APLIKACIJA ZA DIJELJENJE TITLOVA

Kazinoti, Toni

SVEUČILIŠTE U SPLITU

SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Računarstvo

TONI KAZINOTI

Z A V R Š N I R A D

APLIKACIJA ZA DIJELJENJE TITLOVA

Split, rujan 2023.

SVEUČILIŠTE U SPLITU
SVEUČILIŠNI ODJEL ZA STRUČNE STUDIJE

Preddiplomski stručni studij Računarstvo

Predmet: Izrada web aplikacija

Z A V R Š N I R A D

Kandidat: Toni Kazinoti

Naslov rada: Aplikacija za dijeljenje titlova

Mentor: Marina Rodić, viši predavač

Split, rujan 2023.

Sadržaj

Sažetak

U ovom radu opisan je postupak izrade aplikacije za dijeljenje titlova i praćenje TV serija. Aplikacija se oslanja na moderne tehnologije kao što su MongoDB, Node.js, Express.js, TypeScript, Vue.js i Quasar, te se sastoji od korisničkog sučelja i aplikacijsko programskog sučelja temeljenog na arhitekturi za prijenos reprezentativnog stanja (engl. *Representational State Transfer Application Programming Interface;* REST API) koji su zapravo potpuno međusobno neovisni, iako su povezani. Također su detaljno opisane korištene tehnologije i alati za oba dijela aplikacije, uz primjere upotrebe i načina implementacije.

Ključne riječi: Node.js, REST API, TV serije, titlovi, Vue.js

Summary

Subtitle sharing application

This paper explains the creation process of an application designed for the sharing of subtitles and tracking TV series. The application leverages modern technologies such as MongoDB, Node.js, Express.js, TypeScript, Vue.js, and Quasar. It is structured with a user interface and an application programming interface (API) based on the Representational State Transfer (REST) architecture. Fundamentally, these two rely on each other to ensure the application's seamless operation. The utilized technologies and tools for both segments of the application are comprehensively detailed, accompanied by illustrative examples of their practical usage and implementation techniques.

Keywords: Node.js, REST API, subtitles, TV series, Vue.js

1 Uvod

Rijetko se može vidjeti da TV serije danas nisu dio nečije svakodnevnice, bilo to preko lokalnih i stranih televizijskih programa ili *streaming* usluga poput Netflixa i HBO Maxa. Dosta se ljudi upušta i u ilegalne načine gledanja TV serija, iako je za to potrebno nešto više tehničkog znanja. Neovisno o načinu gledanja TV serija, ova aplikacija nudi skup funkcionalnosti koje pojednostavljuju gledanje i praćenje TV serija.

Najvažnije funkcionalnosti aplikacije su pregled svih TV serija i informacija o njima, označavanje odgledanih epizoda, dodavanja TV serija u vlastiti popis za praćenje i na kraju ono najvažnije, a to je dijeljenje titlova. Za dobivanje informacija o pojedinim TV serijama, koristi se The Movie Database (TMDB) API, koji je potpuno besplatan i javan. Također postoji i mogućnost stvaranja i ispunjavanja tuđih zahtjeva za određene titlove, prijave neispravnih titlova, zahvale te stvaranja objava vezanih za pojedine TV serije, gdje se može obavijestiti korisnike o novostima poput obnove za novu sezonu i slično. Korisnici također mogu dobivati i obavijesti (engl. *notifications*) i time lako saznati sve što ih se tiče, na primjer kada njihove titlove odobre administratori. Već iz ovoga je jasno zašto ovakva aplikacija može biti korisna široj masi, a još uz to treba napomenuti i da su tehnologije za stvaranje web aplikacija jako napredovale u zadnje vrijeme. Većina postojećih aplikacija slične tematike je starijeg datuma sa staromodnim korisničkim sučeljem.

SubWrld, kako je nazvana ova aplikacija, oslanja se na tehnologije koje se grupno nazivaju MEVN stog, a izvedena je kao jednostrana aplikacija (engl. *Single-Page Application*; SPA). MEVN izraz je zapravo nastao od početnih slova tehnologija MongoDB, Express.js, Vue.js i Node.js. Osim MongoDB, što je platforma za baze podataka, ostale tehnologije imaju nastavak „js“, iz čega se da naslutiti da su pisane u programskom jeziku JavaScript (dalje u tekstu JS).

U drugom poglavlju su navedene korištene tehnologije, koja im je primarna namjena i specifičnosti koje svaka od njih donosi. Zatim je u trećem poglavlju opisano i uz primjere kôda potkrijepljeno, kako su te iste tehnologije korištene za izradu baze podataka, poslužiteljskog te klijentskog dijela aplikacije. U četvrtom poglavlju su kratko opisane glavne funkcionalnosti aplikacije, a u petom, odnosno zadnjem, poglavlju je donesen zaključak na temelju svega što je prethodno napisano.

2 Korištene tehnologije, alati i biblioteke

U ovom poglavlju ukratko su opisane tehnologije i alati koji su korišteni pri izradi aplikacije. Razlog korištenja velikog broja tehnologija i alata je taj što se time smanjuje vrijeme potrebno za izradu aplikacije, a usput i mogućnost pojave greški (engl. *bugs*) jer su korištene tehnologije i alati najčešće dobro testirani i sigurni za upotrebu.

2.1 Node.js

Node.js je srž ove aplikacije na koju se oslanja cijela poslužiteljska strana aplikacije, ona nevidljiva korisniku. Node.js je inače nastao 2009., a glavna motivacija je bila omogućiti programerima korištenje JS programskog jezika i za poslužiteljsku stranu aplikacije. Do tada se JS isključivo koristio za dinamičku promjenu sadržaja na web stranicama.

```
// Zahtjev za http modulom
const http = require('http');
// Stvaranje server objekta
const server = http.createServer((req, res) => {
      // Obrada http zahtjeva metode GET na putanji '/pozdrav'
      if (req.url === '/pozdrav' && req.method === 'GET') {
res.setHeader('Content-Type', 'text/html');
res.write('<h2>Pozdrav sa Node.js poslužitelja!!</h2>');
            return res.end();
      }
});
// Konfiguriranje poslužitelja
server.listen(3000, () => {
console.log('Poslužitelj sluša na priključku 3000')
});
```

Ispis 1: Jednostavni primjer Node.js aplikacije

U ispisu 1 se može vidjet primjer jednostavne aplikacije napisane u Node.js koja šalje tekst napisan u HTML-u (engl. *HyperText Markup Language*) kada se pristupi web adresi sa domenom i priključkom na kojoj poslužitelj sluša, te putanjom „/pozdrav“. U

ovom slučaju domena je „localhost“ (što je jednako IP adresi 127.0.0.1), budući da je aplikacija pokrenuta samo lokalno, a priključak 3000.

2.2 Express.js

Kako bi se pojednostavnio i ubrzao razvojni proces, koristi se razvojni okvir Express.js, koji se bazira na Node.js tehnologiji i poprilično je minimalističan u smislu da nudi samo najčešće korištene funkcionalnosti.

```
// Inicijalizacija express modula
const express = require('express');
const app = express();
// Obrada http zahtjeva metode GET na putanju '/pozdrav'
app.get('/about', (req,res) => {
    res.send(<h2>Hello from Express.js server!!</h2>');
});
// Konfiguriranje poslužitelja
app.listen(3000, () => {
    console.log('Poslužitelj sluša na priključku 3000');
});
```

Ispis 2: Jednostavni primjer Express.js aplikacije

Primjer u ispisu 2 je sličan primjeru iz ispisa 1, osim što je sada napisan koristeći Express.js. Već je i iz ovoliko jednostavne aplikacije vidljiv značaj korištenja razvojnog okvira. Express.js je dosta fleksibilan razvojni okvir, odnosno ne traži da se slijedi nekakav obrazac ili arhitektura. Imajući to na umu, potrebno je prije početka razvoja pojedinog projekta odlučiti o strukturi aplikacije i držati se iste tijekom cijelog razvoja.

2.3 NPM (Upravitelj paketa za Node)

U svrhu olakšavanja i ubrzavanja razvoja projekata, idealno bi bilo iskoristiti tuđi kôd koji već provjereno radi točno ono što treba, a kako bi sam taj proces bio ubrzan, razvijen je Node upravitelj paketa (engl. *Node Package Manager*, skraćeno NPM). To je jedan od najpopularnijih alata u ekosustavu Node.js koji omogućuje programerima da lako preuzimaju, instaliraju, ažuriraju i upravljaju raznim paketima koje koriste u svojim JS projektima. NPM dolazi predinstaliran sa Node.js-om, i sadrži tisuće paketa otvorenog

kôda (engl. *open-source*) koje su stvorili drugi razvojni timovi i programeri. Ovi paketi obuhvaćaju razne funkcionalnosti, kao što su rad s mrežama, manipulacija podacima, rad s datotekama, rad s korisničkim sučeljima, i još mnogo toga.

Osim preuzimanja paketa, NPM omogućuje i upravljanje verzijama paketa, što je ključno za održavanje stabilnosti i kompatibilnosti projekata. Također omogućuje i jednostavno dijeljenje i objavljivanje vlastitih paketa tako da drugi programeri mogu koristiti njihov kôd.

```
// stvaranje package.json datoteke za početak rada sa NPM
npm init
// instaliranje paketa
npm install [<naziv paketa> ...]
// ažuriranje pakete
npm update [<naziv paketa> ...]
// izvršava naredbu definiranu u package.json, obično pokreće aplikaciju
npm start
```

Ispis 3: Najčešće korištene naredbe pri razvoju aplikacije

U ispisu 3 su prikazane i objašnjene osnovne naredbe za rad sa paketima pri razvoju aplikacije. Tu su još i mnoge druge, od kojih je većina zapravo rijetko korištena. Svrha „package.json“ datoteke je za definiranje metapodataka aplikacije, odnosno samog projekta. Tu se nalaze naziv projekta, verzija projekta, autori i kontakt informacije, skripte koje se mogu pokretati u konzoli, lista ovisnosti (engl. *dependencies*), odnosno paketa potrebnih da se aplikacija pokrene. Također postoji i lista ovisnosti za razvoj, tu se nalaze paketi koji nisu nužni za pokretanje, ali jesu za daljnji razvoj. Postoje još i brojni drugi metapodatci.

```
{
  "name": "api",
  "version": "0.0.0",
  "private": true,
  "scripts": {
    "build": "npx tsc",
    "start": "node dist/server.js",
```

```
    "dev": "concurrently \"npx tsc --watch\" \"nodemon -q
dist/server.js\""
  },
  "dependencies": {
    "archiver": "^5.3.1",
    [...]
  },
  "devDependencies": {
    "concurrently": "^8.0.1",
    [...]
  }
}
```

Ispis 4: package.json datoteka poslužiteljskog dijela aplikacije

U ispisu 4 je prikazana „package.json“ datoteka poslužiteljskog dijela aplikacije, s tim da nisu prikazane sve ovisnosti zbog čitljivosti. Velik broj paketa je korišten, a oni najvažniji su opisani u trećem poglavlju.

2.4 MongoDB – Baza podataka

Baza podataka služi za organiziranje, pohranjivanje i upravljanje informacijama. Podaci se organiziraju tako da se omogući brzu pretragu, manipulaciju i izvještavanje o istima. U ovom slučaju, svaki novi korisnik, titlovi, objava itd., sprema se u bazu podataka kako bi se kasnije ti isti podaci mogli izvući u bilo koje vrijeme.

Ova aplikacija se oslanja na MongoDB platformu. Umjesto klasične tablične strukture koja se koristi u relacijskim bazama podataka, MongoDB koristi dokumentno orijentiranu bazu podataka (slično JSON-u) za pohranu podataka. Ova fleksibilna struktura omogućuje jednostavno skaliranje i prilagodbu podacima različitih oblika i formata.

Ipak, potpuna fleksibilnost sa sobom povlači mnoge probleme jer ne postoje pravila za spremanje dokumenata. Idealno bi bilo ostvariti balans između beneficija takve dinamičke sheme te potrebe za dosljednosti, validacijom i organizacijom podataka. Upravo u tu svrhu je korišten već prethodno naveden paket `mongoose`, a koji ima sljedeće funkcionalnosti:

- Objektno modeliranje podataka (engl. *Object Data Modeling*, ODM) – definiranje struktura podataka u obliku JS objekata. Ovaj pristup olakšava preslikavanje (engl. *mapping*) JS objekata na dokumente u MongoDB bazi.
- Omogućuje definiranje shema za dokumente. Shema definira strukturu dokumenta, uključujući tipove podataka, validacije, zadane vrijednosti i više.
- Omogućuje kreiranje modela koji predstavljaju kolekcije u MongoDB bazi. Modeli olakšavaju operacije poput dodavanja, ažuriranja, brisanja i dohvata dokumenata.
- Podržava međuprograme i udice (engl. *hooks*) koje se mogu koristiti za izvođenje dodatne logike prije ili poslije operacija nad bazom podataka.
- Omogućuje povezivanje dokumenata putem referenci i populacije, što olakšava rad s povezanim podacima.
- Podržava definiranje indeksa nad poljima u dokumentima radi optimizacije upita.

2.5 Vue.js

Vue.js je progresivni JS razvojni okvir za lakšu i bržu izradu interaktivnih i dinamičnih korisničkih sučelja, koristeći komponentnu arhitekturu. Vue.js omogućuje razvoj modernih web aplikacija uz lakšu integraciju s drugim paketima ili već postojećim projektima. Vue.js koristi takozvani MVVM arhitektonski obrazac (engl. Model-View-ViewModel) koji se često koristi u razvoju korisničkih sučelja, posebno u okruženjima koja podržavaju povezivanje podataka (engl. *data binding*), poput WPF, Xamarin, i naravno Vue.js.

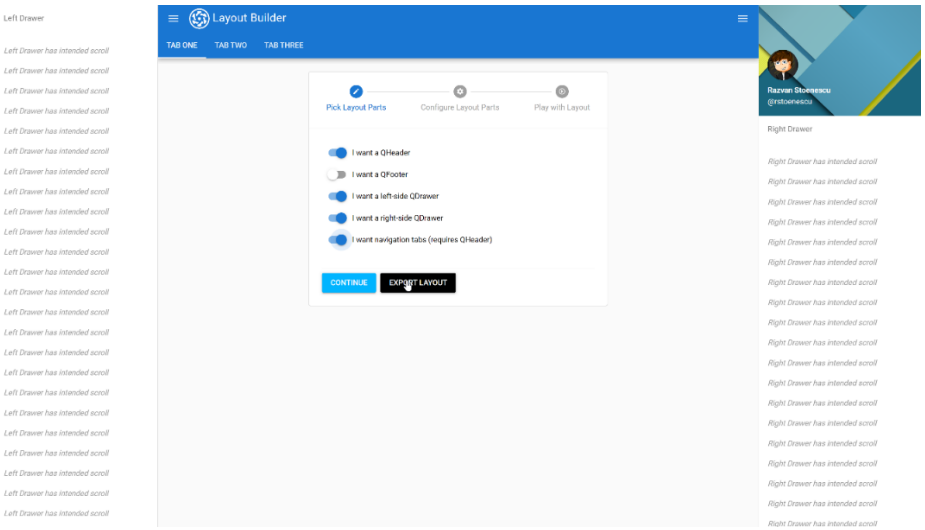
- Model: Predstavlja podatke i poslovnu logiku aplikacije. To može biti baza podataka, servisni sloj ili bilo koja druga komponenta koja manipulira podacima.
- Prikaz: Predstavlja korisničko sučelje i njegov izgled. U web aplikacijama to su HTML i CSS.
- Prikazni model: Ovo je posrednički sloj između modela i prikaza. Sadrži podatke i logiku koji su potrebni za prikazivanje u korisničkom sučelju. Omogućuje povezivanje podataka između modela i prikaza bez direktnog utjecaja na model. Također omogućuje komunikaciju između prikaza i modela.

MVVM se često koristi za razdvajanje odgovornosti u aplikaciji, što omogućuje veću fleksibilnost, testiranje i održavanje kôda. Ova arhitektura je osobito korisna u

aplikacijama s bogatim korisničkim sučeljem gdje se podaci moraju često ažurirati i prikazivati u skladu s promjenama u modelu. Važno je još napomenuti i kako Vue.js također podržava TypeScript (dalje u tekstu TS) za tipizaciju što je pri izradi ove aplikacije i iskorišteno.

2.6 Quasar

Kako bi se još više ubrzao razvoj korisničkog sučelja, koristi se vrlo popularan Vue.js razvojni okvir Quasar koji nudi dizajnirane, stilizirane i responzivne komponente za upotrebu. Ima vrlo bogatu dokumentaciju te nekoliko korisnih alata na službenoj web stranci [1]. Alat koji je potrebno istaknuti je vizualni alat za slaganje izgleda stranice (engl. *layout builder*), a čiji se izgled može vidjeti na slici 1.



Slika 1: Izgled vizualnog alata za slaganje izgleda stranice

Ne samo da je proces vrlo jednostavan i brz, nego se odmah može vidjeti i krajnji rezultat. Aplikacija sa podešenim izgledom se može preuzeti i pokrenuti lokalno pomoću jednostavne naredbe `quasar dev`. Tijekom razvijanja aplikacije, promjene datoteka su odmah prepoznate i nema potrebe za ponovnim pokretanjem aplikacije, kao niti za osvježavanjem u pregledniku. Naravno, nedostatak je što nema potpune kontrole nad izgledom aplikacije, ali ako se to želi postići onda nema smisla uopće koristiti jedan ovakav razvojni okvir.

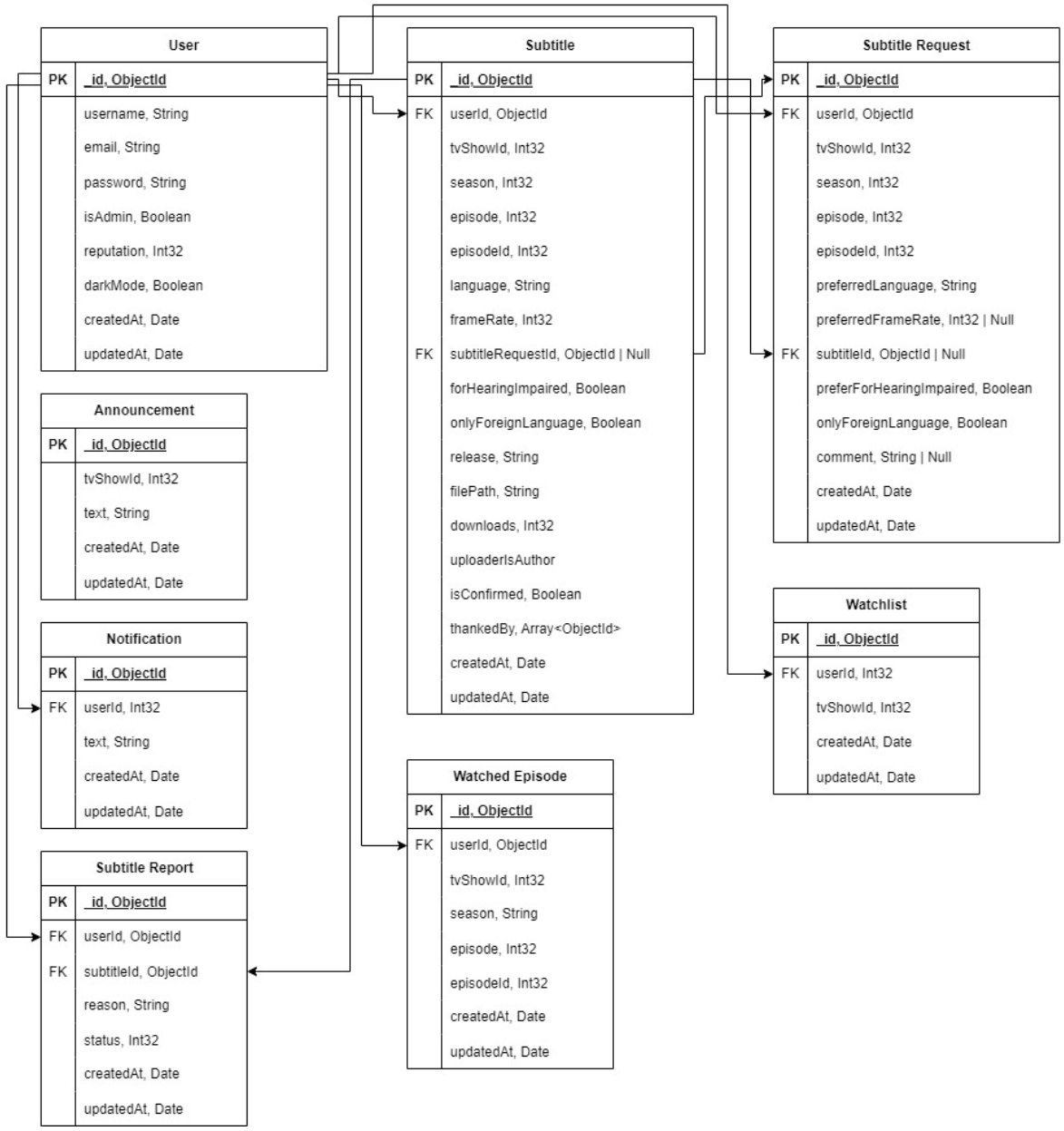
3 Postavljanje i struktura aplikacije

Cilj ovog poglavlja je opisati tehničku izvedbu temeljnih stavki konačnog proizvoda, koji je opisan u sljedećem poglavlju kroz pregled funkcionalnosti i način upotrebe korisničkog sučelja. Većina primjera bit će izvučena iz aplikacije. Poglavlje će biti podijeljeno u tri potpoglavlja, bazu podataka, poslužiteljski, te korisnički dio.

3.1 Baza podataka

Prije svega je potrebno napraviti MongoDB račun i konfigurirati grozd (engl. *cluster*), a zatim u njemu izraditi instancu baze podataka. S time će se dobiti uniformni identifikator resursa (engl. *Uniform Resource Identifier*; URI) koji je moguće koristiti za spajanje na tu instancu preko Express.js aplikacije. Preporuka je posjetiti službene stranice [2] i pratiti upute.

Svaka aplikacija ima drugačije zahtjeve i za svaku je potrebno posebno uzeti u obzir sve te zahtjeve i po njima izgraditi strukturu baze podataka koja će se kasnije implementirati i koristiti pri radu aplikacije. Promatrajući zahtjeve SubWrld aplikacije, došlo se do zaključka da je potrebno ukupno osam kolekcija u MongoDB bazi podataka. Kolekcija je skup, odnosno kolekcija dokumenata, usporediva sa tablicama kod relacijskih baza podataka, međutim bez prisiljavanja na istu strukturu svakog dokumenta. Potrebna je posebna kolekcija za korisnike, titlove, želje za titlove, prijave titlova, praćenje TV serija, praćenje odgledanih epizoda, objave i obavijesti. Na slici 2 se može vidjeti koja polja sadrže te kolekcije, kojeg su tipa, i kako su međusobno povezane.



Slika 2: E-R (engl. *Entity-Relationship*) dijagram baze podataka

Kako bi se ubrzali upiti, potrebno je pažljivo odabrati indeksirana polja. Indeksi omogućuju bazi da brže pronađe, filtrira ili razvrsta dokumente koji zadovoljavaju određene uvjete upita. Ipak, treba paziti na zauzeće prostora koje se može znatno povećati s rastom broja indeksa. Indeksiraju se samo ona polja koja će se često pojavljivati u upitima. Indeks može biti i jedinstven, što znači da nije dozvoljeno ponavljanje iste vrijednosti za to polje. Najbolji primjer za to bi bio „username“ polje kod kolekcije s korisnicima jer nije poželjno imati više korisnika sa istim korisničkim imenom. Jedan indeks može obuhvaćati i više polja (ujedinjeni indeks; engl. *compound index*), primjer bi

bio indeks na poljima „userId“ i „tvShowId“ kod kolekcije s praćenim TV serijama. Naravno, razlog za takav indeks leži u učestalosti upita koji sadrži oba ta polja, a to je svaki tu kada korisnik zatraži vidjeti vlastiti popis za praćenje.

U ispisu 5 je prikazan kôd za stvaranje sheme za obavijesti korištenjem paketa `mongoose`.

```
const notificationSchema = new Schema(
  {
    userId: {
      type: Schema.Types.ObjectId,
      ref: 'User',
      required: true,
      index: true,
    },
    text: {
      type: String,
      required: true,
    },
  },
  {
    timestamps: true,
  }
)
```

Ispis 5: Mongoose shema za obavijesti

Polje „_id“ je automatski dodano u svaku shemu i indeksirano, a svojstvo „timestamps“ dodaje polja „createdAt“ i „updatedAt“ koja automatski zapisuju kada je dokument dodan, odnosno izmijenjen.

Ispod slijedi objašnjenje prikazanih svojstava za polja, kao i nekih kojih nema na ovom konkretnom primjeru:

- type – određuje tip podatka, u aplikaciji su korišteni još i Number te Boolean tipovi.

- ref – označava kako polje referencira na dokument iz neke druge kolekcije, uspostavlja vezu između njih koristeći „_id“ polje. To omogućuje jednostavan pristup podacima iz povezanog dokumenta.
- required – označava je li polje obavezno pri unosu
- index – određuje stvara li se indeks tog polja
- default – ukoliko polje nije navedeno pri unosu, ovo određuje njegovu zadanu vrijednost

Postoje i druga svojstva koja se tiču validacije, ali u aplikaciji je to izvedeno provjerom podataka prije spremanja. U ispisu 6 se može vidjeti kako se stvara ujedinjeni indeks, odnosno indeks s više polja.

```
watchlistSchema.index({ userId: 1, tvSHowId: 1 })
```

Ispis 6: Stvaranje ujedinjenog indeksa

Linija iz ispisa 6 nalazi se odmah ispod prvotnog definiranja sheme. Broj 1 znači da će se indeks stvoriti uzlaznim redoslijedom za to polje. Naravno, -1 označava silazni redoslijed.

Nadalje, mogu se definirati metode na shemi koje će se onda moći pozivati na pojedinom dokumentu. Ovo se u aplikaciji koristi samo u shemi za korisnike, i to kao u ispisu 7.

```
userSchema.methods.matchPassword = async function (enteredPassword:
string) {
  return await bcrypt.compare(enteredPassword, this.password)
}
```

Ispis 7: Dodatna metoda na shemi za korisnike

Budući da „password“ polje u bazi zapravo ne sadrži originalnu lozinku, već je ona jednosmjerno šifrirana (što znači da se ne može dobiti originalna vrijednost iz šifrirane vrijednosti) bcrypt algoritmom. Poziva se metoda iz `bcryptjs` paketa kako bi se saznalo je li unesena lozinka pri prijavi ispravna. Metoda vraća `true` ili `false` ovisno o rezultatu usporedbe.

Također je moguće i dodavanje metoda koje će se automatski izvršiti prije ili nakon određenog događaja. U aplikaciji se ovo konkretno koristi za šifriranje lozinke prije samog unosa u bazu podataka, odnosno dodavanju i izmjeni dokumenata, prikazano u ispisu 8.

```
userSchema.pre('save', async function (next) {
  if (!this.isModified('password')) next()
  const salt = await bcrypt.genSalt(10)
  this.password = await bcrypt.hash(this.password, salt)
})
```

Ispis 8: Šifriranje lozinke prije spremanja podataka u bazu

Metoda „isModified“ se poziva kako bi se provjerilo je li polje „password“ promijenjeno otkad su se podaci prvotno izvukli iz baze podataka. Veći *salt* faktor (što rezultira većem broj iteracija pri šifriranju) povećava sigurnost šifriranja, ali i vrijeme potrebno za izračun šifrirane vrijednosti.

Postoji i mogućnost proširenja sheme s dodacima (engl. *plugins*) kako bi se proširila i promijenila njihova funkcionalnost. Takvi dodaci omogućuju ponovno korištenje određene funkcionalnosti ili operacija. U aplikaciji se koriste dva dodatka, `paginate` i `aggregate-paginate`. Oni se koriste za pojednostavljenje straničenja podataka iz baze. U ispisu 9 je prikazano kako se to postiže.

```
announcementSchema.plugin(paginate)
announcementSchema.plugin(aggregatePaginate)
```

Ispis 9: Proširenje sheme za objave s dodacima `paginate` i `aggregatePaginate`

Potrebno je napraviti razliku između `mongoose` shema i modela. Shema definira strukturu podataka za kolekciju u MongoDB bazi, dok je model instanca sheme koja omogućuje izvođenje operacija nad određenom kolekcijom. U ispisu 10 je prikazan primjer stvaranja instance definirane sheme.

```
const Watchlist = model<IWatchlist>('Watchlist', watchlistSchema)
```

Ispis 10: Instanciranje modela definirane `mongoose` sheme

U metodu `model` se proslijedi naziv modela te samu shemu. Kako bi se mogle iskoristiti prednosti TS-a, izrađeno je sučelje (engl. *interface*) za svaki pojedini model koji sadrži sve podatke i metode određenog modela. Upravo tome služi `<IWatchlist>` dio kôda, tj. kako bi se povezao model sa sučeljem.

Na kraju se model izveze (engl. *export*), npr. `export default Watchlist` i spreman je za uvoz (engl. *import*) i uporabu u drugom dijelu kôda, odnosno datoteci, koristeći `import Watchlist from '<direktorij>'`.

Dakako, potrebno se prije svega ovoga spojiti na bazu podataka koju je MongoDB dodijelio na korištenje, a koja se nalazi na udaljenom poslužitelju u njihovom vlasništvu. U ispisu 11 je prikazano kako je ostvarena konekcija u aplikaciji.

```
const connectDB = async () => {
  try {
    mongoose.set('debug', true)
    const conn = await mongoose.connect(process.env.MONGO_URI as string)
    console.log(`MongoDB connected: ${conn.connection.host}`)
  } catch (error: any) {
    console.log(`Error: ${error.message}`)
    process.exit(1)
  }
}
```

Ispis 11: Ostvarivanje konekcije sa MongoDB bazom podataka

Prije ostvarivanja same konekcije postavlja se `debug` vrijednost tako da `mongoose` u konzoli unosi zapise poput provedenog upita, stvaranja indeksa itd. U produkciji bi to svakako bilo isključeno zbog performansi. Nakon toga se poziva `connect` metoda koja prima URI i prema tome ostvaruje konekciju. Ili, ako bude neuspješno, poziva se `catch` blok, ispisuje se greška u konzolu i aplikacija se prekida izvršavati. Ova metoda se pozove samo jednom u datoteci koja se prva izvršava pri pokretanju Express.js aplikacije.

3.2 Poslužiteljski dio

Poslužiteljski dio aplikacije, odnosno sam web API, prati REST arhitektonski stil, što u prijevodu samo znači da API koristi HTTP (engl. *Hypertext Transfer Protocol*)

metode GET (dohvaćanje resursa), POST (stvaranje resursa), PUT (ažuriranje resursa), PATCH (polovično ažuriranje resursa) i DELETE (brisanje resursa), te da su resursi izloženi preko uniformnih resursnih lokatora (engl. *Uniform Resource Locator*; URL). Osim toga bi bilo još važno napomenuti da je REST bez stanja (engl. *stateless*), što znači da svaki zahtjev sadrži sve potrebne informacije za obradu tog zahtjeva, te da je u ovom slučaju odabrano predstavljanje resursa u JSON formatu. Primjer JSON objekta može se vidjet ispod u ispisu 12.

```
{
    "docs": [
        {
            "_id": "6497a2f62986b297ca39cb97",
            "username": "janedoe",
            [...]
        },
        [...]
    ],
    "totalDocs": 3,
    [...]
}
```

Ispis 12: Primjer JSON objekta

API vraća korisniku resurs u takvom formatu, međutim za lakši rad s takvim podacima u programskom jeziku je potrebno napraviti pretvorbu u pravi JS objekt. Taj proces naziva se deserijalizacija. Vrlo je važan i obrnut proces, serijalizacija, koji se događa neposredno prije slanja odgovora korisniku.

Oba procesa se u slučaju ove aplikacije odvijaju automatski. Kod razvojnog okvira Express.js korištenjem `express.json()` međuprograma, svi zahtjevi će se serijalizirati i svi odgovori deserijalizirati automatski, dok se kod korisničkog dijela aplikacije koristi paket `axios` koji automatski deserijalizira dobiveni odgovor od API-a. U ispisu 13 su navedeni neki primjeri URL-ova resursa, za bolje razumijevanje kako ovaj API funkcionira.

```
// dohvaćanje titlova sa _id 6497a2f62986b297ca39cb95 (GET metoda)
http://localhost:3000/subtitles/6497a2f62986b297ca39cb95
```

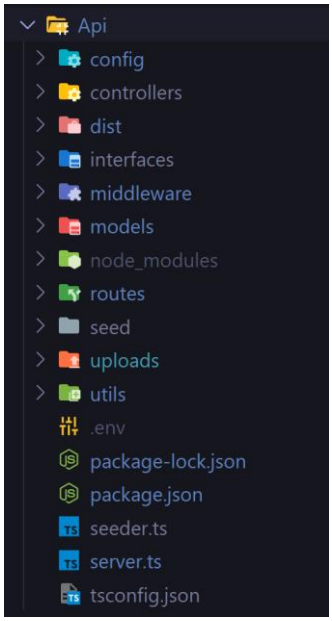
```
// ažuriranje titlova sa _id 6497a2f62986b297ca39cb95 (PUT metoda)
http://localhost:3000/subtitles/6497a2f62986b297ca39cb95
// brisanje titlova sa _id 6497a2f62986b297ca39cb95 (DELETE metoda)
http://localhost:3000/subtitles/6497a2f62986b297ca39cb95
// dodavanje titlova (POST metoda)
http://localhost:3000/subtitles
// dohvaćanje svih titlova sa straničenjem, stranica 1 (GET metoda)
http://localhost:3000/subtitles?page=1
```

Ispis 13: Primjeri postojećih URL-ova resursa SubWrld API-a

3.2.1 Struktura direktorija

Najnovija verzija Node.js-a može se preuzeti sa službenih stranica [3]. Nakon toga se inicira projekt sa `npm init` naredbom, i time dobije datoteka „package.json“, uključujući i instalirane pakete. Nakon toga je još samo potrebno instalirati `express` paket.

Express.js ne nameće nikakvu strukturu direktorija koju treba pratiti, tako da je sasvim moguće sve strpati u jednu datoteku proizvoljnog naziva. Aplikacija se pokreće sa `node <naziv datoteke>`. Naravno, u interesu programera je bolje strukturirati i organizirati kôd. Na slici 3 se može vidjeti struktura poslužiteljskog dijela SubWrld aplikacije.



Slika 3: Struktura direktorija poslužiteljskog dijela

Objašnjenja nekih važnijih direktorija i datoteka slijedi ispod:

- config – unutra se nalaze konfiguracije za `multer` i `mongoose`
- controllers – datoteke sa funkcijama koje se izvršavaju kao posljednji međuprogram u lancu pojedine putanje, odnosno ako prethodne nisu stvorile grešku
- dist – obzirom na korištenje TS-a, potrebno je prevesti cijeli kôd u JS, a upravo taj prevedeni kôd je smješten u ovaj direktorij i on se izvršava pri pokretanju aplikacije
- middleware – nalaze se izrađeni međuprogrami usmjerničkog levela, poput onih za autentikaciju i autorizaciju, validaciju i obrađivanje greški
- models – `mongoose` modeli i sheme za upravljanje kolekcijama
- node_modules – sadrži sve vanjske biblioteke koje vaša aplikacija koristi, kao i biblioteke o kojima ovise
- routes – definicija putanja pojedinog usmjernika i njihovog lanca međuprograma
- server.ts – glavna datoteka aplikacije koja se prva izvršava

3.2.2 Varijable okoline

Varijable okoline (engl. *environment variables*) su dinamičke vrijednosti koje se koriste za konfiguriranje različitih aspekata softverske aplikacije ili operativnog sustava. Ove varijable sadrže informacije kao što su putanje datoteka, korisnička imena, lozinke, tokeni, postavke okruženja i drugi parametri koji utječu na ponašanje aplikacije ili sustava.

Varijable okoline se često koriste kako bi se aplikacije prilagodile različitim okruženjima, kao što su razvoj, testiranje i produkcija. Na taj način, aplikacija može koristiti različite postavke ovisno o okruženju u kojem se izvodi, bez potrebe za mijenjanjem kôda.

U aplikaciji se koristi paket `dotenv`, koji služi za učitavanje varijabli okoline iz „.env“ datoteke u projektu. U ispisu 14 je prikazana ta datoteka u ovoj aplikaciji.

```
PORT=3000
JWT_SECRET=<skriveno zbog sigurnosti>
NODE_ENV=development
MONGO_URI=<skriveno zbog sigurnosti>
TMDB_API=<skriveno zbog sigurnosti>
```

Ispis 14: Varijable okoline poslužiteljskog dijela aplikacije

Kada se razvija bilo koji projekt, koristi se `git` za verzioniranje, što znači prijenos izvornog kôda na neku od platformi kao što je GitHub ili Bitbucket. Naravno, tako osjetljivi podaci ne smiju se prenositi, već se trebaju izolirati u posebnu datoteku koju će `git` ignorirati pri prijenosu. U kôdu se dobije vrijednost tih varijabli koristeći `process.env.<varijabla>`.

3.2.3 Međuprogrami

API koristi brojne međuprograme koji se dijele na međuprograme aplikacijskog levela (engl. *application-level*) i one usmjerničkog levela (engl. *router-level*), neki od njih su instalirani preko NPM-a, dok su ostali definirani samo za potrebe ove aplikacije.

```
dotenv.config()
connectDB()
const app: Express = express()
app.use(express.json())
app.use(morgan('dev'))
app.use(cors({ origin: 'http://localhost:9000' }))

app.use('/users', usersRouter)
app.use('/tv-shows', tvShowRouter)
app.use('/subtitles', subtitleRouter)
app.use('/announcements', announcementRouter)
app.use('/notifications', notificationRouter)

app.use(notFound)
app.use(errorHandler)

const port: string | number = process.env.PORT || 3000
app.listen(port, () => {
  console.log(`Server listening on port ${port}`)
})
```

Ispis 15: Postavljanje poslužitelja u „server.ts“ datoteci

U ispisu 15 je prikazan sadržaj „server.ts“ datoteke koja se prva izvršava pri pokretanju aplikacije. U njoj su definirani međuprogrami te se postavlja slušanje zahtjeva na željenom priključku. Naravno, poviše ovog dijela kôda su linije za uvoz poput `import dotenv from 'dotenv'`.

Počevši od prve linije, najprije se pomoću `dotenv.config()` učitavaju varijable okoline iz „.env“ datoteke u `process.env`. Zatim se stvara konekcija sa bazom podataka, što je uvjet za rad API-a. Nakon toga se dodaju međuprogrami `json` koji je prethodno objašnjen, `morgan` za automatsko ispisivanje dolazećih HTTP zahtjeva u konzolu i `cors` za zabranu zahtjeva koji ne dolaze iz navedenog izvora, a na kojem se nalazi korisnički dio aplikacije.

Ispod toga je definirano koji usmjernici će biti zaduženi za koje putanje. Razlog stvaranja više usmjernika je ponajprije organizacija kôda. Naravno, može biti slučaj da je zahtjev poslan na putanju za koju nije određen usmjernik. Za takav slučaj je dodan međuprogram ispod svih ostalih usmjernika bez navedene putanje i s referencom na funkciju koja će se pozvati. Ta funkcija se može vidjeti u ispisu 16.

```
const notFound = (req: Request, res: Response, next: NextFunction) => {
  const error = new Error(`Not found - ${req.originalUrl}`)
  res.status(404)
  next(error)
}
```

Ispis 16: Funkcija za obradu zahtjeva sa pogrešnom putanjom

Funkcija stvara `Error` objekt sa porukom koja će kasnije biti poslana korisniku. Postavlja se HTTP status 404 što označava da resurs nije pronađen, te se pomoću `next(error)` objekt prosljeđuje funkciji za obradu pogrešaka, o kojoj će biti više detalja u nastavku.

Na kraju lanca stavlja se međuprogram s referencom na funkcijom koja se poziva u slučaju greški. U ispisu 17 se može vidjeti kako ta funkcija izgleda.

```
const errorHandler = (err: any, req: Request, res: Response,
next: NextFunction) => {
```

```
  const statusCode = err.statusCode ||
                  err.response?.status ||
                  (res.statusCode === 200 ? 500 : res.statusCode)
  res.status(statusCode)
  res.json({
    message: err.message || 'Error occurred',
    stack: process.env.NODE_ENV === 'prod' ? null : err.stack,
  })
}
```

Ispis 17: Funkcija za obradu greški

Ono što je bitno napomenuti je da se može dogoditi da se ne postavi status odgovora prije pozivanja ove funkcije nakon greške. U tom slučaju potrebno je statusni kod promijeniti na 500. Također se vidi da funkcija najprije provjerava je li `error` objekt sadrži `statusCode`. Obzirom da `Error` klasa nema mogućnost postavljanja statusa uz poruku, za potrebe aplikacije napravljena je posebna klasa koja se nadograđuje na nju. U ispisu 18 se može vidjeti kôd iste.

```
export class CustomError extends Error {
  statusCode: number
  constructor(message: string, statusCode?: number) {
    super(message)
    this.statusCode = statusCode || 500
    this.name = this.constructor.name
    Error.captureStackTrace(this, this.constructor)
  }
}
```

Ispis 18: CustomError klasa koja proširuje Error klasu

Ovo omogućuje da se, na primjer, ako ne postoje traženi titlovi, funkciju za obradu grešaka može ručno pozvati jednostavnim `throw new CustomError('Titlovi nisu pronađeni', 404)`.

Ovo su bili međuprogrami aplikacijskog levela, a sada slijede međuprogrami usmjerničkog levela. Najprije će biti riječ o autentikacijskim i autorizacijskim. Naime, za te potrebe aplikacija koristi JWT. Svaki zahtjev za privatni resurs mora u zaglavlju

sadržavati JWT, čija se valjanost provjerava prije izvršavanja zahtjeva. Za te potrebe je napravljeno nekoliko funkcija, od kojih je jedna pokazana u ispisu 19.

```
const authenticate = asyncHandler(async (req: IAuthUserRequest, res,
next) => {
  if (
    !req.headers.authorization ||
    !req.headers.authorization.startsWith('Bearer')
  ) {
    throw new CustomError('Not authorized', 401)
  }

  const token = req.headers.authorization.split(' ')[1]
  let decoded: JwtPayload
  try {
    decoded = jwt.verify(
      token,
      process.env.JWT_SECRET as jwt.Secret
    ) as JwtPayload
  } catch (err: any) {
    throw new CustomError('Not authorized', 401)
  }

  req.user = (await User.findById(decoded._id).select('-password')) as
IUser
  next()
})
```

Ispis 19: Funkcija za provjeru valjanosti JWT-a

JWT se nalazi u zaglavlju HTTP zahtjeva pod imenom „authorization“. Zapisuje se u formatu `Bearer <JWT>`, tako da je potrebno zanemariti dio prije razmaka, a ono što ostane provjeriti pomoću `verify` metode iz `jsonwebtoken` paketa, u koju se, osim samog JWT-a, prosljeđuje i JWT tajna. Metoda `verify` vraća objekt s podacima iz JWT-a (korisničko ime i neka druga polja iz kolekcije korisnici). Ako je provjera uspješna, objekt s podacima korisnika se dodaje u zahtjev, tako da bude dostupan kroz `req.user`, a potom se poziva idući međuprogram u lancu. U suprotnom, stvara se greška sa statusom 401, što označava neovlašten pristup. Postoje još i dvije slične funkcije, jedna koja uz ovo još i

provjerava je li korisnik administrator (objekt ima svojstvo `isAdmin`), te druga koja samo dodaje objekt u zahtjev, bez stvaranja greške.

Budući da korisnici mogu slati zahtjeve sa proizvoljnim podacima pri dodavanju i ažuriranju resursa, potrebno je provjeriti valjanost tih podataka. U ispisu 21 je prikazana ta funkcija.

```
const validateBody = (validator: Joi.ObjectSchema<any>) =>
  (req: Request, res: Response, next: () => void) => {
    const { error } = validator.validate(req.body)
    if (error) throw new CustomError(error.details[0].message, 400)
    next()
  }
```

Ispis 21: Funkcija za provjeru valjanosti poslanih podataka

Struktura i pravila podataka za pojedini resurs definirana su koristeći paket `joi`. Primjer se može vidjeti u ispisu 22.

```
const announcementCreateValidator = Joi.object({
  tvShowId: Joi.number().positive().required(),
  text: Joi.string().min(10).max(500).trim().strict(true).required(),
})
```

Ispis 22: `joi` validacijska shema za dodavanje obavijesti

Paket `joi` omogućuje definiranje pravila i uvjeta za validaciju različitih tipova podataka, poput *stringova*, brojeva, objekata itd. Pozove se odgovarajuća validacijska shema i proslijedi podatke koje se želi provjeriti. Metoda `validate` provjerava unesene podatke prema definiranim pravilima i vraća objekt s „error“ svojstvom ako validacija nije uspješna, odnosno „value“ objektom ako su podaci uspješno prošli validaciju.

Paket `multer` koristi se kako bi se pojednostavnio proces prijenosa datoteka. Potrebno je postaviti ograničenja za datoteke te osnovnu konfiguraciju.

```
const fileSizeLimitMegabytes = 2
export const allowedExtensions = ['.srt', '.sub', '.ssa', '.ass', '.vtt']
export const allowedMimeTypes = [
```

```
  'application/x-subrip',
  'application/x-matroska',
  'application/vnd.ms-ssa',
  'application/vnd.nikse.subtitleeditor',
  'text/plain',
]
export const tempFolderPath = path.join('uploads', 'temp')
export const subtitlesFolderPath = path.join('uploads', 'subtitles')
```

Ispis 23: Definirana ograničenja i direktoriji za spremanje datoteka

U ispisu 23 prikazan je dio kôda koji definira ograničenja datoteka te direktorije za privremeno i stalno spremanje tih datoteka, odnosno titlova. Privremeni direktorij služi za spremanje datoteka kakve su prenesene, dok se prije spremanja u stalni direktorij te datoteke arhiviraju koristeći biblioteku paket `archiver`. Najvažnije ograničenje za veličinu datoteke (dva megabajta) jer bi u suprotnom korisnici mogli prenijeti datoteke i od više gigabajti te tako vrlo brzo popuniti dostupan prostor na disku.

```
const subtitleMulter: Multer = multer({
  storage,
  limits: {
    fileSize: fileSizeLimitMegabytes * 1024 * 1024,
  },
  fileFilter,
})
```

Ispis 24: Instanciranje `multer` objekta sa opcijama

U ispisu 24 prikazano je instanciranje `multera`. Svojstvo `storage` definira način spremanja prenesenih datoteka. U ovom slučaju izabrano je spremanje na disk.

```
const storage = multer.diskStorage({
  destination: (
    req: Request,
    file: Express.Multer.File,
    cb: (error: Error | null, destination: string) => void
  ) => {
    if (!fs.existsSync(tempFolderPath)) fs.mkdirSync(tempFolderPath)
    cb(null, tempFolderPath)
```

```
  },
  filename: (
    req: Request,
    file: Express.Multer.File,
    cb: (error: Error | null, filename: string) => void
  ) => {
    cb(null, uuidv4() + '.' + file.originalname.split('.').pop())
  },
})
```

Ispis 25: Konfiguracija načina spremanja datoteka

U ispisu 25 se može vidjeti točna konfiguracija za spremanje datoteka. Svojstvo `destination` određuje odredišni direktorij za spremanje datoteka; `existsSync` metoda daje informaciju postoji li direktorij, dok `mkdirSync` pravi novi direktorij. Svojstvo `filename` određuje način generiranja naziva nove datoteke. Konkretno, naziv nove datoteke koja će se spremati u privremeni direktorij je dobiveni jedinstveni identifikator (engl. *Universal Unique Identifier*, UUID) uz ekstenziju dobivenu iz originalnog naziva datoteke.

Svojstvo `fileFilter` kod instanciranja `multer` objekta kontrolira koje datoteke će se spremiti, a koje će biti ignorirane.

```
const fileFilter = async (
  req: Request,
  file: Express.Multer.File,
  cb: FileFilterCallback
) => {
  const extension = '.' + file.originalname.split('.').pop()
  if (allowedExtensions.includes(extension)) {
    cb(null, true)
  } else {
    cb(new CustomError('Unsupported files', 415))
  }
}
```

Ispis 26: Konfiguracija dopuštenih datoteka

Filtriranje funkcionira tako da se pri pronalasku barem jedne nedozvoljene datoteke, obustavlja i prijenos svih ostalih te stvara greška sa statusom 415, a koja označava nedozvoljeni tip medija. To se može i vidjeti u ispisu 26.

3.2.4 Usmjernici i putanje

Već je ranije spominjan direktorij sa putanjama, odnosno usmjernicima. U njemu se nalazi po jedna datoteka za svaki usmjernik. Koristeći `express.Router()` definira se novi usmjernik. Sve putanje su definirane na način prikazan u ispisu 27.

```
const noviUsmjernik = express.Router()
noviUsmjernik.get(<putanja>, prviMeđuprogram, drugiMeđuprogram)
noviUsmjernik.post(<putanja>, prviMeđuprogram, drugiMeđuprogram)
noviUsmjernik.put(<putanja>, prviMeđuprogram, drugiMeđuprogram)
noviUsmjernik.patch(<putanja>, prviMeđuprogram, drugiMeđuprogram)
noviUsmjernik.delete(<putanja>, prviMeđuprogram, drugiMeđuprogram)
```

Ispis 27: Način definiranja usmjernika i njegovih putanji

Najprije je važno napomenuti da je pri vrhu svake datoteke potrebno uvesti korištene biblioteke, funkcije i sučelja. Lanac međuprograma svake putanje može biti teoretski beskonačno dug, u primjeru sa ispisa 27 ih je samo dva za svaku putanju. Obzirom da su neke putanje zaštićene ili je potrebno provjeriti podatke, funkcija koja obrađuje zahtjev se uvijek postavlja na kraj međuprogramskog lanca, i izvršava se jedino ako do tada nije stvorena greška.

```
usersRouter.delete(
  '/mark-unwatched/:episodeId',
  authenticate,
  removeEpisodeFromWatched
)
```

Ispis 28: Primjer putanje korisničkog usmjernika

Primjer iz ispisa 28 pokazuje putanju za brisanje pogledane epizode. Prije same funkcije za obradu zahtjeva `removeEpisodeFromWatched`, u međuprogramsku lancu nalazi se funkcija `authenticate` koja provjerava je li u zaglavlju postoji ispravan JWT.

Dinamičke putanje su moguće pomoću dvotočke, kao na primjer `:episodeId`. Na ovaj način će u `req.params.episodeId` biti dostupno ono što se u putanji zahtjevu nalazi na mjestu `:episodeId`. Naravno, tip varijable će biti `string` (niz znakova), a ne broj. Moguće je svakako naknadno obaviti pretvorbu u broj ako je potrebno.

Svaka funkcija u međuprogramskom lancu prima objekt zahtjeva (predstavlja HTTP zahtjev koji dolazi od klijenta prema API-u) i odgovora (predstavlja odgovor koji se šalje s API-a klijentu kao odgovor na HTTP zahtjev), te funkciju `next` koja služi za pozivanje sljedeće funkcije u međuprogramskom lancu.

3.2.5 Obrada zahtjeva

Zadnja funkcija u međuprogramskom lancu je ona koja sadrži logiku, obrađuje sam zahtjev i šalje odgovor. Na primjer, ako je zatražen neki dokument, ona izvlači podatke iz baze podataka i šalje odgovor s podacima u JSON-u i statusom 200 do 299 (inače poznati kao statusi uspjeha, najčešći je 200, što znači ok). Odnosno, to je u slučaju da je dokument pronađen, u suprotnom se stvara greška i status je 400-499 (inače poznati kao greške s korisničke strane), u ovom slučaju bi najprikladniji bio 404. Ove funkcije smještene su u direktorij „controllers“. U ispisu 29 prikazana je funkcija koja obrađuje zahtjev ažuriranja objava.

```
const updateAnnouncement = asyncHandler(async (req: Request, res:
Response) => {
  const announcementId = req.params.announcementId
  const announcement: IAnnouncementForm = req.body
  const existingAnnouncement = await
Announcement.findById(announcementId)

  if (!existingAnnouncement)
    throw new CustomError('Announcement not found', 404)

  existingAnnouncement.text = announcement.text
  await existingAnnouncement.save()

  res.status(201).json(existingAnnouncement)
})
```

Ispis 29: Obrada zahtjeva za ažuriranje objave

Svaka funkcija je asinkrona jer svaki zahtjev na bazu podataka ili datotečni sustav zahtjeva određeno vrijeme za izvršavanje. U tom slučaju nije poželjno blokirati izvršavanje drugih operacija neovisnih o tome. `async/await` omogućuje pisanje asinkronog kôda kao da je sinkroni, što poboljšava čitljivost.

Kako bi hvatanje bačenih greški bilo olakšano, koristi se paket `async-express-handler`, odnosno točnije njegova funkcija `asyncHandler` koja hvata sve bačene greške i prosljeđuje ih u ranije spomenutu funkciju za obradu greški.

U funkciji sa ispisa 29, najprije se dohvaća resurs iz baze podataka. Zatim se provjerava je li postoji resurs i ako nije, stvara se greška sa statusom 404. Ako postoji, ažurira se polje `text` i spremaju se promjene. Na kraju se postavlja status odgovora na 201 (označava da je resurs stvoren) te konačno šalje odgovor koji sadrži izmijenjene podatke.

Podatke je moguće ubaciti i u samu putanju, na primjer „/<putanja>?naziv=vrijednost&hello=world“. Tako se može ubaciti teoretski beskonačno podataka, ali preporuka je to koristiti samo za podatke koje nisu osjetljivi. Slati podatke za autentikaciju na taj način ni u kojem slučaju nije prihvatljivo, jer su izloženi hakerskom napadu. U Express.js takvi podaci se mogu dobiti kroz `req.query`. U navedenom primjeru bi `req.query.hello` imao vrijednost „world“.

3.2.6 Operacije nad bazom podataka – Mongoose

Obzirom da je ovo dosta opširan paket, ovdje će biti objašnjene samo neke od funkcionalnosti korištenih u samoj aplikaciji, a one koje nisu spomenute je uvijek moguće pronaći u službenoj dokumentaciji [4]. Postoji više načina korištenja pojedinih metoda, ali isključivo je opisan način na koji su korišteni u aplikaciji. Najprije je potrebno uvesti `mongoose` model na kojem će se obavljati operacije. Sve metode koje pristupaju bazi podataka su asinkrone.

Metoda `findById` traži dokument u kolekciji prema zadanom ID-u. Vraća dokument ako se pronađe ili `null` ako dokument s tim ID-om ne postoji. Na nju se može lančano dodati druge metode poput `select`, `populate`, `sort` i `lean`. Metoda `select` prima

niz znakova i omogućuje definiranje polja koja će biti sadržana u vraćenom objektu koji predstavlja traženi resurs. Po zadanom će sva polja biti uključena. Moguće je odabrati konkretna polja ili neka polja zanemariti stavljajući „-“ ispred imena polja. Metoda `populate` omogućuje dohvaćanje referenciranog dokumenta pomoću „_id“ polja. Metoda prima niz znakova sa imenom polja kojeg se želi popuniti ili objekt u formatu `{ path: 'imePolja', select: ..., populate: ..., ... }` ako je potrebno više opcija. Metoda `sort` prima objekt u formatu `{ imePolja: 1 ili -1, ... }`, a omogućuje razvrstavanje dokumenata po određenom polju ili poljima, gdje 1 označuje uzlazni, a -1 silazni redoslijed. Metoda `lean` omogućuje direktno dohvaćanje podataka iz baze podataka kao običnih JS objekata, umjesto da ih „omota” u složene `mongoose` modele. Ovo poboljšava performanse i smanjuje potrošnju memorije kada se radi s velikim količinama podataka, koristi se kada je potrebno samo čitanje podataka iz baze. Primjer jednog upita sa `findById` korištenog u aplikaciji može se vidjeti u ispisu 30.

```
const user = await User.findById(req.params.userId).select(
    '-password -watchlist'
  )
```

Ispis 30: Primjer korištenja metode `findById`

Metoda `find` pretražuje kolekciju i vraća sve dokumente koji odgovaraju zadanim uvjetima. Metoda prima objekt formata `{ imePolja: uvjet }`, s tim da uvjet može biti još jedan objekt ako se ne radi o točnoj vrijednosti. Vraća niz dokumenata ili prazan niz. Također podržava ulančavanje metoda. Primjer korištenja u ispisu 31.

```
const subtitles = await Subtitle.find({
      userId,
      episodeId,
      subtitleRequestId: null,
      filePath: { $ne: null },
    })
      .select('_id language release frameRate createdAt updatedAt')
      .sort({ updatedAt: -1 })
```

Ispis 31: Primjer korištenja metode `find`

U primjeru sa ispisa 31 `$ne` ima značenje „nije jednako“, odnosno traže se dokumenti u kojima `filePath` nije `null`.

Metoda `findOne` je slična metodi `find`, ali vraća samo prvi dokument koji zadovoljava uvjete ili `null` ako ne postoji odgovarajući dokument. Također podržava ulančavanje metoda.

Metoda `deleteOne` briše prvi dokument koji zadovoljava zadane uvjete. Može se koristiti direktno na vraćenom dokumentu, u tom slučaju će taj dokument biti obrisan. Vraća objekt s informacijom o brisanju

Metoda `aggregate` omogućuje izradu naprednih upita koji se koriste za obradu, transformaciju i analizu podataka unutar kolekcija u MongoDB bazi. Ova metoda koristi koncept agregacijskog cjevovoda (engl. *aggregation pipeline*), gdje se serija operacija primjenjuje na podatke kako bi se generirao krajnji rezultat. Najbolje je objasniti na konkretnom primjeru iz aplikacije, koji se može vidjeti u ispisu 32.

```
const subtitles = await Subtitle.aggregate([
      {
        $match: { episodeId },
      },
      {
        $lookup: {
          from: 'users',
          localField: 'userId',
          foreignField: '_id',
          as: 'user',
        },
      },
      {
        $addFields: {
          isOwner: {
            $eq: ['$userId', userId],
          },
          thankedByCount: { $size: '$thankedBy' },
          isThankedByUser: { $in: [userId, '$thankedBy'] },
        },
      },
```

```
    {
      $project: {
        'user.email': 0,
        'user.password': 0,
        'user.updatedAt': 0,
        'user.createdAt': 0,
        thankedBy: 0,
      },
    },
    {
      $sort: {
        isOwner: -1,
        updatedAt: -1,
      },
    },
  ])
```

Ispis 32: Agregacijski cjevovod koji vraća sve titlove za epizodu

Korak `$match` filtrira dokumente tako da odabere samo one koji imaju odgovarajući „episodeId“. Korak `$lookup` izvodi operaciju lijevog vanjskog spajanja (engl. *left outer join*) između kolekcija „subtitles“ i „users“. Rezultat spajanja sprema se u polje "user" kao objekt. Lijevo vanjsko spajanje znači da ako je polje „userId“ `null`, taj će dokument i dalje bit sadržan u krajnjem rezultatu. Korak `$addFields` dodaje nova polja dokumentima, a svako polje izračunava se na temelju podataka iz dokumenta i/ili spajanih podataka. Korak `$project` oblikuje izlaz tako da se iz dokumenta izdvajaju samo odabrana polja. Korak `$sort` razvrstava dokumente po definiranim kriterijima.

Metoda `create` stvara novi dokument u kolekciji prema zadanom objektu, a vraća stvoreni dokument. Metoda `save` ažurira postojeći dokument. Ako je dokument stvoren putem modela, isti se može ažurirati pozivanjem `save`. Metoda `updateOne` ažurira prvi dokument koji zadovoljava zadane uvjete. Mogu se ažurirati pojedinačna polja ili zamijeniti cijeli dokument.

Iako `mongoose` podržava straničenje metodama `skip` i `limit`, kako bi se ubrzao razvoj aplikacije korišteni su paketi `mongoose-paginate-v2` i `mongoose-aggregate-`

paginate-v2. Obzirom da mongoose-paginate-v2 ne podržava rad s agregatnim cjevovodima, potrebna su oba paketa.

```
const options = {
      page: pageNumber,
      limit: pageSize,
      sort: { updatedAt: -1 },
      lean: true,
    }

const result = await Announcement.paginate({ tvShowId }, options)
```

Ispis 33: Straničenje pomoću metode paginate

Kao što je vidljivo u ispisu 33, straničenje se vrši pozivanjem metode paginate koja prima objekt za filtriranje podataka i objekt s opcijama. U opcijama je moguće definirati sve one spomenute mongoose metode. Primjer dobivenog JSON-a kao odgovora je u ispisu 34.

```
{
    "docs": [...],
    "totalDocs": 22,
    "limit": 10,
    "totalPages": 3,
    "page": 1,
    "pagingCounter": 1,
    "hasPrevPage": false,
    "hasNextPage": true,
    "prevPage": null,
    "nextPage": 2
}
```

Ispis 34: Primjer dobivenog odgovora nakon straničenja metodom paginate

Na isti način funkcionira i metoda aggregatePaginate iz paketa mongoose-aggregate-paginate-v2, samo što ona prima agregacijski cjevovod kao prvi parametar.

3.2.7 Arhiviranje i spremanje datoteka

Osim već objašnjenog `multera`, koristi se i paket `mmmagic` koji, pregledavajući sadržaj datoteke, daje informaciju o tipu datoteke. Također se koristi i paket `archiver`, koji nudi funkcionalnosti za arhiviranje datoteka. Svrha arhiviranja u ovom slučaju je čisto mogućnost referenciranja više datoteka s jednom datotečnom putanjom. Na taj način se izbjegava nepotrebna kompleksnost baze podataka.

U aplikaciji postoje dvije funkcije koje se tiču ovoga, jedna koja se bavi provjerom datoteka i druga koja se bavi arhiviranjem i konačnim spremanjem na lokalni disk.

```
const ensureAllowedMimeTypeForFiles = (files: Express.Multer.File[]) => {
  const magic = new Magic(MAGIC_MIME_TYPE)
  for (const file of files) {
    const filePath = path.join(tempFolderPath, file.filename)
    magic.detectFile(filePath, function (err, result) {
      if (err) throw new CustomError('Unsupported files', 415)
      if (!allowedMimeTypes.includes(result as string))
        throw new CustomError('Unsupported files', 415)
    })
  }
}
```

Ispis 35: Provjera sadržaja datoteka pomoću paketa `mmmagic`

U ispisu 35 je funkcija koja provjerava jesu li sve datoteka dozvoljenog tipa, a oslanja se na metodu `detectFile` iz paketa `mmmagic`. Metoda `detectFile` prima dva argumenta, datotečnu putanju i funkciju povratnog poziva (engl. *callback function*) kojoj se prosljeđuju objekt greške i sam tip datoteke. Funkcija povratnog poziva je funkcija koja se prosljeđuje kao argument drugoj funkciji i izvršava nakon što ta druga funkcija završi izvođenje. U slučaju nedozvoljenog tipa datoteke, stvara se greška.

```
const uploadAndZipFiles = (files: Express.Multer.File[]) => {
  const zipFileName = uuidv4() + '.zip'
  const zipFilePath = path.join(subtitlesFolderPath, zipFileName)
  const output = fs.createWriteStream(zipFilePath)
  const archive = archiver('zip', {
    zlib: { level: 9 },
```

```
  })
  archive.pipe(output)
  for (const file of files) {
    const filePath = path.join(tempFolderPath, file.filename)
    archive.file(filePath, { name: file.originalname })
  }
  archive.finalize()
  return zipFileName
}
```

Ispis 36: Arhiviranje i spremanje datoteka na disk

Nakon provjere tipa datoteka obavlja se arhiviranje i spremanje. Za to služi funkcija prikazana u ispisu 36, a koja je objašnjena u koracima ispod:

1. Kreira se putanja za zip arhivu sa jedinstvenim imenom koristeći metodu `uuidv4` iz paketa `uuid`.
2. Metodom `fs.createWriteStream` se stvara tok za pisanje (`output`) koji će se koristiti za spremanje zip datoteke.
3. Stvara se instanca `archiver` objekta s postavkama za kompresiju.
4. Arhiva se povezuje s tokom za pisanje podataka.
5. Petljom se prolazi kroz svaku datoteku u nizu:
    - Stvara se putanja do trenutne datoteke koristeći privremeni direktorij i naziv datoteke.
    - Datoteka se dodaje u arhivu s odgovarajućim imenom (`originalname`).
6. Na kraju, arhiva se zatvara za pisanje metodom `finalize` i vraća se naziv stvorene zip datoteke.

3.2.8 Generiranje JWT-a

U ispisu 37 je prikazana funkcija za generiranje JWT-a. Ona koristi metodu `sign` iz paketa `jwt` kako bi generirala JWT koji će vrijediti onoliko vremena koliko je definirano u polju „expiresIn“, odnosno u ovom slučaju jedan dan.

```
const generateToken = (user: IUser) => {
  return jwt.sign(
    {
```

```
      _id: user._id,
      username: user.username,
      email: user.email,
      isAdmin: user.isAdmin,
      darkMode: user.darkMode,
    },
    process.env.JWT_SECRET as jwt.Secret,
    {
      expiresIn: '1d',
    }
  )
}
```

Ispis 37: Funkcija za generiranje JWT-a

JWT se sastoji od 3 dijela, zaglavlja (algoritam i tip tokena), podataka i potpisa koji služi za verifikaciju da je JWT generiran korištenjem ispravne tajne ključa.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJfaWQiOiI2NDk3YTJmNjI5ODZiMjk3Y2Ez
OWNiOTciLCJ1c2VybmFtZSI6ImphbmVkb2UiLCJlbWFpbCI6ImphbmVAZXhhbXBsZS5jb20iL
CJpc0FkbWluIjpmYWxzZSwiZGFya01vZGUiOnRydWUsImlhdCI6MTY5MTk1NDIxMCwiZXhwIj
oxNjkyMDQwNjEwfQ.FjIvP1lwrGjXhTE4QlMokq6gr0HtHPvqz6WBWg3tC5w
```

Ispis 38: Primjer generiranog JWT-a

U ispisu 38 je prikazan jedan generirani JWT prilikom prijave, a u ispisu 39 kako izgleda taj isti JWT dekodiran.

```
{
  "alg": "HS256",
  "typ": "JWT"
}
{
  "_id": "6497a2f62986b297ca39cb97",
  "username": "janedoe",
  [...]
  "iat": 1691954210,
  "exp": 1692040610
}
```

```
<JWT_SECRET>
```

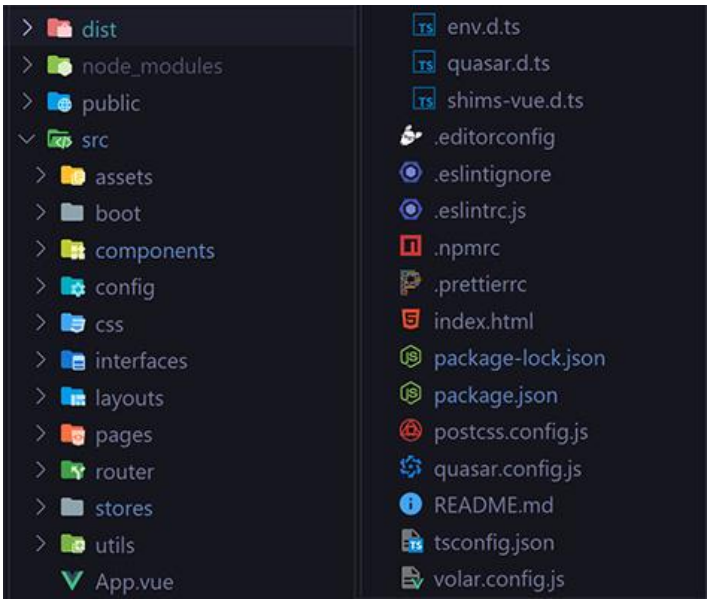
Ispis 39: Primjer dekodiranog JWT-a

3.3 Korisničko sučelje

Novi Quasar projekt može se stvoriti naredbom `npm init quasar`. Prilikom stvaranja projekta, u konzoli treba odabrati postavke, poput općih informacija, stila Vue.js komponenti, upravitelja stanja aplikacije, CSS predprocesora, TS ili JS, alata za izgradnju aplikacije (Vite ili Webpack) itd. Preporuka je i globalno instalirati sučelje naredbene linije (engl. *Command Line Interface*; CLI) za Quasar naredbom `npm i -g @quasar/cli`, koji nudi brojne mogućnosti. Najpotrebnija je definitivno mogućnost pokretanja aplikacije na razvojnom poslužitelju naredbom `quasar dev`. Još neke naredbe koje su često korištene pri izradi aplikacije su `quasar new component <naziv>` za stvaranje komponente i `quasar new page <naziv>` za stvaranje stranica. A tu je još i naredba `quasar build`, koja služi za izgradnju aplikacije za produkciju koristeći Vite ili Webpack. U ovom slučaju je odabran Vite jer nudi brži razvoj. Webpack je nešto stariji, ali ima više mogućnosti konfiguracije i veću zajednicu.

3.3.1 Struktura direktorija

Budući da se, kao i u poslužiteljskom dijelu, koristi NPM i TS, postoje neke iste datoteke i direktoriji koji se neće posebno objašnjavati.



Slika 4: Struktura direktorija korisničkog dijela

Na slici 4 je prikazana struktura direktorija poslužiteljskog dijela, a ispod su objašnjeni neki važniji direktoriji i datoteke.

- public – statičke datoteke (ikone, CSS, JS itd.), dostupne preko URL-a
- src – izvorni kôd projekta
- src/assets – statičke datoteke koje su dio aplikacije i prolaze kroz izgradnju
- src/boot – inicijalizacijske datoteke koje se izvrše prije nego što se aplikacija pokrene
- src/components – stvorene Vue.js komponente
- src/layouts – stvoreni Quasar rasporedi korisničkog sučelja
- src/pages – stvorene Vue.js stranice
- src/router – konfiguracija usmjernika i putanja
- src/stores – Pinia konfiguracija i skladišta (engl. *store*)
- App.vue – korijen hijerarhije komponenti, služi kao ulazna točka za cijelu aplikaciju i sadrži ukupan raspored i strukturu

Nakon svega, nužno je napomenuti da pri razvoju aplikacije većinu ovih datoteka i direktorija nije potrebno mijenjati. Direktoriji koji se najviše mijenjaju prilikom izrade aplikacija su „src/pages“, „src/components“, i eventualno „src/layouts“.

3.3.2 Usmjernik i putanje

Vue Router je službeni usmjernik za Vue.js aplikacije koji omogućuje stvaranje dinamičnih i statičkih putanja, odnosno da različite dijelove aplikacije prikaže korisniku na temelju URL-a, omogućujući time korisnicima navigaciju između različitih stranica ili komponenti aplikacije bez potrebe za ponovnim učitavanjem stranice.

Vue Router je potrebno posebno instalirati koristeći naredbu `npm i vue-router` i zatim konfigurirati u projektu. Međutim, obzirom da je korišten razvojni okvir Quasar, to je napravljeno unaprijed i dostupno je u direktoriju „router“. U njemu se nalaze dvije datoteke, „index.ts“, u kojoj se konfigurira Vue Router i stvara povijest putanja, te „router.ts“ u kojem se nalaze same putanje u obliku JS objekta. Postoje dva načina

stvaranja povijesti, *hash* (putanja sadrži znak „#“ ispred) i HTML5 (putanja nema znak „#“ ispred). Preporučeni način je HTML5, zbog negativnog utjecaja *hash* načina na optimizaciju za tražilice (engl. *Search Engine Optimization*; SEO), i upravo je taj način korišten u aplikaciji. Putanje u „router.ts“ su definirane kao u primjeru iz ispisa 40.

```
const routes: RouteRecordRaw[] = [
  {
    path: '/',
    component: MainLayout,
    children: [
      { path: '', component: IndexPage },
      {
        path: '/login',
        component: LogInPage,
        beforeEnter: (to, from, next): any => {
          const auth = useAuthStore()
          if (auth.isLoggedIn) {
            next('/')
          }
          next()
        },
      },
  ...
  }
  // uvijek mora biti zadnji
  {
    path: '/:catchAll(.*)*',
    component: ErrorNotFound,
  }
]
```

Ispis 40: Definiranje putanji Vue Routera

Svaki objekt sadrži samu putanju i komponentu koja će se prikazati kada se pristupi toj putanji. Moguće je i definirati potputanje, čija se putanja nastavlja na korijensku putanju. U Quasaru, „MainLayout“ je komponenta koja je korijen svih ostalih putanji, a sadrži onaj dio sučelja koji je nužan da bude prikazan na svakoj putanji, kao što su na primjer izbornici. Naravno, sve komponente koje se ovdje koriste je potrebno uvesti, a one

bi se trebale po pravilu nalaziti u „pages“ direktoriju. Iako, po sintaksi, ne postoji razlika između tih komponenti i onih u „components“ direktoriju, već su samo razdvojeni po namjeni.

Također jedno od korisnih svojstava pojedinog objekta putanji koje je korišteno u aplikaciji je „beforeEnter“, a koja daje mogućnost da se odradi nekakva provjera prije pristupa korisnika određenoj putanji. Tu se definira funkcija kojoj su proslijeđena tri argumenta, „to“ (putanja kojoj se želi pristupiti), „from“ (zadnja putanja kojoj se pristupilo“, te funkcija „next“ koja preusmjerava korisnika na putanju koju je primila kao argument ili na putanju „to“ ako je pozvana bez argumenta.

Dinamičke putanje je moguće definirati kao `path: ':tvShowId'`, nakon čega je `tvShowId` dostupan u navedenoj komponenti pomoću `useRoute().params.tvShowId`. Metoda `useRoute` se uvozi iz biblioteke „vue-router“. Na samom kraju se uvijek nalazi putanja koja hvata sve druge putanje koje nisu definirane poviše.

3.3.3 Komponente

Komponente su temeljne gradivne jedinice Vue.js aplikacije. One omogućuju modularnost, ponovno korištenje kôda i lakše upravljanje složenijim korisničkim sučeljima. Svaka komponenta se sastoji od samo jedne datoteke u kojoj se nalazi HTML, CSS i JS, odnosno u ovom slučaju TS. Primjer jedne komponente može se vidjeti u ispisu 41.

```
<template>
  // <template> može imati samo jedno dijete
  <div>
    <h1>Pozdrav iz MojaKomponenta!</h1>
  </div>
  // nije dopušteno, greška
  <div>...</div>
</template>

<script>
// ili samo export default {}, defineComponent je tu zbog tipizacije
export default defineComponent({
  name: 'MojaKomponenta',
```

```
  ...
})
</script>

<style>
/* Stilovi specifični za ovu komponentu */
</style>
```

Ispis 41: Primjer Vue.js komponente

Postoje dva pristupa pisanja komponenti, Options API (tradicionalno) i Composition API (moderno). Options API je zastarjeli pristup koji je još ostao dostupan zbog podrške i neće biti ovdje objašnjavan, dok primjer Composition API-a se može vidjeti u ispisu 42.

```
<template>
  <div>
    <button @click="increaseCount">Increase Count</button>
    <p>{{ count }}</p>
  </div>
</template>

<script setup>
import { ref, computed, onMounted, watchEffect } from 'vue';

export default defineComponent({
  const count = ref(0);
  const increaseCount = () => {
    count.value++;
  }
  return {
    count,
    increaseCount,
  }
})
</script>
```

Ispis 42: Primjer komponente napisane u Composition API

Quasar donosi mnoštvo gotovih komponenti i svaka od njih je detaljno dokumentirana na službenim stranicama [1] sa primjerima. Na primjer, veoma zanimljive su komponente koje se tiču formi. Postoje komponente za svaki tip unosa, tekst, radijski botun (engl. *radio button*), potvrdni okvir (engl. *checkbox*) itd. U ispisu 43 je prikazana forma za prijavu.

```
<q-form @submit="onLoginSubmit">
  <q-input
    v-model="username"
    type="text"
    label="Username"
    :rules="required"  />
  <q-input
    v-model="password"
    type="password"
    label="Password"
    :rules="required"  />
  <q-btn
    type="submit"
    color="primary"
    label="Log in"
    class="q-mt-md"
    :loading="auth.isLoading"  />
</q-form>
```

Ispis 43: Forma za prijavu koja koristi Quasar komponente

Sve Quasar komponente je moguće prilagoditi upotrebom svojstava (enl. *props*), direktiva (engl. *directives*) i događaja (engl. *events*), a koji su detaljno objašnjeni u idućim poglavljima pa će kôd iz ispisa 55 biti tada jasniji.

3.3.4 Reaktivnost

Reaktivnost je osnovni koncept u Vue.js koji omogućuje automatsko praćenje promjena podataka i dinamičko ažuriranje korisničkog sučelja kad god se ti podaci mijenjaju. Ako se svojstvo označi kao reaktivno, Vue prati promjene na tom svojstvu i automatski ažurira korisničko sučelje. U Composition API-u, reaktivnost se postiže pomoću funkcija kao što su `ref`, `reactive`, `watch`, i druge. Ispod je osnovni pregled:

- `ref`: Kreiranje reaktivnih varijabli. Vraća objekt s „value“ svojstvom koje sadrži trenutnu vrijednost varijable. Promjena vrijednosti će automatski osvježiti korisničko sučelje. U HTML-u se izostavlja „.value“.
- `reactive`: Stvaranje reaktivnih objekata. Svojstva objekta postaju reaktivna, što znači da će promjene tih svojstava automatski osvježiti korisničko sučelje.
- `watch`: Promatranje promjena reaktivnih varijabli i izvođenje odgovarajuće logike kad se te promjene dogode.

U ispisu 44 se mogu vidjeti primjeri korištenja ovih funkcija u aplikaciji.

```
// primjer 1: ref
const isLoading: Ref<boolean> = ref(false)
// primjer 2: reactive
const episodeForDialog = reactive<ITVShowEpisodeForDialog>({
      details: null,
      tvShowId: null,
      justAddedSubtitleRequestId: null,
      justAddedSubtitleId: null,
    })
// primjer 3: watch
watch(tab, async (newValue) => {
      await loadReports(newValue)
})
```

Ispis 44: Primjeri korištenja funkcija za reaktivnost

3.3.5 Svojstva

Svojstva (engl. *props*) u Vue.js-u služe za jednosmjerno prenošenja podataka od nadređene komponente (roditelja) prema ugniježđenim komponentama (djeci). To omogućuje komunikaciju i dijeljenje podataka između komponenti. Svojstva i njihovi tipovi se definiraju kao na ispisu 45.

```
export default defineComponent({
  props: {
    tvShowId: {
      type: Number,
      required: true,
```

```
    },
    announcement: {
      type: Object as PropType<IAnnouncement | null>,
    },
  },
  setup(props, ...) {...},
  ...
}
```

Ispis 45: Primjer definiranja svojstava u Vue.js komponenti

Svojstva mogu bit i neobavezna, u tom slučaju se treba provjeriti vrijednost prije korištenja. Njihova vrijednost se može dobiti tako da se u `setup` opciji doda „props“ kao argument. Nakon toga će „tvShowId“ biti dostupan u `setup` opciji koristeći `props.tvShowId`. Ono što je često korišteno kroz razvoj ove aplikacije je pretvorba svojstava u `ref` uz pomoć funkcije `toRefs`, na primjer `const { announcement, tvShowId } = toRefs(props)`. Svojstva se prosljeđuju primjerice `<announcement-form :tvShowId="123" />`.

3.3.6 Događaji

Događaji (engl. *events*) omogućuju komponentama slanje podataka ili signala roditeljskoj komponenti. Događaji se tipično koriste za slanje informacija o korisničkim akcijama, kao što su klikovi na botun, unos teksta i slično. Postoje dva dijela ove funkcionalnosti, emitiranje događaja i osluškivanje tog događaja.

Osluškivanje događaja moguće je korištenjem `v-on` direktive, odnosno primjerice `v-on:submit= "<funkcija>"`, što se skraćeno može napisati kao `@submit= "<funkcija>"`. U funkciju je automatski proslijeđen nativni JS „događaj“ koji se može pristupiti sa `$event`. Funkcija može biti napisana u liniji, a može se i pozvati neku funkciju iz komponente. Naziv događaja se može ulančavati sa modifikatorima poput:

- `stop`: Zaustavlja daljnje širenje događaja. To je korisno kada se želi spriječiti širenje događaja do roditeljskih komponenti ili drugih elemenata.
- `prevent`: Sprječava uobičajene radnje povezane s događajem, kao na primjer slanje forme za `submit` događaje.

Opcija `setup` u komponenti prima dva argumenta, jedan je već rečeni `props`, a drugi kontekstualni objekt, koji omogućuje pristup funkcijama koje se često korištene u komponenti. Jedna takva funkcija je `emit`, koja služi za emitiranje vlastitih događaja. Prima naziv događaja i opcionalno podatke koji će se poslati do roditelja. Primjer korištenja je u ispisu 46.

```
setup(props, { emit }) {
      cont onSubmitted = () => {
            emit('saved', savedData)
      }
}
```

Ispis 46: Primjer emitiranja vlastitih događaja

Kako bi taj događaj bio uhvaćen kad se dogodi, potrebno ga je osluškivati pri definiciji komponente komponenta, na primjer `<q-form @saved="onSaved" />`.

3.3.7 Udice životnog ciklusa

Udice životnog ciklusa (engl. *lifecycle hooks*) su specijalne metode koje Vue poziva u različitim fazama životnog ciklusa komponente. One omogućuju izvršavanje određene akcije u ključnim trenucima životnog ciklusa komponente. Ove metode su korisne za izvršavanje inicijalizacije, ažuriranja, i slično. Primjer korištenja ovakvih udica je u ispisu 59.

```
setup() {
    onMounted(() => {
      console.log('Montirano');
    });
}
```

Ispis 47: Udice životnog ciklusa Vuea

Naravno, postoji ih više nego je vidljivo u ispisu 47, međutim `onMounted` je jedina udica korištena u aplikaciji za potrebe inicijalnog dohvaćanje podataka iz baze.

3.3.8 Direktive

Vue direktive su posebni atributi koji se koriste u HTML-u kako bi se pružila dodatna funkcionalnost elementima ili kako bi se kontroliralo njihovo ponašanje. Direktive se identificiraju po prefiksu „v-“ na atributu HTML elementa. Evo nekoliko često korištenih Vue direktiva:

- `v-if`, `v-else-if`, `v-else`: Omogućuju uvjetno prikazivanje elemenata u zavisnosti od ispunjenosti uvjeta. Na primjer `v-if="isLoading"`, `gdje je isLoading varijabla tipa Boolean.`
- `v-for`: Omogućuje iteraciju kroz niz podataka i stvaranje više elemenata na temelju tih podataka. Na primjer `v-for="user in users.docs"`.
- `v-show`: Slično kao `v-if`, ali koristi CSS umjesto da potpuni ukloni komponentu.
- `v-model`: Omogućuje dvosmjernu vezu između podataka u komponenti i elementa za unos na korisničkom sučelju.
- `v-bind`: Dinamičko povezivanje atributa HTML elementa sa podacima iz Vue komponente, omogućujući tako reaktivno ažuriranje sučelja.

3.3.9 Pinia – upravitelj stanja aplikacije

Paket `pinia` služi za upravljanje stanjem (engl. *state management*) u Vue.js aplikacijama, pruža jednostavan i efikasan način za organiziranje, pristup i manipulaciju globalnim podacima u aplikaciji. U Vue.js aplikacijama, stanje se može podijeliti između komponenti preko svojstava ili može biti i lokalno unutar pojedinih komponenti. Međutim, kako aplikacija postaje veća i složenija, sve je veća i potreba za efikasnijim i centraliziranim načinom upravljanja stanjem. `pinia` pohranjuje globalno stanje aplikacije na jednom mjestu, a pojedine komponente dijele to stanje.

Ako je paket `pinia` odabran kao upravitelj stanja u inicijalizaciji Quasar projekta, unaprijed je instaliran. Postoji direktorij „stores“ u kojem se nalazi „index.ts“ datoteka koja stvara `pinia` instancu pomoću `createPinia` funkcije. Tu se još nalaze i skladišta. Skladište je zapravo način odvajanja podataka u više nezavisnih dijelova, što poboljšava performanse jer ne treba uvijek vraćati sve podatke. U ovoj aplikaciji je zapravo korišteno samo jedno skladište, ono sa podacima prijavljenog korisnika.

```
export const useAuthStore = defineStore({
  id: 'auth',
  state: (): AuthState => ({
    isLoading: false,
    darkMode: false,
    isSettingDarkMode: false,
    userInfo: null,
  }),
  getters: {
    isLoggedIn: (state) => !!state.userInfo,
    isAdmin: (state) => state.userInfo?.isAdmin || false,
  },
  actions: {...},
})
```

Ispis 48: Definicija skladišta za spremanje korisničkih podataka

U ispisu 48 se može vidjeti kako izgleda skladište sa korisničkim podacima. Svojstvo „id“ služi za pristup skladištu u komponentama preko `useStore(<id>)` funkcije, međutim u aplikaciji se za to koristi `useAuthStore()`, što je otpornije na greške u pisanju. Svojstvo „state“ služi za definiranje podataka i njihovih zadanih vrijednosti. Svojstvo „getters“ služi za definiranje funkcija koje ne mijenjaju stanje, odnosno podatke. Svojstvo „actions“ služi za definiranje složene funkcija koje mijenjaju stanje, iako se stanje može i direktno izmijeniti. U idućim ispisima će biti opisane funkcije koje se nalaze u „actions“.

```
loadUserInfo(userInfo: any, isFromLogin = false): void {
  this.userInfo = {
    _id: userInfo._id,
    username: userInfo.username,
    email: userInfo.email,
    isAdmin: userInfo.isAdmin,
  }
  userInfo.token && localStorage.setItem('token', userInfo.token)
  if (isFromLogin)
    localStorage.setItem('darkMode', userInfo.darkMode.toString())
  this.initDarkMode(userInfo.darkMode)
}
```

Ispis 49: Funkcija za postavljanje korisničkih podataka u Pinia-i

Funkcija iz ispisa 49 postoji da se ne bi ponavljao kôd u `login` i `singup` funkcijama. Prima korisničke podatke s kojima postavlja novo stanje i učitava JWT u memoriju preglednika kako bi se isti slao u svakom zahtjevu na API. Prima također i `isFromLogin` jer prioritet za postavljanje teme sučelje ima memorija preglednika, osim ako se korisnik tek prijavio jer tada se dohvaća postavljena tema iz baze podataka za tog korisnika.

```
async login(username: string, password: string): Promise<void> {
  this.isLoading = true
  try {
    const { data } = await api.post(ApiEndpoints.loginPath, {
      username, password
    })
    this.loadUserInfo(data, true)
    this.router.push('/')
  } catch (error: any) {
    throw new Error(error.response ? 'Wrong credentials' : 'Server
error')
  } finally {
    this.isLoading = false
  }
}
```

Ispis 50: Funkcija za prijavu

Funkcija iz ispisa 50 prima korisničko ime i lozinku te šalje zahtjev s tim podacima na API, a koji vraća JWT i podatke korisnika ili grešku ako je prijava neuspjela. `catch` će „uhvatiti“ grešku i stvoriti novu sa proizvoljnom porukom. Ako je prijava uspješna, postavit će se novo stanje te preusmjeriti korisnika na početnu stranicu. Slična je i funkcija `signup`, koja još prima i elektroničku poštu te šalje zahtjev na putanju za registraciju.

Postoji i funkcija za odjavu koja briše korisničke informacije iz stanja, briše JWT iz memorije preglednika, te preusmjerava korisnika na stranicu za prijavu. Osim navedenih, postoje još i dvije manje bitne funkcije za mijenjanje teme sučelja.

3.3.10 Axios – HTTP klijent

Prilikom inicijalizacije Quasar projekte, odabran je paket `axios` kao HTTP klijent. Tako se u „boot“ direktoriju dobije „axios.ts“ datoteka u kojoj se stvara instanca `axiosa` i postavlja konfiguracija. Na primjer, tu se nalazi presretač (engl. *interceptor*) koji dohvaća JWT iz memorije preglednika i ubacuje ga automatski prije slanja svakog zahtjeva. Kôd stvaranja `axios` instance i presretača se može vidjet u ispisu 51.

```
const api = axios.create({ baseURL: ApiEndpoints.baseUrl })
api.interceptors.request.use((config) => {
  const token = localStorage.getItem('token')
  if (token) config.headers.Authorization = `Bearer ${token}`
  return config
})
```

Ispis 51: `axios` instanca i presretač

Presretač provjerava nalazi li se u memoriji preglednika JWT, i ako jest stavlja ga u zaglavlje zahtjeva. U klasi `ApiEndpoints` se nalaze sve putanje SubWrld API-a, kao što se može vidjet na primjeru iz ispisa 52.

```
export class ApiEndpoints {
  static baseUrl = 'http://localhost:3000'
  static loginPath = '/users/login'
  static getTVShowDetailsPath = (tvShowId: string) => `/tv-
shows/${tvShowId}`
  ...
}
```

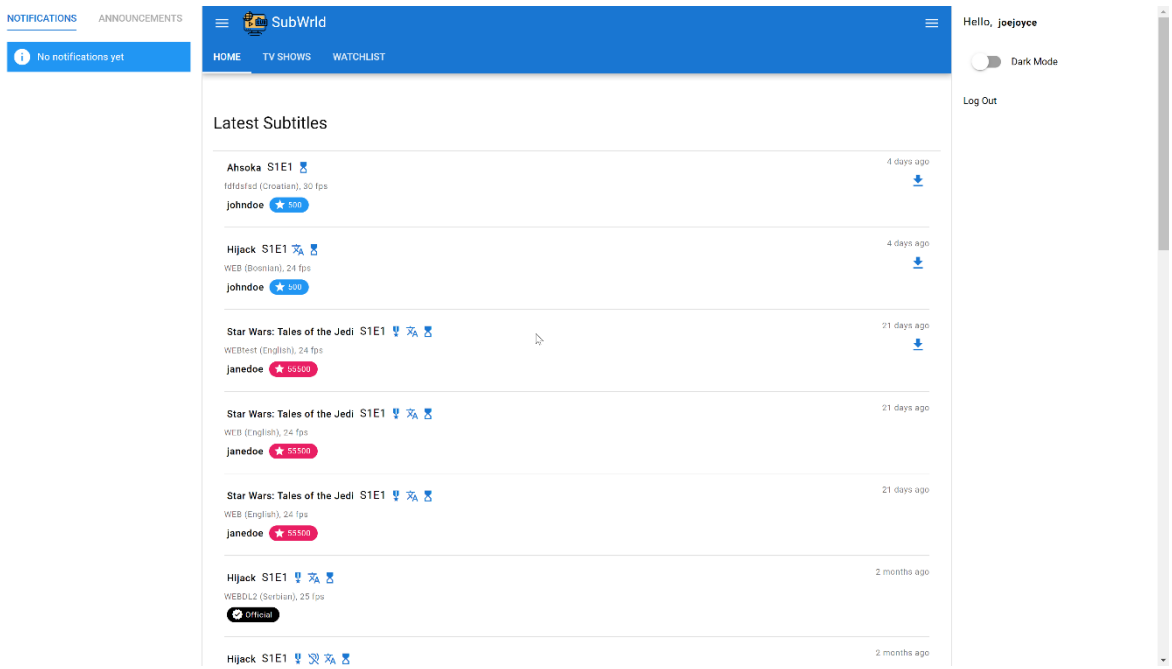
Ispis 52: `ApiEndpoints` klasa sa putanjama

4 Pregled funkcionalnosti

U ovom poglavlju su ukratko opisane i prikazane najvažnije funkcionalnosti aplikacije, odnosno samog korisničkog sučelja.

4.1 Glavni raspored

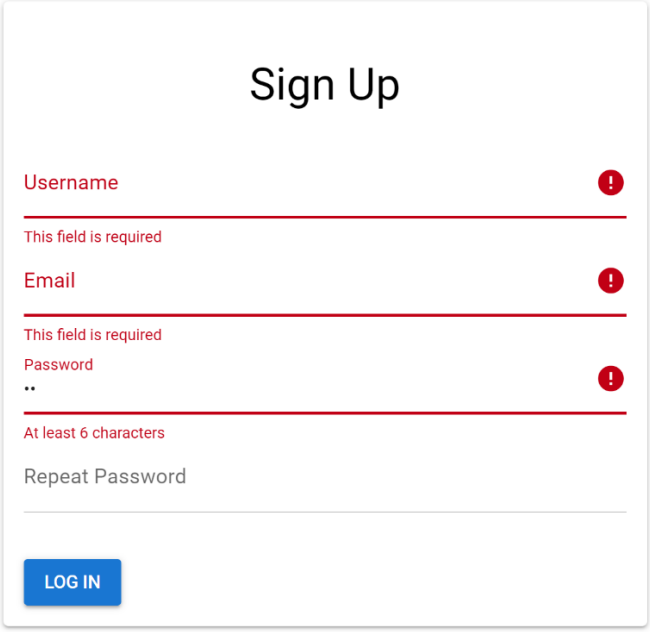
Korisničko sučelje se sastoji od nekoliko dijelova. Na vrhu se nalazi glavni izbornik sa logotipom, nazivom aplikacije i tipkama za prikaz ili skrivanje kliznih izbornika sa obje strane. Ispod toga u glavnom izborniku se nalaze navigacijski botuni za početnu stranicu, pretragu TV serija, listu za praćenje te listu sa prijavama ako je korisnik administrator. Sa strana se nalaze klizni izbornici koji su po zadanom prikazani na desktopu, a skriveni na manjim ekranima poput onih na mobitelu. U lijevom su prikazane obavijesti i objave vezane za serije koje su u korisničkoj listi za praćenje. Taj je izbornik prikazan jedino ako je korisnik prijavljen. U desnom izborniku se nalazi pozdravna poruka korisniku ili gostu, prekidač za prebacivanje između tamne i svijetle teme te navigacijske tipke za prijavu i registraciju, odnosno odjavu ako je korisnik prijavljen. Izgled sučelja se može vidjeti na slici 5.



Slika 5: Glavni raspored prijavljenog korisnika (svjetla tema)

4.2 Prijava i registracija

Prijava i registracija su osmišljene da budu vrlo jednostavne jer korisnici u većini slučajeva izbjegavaju registraciju ako forma zahtjeva previše vremena. Registracijska forma se može vidjeti na slici 6.



Slika 6: Registracijska forma

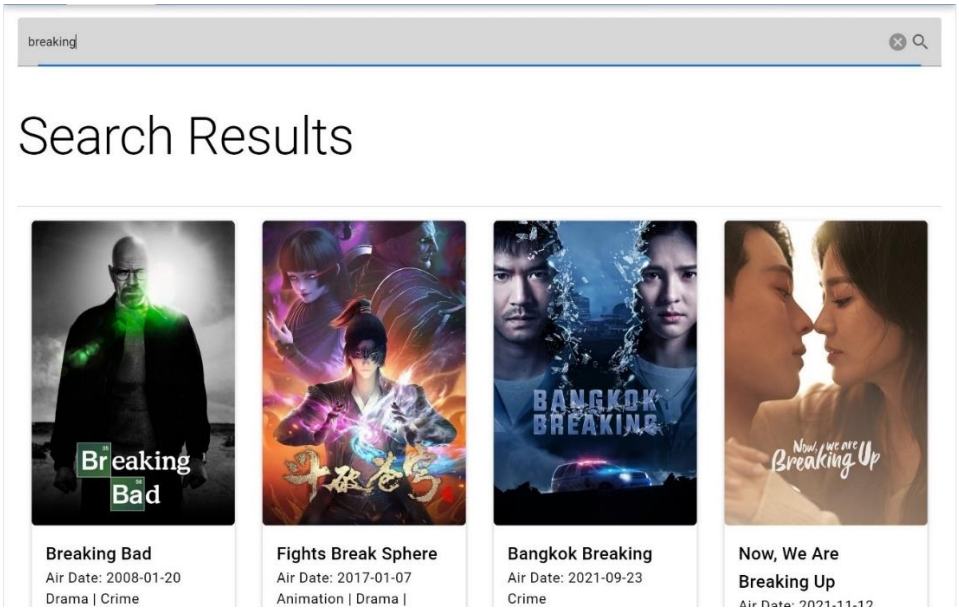
Registracijska forma ima samo četiri polja, a na slici se još može vidjet i kako izgleda validacija vrijednosti polja. Naravno, postoje i neka pravila kod registracije, na primjer lozinka mora imati bar šest znakova. Slična je i forma za prijavu, međutim ona sadrži samo polje za korisničko ime i lozinku.

4.3 Pretraga i pregled TV serija

Ovo je, uz same titlove, najvažniji dio aplikacije. Moguće je vidjeti TV serije koje su u tom trenutku popularne ili se može pretraživati po nazivu. Klikom na pojedinu TV seriju, mogu se vidjeti informacije o njima poput godine izlaska, broja epizoda, trajanje epizoda, ocjene korisnika, radnja itd., a naravno i popis svih sezona i epizoda.

Ako je korisnik prijavljen, ima mogućnost i dodavati TV serije u listu za praćenje te tako osim što ih ima sve na jednom mjesto, ima i sve objave za praćene TV serija na

jednom mjestu, a to je u lijevom izborniku pod „Announcements“. Osim toga, postoji i mogućnost označavanja epizode kao odgledane. Na slici 7 je kao primjer prikazano pretraživanje TV serija.



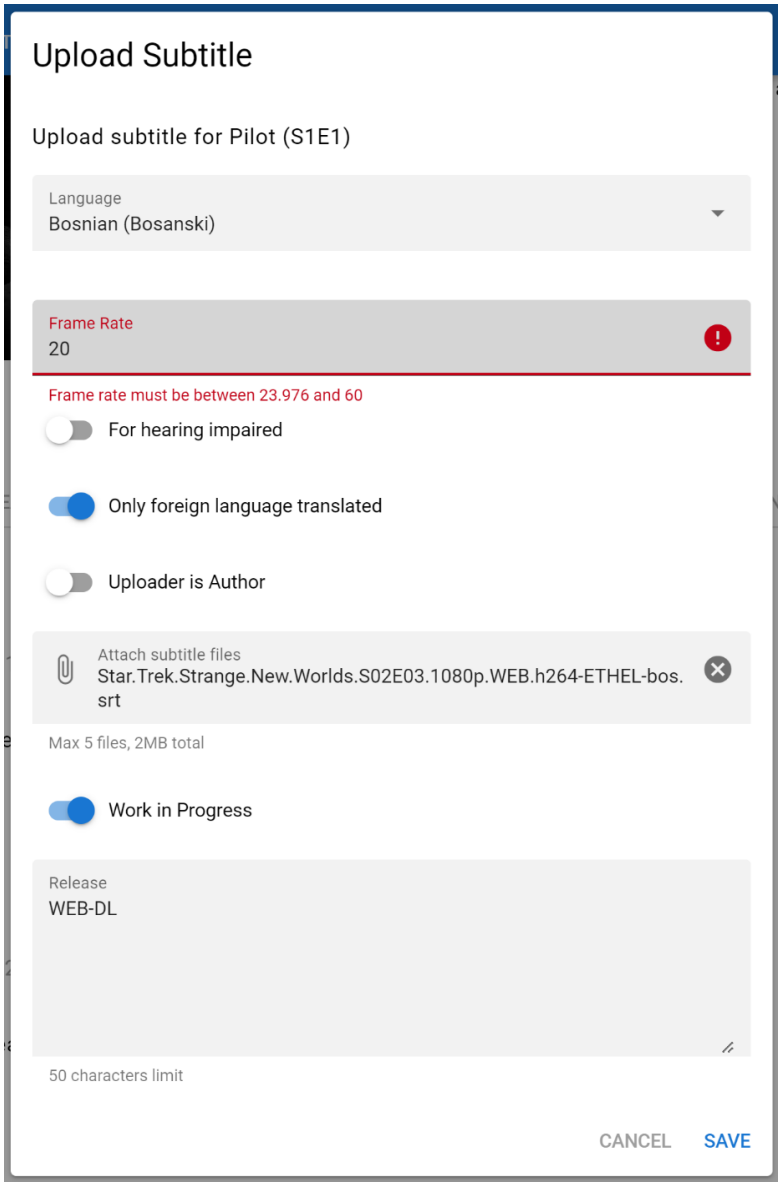
Slika 7: Pretraživanje TV serija

Na svaku TV seriju s popisa se može kliknuti i tada se učitavaju informacije o istoj, kao na slici 8.

Slika 8: Pregled informacija TV serije

4.4 Prijenos titlova

Još jedna funkcionalnost koju je potrebno posebno spomenuti je prijenos titlova. Svaka epizoda ima svoj padajući izbornik u kojem se može kliknuti na prijenos novih titlova ili pregled već postojećih za tu epizodu. Forma za prijenos titlova izgleda kao na slici 9.

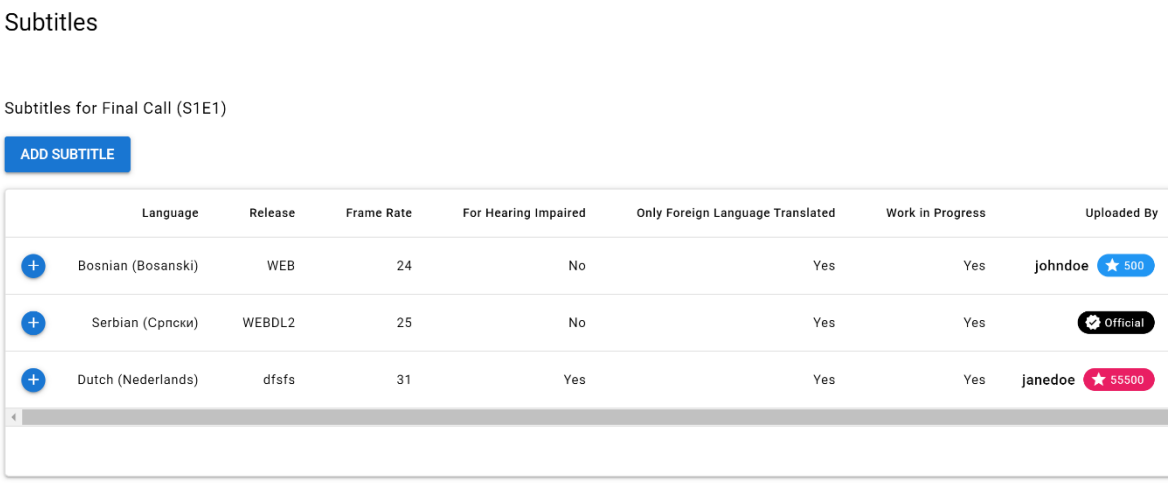


Slika 9: Forma za titlove

U formi je osim samih datoteka potrebno unijeti i neke opće informacije poput jezika, broja sličica u sekundi, verzije itd. Aplikacija podržava do pet datoteka po jednim

titlovima, koji se kasnije mogu preuzeti u arhiviranom formatu. Korisnik može označiti da su titlovi u izradi, čak i postaviti titlove bez datoteke kako bi dao do znanja drugim prevoditeljima da se već netko radi na tim titlovima. Naravno, administrator mora potvrditi titlove jednom kada su gotove, što korisnicima pruža sigurnost kod preuzimanja.

Na slici 12 se može vidjeti kako izgleda popis titlova za određenu epizodu, a koji su poredani po datumu zadnje izmjene.



Slika 10: Popis dostupnih titlova za epizodu TV serije

Na slici 10 nisu prikazane kolone kojih ima još dosta, međutim prikazano je pri kraju ono što je važno za spomenuti – reputacija. U polju „Uploaded By“ se osim samog korisničkog imena može vidjeti i reputacija korisnika. Ostali korisnici mogu zahvaliti za titlove te tako dati jedan reputacijski bod korisniku koji je prenio titlove. Također, nakon što su titlovi odobreni od administratora, dobije se 50 reputacijskih bodova.

4.5 Ostale funkcionalnosti

Korisnici mogu zatražiti titlove neke epizode na željenom jeziku, broju sličica u sekundi itd, a drugi korisnici zatim imaju mogućnost ispuniti tu želju. Naravno, korisnici svoje objavljene želje mogu izbrisati, izmijeniti, te ponovno otvoriti ako titlovi povezani sa željom nisu zadovoljavajući. Korisnici također mogu prijavljivati tuđe titlove, nakon čega administrator odlučuje o ishodu prijave. Osim toga, na početnoj stranici su vidljivi zadnji titlovi svih epizoda, zadnje objave vezane za TV serije, te popis korisnika poredanih silazno po reputacijskim bodovima.

5 Zaključak

Cilj aplikacije je povezati dvije stvari, dijeljenje titlova i praćenje TV serija. Takvo nešto još ne postoji na internetu i uz prikladan marketing aplikacija može lako pronaći svoj put do uspjeha. Mogućnost jednostavnog označavanja odgledanih epizoda i samih TV serija koje se prate je jako korisno, pogotovo u današnje vrijeme kada je često da se čeka godinama na novu sezone TV serije pa se lako zaboravi što se odgledalo, a što ne. Uz to imati i mogućnost preuzimanja titlova dosta pridonosi korisnošću aplikacije.

Naravno, budući da razvoj jedne ovakve aplikacije traje jako dugo, potrebno je odlučiti koje funkcionalnosti uključiti u početnu verziju, a koje ostaviti za daljnja ažuriranja. Premalo funkcionalnosti može odbiti korisnike od korištenja aplikacije, dok previše funkcionalnosti može odužiti razvoj dovoljno dugo da nestane potreba za njom.

Nadovezujući na to, puno toga se može poboljšati u trenutnoj verziji aplikacije. Prije svega, kada se poveća posjećenost, bilo bi korisno dodati mogućnost ocjenjivanja TV serija i pojedinačnih epizoda, kao i komentiranje, objave detaljnih osvrta (i korisničkih i službenih od administracije) itd. Ukratko, dugoročno je cilj stvoriti zajednicu koja bi dolazila dijeliti svoje mišljenje u iščekivanju novih epizoda sa ljudima koji dijele iste interese.

Postoje još neka druga poboljšanja koja su trenutno u vidu. Na primjer, korisnik bi mogao odabrati jezike koje razumije, nakon čega mu se prikazuju samo titlovi u tim jezicima. Sustav bi zapamtio odabire, a najbolje bi ih bilo zapisivati u bazu podataka. Nadalje, trebalo bi osmisliti funkcionalnost lakšeg preuzimanja titlova za više epizoda, recimo za cijelu sezonu. Osim toga, bilo bi korisno dodati i mogućnost administracije da označi određene titlove kao preporučene, olakšavajući tako korisnicima odabir titlova.

Još je potrebno spomenuti da je aplikacija izrađena koristeći moderne tehnologije tako da je korisničko iskustvo izuzetno dobro. Jako puno web stranica za dijeljenje titlova su zastarjele, stvarajući tako veliku potrebu za modernijim pristupom kao što ima upravo SubWrld. Svijet weba brzo napreduje, stoga je uvijek veliki izazov ostati konkurentan, a upravo je to ono što ova aplikacija može iskoristiti kako bi se izdvojila od ostalih slične tematike.

Literatura

[1] Quasar, „Quasar: Beyond the Framework,“ https://quasar.dev (posjećeno 7.8.2023.)

[2] MongoDB, „For the next generation of intelligent applications,“ https://www.mongodb.com/ (posjećeno 8.8.2023)

[3] ExpressJS, „Express: Fast, unopinionated, minimalist web framework for Node.js,“ https://expressjs.com (posjećeno 8.8.2023.)

[4] Mongoose, „Guide,“ https://mongoosejs.com/docs/guides.html (posjećeno 10.8.2023)