Web aplikacija za vođenje kina

Bošnjaković, Denis

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU

FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Preddiplomski stručni studij računarstvo

WEB APLIKACIJA ZA VOĐENJE KINA

Završni rad

Denis Bošnjaković

Osijek, 2023.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

Obrazac Z1S: Obrazac za imenovanje Povjerenstva za završni ispit na preddiplomskom stručnom studiju

Osijek, 06.09.2023.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za završni ispit na preddiplomskom stručnom studiju

Ime i prezime Pristupnika:	Denis Bošnjaković
Studij, smjer:	Računarstvo
Mat. br. Pristupnika, godina upisa:	AR 4714, 19.07.2019.
OIB Pristupnika:	08629434362
Mentor:	Robert Šojo, mag. ing. comp.
Sumentor:	,
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	mr. sc. Željko Štanfel
Član Povjerenstva 1:	Robert Šojo, mag. ing. comp.
Član Povjerenstva 2:	Marina Peko, dipl. ing.
Naslov završnog rada:	Web aplikacija za vođenje kina
Znanstvena grana završnog rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak završnog rada	Web aplikacija koja omogućava vođenje kina. Postoje tri uloge korisnika koji koriste web aplikaciju. Administrator imamo mogućnost dodavanje filmova, kreiranje rasporeda projekcije filmova, vođenje skladišta pića, grickalica te suvenira. Registrirani korisnik ima mogućnost rezerviranja projekcije, vrste dvorane, odabira mjesta sjedenja, također odabir pića i ostale ponude. Gost ima samo mogućnost pregleda web stranice, tj. filmova i uvid u raspored projekcije. Korisnici mogu komentirati i ocjenjivati filmove. Postavljanje filmova i definiranje njihove projekcije koje trebaju biti vidljive u rasporedu, postaviti također najave filmova koji će se uskoro prikazivati. Student treba razviti funkcionalnu aplikaciju koristeći različite web tehnologije. Rezervirano za studenta: Denis Bošnjaković
Prijedlog ocjene pismenog dijela ispita (završnog rada):	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene od strane mentora:	06.09.2023.

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

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA **OSIJEK**

IZJAVA O ORIGINALNOSTI RADA

Osijek, 29.02.2024.

Ime i prezime studenta:	Denis Bošnjaković
Studij:	Računarstvo
Mat. br. studenta, godina upisa:	AR 4714, 19.07.2019.
Turnitin podudaranje [%]:	5

Ovom izjavom izjavljujem da je rad pod nazivom**: Web aplikacija za vođenje kina**

izrađen pod vodstvom mentora Robert Šojo, mag. ing. comp.

i sumentora ,

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

Potpis studenta:

Sadržaj

1. UVOD

Aplikacija za vođenje kina može otključati brojne pogodnosti za vlasnike kina i publiku. Ona rješava brojne probleme kao što su smanjenje ljudske pogreške, smanjenje količine ručnih zadataka i optimizaciju operacija te može pomoći u poboljšanju ukupne učinkovitosti kina. Ljudska pogreška je nešto što se ne može izbjeći kada operacijama upravlja osoblje, ali kada se takve greške dogode, one mogu imati potencijalno katastrofalne posljedice za poslovanje. Kada se žele smanjiti potencijalne greške koje mogu nastati od strane osoblja, najbolji mogući način na koji se to može riješiti je automatizacijom glavnih procesa koji se odvijaju u kinu. Također, smanjenjem broja zadataka koje treba obaviti ručno, manje osoblja može biti dostupno jer je potrebno manje ruku na licu mjesta. Praćenjem prodaje ulaznica, upravljanjem zaliha i rukovanjem podacima o klijentima, aplikacija može pomoći da svaki element poslovanja teče glatko što dugoročno može uštedjeti vrijeme i novac.

U ovom radu bit će objašnjen postupak realizacije *web* aplikacije, usporedba s postojećim aplikacijama slične namjene te prednosti i nedostaci *web* aplikacije. Nakon usporedbe u drugom poglavlju, u trećem poglavlju opisani su korišteni programski jezici i tehnologije potrebne za realizaciju poslužiteljskog i klijentskog dijela aplikacije. U četvrtom poglavlju opisana je odabrana arhitektura aplikacije, a kroz primjere programskog koda opisani su i razni načini implementacije funkcionalnosti. Na kraju, u petom poglavlju, opisane su upute za korištenje *web* aplikacije.

1.1. Zadatak završnog rada

Zadatak završnog rada je razviti *web* aplikaciju koja omogućava vođenja kina. Aplikacija razlikuje tri uloge korisnika koji ju koriste. Neregistrirani korisnici imaju mogućnost pregleda *web* stranice, filmova te uvid u dostupne projekcije. Informacije o filmu uključuju prikaz godine izlaska filma, trajanje filma, žanrove, glumce i redatelje. Registrirani korisnici mogu rezervirati projekcije, odabrati dvoranu i mjesto sjedenja te piće, grickalice i suvenire. Korisnici mogu komentirati i ocjenjivati filmove. Administratori imaju mogućnost uklanjanja, uređivanja i dodavanja žanrova, filmova, glumaca, sjedala, dvorana te vođenje skladišta pića, grickalica i suvenira. Administratori također određuju datume projekcija u rasporedu, određuju prikazivanje filmova na početnoj stranici i najave filmova koji će se uskoro prikazivati.

2. PREGLED PODRUČJA

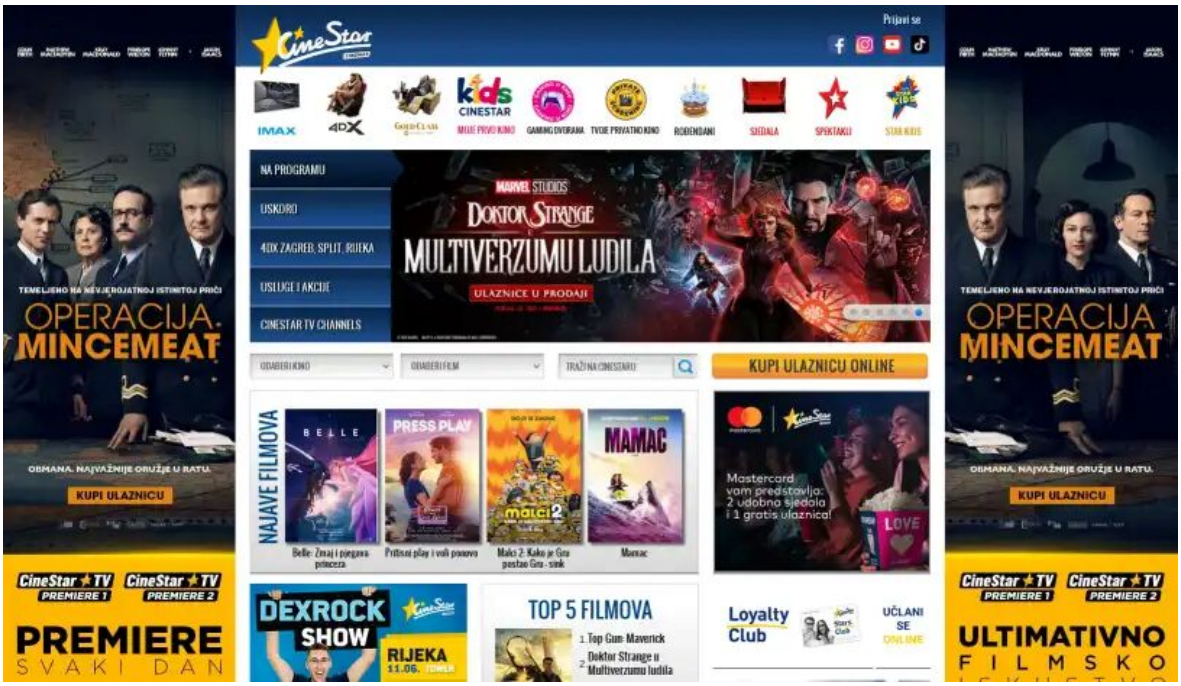
U ovom poglavlju prikazana su i opisana slična rješenja *web* (mrežnih) aplikacija za kino i/ili prikazivanje informacija o filmovima, vijestima iz filmskog svijeta te mogućnosti kupnje ulaznice za film. Također, opisana je i tehnologija koja napaja spomenuta rješenja.

2.1. Postojeća rješenja

2.1.1. CineStar

CineStar grupa svoje početke bilježi još davne 1948. godine, kada je tvrtka *Kieft & Kieft Filmtheater GmbH* otvorila svoje prvo klasično kino u *Lübecku* u Njemačkoj. Nakon više od četiri desetljeća rada, 1993. godine otvaraju svoje prvo CineStar multipleks kino u Njemačkoj i započinju sa širenjem po cijeloj Europi [1].

Na početnoj stranici prikazani su filmovi koji se trenutno prikazuju u kinima. Odabirom pojedinog filma postoji kratak uvid u osnovne informacije koje su jednostavno prikazane. Stranica omogućuje pregled i pretraživanje samo onih filmova koji se trenutno prikazuju ili će se uskoro prikazivati. Također su prikazane dvorane i različite tehnologije u kojima je moguće pogledati film. Ulaznice se mogu kupovati *online* ili uživo. Slika 2.1 prikazuje naslovnu stranicu CineStar kina.

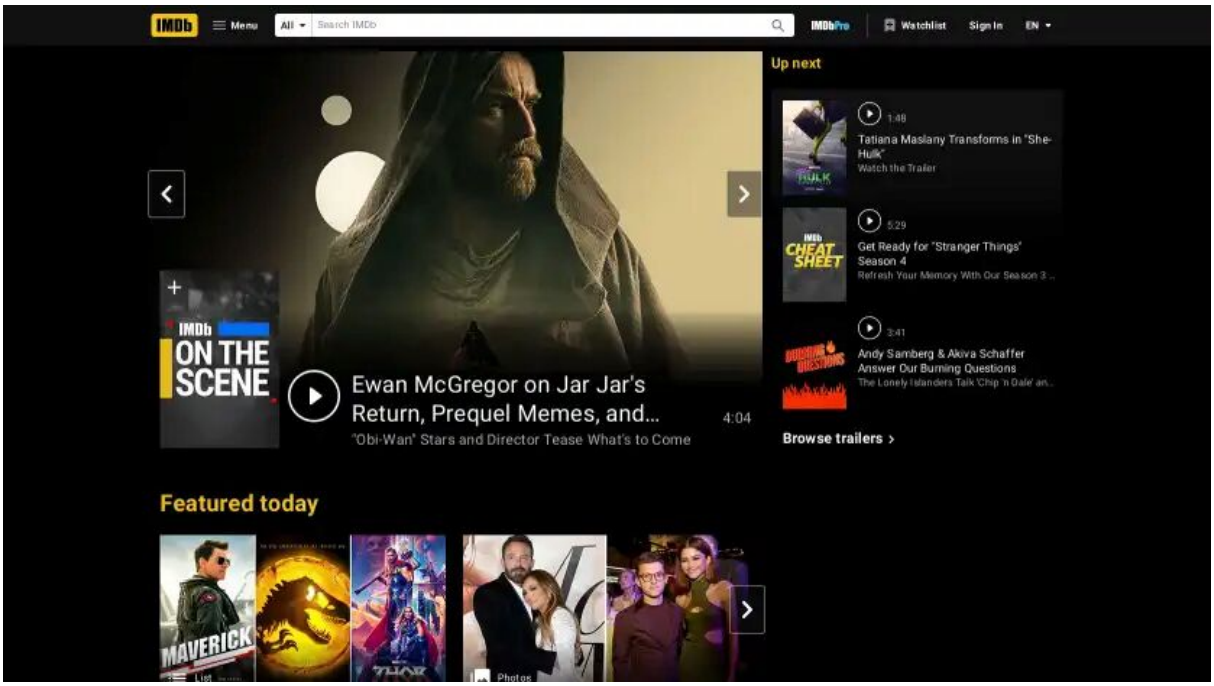


Slika 2.1. *Blitz-CineStar kino.*

2.1.2. IMDb

Pokrenut 1990. godine, IMDb je najpopularniji i najmjerodavniji izvor na svijetu za filmske i TV sadržaje te sadržaje slavnih osoba, osmišljen kako bi pomogao obožavateljima da istraže svijet filmova i emisija i odluče što će gledati. IMDb-ova baza podataka koja se može pretraživati uključuje milijune filmova, TV i zabavnih programa te glumačke ekipe i članove [2].

Na početnoj stranici prikazane su najnovije vijesti vezane uz serije i filmove te nadolazeće filmove. Odabirom filma mogu se pogledati najava filma i plakat, može se vidjeti popularnost filma i ocjene korisnika te manje poznate znamenitosti o filmu. Stranica omogućuje pretraživanje raznih filmova, a prioritet imaju oni popularniji. Moguće je pogledati najave filmova, informacije i najnovije vijesti vezane uz raznolik sadržaj kao što je prikazano na slici 2.2.

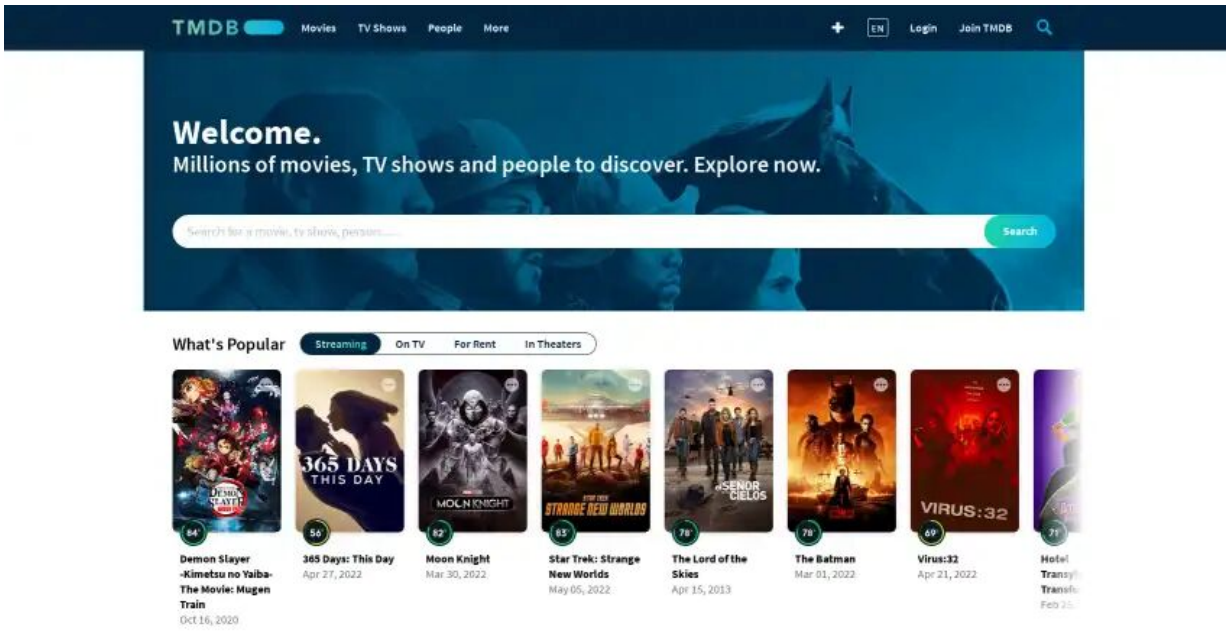


Slika 2.2. *Početna IMDb stranica.*

2.1.3. The Movie Database

The Movie Database (TMDB) je baza podataka filmova i TV serija izgrađena od strane zajednice. Zajednica je dodala svaki podatak još od 2008. godine. TMDB-ov snažan međunarodni fokus i širina podataka uvelike su neusporedivi i nešto na što su nevjerojatno ponosni [3].

Na početnoj stranici prikazani su trenutno popularni filmovi i serije na TV-u, u kinima ili na *streaming* platformama. Odabirom pojedinog filma može se vidjeti sažet i jednostavan prikaz informacija o filmu. Dodatne informacije su najave, pozadinske slike, ocjene i komentari korisnika. Također postoji mogućnost pronalaska odabranog filma na *streaming* platformama. Stranica pruža detaljan opis filmova i TV serija, više inačica filmskih plakata te prijevod na 39 jezika. Na slici 2.3 je prikazana početna stranica.



Slika 2.3. *The Movie Database naslovna stranica.*

2.2. *Web* aplikacija

Web aplikacija je računalni program koji pomoću *web* preglednika obavlja određene funkcije i radi na principu klijent-poslužitelj. To je dio okruženja u kojem više računala istodobno razmjenjuju informacije. Pojam klijent se odnosi na program koji korisnik koristi za pokretanje aplikacije. U slučaju baze podataka, klijent je program kroz koji korisnik unosi podatke, a poslužitelj je aplikacija koja pohranjuje te informacije. Za pristup *web* aplikaciji potrebni su internetska veza i *web* preglednik [4].

Postoje tri elementa koja *web* aplikacija zahtijeva za funkcioniranje: *web* poslužitelj za rukovanje zahtjevima korisnika, aplikacijski poslužitelj za izvršavanje traženih zadataka i baza podataka za pohranu informacija. Programeri kodiraju *web* aplikacije s dvije vrste programskih jezika. Za funkcioniranje aplikacije koristi se kombinacija skripti na strani poslužitelja (engl. *server-side script*) i skripti na strani klijenta (engl. *client-side script*). Skripta na strani poslužitelja bavi se pohranjivanjem i dohvaćanjem informacija i koristi programske jezike kao što su Python, Ruby, PHP, C# ili Java. Skripta na strani klijenta zahtijeva jezike i tehnologije poput HTML-a, CSS-a, JavaScripta i bavi se prezentacijom informacija korisniku. Tehnologije na klijentskoj strani oslanjaju se na preglednik za izvođenje programa [4].

Web aplikacija radi na sljedeći način:

- Korisnik stvara zahtjev prema *web* poslužitelju putem korisničkog sučelja aplikacije.
- *Web* poslužitelj taj zahtjev šalje aplikacijskom poslužitelju.
- Aplikacijski poslužitelj izvršava traženi zadatak, a zatim generira rezultate s potrebnim podacima.
- Aplikacijski poslužitelj te rezultate šalje natrag na *web* poslužitelj.
- *Web* poslužitelj prenosi tražene podatke klijentu.
- Zatražene informacije pojavljuju se na zaslonu korisnika [4].

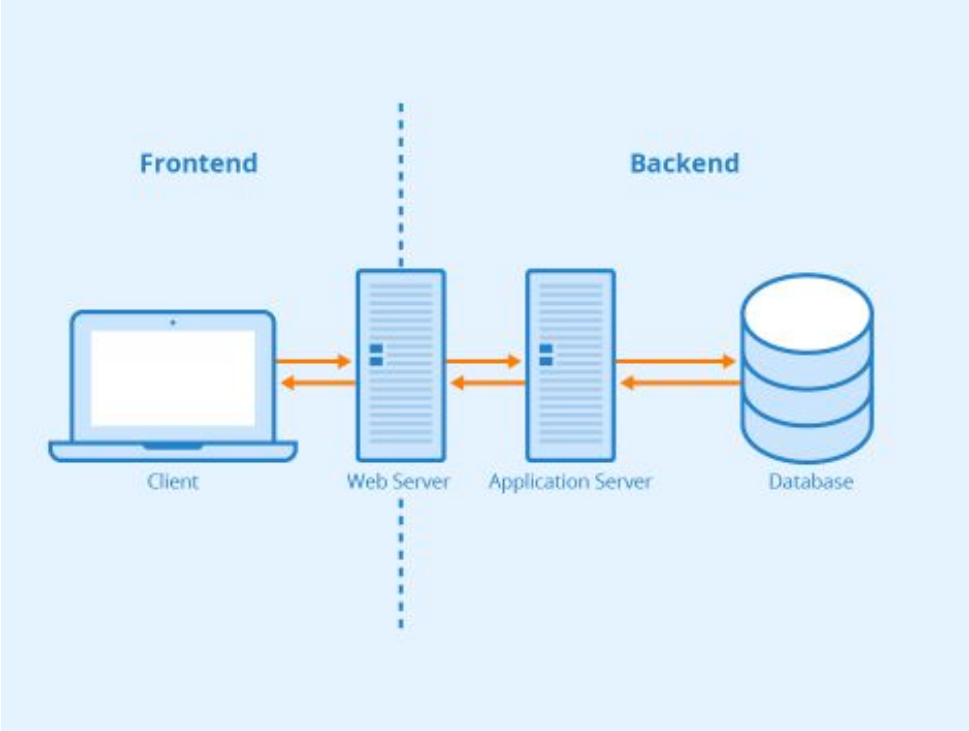
Prilikom izrade *web* aplikacije tehnologije se mogu podijeliti u dvije skupine: *frontend* (klijentske) i *backend* (poslužiteljske). Kada se govori o *frontendu* i *backendu* u većini slučajeva se to odnosi na dinamičku *web* aplikaciju. U takvoj aplikaciji korisnik preko *frontenda* komunicira s *backendom*, šalje zahtjev na obradu, prima odgovor i prikazuje povratnu informaciju korisniku tako što se pojedini elementi *web* aplikacije ažuriraju. Primjeri dinamičkih stranica su Facebook, Twitter, Google Docs i Sheets te YouTube. Za razliku od dinamičkih postoje i statičke aplikacije. One ne zahtijevaju *backend* te uvijek prikazuju isti sadržaj ili se sadržaj ne mijenja puno. Statičke stranice najčešće prikazuju životopise, odnosno portfelj (engl. *portfolio*).

Web aplikacije imaju najveću prednost u tome što ih nije potrebno instalirati ili preuzeti na korisnički uređaj, a time se uklanjaju ograničenja prostora. Zahtijevaju manje podrške i manje održavanja te niže tehničke zahtjeve od korisničkog uređaja te smanjuju troškove i za krajnjeg korisnika i za poslovanje. Ažuriranja su jednostavna za primijeniti i uvijek su ažurne. Svi se korisnici uvijek povezuju s istom inačicom tako da su problemi s kompatibilnošću eliminirani. Zbog mogućnosti pristupa s bilo kojeg mjesta putem *web* preglednika, omogućavaju rad na više platformi bez obzira na operativni sustav ili uređaj. Najvažnije, programer ima više slobode prilikom njihove izgradnje jer nije ograničen na određeni tip računala ili sustava [4].

Frontend se odnosi na to kako aplikacija izgleda, tj. ono što korisnik može vidjeti i koristiti. Glavne odgovornosti su korisničko sučelje (engl. UI - *User Interface*) i korisničko iskustvo (engl. UX - *User Experience*). Razlika između korisničkog sučelja i korisničkog iskustva je u tome što se sučelje odnosi na estetske elemente pomoću kojih korisnici komuniciraju s proizvodom, a ono uključuje sve što je korisniku na raspolaganju, od gumba, kartica i tipografije do slika, boje izbornika i formi. Korisničko iskustvo odnosi se na iskustvo koje korisnik ima s proizvodom ili

uslugom, tj. fokusira se na korisnika i njegovo korištenje i snalaženje proizvodom [5].

Backend se odnosi na to kako aplikacija radi, točnije na funkcionalnost koja korisniku nije vidljiva i koju ne može direktno koristiti. Primarna funkcionalnost je povezivanje s bazom podataka, dohvaćanje i brisanje podataka. Programer je odgovoran za izgradnju i održavanje svih komponenti s kojima obični korisnik ne komunicira, a to uključuje baze podataka, poslužitelje, logiku aplikacije i API-je (engl. *Application Programming Interface*). Programeri stvaraju ovu infrastrukturu i rade kako bi osigurali da sve ove komponente pravilno funkcioniraju [6]. Na slici 2.4 je prikazan pojednostavljeni koncept prijenosa podataka između klijentskog i poslužiteljskog dijela *web* aplikacije.



Slika 2.4. *Komunikacija frontend i backend dijela [7].*

Iako se čine odličnim za rješavanje višeplatformskih problema, *web* aplikacije imaju i određene nedostatke. Glavni nedostatak je stalna internetska veza. Pouzdana internetska veza neophodna je u svakom trenutku za pregledavanje *web* stranice i pokretanje aplikacije. Također, *web* aplikacije su znatno sporije od *native* aplikacija, tj. aplikacija izrađenih za pojedinu platformu. Zbog toga što u potpunosti rade na Internetu često se mogu činiti sporijima zbog kvalitete internetske veze. Isto tako, budući da web aplikacije nisu *native*, ponekad nisu sposobne učinkovito surađivati sa svim hardverskim i operativnim sustavima određenih uređaja na kojima se izvode [8].

Sve više je popularan trend izrade aplikacija pomoću Electron.js tehnologije. Electron omogućuje izradu Linux, Windows i macOS *desktop* aplikacija pomoću JavaScripta, HTML-a i CSS-a [9]. Iako to zvuči sjajno, Electron aplikacije nisu *native*, ne uklapaju se u oblikovanja radnog okruženja sustava i koriste Chromium preglednik kao *frontend* aplikacije, a poznato je kako Chromium preglednici nisu efikasni u korištenju radne memorije. Najveći nedostatak je što svaka Electron aplikacija dolazi sa svojom kopijom Chromium preglednika [10]. *Web* aplikacije bi se trebale izvoditi u pregledniku, a ne lažno predstavljati kao *native* aplikacije. Poznate aplikacije bazirane na Electron tehnologiji su Discord, Slack, MS Teams, Visual Studio Code i Telegram.

3. KORIŠTENE TEHNOLOGIJE

U današnje vrijeme postoji jako puno alata i biblioteka u kojima je moguće odmah započeti raditi na vlastitoj aplikaciji. Svako malo pojavi se novi *framework* koji pokušava zamijeniti ili poboljšati prethodni te je jako zbunjujuće s kojim početi. U nastavku poglavlja navedene su i objašnjene tehnologije izabrane za izradu ovog završnog rada.

3.1. Poslužiteljske tehnologije

3.1.1. .NET Core

.NET je besplatna razvojna platforma opće namjene i otvorenog koda (engl. *open-source*) koju održava Microsoft. To je višeplatformski (engl. *cross-platform*) *framework* koji se pokreće na Linux, Windows i macOS operativnim sustavima. Postoje različite inačice za svaki operativni sustav koje izvršavaju i generiraju isti kod. Služi za izgradnju različitih vrsta aplikacija, poput *web* aplikacija i *web* API-ja, mikroservisa, mobilnih i *desktop* aplikacija, igara, *Internet of Things* i strojnog učenja [11].

.NET Core dolazi u različitim implementacijama i napisan je skroz ispočetka kako bi se učinio modularnim, laganim i brzim. Uključuje osnovne značajke koje su potrebne za pokretanje aplikacije, dok su ostale značajke dostupne kao NuGet paketi koji se po potrebi mogu jednostavno dodati u aplikaciju. Tako se ubrzavaju performanse i smanjuje trag memorije, a aplikacije postaju lakše za održavanje. .NET Core podržava tri programska jezika: C#, F# i Visual Basic [11]. Za izradu *backend* dijela aplikacije odabran je C# programski jezik.

C# je moderan, inovativan, otvorenog koda, višeplatformski objektno orijentirani programski jezik kojega je razvio Microsoft. Jedan je od najboljih 5 jezika koje koriste projekti na *GitHubu* i jedan od najomiljenijih jezika prema istraživanju programera na *Stack Overflowu*. Stotine tvrtki širom svijeta koriste C# kako bi poboljšali svoje poslovanje u velikom nizu industrija, uključujući medije, financije, zdravstvenu zaštitu, igre i još mnogo toga [12].

C# je vrlo sličan Java programskom jeziku. Oba jezika su objektno-orijentirani, no C# ima par značajki koje ga razlikuju od Jave. C#, za razliku od Jave, podržava preopterećivanje operatora.

Jedna od značajki je tzv. *Language Integrated Query*. LINQ je naziv za skup tehnologija koje se temeljene na integraciji mogućnosti upita izravno u C# programski jezik. Koristi se za rad s podacima iz različitih izvora kao što su objekti, skupovi podataka, SQL i XML. Podaci se mogu dohvatiti iz bilo koje instance klase koja implementira *interface* (sučelje) IEnumerable<T> [13].

Druga od tih značajki su *properties* (svojstva). *Properties* su kombinacija atributa i metoda. Najčešće se koriste za zamjenu *gettera* i *settera* unutar klasa. Nisu ograničeni na klase, mogu zamijeniti i ostale atribute unutar struktura i *interfacesa*. Na slici 3.1 je prikazana klasa pisana s *getterima* i *setterima*.

```
public class Genre
{
        private Guid _id;
        private Instant _createdAt;
        private Instant? _modifiedAt;
        private string _name = null!;
        private ICollection<MovieGenre> _movieGenres = null!;

        public Guid GetId() { return _id; }
        public void SetId(Guid id) { _id = id; }
        public Instant GetCreatedAt() { return _createdAt; }
        public void SetCreatedAt(Instant createdAt) { _createdAt = createdAt; }
        public Instant? GetModifiedAt() { return _modifiedAt; }
        public void SetModifiedAt(Instant? modifiedAt) { _modifiedAt = modifiedAt; }
        public string GetName() { return _name; }
        public void SetName(string name) { _name = name; }
        public ICollection<MovieGenre> GetMovieGenres() { return _movieGenres; }

        public void SetMovieGenres(ICollection<MovieGenre> movieGenres)
        {
                _movieGenres = movieGenres;
        }
}
```

Slika 3.1. *Klasa koja ne koristi properties.*

Slika 3.2 prikazuje istu klasu koja koristi *properties*. Može se zaključiti kako *properties* omogućuju istu funkcionalnost s manje koda te čitljiviji programski kod.

```
public sealed class Genre : IEntity
{
        public Guid Id { get; set; }
        public Instant CreatedAt { get; set; }
        public Instant? ModifiedAt { get; set; }
        public string Name { get; set; } = null!;
        public ICollection<MovieGenre> MovieGenres { get; set; } = null!;
}
```

Slika 3.2. *C# klasa koja koristi properties.*

3.1.2. PostgreSQL

PostgreSQL je moćan sustav baze podataka koji koristi i proširuje SQL jezik u kombinaciji s mnogim značajkama koje sigurno pohranjuju i razmjenjuju podatke u najkompliciranijim opterećenjima. Podrijetlo PostgreSQL-a datira iz 1986. godine u sklopu POSTGRES projekta na Kalifornijskom sveučilištu u Berkeleyu te ima više od 30 godina aktivnog razvoja. PostgreSQL je stekao snažnu reputaciju svoje dokazane arhitekture, pouzdanosti, integriteta podataka, robusnog skupa značajki, proširivosti i posvećenosti zajednice otvorenog koda koja stoji iza softvera kako bi dosljedno isporučila izvedbena i inovativna rješenja. Radi na svim glavnim poslužiteljskim operativnim sustavima kao što su Linux, BSD i Solaris, a i onim manje popularnim poput macOS i Windows te poštuje ACID (engl. skraćenica za *atomicity, consistency, isolation, durability*) svojstva transakcija baza podataka od 2001. godine [14].

PostgreSQL dolazi s mnogim značajkama usmjerenim na pomoć programerima da izgrade aplikacije, administratorima da zaštite integritet podataka i izgrade okruženja otporna na pogreške te pomaže u upravljanju podacima bez obzira na veličinu skupa podataka. Pokušava se uskladiti sa SQL standardom gdje takva usklađenost ne proturječi tradicionalnim značajkama ili bi mogla dovesti do loših odluka u arhitekturi sustava. Od verzije 14 izdane u rujnu 2021. godine, PostgreSQL je u skladu s najmanje 170 od 179 obveznih značajki za SQL:2016 Core Conformance [14].

3.1.3. Entity Framework Core

Entity Framework Core je lagana, proširiva, otvorenog koda i višeplatformska verzija tehnologije pristupa podacima. EF Core služi kao ORM (*Object Relational Mapper*) alat koji omogućuje

.NET programerima da upravljaju bazom podataka pomoću .NET objekata i koji eliminira potrebu za pisanjem većine koda za pristup podacima. Pristup podacima vrši se pomoću modela. Model se sastoji od klasa entiteta i objekta konteksta koji predstavlja sesiju s bazom podataka. Objekt konteksta omogućuje izvršavanje upita i spremanje podataka [15].

Entitet predstavlja klasu koja opisuje tablicu u bazi podataka. Slika 3.3 prikazuje jednu takvu klasu. Atributi klase su predstavljeni stupcima u tablici, a imena su ista kao i atributi osim ako ih eksplicitno ne promijenimo.

```
public sealed class Hall : IEntity
{
        public Guid Id { get; set; }
        public Instant CreatedAt { get; set; }
        public Instant? ModifiedAt { get; set; }
        public string Name { get; set; } = null!;
        public Guid CinemaId { get; set; }
        public Cinema Cinema { get; set; } = null!;
        public ICollection<Projection> Projections { get; set; }
        public ICollection<HallSeat> HallSeats { get; set; }
}
```

Slika 3.3. *Klasa entiteta prema kojoj će se stvoriti tablica.*

Prilikom kreiranje tablice EF Core će zamijeniti tipove podataka iz klase s onima koje SQL razumije, tako će *Guid* prijeći u *uuid*, a *string* u *varchar*. U spomenutoj klasi postoje i dodatni atributi, ali oni ne predstavljaju stupce u tablici već samo olakšavaju pristup ostalim entitetima u programskom kodu. Na slici 3.4 je prikazana tablica unutar baze podataka. Tablica je modelirana prema definiranom entitetu.

```
Cinema=# select * from "Halls";
                  Id                  |          CreatedAt          |          ModifiedAt          |     Name     |              CinemaId
--------------------------------------+-----------------------------+------------------------------+--------------+--------------------------------------
 9b48e240-05f8-4a4a-b6c6-142c0cda2111 | 2022-03-25 19:16:38.147232+01 | 2022-03-25 19:18:15.447532+01 | Dvorana 2 3D | f8b42926-2863-471d-ab80-67d808f77968
 c95e3a7e-5872-4586-90da-2f51b9db9d64 | 2022-03-25 19:20:08.631347+01 | 2022-04-03 01:31:44.586674+02 | Dvorana 1    | f8b42926-2863-471d-ab80-67d808f77968
(2 rows)
```

Slika 3.4. *Kreirana tablica u bazi.*

3.1.4. Autofac

Jedno od SOLID objektno-orijentiranih načela je *Dependency Inversion Principle* (DIP). Definicija načela je sljedeća:

1. Moduli na visokoj razini ne bi trebali ovisiti o modulima na niskoj razini. Oboje bi trebali ovisiti o apstrakciji.
2. Apstrakcije ne bi trebale ovisiti o detaljima. Detalji trebaju ovisiti o apstrakcijama [16].

DIP je vrlo korisno načelo i za korištenje *dependency injectiona* (ubrizgavanje ovisnosti). *Dependency injection* omogućava stvaranje objekata o kojima klase ovise te pružanje tih objekata klasama. Koristeći *dependency injection* razdvajamo stvaranje i vezanje ovisnih objekata izvan klase, što je vrlo korisno tijekom testiranja. Autofac je jedna od mnogih biblioteka koja omogućava lakše korištenje *dependency injectiona* u programskom kodu. Jedan od preporučenih načina korištenja je preko konstruktora klase. Na slici 3.5 je vidljivo kako se u konstruktoru ovisne klase predaje apstrakcija (najčešće *interface*, ali postoje i slučajevi gdje se predaje i konkretna implementacija). Prilikom kreiranja objekta Autofac će se pobrinuti da su sve ovisnosti zadovoljene.

```
public sealed class UserCommandService : IUserCommandService
{
        private readonly UserManager<User> _userManager;
        private readonly IClock _clock;
        private readonly IEmailSender _emailSender;

        public UserCommandService(
                UserManager<User> userManager,
                IClock clock,
                IEmailSender emailSender
        )
        {
                _userManager = userManager;
                _emailSender = emailSender;
                _clock = clock;
        }
```

Slika 3.5. *Umetanje ovisnosti preko konstruktora.*

3.1.5. Automapper

Automapper je jednostavna, ali vrlo moćna biblioteka koja omogućuje kopiranje atributa jednog objekta u drugi. Najbolji primjer su entiteti tablice i objekti za prijenos podataka (engl. *Data Transfer Object*, skraćeno DTO). Oboje imaju velik broj sličnih atributa, no DTO ih može imati više, a isto tako neke ne mora uopće sadržavati. Prilikom kreiranja entiteta u bazi korisniku se ne želi omogućiti unos primarnog ključa i datuma stvaranja. Kada korisnik ispuni formu i pošalje zahtjev, Automapper će stvoriti instancu entiteta kopirajući atribute iz objekta za prijenos podataka. Slika 3.6 prikazuje konfiguraciju za kopiranje atributa dva objekta. Atributi s istim imenom i tipom podatka će se automatski kopirati dok se za ostale atribute mora navesti kako će se kopirati ili hoće li biti ignorirani.

```
public FilmRatingMapper()
{
        CreateMap<FilmRating, FilmRatingDto>();

        CreateMap<CreateFilmRatingDto, FilmRating>()
                .ForMember(filmRating => filmRating.Id, options
                        => options.Ignore())
                .ForMember(filmRating => filmRating.CreatedAt, options
                        => options.Ignore())
                .ForMember(filmRating => filmRating.ModifiedAt, options
                        => options.Ignore())
                .ForMember(filmRating => filmRating.Movies, options
                        => options.Ignore());
}
```

Slika 3.6. *Kopiranje atributa uz ignoriranje nepostojećih.*

3.1.6. Noda Time

U C# postoji samo *DateTime* tip podatka za datum i vrijeme. Nedostatak je što se u bazu podataka moraju spremiti i datum i vrijeme. Za primjer je moguće uzeti datum rođenja jer vrijeme nije potrebno spremati. Sa standardnim *DateTime* tipom podatka moraju se spremiti, uz potreban datum, nepotrebne informacije o vremenu. Međutim, nije moguće samo spremiti točan datum s nasumičnim vremenom jer se prilikom prikaza mogu dogoditi pogreške vezane s vremenskom zonom korisnika. Noda Time je alternativna biblioteka za datum i vrijeme u .NET okruženju.

Pruža mnoštvo novih tipova podataka za rukovanje datumom i vremenom, od kojih su *Instant*, *Offset*, *LocalDateTime*, *LocalDate* i *LocalTime* najvažniji [17].

Instant je točka na zamišljenoj globalnoj vremenskoj liniji, bez obzira na kalendarski sustav i vremensku zonu. Predstavlja broj nanosekundi koji je protekao od UNIX epohe koja odgovara ponoći 1. siječnja 1970. u UTC-u. *Offset* se koristi za izražavanje razlike između UTC vremena i lokalnog vremena. Dodaje se UTC vrijednosti kako bi se dobilo lokalno vrijeme, a oduzima se od lokalnog vremena kako bi se dobila UTC vrijednost. *LocalDateTime* je točka na vremenskoj crti u određenom kalendarskom sustavu, ali bez koncepta pomaka od UTC-a. *LocalTime* je samo vremenski dio *LocalDateTime*, a *LocalDate* je samo dio datuma [17].

3.2. Klijentske tehnologije

3.2.1. TypeScript

JavaScript, jedan od najčešće korištenih programskih jezika na svijetu, postao je službeni jezik Interneta. Programeri ga koriste za pisanje višeplatformskih aplikacija koje se mogu pokretati na bilo kojoj platformi i u bilo kojem pregledniku. Iako se JavaScript koristi za stvaranje višeplatformskih aplikacija, nije zamišljen za velike aplikacije koje uključuju tisuće ili čak milijune linija koda. JavaScriptu nedostaje nekih značajki zrelijih jezika koje napajaju današnje kompleksne aplikacije. TypeScript se odnosi na ograničenja JavaScripta, radeći to bez ugrožavanja njegove ključne vrijednosti: mogućnost pokretanja koda bilo gdje i na svakoj platformi, pregledniku ili poslužitelju. TypeScript je jezik otvorenog koda kojega je razvio Microsoft [18].

JavaScript je *loosely typed* programski jezik što znači da može biti vrlo teško razumjeti koji tipovi podataka se koriste za varijable i parametre funkcija. TypeScript omogućuje određivanje tipova podataka koji se koriste unutar koda. Ima mogućnost prijavljivanja pogrešaka kada se tipovi podataka ne podudaraju. Pozivanje funkcije s parametrom koji očekuje niz znakova, a predan joj je parametar s tipom podatka broj će prouzročiti pogrešku prilikom prevođenja (engl. *compiling*). U JavaScriptu pogreška će se dogoditi tijekom izvođenja aplikacije.

TypeScript je *superset* JavaScripta, dijeli istu sintaksu, ali i dodaje par stvari. Kako preglednici razumiju samo JavaScript, prije izvođenja potrebno je prevesti TypeScript kod u JavaScript kod.

Ako se TypeScript koristi unutar *frameworka*, poput React.js, prevođenje će se provesti automatski. Na slici 3.7 je prikazana generička klasa napisana u TypeScriptu.

```
class ApiStoreBase<T, U extends QueryParameters> implements IApiStoreBase<T, U> {
    protected readonly _endpoint: string;
    public _errors: ApiErrors;
    public _loading: ApiLoading;
    public _queryParameters: U;
    public _response: T | undefined;
    public _responses: T[] | undefined;
    public _records: number;
    public _status: ApiStatus;

    public constructor(endpoint: string) {
        this._endpoint = endpoint;
        this._records = 0;
        this._queryParameters = { page: 1, size: 10, orderBy: 1 } as U;
        this._status = { get: 0, getAll: 0, post: 0, put: 0, delete: 0 };
        this._errors = { get: [], getAll: [], post: [], put: [], delete: [] };
        this._loading = { get: false, getAll: true, post: false, put: false, delete: false };
```

Slika 3.7. *Isječak koda TypeScript klase.*

Slika 3.8 prikazuje istu klasu prevedenu u JavaScript. Nakon prevođenja uočljivo je nekoliko razlika. Prva vidljiva razlika je u tome kako su tipovi podataka i modifikatori pristupa jednostavno izbačeni. Oni u JavaScriptu ne postoje, ali zato uvelike pomažu prilikom pisanja koda u TypeScriptu. Druga razlika je u tome što je klasa pretvorena u funkciju. Prilikom prevođenja, TypeScript kompilatoru (engl. *compiler*) se može reći u koju JavaScript verziju da prevede kod. Ako JavaScript kod mora raditi na što većem broju preglednika mora se ciljati starija verzija. Klase u JavaScriptu su uvedene verzijom ECMAScript 2015. Kod se može pisati u novijoj verziji, a kompilator će ga prevesti na stariju verziju bez gubitka funkcionalnosti. Identičnu funkciju izvršava i Babel.js kompilator.

```
var ApiStoreBase = /** @class */ (function () {
    function ApiStoreBase(endpoint) {
        var _this = this;
        this._endpoint = endpoint;
        this._records = 0;
        this._queryParameters = { page: 1, size: 10, orderBy: 1 };
        this._status = { get: 0, getAll: 0, post: 0, put: 0, delete: 0 };
        this._errors = { get: [], getAll: [], post: [], put: [], delete: [] };
        this._loading = { get: false, getAll: true, post: false, put: false, delete: false };
```

Slika 3.8. *Isječak koda prevedene JavaScript klase.*

TypeScript ima dodatne značajke koje ne postoje u JavaScriptu, a to su:

- *interfaces*
- *namespaces*
- *generics*
- apstraktne klase
- dekorateri klasa
- modifikatori pristupa
- preopterećivanje funkcija
- *readonly* ključnu riječ [18]

3.2.2. React.js

Za izradu *frontend* dijela aplikacije izabran je React.js zajedno s TypeScriptom. React je biblioteka za razvoj sučelja bazirana na JavaScriptu. Temelji se na stvaranju manjih komponenti korisničkog sučelja koje, svaka zasebno, brinu o svojim stanjima. Komponente prikazuju podatke koji se mijenjaju s vremenom. Takve komponente se mogu iskoristiti na više različitih mjesta, a zajedno se mogu slagati kako bi stvorili veće i složenije komponente. Umjesto da izravno manipulira DOM-om (engl. *Document Object Model*) preglednika React stvara virtualni DOM u memoriji gdje čini sve potrebne manipulacije prije nego što te promjene primjeni u stvarni DOM. Ukratko, uspoređuje prethodna stanja komponenti i ažurira samo one stavke koje su promijenjene [19].

React komponente koriste specifičnu sintaksu zvanu JSX. JSX je kombinacija HTML i JavaScript izraza. Na slici 3.9 prikazana je JSX komponenta. Iako prikazana komponenta izgleda jednostavno, ona se sastoji od dodatne tri komponente koje se također sastoje od više manjih komponenata. Važno je napomenuti kako komponente, kao i funkcije, mogu vratiti samo jednu vrijednost. Kako bi zaobišli to ograničenje vrijednosti koje vraćamo možemo omotati unutar React fragmenata. To činimo tako da vrijednosti stavimo unutar znakova <> i </>. Fragmenti nisu ispravni HTML elementi te se ne prikazuju u DOM-u.

```
const Movies = () => {
    useEffect( effect: () => {
        document.title = "Movies | Cinema";
    }, deps: []);

    return (
        <Grid>
            <Grid.Col md={3} xs={12}>
                <MovieFilter />
            </Grid.Col>
            <Grid.Col md={9} xs={12}>
                <MovieAlbum />
            </Grid.Col>
        </Grid>
    );
};

export default memo(Movies);
```

Slika 3.9. *Funkcijska React komponenta.*

3.2.3. Axios

Axios je HTTP klijent temeljen na JavaScript obećanjima (engl. *promises*) za Node.js i preglednik. Izomorfan je, što znači da se može pokrenuti i u pregledniku i u Node.js poslužitelju s istim kodom. Na strani poslužitelja koristi izvorni Node.js HTTP modul, dok na klijentu koristi XMLHttpRequest [20]. Axios je alternativa za ugrađeni JavaScript *Fetch* API. Jednostavniji je za korištenje jer eliminira ručno pisanje dodatnog koda za upravljanje greškama i ubacivanjem parametara u zaglavlje zahtjeva. Na primjer, ako se pošalje zahtjev *backendu* i dogodi se pogreška sa statusnim kodom 400 ili 500, Axios će automatski znati da treba baciti iznimku (engl. *exception*). Kod *Fetcha* bi se dodatno morao provjeravati HTTP statusni kod. Također, automatski vrši *parsiranje* (raščlanjivanje) JSON objekata nakon uspješnog izvršenja zahtjeva.

Dodatne značajke su:

- XMLHttpRequest zahtjevi iz preglednika
- Podržava JavaScript *promise* API
- Presretanje zahtjeva i odgovora
- Otkazivanje zahtjeva
- Klijentska podrška protiv XSRF napada [20]

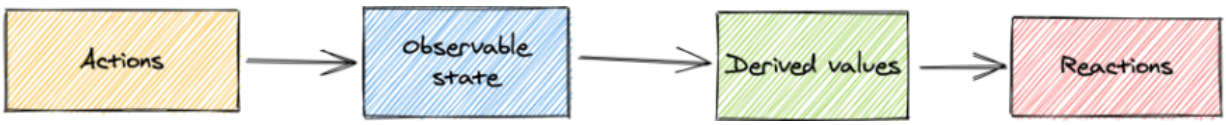
3.2.4. MobX

MobX je biblioteka koja upravljanje stanjem čini jednostavnim i skalabilnim. MobX razlikuje sljedeća tri koncepta u aplikaciji:

1. stanja
2. radnje
3. derivacije [21]

MobX smatra da stanje treba definirati i učiniti ga promatranim (engl. *observable*). Stanja su podaci koji pokreću aplikaciju te mogu biti bilo kojeg tipa, poput običnih objekata, nizova, klasa, cikličkih struktura podataka ili referenci. Radnja je bilo koji dio koda koji mijenja stanje kao što su korisnički događaji i odgovori *backenda*. Korištenje radnji pomaže u strukturiranju koda i sprječava da se stanje promijeni kada to nije potrebno. Metode koje mijenjaju stanje nazivaju se radnjama. Nadalje, sve što se može izvesti iz stanja bez daljnje interakcije je derivacija. MobX razlikuje dvije vrste derivacija, a to su izračunate vrijednosti i reakcije. Izračunate vrijednosti se uvijek mogu izvesti iz trenutno promatranih stanja pomoću čiste (engl. *pure*) funkcije, a reakcije su nuspojave koje se moraju dogoditi automatski kada se stanje promijeni [21].

MobX koristi jednosmjerni protok podataka u kojem radnje mijenjaju stanje, što zauzvrat ažurira sve zahvaćene poglede. Primjer tog protoka dan je slikom 3.10. Sve derivacije se ažuriraju automatski i atomski kada se stanje promijeni. Kao rezultat toga, nikada nije moguće promatrati međuvrijednosti. Sve derivacije se ažuriraju sinkrono. To znači da radnje mogu sigurno provjeriti izračunatu vrijednost neposredno nakon promjene stanja. Izračunate vrijednosti se lijeno ažuriraju, a to znači da bilo koja izračunata vrijednost koja nije u aktivnoj upotrebi se neće ažurirati sve dok nije potrebna za nuspojavu. Sve izračunate vrijednosti moraju biti čiste, tj. ne smiju mijenjati stanje [21].



Slika 3.10. *MobX protok podataka [21].*

3.2.5. React Router DOM

Kako se React fokusira na izgradnju korisničkih sučelja, nema ugrađeno rješenje za usmjeravanje. React Router DOM je biblioteka na strani klijenta i poslužitelja koja omogućuje implementaciju dinamičkog usmjeravanja unutar *web* aplikacije. Omogućuje prikaz različitih podstranica i dozvoljava korisnicima kretanje po njima. Te stranice se ne osvježavaju nego se dinamički dohvaćaju na temelju URL-a. Koristi se za izradu *Single-Page* aplikacija (SPA), odnosno *web* aplikacija koje dinamički osvježavaju sadržaj umjesto ponovnog učitavanje stranice [22].

3.2.6. Mantine UI

Mantine UI je biblioteka otvorenog koda bazirana na TypeScriptu koja sadrži preko 120 prilagođenih React komponenti za bržu izgradnju sučelja. Sve komponente su responzivne te podržavaju svijetlu i tamnu temu [23]. Mantine UI odlikuju odlična dokumentacija i mnoštvo različitih primjera. Najčešće korištene komponente su *Skeleton*, *Grid*, *Stack*, *Select*, *Modal* i *Spoiler*.

Grid omogućuje slaganje komponenti i elemenata u stupce, dok *Stack* vertikalno slaže elemente. Ovisno o veličini zaslona, može se odrediti broj prikazanih elemenata u *Gridu*. *Select* komponenta omogućuje korisniku pretraživanje i odabir jedne ili više ponuđenih opcija. *Modal* komponenta se prikazuje ispred svih ostalih i služi za prikazivanje određene informacije. *Skeleton* komponenta prikazuje stanje učitavanja sadržaja te se može koristiti kao *placeholder* dok se ostali elementi, najčešće slike, učitavaju. *Spoiler* omogućava sakrivanje dugih dijelova sadržaja, najčešće teksta, koji se mogu proširiti ili smanjiti.

4. FUNKCIONALNOST *WEB* APLIKACIJE

Kako bi se mogla razviti i implementirati *web* aplikacija se mora razdvojiti na pojedine dijelove. Tako se aplikacija može brže i jednostavnije razviti jer su slojevi neovisni, a kada dođe novi poslovni zahtjev promjene su minimalne. Raspodjela uvelike ovisi o odabiru arhitekture. U ovom poglavlju opisana je odabrana arhitektura aplikacije, njezini slojevi i programska implementacija tih slojeva te načini i razlozi zašto je funkcionalnost implementirana na određen način.

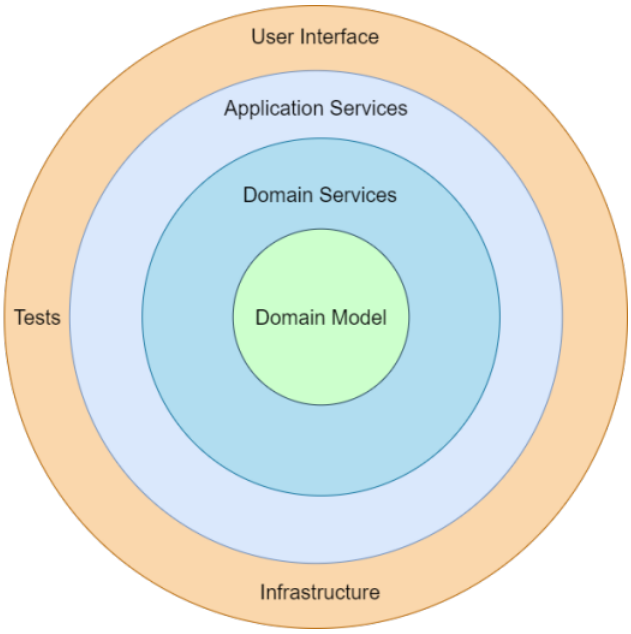
4.1. Arhitektura

Arhitektura aplikacije opisuje obrasce i tehnike koje se koriste za dizajniranje i izgradnju aplikacije. Daje savjete i najbolje prakse koje treba slijediti prilikom izgradnje aplikacije kako bi rezultat bio dobro strukturirana aplikacija. Vrste arhitektura koje postoje su:

- Slojevita ili N-slojna arhitektura
- Monolitna arhitektura
- Arhitektura mikroservisa
- Arhitektura vođena događajima (engl. *event-driven*)
- Servisno orijentirana arhitektura [24]

Za izradu završnog rada odabrana je izmijenjena vrsta slojevite arhitekture zvana *onion* arhitektura (arhitektura luka). Arhitekturu je prvi predstavio Jeffrey Palermo kako bi ponudio bolji način za stvaranje rješenja za bolje održavanje, testiranje i pouzdanost [25].

Glavni cilj je kontrola spajanja (engl. *coupling*). Temeljno pravilo je da kod u određenom sloju može ovisiti o nižim slojevima, ali ne može ovisiti o slojevima koji se nalaze iznad njega. Model domene je središte arhitekture i sadrži sve entitete domene. Ti entiteti nemaju nikakvu vrstu ovisnosti. Sloj usluga domene odgovoran je za definiranje potrebnih sučelja za pohranu i dohvaćanje objekata. Vanjski sloj luka namijenjen je komponentama koje se često mijenjaju. Ovdje se nalaze korisničko sučelje sa svim svojim pogledima, kontrolerima i svim stvarima vezanim uz logiku prezentacije [25]. Slika 4.1 prikazuje pojednostavljeni prikaz arhitekture luka.



Slika 4.1. *Slojevi arhitekture luka [25].*

Arhitektura se uvelike oslanja na DIP načelo. Poslužiteljski dio aplikacije, s kontrolerima i testovima, predstavlja središnji, odnosno unutarnji dio arhitekture luka. Klijentski dio isključivo predstavlja korisničko sučelje vanjskog sloja.

4.2. Sloj domene

Za početak se stvara novi *solution* s projektima *Domain*, *Repository*, *Service* i *API*. Ti projekti će služiti kao slojevi arhitekture. Svaki sloj ima dodatan projekt koji sadrži testove specifične za taj sloj. Također, *Repository* i *Service* slojevi imaju dodatne projekte, a to su *Repository.Contracts* i *Service.Contracts*. *Repository* i *Service* sadrže konkretne implementacije, dok se *Repository.Contracts* i *Service.Contracts* projekti sastoje isključivo od *interfacea*. S implementacijom se najčešće kreće od sloja domene.

Domena sadrži entitete, objekt konteksta koji predstavlja konekciju s bazom, konfiguracije entiteta i migracije. Prvo se definiraju entiteti. Entiteti predstavljaju tablice u bazi, a također sadržavaju i dodatne atribute koji pomažu prilikom korištenja EF Core ORM alata. Slika 4.2 prikazuje zapečaćenu (engl. *sealed*) klasu entiteta za film. U C# programskom jeziku ključna riječ *sealed* označava klasu koja se ne može naslijediti. Entiteti su središte sloja domene, oko njih se razvijaju ostali slojevi pa ih nema smisla nasljeđivati.

```
public sealed class Movie : IEntity
{
        public string Id { get; set; } = null!;
        public Instant CreatedAt { get; set; }
        public Instant? ModifiedAt { get; set; }
        public string Title { get; set; } = null!;
        public uint Year { get; set; }
        public uint Duration { get; set; }
        public string Tagline { get; set; } = null!;
        public string Summary { get; set; } = null!;
        public bool InCinemas { get; set; }
        public bool IsFeatured { get; set; }
        public bool IsRecommended { get; set; }
        public string PosterUrl { get; set; } = null!;
        public string PosterId { get; set; } = null!;
        public string BackdropUrl { get; set; } = null!;
        public string BackdropId { get; set; } = null!;
        public string TrailerUrl { get; set; } = null!;
        public Guid FilmRatingId { get; set; }
        public FilmRating FilmRating { get; set; } = null!;
        public ICollection<MovieGenre> MovieGenres { get; set; } = null!;
        public ICollection<MoviePerson> MoviePeople { get; set; } = null!;
        public ICollection<UserRating> Ratings { get; set; } = null!;
        public ICollection<Projection> Projections { get; set; } = null!;
}
```

Slika 4.2. *Entitet film.*

Prije korištenja mogućnosti koje pruža EF Core potrebno je definirati objekt konteksta. Objekt konteksta predstavlja sesiju s bazom podataka i omogućava izvršavanje naredbi i upita nad instancama entiteta. U kontekstu se definiraju veze između entiteta, njihova ograničenja koja će se primijeniti na bazu i reference na entitete u bazi. Na slici 4.3 je prikazana konfiguracija žanr tablice. Prilikom konfiguracije entiteta, ako se eksplicitno ne postavi veličina određenih tipova podataka, EF Core će postaviti duljinu na najveću koju baza podržava. Također, klasa je definirana s ključnom riječi *internal*. Kako se spomenuta klasa koristi samo u sloju domene za konfiguraciju konteksta nema potrebe da bude vidljiva u ostalim slojevima.

```
internal sealed class GenreMap : IEntityTypeConfiguration<Genre>
{
        public void Configure(EntityTypeBuilder<Genre> builder)
        {
                builder.Property(genre => genre.Name).HasMaxLength(32);
                builder.HasIndex(genre => genre.Name).IsUnique();
        }
}
```

Slika 4.3. *Ograničenja u tablici.*

Nakon konfiguriranja konteksta potrebno je napraviti migracije. Migracije predstavljaju promjene u shemi baze podataka. Kada se promijeni bilo koji atribut, tip podatka ili ime entiteta potrebno je napraviti migraciju kako bi se te promijene primijenile i u bazi. EF Core vodi brigu o kreiranju i primijeni migracija. Može generirati migracije, ali one neće biti automatski primijenjene.

4.3. *Repository* sloj

Repository sloj koristi se za izvršavanje upita nad bazom podataka. Ti upiti su osnovne operacije, a one uključuju dohvaćanje, sortiranje, filtriranje, ažuriranje i brisanje podataka. *Repository* sloj ovisi o sloju domene jer se sve operacije obavljaju kroz objekt konteksta. Za lakšu manipulaciju podacima koriste se *Unit of Work* i *Specification* oblikovni obrasci.

Specification oblikovni obrazac omogućuje enkapsulaciju pojedinog znanja o domeni u specifikaciju. Specifikacija opisuje logiku upita, sortiranja i paginacije (engl. *pagination*) jer su svi upiti pohranjeni u nju. Slika 4.4 prikazuje *interface* za specifikaciju. Sve metode (osim paginacije) kao parametar primaju LINQ izraze. Nakon primjene specifikacije EF Core će LINQ izraze pretvoriti u efikasne SQL upite. Za primjenu specifikacije koristi se *Specification Evaluator* klasa. *Specification Evaluator* primjenjuje zadane upite iz specifikacije na objekt konteksta entiteta.

```
public interface ISpecification<TEntity> where TEntity : class
{
        Expression<Func<TEntity, bool>>? Filter { get; }
        Func<IQueryable<TEntity>, IIncludableQueryable<TEntity, object>>? Include { get; }
        Expression<Func<TEntity, object>>? OrderBy { get; }
        Expression<Func<TEntity, object>>? OrderByDescending { get; }

        int Skip { get; }
        int Take { get; }
        bool PaginationEnabled { get; }
        bool Tracking { get; }

        void AddFilter(Expression<Func<TEntity, bool>> filterExpression);
        void AddInclude(
                Func<IQueryable<TEntity>, IIncludableQueryable<TEntity, object>>?
                        includeExpression
        );
        void AddOrderBy(Expression<Func<TEntity, object>> orderByExpression);
        void AddOrderByDescending(
                Expression<Func<TEntity, object>> orderByDescendingExpression
        );
        void AddPagination(int pageNumber, int pageSize);
        void WithTracking(bool withTracking);
}
```

Slika 4.4. *Generička specifikacija.*

Na slici 4.5 je prikazana *GetAsync* metoda generičke *repository* klase. U metodi se na objekt konteksta entiteta primjenjuje specifikacija i vraća dobiveni rezultat izvršenog upita. *Repository* klasa je također definirana s ključnom riječi *internal* jer se sve *repository* klase nalaze unutar *Unit of Work* obrasca. One se ne koriste direktno nego se *Unit of Work* preko *dependency injectiona* ubrizgava u klase servisa.

```
public virtual Task<TEntity?> GetAsync(ISpecification<TEntity> specification)
{
        IQueryable<TEntity> query = DbSet;
        query = SpecificationEvaluator<TEntity>.GetQuery(query, specification);
        return query.FirstOrDefaultAsync();
}
```

Slika 4.5. *Primjena specifikacije prije dohvaćanja podataka.*

4.4. Sloj servisa

Sloj servisa služi za primjenu poslovnih pravila i predstavlja apstrakciju pristupa podacima. U ovom sloju definirani su filtri i servisi za svaki entitet te moguće iznimke (engl. *exceptions*). Filtri sadrže atribute kao što su broj stranice, broj podataka po stranici, vrijednost sortiranja i ključna riječ za pretraživanje. Ovisno o entitetu tih atributa može biti i više. Servisi su podijeljeni u dvije kategorije: *QueryService* i *CommandService*. *Query* servisi se brinu isključivo o načinu dohvaćanja podataka iz baze, dok se *Command* servisi brinu isključivo o modifikaciji tih podataka.

Slika 4.6 prikazuje par metoda iz generičke apstraktne klase *QueryServiceBase*. Većina metoda se može primijeniti na velik broj entiteta bez promjene njihove implementacije. No zbog određenih poslovnih pravila metode se mogu razlikovati u nekoliko linija koda te su zbog toga označene s ključnom riječi *virtual*. Virtualne metode imaju implementaciju i mogu se nadjačati (engl. *override*) ako poslovna pravila zahtijevaju određenu logiku. Najbolji primjer za poslovnu logiku su projekcije. Film ne može imati projekciju u istoj dvorani unutar vremena trajanja druge projekcije. U tom slučaju potrebno je nadjačati metodu, obaviti potrebnu provjeru i ponovno pozvati metodu naslijeđene klase pomoću ključne riječi *base*.

```
public abstract Task<ICollection<TEntity>> GetAllAsync(TFilter entityFilter);
public abstract Task<ICollection<TDto>> GetAllAsync<TDto>(TFilter entityFilter)
        where TDto : class, new();

public virtual async Task<TEntity> GetAsync(
        Expression<Func<TEntity, bool>> expression,
        bool withTracking = false
)
{
        var entity = await TryGetAsync(expression, withTracking);
        if (entity is null) throw new EntityNotFoundException();
        return entity;
}
```

Slika 4.6. *Virtualne i apstraktne metode servisne klase.*

Apstraktne metode ne smiju imati implementaciju i moraju se nadjačati. Metoda *GetAllAsync* je označena s ključnom riječi *abstract*. Iako se mogla definirati kao virtualna metoda s jednostavnom

implementacijom, postoje dva problema s takvim rješenjem. Prvi problem je u tome što se ne može znati koje sve atribute filter može sadržavati. Naravno, postoje četiri osnovna atributa, ali kako je klasa generička nije moguće uzeti u obzir i dodatne atribute.

Drugi problem je vezan uz način kako EF Core funkcionira. Za primjer se može uzeti film. Film ima relaciju sa žanrovima. I film i žanr su u bazi spremljeni u dvije različite tablice. Ako se EF Core alatu eksplicitno ne kaže da film uključuje žanrove, prilikom dohvaćanja filmova, žanrovi će imati *null* vrijednosti. Problem se može riješiti tako što se u specifikaciji navede koje dodatne tablice se žele uključiti. Metoda *GetAllAsync* bi se uvijek morala nadjačati, bez obzira imala ona implementaciju ili ne. Zbog toga je bolje ostaviti ju apstraktnom.

Iako metoda *GetAsync* sadrži implementaciju za dohvaćanje jednog podatka iz baze ona također ne zna koje dodatne tablice treba uključiti. Razlog zašto metoda *GetAllAsync* nema implementaciju je u tome što ima jedan problem više koji ovisi o entitetu. Implementacija metode *GetAsync* izbjegava pisanje istog koda za entitete koji ne uključuju dodatne tablice. Isto tako, *GetAsync* metoda ne ovisi o specifičnom filtru.

Sloj servisa nije ograničen samo za apstrakciju nad *repository* slojem. On može sadržavati dodatne klase koje rješavaju određenu poslovnu logiku ili komuniciraju s vanjskim servisima. Primjeri takvih servisa su servis za generiranje *JWT* tokena, slanja e-pošte i prijenos slika. Slika 4.7 prikazuje metodu jednog takvog servisa. Isječak servisa predstavlja oblikovni obrazac fasadu, a prikazana metoda služi kao omotač oko vanjskog servisa za prijenos slika te uvelike olakšava njegovo korištenje.

```
public async Task<Result> UploadImageAsync(IImageRequest request)
{
        var imageHeader = request.Image[..10];
        var image =
                imageHeader.Contains("https://") || imageHeader.Contains("http://")
                ? request.Image : MagickConvert(request);

        return await _imagekit.UploadAsync(new FileCreateRequest
        {
                file = image,
                fileName = request.Name,
                folder = request.Path,
                useUniqueFileName = true
        });
}
```

Slika 4.7. *Metoda fasade za prijenos slika.*

4.5. API sloj

API sloj pruža odvojeno sučelje za podatke aplikacije te omogućuje interakciju s aplikacijom neovisno o programskom jeziku. Dvije najčešće korištene vrste API-ja su REST (engl. *Representational State Transfer*) i SOAP. REST API-ji su dizajnirani da iskoriste prednosti postojećih protokola, a u slučaju Web API-ja to je HTTP protokol. Za razliku od SOAP-a, REST nije ograničen samo na XML, već umjesto toga može vratiti XML, JSON, YAML ili bilo koji drugi format, ovisno o tome što klijent zahtijeva [26].

REST definira 6 arhitekturalnih ograničenja koje web usluge moraju poštovati:

1. klijent-poslužitelj - klijent i poslužitelj trebaju biti odvojeni jedan od drugog i dopušteno im je da se razvijaju pojedinačno
2. bez stanja (engl. *stateless*) - pozivi se mogu obavljati neovisno jedan o drugom, a svaki poziv sadrži sve podatke potrebne da se uspješno završi
3. predmemoriranje (engl. *cacheing*) - treba biti osmišljen tako da potiče pohranu podataka koji se mogu predmemorirati
4. ujednačeno sučelje (engl. *uniform interface*) - omogućuje razvoj aplikacije bez potrebe za čvrstom povezanošću sa samim API slojem
5. slojeviti sustav - različiti slojevi arhitekture rade zajedno na izgradnji hijerarhije koja pomaže u stvaranju skalabilnije aplikacije
6. kod na zahtjev (nije obavezno) - omogućuje prijenos koda putem API-ja za korištenje unutar aplikacije [26]

Tip API-ja u ovom završnom radu je REST API, a radi jednostavnosti implementacije tip podatka koji vraća je JSON.

Najvažniji dio svakog REST API-ja je kontroler. Kontroler obrađuje dolazne HTTP zahtjeve i šalje odgovor pozivatelju. Odgovor se sastoji od tri stvari: zaglavlja i statusnog koda koji su obavezni te tijela odgovora koje, ovisno o zahtjevu, može biti neobavezno. Kontroler nije ništa drugo nego obična klasa, mora biti javna, mora naslijediti apstraktnu klasu *ControllerBase* i mora završavati s riječi *"Controller"*. Sve javne metode kontrolera nazivaju se akcijske metode. Prema slici 4.8, na kojoj je isječak kontroler klase, odmah su uočljive dvije stvari: kojoj ruti kontroler pripada te koji mu je povratni tip. Svaka akcijska metoda unutar kontrolera može nadjačati te parametre.

```
[Route("api/movies")]
[Produces(MediaTypeNames.Application.Json)]
public sealed class MovieController : ControllerBase
{
        private readonly IMovieCommandService _commandService;
        private readonly IMovieQueryService _queryService;
        private readonly IMapper _mapper;
        private readonly IImageFacade _image;

        public MovieController(
                IMovieQueryService queryService,
                IMovieCommandService commandService,
                IMapper mapper,
                IImageFacade image
        )
        {
                _queryService = queryService;
                _commandService = commandService;
                _mapper = mapper;
                _image = image;
        }
```

Slika 4.8. *Klasa kontrolera.*

Na slici 4.9 prikazana je *GetAsync* akcijska metoda. Ovisno o tome kako je ruta u metodi napisana, ona može proširiti postojeću rutu kontrolera, kao što je prikazano na slici, ili ako počinje sa znakom '/', onda će ju nadjačati. Atribut *HttpGet* označava da će se metoda izvršiti samo na *GET* zahtjeve. *GetAsync* metoda nema tijela zahtijeva, tj. ne prima nikakve podatke nego samo vraća odgovor. Također, atribut *AllowAnonymous* omogućuje svima da pošalju zahtjev i dobiju odgovor.

```
[HttpGet]
[Route("{id}")]
[AllowAnonymous]
public async Task<IActionResult> GetAsync(string id)
{
        var movie = await _queryService.GetAsync<IndividualMovieDto>(movie => movie.Id == id);
        return StatusCode(StatusCodes.Status200OK, movie);
}
```

Slika 4.9. *GET Akcijska metoda.*

Zatim imamo *CreateAsync* akcijsku metodu prikazanu slikom 4.10. Ona odgovara samo na *POST* zahtjeve te mora sadržavati tijelo zahtjeva koje je u JSON obliku. Također, samo registrirani korisnici koji imaju "Admin" ovlasti mogu pozvati metodu.

```
[HttpPost]
[Authorize(Roles = nameof(Roles.Admin))]
[Consumes(MediaTypeNames.Application.Json)]
public async Task<IActionResult> AddAsync([FromBody] CreateGenreDto createGenreDto)
{
        var genre = _mapper.Map<Genre>(createGenreDto);
        await _commandService.AddAsync(genre);
        return StatusCode(StatusCodes.Status201Created);
}
```

Slika 4.10. *POST akcijska metoda.*

Može se uočiti kako se u tijelu metode nigdje ne provjeravaju ovlasti korisnika, niti se vraća drugačiji status kod. Kako kontroler zna da korisnik ima pristup? Što ako korisnik nije registriran? Što ako nema "Admin" ovlasti? Hoće li mu kontroler uvijek vratiti status 201? Zbog toga što kontroler nasljeđuje apstraktnu klasu *ControllerBase* .NET Core će se sam pobrinuti o tim stvarima. Npr. ako korisnik nije registriran kontroler će mu vratiti status 401, a ako nema "Admin" ovlasti status 403. Isto tako, ako validacije tijela zahtjeva na prođu, .NET Core će vratiti status 400. To su zadane postavke, no one se mogu nadjačati. Status 400 se ne čini prikladnim kao rezultat neuspješne validacije te je bolje vratiti 422. To se može učiniti tako da se definira vlastita *ActionFilter* klasa, kao što je prikazano slikom 4.11. U *ActionFilter* klasi može se definirati status kod koji će se vratiti zajedno s tijelom odgovora.

```
internal sealed class ValidationFailedResult : ObjectResult
{
        public ValidationFailedResult(ModelStateDictionary modelState)
                : base(new ValidationResultModel(modelState))
        {
                ContentTypes.Add(MediaTypeNames.Application.Json);
                StatusCode = StatusCodes.Status422UnprocessableEntity;
        }
}
```

Slika 4.11. *Akcijski filtar.*

Što je s *GET* metodom sa slike 4.9? Nema nikakve provjere postoji li film s traženim ID-jem, a ako se bolje pogleda slika 4.6 može se vidjeti da metoda *GetAsync* baca iznimku *EntityNotFoundException* ako je traženi entitet *null*. Također, metoda se ne nalazi u *try-catch* bloku. Prema zadanim postavkama .NET Core će za sve neuhvaćene iznimke vratiti isti status kod. To nije dobra stvar jer onda se ne zna u kojem dijelu aplikacije je problem nastao i zbog čega je nastao. U tom slučaju može se definirati vlastiti *middleware* koji će hvatati iznimke te ovisno o vrsti iznimke vratiti određeni status i poruku. Slika 4.12 sadrži pojednostavljeni prikaz jednog takvog *middlewarea*. Svaki *middleware* sadrži *Invoke* metodu u kojoj se izvršava slijedeći *middleware*. Ako se u nekom od slijedećih dogodi iznimka, *exception middleware* će ju uhvatiti i vratiti odgovarajuću poruku. Zbog toga mora biti prvi u nizu, a ovisno o vrsti iznimke mogu se definirati različita tijela odgovora te različiti status kodovi. *Middleware* se može smatrati oblikovnim obrascem lanac odgovornosti.

```
public async Task Invoke(HttpContext context)
{
        try
        {
                await _next(context);
        }
        catch (Exception exception)
        {
                context.Response.ContentType = MediaTypeNames.Application.Json;
                await HandleException(context, exception);
        }
}

private async Task HandleException(HttpContext context, Exception exception)
{
        var type = exception switch
        {
                JsonException => typeof(MalformedJsonNotification),
                IDuplicateEntityException => typeof(DuplicateEntityNotification),
                IEntityNotFoundException => typeof(EntityNotFoundNotification),
                IProjectionOverlapException => typeof(ProjectionOverlapNotification),
                ISeatTakenException => typeof(SeatTakenNotification),
                IEmailSenderException => typeof(EmailSenderNotification),
                _ => typeof(UnknownErrorNotification)
        };

        var notification = CreateResponse(type, context, exception);
        var response = JsonSerializer.Serialize(notification, JsonOptions.GetOptions());
        await context.Response.WriteAsync(response);
}
```

Slika 4.12. *Exception middleware.*

Na slici 4.13 je prikazan DTO za entitet film. Prikazani DTO služi za slanje podataka o filmu prilikom *GET* zahtjeva za dohvaćanje svih filmova. DTO za dohvaćanje pojedinog filma se razlikuje od onog za dohvaćanje njih svih. Nadovezujući se na sliku 4.2 vidljiva je razlika u imenima i tipovima podataka par atributa, a neki nisu prisutni. Film ima relaciju više na više prema entitetima *Genres* i *People* te je potrebna treća tablica u bazi kako bi se relacije uspješno spremile. Prilikom *GET* zahtjeva u objektu za prijenos podataka informacija o trećoj tablici neće se slati.

```
public record MovieDto
{
        public string Id { get; init; } = null!;
        public string Title { get; init; } = null!;
        public uint Year { get; init; }
        public uint Duration { get; init; }
        public string Tagline { get; init; } = null!;
        public string Summary { get; init; } = null!;
        public bool InCinemas { get; init; }
        public string PosterUrl { get; init; } = null!;
        public string BackdropUrl { get; init; } = null!;
        public string TrailerUrl { get; init; } = null!;
        public decimal Rating { get; init; }
}
```

Slika 4.13. *DTO za film.*

Za kopiranje podataka iz objekta za prijenos podataka u objekt entiteta koristi se Automapper. Kopiranje podataka između jednostavnih objekata Automapper može odraditi automatski, ali u slučaju filma gdje se koristi dodatna tablica, gdje postoje različiti tipovi podataka između atributa i gdje postoje različita imena atributa, potrebno je napraviti dodatnu konfiguraciju kako bi se pomoglo Automapperu. Stvari se dodatno kompliciraju kada treća tablica, koja povezuje entitete, sadrži i dodatne atribute. Na slici 4.14 prikazana je metoda koja kopira podatke u entitet koji će se kasnije spremiti u bazu. U bazi, glumci i redatelji su prikazani istom tablicom, a razlika se čini samo prilikom dohvaćanja podataka. Razlog tome je što glumci mogu biti i redatelji te nema potrebe za udvostručavanjem podataka. Zbog toga tablica koja povezuje filmove s osobama ima stupac *Type* koji određuje glumca ili redatelja. U slučaju glumca dodatno se spremaju i redoslijed te ime uloge u filmu.

```
private static IEnumerable<MoviePerson> MapActorsAndDirectorsFromDto(
        CreateMovieDto movieDto, Movie movie)
{
        var moviePeople = new List<MoviePerson>();
        moviePeople.AddRange(movieDto.Actors.Select((actorDto, index) => new MoviePerson {
                CreatedAt = SystemClock.Instance.GetCurrentInstant(),
                PersonId = actorDto.Id, MovieId = movie.Id,
                Type = "Actor", Order = (uint) index + 1,
                Character = actorDto.Character
        }));
        moviePeople.AddRange(movieDto.Directors.Select(personId => new MoviePerson {
                CreatedAt = SystemClock.Instance.GetCurrentInstant(),
                PersonId = personId, MovieId = movie.Id,
                Type = "Director", Order = 0,
                Character = "Director"
        }));
        return moviePeople.ToImmutableList();
}
```

Slika 4.14. *Isječak metode iz klase koja kopira podatke iz DTO u entitet.*

Dalje, potrebno je definirati DTO za *POST*, *PUT* i *PATCH* zahtjeve. Kako ovi zahtjevi služe za stvaranje i ažuriranje podataka od važnosti je pobrinuti se da dolazni podaci odgovaraju određenim kriterijima. Za utvrđivanje podataka koriste se validacije. Validacije se pišu iznad *propertiesa*, slično anotacijama u *Spring Bootu*. Sa slike 4.15, koja prikazuje DTO žanra, odmah je uočljivo kako ime žanra ne smije sadržavati brojeve, mora biti između 2 i 32 znaka te mora započeti velikim slovom. Navedene validacije odnose se samo na DTO. Za validacije u bazi koristi se drugačija konfiguracija.

```
public record CreateGenreDto
{
        [ForbidNumbers]
        [FirstLetterUppercase]
        [MinLength(2, ErrorMessage = "Minimum length is {1} characters.")]
        [MaxLength(32, ErrorMessage = "Maximum length must not exceed {1} characters.")]
        [Required(ErrorMessage = "This field is required.")]
        public string Name { get; init; } = null!;
}
```

Slika 4.15. *Validacija atributa za kreiranje žanra.*

5. KORIŠTENJE *WEB* APLIKACIJE

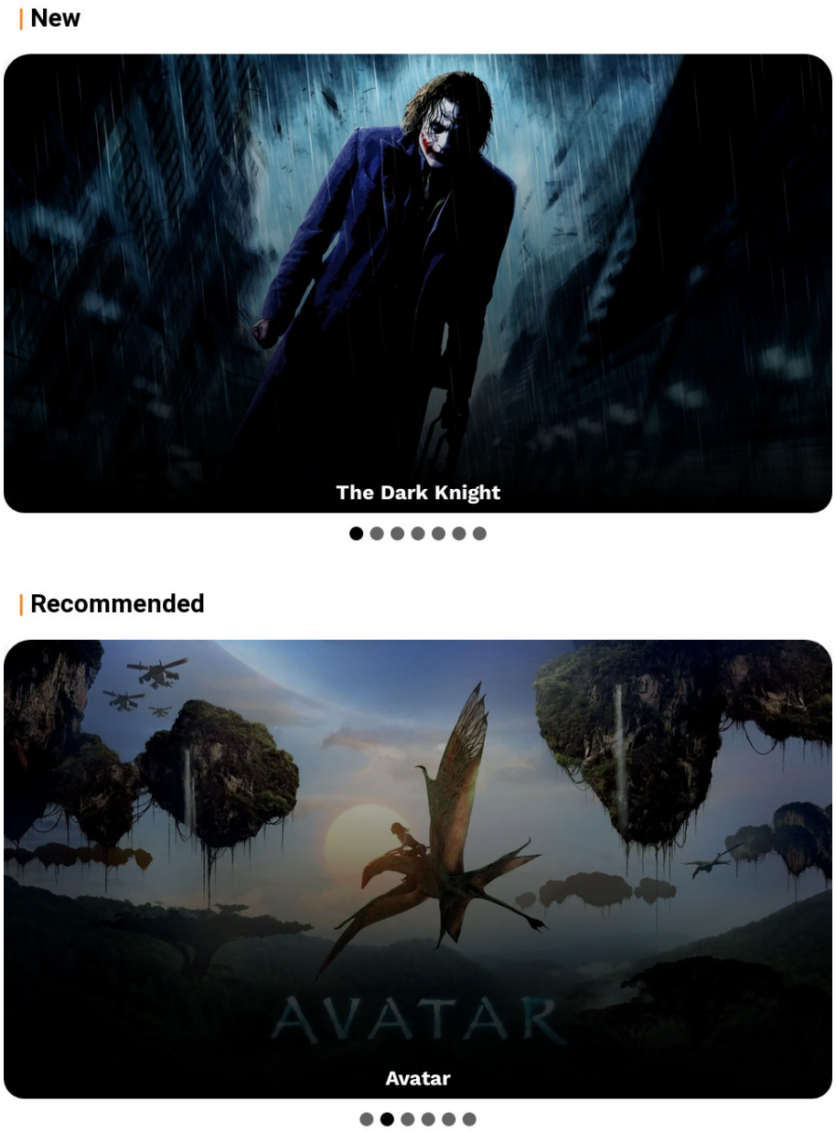
U ovom poglavlju opisane su korisničke upute, tj. načini korištenja *web* aplikacije. Sama aplikacija pokreće se preko Dockera, a to znači da nije potrebno instalirati specifične inačice Node.js i .NET SDK biblioteka. Dostupne mogućnosti prilikom korištenja aplikacije ovise o ulozi korisnika. Uloge su neregistrirani korisnik, registrirani korisnik i korisnik administrator. U nastavku poglavlja opisane su mogućnosti koje svaka uloga ima.

5.1. Neregistrirani korisnik

Dolaskom na početnu stranicu korisniku su prikazani istaknuti filmovi koje je administrator postavio. Slika 5.1 prikazuje istaknute filmove. Prvi i najnoviji film se uvijek nalazi na vrhu, dok su ostali niže postavljeni u *carousel* (vrtuljak). Kartica istaknutog filma se sastoji od pozadinske slike i pozadinskog preklapanja (engl. *overlay*) tamne ili svijetle boje radi lakšeg čitanja teksta.

Slika 5.1. *Istaknuti filmovi.*

Pomicanjem stranice prema dolje korisnik dolazi do dva nova *carousela* koji su prikazani slikom 5.2. Na prvom *carouselu* izlistavaju se novi dodani filmovi, tj. filmovi su sortirani prema datumu kreiranja. Drugi *carousel* prikazuje preporučene filmove. To su filmovi koji su najbolje ocijenjeni, koji su dobili razne nominacije ili su manje poznati no svejedno su vrijedni gledanja. Njih također izabire administrator.

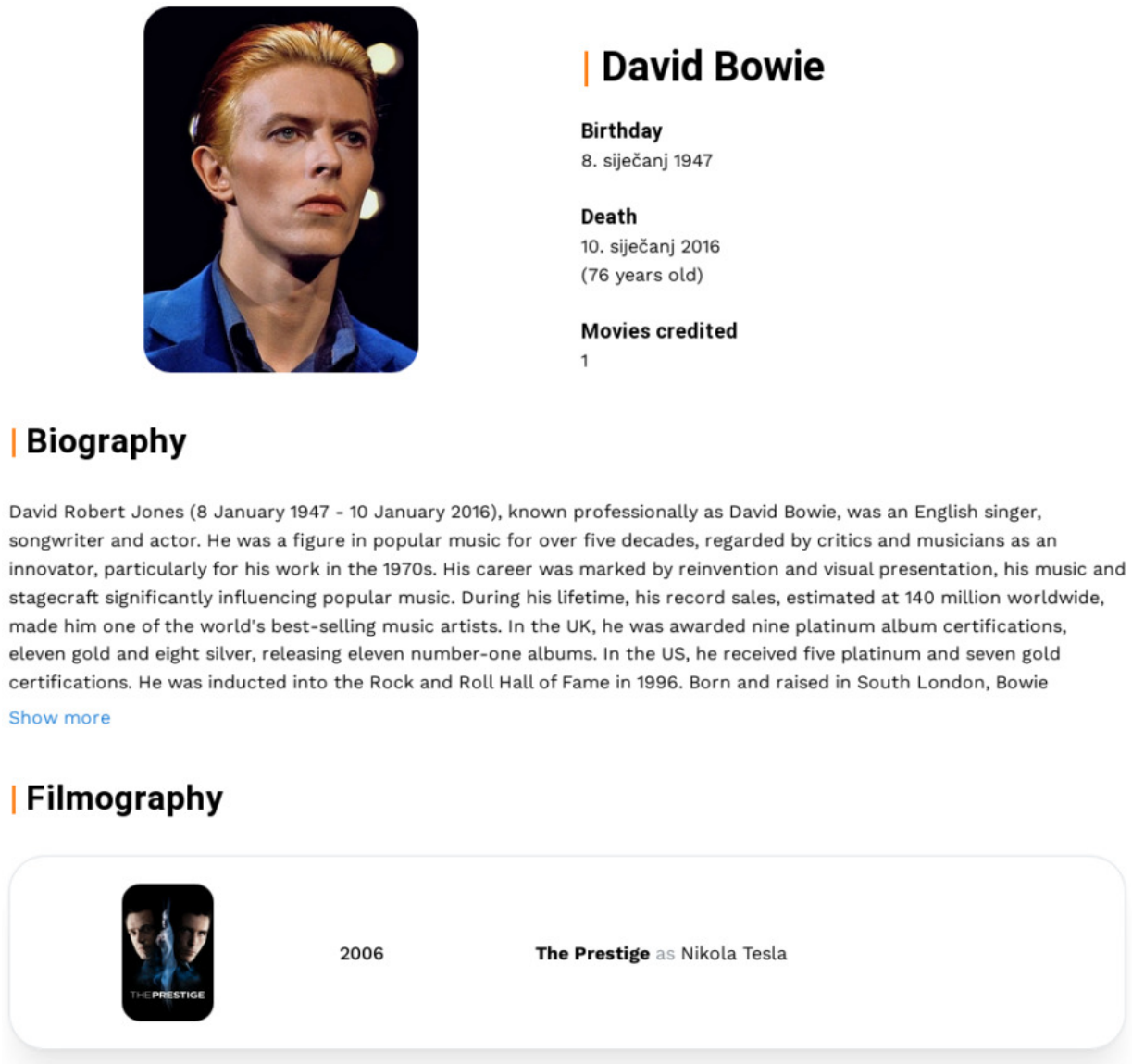


Slika 5.2. *Novi i preporučeni filmovi.*

Dalje, pri kraju stranice prikazane su drugačije kartice filmova. Na karticama su prikazani filmovi koji se na taj dan prikazuju ili su se prikazivali tijekom dana. Kartice su manje u odnosu na prethodne, nalaze se unutar *carousela* te sadrže poster umjesto pozadinske slike i ime filma. Pri samom kraju stranice nalaze se kartice glumaca i redatelja rođenih na taj dan. Kartica osobe sadrži sliku koja je zaobljena, puno ime te koliko godina imaju. Mobilna inačica početne stranice je

identična *desktop* inačici. Također, sve prikazane kartice su klikabilne.

Klikom na karticu osobe korisnik je preusmjeren na zasebnu stranicu. Na toj stranici, prikazanoj slikom 5.3, korisnik ima uvid u više detalja pojedine osobe. Detalji stranice sadrže sliku osobe u većoj rezoluciji, datum rođenja i smrti, biografiju te filmografiju. Kartica filmografije zauzima cijelu širinu zaslona, a na njoj se nalaze poster filma, godina filma te ime filma i uloga u tom filmu. Ako je osoba redatelj uloga se neće prikazivati. Kako biografija može biti poduža, prikazuje se prvih nekoliko rečenica, a klikom na gumb "Prikaži više" prikazuje se cijeli tekst biografije. U mobilnoj inačici slika osobe nalazi se na vrhu dok se ostali detalji vertikalno slažu jedan ispod drugog, a na kartici filmografije poster filma je sakriven.

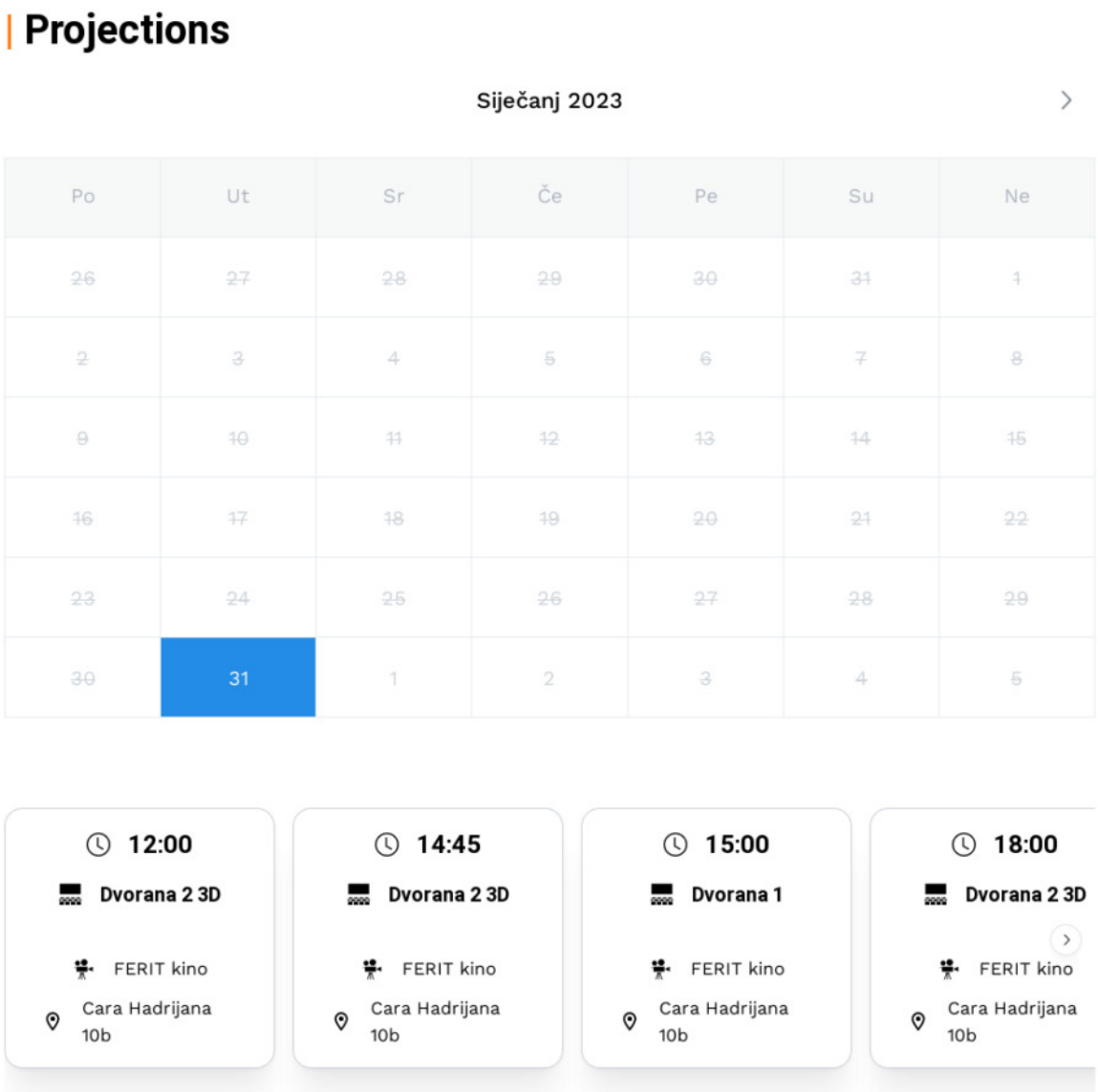


Slika 5.3. *Stranica osobe.*

Klikom na ime istaknutog filma (ili na poster unutar kartice filma koji se prikazuje) korisnik je preusmjeren na stranicu tog filma. Slika 5.4 prikazuje početak stranice gdje se nalaze najvažnije informacije o filmu. Informacije sadrže žanrove filma, godinu izdanja, trajanje, direktore, glumce, slogan filma i sažetak radnje filma. Informacije i poster filma nalaze se iznad pozadinske slike. Direktori i žanrovi predstavljeni su značkama (engl. *badge*). Značka direktora sadrži smanjenu sliku koja je poravnata lijevo i ime direktora koje je centrirano. Ispod pozadinske slike, koja se prostire preko cijele širine zaslona, nalazi se *carousel* s karticama glumaca. Kartica glumca sadrži sliku i ime glumca te ulogu koju igra u filmu.

Slika 5.4. *Početak stranice filma.*

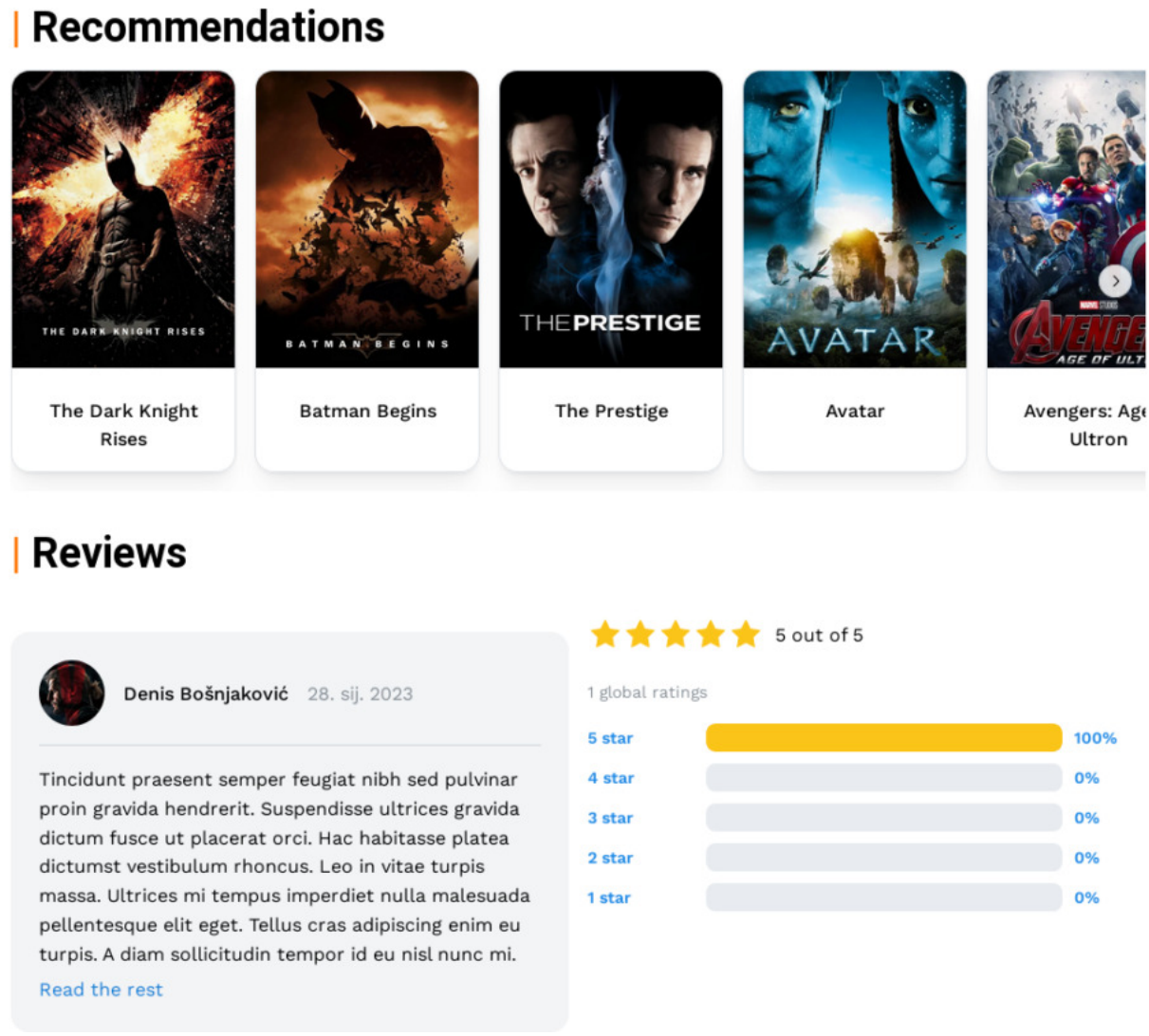
Nakon informacija o filmu dolaze projekcije. Prema slici 5.5 korisniku je prikazan kalendar. Dani koji nemaju projekcije su prekriženi. Korisnik može vidjeti projekcije za trenutni mjesec i sljedeći, a starije projekcije se ne prikazuju. Odabirom dana u kalendaru korisniku se prikazuju projekcije za taj dan. Projekcije su predstavljene karticama unutar *carousela*. Svaka kartica projekcije sadrži vrijeme početka filma, ime dvorane i kino te lokaciju kina. Kako korisnik nije registriran nema opciju odabira projekcije.



Slika 5.5. *Odabir projekcije.*

Poslije projekcija korisnik vidi prijedloge sličnih filmova i, naravno, komentare i ocjene filma. Na slici 5.6 se nalaze iste kartice filmova kao i na početnoj stranici. Prijedlozi filmova dohvaćaju se po imenu trenutnog filma, po direktoru, glavnim ulogama i na kraju sličnim žanrovima. Na

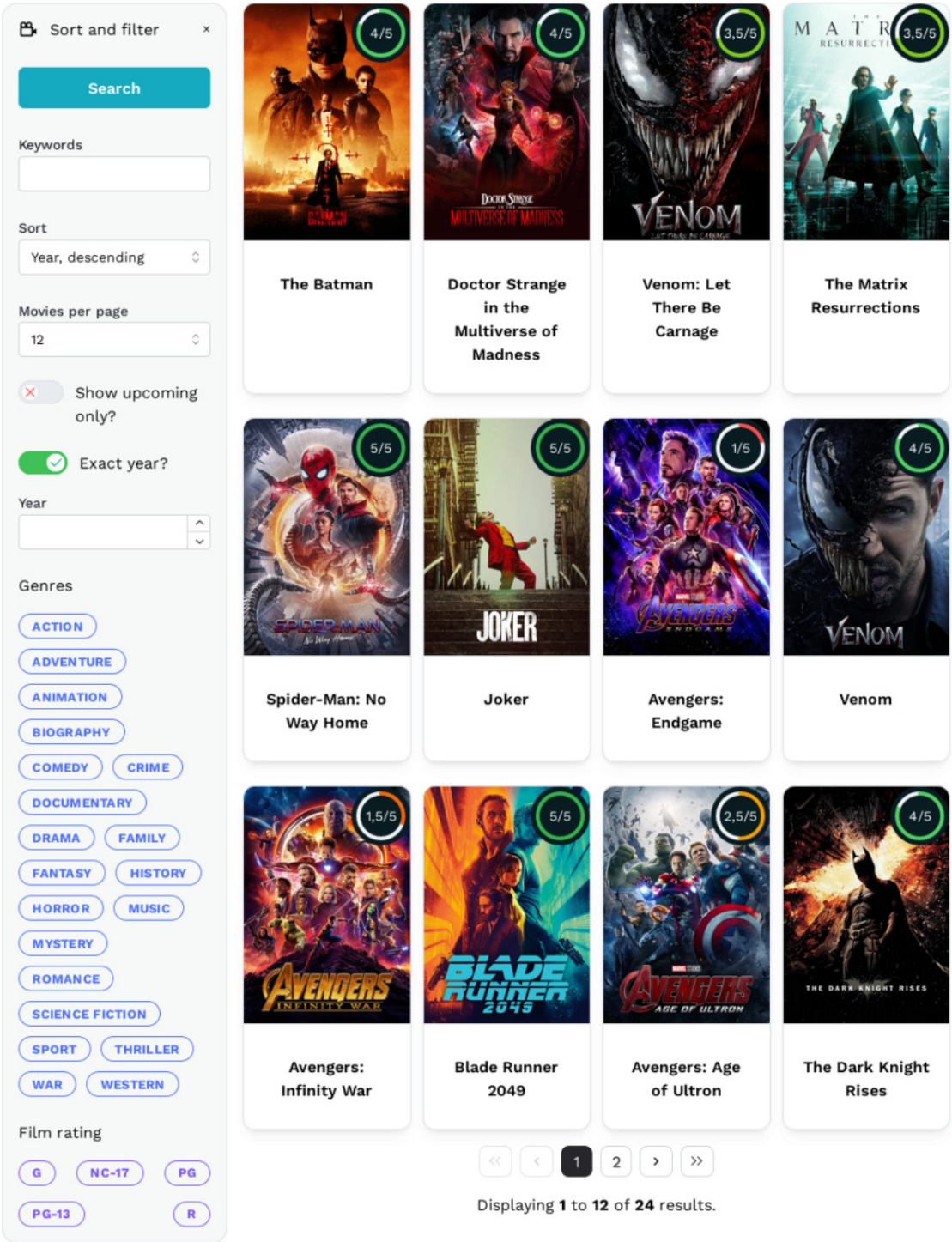
samom kraju nalaze se komentari i ocjene ostalih korisnika. Samo registrirani korisnici mogu ostaviti komentar i ocijeniti film. Prikazuju se dva najnovija komentara, a ostali se mogu dohvatiti paginacijom. Desno od komentara nalaze se ukupna ocjena filma, ukupan broj ocjena i postotak svake ocjene. U mobilnoj inačici ocjene se nalaze na vrhu, a komentari se vertikalno slažu jedan ispod drugoga. Ako je komentar predugačak klikom na gumb "Pročitaj više" otvara se modal u kojem se može pročitati sadržaj cijelog komentara.



Slika 5.6. *Prijedlozi, komentari i ocjene.*

Korisnik nije ograničen samo na filmove koji su prikazani na početnoj stranici. Klikom na gumb "Filmovi" u navigacijskoj traci korisnik je preusmjeren na stranicu sa svim dostupnim filmovima i filtrom sa strane. Slika 5.7 prikazuje sve dostupne filmove. Kartica filma je slična prethodnim, a dodatno sadrži i ukupnu ocjenu filma. Kartice se nalaze unutar *grida* (rešetke) te se automatski

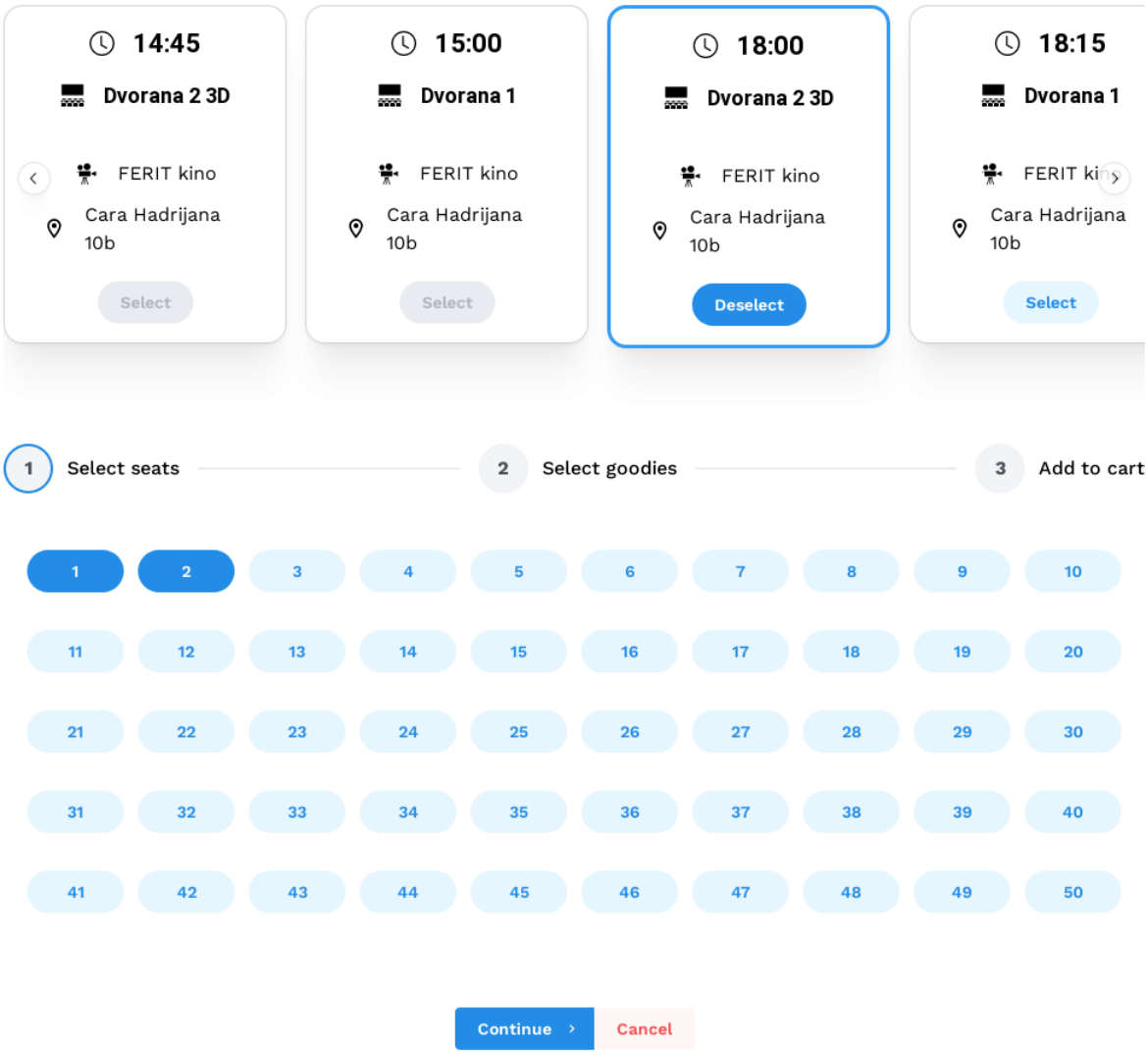
skaliraju prema veličini zaslona. Najveći dozvoljeni broj kartica po retku je pet, a najmanji dva. S desne strane *grida* nalazi se filtar. Korisnik ima mogućnost filtriranja filmova prema žanrovima, godinama i ključnim riječima ili može samo jednostavno pretraživati filmove po određenim riječima. Također može izabrati hoće li se filmovi sortirati uzlazno ili silazno, po godini ili po naslovu te broj filmova po stranici. U mobilnoj inačici filtar se nalazi iznad dostupnih filmova.



Slika 5.7. *Filtar dostupnih filmova.*

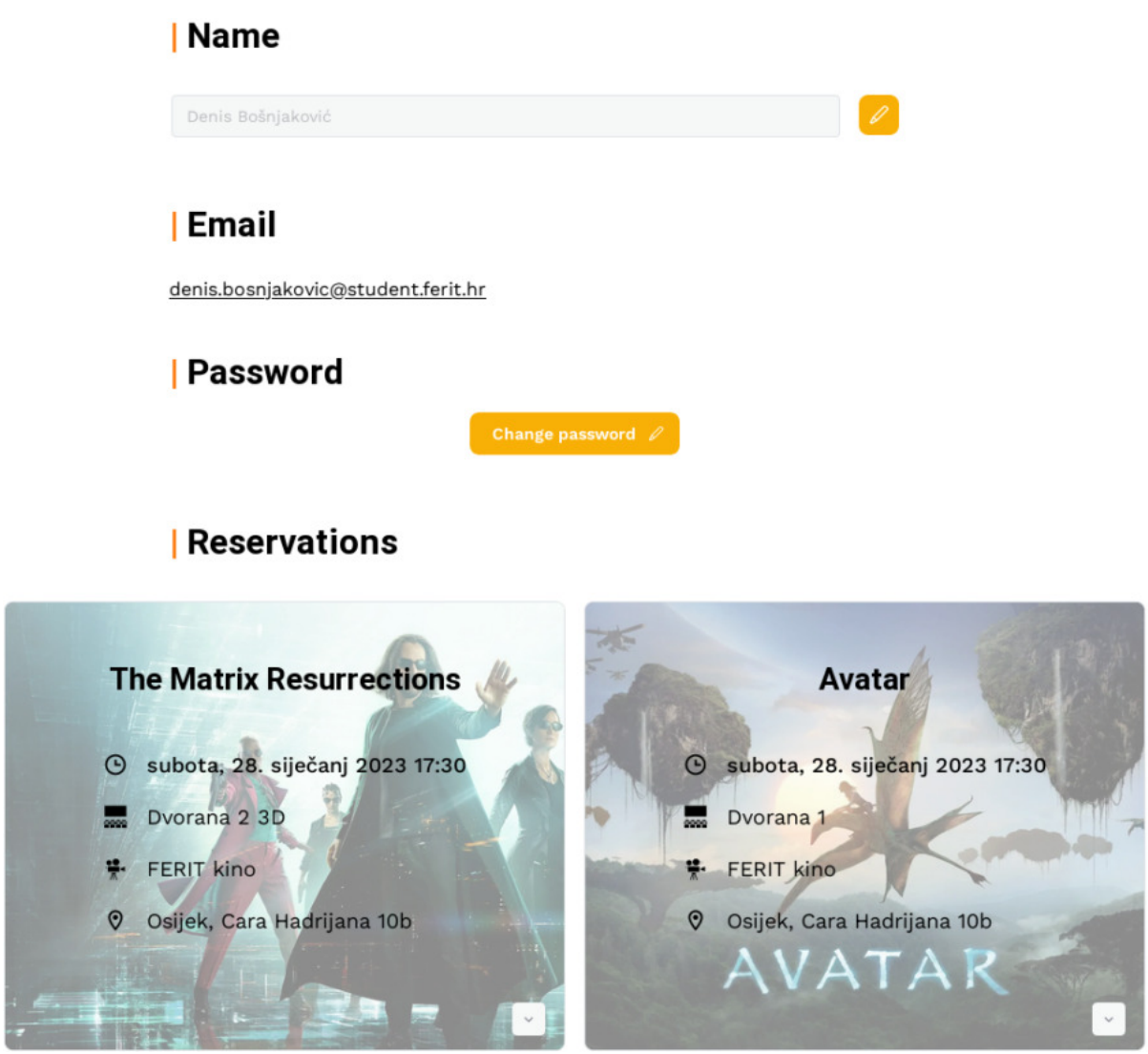
5.2. Registrirani korisnik

Nakon uspješne registracije korisnik može odabrati projekciju. Na slici 5.8 prikazan je prvi korak procesa rezervacije. Korisnik odabire željenu projekciju. Nakon odabira korisniku se prikaže raspored sjedala odabrane dvorane. Sjedalo se rezervira klikom na željeno mjesto. Naravno, moguće je odabrati više od jednog sjedala. Proces rezervacije se može pratiti unutar *steper* komponente koja se sastoji od tri koraka. Nakon odabira sjedala korisnik može odabrati piće i grickalice. U zadnjem koraku korisnik ima pregled odabranih sjedala i grickalica, a ako je zadovoljan s rezervacijom ona se dodaje u košaricu. U košarici je ispisana cijena svake rezervacije pojedinačno te ukupna cijena svih rezervacija u košarici.



Slika 5.8. *Rezervacija sjedala.*

Na stranici računa, koja je prikazana slikom 5.9, korisnik ima uvid u postavke svog računa. Može promijeniti sliku računa, ime i lozinku. Na kraju stranice korisnik može vidjeti sve projekcije koje je rezervirao. Projekcije su prikazane karticama i nalaze se u *gridu*. Klikom na padajući izbornik na dnu kartice korisnik može vidjeti detalje pojedine rezervacije kao što su broj rezerviranih sjedala te odabrana pića i grickalice.

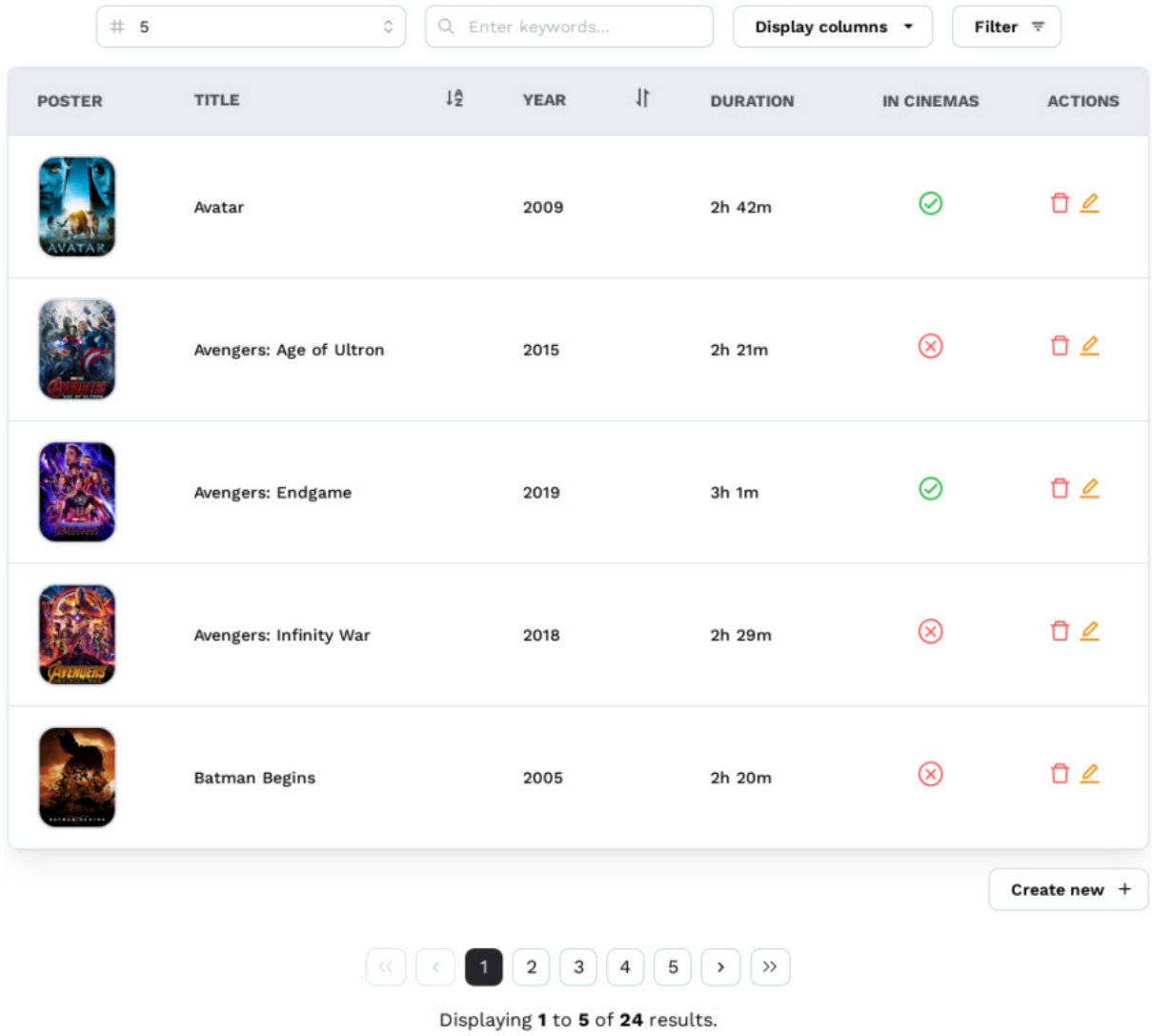


Slika 5.9. *Stranica računa.*

5.3. Korisnik administrator

Korisnik kao administrator ima najveće ovlasti prilikom korištenja aplikacije. Ima sve mogućnosti kao i običan korisnik te mnoštvo ostalih. Administrator može dodavati nove filmove te

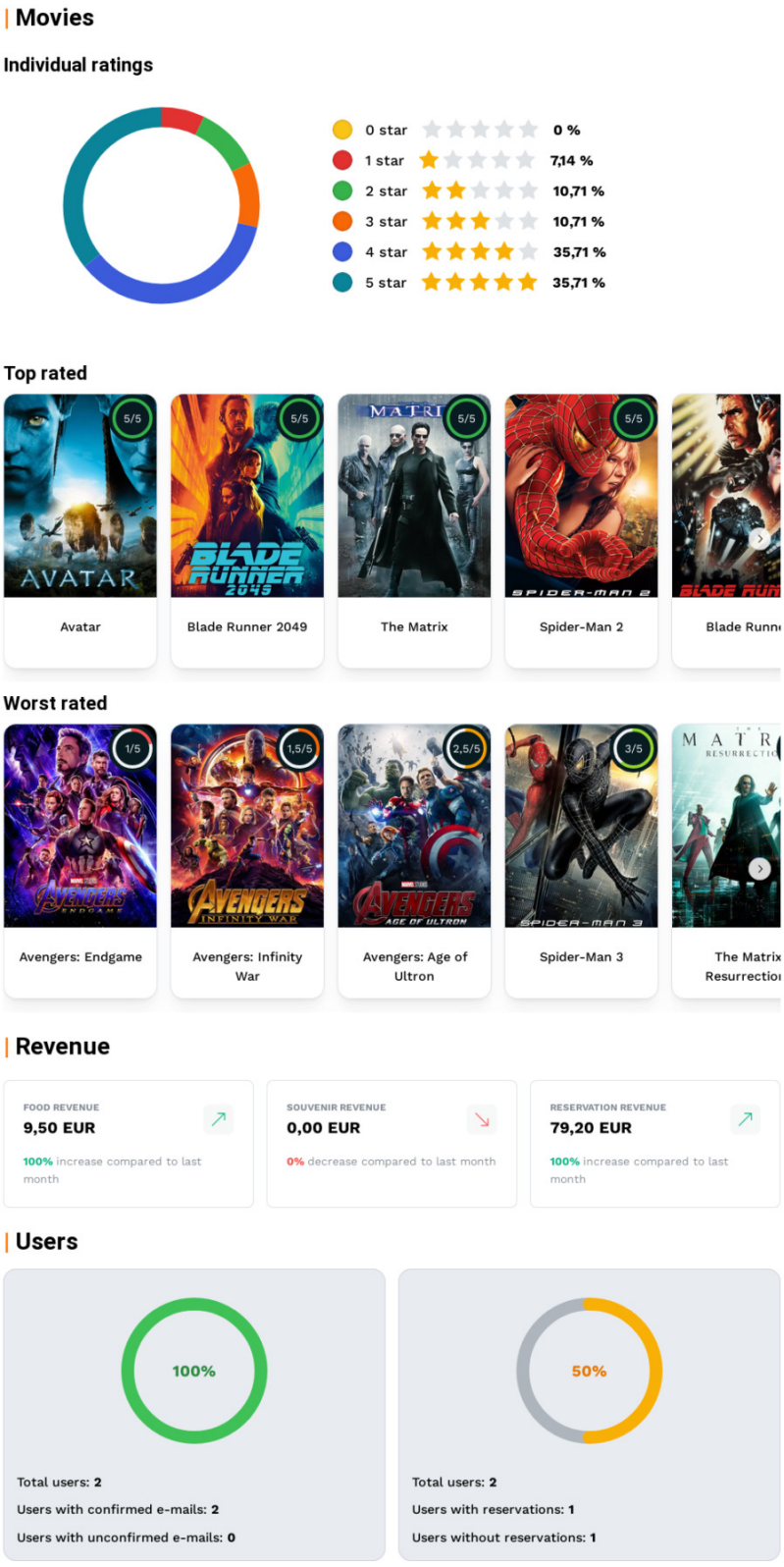
uređivati i brisati postojeće. Ovlast se ne odnosi samo na filmove, tu pripadaju i žanrovi, osobe, dvorane, sjedala pa čak i komentari te ocjene. Ukratko, administrator može upravljati gotovo svim entitetima koji postoje u aplikaciji. Primjer upravljanja se može vidjeti na slici 5.10 na kojoj je prikazana tablica filmova. Administrator ima mogućnost filtriranja filmova kao i običan korisnik, razlika je u tome što su neki od njih na "brzom biranju" poput pretraživanja, sortiranja i broja rezultata po stranici. Tablice administracije su promjenjive, što znači da se pojedini stupci, ako su nepotrebni, mogu privremeno sakriti što omogućava lakše pregledavanje tablica na mobilnom uređaju.



Slika 5.10. *Administracijska tablica.*

Vrlo važan dio svake administracije je statistika. Slika 5.11 prikazuje primjer jedne na kojoj su prikazani postotci uspješno registriranih korisnika te broj korisnika koji imaju barem jednu rezervaciju. Također, administrator vidi najbolje i najlošije ocijenjene filmove te postotak za svaku

ocjenu. Najvažniji dio je prikaz prihoda gdje je omogućen uvid u količinu dobivenu u odnosu na prethodni mjesec za pića, grickalice, suvenire i rezervacije.



Slika 5.11. *Statistika filmova, prihoda i korisnika.*

6. ZAKLJUČAK

U ovom završnom radu prikazan je postupak izrade *web* aplikacije za kino koja osim mogućnosti rezerviranja sjedala i pića može poslužiti i kao aplikacija za prikaz detaljnih informacija o filmovima te osobama koje su dio filma. Klijentska strana aplikacije je intuitivna te je responzivna i na mobilnim i na *desktop* uređajima. Cijeli proces poslužiteljske strane prati najbolje konvencije pisanja koda te konvencije koje su se u praksi učinile najboljima, a korištene tehnologije i razlozi implementacije su detaljno opisani. Za pregled informacija o filmu nije se potrebno registrirati, one su dostupne svima. Kako bi mogli rezervirati sjedalo ili ostaviti komentar na film potrebno je registrirati se. Prilikom registracije korisniku će doći verifikacijska e-pošta kojom mora potvrditi svoj račun radi sigurnosti, a do tada se ne može prijaviti u aplikaciju.

Prvi korak unaprjeđenja aplikacije bio bi pretvorba poslužiteljskog dijela u mikroservise. Pojedini dijelovi aplikacije, poput onih za dodavanje žanrova i dvorana se ne koriste toliko često kao ostali te bi bilo idealno njima smanjiti resurse. U monolitnim aplikacijama to nije izvedivo jer se onda cijelom sustavu moraju uskratiti resursi. Također, spomenuti dijelovi ne moraju raditi u stvarnom vremenu tako da je njihova implementacija zadovoljavajuća. Drugi korak unaprjeđenja odnosi se na prethodno spomenuti problem. Rezervacija sjedala i ažuriranje inventara je nešto što bi se trebalo izvoditi u stvarnom vremenu kako ne bi došlo do dohvaćanja krivih podataka. Relacijske baze podataka su dosta spore ako se konstantno mora izvršavati veliki broj takvih operacija u kratkom vremenu. Rješenje bi bilo koristiti bazu podataka kao Redis, koja je vrlo brza *in-memory* baza, za mikroservis koji je zadužen za ažuriranje i prikazivanje inventara korisnicima.

LITERATURA

[1] *CineStar. O nama*. https://www.blitz-cinestar.hr/o-nama. Pristupljeno 27. svibnja 2022.

[2] *IMDb. Help*. https://help.imdb.com/article/imdb/general-information/what-is-imdb/G836CY29Z4SGNMK5. Pristupljeno 27. svibnja 2020.

[3] *The Movie Database. About TMDB*. https://www.themoviedb.org/about. Pristupljeno 27. svibnja 2022.

[4] *Indeed. What Is a Web Application? How It Works, Benefits and Examples*. https://www.indeed.com/career-advice/career-development/what-is-web-application. Pristupljeno 31. svibnja 2022.

[5] *Maze. UX vs UI: Is There a Difference Between UX and UI?* https://maze.co/blog/ui-vs-ux. Pristupljeno 4. lipnja 2022.

[6] *Hubspot. What is Back-End Developer?* https://blog.hubspot.com/website/back-end-developer. Pristupljeno 4. lipnja 2022.

[7] *Seobility. What is Frontend?* https://www.seobility.net/en/wiki/Frontend. Pristupljeno 4. lipnja 2022.

[8] *iTrobes. What are the advantages and disadvantages of web applications?* https://www.itrobes.com/what-are-the-advantages-and-disadvantages-of-web-applications. Pristupljeno 4. lipnja 2022.

[9] *Electron. Build cross-platform desktop apps with JavaScript, HTML, and CSS*. https://www.electronjs.org. Pristupljeno 11. lipnja 2022.

[10] *Drew DeVault's blog. Electron considered harmful*. https://drewdevault.com/2016/11/24/Electron-considered-harmful.html. Pristupljeno 11. lipnja 2022.

[11] *TutorialsTeacher. .NET Core Overview*. https://www.tutorialsteacher.com/core/dotnet-core. Pristupljeno 12. lipnja 2022.

[12] *Microsoft Docs. C# | Modern, open-source programming language for .NET*. https://dotnet.microsoft.com/en-us/languages/csharp. Pristupljeno 12. lipnja 2022.

[13] *C# Corner. LINQ In C#.* https://www.c-sharpcorner.com/UploadFile/72d20e/concept-of-linq-with-C-Sharp. Pristupljeno 12. lipnja 2022.

[14] *PostgreSQL. About*. https://www.postgresql.org/about. Pristupljeno 13. lipnja 2022.

[15] *Microsoft Docs. Overview of Entity Framework Core*. https://docs.microsoft.com/en-us/ef/core. Pristupljeno 13. lipnja 2022.

[16] *TutorialsTeacher. Dependency Inversion Principle*. https://www.tutorialsteacher.com/ioc/dependency-inversion-principle. Pristupljeno 14. lipnja 2022.

[17] *NodaTime. Core types quick reference*. https://nodatime.org/3.1.x/userguide/core-types. Pristupljeno 14. lipnja 2022.

[18] *Microsoft Docs. Overview of TypeScript*. https://docs.microsoft.com/en-us/learn/modules/typescript-get-started/2-typescript-overview. Pristupljeno 12. lipnja 2022.

[19] *React. A JavaScript library for building user interfaces*. https://reactjs.org. Pristupljeno 13. lipnja 2022.

[20] *Axios Docs. Getting Started*. https://axios-http.com/docs/intro. Pristupljeno 15. lipnja 2022.

[21] *MobX. The gist of MobX*. https://mobx.js.org/the-gist-of-mobx. Pristupljeno 15. lipnja 2022.

[22] *GeeksforGeeks. What is React-Router-DOM?* https://www.geeksforgeeks.org/what-is-react-router-dom. Pristupljeno 15. lipnja 2022.

[23] *Mantine*. https://mantine.dev. Pristupljeno 15. lipnja 2022.

[24] *RedHat. What is an application architecture?* https://www.redhat.com/en/topics/cloud-native-apps/what-is-an-application-architecture. Pristupljeno 18. lipnja 2022.

[25] *Medium. Onion Architecture VS Three Layer*. https://medium.com/swlh/onion-architecture-vs-three-layer-59a9ba2c6e02. Pristupljeno 18. lipnja 2022.

[26] *What is a RESTful API?* https://www.mulesoft.com/resources/api/restful-api. Pristupljeno 30. siječnja 2023.

SAŽETAK

U ovom završnom radu ostvarena je i izrađena *web* aplikacija za upravljanje kinom i prikaz informacija o filmu. Na početku rada su opisana slična rješenja, korištene tehnologije potrebne za izradu, a kasnije načini implementacije pojedinih dijelova aplikacije. Klijentska strana *web* aplikacije ostvarena je bibliotekom React.js u programskom jeziku TypeScript, a poslužiteljska strana u .NET Core razvojnom okruženju. Za spremanje podataka odabrana je PostgreSQL baza podataka, a *web* aplikacija je postavljena na udaljenom poslužitelju koristeći Docker kontejnerizaciju za poslužiteljsku stranu i CDN (engl. *Content Delivery Network*) za klijentsku stranu. Na početnoj stranici dostupni su svi filmovi koji se prikazuju u kinu. Za rezervaciju i komentare je potrebno prijaviti se, dok za pregled informacija nije potrebno. Aplikacija je responzivna i jednostavna za uporabu.

Ključne riječi: arhitektura luka, baza podataka, .NET Core, kino, React.js

ABSTRACT

Title: Web application for running a movie theater

In this final thesis, a web application for managing the cinema and displaying information about the movie was realized and created. At the beginning of the thesis, similar solutions were described, as well as the technologies used for their creation and later the ways of implementing individual parts of the application. The client side of the web application was created with the React.js library in the TypeScript programming language and the server side in the .NET Core development environment. A PostgreSQL database is chosen for data storage and the web application is deployed on a remote server using Docker containerization for the server side part of the application and Content Delivery Network for the client side. All movies available in the cinema are shown on the home page. An account is needed to be able to make a reservation and leave a comment, while for viewing information it isn't needed. The application is responsive and easy to use.

Keywords: cinema, database, .NET Core, onion architecture, React.js

ŽIVOTOPIS

Denis Bošnjaković rođen je 2000. godine u Osijeku. Nakon osnovne škole upisuje Elektrotehničku i prometnu školu Osijek. Srednju školu završava 2019. godine te upisuje Fakultet elektrotehnike, računarstva i informacijskih tehnologija u Osijeku, smjer Preddiplomski stručni studij računarstvo gdje 2022. godine dobiva priznanje za uspjeh u studiranju. Na drugoj godini studija odrađuje praksu u tvrtki Mono gdje se počinje zanimati o *web* tehnologijama, a na trećoj godini studija uspješno odrađuje studentsku praksu u tvrtki CROZ gdje dodatno unaprjeđuje znanje o *web* tehnologijama i DevOps vještinama. Tijekom 2022. godine počinje raditi kao *full-stack developer*.

PRILOZI

Izvorni kod izrađene *web* aplikacije, zajedno s izvornim kodom ovog LaTeX dokumenta, može se pronaći na poveznici https://gitlab.com/db_git/vodenje-kina.