Razvoj mrežnih aplikacija pomoću okvira Ember.js

Mikulčić, Dominik

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Dominik Mikulčić

RAZVOJ MREŽNIH APLIKACIJA POMOĆU OKVIRA EMBER.JS

Diplomski rad

Voditelj rada:
v. pred. dr. sc. Goran Igaly

Zagreb, veljača, 2022.

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Mojoj obitelji koja mi je pružila bezuvjetnu podršku
i prijateljima koji su imali beskrajnu količinu razumijevanja.

Sadržaj

Uvod

Razvoj aplikacija je složen i iznimno dinamičan proces. Konstantno se javljaju nove tehnologije koje zamjenjuju one stare ili ih jednostavno potiskuju. Jedna od tehnologija za razvoj aplikacija je Ember i relativno je nova: nastala je 2011. godine. Međutim, Ember se otad redovito ali postupno razvija i s vremenom je stekao reputaciju tehnologije koja je komplicirana za svladati.

U ovome radu bavit ćemo se upravo Emberom i pokazat ćemo da, sustavnim pristupom (vidi [15]), Ember nije toliko složen kao što se čini. U prvom ćemo poglavlju napraviti povijesni pregled Embera, JavaScripta i drugih tehnologija koje Ember koristi. Također ćemo proći osnovna načela Embera i njegovu instalaciju.

Drugo poglavlje fokusirat će se na osnovne strukture Embera: predloške i komponente, a u paru s time i Handlebars i Glimmer tehnologije. U trećem poglavlju ćemo proučiti Emberov usmjerivač, način na koji upravlja putanjama i kontrolere. Pritom ćemo primijeniti znanja stečena iz prethodnih poglavlja i vidjeti na koji način Emberove strukture međusobno komuniciraju.

U četvrtom poglavlju ćemo se fokusirati na najbolji način da određene skupove podataka učinimo dostupnima unutar cijele aplikacije, a ne samo u jednoj putanji. Stoga će nam predmet proučavanja tog poglavlja biti servisi.

Peto poglavlje povezuje sve dotad navedene funkcionalnosti s nekim vanjskim API-jem. Ondje ćemo se baviti dohvaćanjem relevantnih, stvarnih podataka i njihovim prosljeđivanjem putanjama i komponentama. Na kraju, u šestom poglavlju, fokus ćemo staviti na aplikaciju koja će služiti kao primjer svih funkionalnosti Embera. Aplikacija koju ćemo razvijati će korisniku prikazivati podatke iz njegova Spotify računa i to koristeći Spotifyjev API. Stoga se mi uopće nećemo baviti serverskom stranom, već samo klijentskom.

U našoj aplikaciji ćemo omogućiti prikaz pjesama i umjetnika koje je korisnik najviše slušao, reprodukciju tih pjesama i prikaz statističkih podataka o umjetnicima i žanrovima koje sluša. Stoga ćemo kroz sva poglavlja od drugog do petog prikazivati i isječke koda koji su vezani uz dohvaćanje, prikazivanje i reprodukciju najslušanijih pjesama. Ti isječci će služiti za lakše shvaćanje Emberovih struktura, no glavni fokus na aplikaciji i dalje će biti u šestom poglavlju. Ondje ćemo povezati sve dotad obrađene strukture, usredotočiti se na sve dijelove aplikacije i pokazati da Ember doista nije bauk.

Poglavlje 1

Ember.js

Ember je razvojni okvir klijentske strane baziran na JavaScriptu. Prvo izdanje okvira bilo je 9. prosinca 2011., a originalni autor je bio Yehuda Katz. Otada je okvir doživio mnoštvo izdanja, a trenutno stabilno izdanje je 4.1.0. Okvir se koristi u mnogim aplikacijama, a neke od njih su *Apple Music*, *LinkedIn*, *Twitch*, *Productive*. Više o okviru možete pročitati u [5] i [6].

1.1 JavaScript i razvojni okviri

JavaScript je objektno orijentiran programski jezik baziran na prototipovima. Za programski jezik kažemo da je objektno orijentiran ako je temeljen na *objektima* koji sadrže podatke reprezentirane poljima i programske kodove reprezentirane metodama. Nadalje, za takav programski jezik kažemo da je baziran na prototipovima ako objekti mogu biti generalizacije širih pojmova koji se onda mogu klonirati ili dodatno proširiti neki drugi pojam (više u [2]).

JavaScript se prvi put pojavio 1995. godine te danas slovi kao jedan od najpopularnijih programskih jezika. Koristi se u većini web aplikacija i to primarno na klijentskoj strani (iako se može koristiti i na serverskoj strani). Neke od značajki koje ovaj programski jezik čine primamljivim jesu: učitavanje novih web sadržaja bez ponovnog učitavanja web stranice, animacije, validacija unešenih vrijednosti u web formi i mnoge druge. Naravno, uz JavaScript valja znati i HTML i CSS, jer dok posljednje dvoje daje programu strukturu i dizajn, JS pruža interaktivne elemente koji znatno poboljšavaju iskustvo korisnika.

Razvojni okvir je alat stvoren u svrhu poboljšavanja funkcionalnosti nekog već postojećeg alata, konkretno dodavanjem isječaka kodova napisanih od korisnika. Ideja iza toga je prilagoditi taj alat potrebama određenih vrsta aplikacije. Upravo zbog toga danas postoji mnoštvo različitih okvira: potrebe korisnika su sve veće i raznolikije pa je nužno prilagođavati im se i osmišljati nova oruđa. Tako postoje i mnogi okviri bazirani upravo na

JS-u, najpopularniji od kojih su: Node.js, Vue.js, Angular.js, React i – onaj kojim ćemo se mi baviti u ovom radu – Ember.js.

1.2 Uvod u Ember.js

Ember je, kako kaže njegov moto, *okvir za ambiciozne web developere*. Riječ je o sveobuhvatnom okviru za razvoj web aplikacija, ali to ne znači da je to jedan, samodostatan blok koda. On obuhvaća i neke JS biblioteke kao što je Glimmer (kojim ćemo se baviti u sekciji 2.2). Upravo je izdavanje Glimmera označilo početak Embera kao pravog modularnog okvira.

Ember je svakako *krut* okvir, to jest ima vrlo jasne stavove od kojih ni pod kojim uvjetima ne odstupa. Ti su stavovi duboko utkani u njegovu jezgru te za svaki zadatak obično postoji najbolji način, najbolji kod i najbolje mjesto za njega. Međutim, to ne znači da programer postaje rob ovog okvira. To samo znači da se programer može više usredotočiti na logiku aplikacije, poboljšavanju iskustva korisnika i prepustiti se kreativnosti, a ne previše razmišljati o tehnikalijama samog koda(vidi [14]).

Navedimo sada pet Emberovih beskompromisnih stavova:

1. **URL-ovi su karakteristični dijelovi aplikacije, stoga svaki okvir mora podržavati usmjerivač** (eng. *router*). Stoga i Ember ima izvrstan usmjerivač koji prihvaća razne parametre upita (eng. *query parameters*).

2. **Konvencija iznad konfiguracije**. Ember čvrsto vjeruje da se svi veći blokovi kodova (poput putanja, kontrolera i predložaka) trebaju međusobno slagati, bez da programer eksplicitno kaže kako se točno trebaju slagati. Zato Ember razvija striktnu konvenciju imenovanja tih objekata koja stvara veze među njima.

3. **Isključivo kod vrijedan pisanja treba biti napisan**.

4. **Deklarativni predlošci i Pravilo najmanje moći**. Predlošci u Emberu su napisani u *šablonskom* jeziku Handlebars. U njemu postoji vrlo ograničen skup pomagača koji se mogu koristiti, a JavaScript kod nije dobrodošao u predlošcima. Ovo je u skladu s Pravilom najmanje moći: da bi se riješio neki problem, potrebno je koristiti najmanje moćan jezik.

5. **Stabilnost bez stagnacije**. Tokom razvoja Ember 2.0, razvojni tim nije naprosto izdao mnoštvo novih funkcionalnosti i uništio stare. Želeći uvesti promjenu ali ne ugroziti rad postojećih aplikacija koje su tada radile na verzijama 1.x, tim je postupno uvodio nove funkcionalnosti jednu po jednu. Nakon nekog vremena, kad su se

korisnici navikavali na novosti, zastarjele funkcionalnosti bivale su postupno izbacivane. Na taj način se održala stabilnost okvira, ali on nije stagnirao i taj princip se održava i dan danas.

1.3 Ember CLI

Ember CLI je službeni način za stvaranje, izgradnju, testiranje i razvoj Ember aplikacije. Glavni razlog zašto se uopće koristi je zato da olakša život programeru. Kao što je već navedeno, u Emberu se teži ka tome da se programer fokusira na programiranje. Stoga je razvijen Ember CLI koji služi kao paker ovisnosti (eng. *dependency packager*), pokretač testova, optimizator i lokalni server. Međutim, to ne ograničava razvojnog programera na ugrađene sustave, već je moguće uvrstiti vlastite prilagodbe poput druge biblioteke za testiranje.

Da bismo mogli instalirati Ember CLI, prvo nam je potreban `npm` (vidi [9]). `npm` je zadani upravitelj paketa (eng. *package manager*) za JavaScript *runtime* okruženje Node.js. Sastoji se od klijenta naredbenog retka i baze javnih i privatnih paketa. Jedan od tih paketa je upravo Ember CLI.

Instalacija

Prije nego što počnemo s instalacijom, važno je napomenuti da se u ovome radu radi primarno u sustavu Windows. Međutim, svi koraci u instalaciji, razvoju i pokretanju bilo koje komponente su identični i u drugim operativnim sustavima (osim ako bude navedeno drukčije).

Prvi korak u instalaciji je instaliranje Node.js. Najbolji način za to je putem službene stranice (vidi [3]). Ondje odaberite LTS verziju, preuzmite na svoje računalo, otvorite datoteku i slijedite upute za instalaciju. Da biste se uvjerili da je instalacija dobro prošla, pokrenite sljedeću naredbu u naredbenom retku[1]:

```
$ node -v
```

Ukoliko je Node dobro instaliran, povratna informacija bit će verzija instaliranog softvera. Ovdje je bitno da ta verzija bude veća od 10, jer je to preduvjet za korištenje Ember CLI. Na sličan način može se provjeriti je li prisutan i `npm`:

```
$ npm -v
```

Tada će se ispisati koja verzija `npm`-a je instalirana. Nadalje, instalirat ćemo Yarn. Yarn je također upravitelj paketa koji služi i kao upravitelj projekata (vidi [4]). Iako `npm` i Yarn

[1] Budući da razvijamo unutar Windowsa, ovdje izraz *naredbeni redak* podrazumijeva *Command Prompt* u Windowsu. U sustavima Linux i macOS je ovdje riječ o terminalu, no naredbe su jednake.

imaju gotovo istu ulogu u razvoju nekog projekta, često se preporuča korištenje Yarna prije svega jer je mnogo brži, pouzdaniji i sigurniji. To se prije svega očituje u činjenici da `npm` koristi nedeterminističke algoritme za instalaciju ovisnosti, dok Yarn koristi deterministične algoritme. Međutim, i `npm` je konstantno nadograđivan i poboljšavan stoga ga mnogi programeri i dalje koriste, dok će se u ovom radu koristiti Yarn. Instalacija se radi na sljedeći način:

```
$ npm install --global yarn
```

Oznaka `--global` znači da će se paket instalirati globalno, to jest biti dostupan iz bilo kojeg dijela računala. Također možemo provjeriti uspješnost instalacije:

```
$ yarn -v
```

Sljedeći korak je upravo instalacija Ember CLI:

```
$ npm install -g ember-cli
```

Oznaka `-g` je pokrata za oznaku `--global`. Sama instalacija mogla bi potrajati nekoliko minuta, a po njezinom završetku opet možemo provjeriti uspješnost:

```
$ ember -v
```

Kao rezultat ispisat će se informacije o verziji Ember CLI-ja, `npm`-a i operacijskog sustava. Riječ `ember` je izvršna naredba koja se koristi za operacije za Ember CLI stoga ćemo je puno susretati u ovom radu.

Novi Ember projekt

Sad ćemo i službeno početi razvijati aplikaciju. Prvo se smjestimo u direktorij u kojem želimo stvoriti projekt, a potom pokrenemo sljedeću naredbu:

```
$ ember new hello-world --yarn
```

Ovdje je `ember` ključna riječ koja ukazuje na korištenje Ember CLI. Riječ `new` koristimo za kreiranje novog projekta, dok je `hello-world` ime projekta koji stvaramo. Na kraju, oznakom `--yarn` upravljanje projektom i njegovim ovisnostima dodjeljujemo upravo Yarnu. Stvaranje projekta bi moglo potrajati nekoliko minuta te kad to završi, dobit ćemo kompletnu strukturu direktorija s kojom Ember CLI radi.

U tom direktoriju imamo nekoliko različitih datoteka i poddirektorija:

- `app` - ovo je direktorij u kojem programer provodi najviše vremena tokom razvoja aplikacije. Sadrži poddirektorije koji se bave sastavnim dijelovima aplikacije o kojima ćemo govoriti u daljnjim poglavljima.
- `config` - ovaj direktorij sadrži konfiguracijske elemente projekta. Datoteka `environment.js` je zadužena za konfiguraciju aplikacije to jest okruženja (npr. razvojno okruženje, testno okruženje); `target.js` definira koje verzije preglednika

podržava naša aplikacija; `optional-features.js` upravlja neobaveznim značajkama aplikacije, a `ember-cli-update.json` sadrži konfiguraciju izgradnje aplikacije.

- `public` obuhvaća sve one datoteke za koje želimo da budu *javno* dostupne, poput fontova ili slika.
- `node_modules` je direktorij u koji se instaliraju svi `npm` paketi.
- `tests` sadrži testove za aplikaciju
- `vendor` sadrži vanjske ovisnosti koje ne instalira `npm`
- `package.json` definira potrebne `npm` pakete za projekt
- `ember-cli-build.js` opisuje kako bi Ember CLI trebao izgraditi aplikaciju
- `testem.js` sadrži konfiguraciju pokretača testova, Testem, kojeg koristi Ember CLI
- `yarn.lock` *zaključava* verzije potrebnih paketa

Budući da CLI dolazi sa svojim razvojnim serverom, možemo ga i pokrenuti, nakon što se pozicioniramo u direktorij projekta:

```
$ cd hello-world
$ ember -s
```

Ovdje je naredba `-s` samo pokrata za naredbu `--server` te označava upravo pokretanje servera. Taj proces bi mogao potrajati oko jedne minute, te će se tijekom njegova trajanja prikazivati poruke o izgradnji aplikacije za što se koristi Babel.js. Kad proces završi, dobit ćemo sljedeću informaciju:

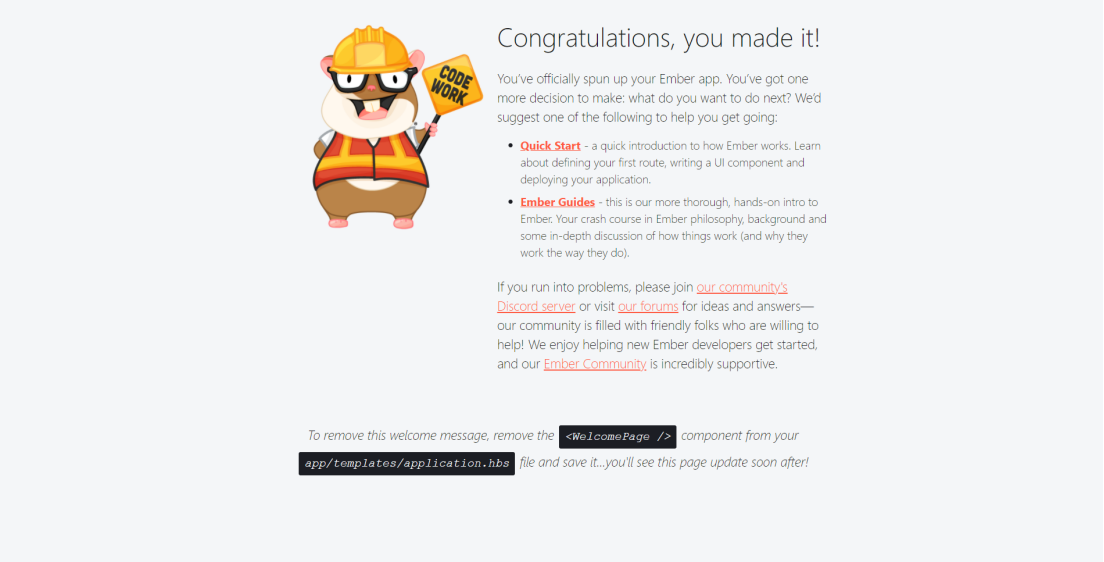
```
Build successful (13190ms) - Serving on http://localhost:4200/
```

Otvaranjem navedene stranice na lokalnom serveru, dobivamo prikaz kao na slici 1.1.

Ako u bilo kojem trenutku želimo zaustaviti izvođenje servera, dovoljno je u naredbenom retku preko tipkovnice pritisnuti kombinaciju `Ctrl+c` (u macOS-u `Cmd+c`). Sad ćemo poslušati savjet s početne stranice i ući u datoteku `app/templates/application.hbs` i doista maknuti sljedeći isječak koda:

```
1 {{!-- The following component displays Ember's default welcome message.
      --}}
2 <WelcomePage />
3 {{!-- Feel free to remove this! --}}
```

Ovime smo dobili "praznu" Ember aplikaciju koju sad možemo početi izgrađivati.



Slika 1.1: početna stranica aplikacije koja se dobije odmah nakon njezina stvaranja

Poglavlje 2

Osnovne Emberove strukture

U ovom poglavlju proći ćemo kroz dvije osnovne Emberove strukture: predlošci i komponente. Cijeli prikaz aplikacija počiva upravo na njima i zato je bitno dobro ih razumjeti. Stoga ćemo informativni dio ovog poglavlja upotpuniti kodom koji služi za implementaciju *Favourites* sekcije u aplikaciji. Ta sekcija služit će za dohvaćanje pjesama koje je korisnik najviše slušao, omogućit će reprodukciju tih pjesama, prikazivanje njihovih detalja i filtriranje po nazivu.

2.1 Predlošci

Emberovo korisničko sučelje je prije svega određeno HTML-om. To znači da je svaki dio aplikacije definiran nekim HTML predloškom te su oni ključni dijelovi aplikacije. Glavni predložak smješten je u datoteci `app/templates/application.hbs`. Stoga za početak možemo u nju staviti neki HTML kod i vidjeti kako će se to odraziti na našu aplikaciju:

```
<div class="favourite">
  Romeo and Juliet by Sergei Prokofiev (composed in 1935.)
</div>

<div class="favourite">
  Spartacus by Aram Khachaturian (composed in 1954.)
</div>

<div class="favourite">
  Swan Lake by Pyotr Ilyich Tchaikovsky (composed in 1876.)
</div>
```

U ovom kodu smo dodali tri `div` elementa koje nazivamo stavljamo u klasu `favourite` i u njih stavljamo po jedan favorit. Nakon što spremimo promjene i otvorimo lokalnu

stranicu, uočit ćemo vrlo jednostavan cron-bijeli prikaz. To se, dakako, može promjeniti korištenjem CSS-a, no to nam trenutno nije u prvom planu. Međutim, ovdje je bitno istaknuti da su osnovne tehnologije koje Ember koristi upravo HTML i CSS – temeljne tehnologije za razvoj web aplikacija. Upravo se u ovome očituje Emberovo Pravilo najmanje moći.

Handlebars

Poanta predloška jest, kao što i samo ime kaže, jest da se unaprijed nešto konstruira. Stoga kad pogledamo naš jednostavan kod s prethodnog prikaza, možemo primijetiti da sadrži izvjesnu količinu ponovljenog teksta koji bi se mogao nekako reducirati. Da bismo to mogli, prvo se trebamo upoznati s Handlebars.

Handlebars je, po definiciji, *jednostavan šablonski jezik* (vidi [8]). U stvarnosti, on koristi predložak i ulazne podatke kako bi generirao HTML ili kakav drugi tekstualni objekt. Handlebars izraz je neki tekst okružen dvama parovima vitičastih zagrada, na primjer: {{`izraz`}}. Kad se taj predložak izvrši, `izraz` se zamijeni odgovarajućom vrijednosti iz ulaznog objekta. Kao primjer možemo uzeti sljedeći kod koji uključuje Handlebars izraze:

```
<p> {{first_name}} {{last_name}} </p>
```

Ukoliko je ulazni objekt sljedeći:

```
{first_name: "Sergei", last_name: "Prokofiev" }
```

rezultirajući kod će biti sljedeći:

```
<p> Sergei Prokofiev </p>
```

Kao što možemo zaključiti, izraz `first_name` je zamijenjen sa `Sergei`, a `last_name` s `Prokofiev`. Sada naš prethodni kod možemo skratiti na način da iskoristimo nekoliko Handlebars izraza i pomoćnika:

- `each` je petlja koja prolazi po listi i nad svakim njenim elementom vrši danu operaciju
- `array` je ključna riječ koja označava listu
- `hash` je ključna riječ koja označava rječnik

Koristeći ove elemente, možemo sistematizirati naš kod:

```
{{#each
  (array
    (hash title="Romeo and Juliet" composer="Sergei Prokofiev"
        year=1935)
    (hash title="Spartacus" composer="Aram Khachaturian" year=1954)
    (hash title="Swan Lake" composer="Pyotr Ilyich Tchaikovsky"
        year=1876)
  ) as |favourite|
}}
    <div class="favourite">
        {{favourite.title}} by {{favourite.composer}}
            (composed in {{favourite.year}}.)
    </div>
{{/each}}
```

Izvršavanje ovako napisanog predloška daje HTML kod koji je jednak kodu koji smo već napisali, pa samim time dobivamo i isti izgled stranice. Međutim, ovakav kod je pregledniji i mnogo je jasnije što se u njemu doista zbiva. Ovo su, dakako, tek neke od funkcionalnosti koje Handlebars nudi, a vidljivo je da mogu značajno pomoći prilikom razvoja aplikacije.

Jedna zanimljivost Handlebarsa je njegova tolreantnost prema neuspjehu (eng. *failsoftness*). Naime, ukoliko neki atribut u predlošku nedostaje, on to jednostavno ignorira. Dakle ako imamo sljedeći unos:

```
<p> {{favourite.composer.year_of_birth}} <p>
```

rezultatni HTML kod bit će jednostavno , tj. neće biti javljena greška, iako `composer` nema atribut `year_of_birth`. To znači da se kao programeri ne moramo opterećivati rubnim slučajevima i brojnim `if-else` uvjetima. Međutim, to također znači da ukoliko negdje doista napravimo grešku (npr. tipfeler), ona neće biti prijavljena što otkrivanje greške čini iznimno teškim.

2.2 Komponente

Kad je kod dug i složen, uvijek težimo ka tome da ga razdvojimo u manje logične cjeline. Koristeći Handlebars, htjeli smo postići nešto slično, tako što smo određeni dio koda podijelili u manje segmente koristeći varijabilne ulazne podatke. Za sistematizaciju HTML koda u manje cijeline služe upravo **komponente**. Riječ je o manjim isječcima koda koje spremamo u zasebne datoteke i koristimo za stvaranje kompletnog prikaza. Komponente se smještaju upravo u direktorij `app/components`. One, poput predložaka, također prihvaćaju ulazne podatke koje onda koristimo u generiranju HTML koda. Jedino što se u ovom slučaju ti podatci šalju sa znakom `@` na početku naziva.

Na primjeru koda koji smo dosad razvijali, možemo uočiti jednu logičnu komponentu: `favourite`. Favorite ipak koristimo kao osnovnu cjelinu ove stranice, stoga ima smisla pretvoriti ih u komponente. Iz početnog koda vidimo da su nam kao ulazni podatci za komponentu potrebni naslov pjesme, ime autora i godina nastanka. Stoga kreiramo datoteku `favourite.hbs` u već spomenutom direktoriju i u nju napišemo sljedeći kod:

```
<div class="favourite">
  {{@title}} by {{@composer}} (composed in {{@year}}.)
</div>
```

Sad u datoteci `application.hbs` iskoristimo tu komponentu. Pozivamo je kao i bilo koju drugu HTML oznaku, ali će njoj ime biti zapisano s velikim početnim slovom te se unutar oznake prosljeđuju i ulazni parametri:

```
{{#each
  (array
    (hash title="Romeo and Juliet" composer="Sergei Prokofiev"
          year=1935)
    (hash title="Spartacus" composer="Aram Khachaturian" year=1954)
    (hash title="Swan Lake" composer="Pyotr Ilyich Tchaikovsky"
          year=1876)
  ) as |favourite|
}}
    <Favourite @title={{favourite.title}}
               @composer={{favourite.composer}}
               @year={{favourite.year}} />
{{/each}}
```

Ponovno, rezultatni HTML kod je isti kao i u prethodnim slučajevima, jednako kao i prikaz na stranici. Međutim, čitajući ovaj kod ipak vidimo jednu priču koja je razumljivija i nekom programeru koji se možda prvi put susreće s ovim kodom. Ukoliko želi vidjeti što `Favourite` doista radi, znat će gdje treba pogledati.

Uočimo ovdje kako nije bilo potrebno specificirati da je `Favourite` komponenta, niti reći koji je put do pripadne datoteke. U ovome se upravo očituje Emberov stav *Konvencija iznad konfiguracije*, jer mi ovdje doista ne moramo brinuti o problemima povezivanja ovih komponenti i predložaka. Na nama je da komponentu smjestimo u pravi direktorij, dodijelimo joj smisleno ime i to ime koristimo u ostalim dijelovima aplikacije. Ember odrađuje sve ostalo!

Glimmer.js

Glimmer je fleksibilan *rendering engine*[1] niske razine za izgradnju "živih" DOM-ova (objektni model dokumenta, eng. *Document Object Model*) pomoću Handlebars predložaka. DOM je aplikacijsko programersko sučelje (eng. *Application programming intefrace - API*) za HTML i XML dokumente koje definira logičku strukturu i način pristupanja dokumentima. Glimmer je pisan u TypeScriptu, a njegove komponente se mogu vrlo jednostavno prilagođavati promijenjenim podatcima. Više informacija možete saznati na službenoj stranici (vidi [7]).

Glavne prednosti Glimmer komponenata su:

- pisane su u obliku nativnih klasa. Naime, komponente iz starijih verzija Embera ličile su više na metode, dok nove, Glimmer komponente imaju klasnu strukturu.
- ne proširuju `EmberObject` kao što je to bio slučaj u starijim verzijama. Sad je za inicijalizaciju komponente dovoljno koristiti samo konstruktor.
- argumenti pripadaju `this.args` nazivnom prostoru (eng. *namespace*) koji je nepromjenjiv. Stoga je jasno kad se pristupa ulaznim podatcima, a kad se pristupa podatcima koji pripadaju samoj komponenti.

Dobar primjer potencijalne Glimmer komponente bi za našu aplikaciju bio sustav ocjenjivanja sa zvjezdicama. Pretpostavljamo da korisnik svakoj pjesmi može dodijeliti jednu do deset zvjezdica. Ukoliko korisnik odabere, na primjer, 8 zvjezdica, tada ćemo osam zvjezdica prikazati ispunjeno, a dvije prazno. Takvu ćemo komponentu nazvati `rating`. Sad ćemo u `app/components` dodati datoteku `rating.js`:

```
1  import Component from '@glimmer/component';
2
3  export default class RatingComponent extends Component {
4    get stars() {
5      let stars = [];
6      for (let i = 1; i <= 10; i++) {
7        stars.push({
8          star: i,
9          selected: i <= this.args.rating
10       });
11     }
12     return stars;
13   }
14 }
```

[1] *Rendering engine* je jedna od temeljnih softverskih komponenata svakog većeg web preglednika koja služi za transformaciju HTML i drugih objekata na stranici u vizualne reprezentacije na korisnikovom uređaju.

Prva linija ovog koda je umetanje Glimmer komponenti. U trećoj liniji definiramo našu komponentu koja proširuje klasu `Component`. Od četvrte do trinaeste linije definiramo funkciju `stars` na način da u listu `stars` pohranimo deset zvjezdica, od kojih su neke pune, a neke prazne. *i*-ta zvjezdicu u prikazu (numeracijom od 1 do 10) bit će ispunjena ukoliko je *i* manje od dane ocjene. Stoga ćemo za svaku zvjezdicu u listu dodati jedan hash objekt koji sadrži `star` kao redni broj zvjezdice i `selected` zastavicu koja označava je li zvjezdica odabrana ili nije.

Međutim, ovo je samo funkcija. Mi sad ove zvjezdice trebamo i prikazivati. Stoga trebamo dodati datoteku `app/components/rating.hbs` :

```
{{#each this.stars as |star|}}
  {{if star.selected "+" "-"}}
{{/each}}
```

Ovdje je riječ o običnoj Handlebars komponenti u kojoj prolazimo po svim zvjezdicama nekog `rating`-a i ukoliko je `selected` postavljeno na `true`, stavit ćemo plusić. Ukoliko nije, stavljamo minus. Ovdje će nam plusić označavati ispunjenu zvjezdicu, a minus praznu.

Sad je potrebno ovu komponentu uključiti u predložak `application.hbs` :

```
{{#each
  (array
    (hash title="Romeo and Juliet" composer="Sergei Prokofiev" year=1935
     rating=9)
    (hash title="Spartacus" composer="Aram Khachaturian" year=1954
    rating=10)
    (hash title="Swan Lake" composer="Pyotr Ilyich Tchaikovsky" year
    =1876 rating=8)
  ) as |favourite|
}}
    <Favourite @title={{favourite.title}} @composer={{favourite.composer
    }} @year={{favourite.year}} @rating={{favourite.rating}}/>
{{/each}}
```

i još modificirati komponentu `Favourite`:

```
<div class="favourite">
  {{@title}} by {{@composer}} (composed in {{@year}}.)
  <br>
  <Rating @rating={{@rating}} />
</div>
```

Vidimo da smo u predlošku za svaku skladbu dodali neku ocjenu, to jest još nije omogućeno dodjeljivanje ocjene. Također, sad i `rating` uzimamo kao ulazni podatak za `Favourite` komponentu. Ponovno, ove komponente su povezane preko nazivnog pros-

tora i mi ne trebamo komponenti `Favourite` eksplicitno reći da za `@rating` treba koristiti komponentu `Rating`.

U daljnjem kodu niti u aplikaciji **ne ćemo imati sustav ocjenjivanja**. Ovdje nam je *rating* služio kao jako dobar primjer što se s Glimmer (JS) komponentama može napraviti. Kasnije ćemo dodavati komponente koje će u kodu biti jednostavnije od *rating* komponente, ali su u ovome trenutku bile idejno previše napredne, to jest zahtjevaju još neke funkcionalnosti da bi se mogle implementirati.

Poglavlje 3

Usmjerivač

Jedan od Emberovih pet beskompromisnih stavova odnosi se upravo na URL-ove i njihovu važnost za aplikaciju. Ta važnost leži u činjenici da URL-ovi pričaju priču o aplikaciji. Uvijek postoji korijenska putanja na koju se nadovezuju ključne riječi koje govore o tome što korisnik traži i gleda. Na primjer: `https://my-app/artists/prokofiev/tracks` nam govori da je korisnik u nekoj aplikaciji *MyApp* pogledao sve umjetnike, odabrao Prokofieva te sad gleda popis njegovih skladbi.

Dakle, putanje možemo protumačiti kao prikaz trenutnog stanja aplikacije. To stanje se prosljeđuje mehanizmu za usmjeravanje koji onda na temelju putanje može iscrtati predložak, učitati model i učiniti ga raspoloživim za neki predložak, presumjeriti korisnika na drugu putanju ili rukovoditi akcijama koje mijenjaju model. U skladu s time, datoteka koja nas prije svega zanima je `app/router.js` :

```
1 import EmberRouter from '@ember/routing/router';
2 import config from 'hello-world/config/environment';
3
4 export default class Router extends EmberRouter {
5   location = config.locationType;
6   rootURL = config.rootURL;
7 }
8
9 Router.map(function () {})
```

Prije svega, u prvoj liniji uvozimo Emberove putanje. U drugoj liniji uvozimo konfiguraciju, tj. modul stvoren iz datoteke `app/config/environment.js` . Pomoću te konfiguracije postavljamo atribut `location` koji označava na koji će se način trenutna putanja aplikacije manifestirati u URL-u. U datoteci `app/config/environment.js` je `locationType` postavljen na `auto`. To je prema zadanim postavkama podešeno na `history`, što znači da se URL ažurira pomoću preglednikove funkcije `history.pushState`

i `history.replaceState`.

Deveta linija mapira sve putanje aplikacije. Budući da trenutno nemamo definiranu niti jednu putanju (osim korijenske), ta je funkcija zasad prazna. Međutim, jednom kad kreiramo putanje, one će se tamo prikazati te ćemo onda ovu datoteku moći koristiti i kao svojevrstan *kratak sadržaj* aplikacije. Često se programerima koji dolaze na neki novi projekt kaže da prvo pogledaju ovu datoteku upravo zato da bi dobili opći dojam kako aplikacija funkcionira.

Sad bi valjalo stvoriti prvu putanju. Budući da smo se dosad bavili favoritima i načinima na koje ih prikazujemo, sad ćemo kreirati putanju do tog prikaza. Stvaramo je jednostavnom i intuitivnom naredbom u terminalu:

```
$ ember g route favourites
```

Sad vidimo da se u našu datoteku `route.js` dodala nova putanja:

```
// ...

Router.map(function () {
  this.route('favourites');
})
```

Također smo dobili i novu datoteku `app/routes/favourites.js` . Osim toga, kreirao nam se i novi predložak `favourites.hbs` u koji sad možemo prebaciti kompletan kod datoteke `app/templates/application.hbs` jer taj kod doista pripada tom predlošku.

3.1 Navigacija među putanjama

Jednom kada je stvorena nova putanja i pripadni predložak, treba omogućiti prikazivanje sadržaja tog predloška na toj putanji. Ukoliko sad odemo na stranicu favorita `http://localhost:4200/favourites`, vidjet ćemo prazan ekran. Razlog za to leži u utičnicama (eng. *outlet*).

Naime, maločas navedeni URL prije svega iscrtava predložak `app/templates/application.hbs` što proizlazi iz dijela `http://localhost:4200/`. Nadalje, zbog ključne riječi `favourites` se poziva predložak `favourites.hbs` , ali se ne iscrtava zato što nigdje u predlošku `application.hbs` nismo naznačili gdje treba ubaciti taj predložak. Stoga na željeno mjesto upišemo Handlebars izraz {{`outlet`}}. Sad će se na tom mjestu iscrtavati predložak za favorite pa kad otvorimo pripadnu putanju, doista ćemo vidjeti željeni sadržaj.

Međutim, sad ako odemo na početnu stranicu `http://localhost:4200/`, glavni predložak ne dobiva informaciju u {{`outlet`}} o iscrtavanju nekog drugog predloška. Stoga je ta stranica sada prazna. Za početak bismo u nju onda stavili vezu na novu putanju koju smo stvorili. HTML oznaka za te veze je `LinkTo`. Sad glavni predložak izgleda ovako:

```
<LinkTo @route="favourites"> Favourites </LinkTo>
```

Time nam se stvori poveznica koja, kada na nju kliknemo, vodi na stranicu *Favourites*. U tom slučaju, ta će nam se poveznica prikazivati i na toj stranici s favoritima jer smo je smjestili u glavni predložak koji se iscrtava na svakoj putanji. To, ukoliko želimo, možemo i promijeniti. Međutim, u ovom ćemo slučaju ostaviti da ta poveznica bude uvijek vidljiva jer ćemo kasnije dodavati nove pomoću kojih ćemo onda kreirati glavni izbornik.

3.2 Modeli

Budući da je web aplikacija koju stvaramo dinamička, a ne statička, logično je za očekivati da će se *od nekud* dohvaćati podatci. U našem slučaju će se podatci dohvaćati iz Spotifyjevog API-ja. Da bi to bilo uspješno, treba iz naše aplikacije slati zahtjeve prema vanjskom API-ju te obraditi i prikazati dohvaćene podatke. Uobičajeno mjesto za to je upravo putanja. Zasad, mi ćemo se još uvijek baviti statičkim podatcima, a spajanjem na vanjski API ćemo se baviti nešto kasnije.

Za prosljeđivanje podataka predlošku (ili komponenti) korištenjem putanje, koristimo metodu `model`. Ta se metoda pokreće pri prijelazu na zadanu rutu te može vratiti bilo koji tip podatka. Bitna značajka modela jest da podatci koje vraća postaju automatski raspoloživi pripadnom predlošku u oznaci `@model` i kontroleru u oznaci `this.model`.

Obećanje

Model može vratiti JavaScriptov objekt obećanja (eng. *promise*). Objekt `Promise` predstavlja eventualno dovršenje ili neuspjeh neke asinkrone operacije. Ustvari je riječ o objektu koji se veže na uzvrat poziva (eng. *callback*). Uzvrat poziva je argument (obično isječak koda ili neka metoda) koji se prosljeđuje nekoj funkciji. Ta funkcija poziva argument nakon što se ispune određeni uvjeti. Više informacija o ovom objektu možete saznati u službenoj dokumentaciji Embera (vidi [10]).

U kontekstu naše aplikacije možemo zamišljati funkciju koja pokušava dohvatiti podatke o nekom favoritu, npr. `getFavourites`. Za nju imamo i dva uzvrata: jedan za situaciju kad su podatci uspješno dohvaćeni (`success`), a drugi za situaciju kada nisu dohvaćeni (`failure`). Inače bismo takvu funkciju pozivali na sljedeći način:

```
getFavourites(parameters, success, failure);
```

Međutim, ako funkciju `getFavourites` napišemo tako da vrati obećanje, onda uzvrate vežemo za samo obećanje, a ne za funkciju:

```
getFavourites(parameters).then(success, failure);
```

Glavne prednosti korištenja obećanja su da se uzvrati dodani pomoću metode `then` nikad neće pozivati prije nego što se izvrši pripadni slijed događaja vezan uz funkciju koja vraća obećanje. Također, uzvrati će se pozivati čak i ako su dodani nakon izvršenja tih događaja. Uzvrate je moguće pozivati i više puta pomoću `then` metode, a u tom slučaju će se oni i izvršiti više puta, onim redoslijedom kojim su pozvani.

Ukoliko obećanje koristimo u modelu, putanja će čekati da se to obećanje razriješi, to jest uđe u stanje uspjeha ili neuspjeha, prije nego što se iscrta predložak. Upravo zbog toga je model vrlo praktičan za korištenje prilikom slanja API zahtjeva. Budući da je potrebno čekati neko vrijeme da se dohvate podatci, a mi ne znamo hoće li zahtjev biti uspješan ili neuspješan sve do trenutka pristizanja podataka, obećanje za to vrijeme zaustavlja putanju.

Kako bismo prilikom preusmjeravanja na putanju `favourites` dobili potrebne podatke iz modela, treba ih prebaciti iz predloška `favourites.hbs` gdje se trenutno nalaze, u putanju:

```
import Route from '@ember/routing/route';

export default class FavouritesRoute extends Route {
  model() {
    return [
      { title: 'Romeo and Juliet', composer: 'Sergei Prokofiev',
        year: 1935, rating: 9 },
      { title: 'Spartacus', composer: 'Aram Khachaturian',
        year: 1954, rating: 10 },
      { title: 'Swan Lake', composer: 'Pyotr Ilyich Tchaikovsky',
        year: 1876, rating: 8 }
    ];
  }
}
```

Ovaj model *dohvaća podatke* (u ovom slučaju su podatci navedeni u kodu, kasnije će se dohvaćati preko API-ja) i prosljeđuje ih predlošku `favourites.hbs` gdje ih sad možemo direktno koristiti:

```
{{#each @model as |favourite| }}
    <Favourite @title={{favourite.title}}
       @composer={{favourite.composer}} @year={{favourite.year}} />
{{/each}}
```

Vidimo da se ovdje poziva model u obliku `@model`. Budući da je on vratio listu, možemo po njoj iterirati na jednak način, a konačan rezultat je isti kao i prije.

3.3 Ugniježđene putanje

Unutar glavnog predloška se uvijek iscrtavaju drugi predlošci. Međutim, ponekad želimo unutar jednog od sporednih predložaka iscrtati drugi sporedni predložak. U slučaju liste favorita, možda želimo klikom na neki favorit prikazati detalje o toj pjesmi poput popisa izvođača ili žanrova kojima pripada. Tada koristimo ugniježđene putanje.

Ovdje bi primjer takve putanje bio `/favourites/1`, gdje `favourites` označava putanju prema favoritima, `1` označava identifikaciju favorita, odnosno pjesme čije detalje želimo vidjeti. Uočimo da i putanju `/favourites` možemo promatrati kao svojevrsnu ugniježđenu putanju jer `/` upućuje na glavni predložak unutar kojeg se iscrtava predložak za favorite, očitovan u izrazu `favourites`.

Da bismo stvorili putanju `/favourites/favourite`, u terminalu pišemo sljedeći kod:

```
    $ ember g route favourites/favourite --path=:id
```

Ovom naredbom Emberu govorimo da generiramo ugniježđenu rutu `favourite` koja pripada ruti `favourites`, a toj ruti pristupamo pomoću argumenta `id`. Ako pogledamo u datoteku usmjerivača `router.js`, uočit ćemo sljedeći kod:

```
Router.map(function() {
  this.route('favourites', function() {
    this.route('favourite', {
      path: ':id',
    });
  });
});
```

Kod usmjerivača nam točno govori ono što smo i mi Emberu govorili naredbom u konzoli. Sad je dodana nova putanja prema favoritu, koristeći identifikaciju kao put.

Prije svega, da bi ovakva putanja funkcionirala, treba podatcima koji su trenutno u putanji `favourites` dodati odgovarajuće atribute. Svakoj preporuci pridružimo proizvoljni identifikator (može biti broj ili string). Potom u usmjerivaču za favorit `app/routers/favourites/favourite.js` treba prilagoditi model:

```
import Route from '@ember/routing/route';

export default class FavouritesFavouriteRoute extends Route {
  model(params) {
    let favourites = this.modelFor('favourites');
    return favourites.find((favourite) =>
    favourite.id === params.id);
  }
}
```

Prvo uočimo da je naziv usmjerivača analogan nazivu putanje – dakle uključuje i putanju-roditelja i putanju-dijete. Argument ovog modela jest `params` koji označava parametre koji su pruženi u putanji, u ovom slučaju identifikator. U petoj liniji pomoću metode `modelFor` pozivamo metodu `model` u usmjerivaču za favorite te dohvaćamo sve favorite. Potom vraćamo onaj favorit u listi čiji identifikator se podudara s identifikatorom navedenim u putanji.

Dakako, potrebno je prije svega za svaki favorit u popisu favorita dodati poveznicu prema njegovim detaljima. No, to nam neće biti jako težak zadatak. Sjetimo se, među putanjama navigiramo pomoću HTML oznake `LinkTo`. Stoga je dovoljno da svaki favorit "zamotamo" u tu oznaku - a to ćemo jednostavno napraviti unutar komponente `favourite.hbs` :

```
<LinkTo class="favourite" @route="favourites.favourite" @model={{@id}}>
  {{@title}} by {{@composer}} (composed in {{@year}}.)
</LinkTo>
```

Uočimo da ovdje točno specificiramo model koji se prosljeđuje ruti. Mi ćemo ga postaviti na ID favorita umjesto na sam favorit. Razlog za to je što po ID-ju uvijek možemo pretražiti listu kako bismo dobili potrebne informacije, kao što vidimo u isječku koda u putanji `routes/favourites/favourite.js` :

```
let favourite = favourites.find(
  (favourite) => favourite.id === params.id
);
```

Prije svega, vidimo da u putanji-dijete imamo pristup modelu za putanju-roditelj. Stoga iskoristimo taj pristup da dohvatimo traženi favorit po ID-ju. Možda će se netko zapitati zašto ne proslijedimo jednostavno cijeli model umjesto da pretažujemo listu. U ovome slučaju je to stvar osobne preference. Ne gubi se mnogo u performansama ako moramo naknadno vršiti pretragu, ali ako proslijedimo cijeli model, onda prosljeđujemo potencijalno mnogo teži objekt od jednog običnog stringa. Opet, niti ovdje performanse nisu značajno umanjene. Stoga je ova situacija doista stvar osobnog izbora, a meni je jednostavno ljepše vidjeti ID nego cijeli objekt.

Nakon napisane rute, treba napisati i predložak za favorit. U ovome trenutku su naši favoriti vrlo jednostavni, ali možemo zamisliti da se u riječniku svakog favorita nalazi i par gdje je ključ `artists`, a vrijednost neka lista s popisima umjetnika (u obliku stringova) koji su sudjelovali u stvaranju ili izvođenju te skladbe. Stoga bi naš predložak mogao, za početak, izgledati ovako:

```
<div class="favourite-info">
  <p>{{@name}}</p>
  <p>{{@year}}</p>
  <div class="favourite-top-artists">
    {{#each @artists as |artist|}}
      <p> {{artist}} </p>
    {{/each}}
  </div>
</div>
```

Ovdje smo sve informacije jednostavno formatirali na drukčiji način, prikladan za neku vrstu pregleda detalja. Naravno, jednom kad se povežemo na API, imat ćemo pristup većem broju informacija pa će taj pregled biti napredniji. Na kraju je još samo potrebno na kraj predloška `templates/favourites.hbs` dodati utičnicu {{`outlet`}}. U nju će se priključiti odabrani favorit i na njenom mjestu će se prikazati njegov predložak.

Iz ovoga doista vidimo važnost putanja za Ember aplikaciju. Prije svega, Ember im daje jako veliku moć: one ne samo da navigiraju stranicom, već dohvaćaju i dalje prosljeđuju odgovarajuće podatke. Stoga je ključ razvoja kvalitetne Ember aplikacije upravo u dobrom poznavanju putanja.

3.4 Kontroleri

Dok se u većini drugih okvira kontroler shvaća kao relativno nezavisan, samostojeći objekt, ovdje kontroler promatramo kao objekt vrlo usko vezan uz usmjerivač i putanje. Stoga kontroler možemo definirati kao usmjeriv objekt koji od putanje prima jedno svojstvo - model, a to je upravo povratna vrijednost `model()` funkcije u putanji.

Kontroler se uparuje s istoimenom putanjom i tako ponovno slijedi pravilo *Konvencije iznad konfiguracije*. Model se, po zadanom, iz pripadne putanje prosljeđuje kontroleru funkcijom `setupController()`. Obično se kontroler koristi za modifikacije tog modela kako bi se dobila neka dodatna svojstva.

Jedno dodatno svojstvo na objektima *Favourites* koje je korisno, jest zasigurno pretraživanje po nekom parametru. Prije nego što generiramo kontroler i u njega smjestimo neki kod, treba promisliti što uopće želimo da kontroler radi. Svakako ćemo na stranici `favourites` htjeti imati polje za unos parametra. Međutim, nećemo imati tipku koja će pokrenuti pretragu, već ćemo pretragu vršiti pri svakoj promjeni polja za unos. Stoga generirajmo kontroler:

```
    $ ember g controller favourites
```

U datoteku `controllers/favourites.js` trebamo uvesti Ember paket `controller`, koji služi za implementaciju kontrolera. Druga bitna stvar koju ćemo uvesti jest

Glimmer komponenta `tracking`, to jest Glimmerov dekorator `tracked`. O Glimmerovim komponentama smo već pričali u sekciji 2.2, a ova ugrađena komponenta nam služi da pratimo promjene vrijednosti nad kontrolerom. Usto, svaki put kad se praćena vrijednost promijeni, to je Emberu znak da treba ponovno iscrtati stranicu. Mi ćemo uvesti praćenu vrijednost `searchFrase` i inicijalno je postaviti na prazan string. Ta vrijednost će predstavljati uneseni parametar korisnika po kojemu se vrši pretraga favorita.

Sad kad pratimo potrebnu vrijednost, treba po njoj pretražiti naslove i vratiti one favorite koji se podudaraju s unosom. Za to će nam služiti metoda `foundTracks`. Sada naš kontroler izgleda ovako:

```
import Controller from '@ember/controller';
import ENV from 'hello-world/config/environment';
import { tracked } from '@glimmer/tracking';

export default class FavouritesController extends Controller {
  @tracked searchFrase = '';

  get foundTracks() {
    return this.model.filter((track) => {
      return track.name.toLowerCase().includes(this.searchFrase.
   toLowerCase());
    });
  }
}
```

Primijetimo da metoda `foundTracks` dohvaća `model` dobiven iz istoimene rute pa ga pretražuje po imenima naslova. Ali ta imena prvo pretvorimo u stringove s isključivo malim slovima, isto kao i korisnikov unos. Ne želimo baš korisniku zadavati muke da mora paziti na veliko i malo slovo.

Vratimo se sad na predložak `favourites.hbs`. Ovdje želimo povezati akcije korisnika s akcijama u kontroleru. Prvo ćemo na mjestu gdje se dosad u predlošku prikazivao `@model`, sad prikazivati dobivene rezultate iz metode `foundTracks` u kontroleru:

```
{{#each this.foundTracks as |favourite| }}
   <Favourite @title={{favourite.title}}
    @composer={{favourite.composer}} @year={{favourite.year}}/>
{{/each}}
```

Uočavamo da se na kontroler referenciramo pomoću ključne riječi `this`, pa potom na njemu pozivamo metodu. Sad nam još ostaje u predložak dodati i prikazivanje polja za unos parametra pretrage:

```
<div class="search-items">
  <label for="favourites-search">Search tracks: </label>
  <input type="text" name="favourites-search" id="favourites-search"
   value={{this.searchFrase}} {{on "input" this.updateSearchFrase}}>
</div>
```

Osim standardnog HTML koda i već uvedenih Handlebars komponenti, za vrijednost unosa stavljamo upravo `searchFrase` iz kontrolera. Da bismo povezali akciju promjene unosa s akcijom pretrage, stavljamo još jednu Handlebars komponentu: `on "input" this.updateSearchFrase`. Ona znači da za svaku *input* akciju, to jest promjenu unosa, izvršavamo metodu `updateSearchFrase` u kontroleru. Tu ćemo metodu u kontroleru definirati na sljedeći način:

```
@action
  updateSearchFrase(event) {
    this.searchFrase = event.target.value;
  }
```

Metoda je vrlo direktna: kao ulaz prima događaj, dohvaća *metu* događaja, to jest njezinu vrijednost i pridružuje je našoj praćenoj vrijednosti `searchFrase`. Ovdje koristimo dekorator `@action` koji označava upravo da je riječ o akciji koju pozivamo iz nekog predloška. Stoga taj dekorator treba uvesti u predložak naredbom:

```
import { action } from '@ember/object';
```

Ovime je kontroler `favourites` dovršen. Navedimo sad još točno tok pretrage s korisničke i aplikacijske strane:

1. korisnik unese parametar pretrage
2. okine se akcija na kontroleru `updateSearchFrase`
3. promijeni se praćena vrijednost `searchFrase`
4. Ember ponovno iscrta stranicu, a filtrirane favorite dohvati putem metode `foundTracks`

Netko će se možda zapitati zašto kontrolerima ne dajemo više pažnje. U mnogim drugim aplikacijskim okvirima kontroleri ipak imaju jako bitnu ulogu, ako ne i ključnu. Razlog je u tome što Ember već nudi dva jako moćna alata o kojima smo već govorili: komponente i putanje. Komponente primarno služe za sistematizaciju podataka, dok putanje služe za njihovo dohvaćanje i dalje prosljeđivanje komponentama (ili čak direktno predlošcima).

Čemu onda služi kontroler i treba li ga uopće koristiti? Neki će Ember developeri reći da kontroleri u Emberu nisu potrebni, neki će možda *po starim navikama* veliki dio tereta aplikacije staviti upravo na kontrolere. Međutim, preporuka je kontrolere shvatiti kao *bazne* komponente, to jest one koje dobivaju već obrađene podatke od putanje, uređuju ih na različite načine i, ovisno o potrebi, prosljeđuju dalje komponentama ili predlošcima. Time se smanjuje određena količina tereta na drugim strukturama i jasnije se odjeljuju funkcionalnosti aplikacije. Zanimljiv članak na tu temu možete pročitati na stranici [16].

Poglavlje 4

Servisi

Dosad smo prošli modele, komponente, putanje i kontrolere, i svi ovi dijelovi su nam pokazali kako unutar jedne putanje možemo koristiti podatke koji su nam na raspolaganju. Ponekad određene podatke želimo dijeliti među različitim putanjama, a dosad obrađeni dijelovi nam za to nisu dovoljni.

Servisi služe upravo za dijeljenje podataka među različitim dijelovima aplikacije, a životni vijek jednog servisa je jednak kao i životni vijek aplikacije. Česti primjeri korištenja servisa su:

- autentikacija korisnika (*login* i *logout* akcije)
- obavijesti korisniku poslane od servera
- pozivanje API-ja treće strane
- sastavljanje dnevnika korištenja aplikacije (*logging*)

Mi ćemo u ovoj aplikaciji servise koristiti za autentikaciju korisnika, upravljanje odabranim umjetnikom ili pjesmom na odgovarajućoj stranici i reprodukciju pjesama na bilo kojem mjestu u aplikaciji. Budući da nam je, zbog prirode naše aplikacije, za autentikaciju korisnika potreban Spotify API, time ćemo se baviti u sljedećem poglavlju. Biranje umjetnika ili pjesme će nam biti potrebno samo unutar jedne putanje pa nam to nije dovoljno kvalitetan primjer. Stoga ćemo se u ovome poglavlju fokusirati na reprodukciju pjesama. Iako nam je i za to potreban Spotify API, pozive prema API-ju nećemo raditi unutar servisa. Stoga ćemo korištenje servisa u ovom poglavlju oprimjeriti reprodukcijom pjesama.

4.1 Postavljanje temelja

Osim što u aplikaciji prikazujemo pjesme u obliku liste, u sljedećim ćemo poglavljima omogućiti i prikazivanje informacija o nekoj pjesmi. Usto, želimo omogućiti i reprodukciju te pjesme klikom miša na tipku *Play* uz tu pjesmu. Također, želimo omogućiti i kontrolu reprodukcije u podnožju stranice. Kao i u svim drugim programima za slušanje pjesama, tu kontrolu želimo na bilo kojem mjestu u aplikaciji.

Prije nego što napišemo kod za servis, proći ćemo po predlošcima i komponentama te modificirati njihov kod. Ovisno o potrebama koje nam se ukažu u ovom procesu, pronaći ćemo metode koje su nam potrebne za servis. Budući da želimo da svirač bude na bilo kojem mjestu u aplikaciji, očito je riječ o kodu koji ćemo više puta koristiti. Stoga nam prvo treba Handlebars predložak koji ćemo nazvati `player.hbs`. Njega ćemo moći jednostavno umetnuti u neki dio aplikacije i imati na raspolaganju funkcionalan svirač.

```
<button id="togglePlay" type="button" {{on "click" this.click }}> <img
    src={{this.icon_path}} /> </button>
{{this.song_description}}
```

Naravno, svirač je zapravo samo tipka s opcijom *Play* i *Pause* i opisom pjesme. Stoga u prethodnom kodu na klik tipke pozivamo akciju `click` u JS komponenti koju ćemo tek napisati. Ta akcija će služiti za pokretanje ili zaustavljanje reprodukcije. Na sličan način ćemo dohvaćati putanju do ikone koja će se prikazivati na tipki, ovisno o tome svira li pjesma ili je pauzirana.

Ključna akcija je klik mišem. Kada kliknemo, imamo dvije opcije koje su usko povezane s problematikom oko ikone:

1. pjesma je trenutno pauzirana → pokrećemo reprodukciju
2. pjesma se trenutno reproducira → pauziramo je.

Sad već vidimo što će nam trebati u servisu: akcije `play` i `pause`, metoda `isPlaying` (koja provjerava je li u tijeku reprodukcija) i praćeni atributi `author`, `songTitle` i `playing`. Međutim, napišimo prvo JS komponentu `player.js` uz pretpostavku da servis imamo već napisan s navedenim akcijama i metodama:

```
export default class PlayerComponent extends Component {
  @service player;

  @action
  click() {
    if(this.player.isPlaying()) this.player.pause();
    else {
      var response = getCurrentSong();
      this.player.set('song_title', response.item.name);
```

```
      this.player.set('author', response.item.author);
      this.player.play();
    }
  }

  get song_description() {
    var prefix = this.player.isPlaying() ? 'Now playing: ' : 'Paused: ';
    var suffix = this.player.author;
    return prefix + this.player.song_title + ' by ' + suffix;
  }

  get icon_path() {
    if (this.player.isPlaying()) return "assets/images/pause.png";
    return "assets/images/play.jpg";
  }
}
```

U ovome kodu vidimo da akcija `click`, kao što smo već naveli, pauzira pjesmu ukoliko ona trenutno svira, inače pjesmu pokreće. No, u tom slučaju još i ponovno postavljamo ključne argumente u sviraču. Na sličan način, u ovisnosti o tome svira li pjesma ili je pauzirana, postavljamo opis pjesme (što se očituje u prefiksu i sufiksu) i njezinu ikonu.

Uočimo da servis pozivamo na sličan način kao i praćene vrijednosti: `this.player`. Metode i atribute na servisu pozivamo jednostavnim nadodavanjem točke i imena metode/atributa: `this.player.isPlaying()`. Za servise također postoje već ugrađene metode, kao što je metoda `set`. Ona služi za postavljanje vrijednosti nekog atributa na servisu i poziva se na način da se kao prvi argument daje ime atributa koji se postavlja u obliku stringa, a kao drugi argument se daje vrijednost na koju se postavlja atribut.

Prilikom dohvaćanja pjesme koja je u sviraču, koristimo metodu `getCurrentSong`. Njezinu implementaciju ćemo dati u sljedećem poglavlju kad se spojimo na Spotify API.

Postavljanje pjesme

Uočimo da smo dosad kod našeg svirača promatrali isključivo slučaj kad već imamo postavljenu pjesmu te treba samo kliknuti na sviranje ili pauzu. Sad još valja implementirati i postavljanje pjesme iz liste favorita na svirač. Da bismo to učinili, trebamo se vratiti na Handlebars komponentu `favourites.hbs` gdje smo definirali konstrukciju pojedinog favorita. Sad ćemo, osim informacija o pjesmi prikazivati i tipku za pokretanje reprodukcije sljedećim isječkom koda:

```
<button type="button" name="button" class="play-favourite"
    {{on "click" this.play}}><img src="assets/images/play.png" />
    </button>
```

Ova tipka funkcionira na analogan način kao i kod svirača, samo što vidimo da nam je ikona uvijek postavljena na *play*. To je u redu, jer iz liste favorita nam nije toliko bitno kontrolirati kompletnu reprodukciju, već želimo određenu pjesmu prebaciti na svirač.

Sad još samo treba napisati kod akcije `play` u JS komponenti `favourites.js`. Međutim, nema potrebe pisati taj kod jer u toj akciji putem informacija o pjesmi na koju smo kliknuli dohvaćamo sa Spotify API-ja dodatne informacije o reproduciranju pjesme. Potom postavljamo atribute na servisu o nazivu pjesme i imenu autora te na njemu pozivamo akciju `play`.

4.2 Servis *Player*

Nakon što smo postavili temelje, treba napisati kod za sam servis. Već smo naveli koje sve atribute, akcije i metode će servis imati i vidjet ćemo da njihove implementacije neće biti komplicirane. Najsloženiji dio je bio osmisliti kako servis funkcionira i kako će se on uklopiti u ostale dijelove aplikacije.

Za praćene atribute `author` i `songTitle` želimo, dakako, da pamte ime autora i naslov pjesme, redom. Za atribut `playing` želimo da bude postavljen na istinu ako se u danom trenutku reproducira neka pjesma, a u suprotnom na laž. Tako će nam metoda `isPlaying` zapravo vraćati vrijednost tog atributa. Usto, akcije `play` i `pause` će zapravo postavljati vrijednost tog atributa: istina za `play` a laž za `pause`. Pogledajmo stoga kod:

```
import Service from '@ember/service';
import { tracked } from '@glimmer/tracking';

export default class PlayerService extends Service {
  @tracked song_title = '';
  @tracked author = '';
  @tracked playing = false;

  play() {
    this.playing = true;
  }

  pause() {
    this.playing = false;
  }

  isPlaying() {
    return this.playing;
  }
}
```

I doista, kod nije kompliciran ali je izrazito moćan upravo zato što je riječ o servisu. Sad je ovaj kod, to jest njegova funkcionalnost, dostupan u bilo kojem dijelu aplikacije u kojem želimo pozivati servis.

Ovdje je dobro ponovno istaknuti kolika je prednost građenja koda u manjim i organiziranim cijelinama. Napravimo mali pregled načina na koji naš svirač funkcionira. Imamo servis `Player` koji služi za dijeljenje informacija o sviraču među različitim putanjama. Njegove metode poziva JS komponenta `Player` koja služi kao izravan posrednik između prikaza, to jest korisničke akcije, i servisa, to jest informacija koje će nam biti dostupne kroz cijelu aplikaciju. Na kraju imamo i Handlebars komponentu `Player` koja služi za generiranje prikaza. Nama je sada dovoljno upravo tu komponentu uključiti na bilo koje mjesto u kodu i naš svirač će biti prikazan na tom mjestu u aplikaciji i bit će potpuno funkcionalan. Dakle, mi ponovno uviđamo da ovakav način programiranja ima veliku moć i programeru pojednostavljuje posao na dulje staze.

Poglavlje 5

Povezivanje s vanjskim API-jem

Kao što smo govorili u Uvodu, fokus ovog rada je isključivo na klijentskoj strani aplikacije. Zapravo, mi se uopće ne opterećujemo onime što se događa na serverskoj strani. Znamo da ona postoji i da su svi podatci koji su nam potrebni upravo ondje. Mi na klijentskoj strani dohvaćamo podatke sa servera, eventualno ih mijenjamo da dobijemo željene informacije u željenom formatu i te informacije prikazujemo korisniku aplikacije.

Međutim, budući da mi ne pišemo serversku stranu aplikacije, od nekud trebamo dohvaćati podatke. Stoga ćemo za to koristiti Spotifyjev API i u ovom poglavlju ćemo se fokusirati na vezu između API-ja i korisnika, upravo preko naše aplikacije.

Spotify API

Spotify je švedski pružatelj audio *streaming* usluga. Osnovan je 2006. godine i danas broji preko 381 milijun aktivnih korisnika mjesečno, čime je najveći svjetski pružatelj takvih usluga. Preko 172 milijuna korisnika je na plaćenoj verziji usluge i na taj način dobiva sadržaje bez reklama i s mogućnošću slušanja glazbe bez internetske veze. Spotify je dostupan u preko 180 zemalja diljem svijeta te na većini modernih uređaja današnjice.

Glavna je namjena Spotifyja upravo slušanje glazbe, ali dakako nudi se i slušanje *podcast* emisija. Sadržaj se može pretraživati putem naslova, naziva umjetnika, albuma, a može se i organizirati unutar *playlisti* koje se pak mogu dijeliti s drugim korisnicima (vidi [12] i [13]).

Također, Spotify daje pristup svom API-ju koji je REST API. REST (eng. *Representational state transfer*) označava softverski stil koji služi kao vodič za dizajn i razvoj arhitekture *World Wide Web*-a. REST zapravo definira skup ograničenja koja bi ta arhitektura trebala poštovati. Većina REST API-ja, pa tako i Spotifyjev, su temeljeni na HTTP metodama: GET (dohvatiti - čitanje resursa), PUT (staviti - uređivanje resursa), POST (objaviti - stvaranje resursa) i DELETE (brisati - uništavanje resursa). Resursima se manipulira preko URL-ova, a podatci se šalju korištenjem JSON i XML formata.

U slučaju Spotify API-ja, korišteni format je upravo JSON. JSON (*JavaScript Object Notation*) je podatkovni format koji je namijenjen za razmjenu podataka na način da budu čitljivi i ljudima. Tako se JSON zapisuje kao rječnik, to jest u formatu:

$$\{\ \text{ključ}_1 \Rightarrow \text{vrijednost}_1,\ \text{ključ}_2 \Rightarrow \text{vrijednost}_2,\ \ldots,\ \text{ključ}_n \Rightarrow \text{vrijednost}_n\ \}$$

gdje je za svaki $i \in \{1, \ldots, n\}$ ključ_i string, a vrijednost_i proizvoljni podatak, no najčešće string, broj, lista ili rječnik. Ovaj format se prvi put javio upravo u JavaScriptu (odatle i naziv), ali danas taj format ne ovisi o programskom jeziku. Štoviše, većina jezika omogućuje generiranje i parsiranje podataka u JSON formatu. Isti se, dakle, format koristi i u slanju i primanju podataka kroz Spotify API. Putem njega možemo dohvaćati informacije o najdražim umjetnicima ili pjesmama nekog korisnika, dohvatiti informacije za pojedinu pjesmu i reproducirati je i mnogo više.

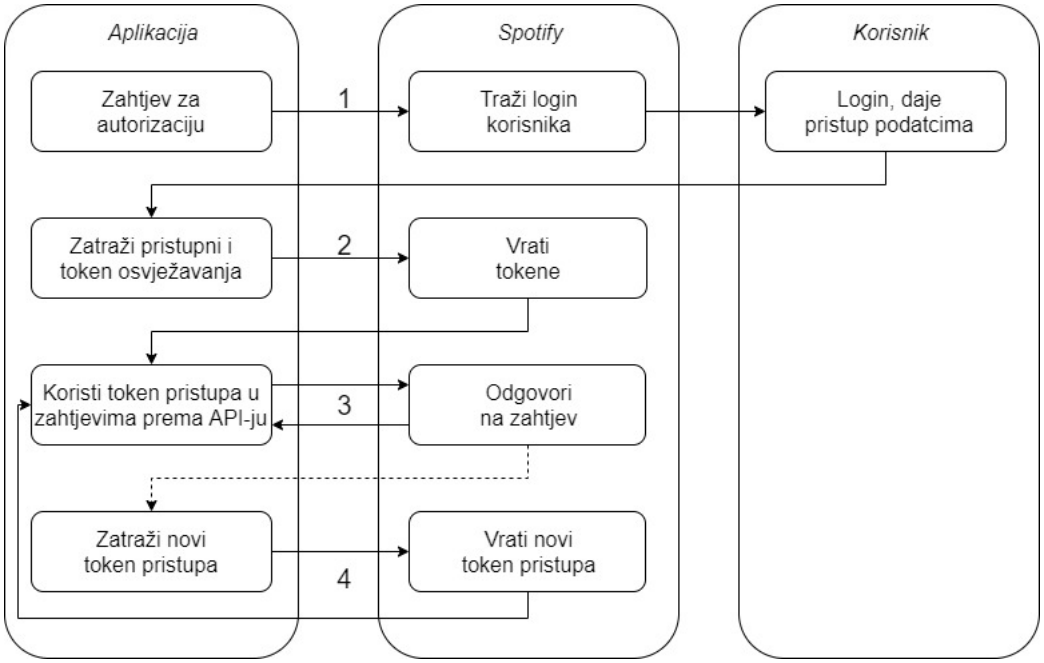
5.1 Autentikacija korisnika

Da bismo uopće mogli dohvaćati podatke vezane uz pojedinog korisnika, prvo nam taj korisnik treba dati dopuštenje za to putem prijave. Dakle, glavni fokus ovog odjeljka je implementacija procesa autentikacije korisnika.

Prvi korak je postaviti aplikaciju u postavkama stranice *Spotify for Developers* (vidi [11]). Ovdje nećemo ulaziti u pretjeranu dubinu samog procesa, budući da su upute na stranici vrlo jasne. No, uspješnim postavljanjem aplikacije, ovdje dobivamo dva vrlo bitna podatka: ID klijenta - `client_id` i tajnu klijenta - `client_secret`. ID je jedinstveni identifikacijski kod naše novokreirane Spotify aplikacije. Tajna je ključ koji služi za autorizaciju poziva prema API-ju ili SDK-u (o kojem ćemo kasnije reći nešto više). Da bismo bolje razumijeli i pratili proces autentikacije, pogledajmo dijagram na slici 5.1.

Prvi dio procesa uključuje autorizaciju korisnika putem Spotifyja. Naša aplikacija odgovara na korisnički zahtjev za prijavu te šalje Spotifyju zahtjev za autorizaciju. Tada aplikacija preusmjeri korisnika na Spotifyjevu stranicu gdje korisnik izvrši prijavu. Zauzvrat, naša aplikacija dobije kod.

U drugom dijelu, naša aplikacija koristi dobiveni kod i šalje ga Spotifyju kako bi dobila tokene pristupa i osvježavanja. U tom zahtjevu također koristi ID i tajnu klijenta. U trećem dijelu za zahtjeve prema API-ju (npr. za dohvaćanje favorita ili umjetnika) naša aplikacija koristi token pristupa. Taj token ima rok trajanja, a kada istekne, prelazimo na četvrti dio. Tada pomoću tokena osvježavanja dohvaćamo novi token pristupa i ponavljamo treći dio.



Slika 5.1: prikaz procesa autentikacije.

Implementacija autentikacije

U prvom dijelu, to jest zahtjevu za autorizaciju, riječ je o akciji unutar naše aplikacije. Ta akcija će se i kod naše aplikacije realizirati kao tipka s natpisom *Login*, stoga tu akciju možemo nazvati `login`. Dosad smo većinu akcija stavljali u JS komponente, a poneke smo stavljali u kontrolere. Ovu akciju mogli bismo staviti u kontroler `application.js` i to bi bilo sasvim u redu. Međutim, možemo i razmišljati unaprijed: jednom kad korisnik izvrši login, htjet ćemo prikazivati njegove podatke u nekoj sekciji koja će biti prisutna na bilo kojoj stranici. Stoga nam je radi preglednosti i organiziranosti koda zgodnije definirati neku komponentu `User`, pa onda u pripadnoj JS komponenti definirati tu akciju.

Pogledajmo sad isječak koda koji implementira akciju `login`:

```
@action
  async login() {
    const authEndpoint = 'https://accounts.spotify.com/authorize';
    const clientId = 'client_id';
    const scopes = [
      'user-top-read',
      'user-follow-read',
      'user-read-recently-played',
      'streaming',
```

```
10        'user-library-read',
11        'user-read-email',
12        'user-read-private',
13        'user-read-playback-state'
14      ];
15      const redirectUri = 'http://localhost:4200/callback';
16      window.location = `${authEndpoint}?client_id=${clientId}
            &redirect_uri=${redirectUri}&scope=${scopes.join('%20')}
            &response_type=code&show_dialog=true`;
17    }
```

Srž ovog koda je zapravo preusmjeravanje na točno određeni URL u liniji 16. Mi želimo korisniku omogućiti da se u našu aplikaciju ulogira putem Spotifyjeve stranice, te ga ondje treba preusmjeriti. Varijabla `authEndpoint` je korijenski dio URL-a na koji preusmjeravamo korisnika. Nadalje, `clientId` je ID klijenta koji smo dobili prilikom postavljanja aplikacije (ovdje, iz mjera predostrožnosti, ne navodimo taj podatak u stringu). U idealnoj situaciji, kad bismo ovu aplikaciju objavljivali, koristili bismo neku javnu uslugu gdje bismo pohranjivali ovakve osjetljive podatke. Dobar primjer je *Amazon Web Services* - AWS koji nudi uslugu trezora (*vault*) u kojem se onda ovakve informacije pohra ne i odakle se mogu dohvaćati unutar aplikacije i sigurno koristiti.

Varijabla `scopes` određuje koliki opseg dopuštenja će imati korisnik unutar naše aplikacije, odnosno što će smjeti vidjeti i raditi. Na web stranici *Spotify for Developers* možemo vidjeti koje su sve moguće vrijednosti i njihove opise pa pomoću toga odlučiti koje vrijednosti su nam potrebne za našu aplikaciju. Međutim, određene vrijednosti funkcioniraju u određenim kombinacijama, to jest ne funkcioniraju samostalno. Stoga sam na temelju toga došao do ove liste vrijednosti. Vrijednosti koje su nam neophodne za aplikaciju su `user-top-read` pomoću čega možemo pristupiti korisnikovim najslušanijim pjesmama i umjetnicima, `user-read-playback-state` čime dobivamo pristup čitanju stanja reprodukcije pjesme i `streaming` čime kontroliramo reprodukciju pjesme.

Varijabla `redirectUri` govori Spotifyju gdje treba preusmjeriti korisnika i poslati podatke nakon što je izvršen login. Vidimo da će nam za to biti potrebna nova putanja - `callback`. Međutim, da bi preusmjeravanje natrag na tu putanju funkcioniralo, treba je navesti u postavkama naše aplikacije na *Spotify for Developers* u *Dashboard* → *ImeAplikacije* → *Edit settings* → *Redirect URI's*. Na kraju, sve dosad navedene podatke stavimo kao parametre unutar našeg URL-a. Time smo prošli prvi dio autorizacijsko-autentikacijskog procesa.

Nakon što korisnik izvrši prijavu preko Spotifyja, nalazimo se u `callback` putanji gdje treba zatražiti tokene pristupa i osvježavanja. Dohvatiti te tokene je relativno jednostavan zadatak sad kad smo od Spotifyja dobili kod. Međutim, te tokene negdje treba pohraniti i oni trebaju biti dostupni u svim dijelovima aplikacije. Stoga ćemo napraviti novi servis `User` koji će kao atribute imati `accessToken` i `refreshToken`. Pogledajmo sad kod

unutar `callback` putanje:

```
import Route from '@ember/routing/route';
import ENV from 'hello-world/config/environment';
import { inject as service } from '@ember/service';
import fetch from 'fetch';

export default class CallbackRoute extends Route {
  @service user;

  async model() {
    var _code = window.location.toString().substring(1).split('=')[1];
    const tokenEndpoint = 'https://accounts.spotify.com/api/token';
    const clientId = 'client_id';
    const clientSecret = 'client_secret';
    const redirectUri = 'http://localhost:4200/callback';
    let response = await fetch(
      `${tokenEndpoint}?grant_type=authorization_code&code=${_code}
       &redirect_uri=${redirectUri}`,
      {
        method: 'POST',
        headers: {
          Authorization: 'Basic ' + btoa(clientId + ':' + clientSecret),
          'Content-Type': 'application/x-www-form-urlencoded',
        },
      }
    );
    let json = await response.json();
    this.user.set('access_token', json.access_token);
    this.user.set('refresh_token', json.refresh_token);
  }
}
```

Prođimo prvo kroz slijed događaja u metodi `model`. Najprije iz URL-a dohvatimo kod koji nam šalje Spotify, a potom postavimo nekoliko potrebnih varijabli. Prva varijabla, `tokenEndpoint` je krajnja točka od koje preko Spotifyja dobivamo tokene. Zatim imamo `clientId` i `clientSecret` koji su potrebni za dohvaćanje tokena i to šifrirani u formatu *base64*.

Format *base64* funkcionira tako da se svakom znaku u nekom stringu u ASCII formatu prvo pridruži njegov binarni zapis. Potom se ti binarni zapisi konkateniraju u jednakom redoslijedu kao i pripadni ASCII znakovi u stringu. Zatim se prođe po novodobivenom zapisu te se svakom šesteroznamenkastom isječku pridruži novi simbol prema tablici. Po-

gledajmo kako to funkcionira na primjeru riječi *Ember*:

$$
\begin{array}{lll}
 & & 010001 \to R \\
E \to 01000101 & & 010110 \to W \\
m \to 01101101 & & 110101 \to 1 \\
b \to 01100010 \Longrightarrow & 010001010110110101100010011001010111010 \Longrightarrow & 100010 \to i \\
e \to 01100101 & & 011001 \to Z \\
r \to 01110010 & & 010111 \to X \\
 & & 0010 \to ?
\end{array}
$$

Budući da zadnji binarni zapis ima samo 4 znamenke, ne možemo mu pridružiti niti jedan znak iz *base64* tablice (vidi [1]). Stoga ga nadopunjujemo dvjema nulama da dobijemo šesteroznamenkasti zapis (uočimo: duljina našeg zapisa mora biti djeljiva sa 6). U tablici vidimo da tom zapisu pripada znak *I*. Sad dobivamo `RW1iZXI`. No, još nismo gotovi. Uočimo da naš rezultat ima duljinu 7. Kad bismo ga išli dešifrirati, zapisali bismo tih 7 znakova u šesteroznamenkastim binarnim zapisima i dobili bismo binarni zapis duljine $7 \times 6 = 42$. Zatim bismo taj zapis razdijelili u 8 binarnih brojeva i svakome pridružili znak iz ASCII tablice. Međutim, broj 42 nije djeljiv s 8 pa bismo se tu našli u velikom problemu. Iz ovoga zaključujemo da binarni zapis mora biti djeljiv s 8. Kako mora biti djeljiv i sa 6 i s 8, zaključujemo da mora biti djeljiv s 24. To znači da duljina našeg šifrata u *base64* mora biti djeljiva s $24 : 6 = 4$. Zato ćemo na šifrat nadodati potreban broj razmaka (simbol =) kako bi duljina bila djeljiva s 4 pa je konačan rezultat `RW1iZXI=`. Srećom, u JavaScriptu postoji funkcija koja to radi: `btoa`.

Vratimo se sad na dohvaćanje tokena. Sljedeća varijabla, `redirectUri`, koristi se samo u svrhu validacije procesa. Ovdje zapravo nema nikakvog preusmjeravanja. Sljedeći korak jest poslati zahtjev prema Spotify API-ju. Ovdje koristimo metodu `fetch` i u nju ubacujemo sve parametre koje smo naveli i uz njih još `grant_type` postavljamo na `authorization_code`.

Ovdje koristimo i ključnu riječ `await`. Koristimo je zato što metoda `fetch` ne vraća neki *gotov* objekt, već vraća obećanje. Ako se sjećamo iz sekcije 3.2, obećanja se moraju razriješiti. Stoga stavljamo `await`, što znači da prvo pričekamo da se neka metoda pozove i izvrši, to jest da se obećanje razriješi, a tek nakon toga se izvršava kod koji dolazi nakon te metode. Uočimo, isto činimo i u liniji 25: `response` je i dalje obećanje, pa moramo stavljati `await` da osiguramo razrješenje prije izvršavanja daljnjeg koda. Nakon što smo dohvatili odgovor u JSON formatu, u `user` servis spremamo tokene pristupa i osvježavanja.

5.2 Dohvaćanje podataka

Prethodna poglavlja vezana uz strukturu Embera smo oprimjerili implementacijom *Favourites* dijela aplikacije. Međutim, dosad smo imali statičke, gotove podatke. Sad kad smo omogućili prijavu korisnika i kad imamo korisnikov token pristupa, možemo ga koristiti kako bismo dohvatili prave, to jest relevantne podatke sa Spotifyja.

Korijenski URL Spotifyjevog API-ja jest `https://api.spotify.com/`. Na njega nadodajemo krajnje točke i parametre pomoću čega dohvaćamo željene podatke. Za dohvaćanje pjesama koje korisnik najviše sluša, koristimo krajnju točku `me/top/tracks`. U sljedećem isječku dan je kod metode `model` u putanji `routes/favourites.js`:

```
1  async model() {
2    let response = await fetch('https://api.spotify.com/v1/me/top/tracks'
       + '?limit=50', {
3      method: 'GET',
4      headers: {
5        Authorization: 'Bearer ' + this.user.get('access_token'),
6        'Content-Type': 'application/json',
7      }
8    });
9    return await response.json();
10   let tracks = [];
11     for (let datum of json.items) {
12     tracks.push(new Favourite(datum));
13   }
14   return tracks;
15 }
```

Način na koji šaljemo zahtjev na API je vrlo sličan kao i u procesu autentikacije. Jedino što ovaj put ne moramo slati mnogo parametara. Šaljemo jedan parametar, limit, i postavljamo ga na najveću dopuštenu vrijednost - 50. U zaglavlju zahtjeva postavljamo autorizaciju na token pristupa iz `User` servisa, a kao tip sadržaja stavljamo `application/json`.

Nadalje, u listu `tracks` želimo spremiti sve dohvaćene favorite. Međutim, u liniji 12 vidimo da u listu dodajemo neki `Favourite` objekt. Ovdje je riječ o **modelu**. Imajmo na umu: dosad smo o modelima govorili u kontekstu putanja. Međutim, Ember ima strukturu koja se također zove model te je njezina namjena analogna namjeni klase u objektno orijentiranom programiranju. Možemo definirati atribute koje sadrži model i način na koji ih postavlja te kreirati instance modela.

Namjena modela jest zapravo da bolje organizira kod i način na koji upravljamo podatcima. Konkretno, u dvanaestoj liniji mi modelu `Favourite` prosljeđujemo jedan JSON objekt. U njegovom konstruktoru iz JSON-a izvučemo i organiziramo podatke i na taj način

ih pripremimo za daljnje korištenje. Pogledajmo kako to izgleda u modelu `models/favourite.js` :

```
import Model from '@ember-data/model';

export default class Favourite {
  constructor(data) {
    this.name = data.name;
    this.id = data.id;
    this.url = data.external_urls.spotify;
    this.preview_url = data.preview_url;
    this.duration_ms = data.duration_ms;
    let image_data = data.album.images[0];
    this.image_url = image_data.url;
    this.image_width = image_data.width;
    this.image_height = image_data.height;
    this.artists = data.artists;
    this.uri = data.uri;
  }
}
```

Vidimo da model sadržava slične argumente kao i favoriti koje smo definirali na samom početku. Dodali smo vanjske poveznice za otvaranje stavki u Spotifyju, trajanje pjesme i podatke o naslovnoj slici pjesme.

Lista koju smo na ovaj način definirali se dalje koristi u kontroleru `Favourites` gdje se filtrira i šalje istoimenom predlošku i potom prikazuje korisniku. Ovime zaključujemo i posljednje poglavlje kojim prolazimo kroz teorijska i osnovna tehnička znanja potrebna za stvaranje aplikacije u Emberu. Prošli smo sve osnovne konstrukcije Embera, definirali njihove uloge i povezali ih u smislenije cjeline na temelju jednog primjera - *Favourites* - i sad je na redu povezivanje svih cjelina u funkcionalnu aplikaciju.

Poglavlje 6

Aplikacija

Kroz dosadašnja poglavlja smo prolazili pojedinačno svaku strukturu i iznosili njenu funkciju, prednosti i slabosti. Primjerima smo pokazivali na koji način struktura funkcionira i kako komunicira s ostalim strukturama, pritom se fokusirajući na sekciju s favoritima. U ovom poglavlju ćemo proći strukturu aplikacije, utvrditi slijed događaja koji se odnosi na sekciju *Favourites* i obratiti pozornost na ostale sekcije.

6.1 Struktura

Kao što smo već naveli, funkcija ove aplikacije je pružiti uvid u pjesme i umjetnike koje korisnik sluša preko svog Spotify računa. Možemo reći da je riječ o nekom uvidu u glazbu, pa odatle i naziv aplikacije - *MuseSight* (eng. *music* - glazba, *insight* - uvid).

Ona ima četiri glavne sekcije: *Favourites*, *Artists*, *Insights* i *About*. Sekcija *Favourites* će prikazivati listu 50 najslušanijih pjesama i omogućiti prikaz detalja pojedine pjesme kao i njezinu reprodukciju. Na sličan način funkcionira i sekcija *Artists*, samo što je riječ o umjetnicima i ne nudi se mogućnost reprodukcije sadržaja za te objekte. Sekcija *Insights* sadržava grafove koji prikazuju najslušanije umjetnike i žanrove korisnika. Sekcija *About* daje informacije o aplikaciji i upute za korištenje.

Jedna podsekcija je svakako *Login* i *Logout* opcija, a druga je *Player*. O njima nećemo govoriti kao o osnovnim sekcijama jer pripadaju izborniku ili zaglavlju aplikacije i nemaju svoje zasebne putanje. Ali svejedno ćemo njima posvetiti određenu pozornost jer je prijava korisnika nužna za funkcioniranje aplikacije, a reprodukcija pjesama je vrlo snažna funkcionalnost.

6.2 Početak korištenja i prijava korisnika

Kad prvi put otvorimo stranicu `http://localhost:4200`, vidimo osnovne informacije o aplikaciji, njezinoj namjeni i korištenju (Slika 6.1).

Slika 6.1: Prikaz početnog ekrana aplikacije

Ukoliko kliknemo na bilo koju od ostale tri ponuđene sekcije, nađemo se na stranici koja nas upozorava da nismo prijavljeni i da je potrebno prijaviti se kako bismo mogli koristiti aplikaciju. Klikom na tipku *Login* u donjem lijevom kutu, preusmjereni smo na Spotifyjevu stranicu gdje se od nas traži da se prijavimo. Uspješnom prijavom preusmjereni smo natrag na aplikaciju. Implementacija korisničke prijave je detaljno objašnjena u sekciji 5.1.

Sad u donjem lijevom kutu vidimo obavijest da smo prijavljeni s našim korisničkim imenom. Uočimo da se ondje nalazi i Spotifyjev logo. Ukoliko na njega kliknemo, preusmjereni smo na prikaz našeg Spotify profila. Ovaj dio je implementiran putem `User` komponente. U Glimmer komponenti već se nalazi implementacija `login` akcije, ali naknadno dodajemo pomoćne metode koje nam daju informacije o korisniku preko `User` servisa. Naime, ukoliko je u servisu postavljen `access_token`, korisnik je prijavljen pa u Handlebars komponenti `components/user.hbs` nudimo opciju odjave. Inače, korisnik nije prijavljen pa nudimo mogućnost prijave. Ovdje možemo zaključiti da klikom na tipku *Logout*, to jest odjavom korisnika, vrijednost varijable `access_token` u servisu postavljamo na `null`.

6.3 Favoriti

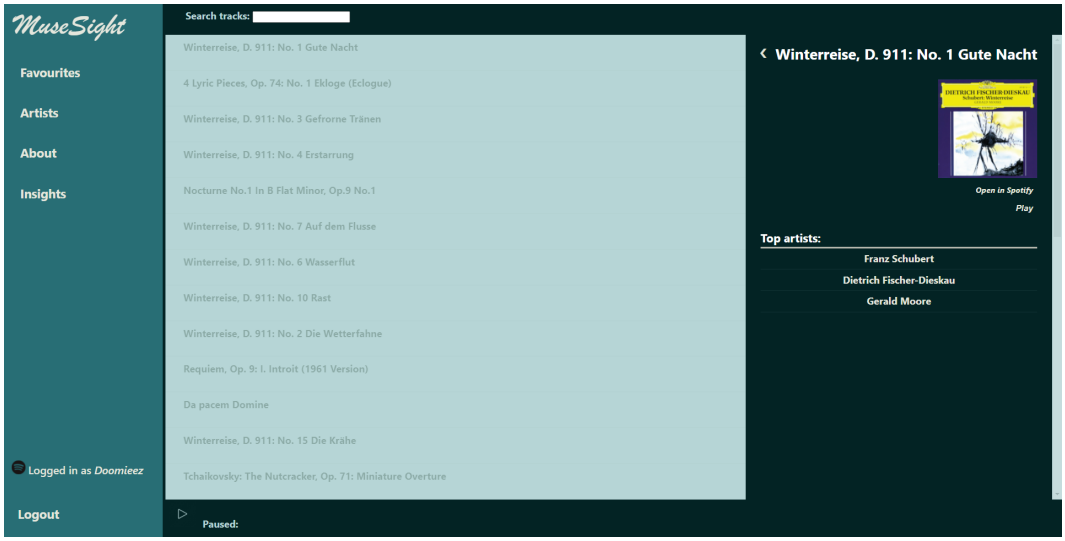
Nakon što se korisnik prijavi, na njegovom ekranu se prikaže *Favourites* sekcija (Slika 6.2). O njoj smo već mnogo govorili u prethodnim poglavljima pa znamo koja je funkcija ove sekcije i na koji je način implementirana. Stoga ćemo ovdje proći korisničke akcije i eventualno nadopuniti kod.

Slika 6.2: Prikaz popisa najslušanijih pjesama

Prijeđemo li mišem preko bilo kojeg favorita, njegova će pozadina promijeniti boju i dodatno će nam se prikazati kolika je duljina pjesme i popis umjetnika vezanih uz tu snimku. Ta dva podatka dodajemo u `favourite.js` komponenti, a potom ih prikazujemo u odgovarajućoj Handlebars komponenti. Prikazivanje i nestajanje tih podataka smo jednostavno implementirali pomoću CSS-a, postavljajući odgovarajuće `div` HTML elemente na (ne)vidljive. Također, cijeli dizajn aplikacije je napravljen pomoću CSS-a, no nećemo pridavati pažnju kodovima u CSS stilovima.

Klikom na bilo koji favorit, prelazimo na ugniježđenu putanju o kojoj smo govorili u sekciji 3.3. Pritom se otvara prozor na desnoj strani koji prikazuje detalje o tom favoritu (Slika 6.3). U tom prikazu vidimo sliku koja pripada toj pjesmi, tipku za otvaranje te pjesme u Spotifyju, tipku za reprodukciju unutar naše aplikacije i popis umjetnika. Klikom na bilo kojeg umjetnika na tom popisu otvara se stranica tog umjetnika na Spotifyju.

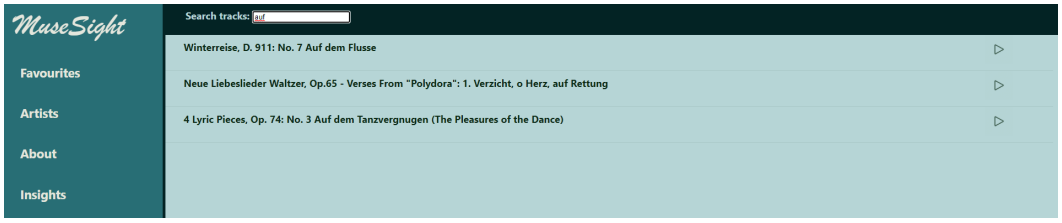
Uočimo da je u ovakvom prikazu i dalje omogućeno prolaziti po popisu favorita i klikom prijeći na drugi favorit. Također, klikom na lijevu strelicu izlazimo iz tog prikaza te je lista ponovno u prvom planu. Ulaz u i izlaz iz prikaza favorita, zajedno s odabirom novog favorita za prikaz, implementiran je servisom `favourite-clicked.js`. On pamti



Slika 6.3: Prikaz favorita

trenutno stanje prikaza: je li aktivan i što je na njemu prikazano. Svaka promjena na prikazu koju korisnik napravi prolazi direktno kroz servis pa se rezultantni prikaz iscrtava korisniku.

Ukoliko želimo omogućiti reprodukciju pjesama unutar naše aplikacije, prvo trebamo otići u Spotify player (može biti na mobilnoj, web ili desktop aplikaciji). Ondje treba kliknuti na tipku *Connect to a device* (tipka s prikazom ekrana i zvučnika) i odabati *MuseSight Player*. Nakon toga se možemo vratiti u našu aplikaciju i odabrati nekog favorita te stisnuti tipku za reprodukciju. Nakon toga možemo zaustaviti i opet nastaviti reprodukciju putem našeg svirača, a tipka i opis će se mijenjati sukladno tome.

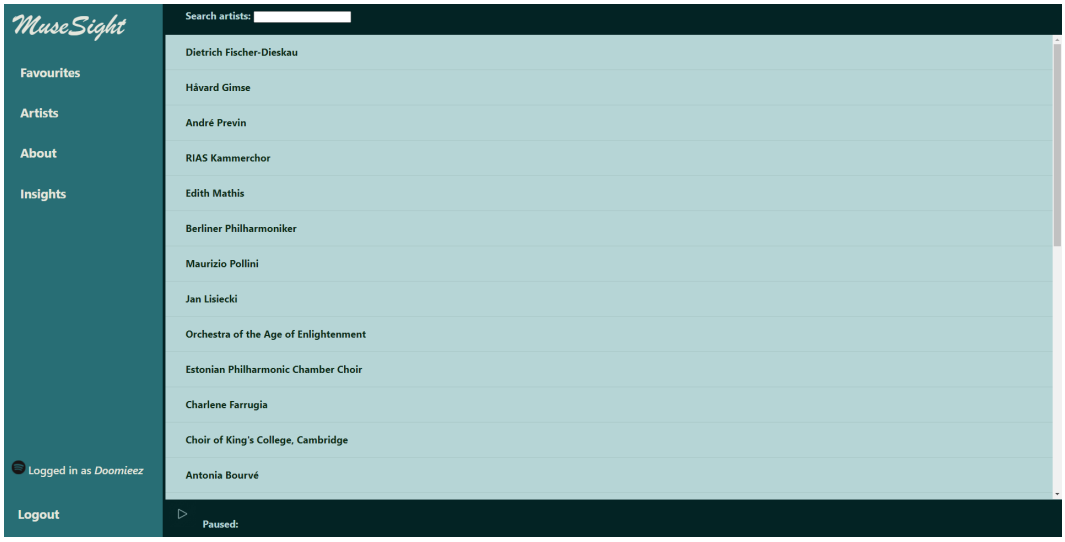


Slika 6.4: Pretraga favorita

Ukoliko nešto unesemo u polje za pretragu na vrhu ekrana, lista će se filtrirati i skraćivati, i to sa svakom promjenom unosa (Slika 6.4).

6.4 Umjetnici

Klikom na tipku *Artists* u glavnom izborniku s desne strane, nađemo se na popisu umjetnika. Uočimo odmah da taj prikaz (Slika 6.5) izgleda gotovo identično kao i popis favorita. To je vrlo očekivano jer je ideja iza tog dijela aplikacije jednaka ideji za favorite. Želimo prikazati 50 umjetnika koje je korisnik najviše slušao i detaljnije informacije o njima.



Slika 6.5: Prikaz liste umjetnika

Stoga ću odmah napomenuti da za ovu sekciju nema potrebe analizirati kod jer je analogan kodu za favorite. Imamo jednaku strukturu ruta: `artists` i `artists/artist`, Handlebars i Glimmer `artists` i `artist` komponente, istoimene predloške. Pretraga po unosu se vrši na jednak način, putem odgovarajućeg kontrolera.

Ako mišem prijeđemo preko nekog umjetnika, prikazat će se popis kategorija ili žanrova u kojima je taj umjetnik djelovao. Ako pak kliknemo na umjetnika, ponovno se otvori prozor koji prikazuje sliku, popis žanrova, link na Spotify stranicu i popis najpopularnijih pjesama (Slika 6.6). Također, klikom na pjesmu iz liste se otvara Spotify stranica te pjesme.

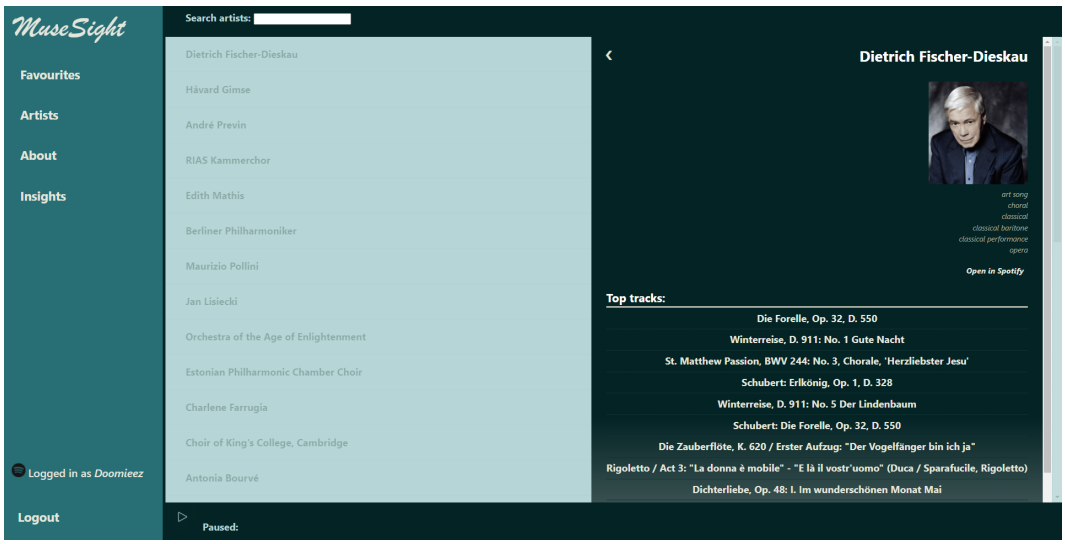
Sad kad smo prošli sve funkcionalnosti vezane za umjetnike doista možemo vidjeti da nema prevelikih razlika. Upravo zbog toga i radi čitljivosti koda, napisana je pomoćna metoda `apiCall` u datoteci `helpers/api-call.js` . U njoj generaliziramo zahtjev koji se šalje prema API-ju. Ona prima krajnju točku `endpoint`, tip metode `method`, korisnikov token pristupa `token` i limit. Kad pogledamo kod, možemo vidjeti da je doista riječ samo o slanju zahtjeva na API. Ali, ova pomoćna metoda nam sistematizira kod i smanjuje količinu ponavljanja:

```
import { helper } from '@ember/component/helper';
import fetch from 'fetch';

export async function apiCall(endpoint, method, token, limit) {
  let response = await fetch(endpoint + '?limit=' + limit.toString(), {
    method: method,
    headers: {
      Authorization: 'Bearer ' + token,
      'Content-Type': 'application/json',
    }
  });
  return await response.json();
}

export default helper(apiCall);
```

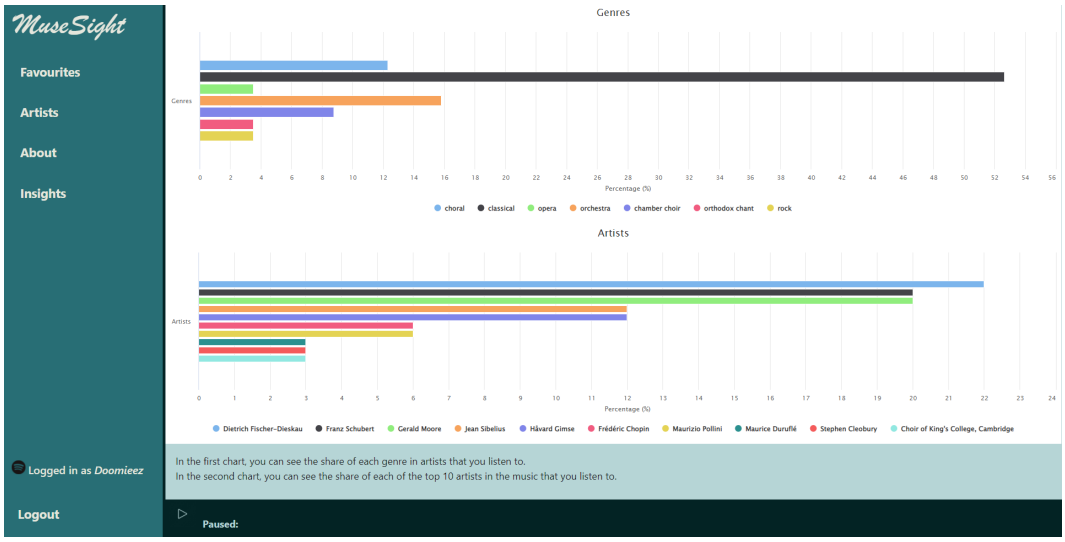
Na kraju, navigacija između umjetnika i ulaz u/izlaz iz prikaza je implementiran također jednim servisom: `services/artist-clicked.js` .



Slika 6.6: Prikaz umjetnika

6.5 Uvidi

Posljednja velika funkcionalnost je *Insights*. Otvaranjem te kartice, prikažu nam se dva grafa. Prvi graf prikazuje zastupljenost kategorija (žanrova) umjetnika koje najviše slušamo. Drugi graf nam prikazuje zastupljenost samih umjetnika u pjesmama koje slušamo (Slika 6.7).



Slika 6.7: Prikaz uvida

Prije svega, ovdje koristimo npm paket `ember-highcharts` koji služi upravo za crtanje grafova. Instaliramo ga naredbom u konzoli:

```
ember install ember-highcharts
```

Nadalje, definiramo komponentu `insights-chart.hbs`:

```
<HighCharts
  @chartOptions={{this.chartOptions}}
  @content={{this.chartData}}
/>
```

U ovom kodu, `@chartOptions` se odnosi na tip grafa (stupčasti, linijski, *pie chart* i sl.), a `@content` prima sadržaj grafa, to jest njegove podatke. Ovu komponentu dalje pozivamo u predlošku:

```
{{page-title "Insights"}}
<InsightsChart @info={{this.model}} @type=1 />

<InsightsChart @info={{this.model}} @type=2 />
```

Kao `@info` prosljeđujemo model iz rute, dok nam je `type` zapravo zastavica koja detektira je li riječ o grafu sa žanrovima (1) ili umjetnicima (2). Ovdje ne ću prolaziti kroz kod putanje `routes/insights.js` jer je vrlo dugačak, a možemo jednostavno objasniti što se ondje događa.

Naime, prvo pozivamo metodu `apiCall` kako bismo dohvatili umjetnike. Ukoliko je to već prethodno dohvaćeno, metoda se ne izvršava već se dohvate stari podatci. Nadalje sjedinjujemo neke od žanrova i formatiramo podatke, to jest svakome žanru pridružimo udio broja njegova pojavljivanja u broju pojavljivanja svih žanrova. Sličnu stvar radimo i pri dohvaćanju najslušanijih umjetnika: prolazimo po svim pjesmama, prebrojavamo umjetnike u njima i gledamo njihovu zastupljenost.

Također nećemo ulaziti u kod JS komponente jer je kod vrlo jasan. Ondje definiramo opcije za grafove (tip grafa, imena osi, nazive varijabli) i još dodatno formatiramo podatke za prikaz ovisno o tome koji je tip grafa.

Ovime dolazimo do kraja ovog poglavlja. Prošli smo sve funkcionalnosti i naglasili bitne promjene i dodatke u kodu, koji je dostupan na sljedećem linku: https://github.com/dominikmik195/musesight. Sad su sve strukture i stranice međusobno povezane i time je naša aplikacija završena.

Poglavlje 7

Zaključak

Kroz ovaj rad smo upoznali razvojni okvir Ember i način na koji on prije svega koristi JavaScript. Prošli smo kroz sve strukture: predloške, komponente, putanje, kontrolere, servise i modele. Pokazali smo kako se možemo povezati na vanjski API i sve smo to radili na primjeru naše vlastite aplikacije.

Već smo u Uvodu naveli da Ember ima reputaciju vrlo složenog i teško savladivog okvira. No, vjerujem da smo u ovome radu pokazali da to ne mora biti tako. Jasno je da Ember ima striktno određene procese. Govorili smo o načelima od kojih Ember ne odstupa nikada i umjesto da smo ih shvatili kao ograničenja, shvatili smo ih kao smjernice i preporuke. Na taj način smo Ember iskoristili na kvalitetan način; kod smo podijelili u mnogo manjih cijelina te je on postao pregledniji i jasnije strukturiran. Pritom se nismo morali opterećivati ručnim povezivanjem svih struktura jedne s drugima, nego nam je zadatak bio dodijeliti odgovarajuće ime i tog imena se držati za sve povezane strukture. Na kraju je naš zadatak kao programeru bio usredotočiti se upravo na programiranje, a ne na razvojno okruženje u kojem radimo. Na taj način, Ember nam je proces razvoja aplikacije učinio direktnijim i jednostavnijim, ali također i zanimljivim. Stoga pouzdano mogu reći da je rad u ovom okviru bio užitak te je ovim radom moje putovanje kroz Ember tek počelo.

Bibliografija

[1] *Base64*, (2021), https://en.wikipedia.org/wiki/Base64, posljednji put posjećeno 28. prosinca 2021.

[2] *Javascript*, (2021), https://en.wikipedia.org/wiki/JavaScript, posljednji put posjećeno 23. studenog 2021.

[3] *Node.js*, (2021), https://nodejs.org/, posljednji put posjećeno 21. rujna 2021.

[4] *Yarn*, (2021), https://yarnpkg.com/, posljednji put posjećeno 2. studenog 2021.

[5] *Ember.js*, (2022), https://emberjs.com/, posljednji put posjećeno 21. siječnja 2022.

[6] *Ember.js*, (2022), https://en.wikipedia.org/wiki/Ember.js, posljednji put posjećeno 11. veljače 2022.

[7] *Glimmer.js*, (2022), https://glimmerjs.com/, posljednji put posjećeno 8. veljače 2022.

[8] *Handlebars*, (2022), https://handlebarsjs.com/, posljednji put posjećeno 8. veljače 2022.

[9] *npm*, (2022), https://www.npmjs.com/, posljednji put posjećeno 13. siječnja 2022.

[10] *Promise*, (2022), https://api.emberjs.com/ember/release/classes/Promise, posljednji put posjećeno 3. veljače 2022.

[11] *Spotify for Developers*, (2022), https://developer.spotify.com/, posljednji put posjećeno 9. veljače 2022.

[12] *Spotify*, (2022), https://www.spotify.com/, posljednji put posjećeno 2. veljače 2022.

[13] *Spotify*, (2022), https://en.wikipedia.org/wiki/Spotify, posljednji put posjećeno 19. siječnja 2022.

[14] Balint Erdi, *Rock and Roll with Ember.js*, Balint Erdi, 2021.

[15] Ember tim, *Ember.js Guides*, (2022), https://guides.emberjs.com/, posljednji put posjećeno 12. veljače 2022.

[16] Jen Weber, *What are Controllers used for in Ember.js?*, (2018), https://medium.com/ember-ish/what-are-controllers-used-for-in-ember-js-cba712bf3b3e, posljednji put posjećeno 16. siječnja 2022.

Sažetak

Za razvoj web aplikacija danas postoji mnoštvo razvojnih okruženja. Ona nisu nužan alat za razvoj aplikacija, ali znatno olakšavaju cijeli proces. Sadrže razne predloške kodova koji olakšavaju sastavljanje aplikacije. Time se također omogućuje izgradnja pouzdane i relativno lako održive web aplikacije. Početkom 2000-ih godina su se ta okruženja počela koristiti za razvoj većine profesionalnih web projekata i otada su nezaobilazan dio razvoja web aplikacije.

Jedno od takvih okruženja je Ember. Ember je open-source okvir baziran na JavaScriptu. Prvi je put pušten u javnost u prosincu 2011. godine, a danas se njime grade mnoge popularne web stranice. Neki od razloga za to su korištenje Glimmer rendering engine za ubrzavanje procesa renderiranja, i Handlebars koji služi za stvaranje predložaka.

U ovom radu bavit ćemo se upravo Emberom. Proučit ćemo funkcionalnosti ovog okvira i njegovih komponenti te pomoću tih znanja sastaviti vlastitu aplikaciju.

Summary

There are many frameworks used for web development today. They aren't a necessary tool for development, but they make the whole process a lot easier. They contain different code templates which help create the app. They also enable building a reliable and maintainable web applications. In the early 2000s, these environments were started to be used to develop most of the professional web projects and have been an integral part of web application development ever since.

One of these environments is Ember. Ember is an open source framework based on JavaScript. It was first released in December 2011 and today it is being used to build many popular websites. Some of the reasons for its popularity are Glimmer rendering engine used to speed up the rendering process and Handlebars which creates templates.

In this paper, we will study the functionalities of this framework and its components and use it to compile our own application.

Životopis

Rođen sam 6. kolovoza 1997. u Zagrebu. Osnovnoškolsko obrazovanje započeo sam u OŠ Mate Lovraka, a završio 2012. godine u OŠ Vugrovec-Kašina. Iste godine upisujem matematički smjer zagrebačke V. gimnazije. Po završetku srednjoškolskog obrazovanja 2016. godine, upisujem Preddiplomski sveučilišni studij matematika na Prirodoslovno-matematičkom fakultetu u Zagrebu. Nakon završetka preddiplomskog studija, 2019. godine na istom fakultetu upisujem diplomski sveučilišni studij Računarstvo i matematika.