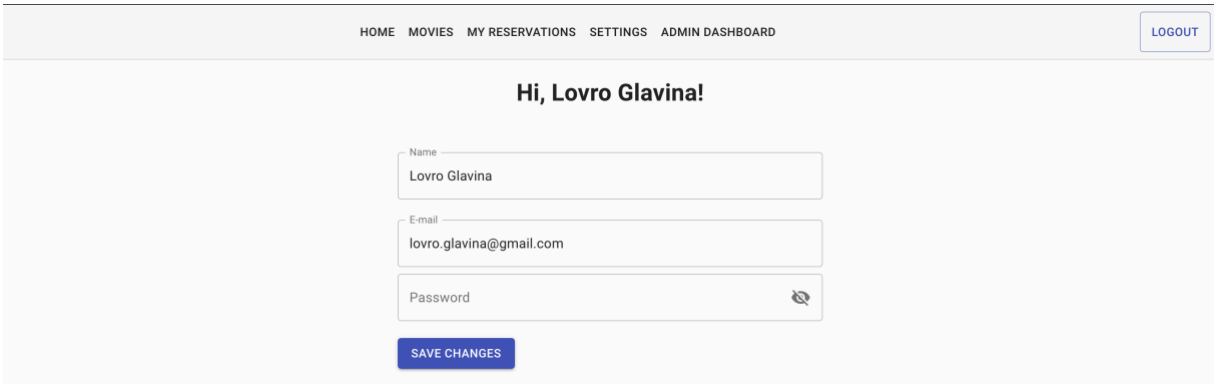
Ako opet bacimo pogled na navigacijsko okno, primijetit ćemo da registrirani korisnik ima pristup pregledu svih svojih rezervacija:



Slika 26. Pregled rezervacija

Registrirani korisnik, za svaku od svojih rezervacija, može vidjeti na koji film i kino se ista odnosi, koja sjedala je rezervirao, te početak i kraj vremena prikazivanja koje je rezervirao. Naposljetku, vidljiva mu je ukupna cijena koju je platio te ima mogućnost ocjenjivanja filma. Dakle, korisnik može ocijeniti samo one filmove za koje je napravio rezervaciju. Nadalje, budući da sam za ovu mogućnost koristio npm modul *mui-datatables*, ovaj tablični prikaz ima još neke korisne mogućnosti, a koje uključuju pretragu zapisa tablice, prikaz/skrivanje pojedinih stupaca tablice, filtriranje zapisa po određenim vrijednostima, izvezivanje (eng. *export*) sadržaja tablice u CSV (eng. ***C**omma-**S**eparated **V**alues*) obliku i generiranje prikaza za ispis.

Registrirani korisnik ima i sekciju *Settings* gdje može mijenjati svoje osobne podatke, a podaci koje može mijenjati su puno ime i prezime, e-mail te lozinka.

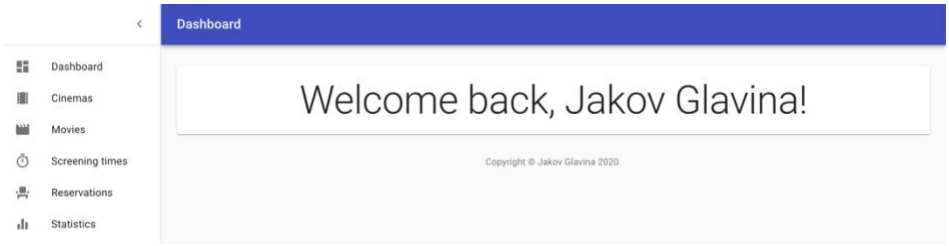


Slika 27. Mijenjanje osobnih informacija (registrirani korisnik)

4.8.4. Administrator

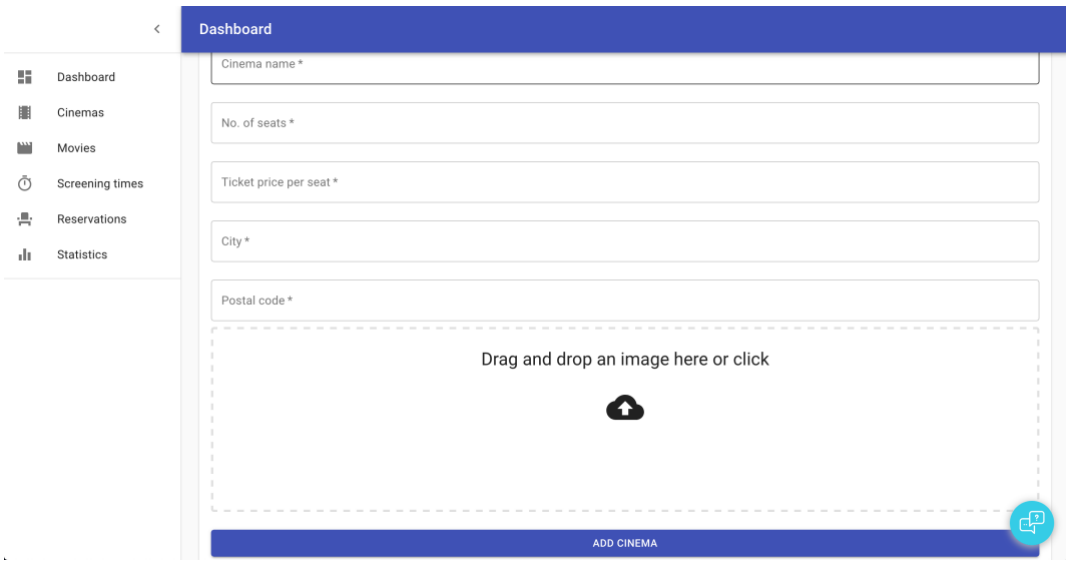
Naposljetku, imamo ulogu administratora koja upravlja svim bitnim elementima aplikacije, tj. upravlja svim kinima, filmovima i vremenima prikazivanja te za njih ima **C**reate,

Read i **D**elete kontrole. Za ulogu administratora je posebno razvijena kontrolna ploča koja sa svoje lijeve strane ima šest distinktnih sekcija: *Dashboard* (početna), *Cinemas*, *Movies*, *Screening times*, *Reservations* i *Statistics*.

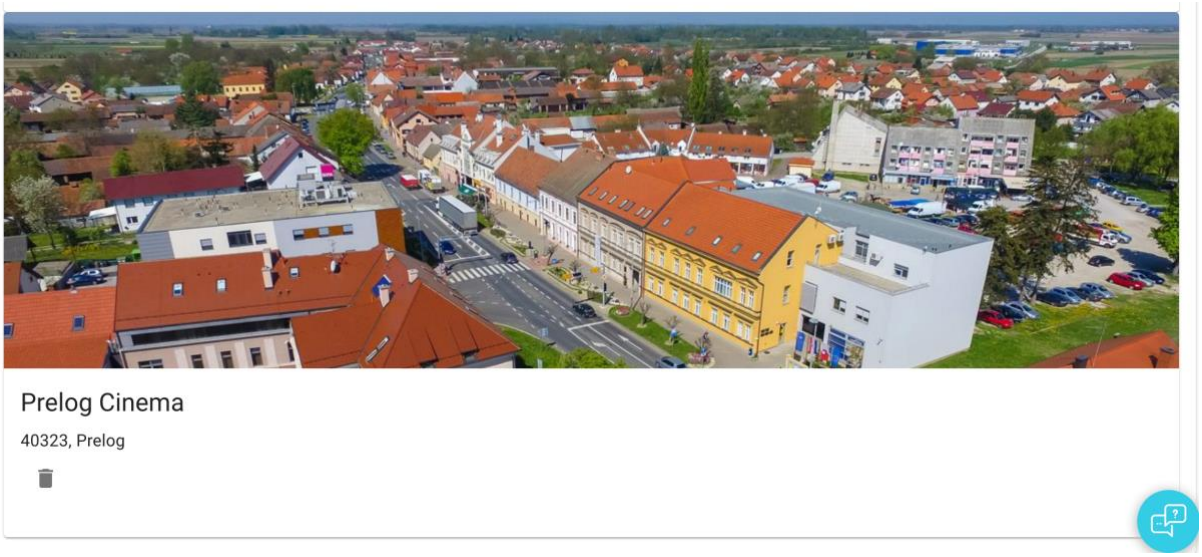


Slika 28. Početni ekran administratorske nadzorne ploče

Kod dodavanja novih kina, administrator mora upisati podatke o: imenu kina, ukupnom broju sjedala, cijeni ulaznice po sjedalu, gradu u kojem je kino te poštanski broj pripadnog mjesta. Također treba staviti i sliku kina koja je vidljiva samo njemu. Za dodavanje slike korišten je modificirani *file picker* widget koji se drži smjernica Material UI dizajna. Konkretno, npm modul koji sam koristio za navedeni *file picker widget* se zove *material-ui-dropzone.*



Slika 29. Dodavanje novog kina



Slika 30. Dodano kino

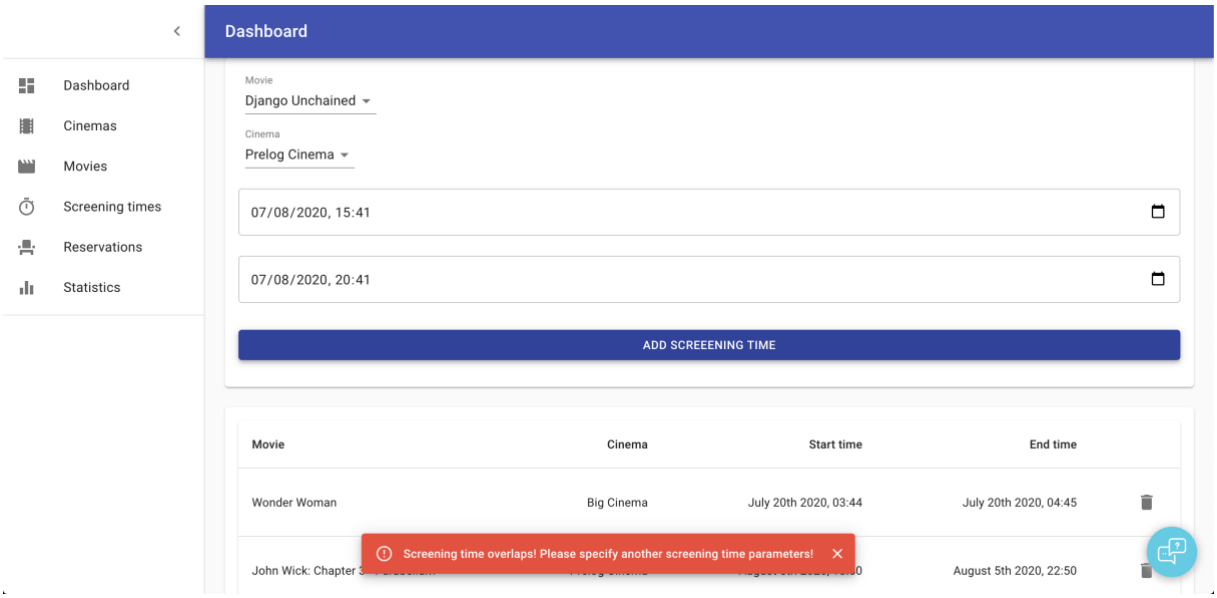
Što se tiče filmova, situacija je prilično slična uz male razlike. Prva i najveća razlika je u tome što forma za unos novog filma ima jedno jedini polje: **ime filma**. Osim imena filma, korisnik treba dodati i pozadinsku sliku (eng. *backdrop image)* film, a ista se koristi kod prikaza svih filmova koje vidi uloga neregistrirani korisnik i više. Kada administrator pritisne na *Add movie*, zahtjev s imenom filma i slika se šalje na backend gdje se radi zahtjev prema OMDb API-ju, neslužbenom API-ju IMDb-a, tj. najveće „baze podataka“ za filmove. Na temelju imena filma, pronalazi se pripadajući film i dohvaćaju se mnogobrojni podaci o filmu, no u bazu spremam samo ono što je potrebno na stranici za prikaz filma, a to je: **ime filma, režiser, poster, pozadinska slika, žanrovi, glumci, kratak opis filma, datum izlaska i ocjena publike**.


```
_id: ObjectId("5f102a352b330c0ef6a6a23f")
title: "Sunshine"
director: "Danny Boyle"
coverArt: "https://m.media-amazon.com/images/M/MV5BMTU5Nzg2OTk2NF5BMl5BanBnXkFtZT..."
backdropImage: "http://res.cloudinary.com/cinemaapp/image/upload/v1594894900/ckxzq6uiz..."
genre: "Sci-Fi, Thriller"
actors: "Cliff Curtis, Chipo Chung, Cillian Murphy, Michelle Yeoh"
summary: "50 years into the future, the Sun begins to die, and Earth is dying as..."
releaseDate: 2007-07-26T22:00:00.000+00:00
rating: 7
```


Slika 31. Primjer zapisa filma nakon dodavanja u bazu

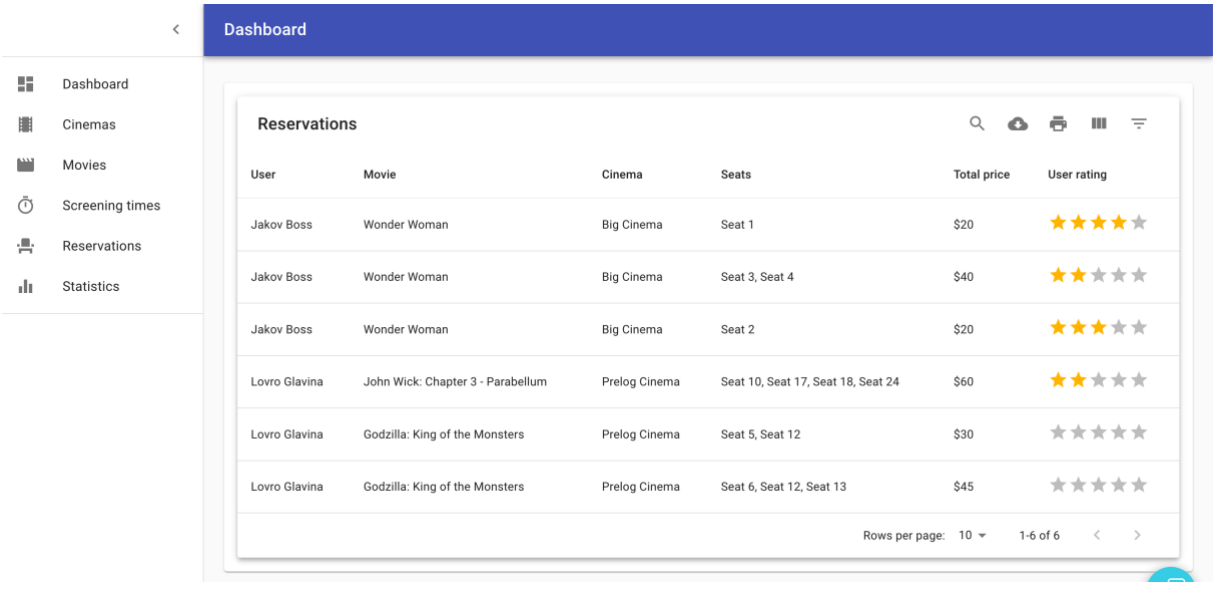
Isto tako, administrator može dodati i novo vrijeme prikazivanja pri čemu odabire na koji film i na koje kino se odnosi vrijeme prikazivanja te početak i kraj vremena prikazivanja. Na klijentskoj strani je ugrađena validacija unosa u smislu da se prije slanja novog vremena prikazivanja na backend, provjerava postoji li kakva kolizija s već postojećim vremenima

prikazivanja. Ako postoji, slanje forme se prekida i korisniku se prikazuje upozorenje u snackbaru koji ga upozorava na to da ne može dodati takvo vrijeme prikazivanja te da treba promijeniti unos.



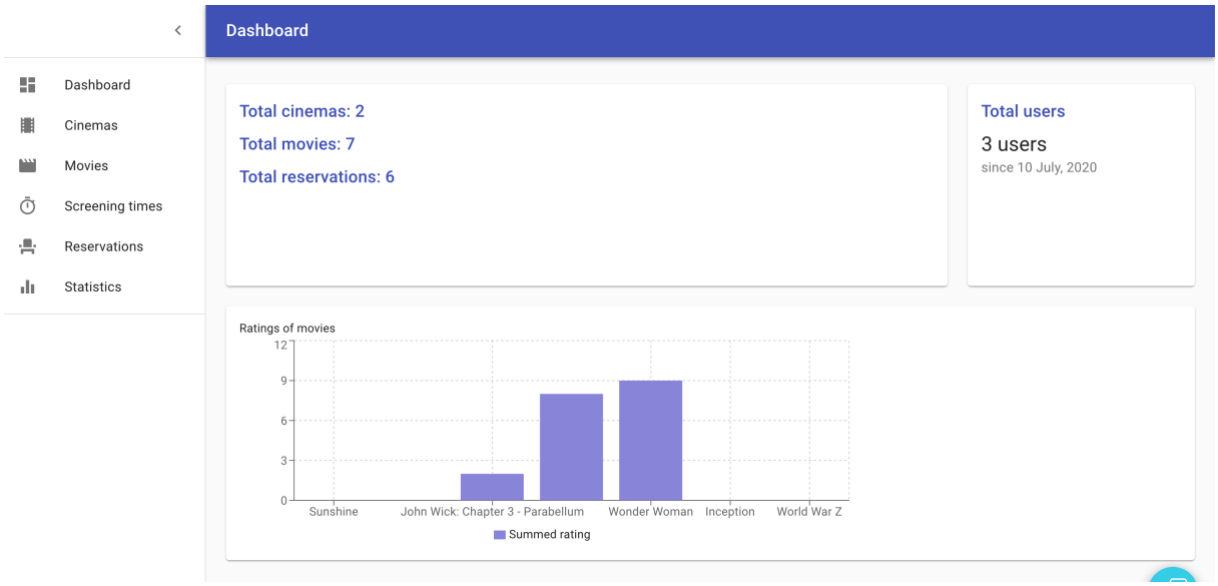
Slika 32. Dodavanje novog vremena prikazivanja i greška kod dodavanja

Kao i registrirani korisnik koji je napravio neke rezervacije, tako i administrator ima prikaz rezervacija u kojem može vidjeti tko je rezervaciju napravio, za koji film, koja mjesta je rezervirao, koja je ukupna cijena rezervacije, te koju ocjenu je korisnik dao filmu. Što se tiče posebnih mogućnosti za tablicu, analogne su onima kod pregleda rezervacija individualnog korisnika.



Slika 33. Pregled rezervacija - administrator

Naposljetku, u kartici *Statistics,* administrator ima pregled nekih statistika kao što su broj ukupnih filmova, rezervacija, kina, registriranih korisnika te grafički prikaz ukupnih ocjena filmova (od strane korisnika) po filmu.



Slika 34. Statistike u administratorskoj kontrolnoj ploči

4.9. Stavljanje aplikacije na web

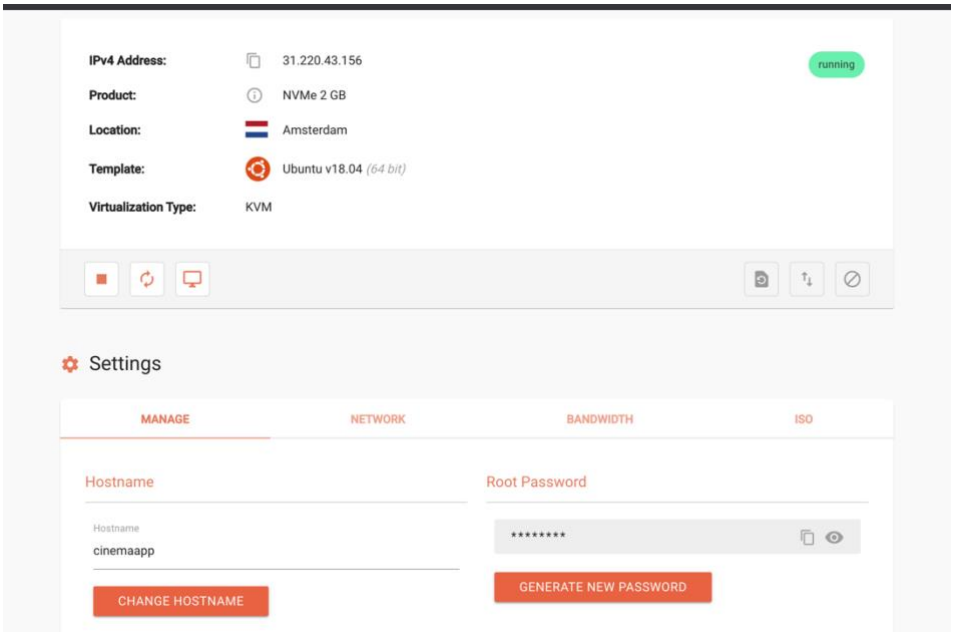
Na samom kraju, kada smo gotovi s razvijanjem aplikacije na lokalnom računalo, spremni smo postaviti ju na web. Pošto smo razvijali aplikaciju u dva distinktna dijela, tj. frontend i backend, i to na lokalnom računalo, nije bilo teško „spojiti“ ta dva dijela. Bilo je potrebno pokrenuti poslužitelj na backendu te pokrenuti skriptu za pokretanje aplikacije na frontendu da bi aplikacija radila. Budući da su i frontend i backend na istom poslužitelju, to je također značilo da u frontend projektu aplikacije nije bilo potrebno posebno modificirati putanje poziva prema backend dijelu aplikacije, nego je bilo dovoljno napisati npr. `axios.get('/api/cinemas/')` da bi dohvatili sva kina u aplikaciji. Kada ima mnogo poziva prema backendu, to nedvojbeno smanjuje količinu kompleksnosti i eventualno količinu dodatnog koda kojeg bi trebalo pisati da se uskladi frontend i backend.

Kada je riječ o stavljanju aplikacije na Web, kao i za sve ostalo u ovom radu postoji više načina. Jedan od tradicionalnijih načina je nabavka fizičkog poslužitelja ili unajmljivanje istog (tzv. *Virtual Private Server*, *VPS*) čija prednost je potpuna sloboda i kontrola nad okruženjem na kojem se aplikacija vrti. S druge strane, nedostatak ovog pristupa je to što

zahtijeva malo veću pismenost sa „sistemaške“ strane, jer ovaj pristup zahtijeva da programer sve potrebne alate i servise na poslužitelj instalira i konfigurira ručno.

Ustajalost ovog pristupa utrla je put novoj generaciji i novom načinu stavljanja web aplikacija na Web, a to su tzv. *Platform-as-a-Service* (PaaS) proizvodi koji pojednostavljuju i nerijetko u potpunosti automatiziraju ovaj „sistemaški“ dio koji uključuje konfiguraciju aplikacijskog okruženja. Neki od popularnih servisa s ovom svrhom uključuju Heroku, Netlify, Vercel, Azure i sl. Ja sam odabrao klasični, tradicionalniji pristup s VPS-om iz prostog razloga što sam bolje upoznat s tim načinom.

Prvi korak bio je unajmiti VPS kod nekog od mnoštva pružitelja te usluge, a ja sam odabrao primjerice HostHatch. Nakon unajmljivanja servera, na kontrolnoj ploči HostHatch servisa vidimo informacije o našem VPS-u kao što je primjerice **IP adresa** (31.220.43.156), **količina diskovne pohrane** (2 GB), **lokacija** (Amsterdam), **operacijski sustav** (Ubuntu 18.04) te **tip virtualizacije** (KVM).



Slika 35. Kontrolna ploča servisa HostHatch

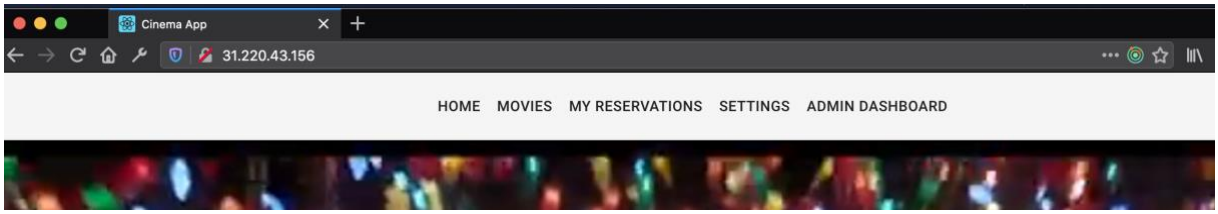
Prije nego krenemo dalje, kreirat ćemo tzv. *production build* naše React aplikacije. Svo ovo vrijeme dok smo programirali i razvijali aplikaciju, React se izvršavao u development načinu, što znači da je npr. imao mnogo povratnih poruka za vrijeme izvršavanja koji programeru uvelike olakšavaju proces otkrivanja pogrešaka budući da su vrlo opširni i programeru daju uvid u to u čemu bi mogao biti problem. Production build se kreira izvršavanjem npm komande `npm run build` koja u korijenskom direktoriju projekta kreira *build* mapu koja cjelovitu React aplikaciju u obliku statičkih datoteka, a koja je optimizirana za izvršavanje na poslužitelju i ima bolje performanse od development verzije aplikacije. Nakon

što je kreirana produkcijska verzija React aplikacije, kopirat ćemo *build* mapu koju smo dobili u korijenski direktorij našeg backend projekta. To će nam omogućiti da iskoristimo Express kako bi prikazivali statičke datoteke koristeći ugrađenu middleware funkciju `express.static()` koja služi upravo tome, za serviranje statičkih datoteka.

```
1. app.use(cors({ credentials: true, origin:
   process.env.REACT_APP_URL }));
2. app.use(bodyParser.urlencoded({ extended: true }));
3. app.use(express.json());
4. app.use(express.static(path.join(__dirname, 'build'))); // serviranje
   statičkih datoteka, tj. gotove React aplikacije
5.
6. app.use('/api/movies', moviesRouter);
7. app.use('/api/users', usersRouter);
8. app.use('/api/login', loginRouter);
9. app.use('/api/cinemas', cinemasRouter);
10.  app.use('/api/screeningtimes', authMiddleware,
   screeningTimesRouter);
11.  app.use('/api/misc', miscRouter);
12.  app.use('/api/reservations', reservationsRouter);
13.
14.  app.get('*', (req, res) => {
15.    res.sendFile(path.join(__dirname, 'build/index.html'));
16.  });
```

Budući da i sama React aplikacija koristi vlastitu soluciju za usmjeravanje, tj. prikazivanje pojedinih stranica ovisno o tome koji URL je u adresnoj traci, potrebno je u Express aplikaciji „uloviti“ zahtjeve koji nisu usmjereni API-ju te ih preusmjeriti na index.html (React aplikaciju) da bi ona mogla obaviti svoj dio usmjeravanja (linije 14-16). Također, da bi naša aplikacija pravilno radila na VPS-u, a ne samo na lokalnom računalu, bilo je potrebno napraviti dodatne promjene na backendu. Prvo, dodana je `REACT_APP_URL` varijabla okruženja s vrijednošću IP adrese VPS-a te je proslijeđen dodatni konfiguracijski parametar middleware funkciji `cors()` da se isprave pogreške vezane za CORS kod izvođenja aplikacije (linija 1).

Sada nam je preostalo kopirati cijeli sadržaj korijenskog direktorija backend projekta na VPS. Na VPS je potrebno instalirati node.js i npm te potom u kopiranoj mapi izvršiti komandu `npm install`. Ova komanda će instalirati sve potrebne biblioteke za rad aplikacije, shodno onima koje su definirane u *package.json* datoteci. Na kraju, aplikacija se pokreće s komandom `node index.js &` što znači da će se ovaj proces pokrenuti u pozadini.



Slika 36. Aplikacija pokrenuta na VPS-u

5. Zaključak

U ovom završnom radu objasnio se i na primjeru prikazao cjelokupni tijek (eng. *flow)* razvoja jedne složenije web aplikacije – prvo su opisani najbitniji koncepti potrebni za razumijevanje principa na kojem je implementirana cjelokupna web aplikacija, a to su primjerice razlika između statičkih i dinamičkih web stranica, način funkcioniranja virtualnog DOM-a kod Reacta i mnoge druge. Opisane su i glavne tehnologije korištene za razvoj aplikacije, tehnologije koje su elementarne sastavnice MERN razvojnog okvira, a to su baza podataka – **MongoDB,** razvojni okvir za izvršavanje serverskih aplikacija na poslužitelju – **Express**, biblioteka za razvoj i implementaciju korisničkih sučelja na klijentskoj strani – **React.js**, te naposljetku okruženje za izvršavanje JavaScript aplikacija izvan preglednika, tj. primjerice na poslužitelju – **Node.js.**

Razvoj web aplikacije pomoću MERN razvojnog okvira omogućuje razvijanje vrlo fleksibilnih i skalabilnih web aplikacija. Ako je programer vrstan u svojem zvanju, u stanju je napraviti aplikacije koje nisu teške za održavanje na većoj skali, a to se na primjeru vidi kod Reacta, gdje se kreiranjem mnoštva ponovno iskoristivih komponenti olakšava njihovo ponovno korištenje i implementacija u raznim drugim dijelovima aplikacije, ali se zbog toga olakšava i pronalaženje grešaka u istima. Korištenjem JSON Web Tokena prikazana je alternativna implementacija autentikacije/autorizacije korisnika pri korištenju Web aplikacije gdje se informacije o korisniku šalju sa svakim zahtjevom prema poslužitelju, a ne čuvaju se u sesiji. Važno je napomenuti da ovo posebno pomaže skalabilnosti aplikacije iz razloga što ako se koristi sesija, ona se po svojoj naravi čuva na poslužitelju i samim time zauzima poslužiteljske resurse. Ako je aplikacija na većoj skali i ima mnoštvo korisnika, s vremenom to može dovesti do poteškoća u izvođenju aplikacije, pa se korištenjem JSON Web Tokena poslužitelja „odrješuje“ tog tereta i u zamjenu se to stavlja na svakog pojedinog klijenta. Nadalje, kroz korištenja mnoštva različitih npm modula (uglavnom modula koji pružaju implementaciju raznih React komponenti) pokazana je moć i lakoća korištenja npm ekosustava. Ukoliko želimo koristiti neku React komponentu koja ostvaruje malo kompliciraniju zadaću, primjerice pokreće YouTube video u pozadini, za to postoji *react-youtube-background*

npm modul koji implementira upravo tu funkciju. To pomaže u tome da programer na treba ponovno „otkrivati vatru“, nego može iskoristiti implementacije drugih ljudi koje su oni stavili na Internet da budu otvoreno dostupne. Međutim, ovdje valja istaknuti da se programer mora paziti toga da ne iskoristi previše različitih biblioteka i npm paketa. Na što se ovdje misli? Pa, na neki način dostupnost ovakvog mnoštva paketa može rezultirati time da se programer „ulijeni“ te počinje koristiti pakete i biblioteke za stvari čija implementacija je manje ili više trivijalna te za koju nema potrebe koristiti pakete i biblioteke. Programer, prilikom izrade aplikacije, mora pronaći zlatnu sredinu između previše i premalo npm paketa da bi to rezultiralo što kvalitetnijom i što urednijom aplikacijom. Također je pokazana implementacija WebSocket tehnologije u obliku Socket.IO biblioteke koja omogućuje komunikaciju korisnika aplikacije u realnom vremenu.

Unatoč tome što sam već u prošlosti bio upoznat s razvojem web aplikacija u programskom jeziku PHP te korištenjem tradicionalnog HTML/CSS/JS, razvoj web aplikacije u MERN razvojnom okviru me potaknuo na učenje novih tehnologija, novih paradigmi i principa programiranja, te sam razvojem ove aplikacije stekao neprocjenjivo iskustvo rješavanja problema i razvijanja web aplikacije na nov način. Smatram da će mi svo novostečeno znanje i iskustvo koje je proizašlo kao rezultat izrade ovog završnog rada uvelike pomoći da budem kompetentniji u svojoj struci i na tržištu rada.

Popis literature

[1] Mozilla.org (bez dat.), *Inheritance and the prototype chain*, dostupno 3.8.2020. na https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Inheritance_and_the_prototype_chain

[2] Herman, D. (2013.) *Effective Javascript*, Addison-Wesley. str. 83.

[3] Guru99.com (bez dat.), *SQL vs NoSQL: What's the difference?*, dostupno 3.8.2020. na https://www.guru99.com/sql-vs-nosql.html

[4] GitHub, Arhiva node.js verzija, *node-v0.x-archive*, dostupno 3.8.2020. na https://github.com/nodejs/node-v0.x-archive/tags?after=v0.0.4

[5] Stack Overflow (2015.), *What are the benefits of Express over plain Node.JS?*, dostupno 3.8.2020. na https://stackoverflow.com/questions/28823253/what-are-the-benefits-of-express-over-plain-node-js

[6] OpenJS Foundation (bez dat.), *Express.js API Reference*, dostupno 3.8.2020. na http://expressjs.com/en/api.html

[7] Facebook Inc. (bez dat.), *React.js Docs*, dostupno 3.8.2020. na https://reactjs.org/docs/

[8] Google (bez dat.), *AngularJS: Developer Guide, The Zen of AngularJS,* dostupno 3.8.2020. na https://docs.angularjs.org/guide/introduction

[9] Django Software Foundation (bez dat.), *Django: Design philosophies*, dostupno 3.8.2020. na https://docs.djangoproject.com/en/2.0/misc/design-philosophies/

[10] *Ruby on Rails* (bez dat.), dostupno 3.8.2020. na https://rubyonrails.org

[11] VMware Inc., *Spring* (bez dat.), dostupno 3.8.2020. na https://spring.io

[12] Creative Commons, *BSON Specification Version 1.1.* (bez dat.), dostupno 3.8.2020. na http://bsonspec.org/spec.html

.

Popis slika

Popis tablica