Serverska strana web-aplikacija i JavaScript

Novak, Lorena

Master's thesis / Diplomski rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Science / Sveučilište u Zagrebu, Prirodoslovno-matematički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:217:373409

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-04-23**

Repository / Repozitorij:

Repository of the Faculty of Science - University of Zagreb

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Lorena Novak

SERVERSKA STRANA WEB-APLIKACIJA I JAVASCRIPT

Diplomski rad

Voditelj rada:
izv. prof. dr. sc. Zvonimir Bujanović

Zagreb, 2021.

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. _____

Sadržaj

Sadržaj **iii**

Uvod **1**

1 Razvojni okvir NestJS **3**
1.1 Instalacija 4
1.2 Kreiranje projekta 4
1.3 Controllers 6
1.4 Servisi 8
1.5 Moduli 11
1.6 Middlewares 13
1.7 Guards 15
1.8 Interceptors 17
1.9 Pipes 18
1.10 Iznimke i filteri 20
1.11 Put zahtjeva kroz aplikaciju 22
1.12 TypeORM 23
1.13 Model-View-Controller 27

2 Aplikacija Ticketshop **31**
2.1 Kratki opis aplikacije 31
2.2 Baza i entiteti 31
2.3 Dijelovi aplikacije dostupni svim korisnicima 35
2.4 Registracija i prijava korisnika 40
2.5 Prava pristupa 46
2.6 Dijelovi aplikacije dostupni prijavljenim korisnicima 50
2.7 Dijelovi aplikacije dostupni administratoru 51

Bibliografija **59**

Uvod

Web-aplikacije su aplikacijski programi koji se nalaze na nekom udaljenom računalu (web-poslužitelju), a pristupa im se preko web-preglednika. Komunikacija s web-poslužiteljem odvija se pomoću protokola HTTP (eng. *HyperText Transfer Protocol*). Sastoje se od klijentske i poslužiteljske strane. Za svaku stranu postoje programski jezici u kojima su one implementirane. Za klijentsku stranu najčešće se koristi programski jezik JavaScript. Za poslužiteljsku stranu koriste se Python, Java, Ruby, PHP itd. Pojavom izvršnog okruženja Node.js koje omogućava izvršavanje koda napisanog u JavaScriptu i izvan web-preglednika, JavaScript se počeo sve više širiti i na poslužiteljsku stranu web-aplikacija što je olakšavalo posao programerima jer su morali znati samo jedan programski jezik. Međutim, razvoj web-aplikacija pomoću Node.js okvira je prilično kompliciran i usprkos svim dodatnim alatima i paketima koje su ti okviri mogli koristiti, nije riješen glavni problem kod razvoja aplikacija, a to je efikasna i jasna arhitektura aplikacije. U zadnje vrijeme pojavljuju se napredniji razvojni okviri za Node.js koji omogućavaju jednostavniju izradu web-aplikacija jer olakšavaju neke osnovne postupke kod izrade aplikacije poput pristupa bazi i organizacije koda (arhitekturu same aplikacije). Primjeri takvih naprednijih okvira su AdonisJS i NestJS. Budući da je NestJS trenutno popularniji i da je sve veći broj aplikacija koje ga koriste, u ovom diplomskom radu opisat ćemo njegove mogućnosti. U prvom dijelu diplomskog rada, opisat ćemo osnovne komponente koje su nam potrebne za izradu poslužiteljskog dijela web-aplikacije u NestJS-u, a u drugom dijelu pokazat ćemo još neke od mogućnosti koje nam nudi te opis dodatnih alata koje donosi NestJS na primjeru složenije web-aplikacije koju smo izradili.

Poglavlje 1

Razvojni okvir NestJS

Nest (NestJS) je moderni razvojni okvir za platformu Node.js koji služi za izradu pouzdanih, učinkovitih i skalabilnih web-aplikacija na poslužiteljskoj strani. Implementiran je u programskom jeziku TypeScript [7], a podržava i TypeScript i progresivni JavaScript. Nest objedinjuje elemente objektno orijentiranog programiranja (eng. *object oriented programming*), funkcijskog programiranja (eng. *functional programming*) i funkcijsko reaktivnog programiranja (eng. *functional reactive programming*). Nastao je po uzoru na Angular, no za razliku od Angulara, Nest služi za implementaciju poslužiteljske strane web-aplikacija dok je Angular okvir za klijentsku stranu web-aplikacije. U pozadini, Nest koristi Node.js i okvir Express, a može se konfigurirati i tako da koristi Fastify. Time je postignuta fleksiblnost koja omogućava korištenje logičkih dijelova aplikacija neovisno o HTTP okviru. Nest pruža dodatnu razinu apstrakcije nad tim okvirima, ali omogućava i korištenje svih modula koji su namijenjeni okvirima u pozadini pa time nudi dodatnu slobodu u razvoju aplikacije.

Slika 1.1: NestJS logo

1.1 Instalacija

Kao što smo ranije spomenuli, Nest je razvojni okvir za platformu Node.js što znači da je za instalaciju Nesta potrebno imati instaliran Node.js. Potreban nam je i npm (upravitelj proširenjima za Node.js) koji je uključen u Node.js. U trenutku pisanja ovog rada, aktualna verzija Nesta je 8.0, a aktualna verzija dostupna na službenim stranicama za Node.js je 14.18.0 s uključenom verzijom npm-a 6.14.15. što odgovara zahtjevima Nesta. Nest možemo instalirati tako da instaliramo Nest CLI. Nest CLI je sučelje naredbenog retka (eng. *command-line interface*) koje nam pomaže u inicijalizaciji, razvijanju i održavanju naše aplikacije. Instaliramo ga sljedećom naredbom:

```
npm i -g @nestjs/cli
```

Osim putem web-dokumentacije [5], upute za Nest CLI su dostupne i korištenjem naredbe

```
nest --help
```

1.2 Kreiranje projekta

Novu web-aplikaciju kreiramo kao Nest projekt koristeći naredbu:

```
nest new first-project
```

pri čemu je first-project ime projekta koji kreiramo. U narednim cjelinama razvit ćemo mali projekt pomoću kojeg ćemo demonstrirati osnovne komponente i mogućnosti Nesta. Pri kreiranju projekta, nudi nam se mogućnost odabira upravitelja paketima (eng. *package manager*). Ponuđeni su npm, yarn i pnmp pri čemu smo mi odabrali npm. Nakon kreiranja, projekt možemo otvoriti u nekom od editora poput Visual Studio Code-a. Kreiranjem novog projekta na ranije opisan način, nastaje mapa naziva `first-project` koja sadrži mape `node-modules`, `src` i `test` te još neke dodatne konfiguracijske datoteke. Unutar mape `src` kreirane su sljedeće datoteke:

- *app.module.ts*
- *app.controller.ts*
- *app.service.ts*
- *app.controller.spec.ts*
- *main.ts*

U datoteci *main.ts* implementirana je funkcija `bootstrap` u kojoj je kreirana aplikacija (instanca Nest aplikacije) i u kojoj je definirano na kojem će se portu aplikacija pokrenuti. Ako ne promijenimo vrijednost argumenta metode `listen`, pokrenut će se HTTP server na portu 3000.

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);
}
bootstrap();
```

Kod kreiranja aplikacije možemo odabrati i koji okvir želimo koristiti ovisno o tome što nam treba. Ako želimo koristiti Express, koristimo

```
const app = await NestFactory.create<NestExpressApplication>(AppModule);
```

a ako želimo Fastify

```
const app = await NestFactory.create<NestFastifyApplication>(AppModule);
```

Tip aplikacije moramo specificirati jedino u slučaju da želimo koristiti neke od metoda koje su dostupne samo za određeni okvir.

Aplikaciju pokrećemo iz naredbenog retka naredbom:

```
npm run start
```

a pristupamo joj preko web-preglednika na adresi `http://localhost:3000`. Pokrenemo li aplikaciju pomoću prethodne naredbe, promjene koje napravimo neće nam biti vidljive sve dok ju sami ponovno ne pokrenemo. Ukoliko želimo da su nam promjene vidljive odmah, aplikaciju pokrećemo naredbom:

```
npm run start:dev
```

Aplikacija se tada automatski ponovno pokreće prilikom svakog spremanja promjena. Pokrenemo li aplikaciju koju smo ranije kreirali, na ekranu u web-pregledniku trebala bi nam se prikazati poruka *"Hello world!"*.

Već iz datoteka koje su nastale unutar mape *src* možemo vidjeti koja je struktura aplikacija koje gradimo pomoću Nesta. Glavne komponente aplikacije su *controlleri*, servisi i moduli. Aplikacije funkcioniraju na način da *controlleri* primaju zahtjeve koje onda šalju na obradu servisima i nakon obrade, vraćaju odgovor. Možemo imati više *controllera* i servisa. Moduli služe da bi povezali sve te komponente u jednu cjelinu. U nastavku ovog rada vidjet ćemo malo detaljnije koja je uloga svake od navedenih komponenti te upoznati još neke specifičnosti koje nam Nest nudi.

1.3 Controllers

Controlleri su dijelovi aplikacije koji služe za primanje i obradu pristiglih zahtjeva te vraćanje odgovora korisniku. Svaki *controller* prima i obrađuje samo one zahtjeve koji su poslani njemu. Korištenjem klasa i dekoratora `@Controller()` omogućavamo Nestu da stvori mapu usmjeravanja i na taj način svaki zahtjev dolazi onome *controlleru* kojem je i poslan.

Kreiranje controllera

Kreiranjem projekta, automatski nam se stvara i datoteka *app.controller.ts*. Želimo li kreirati novi *controller* možemo to učiniti tako da kreiramo novu klasu kojoj ćemo dodati dekorator `@Controller()` ili koristeći Nest CLI i naredbu:

```
nest g controller users
```

pri čemu je `users` ime *controllera* koji kreiramo. Ako kreiramo *controller* pomoću Nest CLI, kreirat će se dvije datoteke: *users.controller.ts* i *users.controller.spec.ts*. U datoteci *users.controller.ts* nalazi se klasa `UsersController` koja implementira *controller*, a u datoteci *users.controller.spec.ts* nalaze se testovi za klasu `UsersController`.

Usmjeravanje zahtjeva

Da bismo znali koji zahtjev ide kojem *controlleru* koristimo mehanizam usmjeravanja (eng. *routing mechanism*). Svaki *controller* može obrađivati više ruta čije putanje definiramo unutar dekoratora.

```
@Controller()
export class AppController {

    @Get('hello')
    getHelloMessage: string {
        return "Hello world!";
    }

    @Get('goodbye')
    getGoodbyeMessage: string {
        return "Goodbye!";
    }
}
```

Controller iz prethodnog primjera odgovara na zahtjev GET /hello pri čemu vraća poruku *"Hello world!"* i na zahtjev GET /goodbye pri čemu vraća poruku *"Goodbye!"*. Dio putanje možemo odrediti i u dekoratoru `@Controller()` na sljedeći način:

```
@Controller('users')
export class UsersController {
    @Get('all')
    getUsers(): string {
        return "Ovdje ćemo vratiti popis svih korisnika";
    }
}
```

Navedeni *controller* prima i obrađuje zahtjeve oblika GET /users/all.

Spomenuli smo već u uvodu da se komunikacija odvija pomoću HTTP-a pa se zahtjevi šalju određenim metodama. U ranijim primjerima, *controlleri* odgovaraju na GET zahtjeve što je definirano dekoratorom `@Get()`. Ako želimo da odgovara na neku drugu metodu HTTP zahtjeva, koristimo dekoratore `@Post()`, `@Put`, `@Delete()`, itd. Svi navedeni dekoratori su iz paketa `@nestjs/common`. Na ranije opisani način definirali smo slanje zahtjeva statičkim rutama, no kod izrade aplikacije trebat će nam i rute koje sadržavaju neke parametre. Recimo da želimo poslati zahtjev gdje ćemo kao parametar poslati id korisnika čije podatke želimo dohvatiti. Zahtjev će biti oblika GET /users/id pri čemu je id neki broj. *Controller* za taj zahtjev definiramo na sljedeći način:

```
@Controller('users')
export class UsersController {
    @Get(':id')
    getUserById(): string {
        return "Ovdje ćemo vratiti podatke određenog korisnika";
    }
}
```

Kako pristupiti parametrima iz rute opisat ćemo u sljedećoj cjelini.

Request object

Controlleri imaju pristup detaljima pristiglog zahtjeva. Nest pruža pristup objektu klase `Request` iz pozadinskog okvira (ako nismo definirali koji okvir želimo koristiti, koristi se Express), a pristupamo mu na način da ga predamo kao argument funkcije pomoću dekoratora `@Req()`. Možemo pristupiti i tijelu zahtjeva (eng. *request body*), parametrima, zaglavljima (eng. *header*) i ostalim svojstvima. Pristupamo im preko objekta klase `Request`

ili koristeći dekoratore `@Body()`, `@Param()` , `@Query()`, `@Headers()`, itd. Popis svih dostupnih dekoratora možemo naći u dokumentaciji NestJS-a [5].

Ranije smo spomenuli rute koje sadrže parametre, no nismo opisali kako pristupiti tim parametrima. Koristimo li objekt klase `Request`, parametrima pristupamo na sljedeći način:

```
import { Controller, Get, Req } from '@nestjs/common';
import { Request } from 'express';

@Controller('users')
export class UsersController {
    @Get(':id')
    getUsers(@Req() request: Request) {
        let id = request.params.id;
        return "Vrati podatke od korisnika čiji je id=" + id;
    }
}
```

Možemo im pristupati i pomoću dekoratora što radimo ovako:

```
@Controller('users')
export class UsersController {
    @Get(':id')
    getUserById(@Param('id') id: number): string {
        return "Vrati podatke od korisnika čiji je id=" + id;
    }
}
```

Na isti način koristimo i ostale navedene dekoratore - predajemo ih kao argumente funkcije koja obrađuje određeni zahtjev.

Da bi aplikacija znala koji *controller* treba koristiti, moramo ga navesti unutar dekoratora `@Module()`, ali o tome ćemo pričati u poglavlju o modulima.

1.4 Servisi

Servisi su klase koje se brinu o daljnjoj obradi zahtjeva te o pohrani i dohvatu podataka. *Controlleri* primaju zahtjev i pozivaju metode iz servisa u kojima je implementirana logika koja stoji iza aplikacije.

Providers

Prije nego krenemo detaljnije proučavati servise (eng. *services*), spomenut ćemo općenitiji koncept u Nestu, a to su pružatelji usluga (eng. *providers*). To su zapravo obične klase kojima dodajemo dekorator `@Injected()`. Glavna ideja koja stoji iza njih je da oni mogu biti injektirani kao ovisnosti (eng. *injected as dependency*). Već ranije smo spomenuli da je Nest napravljen po uzoru na Angular. Jedan od koncepata koji je preuzet iz Angulara je *dependency injection*. Općenito, ovisnosti (eng. *dependencies*) su servisi i klase koje su potrebni nekoj drugoj klasi da bi mogla izvršavati svoje funkcije. *Dependency injection* je oblikovni obrazac (eng. *design pattern*) u kojem klase zahtijevaju da im se ovisnosti proslijede iz vanjskog izvora umjesto da one same kreiraju instance tih klasa. Postoje tri načina injektiranja ovisnosti: kroz konstruktor, kroz settere nekih svojstva i kroz metode odnosno argumente tih metoda [2]. Ovdje ćemo koristiti prvi način, odnosno, injektiranje kroz konstruktor. Ako imamo ovisnosti koje ne trebaju nužno biti instancirane, koristimo dekorator `@Optional()` kod njihovog injektiranja.

Kreiranje servisa

Servise kreiramo na sličan način kao i *controllere*. Kreiranjem projekta, kreira se i datoteka *app.service.ts*. Želimo li dodati novi servis, možemo ga kreirati kao klasu kojoj dodamo dekorator `@Injectable()` ili koristeći Nest CLI i naredbu

```
nest g service users
```

pri čemu je users naziv servisa koji kreiramo. Kreirane su dvije datoteke *users.service.ts* i *users.service.spec.ts*. Jednako kao i kod *controllera*, da bi aplikacija znala koji su joj servisi dostupni za korištenje, odnosno u ovom slučaju da bi znala koji su joj servisi dostupni za injektiranje, potrebno ih je navesti u modulu.

Upotreba servisa

Spomenuli smo ranije da nakon primanja zahtjeva, *controller* zove metode iz servisa ako je potrebna nekakva dodatna obrada zahtjeva. Da bi *controller* mogao pozvati metode iz servisa, potrebno je injektirati servis i to ćemo napraviti injektiranjem kroz konstruktor klase. Dakle, konstruktor *controllera* `UsersController` izgleda ovako:

```
constructor(private usersService: UsersService) {}
```

pri čemu je `UsersService` servis koji smo kreirali na ranije opisan način. Pogledajmo na koji način *controller* koristi funkcije iz servisa i kako su one implementirane u servisu na primjeru prethodno kreiranog `UsersController`-a i `UsersService`-a.

```
@Controller('users')
export class UsersController {
    constructor(private usersService: UsersService) {}

    @Get('all')
    getUsers() : User[] {
        return this.usersService.getUsers();
    }

    @Get(':id')
    async getUserById(@Param('id') id: number): Promise<User> {
        return await this.usersService.findUserById(id);
    }
}
@Injectable()
export class UsersService {
    private listOfUsers:User[];

    getUsers() : User[] {
        return this.listOfUsers;
    }

    async findUserById(id: number) : Promise<User> {
        return this.listOfUsers.find(user => user.id == id);
    }
}
```

U ovom primjeru, `User` je neka klasa ili sučelje (eng. *interface*) koje opisuje korisnika.

Asinkronost

U prethodnom primjeru vidimo da se koriste `Promise`, `await` i `async`. Asinkrone funkcije označavamo sa `async` i one uvijek vraćaju objekt tipa `Promise` u koji je upakirana vrijednost koju zapravo čekamo. `Promise` je objekt koji reprezentira zadatak koji može završiti upravo sad ili malo kasnije. Pomoću naredbe `await` zaustavimo izvršavanje programa sve dok funkcija koju se čeka ne vrati rezultat. `await` nam ujedno i otpakira vrijednost koja nam se šalje unutar `Promise` objekta [1].

1.5 Moduli

Moduli (eng. *modules*) su klase koje označavamo dekoratorom `@Module()` i koje služe za organizaciju strukture aplikacije. Kroz module povezujemo dijelove aplikacija. Već kod poglavlja o *controllerima* smo naveli da aplikacija ne zna koji *controller* koristi sve dok ga ne navedemo unutar modula. Isto vrijedi i za servise. Svaka aplikacija ima barem jedan modul, a preporuča se, radi bolje strukture, podijeliti aplikaciju na više manjih dijelova i povezati ih upravo kroz module.

Kreiranje modula

Kreiranjem projekta kreira se i glavni modul aplikacije, *app.module.ts*, u kojem možemo povezati sve dijelove. Module možemo kreirati kao običnu klasu kojoj dodamo dekorator `@Module()` ili koristeći Nest CLI. Koristimo li Nest CLI, modul kreiramo naredbom

```
nest g module users
```

pri čemu je users ime modula. Izvršavanjem ove naredbe, nastaje datoteka *users.module.ts* u kojoj se nalazi klasa `UsersModule`. Koristimo li Nest CLI za kreiranje modula, automatski nam se ažurira glavni modul u kojem se u listu uvezenih modula navodi i novonastali modul te se na taj način povezuje aplikacija. Isto vrijedi i za kreiranje *controllera* i servisa pomoću Nest CLI-a. Ako kreiramo prvo modul `UsersModule`, a zatim `UsersController` i `UsersService`, modul `UsersModule` se automatski ažurira i oni se navode pod parametre tog modula koji je već povezan s glavnim modulom. Ako prvo kreiramo `UsersController` i `UsersService`, ažurira se glavni modul i oni se navode pod njegove *controllere* i *providere*. Ako ne koristimo Nest CLI, aplikaciju moramo povezati sami.

Povezivanje aplikacije pomoću modula

Dodatni modul, *controller* i servis koji smo kreirali za *usera* želimo imati kao zasebnu manju cjelinu aplikacije. Njih povezujemo unutar `UserModule`-a na sljedeći način:

```
import { Module } from '@nestjs/common';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';
@Module({
  controllers: [UsersController],
  providers: [UsersService]
})
export class UsersModule {}
```

`UserModule` zatim povezujemo s glavnim modulom pa nam glavni modul izgleda ovako:

```
@Module({
  imports: [UsersModule],
  controllers: [AppController],
  providers: [AppService],

})
export class AppModule {}
```

Ukoliko ne koristimo `AppController` i `AppService`, možemo ih obrisati.

Pogledajmo općenito svojstva objekta koji prosljeđujemo dekoratoru `@Module()`. Objekt ima sljedeća svojstva: `providers`, `controllers`, `imports`, `exports`. Svojstvu `providers` pridružujemo listu svih *providera* koji se mogu koristiti kroz cijeli modul, dok svojstvu `controllers` pridružujemo listu svih *controllera* koji se koriste u ovom modulu. Svojstvu `exports` pridružujemo listu svih *providera* koje želimo "dijeliti" s drugim modulima koji uvezu ovaj modul. Za svojstvo `imports` navodimo listu svih modula koje želimo uvesti kako bismo mogli koristiti njihove dijeljene *providere*. Važno je naglasiti da je nemoguće koristiti *providere* koji nisu dio trenutnog modula ili koji nisu uvezeni iz nekog drugog modula. Također, važno je naglasiti i da modul može omogućiti korištenje samo svojih *providera* odnosno da ne može istovremeno uvesti i izvesti isti *provider*.

Globalni modul

Želimo li da neki skup *providera* bude dostupan cijeloj aplikaciji, definiramo globalni modul koristeći dekorator `@Global()`. Na taj način smo postigli da se svi *provideri* koje želimo dijeliti s ostalim modulima ne moraju posebno uvoziti. Globalni modul definiramo samo jednom u cijeloj aplikaciji i to najčešće bude glavni modul. Ako nema puno mjesta gdje modul želimo uključiti, bolje je izbjeći korištenje globalnog modula. Pogledajmo kako bi izgledao *users.module.ts* da definiramo da je `UsersModule` globalni modul.

```
@Global()
@Module({
  controllers: [UsersController],
  providers: [UsersService],
  exports: [UsersService]
})
export class UsersModule {}
```

Sada je `UsersService` dostupan kroz cijelu aplikaciju bez potrebe da se u nekom drugom modulu uveze `UsersModule`.

Sada kad smo prošli osnovnu strukturu aplikacije, spomenut ćemo još neke vrste komponenti koje nam Nest nudi, a koji su vrlo korisni u izradi aplikacije i organizaciji koda.

1.6 Middlewares

Middleware je funkcija ili klasa koja se izvršava prije nego što neki zahtjev dođe do *controllera* koji je zadužen za njegovo primanje i obradu. *Middleware* ima pristup objektu zahtjeva i objektu odgovora te funkciji `next()` koja se mora izvršiti nakon njega. Može izvršavati bilo kakav kod unutar sebe, može mijenjati zahtjev ili odgovor, može pozvati sljedeću funkciju (funkciju `next()`) ili prekinuti cijeli proces slanja zahtjeva. Na primjer, možemo unutar njega provjeriti je li poslan parametar i ako nije, vratiti odgovor korisniku. Možemo i mijenjati podatke koji se šalju u zahtjevu, recimo ako korisnik ne pošalje neku vrijednost, možemo u *middlewareu* postaviti tu vrijednost. Moguće je i napraviti upit na bazu i provjeriti postoji li id korisnika čije podatke želimo dohvatiti, itd. Važno je samo da ili proslijedimo zahtjev pozivajući sljedeću funkciju (`next()`) ili da prekinemo daljnje slanje zahtjeva jer će u suprotnom zahtjev "zaglaviti" i neće biti obrađen.

Implementacija i primjena

Postoje dva načina za implementaciju *middlewarea*. Prvi način je implementacija *middlewarea* u obliku klase, a drugi u obliku funkcije. Ako ga implementiramo kao klasu, moramo implementirati sučelje `NestMiddleware`. Da bismo implementirali sučelje `NestMiddleware`, potrebno je implementirati metodu `use` koja kao argumente prima zahtjev, odgovor i funkciju koja se izvršava sljedeća, odnosno, objekte klasa `Request`, `Response` i `NextFunction` uvezene iz Expressa [3]. Klasi je potrebno dodati dekorator `@Injectable()`. Klasu implementiramo unutar nove datoteke *custom.middleware.ts*.

```
@Injectable()
export class CustomMiddleware implements NestMiddleware {
    use(req: Request, res:Response, next:NextFunction()) {
        if(!req.params.id) {
            res.send("Nije poslan parametar id");
        } else {
            next();
        }
    }
}
```

Drugi način implementacije je implementacija *middlewarea* kao funkcije. Njih nazivamo funkcijskim *middlewareima* (eng. *functional middleware*). U ovom slučaju nam je *middleware* funkcija koja prima iste argumente kao i funkcija `use` iz sučelja `NestMiddleware`.

```
export function customFunctionalMiddleware(req:Request, res:Response,
  next:NextFunction()) {
    if(!req.params.id) {
        res.send("Nije poslan parametar id");
    } else {
        next();
    }
}
```

Želimo li *middleware* primjeniti globalno u aplikaciji (na sve rute), samo u *main.ts* u funkciju `bootstrap()` dodamo sljedeću liniju koda (prije pozivanja funkcije `listen()`):

```
app.use(CustomMiddleware);
```

Da bismo primijenili *middleware* na samo neke rute u aplikaciji, moramo ga navesti u modulu. Za razliku od *controllera* i *providera*, *middleware* ne navodimo unutar dekoratora `@Module()` nego modul koji ga koristi mora implementirati sučelje `NestModule`. U tom slučaju unutar modula moramo implementirati i funkciju `configure` čiji je argument objekt klase `MiddlewareConsumer`. Za objekt klase `MiddlewareConstumer` postoje metode `apply()`, `forRoutes()` i `exclude()` pomoću kojih definiramo na koje rute želimo primijeniti *middleware*. U sljedećem primjeru vidimo kako bismo implementirali metodu `consumer` u datoteci *app.module.ts* ako želimo `CustomMiddleware` primijeniti na sve rute iz `UsersControllera` osim GET /users/all.

```
export class AppModule implements NestModule {
    configure(consumer: MiddlewareConsumer) {
        consumer.apply(CustomMiddleware)
                .exclude({
                    path: 'users/all', method: RequestMethod.GET
                })
                .forRoutes(UsersController);
    }
}
```

Funkcionalni *middleware* primjenjujemo na isti način. Ako želimo primjeniti više *middlewarea*, sve ih navedemo kao parametre metode `apply()`. Navodimo ih odvojene zarezom i poredane po redu po kojem želimo da se izvršavaju.

1.7 Guards

Guard je klasa čija je uloga odlučivanje o tome hoće li neki zahtjev biti proslijeđen na daljnju obradu *controlleru* ili neće. Najčešće se koriste za autorizaciju i autentifikaciju korisnika u aplikacijama. Za razliku od *middlewarea* koji zna samo pozvati funkciju koja se izvršava sljedeća, *guardovi* imaju pristup kontekstu izvršavanja (eng. *execution context*), odnosno, instanci klase `ExecutionContext` iz paketa `@nestjs/common` i znaju točno što se izvršava nakon i na koji *controller* se zahtjev usmjerava.

Implementacija i primjena

Klasa koja implementira *guard* mora implementirati sučelje `CanActivate` i njegovu metodu `canActivate` koja kao argument prima objekt klase `ExecutionContext`. Klasi moramo dodati i dekorator `@Injectable()`. Metoda `canActivate` mora vratiti `boolean` vrijednost, odnosno, vraća `true` ako zahtjev može ići na daljnju obradu ili `false` ako se zahtjev odbija. Odgovor može vratiti i asinkrono, samo tada vraća `Promise<boolean>` ili `Observable<boolean>`. Klasa `Observable` nalazi se u paketu `RxJS`, a služi nam za obradu asinkronih zahtjeva. Klasu kojom implementiramo *guard* implementiramo u datoteci *custom.guard.ts*.

```
@Injectable()
export class CustomGuard implements CanActivate {
   canActivate(context: ExecutionContext):boolean | Promise<boolean>
    | Observable<boolean> {
        const request = context.switchToHttp().getRequest();
        const controllerName = context.getClass().name;
        if(request.session.loggedIn) {
            console.log(controllerName);
            return true;
        }
        return false;
   }
}
```

U prethodnom primjeru vidimo kako iz objekta klase `ExecutionContext` možemo doći do zahtjeva pa samim time i do tijela zahtjeva, parametara, itd. Imamo pristup i odgovoru na zahtjev te metodi koja se izvršava sljedeća `next()`. Možemo saznati i koja je ruta zahtjeva te koji *controller* će primiti i obraditi zahtjev. Rutu saznajemo koristeći metodu `getHandler()` na objektu `context`, a *controller* koristeći metodu `getClass()` kao i u primjeru prije. *Guard* iz primjera pokazuje jedan od način na koji možemo implementirati

provjeru je li korisnik prijavljen i ovisno o tome dopustiti daljnju obradu zahtjeva ili ju prekinuti.

Guard možemo primijeniti samo na neke rute, na sve rute iz nekog *controllera* ili rute iz cijele aplikacije. Primjenjujemo ga koristeći dekorator `@UseGuards()`. Dekorator prima jedan *guard* ili listu svih njih koje želimo primijeniti pri čemu su navedeni redom kojim želimo da se izvršavaju i odvojeni zarezom.

```
@Controller('users')
@UseGuards(CustomGuard)
export class UsersController {}
```

Prethodnim primjerom pokazali smo kako primijeniti *guard* na cijeli *controller*. Na isti način primjenjujemo ga i na neku određenu rutu. Možemo navesti samo ime *guarda* pa će Nest sam povezati sve koristeći *dependency injection* ili možemo odmah poslati instancu objekta.

```
@UseGuards(new CustomGuard())
```

Da bismo primijenili *guard* globalno, koristimo metodu `useGlobalGuards`. U *main.ts* u metodu `bootstrap()` dodajemo sljedeću liniju

```
app.useGlobalGuards(new CustomGuard());
```

Važno je napomenuti da ako na taj način primijenimo globalni *guard*, on ne može injektirati ovisnosti jer nije dio ni jednog modula. Da bismo to riješili, *guard* postavimo iz glavnog modula na način da ga navedemo u listu `providers`.

```
import {APP_GUARD} from '@nestjs/core';

@Module({
    providers: [
        {
            provide: APP_GUARD,
            useClass: CustomGuard
        }
    ]
})
export class AppModule {}
```

1.8 Interceptors

Interceptori su klase koje implementiraju sučelje `NestInterceptor` i označavamo ih sa dekoratorom `@Injectable()`. Oni se mogu izvršavati pri slanju zahtjeva prije nego zahtjev dođe do *controllera* ili pri slanju odgovora klijentskoj strani nakon što *controller* obradi zahtjev. Pomoću *interceptora* možemo izvršavati neki kod prije i nakon izvršavanja metode u *controlleru* zadužene za obradu zahtjeva. Možemo transformirati podatke koji se šalju kao odgovor (na primjer, možemo maknuti dio podataka koji se šalje ili dodati neke podatke poput vremena i slično). Možemo mijenjati iznimke koje su bačene iz metode koja obrađuje zahtjev te dodavati dodatnu logiku u obradi zahtjeva.

Implementacija i primjena interceptora

Već smo u uvodu ove cjeline spomenuli da *interceptori* moraju implementirati sučelje `NestInterceptor`. Da bi implementirali sučelje `NestInterceptor`, moraju implementirati metodu `intercept()` koja prima dva paramtera. Prvi parametar je `ExecutionContext`, a drugi `CallHandler`. Sučelje `CallHandler` implementira metodu `handle()` koju možemo koristiti da bismo pokrenuli izvršavanje metode iz *controllera* koja obrađuje zahtjev. Važno je samo spomenuti da se unutar metode `intercept()` mora pozvati metoda `handle()` jer se u suprotnom metoda iz *controllera* zadužena za obradu zahtjeva neće nikad izvršiti. Budući da metoda `handle()` vraća objekt tipa `Observable`, u *interceptoru* možemo raditi bilo kakve transformacije podataka koji se šalju. *Interceptor* možemo implementirati na sljedeći način:

```
@Injectable()
export class CustomInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler):
    Observable<any> {
        console.log("Prije controllera");

        return next
          .handle()
          .pipe(
            tap(() => console.log("Nakon controllera"),
          );
  }
}
```

Implementirani *interceptor* ispisuje poruku ”Prije controllera” i zatim poziva metodu `handle()`. Metoda `handle()` poziva metodu iz *controllera* koja je zadužena za obradu

pristiglog zahtjeva te nakon njenog izvršavanja ispisuje poruku ”Nakon controllera”. *Interceptore* primijenjujemo na rute koristeći dekorator `@UseInterceptors()`. Možemo ih primijeniti na pojedinu rutu, na sve rute iz jednog *controllera* ili na sve rute iz aplikacije. Ako ga primijenjujemo globalno, možemo korisititi metodu `useGlobalInterceptor()` na instanci objekta Nest aplikacije. Dakle, u metodu `bootstrap()` dodajemo liniju:

```
app.useGlobalInterceptors(new CustomInterceptor());
```

Primijenjujemo li ga na taj način, nećemo moći koristiti *dependency injection* kao ni kod *guardova*. Drugi način da ga primijenimo globalno je tako da ga navedemo u modulu `AppModule` unutar dekoratora `@Module()` pod `providers`.

```
{
  provide: APP_INTERCEPTOR,
  useClass: CustomInterceptor,
}
```

1.9 Pipes

Pipes su klase koje služe za transformaciju i validaciju podataka koje metoda u *controlleru*, koja je zadužena za tu rutu, prima kao argument. Izvršavaju se netom prije nego što se krene izvršavati metoda iz *controllera*. Nest ima osam već ugrađenih *pipeova*, a možemo implementirati i nove ako nam je potrebno. Ako implemetiramo novi *pipe*, potrebno ga je označiti dekoratorom `@Injectable()` i mora implementirati sučelje `PipeTransform`. Ugrađeni *pipeovi* su: `ParseIntPipe`, `ParseBoolPipe`, `ParseUUIDPipe`, `ParseEnumPipe`, `ParseFloatPipe`, `ParseArrayPipe`, `ValidationPipe` i `DefaultValuePipe` te se nalaze u paketu `@nestjs/common`.

Pipe možemo primijeniti globalno ili na neku određenu rutu. Opisat ćemo sada kako ga primijeniti na rutu. Imali smo ranije metodu `getUserById` za koju smo rekli da kao argument prima id koji je broj. Da bismo provjerili šalje li se zaista broj u zahtjevu, koristimo `ParseIntPipe`. *Pipe* dodajemo kraj argumenta na koji ga želimo primijeniti. U ovom slučaju, metoda `getUserById` u *users.controller.ts* izgleda ovako:

```
@Get(':id')
async getUserById(@Param('id', ParseIntPipe) id: number):
    Promise<User>{
        return await this.usersService.findUserById(id);
}
```

Ako je zahtjevom poslan broj kao parametar, metoda `getUserById` će se krenuti izvršavati. Ako nije poslan broj, *pipe* će baciti iznimku (eng. *exception*) i neće doći do daljnje obrade

zahtjeva. Možemo poslati ili samo naziv klase kao u gornjem primjeru pa će Nest povezati sve ili instancu klase (`new ParseIntPipe()`).

Želimo li neki *pipe* definirati globalno, koristimo metodu `useGlobalPipes` na instanci objekta Nest aplikacije. Dakle, u *main.ts* u metodu `bootstrap()` dodajemo

```
app.useGlobalPipes(new ParseIntPipe());
```

Javlja se opet isti problem koji smo spomenuli i kod *guardova*. Da bi *dependency injection* bio moguć, moramo *pipe* definirati direktno u glavnom modulu kao `provider`. Dakle, u glavni modul `AppModule` u dekorator pod `providers` dodajemo sljedeće:

```
{
    provide: APP_PIPE,
    useClass: ValidationPipe,
}
```

pri čemu je `APP_PIPE` iz paketa `@nestjs/core`.

Ako trebamo nekakvu transformaciju ili validaciju koja nije ugrađena, možemo sami implementirati svoj *pipe*. Već smo ranije spomenuli da klasa koja predstavlja *pipe* mora implementirati sučelje `PipeTransform`. Sučelje `PipeTransform<T,R>` je generičko sučelje u kojem `T` predstavlja tip ulazne vrijednosti, a `R` predstavlja tip u koji želimo pretvoriti ulaznu vrijednost. Kod implementacije sučelja, potrebno je implementirati metodu `transform()` koja kao argumente prima podatak koji treba validirati i metapodatke o toj vrijednosti. Ako nam je potrebna samo validacija, *pipe* implementiramo ovako:

```
@Injectable()
export class CustomPipe implements PipeTransform {
  transform(value: any, metadata: ArgumentMetadata) {
    if(typeof value == "string" && value.length < 10) {
        return value;
    } else {
        throw new BadRequestException("Greška kod validacije.");
    }
  }
}
```

Ako želimo i transformaciju podataka, implementiramo ga ovako:

```
@Injectable()
export class CustomPipe implements PipeTransform<T,R>{
  transform(value: T, metadata: ArgumentMetadata) {
    let convertedValue = convertToTypeR(value);
```

```
    return convertedValue;
  }
}
```

Klase možemo spremiti u datoteku *custom.pipe.ts*.

1.10 Iznimke i filteri

Iznimke (eng. *exceptions*) su događaji koji prekidaju normalni tijek izvršavanja programa. Filteri za iznimke (eng. *exception filters*) su klase koje se brinu za hvatanje iznimke koja se dogodila i omogućuju nam kontrolu nad iznimkama i modificiranje poruke koju vraćamo korisniku. Nest ima ugrađene iznimke koje nasljeđuju klasu `HttpException`, uključujući i nju, a dio su paketa `@nestjs/common`. Neke od iznimki su: `BadRequestException`, `UnauthorizedException`, `NotFoundException`, `ForbiddenException` i brojne druge. Također, Nest ima ugrađen i sustav filtera za hvatanje iznimki koji hvata sve iznimke tipa `HttpException`. Taj sustav primijenjuje se globalno na cijelu aplikaciju. Ukoliko dođe do iznimke koju ne prepoznaje, korisniku se šalje poruka *"Internal server error"* sa statusom 500.

Iznimke

Kod bacanja iznimke, možemo sami konstruktoru iznimke proslijediti poruku i status. Konstruktor iznimke prima dva argumenta - odgovor (eng. *response*) i status HTTP iznimke. Odgovor može biti ili string ili JSON objekt koji kao svojstva ima poruku i status iznimke. Na primjer, bacimo li sljedeću iznimku

```
throw new HttpException("Niste prijavljeni", HttpStatus.UNAUTHORIZED);
```

korisnik dobiva natrag poruku

```
{
    "statusCode":401,
    "message":"Niste prijavljeni"
}
```

Možemo kreirati i svoje iznimke. Želimo li kreirati novu iznimku moramo implementirati klasu koja će nasljediti klasu `HttpException`. U konstruktoru novokreirane iznimke moramo pozvati konstruktor nadklase, tj. konstruktor od klase `HttpException` i proslijediti argumente koje smo prije opisali. Klasu iz primjera implementiramo u datoteci *unauthorized.exception.ts*.

```
export class UnauthorizedException extends HttpException {
    constructor() {
        super("Niste prijavljeni", HttpStatus.UNAUTHORIZED);
    }
}
```

Novu iznimku bacamo naredbom

```
throw new UnauthorizedException();
```

Filteri

Kao što možemo kreirati novu iznimku, možemo kreirati i novi filter za iznimke. Nest nam nudi filter koji hvata sve iznimke tipa `HttpException`, no želimo li, na primjer, vratiti drugačiju poruku ili preusmjeriti korisnika na drugu rutu, možemo to implementirati u novom filteru. Filter kreiramo kao klasu koja implementira sučelje `ExceptionFilter` i označavamo ga dekoratorom `@Catch()` u koji upisujemo naziv iznimke koju želimo uhvatiti. Ako želimo napraviti univerzalni filter koji će uhvatiti sve iznimke, dekoratoru ne predajemo nikakav parametar. Kod implementacije sučelja `ExceptionFilter` potrebno je implementirati metodu `catch()` čiji su argumenti `exception` koji smo uhvatili i `ArgumentHost` objekt. Drugi argument služi nam za dohvaćanje zahtjeva i odgovora. `ExceptionFilter`, dekorator `@Catch()` i `ArgumentHost` dio su paketa `@nestjs/common`.

```
@Catch(UnauthorizedException)
export class UnauthorizedExceptionFilter implements ExceptionFilter {
    catch(exception: UnauthorizedException, host: ArgumentsHost) {
        const ctx = host.switchToHttp();
        const response = ctx.getResponse<Response>();
        const status = exception.getStatus();

        response.redirect('/goodbye');
    }
}
```

U ovom primjeru, u datoteci *unauthorized-exception.filter.ts* kreirali smo filter za iznimke tipa `UnauthorizedException`. Kada uhvati iznimku, on umjesto da prekine izvođenje programa i vrati poruku o iznimci, preusmjeri korisnika na drugu rutu (u ovom slučaju na rutu GET /goodbye).

Filtere primjenjujemo pomoću dekoratora `@UseFilters()`. Možemo ih primijeniti na rutu, na neki *controller* tj. na sve rute koje on obrađuje ili na cijelu aplikaciju. Ako primijenjujemo na rutu ili na *controller*, koristimo dekorator.

```
@Controller('users')
export class UsersController {
    @Get('all')
    @UseFilters(UnauthorizedExceptionFilter)
    getUsers(@Req() request: Request) {
        throw new UnauthorizedException();
    }
}
```

Ako primijenjujemo globalno, koristimo metodu `useGlobalFilters()` unutar metode `bootstrap()`, tj. dodamo sljedeću liniju:

```
app.useGlobalFilters(new UnauthorizedExceptionFilter());
```

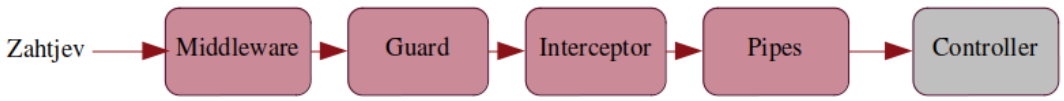
Ako želimo primijeniti više filtera, samo ih redom kojim želimo da se izvrše navedemo unutar dekoratora ili metode.

Ovdje nam se javlja isti problem vezan uz *dependency injection* kao i ranije. Da bismo to izbjegli, filtere možemo globalno primijeniti i iz glavnog modula na način da u dekorator `@Module()` pod `providers` dodamo

```
{
  provide: APP_FILTER,
  useClass: UnauthorizedExceptionFilter,
}
```

1.11 Put zahtjeva kroz aplikaciju

U ovom poglavlju, prošli smo kroz osnovnu strukturu Nest aplikacije i kroz dijelove koji nam olakšavaju implementaciju i organizaciju aplikacije. Preostalo je samo pogledati kojim redoslijedom se svi oni izvršavaju.

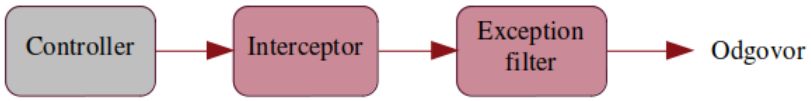


Slika 1.2: Put zahtjeva do *controller*a

Slika 1.2 prikazuje kroz koje dijelove i kojim redoslijedom putuje dolazni zahtjev. Svaki od tih dijelova može prekinuti daljnje slanje zahtjeva. Spominjali smo da se svaki od tih dodatnih dijelova može primijeniti na određenu rutu, *controller* ili globalno. Kod

izvršavanja svakog od njih, prvo se izvršava onaj globalni, zatim onaj koji je primijenjen na sve rute iz nekog *controllera* i na kraju onaj koji je primijenjen samo na rutu. Dakle, prvo će se izvršiti globalni *middleware*, zatim onaj koji je primijenjen samo na rutu. Nakon toga se izvršava globalni *guard* pa onda onaj koji se odnosi na cijeli *controller* i na kraju onaj koji se odnosi samo na tu rutu i tako dalje redom.

Nakon što zahtjev stigne do *controllera*, on ga obrađuje. Može, ali i ne mora, zvati funkcije iz servisa. Nakon obrade, *controller* šalje natrag odgovor. Na slici 1.3 možemo

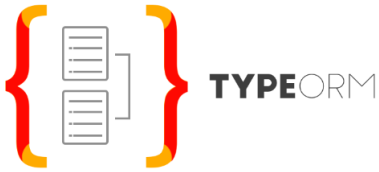


Slika 1.3: Slanje odgovora

vidjeti kroz koje dijelove odgovor prolazi prije nego što stigne do korisnika. Jedina razlika je što ovdje prolazi prvo kroz one dijelove koji su primijenjeni na rutu, zatim kroz one koji su primijenjeni na *controllere* i tek onda kroz globalne. Dakle, prvo će proći kroz *interceptor* koji je primijenjen samo na tu rutu, zatim kroz *interceptor* koji se odnosi na cijeli *controller* i na kraju kroz globalni. Nakon toga prolazi još kroz *exception filter* istim redoslijedom.

1.12 TypeORM

U ovoj cjelini, bavit ćemo se povezivanjem aplikacije s bazom podataka. Nest je fleksibilan što se tiče rada s bazom podataka tako da se može lako povezati s bilo kojom bazom, bila ona SQL baza ili NoSQL baza. Možemo koristiti bilo koju biblioteku (eng. *library*) za integraciju s bazom koje nudi Node.js ili neki ORM (eng. *Object Relational Mapper*), na primjer MikroORM, Sequelize, Knex.js, TypeORM i mnoge druge. TypeORM je trenutno jedan od najpopularnijih ORM-ova za TypeScript pa ćemo u nastavku opisati spajanje i rad s MySQL bazom pomoću TypeORM-a.



Slika 1.4: TypeORM logo

Nest ima paket `@nestjs/typeorm` koji služi za integraciju i sa SQL i s NoSQL bazama. Da bismo ga mogli koristiti za integraciju s MySQL bazom, instaliramo ga naredbom:

```
npm install --save @nestjs/typeorm typeorm mysql2
```

Nakon instalacije, možemo aplikaciju povezati s bazom. Kreiramo novu datoteku *ormconfig.json* u glavnom direktoriju projekta, odnosno, u mapi *`first-project`*. U tu datoteku upisujemo podatke za spajanje s bazom.

```
{
    "type": "mysql",
    "host": "localhost",
    "port": 3306,
    "username": "root",
    "password": "root",
    "database": "users",
    "entities": ["dist/**/**.entity{.ts,.js}"]
 }
```

Navedena su osnovna konfiguracijska svojstva za spajanje s bazom. Postoji još nekoliko dodatnih svojstava koje možemo postaviti, na primjer `retryAttempts` s kojim postavljamo broj pokušaja spajanja s bazom, zatim `autoLoadEntities` pomoću kojeg se entiteti automatski učitavaju, itd. Važno je spomenuti svojstvo `synchronize` koje, ako je postavljeno na `true`, kod svakog pokretanja sinkronizira podatke s bazom. Treba biti posebno oprezan na kojim podacima i s kojom bazom se koristi jer može doći do promjene ili gubitka nekih podataka. Postoje i druga svojstva o kojima se više može naći u dokumentaciji TypeORM-a [6].

Nakon što smo kreirali konfiguracijsku datoteku za spajanje, moramo uključiti modul `TypeOrmModule` u glavni modul aplikacije `AppModule`. Navodimo ga u dekoratoru `@Module()` pod `imports`.

```
@Module({
  imports: [UsersModule, TypeOrmModule.forRoot()],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Drugi način na koji smo mogli povezati aplikaciju s bazom je da objekt s konfiguracijskim svojstvima predamo i kao argument metode `forRoot()` umjesto kreiranja datoteke *ormconfig.json*. U nastavku ćemo pokazati dva načina za dohvaćanje podataka iz baze. Najprije moramo kreirati entitete. Entitet (eng. *entity*) je klasa u koju se mapiraju podaci iz baze. Entitete označavamo dekoratorom `@Entity()`. Kreirat ćemo entitet za *usera* u novoj datoteci *user.entity.ts* koju možemo spremiti u mapu `users` u kojoj su nam i *users.controller.ts*, *users.service.ts* i *users.module.ts*.

```
@Entity()
export class User {
    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    name: string;

    @Column()
    surname: string;
}
```

U konfiguracijskoj datoteci smo već rekli TypeORM-u koja je putanja do svih entiteta, tako da on zna da entitet `User` postoji. Da to nismo naveli, morali bismo navesti nakon što kreiramo entitet jer u suprotnom TypeORM ne bi znao da taj entitet postoji. Mogli smo dodati i svaki entitet pojedinačno na način da umjesto putanje koju smo napisali, navedemo sve entitete.

```
entities: [User]
```

Jedan od načina za dohvaćanje podataka iz baze je korištenje repozitorija (eng. *repository*). Da bismo to mogli, prvo u modulu u kojem želimo koristiti taj repozitorij, moramo navesti da koristimo `TypeOrmModul` i moramo navesti za što ga koristimo pomoću metode `forFeature()` kojoj kao argument predamo entitet čiji repozitorij želimo uključiti. Dakle, u našem slučaju, repozitorij želimo koristiti u `UsersModule`-u pa ga tu i navodimo unutar dekoratora `@Module()`:

```
@Module({
  imports: [TypeOrmModule.forFeature([User])],
  controllers: [UsersController],
  providers: [UsersService],
  exports: [UsersService]
})
export class UsersModule {}
```

Sad kad smo ga definirali unutar modula, možemo `UserRepository` injektirati u konstruktoru klase `UsersService` koristeći dekorator `@InjectRepository()`. U idućem primjeru pokazat ćemo kako injektirati repozitorij i kako napraviti upite pomoću njega unutar `UsersService`.

```
@Injectable()
export class UsersService {
    constructor(
        @InjectRepository(User)
        private userRepository: Repository<User>) {}

    async getUsers() : Promise<User[]> {
        return this.userRepository.find();
    }

    async findUserById(id: number) : Promise<User> {
        return this.userRepository.findOne(id);
    }
}
```

Drugi način na koji možemo izvršavati upite je koristeći `QueryBuilder`. U tom slučaju nam ne treba `UserRepository`, no vidjet ćemo da je dohvaćanje podataka pomoću repozitorija jednostavnije i elegantnije.

```
@Injectable()
export class UsersService {

    async getUsers() : Promise<User[]> {
        return await createQueryBuilder(User, "user")
                        .getMany();
    }
```

```
    async findUserById(id: number) : Promise<User> {
        return await createQueryBuilder(User, "user")
                        .select()
                        .where("user.id= :userId", {userId: id})
                        .getOne();
    }
}
```

Sve ovo navedeno samo su neke od mnoštva mogućnosti koje nudi TypeORM. Postoji još načina za komunikaciju s bazom. Također, neke od značajnijih mogućnosti koje nam nudi TypeORM su migracije, transakcije i definiranje relacija među entitetima. Nešto više o definiranju relacija među entitetima reći ćemo u cjelini (2.2) kod opisa implementacije aplikacije. Također, više o svemu može se pronaći u dokumentaciji TypeORM-a [6].

1.13 Model-View-Controller

MVC (eng. *Model-View-Controller*) je arhitekturalni obrazac koji razdvaja aplikaciju u tri osnovna dijela: model, *view* i *controller*. *View* je dio koji služi za prikaz aplikacije i omogućavanje interakcije korisnika s aplikacijom. *Controller* služi za obradu pristiglih zahtjeva nastalih tom interakcijom i slanje odgovora natrag klijentu, dok je u modelu implementirana cijela logika aplikacije i interakcija s bazom. *Controller* je zapravo dio koji povezuje model i *view*. U našem slučaju, model bi predstavljali servisi.

Da bismo izgradili aplikaciju u skladu s MVC obrascem, nedostaje nam još samo komponenta *view*. Prvo moramo instalirati upravitelj HTML predlošcima (eng. *template engine*) pomoću kojeg ćemo renderirati izgled naše aplikacije. Jezik koji ćemo koristiti za kreiranje predložaka (eng. *templating language*) je *Handlebars.js* [4].

Slika 1.5: *Handlebars.js* logo

Instaliramo ga sljedećom naredbom:

```
npm install --save hbs
```

Da bi aplikacija znala koji *template engine* koristi za renderiranje izgleda, moramo malo prilagoditi metodu `bootstrap()` u datoteci *main.ts*. Rekli smo da Nest u pozadini koristi Express ako mu ništa drugo ne zadamo, zato ćemo ga i mi ovdje koristiti i kreirati Express aplikaciju. Metoda `bootstrap()` nam izgleda ovako:

```
async function bootstrap() {

  const app = await NestFactory.create<NestExpressApplication>(
  AppModule);

  app.useStaticAssets(join(__dirname, '..', 'public'));
  app.setBaseViewsDir(join(__dirname, '..', 'views'));
  app.setViewEngine('hbs');

  await app.listen(3000);
}
```

Aplikacija sada zna da koristi hbs *template engine* za renderiranje izgleda, da su svi predlošci za izgled spremljeni u mapi *views*, a svi statički podaci u mapi *public*. Unutar mape *views* kreiramo *users.hbs* u kojem je napisan HTML koji želimo da se renderira:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>First project</title>
  </head>
  <body>
    Ime: {{ user.name }} <br>
    Id: {{ user.id }
  </body>
</html>
```

Da bi se taj predložak renderirao, moramo u *controlleru* dodati dekorator `@Render()` onoj metodi kojoj želimo da renderira izgled. Unutar dekoratora pišemo ime predloška koji želimo renderirati. Imena svojstava objekta koji vraća metoda koja renderira izgled moraju biti jednaka imenima varijabli u predlošku. Podaci koje vraća *controller* bit će proslijeđeni predlošku koji će zatim generirati HTML tako da sve što je unutar dvostrukih vitičastih zagrada {{ }} zamijeni vrijednostima koje je poslao *controller*. Dobiveni HTML bit će poslan korisniku kao odgovor.

```
@Controller('users')
export class UsersController {
    @Get(':id')
    @Render('users')
    getUserById(@Param('id') id: number) : any {
        let userData = this.usersService.findUserById(id);
        return {user: userData};
    }
}
```

Sada kad smo prošli osnovne komponente Nest aplikacije i kada smo se upoznali s dodatnim mogućnostima koje nam Nest nudi za izgradnju aplikacije i bolju organizaciju strukture, možemo pogledati sve to na primjeru aplikacije koju smo izgradili.

Poglavlje 2

Aplikacija Ticketshop

U ovom poglavlju opisat ćemo aplikaciju koju smo izgradili kako bismo demonstrirali proces izrade i strukturu nešto složenijeg projekta, kao i dodatne mogućnosti koje nam Nest nudi.

2.1 Kratki opis aplikacije

Web-aplikacija koju smo izgradili nalik je web-aplikacijama poput Eventima ili Ticketshopa koje služe za prodaju ulaznica za razne događaje. Web-aplikacija nam nudi mogućnost pretraživanja događaja i kupnje ulaznica za te događaje. Imamo tri tipa korisnika u aplikaciji: neprijavljeni korisnik ili gost, korisnik koji je prijavljen i administrator. Neprijavljeni korisnik može pregledavati i pretraživati događaje te kupiti karte za njih. Nudi mu se i mogućnost registracije. Korisnik koji je prijavljen može sve što može i gost, a nudi mu se i dodatna mogućnost da vidi svoj profil. Na svom profilu korisnik može vidjeti i mijenati svoje podatke te može vidjeti sve događaje za koje je kupio karte. Administrator ima najviše mogućnosti u aplikaciji. On može sve što može i korisnik, ali može i kreirati nove događaje, uređivati postojeće te ih brisati. Kod kreiranja događaja, može dodavati nove izvođače i nove tipove događaja. Može vidjeti i popis svih korisnika koji su registrirani te može vidjeti njihove profile i mijenjati ulogu u aplikaciji (može nekog postaviti za administratora ili mu maknuti ta prava).

2.2 Baza i entiteti

Najprije ćemo ukratko opisati bazu i entitete koje koristimo u aplikaciji i relacije među njima. Već smo ranije spomenuli da nam TypeORM nudi mogućnost definiranja relacija među entitetima, a ovdje ćemo to i demonstrirati. U bazi imamo tablice `events`, `users`,

performers, types, performance i audience. Tablica events služi za opis događaja te sadrži atribute id događaja, ime, id tipa događaja, datum, vrijeme i trajanje događaja, grad i mjesto održavanja događaja, kratki opis događaja, dostupan broj karata, broj prodanih karata i cijenu karte. Tablica types služi za opis tipa događaja i sadrži samo atribute id tipa i naziv tipa događaja. Svaki događaj može biti samo jednog tipa te može postojati više događaja istog tipa. Tablica users opisuje korisnika. Sadrži id korisnika, njegovo ime, prezime, korisničko ime, datum rođenja, email adresu i ulogu u aplikaciji (može biti *user* ili *admin*). Svaki korisnik može kupiti karte za više događaja te za svaki događaj karte može kupiti više korisnika. Podatke o toj relaciji spremamo u tablicu audience. Ona sadrži atribute id događaja i id korisnika te broj karata koje je neki korisnik kupio. Imamo i tablicu performers koja sadrži podatke o izvođačima, odnosno, sadrži atribute id izvođača i njegovo ime. Svaki izvođač može nastupati na više događaja te na istom događaju može sudjelovati više izvođača. Za sve navedene tablice, kreirali smo entitete koje koristimo u aplikaciji. Pogledat ćemo entitet Events koji opisuje događaje i pokazat ćemo način na koji se kreiraju relacije među entitetima. Spomenuli smo dvije relacije koje ćemo dodati u entitet Events. Relaciju između korisnika i događaja, dodat ćemo entitetu Users. Relacije koje dodajemo u entitet Events su relacija između događaja i izvođača i relacija između događaja i tipa događaja.

Relacija između događaja i tipa događaja

Prvo ćemo pokazati kako smo kreirali relaciju između događaja i tipa događaja. Rekli smo da je svaki događaj samo jednog tipa i da više događaja može biti istog tipa. Opisana relacija između entiteta Events i Types je mnogo naprema jedan (eng. *many-to-one*). Relaciju možemo definirati i u entitetima koristeći dekorator @ManyToOne() iz paketa typeorm. Definirat ćemo ju unutar entiteta Events. Rekli smo da se u tablici events sprema id tipa događaja. To je ujedno i strani ključ na tablicu types. Da bismo povezali entitete, unutar dekoratora @ManyToOne() navodimo s kojim entitetom povezujemo. Potreban nam je i dekorator @JoinColumn() koji definira koja strana relacije sadrži strani ključ. Ako želimo da se entiteti spoje samo po stranom ključu, dekoratoru @JoinColumn() nije potrebno predati naziv atributa po kojem ih spajamo. Ako želimo da se spoje po nekom drugom atributu, dekoratoru predajemo ime tog atributa kao vrijednost svojstva referencedColumnName. U našem slučaju to izgleda ovako:

```
@ManyToOne(type => Types)
@JoinColumn({ referencedColumnName: "id" })
type: Types;
```

Dakle, u varijablu type smo spremili objekt entiteta Types koji je pridružen ovom događaju. Svojstvu referencedColumnName pridružili smo naziv stupca tablice types s kojim spa-

jamo. Budući da je već u tablici `events` `typeId` strani ključ na tablicu `types`, nismo morali posebno reći po kojem stupcu spajamo, jer će TypeORM automatski pretpostaviti da je ime stupca jednako konkatenaciji imena varijable i "Id".

Relacija između događaja i izvođača

Rekli smo ranije da svaki događaj može imati više izvođača te da svaki izvođač može nastupati na više događaja. Relacija između događaja i izvođača je mnogo naprema mnogo (eng. *many-to-many*). U entitetima ju definiramo pomoću dekoratora `@ManyToMany()` i `@JoinTable()`. Unutar dekoratora `@ManyToMany()` navodimo s kojim entitetom je u relaciji, a unutar dekoratora `@JoinTable()` navodimo ime tablice koja spaja ta dva entiteta te navodimo po kojim atributima ih spaja.

```
@ManyToMany(type => Performers)
    @JoinTable({
        name: "performance",
        joinColumn: {
            name: "eventId",
            referencedColumnName: 'id'
        },
        inverseJoinColumn: {
            name: 'performerId',
            referencedColumnName: 'id'
        }
    })
    performers: Performers[];
```

U našem slučaju, podaci koji opisuju relaciju su u tablici `performance`. Tablica `performance` ima dva atributa: `eventId` i `performerId` koji su ujedno i strani ključevi na tablice `events` i `performers`. Unutar `joinColumn` definirali smo na temelju kojih atributa spajamo entitete `Events` i `Performance`. Atribut `eventId` iz entiteta `Events` spajamo s atributom `id` iz entiteta `Performance`. Unutar `inverseJoinColumn` definirali smo kako spajamo `Performance` i `Performers`.

Pogledajmo kako nam izgleda entitet `Events` s uključenim opisanim relacijama i s navedenim ostalim svojstvima.

```
@Entity()
export class Events {
    @PrimaryGeneratedColumn({type: 'int'})
    id: number;
```

```
@Column()
name: string;

@Column()
typeId: number;

@Column()
location: string;

@Column()
city: string;

@Column()
date: Date;

@Column()
duration: number;

@Column()
description: string;

@Column()
tickets: number;

@Column()
soldTickets: number;

@Column()
ticketPrice: number;

@ManyToOne(type => Types)
@JoinColumn({ referencedColumnName: "id" })
type: Types;

@ManyToMany(type => Performers)
@JoinTable({
    name: "performance",
    joinColumn: {
```

```
            name: "eventId",
            referencedColumnName: 'id'
        },
        inverseJoinColumn: {
            name: 'performerId',
            referencedColumnName: 'id'
        }
    })
    performers: Performers[];
}
```

Kad već pričamo o relacijama, spomenimo samo usput još jednu važnu stvar. Ranije opisani način je način za spajanje tablica ako nam je baza već kreirana. TypeORM daje mogućnost da na temelju entiteta, definiranih kodom poput gore navedenog, automatski kreira potrebne tablice u bazi podataka. Tablice koje opisuju relacije se također same kreiraju i dodaju se stupci koji služe kao strani ključevi za drugu stranu relacije. Imena im TypeORM postavlja automatski po određenim pravilima, a ako želimo drugačija imena, onda ih sami postavljamo kao što smo i ranije napravili. Više o svemu tome dostupno je u dokumentaciji TypeORM-a [6].

2.3 Dijelovi aplikacije dostupni svim korisnicima

U ovom dijelu proći ćemo kroz dijelove aplikacije koji su dostupni svim korisnicima bili oni prijavljeni ili ne.

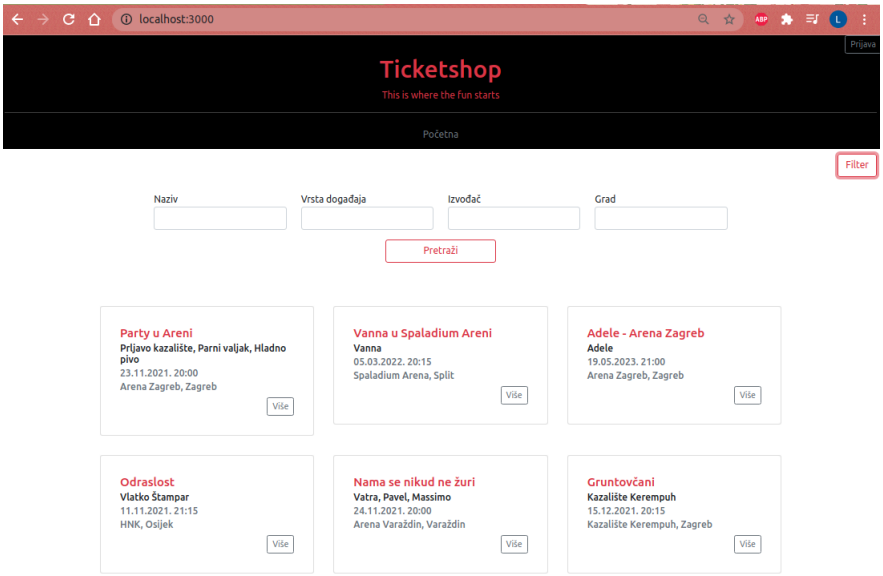
Struktura aplikacije

Najprije ćemo proći kroz strukturu aplikacije. Aplikaciju smo podijelili na četiri osnovna dijela ovisno o funkcijama koje obavljaju. Imamo glavni modul `AppModule` koji povezuje cijelu aplikaciju i `AppController` koji se brine za renderiranje početne stranice i za vraćanje podataka koji su upisani u *session*. Zatim imamo dio koji se brine za prijavu i registraciju korisnika, a to je `LoginModule` i `LoginController`. Budući da su svi zahtjevi koji dolaze na `LoginController` vezani za korisnika, on sve zahtjeve dalje šalje na obradu servisu `UsersService`. Dio koji se brine za obradu svih zahtjeva vezanih za korisnika (promjena podataka, validacije, kreiranje novog korisničkog računa,...) je `UsersController`, `UsersModule` i ranije spomenuti `UsersService`. I na kraju, dio koji se brine o svim zahtjevima vezanima uz događaje čine `EventsController`, `EventsModule` i `EventsService`. Uz te osnovne komponente koje smo sad naveli, postoje i dodatne koje nam olakšavaju izgradnju aplikacije, ali o njima ćemo u nastavku.

Što se tiče renderiranja izgleda same aplikacije, dio se renderira u skladu s MVC obrascem. Glavne stranice renderira server koristeći `viewove`, a sve podatke koje dodatno trebamo dohvaćamo slanjem *ajax* zahtjeva. Početnu stranicu renderira jedna od metoda iz `AppControllera` koristeći dekorator `@Render()`.

Početna stranica

Na početnoj stranici koju možemo vidjeti na slici 2.1 nalazi se popis svih događaja sortiran po popularnosti. Popularnost nekog događaja računamo kao omjer broja prodanih i broja ukupno dostupnih karata. U gornjem desnom kutu nalazi se gumb ”Prijava” gdje se korisnik može prijaviti. Na stranici se nalazi i gumb ”Filter” koji otvara formu u koju možemo upisati parametre i pretraživati događaje.



Slika 2.1: Početna stranica

Događaje možemo pretraživati po nazivu, po tipu događaja, po izvođaču te po gradu u kojem se događaj održava. Korisnik koji pretražuje događaje može upisati ime događaja i ime izvođača. Kod polja u koje unosi ime izvođača, ponude mu se svi izvođači koji su u bazi. Njih dohvaćamo *ajax* zahtjevom. Za vrstu događaja i grad u kojem se događaj izvodi, korisnik može odabrati jednu od ponuđenih opcija. Ponuđene su samo one opcije koje se nalaze u bazi, a dohvaćamo ih slanjem *ajax* zahtjeva serverskoj strani. Sve navedene zahtjeve šaljemo *controlleru* `EventsController`. On poziva metode iz `EventsService`

gdje se onda rade upiti na bazu koristeći `QueryBuilder`. Klikom na gumb "Pretraži", uneseni parametri se šalju *ajax* zahtjevom GET /events/filter. Da bismo lakše odredili tip parametara koji prima metoda koja obrađuje zahtjev, kreirali smo DTO (eng. *Data Transfer Object*) `FilterParamsDTO`. Općenito, DTO je klasa koja definira podatke koji se šalju kroz mrežu. U našem slučaju, `FilterParamsDTO` izgleda ovako:

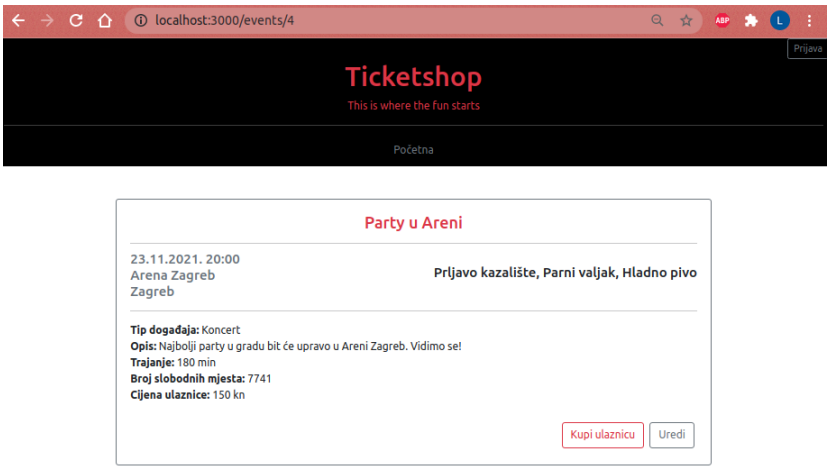
```
export class FilterParamsDTO {

    type: string;
    name: string;
    city: string;
    performer: string;
}
```

Metoda koja obrađuje zahtjev vraća popis svih događaja koji odgovaraju parametrima navedenim u proslijeđenom objektu tipa FilterParamsDTO te se oni prikazuju na početnoj stranici.

Pregled događaja

Osim pretraživanja događaja, korisnik može i pregledavati događaje. Na slici 2.1 vidimo uz svaki događaj postoji gumb "Više". Klikom na taj gumb otvara nam se nova stranica (slika 2.2) na kojoj su prikazani podaci o događaju.

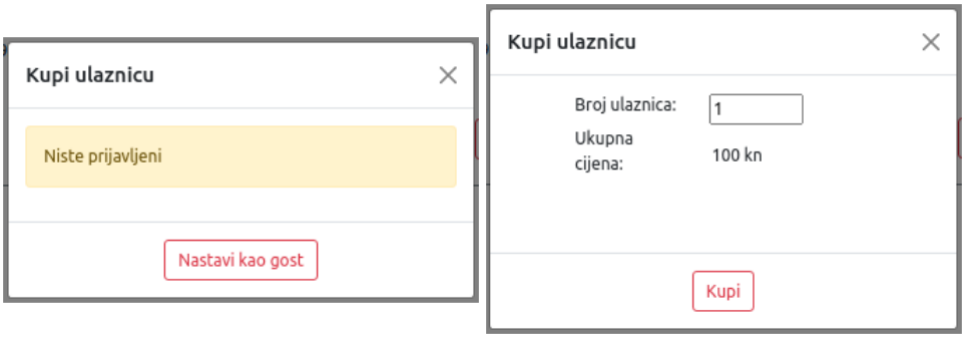


Slika 2.2: Prikaz događaja

Preusmjeravaje na tu stranicu napravljeno je na način da se aplikaciji šalje zahtjev GET /events/:id koji obrađuje `EventsController`. Da bismo provjerili šalje li se id koji mora biti cijeli broj, koristimo `ParseIntPipe` na način koji smo opisali u cjelini 1.9. Metoda koja obrađuje taj zahtjev dohvaća iz `EventsService` podatke o događaju čiji id je poslan zahtjevom i renderira `view` za prikaz događaja. Ako događaj s tim id-jem ne postoji, preusmjerava se na početnu stranicu. Na slici 2.2 vidimo da imamo gumbe "Kupi ulaznicu" i "Uredi".

Kupnja ulaznice kad korisnik nije prijavljen

Klikom na gumb "Kupi ulaznicu" otvara se prozor (slika 2.3 lijevo) u kojem se prvo nalazi poruka upozorenja da korisnik nije prijavljen, no nudi mu se mogućnost da nastavi kao gost. Ako nastavi kao gost, u tom otvorenom prozoru se prikazuje forma u kojoj može odabrati koji broj ulaznica želi kupiti te koja je ukupna cijena tih ulaznica (slika 2.3 desno).



Slika 2.3: Prozor koji se otvara kod kupnje karata

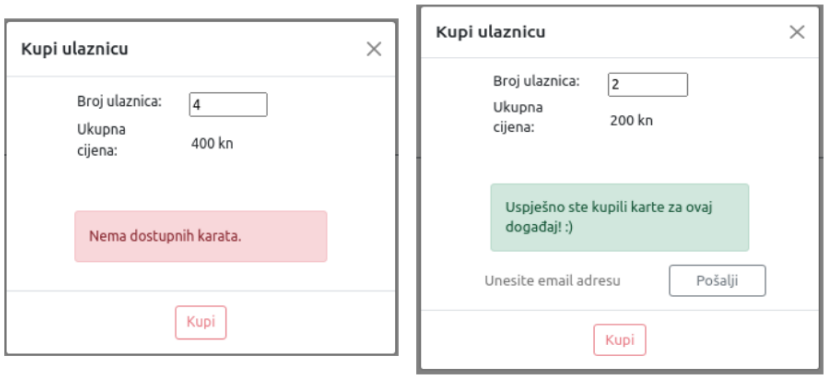
Klikom na gumb "Kupi" šaljemo zahtjev POST /events/buyTickets u čijem tijelu šaljemo broj karata i id događaja za koji kupujemo karte. Na tu rutu primijenili smo *middleware* `CheckAvailableTicketsMiddleware` koji provjerava ima li dovoljno dostupnih karata za taj događaj. Napomenimo samo da se u stvarnosti ova provjera radi unutar *controllera* i da se svi upiti na bazu rade samo iz servisa, no ovdje je stavljena u *middleware* kako bi ilustrirali na koji način se *middleware* koristi. Ako nema dovoljno karata, *middleware* vraća poruku da nema dostupnih karata i time prekida daljnje slanje zahtjeva. Ako su karte dostupne, poziva metodu `next()` i time šalje zahtjev na daljnju obradu. *Middleware* smo primijenili unutar glavnog modula `AppModule` na način koji je opisan u cjelini 1.6. `CheckAvailableTicketsMiddleware` implementirali smo na način da prvo dohvaćamo iz baze događaj čiji je id poslan u tijelu zahtjeva koji šaljemo. Ako događaj postoji, provjerimo ima li dovoljno dostupnih karata. Broj karata koje korisnik želi kupiti čitamo iz

tijela zahtjeva, a broj prodanih i ukupan broj dostupnih karata čitamo iz događaja koji smo dohvatili iz baze. Ako nema dovoljno dostupnih karata, vraćamo odgovor s porukom ”Nema dostupnih karata” što se prikazuje i na prozoru koji je otvoren (slika 2.4 lijevo). U suprotnome, zahtjev šaljemo na daljnju obradu.

```
@Injectable()
export class CheckAvailableTicketsMiddleware implements NestMiddleware{
    async use(req: Request, res: Response, next: NextFunction) {
        let event = await createQueryBuilder(Events, "event")
            .where("event.id = :id", { id: req.body.eventId })
            .getOne();

        if (event) {
            if (event.soldTickets + parseInt(req.body.numOfTickets)
                > event.tickets) {
                res.status(405).send("Nema dostupnih karata.");
            } else {
                next();
            }
        } else {
            next();
        }
    }
}
```

Ako je kupnja uspješna, korisniku se vraća poruka o uspješnoj kupovini (slika 2.4 desno). Ako je negdje došlo do greške, ispisuje se poruka ”Greška.”.



Slika 2.4: Poruke o neuspješnoj i uspješnoj kupovini karata

Mogućnost uređivanja događaja kada korisnik nije prijavljen

Na slici 2.2 vidimo da postoji i gumb za uređivanje događaja. Ako korisnik nije prijavljen u sustav, on nema pravo pristupa uređivanju događaja. Klikom na gumb "Uredi" preusmjeravamo ga na početnu stranicu i na vrhu stranice mu se ispisuje poruka "Morate biti prijavljeni za nastavak.". Ono što se zapravo događa u pozadini je sljedeće: klikom na gumb "Uredi" šalje se zahtjev GET /events/edit/:id pri čemu je id iz rute id događaja koji želimo uređivati. No na toj ruti je postavljen *guard* `LoginGuard` koji provjerava je li korisnik prijavljen i ako nije, usmjerava ga na početnu. Više o tome bit će napisano kasnije kada objasnimo na koji način funkcionira prijava korisnika.

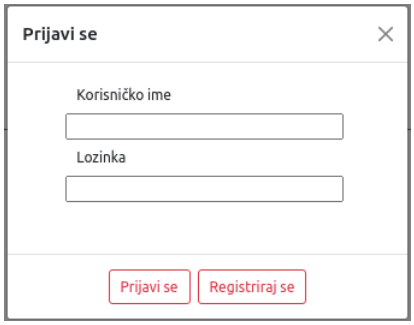
2.4 Registracija i prijava korisnika

Svaki korisnik web-aplikacije može se registrirati ili prijaviti ako je već registriran. Za obradu zahtjeva koji su vezani uz registraciju korisnika i prijavu korisnika zadužen je `LoginController`. `LoginController` sve zahtjeve koje dobije šalje na obradu servisu `UsersService`. Da bi `UsersService` bio dostupan `LoginControlleru`, morali smo ga u dekoratoru `@Module()` u modulu `UsersModule` navesti pod `exports` (što znači da smo modulu rekli da taj servis želimo dijeliti). Isto tako, morali smo `UsersModule` navesti u dekoratoru `@Module()` modula `LoginModule` pod `imports`.

```
//users.module.ts
@Module({
  imports: [TypeOrmModule.forFeature([Users])],
  providers: [UsersService],
  controllers: [UsersController],
  exports: [UsersService]
})
export class UsersModule {}

//login.module.ts
@Module({
  imports: [UsersModule, TypeOrmModule.forFeature([Users])],
  controllers: [LoginController]
})
export class LoginModule {}
```

Na svakoj stranici koju neprijavljeni korisnik vidi, u gornjem desnom kutu postoji gumb za prijavu. Klikom na taj gumb, otvara se prozor s formom za prijavu korisnika koji možemo vidjeti na slici 2.5. Ukoliko korisnik nema kreiran korisnički račun, klikom

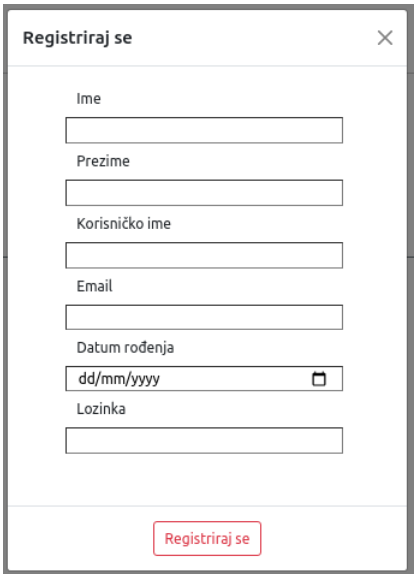


Slika 2.5: Prozor za prijavu

na ”Registriraj se” otvara se forma za registraciju. Prvo ćemo objasniti postupak registracije.

Registracija

Klikom na gumb ”Registriraj se” otvara se forma za unos podataka novog korisnika (slika 2.6). Kod kreiranja korisničkog računa, korisnik mora unijeti svoje ime, prezime, željeno korisničko ime, datum rođenja, email adresu i lozinku.



Slika 2.6: Prozor za registraciju

Klikom na gumb ”Regitriraj se” u prozoru za registraciju, šalje se zahtjev POST /lo-

gin/signUp u čijem tijelu se nalaze podaci koje je korisnik unio. Kreirali smo klasu, odnosno, DTO `NewUserDTO` koja opisuje podatke o novom korisniku koje šaljemo u tijelu zahtjeva. Kod registracije novog korisnika potrebno je napraviti validaciju podataka i provjeriti postoji li već korisnik s unesenim korisničkim imenom.

Provjera je li korisničko ime već zauzeto

Da bi provjerili je li korisničko ime već zauzeto, implementirali smo *middleware* `CheckIfUsernameExistsMiddleware` i primijenili ga na tu rutu. Unutar *middlewarea* radimo upit na bazu postoji li korisnik čije je korisničko ime jednako unesenom korisničkom imenu. Ako korisnik postoji, vraćamo odgovor "Korisničko ime je već zauzeto" i time prekidamo daljnju obradu zahtjeva. Odgovor se prikaže i korisniku na prozoru za registraciju. Ako korisničko ime nije zauzeto, pozivamo funkciju `next()` i zahtjev se dalje obrađuje.

Validacija podataka prije registracije

Prije registracije korisnika, potrebno je još napraviti validaciju podataka. Za validaciju podataka koristili smo `ValidationPipe` i paket `class-validator` koji ima ugrađene dekoratore za validaciju, a pokazat ćemo i kako smo napravili prilagođeni dekorator za validaciju. Da bismo mogli koristiti `class-validator`, morali smo ga prvo instalirati sljedećom naredbom:

```
npm i --save class-validator class-transformer
```

Dekoratore iz paketa `class-validator` koristimo kod definiranja vrijednosti u `NewUserDTO` jer one opisuju podatke koje korisnik šalje iz forme.

```
export class NewUserDTO {

    @Length(1,20)
    @IsValidName({ message: "name contains unallowed characters"}))
    name: string;

    @Length(1,20)
    @IsValidName({ message: "surname contains unallowed characters"})
    surname: string;

    @IsEmail()
    mail: string;
```

```
    @IsDateString()
    dateOfBirth: Date;

    @Length(1,10)
    username: string;

    @Length(8,20)
    password: string;
}
```

Opisat ćemo redom koja je uloga navedenih dekoratora. Za početak ćemo opisati one ugrađene dekoratore, a kasnije pokazati i kako implementirati novi dekorator za validaciju. Dekorator `@Length(1, 20)` provjerava je li duljina od `name` između 1 i 20. Dakle, vrijednost koju šaljemo kao `name` mora imati barem 1 znak, a najviše 20 znakova. Dekorator `@IsValidName()` je dekorator koji smo sami kreirali i njega ćemo opisati malo kasnije. Dekorator `@IsEmail()` provjerava ima li `mail` oblik email adrese. Dekorator `@IsDateString()` provjerava je li vrijednost `dateOfBirth` datum tj. string koji predstavlja datum. Ukoliko vrijednosti ne odgovaraju opisima u dekoratorima, `ValidationPipe` će nam javiti poruku o grešci. Dekoratori `@Length()`, `@IsEmail()` i `@IsDateString()` su iz paketa `class-validator`. Taj paket nam nudi i brojne druge dekoratore, a više o tome možemo naći na poveznici [8].

Pokažimo sada kako smo implementirali dekorator `@IsValidName()`. Njegova uloga je provjeriti je li ime korisnika u formatu u kojem želimo da bude. Dopušteni znakovi u imenu su sva slova, razmak i "-". Dekorator smo spremili u datoteku *IsValidName.decorator.ts*.

```
export function IsValidName(validationOptions?: ValidationOptions) {
  return function (object: Object, propertyName: string) {
    registerDecorator({
      name: 'IsValidName',
      target: object.constructor,
      propertyName: propertyName,
      constraints: [],
      options: validationOptions,
      validator: {
        validate(value: any, args: ValidationArguments) {

          let regexName = /[A-Zšđčćž\s-]/gi;
          let matching = value.match(regexName);

          if(matching && matching.length == value.length &&
```

```
            typeof value == "string") {
               return true;
          }
          return false;
        },
      },
    });
  };
}
```

Kod korištenja u `NewUserDTO`, dekoratoru `@IsValidName()` predali smo objekt koji sadrži poruku koju želimo vratiti ukoliko vrijednost nije onakva kakvu očekujemo. Sad kad smo objasnili koja je uloga kojeg dekoratora za validaciju, potrebno je još `ValidationPipe` primijeniti na metodu, odnosno, na argument metode koja prima i obrađuje zahtjev u `LoginControlleru`. U cjelini 1.9 rekli smo da se *pipe* navodi u dekoratoru koji dohvaća dio zahtjeva u kojem se šalju podaci i to uz argument na koji se primijenjuje. Ovdje se podaci šalju u tijelu zahtjeva i primamo objekt klase `NewUserDTO`.

```
@Post('signUp')
 async createNewUserAccount(@Body(
       new ValidationPipe({
           disableErrorMessages:false,
           skipMissingProperties: true})
       ) newUser: NewUserDTO): Promise<any> {}
```

Konstruktoru za `ValidationPipe` možemo predati objekt koji sadrži svojstva za taj *pipe*. Postavili smo vrijednost svojstva `disableErrorMessage` na `false` jer želimo da nam validator javi poruke o greškama. Svojstvo `skipMissingProperties` smo postavili na `true` i to nam služi tome da validator zanemari svojstva objekta čije vrijednosti nije dobio. Postoji još niz raznih svojstava što se također može naći na poveznici [8]. Metoda `createNewUserAccount()` prima još argumenata, ali to nam nije bitno za ovaj primjer, kao ni što točno metoda radi.

Kreiranje korisničkog računa

Ako korisničko ime nije zauzeto i ako su podaci uspješno prošli validaciju, zahtjev dolazi do *controllera* koji tada poziva metodu iz `UsersService-a` koja kreira novi korisnički račun i automatski prijavljuje korisnika u sustav. Spomenut ćemo još ovdje samo kako spremamo korisnikovu lozinku. Radi korisnikove sigurnosti, lozinku ne pohranjujemo u točnom obliku nego ju prije spremanja hashiramo. To je proces pretvaranja dane vrijednosti

u neku drugu vrijednost pomoću hash funkcije koja koristi neki matematički algoritam za tu pretvorbu. Nakon hashiranja, trebalo bi biti nemoguće saznati točnu vrijednost pohranjene lozinke. Ovdje smo za hashiranje lozinke koristili paket `bcrypt`. Prije korištenja, morali smo ga instalirati naredbama:

```
npm i bcrypt
npm i -D @types/bcrypt
```

U servis `UsersService` prvo smo uključili sve iz paketa `bcrypt`.

```
import * as bcrypt from 'bcrypt';
```

Metodu `hash()` koristimo prije spremanja lozinke u bazu na sljedeći način:

```
const salt = 7;
const hash_password = await bcrypt.hash(newUser.password, salt);
```

Nakon kreiranja korisničkog računa, korisnik je prijavljen u aplikaciju.

Prijava

Ako korisnik već ima svoj korisnički račun, onda se prijavljuje preko forme prikazane na slici 2.5. Klikom na gumb ”Prijavi se” šalje se zahtjev POST /login/validate i u tijelu zahtjeva se šalju korisničko ime i lozinka. Kod prijave korisnika potrebno je provjeriti jesu li podaci točni - postoji li korisnik s upisanim korisničkim imenom i ako postoji, je li upisana lozinka jednaka onoj u bazi. Budući da smo lozinku kod registracije korisnika hashirali, da bismo mogli usporediti lozinke, koristimo funkciju `compare()` iz paketa `bcrypt`. `password` je lozinka koju je korisnik upisao u formu kod prijave, a `hashPassword` je hashirana lozinka koja je spremljena u bazu.

```
let isEqual = await bcrypt.compare(password, hashPassword);
```

Ako su upisani podaci točni, korisnik je prijavljen u aplikaciju. Prijavu pamtimo u *sessionu*. Da bismo mogli raditi sa *sessionom*, prvo smo morali instalirati potrebne pakete.

```
npm i express-session
npm i -D @types/express-session
```

Nakon toga, bilo je potrebno postaviti ga unutar metode `bootstrap()` u datoteci *main.ts*.

```
app.use(
    session({
      secret: 'my-secret',
      resave: false,
      saveUninitialized: false,
      cookie: {
        maxAge: 3600000 //1h
      }
    }),
  );
```

Session dohvaćamo pomoću dekoratora `@Session()` ili preko objekta koji predstavlja zahtjev. Ako neka od funkcija koristi *session*, moramo ga navesti kao jedan od argumenata te funkcije.

Vratimo se na korištenje *sessiona* kod prijave. Ako su podaci bili točni, u *session* spremamo nekoliko podataka: id, korisničko ime i ulogu korisnika. Dakle, u *session* pohranjujemo objekt `user` i postavljamo vrijednost `isLoggedIn` na `true`

```
session.user = {
        username : user.username,
        id: user.id,
        role: user.role
      };
session.isLoggedIn = true;
```

Prijavljeni korisnik može se u bilo kojem trenutku odjaviti. Gumb za odjavu nalazi se u gornjem desnom kutu aplikacije. Klikom na taj gumb šalje se zahtjev GET /login/logout i brišu se podaci o prijavi iz *sessiona*.

2.5 Prava pristupa

Već u uvodu smo spomenuli da imamo tri vrste korisnika u aplikaciji: neprijavljeni korisnik ili gost, prijavljeni korisnik i administrator. Svakoj vrsti korisnika dostupni su određeni dijelovi aplikacije. Da bismo provjerili ima li korisnik koji šalje zahtjev dovoljno prava za pristupiti nekom dijelu aplikacije koristimo *guardove*. Vidjeli smo već u cjelini 2.3 kod uređivanja događaja da neprijavljeni korisnik nema dovoljno prava, no sad ćemo pogledati što stoji iza toga.

LoginGuard

Implementirali smo *guard* `LoginGuard` koji provjerava je li korisnik prijavljen u sustav. Unutar njega dohvaćamo podatke o *sessionu*. Ako postoji zapis o prijavi u *sessionu*, onda zahtjev ide na daljnju obradu. Ako je *session* prazan, korisnik nije prijavljen i bacamo `ForbiddenException` iznimku s porukom "login guard".

```
@Injectable()
export class LoginGuard implements CanActivate {
  canActivate(context: ExecutionContext): boolean | Promise<boolean>
  | Observable<boolean> {
    const request = context.switchToHttp().getRequest();
    if(request.session.loggedIn) {
        return true;
    }
    throw new ForbiddenException("login guard");
  }
}
```

`LoginGuard` primijenili smo na sve rute koje želimo da budu dostupne samo prijavljenim korisnicima. To su sve rute iz `UsersControllera`, a one se odnose na dohvaćanje i izmjenu korisnikovih podataka te na rute iz `EventsControllera` koje se odnose na uređivanje podataka o događajima (uređivanje, kreiranje, brisanje). Primijenili smo ih koristeći dekorator `@UseGuards(LoginGuard)`.

RolesDecorator

Osim ovisno o prijavi, prava pristupa se razlikuju i po ulozi korisnika u aplikaciji. Prijavljeni korisnik može biti samo korisnik ili administrator. Podaci o ulozi korisnika spremljeni su u bazu, a kod svake prijave korisnika u aplikaciju, podaci o njegovoj ulozi spremaju se i u *session*.

Da bismo mogli označiti kojim korisnicima su određene rute dostupne, implementirali smo novi dekorator `@Roles()`. Dekorator `@Roles()` prima jednu ili više uloga kojima je neka metoda, koja obrađuje zahtjev, dostupna. Dekorator smo implementirali na način da postavlja metapodatke koristeći dekorator `@SetMetadata()` za metode na koje ga primjenjujemo.

```
export const Roles=(...roles: string[])=>SetMetadata('roles', roles);
```

Dekorator `@Roles()` navodimo iznad metode na koju se primijenjuje kao i sve dekoratore koje primijenjujemo na neku metodu.

RolesGuard

`RolesGuard` smo implementirali kako bismo mogli kontrolirati prava pristupa ovisno o korisnikovoj ulozi u aplikaciji. Koristeći objekt klase `Reflector` dohvatili smo listu svih uloga koje su postavljene unutar dekoratora `@Roles()` za određenu metodu koja obrađuje zahtjev. Ako uloge za tu metodu nisu definirane, očito svi imaju pristup toj metodi pa zahtjev samo proslijedimo na daljnju obradu. Ako su uloge definirane, onda iz *sessiona* čitamo koja je uloga prijavljenog korisnika i gledamo je li među definiranim ulogama. Ako je, proslijedimo zahtjev na daljnju obradu, a ako nije, bacimo iznimku `ForbiddenException` s porukom ”roles guard”.

```
import { Reflector } from '@nestjs/core';

@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const roles = this.reflector.get<string[]>('roles',
                    context.getHandler());
    if (!roles) {
      return true;
    }
    const request = context.switchToHttp().getRequest();
    const userRole = request.session.user.role;
    const found = roles.find(role => role == userRole);

    if(found) {
        return true;
    }
    throw new ForbiddenException("roles guard");
  }
}
```

`RolesGuard` primjenujemo na sve metode koje želimo zaštititi koristeći dekorator `@UseGuards()`. Budući da prije njega želimo provjeriti je li korisnik uopće prijavljen, onda navodimo prvo `LoginGuard`, a zatim `RolesGuard`.

```
@UseGuards(LoginGuard, RolesGuard)
```

Detaljnije o tome na koje rute primjenjujemo `RolesGuard`, spomenut ćemo kasnije.

Filter za iznimku ForbiddenException

Spomenuli smo ranije da se korisnik preusmjerava natrag na početnu stranicu ako pokuša uređivati događaj, a nije prijavljen. U `LoginGuardu` smo vidjeli da bacamo iznimku ako korisnik nije prijavljen, a pokušava pristupiti nekoj zabranjenoj ruti. Ista stvar se događa i unutar `RolesGuarda`. Za preusmjeravanje natrag na početnu stranicu je zadužen upravo filter za iznimku `ForbiddenException`. Implementirali smo filter `ForbiddenException-Filter` na sljedeći način: kod bacanja iznimke, šaljemo i poruku koji *guard* baca tu iznimku i ovisno o tome onda preusmjeravamo korisnika. Zapravo, preusmjeravamo zahtjev na dvije različite rute iz `AppControllera` koje postavljaju koja poruka će se ispisati korisniku na ekranu i zatim preusmjeravaju korisnika na početnu stranicu. Ukoliko je bačena iznimka, a poruka uz iznimku nije poslana, preusmjeravamo korisnika direktno na početnu stranicu, odnosno, šaljemo zahtjev GET /.

```
@Catch(ForbiddenException)
export class ForbiddenExceptionFilter implements ExceptionFilter {
  catch(exception: ForbiddenException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    const msg = exception.message;

    if(msg == "login guard") {
       response.redirect('/forbiddenExceptionLogin');
    }
    else if(msg == "roles guard"){
      response.redirect('/forbiddenExceptionRoles');

    } else {
      response.redirect('/');
    }
  }
}
```

Filter smo primijenili globalno na cijelu aplikaciju unutar metode `bootstrap()` koristeći metodu `useGlobalFilters()` na način koji je opisan u cjelini 1.10.

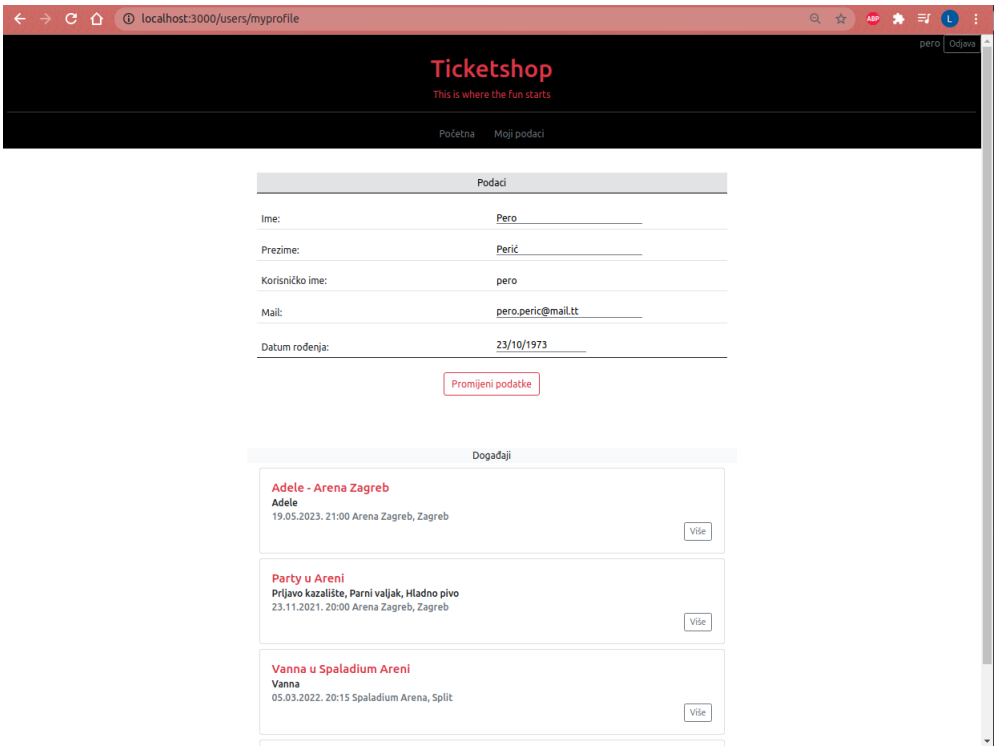
Ako korisnik nije prijavljen, na ekranu mu se ispisuje poruka ”Morate biti prijavljeni za nastavak.”. Ako se radi o običnom korisniku koji želi pristupiti rutama koje su namijenjene samo administratorima, na ekranu mu se ispisuje poruka ”Nemate pravo pristupa.”.

2.6 Dijelovi aplikacije dostupni prijavljenim korisnicima

Nakon što se korisnik prijavi u aplikaciju, on može sve što može i neprijavljeni korisnik, a može i pregledavati svoj profil. U navigacijskom izborniku pojavljuje mu se opcija ”Moj profil”. Klikom na tu opciju, korisnik se preusmjerava na stranicu koja prikazuje njegov profil.

Pregled profila

Klikom na ”Moj profil” šalje se zahtjev GET /users/myprofile. Zahtjev prvo prolazi kroz `LoginGuard`. Ako zahtjev dođe do *controllera* `UsersController`, u *controlleru* se dohvaća korisnikov id iz *sessiona* i poziva se metoda iz servisa `UsersService` koja dohvaća korisnikove podatke po id-u. U svrhu vraćanja korisnikovih podataka, kreirali smo `UsersProfileDTO` koji sadrži korisnikove podatke prilagođene renderiranju (promijenjen je format datuma kako bi se mogao lijepo prikazati na formi). Na svom profilu, korisnik može vidjeti svoje podatke i može vidjeti popis događaja za koje je kupio kartu.



Slika 2.7: Prikaz korisnikovog profila

Promjena podataka

Ispod forme na kojoj su prikazani korisnikovi podaci, nalazi se gumb "Promijeni podatke". Klikom na taj gumb, otvara se forma i korisnik može mijenjati svoje podatke. Korisnik može spremiti podatke klikom na "Spremi" ili odustati od uređivanja i spremanja klikom na "Odustani" čime se forma vraća u prvobitno stanje. Ako korisnik klikne na "Spremi", šalje se zahtjev POST /users/saveData s novim podacima u tijelu zahtjeva. Podaci prvo prolaze validaciju. Validacija je napravljena gotovo jednako kao i kod registracije korisnika, samo što smo kreirali novi DTO `UpdateUserDTO` koji ima samo svojstva `name`, `surname`, `mail` i `dateOfBirth` jer ostale podatke nije dozvoljeno mijenjati. Također, da bismo pokazali koji su još dekoratori dostupni, umjesto dekoratora `@Length()` koristili smo `@MinLength()` i `@MaxLength()`.

```
@MinLength(1)
@MaxLength(20)
@IsValidName({ message: "name contains unallowed characters"})
name: string;
```

Kupovina ulaznica

Kupovina ulaznica funkcionira gotovo jednako kao i za neprijavljenog korisnika. Razlika je u tome što se prijavljenom korisniku ne pojavljuje upozorenje da nije prijavljen. Još jedna razlika je u tome što se kupovina, ako je korisnik prijavljen, sprema i u bazu pa korisnik može vidjeti na svom profilu za koje događaje je kupio kartu.

Uređivanje događaja

Ako prijavljeni korisnik koji nije administrator klikne na gumb "Uredi" kod prikaza događaja, preusmjerava ga se na početnu stranicu i prikazuje mu se poruka "Nemate pravo pristupa."

2.7 Dijelovi aplikacije dostupni administratoru

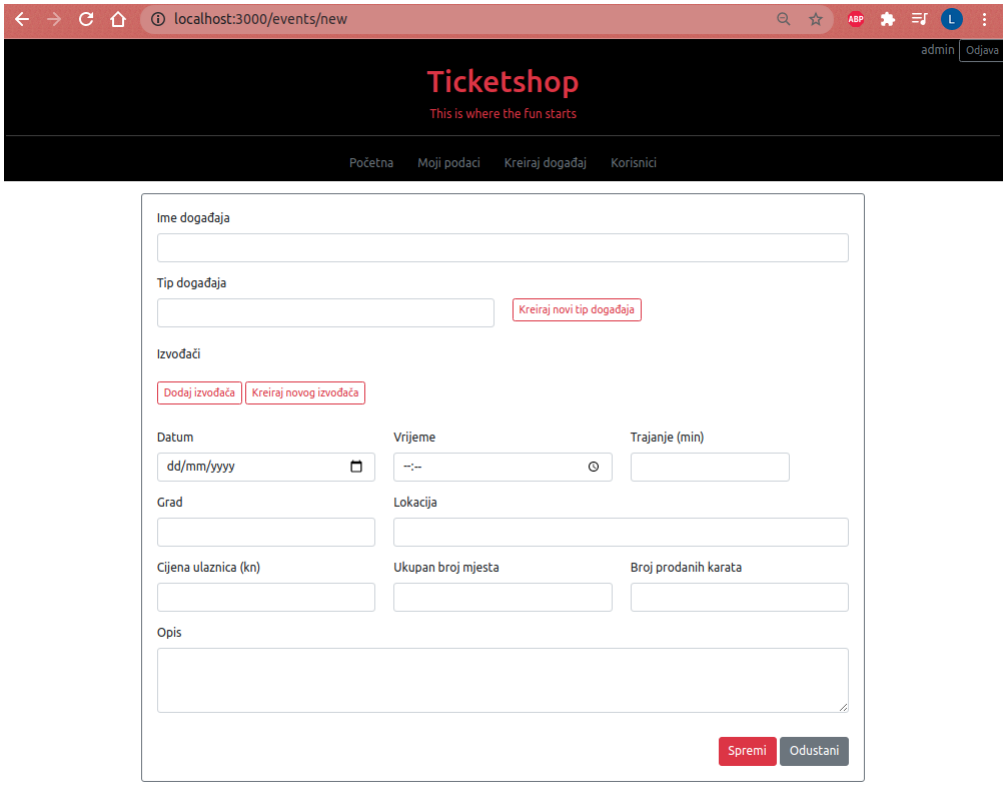
Budući da administrator mora biti prijavljen u sustav, njemu su dostupne sve mogućnosti koje su dostupne svakom prijavljenom korisniku. Dakle, može pretraživati događaje, pregledavati i uređivati svoje podatke te kupovati ulaznice za nove događaje. Dodatne mogućnosti koje mu se nude su uređivanje, brisanje i kreiranje događaja, pregled svih korisnika i njihovih profila te promjena uloge svakog korisnika. Iznad svih metoda u `EventsControlleru` i `UsersControlleru` koje su dostupne samo administratoru, dodali smo dekoratore

```
@UseGuards(LoginGuard, RolesGuard)
@Roles('admin')
```

Ako tim rutama pristupi korisnik koji nije administrator, preusmjerava se ga na početnu stranicu kao što je opisano u prethodnom dijelu. Kada se korisnik koji je administrator prijavi u sustav, u navigacijskom izborniku mu se pokažu dvije dodatne opcije: ”Kreiraj događaj” i ”Korisnici”.

Kreiranje novog događaja

Klikom na opciju ”Kreiraj događaj” u navigacijskom izborniku, administratoru se otvara stranica s prikazom forme za kreiranje novog događaja (slika 2.8).

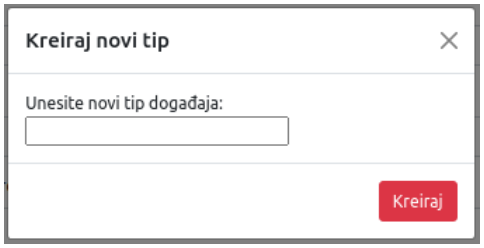


Slika 2.8: Forma za kreiranje novog događaja

Administrator u formu upisuje ime događaja, opis događaja, datum, vrijeme i trajanje događaja, grad i lokaciju održavanja događaja, cijenu ulaznice te broj dostupnih i broj prodanih ulaznica za događaj. Za tip događaja i izvođače, administrator bira neki od već ponuđenih tipova, odnosno, izvođača ili kreira novi.

Kreiranje novog tipa događaja

Administratoru se na formi za kreiranje događaja nudi mogućnost odabira jednog od već postojećih tipova događaja. Podatke o tipovima događaja koji postoje dohvatili smo iz tablice `types`. Ako administrator želi kreirati događaj nekog novog tipa, nudi mu se mogućnost kreiranja tog novog tipa. Pokraj forme za odabir tipa događaja postoji gumb "Kreiraj novi tip događaja". Klikom na taj gumb, otvara se prozor s formom za unos naziva novog tipa događaja.



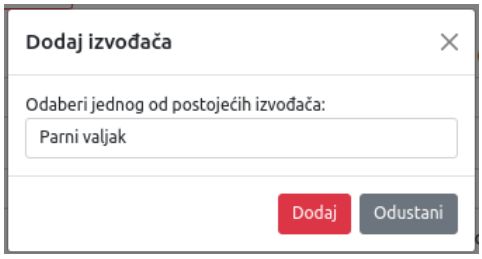
Slika 2.9: Forma za kreiranje novog tipa događaja

Klikom na gumb "Kreiraj", šalje se zahtjev PUT /events/addNewType i u tijelu zahtjeva šalje se naziv tipa koji želimo kreirati. Budući da želimo spriječiti dupliciranje već postojećih tipova događaja, implementirali smo *middleware* `CheckIfTypeExistsMiddleware` koji provjerava postoji li u bazi tip s istim imenom i primijenili ga na tu rutu. `CheckIfTypeExistsMiddleware` je vrlo sličan *middlewareu* `CheckIfUsernameExistsMiddleware` koji smo koristili kod registracije korisnika samo što dohvaća druge podatke. Ako u bazi već postoji tip događaja s istim nazivom, *middleware* vraća odgovor "Taj tip događaja već postoji." i prekida daljnju obradu zahtjeva. Poruka se prikazuje i u prozoru za kreiranje novog tipa. Ako takav tip događaja ne postoji, zahtjev se šalje `EventsControlleru` koji tada poziva metodu iz `EventsService` koja u bazu sprema novi tip događaja. Nakon spremanja, ažurira se lista svih dostupnih događaja i administrator može odabrati ovaj tip koji je kreirao.

Dodavanje postojećeg izvođača ili kreiranje novog izvođača

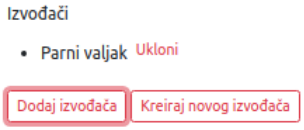
Na formi za kreiranje događaja, imamo gumb "Dodaj izvođača" i gumb "Kreiraj novog izvođača". Ideja dodavanja izvođača je jednaka ideji dodavanja tipa događaja. Administra-

tor može odabrati ili izvođače koji već postoje u sustavu ili dodaje nekog novog izvođača. Klikom na gumb ”Dodaj izvođača” otvara mu se prozor s formom za odabir jednog od ponuđenih izvođača (slika 2.10). Popis svih izvođača dohvaćamo iz tablice `performers`. Nakon što administrator odabere izvođača kojeg želi dodati i klikne na ”Dodaj”, odabrani



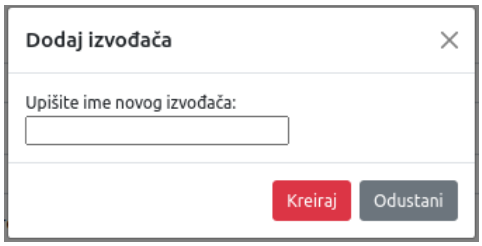
Slika 2.10: Forma za dodavanje izvođača

izvođač se pokaže na formi za kreiranje događaja i kraj njega se pojavljuje i gumb ”Ukloni” kako bi se u svakom trenutku izvođač mogao ukloniti s liste izvođača koji nastupaju na tom događaju (slika 2.12).



Slika 2.11: Prikaz odabranih izvođača na formi za kreiranje događaja

Ako administrator želi dodati izvođača koji nije u sustavu, klikom na gumb ”Kreiraj novog izvođača” otvara mu se prozor s formom za unos imena novog izvođača. Ovaj



Slika 2.12: Forma za kreiranje novog izvođača

dio napravljen je jednako kao i dio za dodavanje novog tipa događaja. Klikom na gumb ”Kreiraj” šalje se zahtjev PUT /events/addNewPerformer. Implementirali smo *middleware*

`CheckIfPerformerExistsMiddleware` koji dohvaća sve izvođače i provjerava postoji li već uneseni izvođač. Ako postoji, *middleware* vraća odgovor s porukom "Izvođač već postoji.", a ako ne postoji, šalje zahtjev na obradu `EventsControlleru`. Nakon spremanja u bazu, lista svih postojećih izvođača se ažurira i administrator može odabrati kreiranog izvođača.

Spremanje podataka o događaju

Novi događaj, administrator kreira klikom na gumb "Spremi". Svi podaci koje je upisao u formu za kreiranje događaja, šalju se u tijelu zahtjeva POST /events/saveEventData. Prije nego što spremimo podatke o događaju u bazu, moramo provjeriti jesu li podaci u redu. Implementirali smo *middleware* `CheckNewEventDataMiddleware` koji provjerava jesu li trajanje, cijena ulaznica, broj dostupnih ulaznica te broj prodanih ulaznica koji se šalju u tijelu zahtjeva pozitivni brojevi. Ako nisu, postavlja ih na 0. Također provjerava i je li broj prodanih ulaznica manji od ukupnog broja ulaznica te mijenja podatke koji se šalju ako je to potrebno. `CheckNewEventDataMiddleware` smo primijenili na rutu POST /events/saveData.

```
@Injectable()
export class CheckNewEventDataMiddleware implements NestMiddleware {
    async use(req:Request, res:Response, next:NextFunction) {
        const duration = parseInt(req.body.duration);
        if(!duration || duration < 0) {
            req.body.duration = '0';
        }
        const price = parseInt(req.body.price);
        if(!price || price < 0) {
            req.body.price = '0';
        }
        const soldTickets = parseInt(req.body.soldTickets);
        if(!soldTickets || soldTickets < 0) {
            req.body.soldTickets = '0';
        }
        const tickets = parseInt(req.body.tickets);
        if(!tickets || tickets < 0) {
            req.body.tickets = '0';
        }
        if(tickets && soldTickets && tickets < soldTickets) {
            req.body.soldTickets = req.body.tickets;
        }
```

```
        next();
    }
}
```

Nakon `CheckNewEventDataMiddleware`, zahtjev prolazi kroz validaciju podataka. Ovdje smo opet koristili `ValidationPipe` kao i u slučaju validacije podataka kod registracije korisnika. Kreirali smo `NewEventDataDTO` u kojem smo opisali kakve podatke očekujemo dobiti i dodali dekoratore za validaciju. Koristili smo dekorator koji smo i ranije koristili: ugrađeni `@IsDateString()`. Ranije smo implemetirali dekorator `@IsValidName()` koji provjerava sadrži li string koji je poslan samo slova, znak ”-” i razmak. Za ime događaja i za naziv mjesta želimo dopustiti još neke znakove. Zbog toga smo implementirali nove dekoratore na jednak način kao što smo implementirali i dekorator `@IsValidName()` jedino je regularni izraz s kojim uspoređujemo danu vrijednost različit. Dekorator `IsValidEventName()` nam provjerava sadrži li string koji je upisan kao ime događaja slova, razmak, ”-”, ”*”, ”!”, ”?”, ”+”, ”.” i ”,”. Drugi dekorator koji smo implementirali je dekorator `@IsValidPlaceName()` koji provjerava sadrže li stringovi slova, razmak i znakove ”-” i ”,”. Njega smo primijenili na grad i na lokaciju događaja. Implementirali smo i dekorator `@CheckTime()` koji provjerava je li uneseno vrijeme događaja u dobrom formatu. Koristili smo i još neke ugrađene dekoratore. Koristili smo `@IsNumberString()` koji provjerava sadrži li string koji je označen samo znamenke. Koristili smo i dekorator `@IsArray()` koji provjerava za izvođače je li poslano polje izvođača. Još jedan koristan dekorator koji smo koristili je `@ValidateIf()` kojem predajemo uvjet kada se određena vrijednost validira. Na primjer, dopuštamo da se kod kreiranja događaja ne unese cijena ulaznice, ali ako je unesena, onda želimo da to bude broj. Validaciju u tom slučaju provodimo na sljedeći način:

```
@ValidateIf( event => event.price != "" && event.price != 'NaN')
@IsNumberString()
price: number;
```

Ako vrijednost cijene ulaznice nije definirana i ako je različita od vrijednosti `NaN`, onda se ”aktivira” dekorator `@IsNumberString()`. Ako vrijednost za cijenu nije poslana, dekorator `@IsNumberString()` će biti zanemaren.

Ako podaci iz zahtjeva za spremanje događaja prođu validaciju, zahtjev se šalje na daljnju obradu `EventsControlleru` koji zatim prosljeđuje podatke metodi `createNewEvent()` iz `EventsService` koja ih sprema u bazu. Nakon spremanja, u aplikaciji se prikazuje stranica poput one na slici 2.2 na kojoj se prikazuju podaci o događaju.

Uređivanje već postojećeg događaja

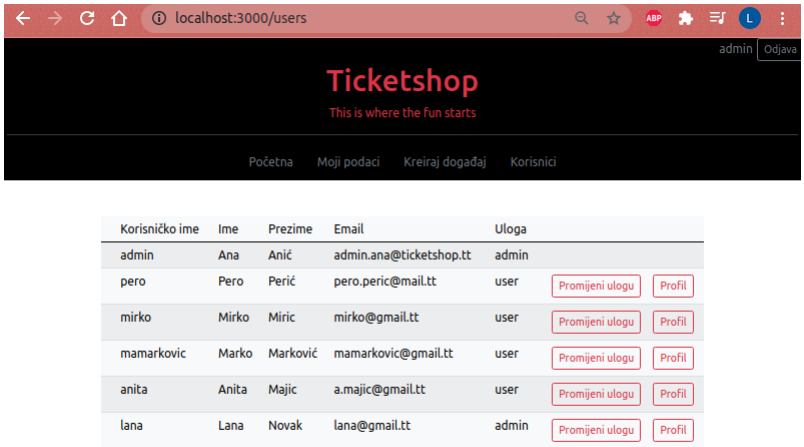
Kod prikaza događaja, imamo gumb ”Uredi” koji smo spominjali već ranije kod prava pristupa. Administrator ima dovoljno prava za uređivanje događaja i klikom na taj gumb šalje se zahtjev GET events/edit/:id koji dohvaća podatke o događaju preko njegovog id-ja i prikazuje te podatke unutar forme koju smo već vidjeli kod kreiranja novog događaja (slika 2.8). Administrator može promijeniti bilo koji podatak o događaju i klikom na gumb ”Spremi” šalje se isti zahtjev kao i kod kreiranja novog događaja. Jedina razlika je u tome što se u metodi unutar `EventsControllera` provjerava je li to postojeći događaj (provjerava je li poslan i id) i ako je, poziva se metoda `updateEvent()` iz `EventsService` umjesto metode `createNewEvent()`.

Brisanje događaja

Pri uređivanju već postojećeg događaja, na dnu forme kraj gumba za spremanje događaja, nalaze se i gumbi ”Odustani” i ”Obriši”. Klikom na gumb ”Obriši”, otvara se prozor u kojem se provjerava želi li administrator stvarno obrisati događaj. Ako da, šalje se zahtjev POST /events/deleteEvent i događaj se briše iz baze.

Pregled svih korisnika

Klikom na opciju ”Korisnici” u navigacijskom izborniku, šalje se zahtjev GET /users/ kojim administrator dohvaća popis svih korisnika koji imaju kreirani korisnički račun.



Slika 2.13: Prikaz popisa svih korisnika

Administrator može vidjeti profil svakog od njih, no ne može uređivati podatke. Klikom na gumb "Profil" kraj imena svakog od njih, šalje se zahtjev GET /users/:id i dohvaćaju se podaci korisnika čiji je id poslan u zahtjevu. Profil koji vidi administrator izgleda jednako kao što ga vidi i korisnik. Jedina razlika je što nema gumba "Promijeni podatke".

Administrator može promijeniti ulogu svakom korisniku s liste osim sebi. Klikom na gumb "Promijeni ulogu" šalje se zahtjev POST /users/changeUsersRole i u tijelu zahtjeva šalju se korisnikov id i trenutna uloga. Za provjeru je li poslan id koji je broj, koristimo `ParseIntPipe`. Ako je korisnik bio običan korisnik (`user`), klikom na "Promijeni ulogu", njemu se dodjeljuje uloga administratora, a ako je bio administrator, postaje običan korisnik.

U ovom dijelu dohvaćanja korisnika koristili smo postupak serijalizacije (eng. *serialization*). To je proces koji se izvršava prije nego su podaci poslani kao odgovor klijentskoj strani. Postupak nam omogućava da neke podatke ne pošaljemo klijentskoj strani web-aplikacije. Kod dohvaćanja korisnika, nismo kreirali DTO kojim bi se slali podaci nego smo klijentskoj strani poslali objekte entiteta `User` među kojima je i korisnikova lozinka. Budući da ne želimo da se pošalje i korisnikova lozinka, primijenili smo *interceptor* `ClassSerializerInterceptor` na cijeli `UsersController` koristeći dekorator `@UseInterceptors()`. Sada će se svaki put kad se šalje objekt entiteta `Users`, korisnikova lozinka maknuti iz objekta koji se šalje. Da bismo označili koje sve podatke želimo maknuti iz objekta koji se šalje, u definiranju klase koja predstavlja entitet `Users` koristimo dekorator `@Exclude()` iz paketa `class-transformer`. Navodimo ga iznad svih svojstva koja želimo izuzeti od slanja. Dakle, iznad lozinke u entitetu `Users` navodimo dekoratore ovako:

```
@Exclude()
@Column()
password: string;
```

Ovime smo prošli kroz sve mogućnosti koje izrađena aplikacija nudi. Opisali smo detalje implementacije i vidjeli na koji način se sve komponente koje smo opisali u prvom poglavlju mogu iskoristiti. Osnovne komponente, ali i dodatne komponente koje nam Nest nudi olakšavaju nam izradu aplikacije i organizaciju strukture aplikacije. Omogućavaju nam i održavanje te strukture jer se točno zna što je čija uloga u izradi aplikacije. Također, pomažu nam u smanjenju duplikacije koda. Upravo zato što neki *middleware* ili *interceptor* ili bilo koju drugu komponentu možemo primijeniti na više ruta odjednom, uvelike se smanjuje potreba za duplikacijom koda što čini aplikaciju bolje organiziranom i lakše ju je nadograđivati ili mijenjati.

Bibliografija

[1] *Dokumentacija programskog jezika JavaScript*, https://javascript.info/, posjećena u studenom 2021.

[2] Angular, *Službena dokumentacija razvojnog okvira - dependency injection*, https://angular.io/guide/dependency-injection, posjećena u studenom 2021.

[3] ExpressJS, *Službena dokumentacija razvojnog okvira*, https://expressjs.com/, posjećena u studenom 2021.

[4] Handlebars, *Službena dokumentacija Handlebars.js*, https://handlebarsjs.com/, posjećena u studenom 2021.

[5] NestJS, *Službena dokumentacija razvojnog okvira*, https://docs.nestjs.com/, posjećena u studenom 2021.

[6] TypeORM, *Službena dokumentacija TypeORM*, https://typeorm.io/#/, posjećena u studenom 2021.

[7] TypeScript, *Službena dokumentacija programskog jezika TypeScript*, https://www.typescriptlang.org/, posjećena u studenom 2021.

[8] TypeStack, *Github repozitorij o class-validatoru*, https://github.com/typestack/class-validator, posjećena u studenom 2021.

Sažetak

U ovom diplomskom radu opisan je moderni razvojni okvir NestJS i izrađena je web-aplikacija koja demonstrira njegove mogućnosti. U prvom dijelu rada, opisane su komponente koje nam NestJS nudi i njihova uloga u izradi aplikacije. Opisan je i način povezivanja s bazom koristeći TypeORM te način izrade aplikacije koristeći *Model-View-Controller* obrazac. U drugom dijelu rada, predstavljena je aplikacija koju smo izradili. Opisane su sve mogućnosti koje aplikacija ima i način na koji su one implementirane pomoću komponenti koje nam NestJS nudi.

Summary

In this master thesis, we studied the progressive Node.js framework NestJS which is used for building server-side web applications. For that purpose, we have built a web application that demonstrates some of the possibilities that NestJS provides. In the first chapter, we talked about NestJS components and their purpose in the application. We have also described how to connect to the database using TypeORM and how to organize and implement our application following the *Model-View-Controller* pattern. In the second chapter, we documented the application that we have built. We described all the features that the application provides and the way we have implemented them using NestJS components.

Životopis

Rođena sam 25. svibnja 1997. godine u Čakovcu. Završila sam Osnovnu školu Kuršanec i nakon toga upisala Gimnaziju Josipa Slavenskog Čakovec. Srednju školu završila sam 2016. godine i iste godine upisala sam preddiplomski studij Matematika na Matematičkom odsjeku Prirodoslovno-matematičkog fakulteta u Zagrebu. Preddiplomski studij završila sam 2019. godine i iste godine upisala sam diplomski studij Računarstvo i matematika na istom fakultetu.