NestJS

Sabo, Mario

Undergraduate thesis / Završni rad

2020

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Josip Juraj Strossmayer University of Osijek, Department of Mathematics / Sveučilište Josipa Jurja Strossmayera u Osijeku, Odjel za matematiku**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:126:058660

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2024-03-28**

Repository / Repozitorij:

Repository of School of Applied Mathematics and Computer Science

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike i računarstva

Mario Sabo

NestJS

Završni rad

Osijek, 2020.

Sveučilište J.J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike i računarstva

Mario Sabo

NestJS

Završni rad

Mentor: izv. prof. dr. sc. Domagoj Matijević
Komentor: mag. math. Jurica Maltar

Osijek, 2020.

Sažetak

Ovaj završni rad opisuje NestJS, okvir za razvoj poslužiteljskih aplikacija. Opisuje tehnologije koje NestJS koristi u pozadini, poput Javascripta, TypeScripta, Node.js-a, Express.js. Također opisuje HTTP protokol koji se koristi unutar samih aplikacija koje razvijemo pomoću NestJS okvira. Kako bi se što bolje opisao rad ovog okvira, tokom rada se izrađuje jednostavna aplikacija.

Ključne riječi

NestJS, okvir, Express.js, Node.js, Javascript, TypeScript, HTTP, poslužiteljska aplikacja

Summary

This bachelor's thesis describes NestJS, framework for backend applications. It describes technologies which NestJS uses under the hood, like Javascript, TypeScript, Node.js, Express.js. It describes HTTP protocol which is used inside applications developed using NestJS framework. An application is developed along with this paper. It helps to clarify some parts of framework.

Key words

NestJS, framework, Express.js, Node.js, Javascript, TypeScript, HTTP, backend application

Sadržaj

1 JavaScript **2**
1.1 Tipovi podataka . . . 2
1.2 Asinkrono izvršavanje . . . 2
1.2.1 Async/await mehanizam . . . 2

2 TypeScript **3**
2.1 Dekoratori . . . 3

3 Node.js **4**
3.1 Npm . . . 4

4 Express.js **4**

5 HTTP **5**
5.1 HTTP metode . . . 5
5.2 Status . . . 5

6 NestJS **6**
6.1 Nest CLI . . . 6
6.2 Opskrbljivači . . . 6
6.2.1 Ubrizgavanje ovisnosti . . . 6
6.2.2 Opskrbiljivači . . . 7
6.3 Moduli . . . 8
6.3.1 Polje providers . . . 8
6.3.2 Polja imports i exports . . . 9
6.3.3 Polje controllers . . . 10
6.3.4 Aplikacija . . . 10
6.4 Kontroleri . . . 11
6.4.1 Parametrizirane rute . . . 12
6.4.2 Aplikacija . . . 13
6.5 Servisi . . . 16
6.5.1 Aplikacija . . . 17
6.6 Cijevi . . . 19
6.6.1 Aplikacija . . . 20
6.7 Presretači . . . 22
6.7.1 Aplikacija . . . 22
6.8 Čuvari . . . 30
6.8.1 Aplikacja . . . 30
6.9 Pokretanje aplikacije . . . 33

Literatura **34**

1 JavaScript

JavaScript je programski jezik prvi puta predstavljen 1995. godine. U početku je to bio jezik koji je omogućavao kontrolu toka na internetskim stranicama koje su se prikazivale Netscape Navigator internetskom pregledniku. S vremenom je JavaScript usvojen i sada je dio svih poznatijih grafičkih internetskih preglednika i omogućuje internetskim stranicama da budu ono što od njih danas očekujemo da jesu, odnosno omogućuje stranicama da koriste animacije, prikaz sadržaja u ovisnosti o parametrima, interakciju sa stranicom i još puno toga.

1.1 Tipovi podataka

JavaScript se sastoji od nekoliko osnovnih tipova podataka. Imamo primitivne tipove: brojeve (*number*), nizove znakova (*string*), logičke vrijednosti(*boolean*) te tipove *null* i *undefined* koji u srži označavaju istu stvar, a to je iščezavanje vrijednosti nekog podatka. Osim primitivnih tipova postoji još i tip podatka "objekt" (*object*) koji je namijenjen stvaranju vlastitih tipova podatka ovisno o potrebi. JavaScript dolazi s još jednim unaprijed proširenim objektom koji se naziva polje (*array*) iako to zapravo nije polje u smislu kojeg podrazumijevamo u računalnoj znanosti, nego je to implementacija jedne vrste "hash mape".

JavaScript nije snažno tipiziran jezik, odnosno prilikom pisanja koda ne specificiramo direktno tip podatka nego se o njemu odlučuje interno tijekom prevođenja te unutar koda istoj varijabli možemo dodjeljivati različite tipove podataka.

1.2 Asinkrono izvršavanje

Radnje kao što su dohvaćanje podataka s nekog vanjskog poslužitelja ne mogu se izvršiti u trenutku zbog fizikalnih ograničenja tehnologije koju koristimo te je potrebno neko vrijeme koje nije unaprijed poznato da od tog poslužitelja dobijemo odgovor. Kada bismo to radili na klasičan (sinkron) način, izvršavanje našeg programa bismo bilo blokirano dok ne dobijemo odgovor. Ovo je samo jedan primjer koji predstavlja problem, a ima ih još mnogo. Iz tog razloga koristimo asinkrono programiranje, odnosno sustav koji će takve radnje započeti i nastaviti s izvršavanjem ostalog koda, ali će na neki način biti obaviješten kada je započeta radnja završila (uspješno ili neuspješno) te će nakon toga izvršiti neki unaprijed zadani kod.

Kako bismo ovo postigli u JavaScriptu možemo koristiti takozvane "callback" funkcije, što znači da ćemo funkciji koja izvršava asinkronu radnju kao parametar proslijediti funkciju koju želimo izvršiti nakon što radnja završi. Iako zvuči jednostavno, ovaj pristup jako brzo zakomplicira kod i postoji niz drugih problema te se zbog toga uveo specijalni objekt "Promise" koji asinkrono programiranje čini puno lakšim.

1.2.1 Async/await mehanizam

Kako bi se asinkrono programiranje dodatno olakšalo, uveden je sloj apstrakcije oko "Promise" objekta koji nazivamo async/await mehanizam.

Ako JavaScript funkciju označimo ključnom riječi "async", ona će rezultat automatski omotati unutar "Promise" objekta te ga kao takvog vratiti.

Ključnu riječ "await" možemo koristiti samo unutar funkcije označene ključnom riječi "async", a omogućava nam da sačekamo da se "Promise" objekt izvrši te nam vraća njegov rezultat.

2 TypeScript

TypeScript je programski jezik koji proširuje JavaScript time što omogućuje snažno tipiziranje, odnosno svakoj varijabli se pridružuje tip podatka. Tip podatka može se pridružiti implicitno (interno tijekom prevođenja) ili eksplicitno tako da ga navedemo direktno u programskom kodu, ali jednom dodijeljen tip ne može se više mijenjati. Preglednici ne mogu izvršavati TypeScript kod nego se on prvo mora prevesti u JavaScript. Za to koristimo TypeScript prevoditelj (*compiler*). Riječ "compile" uobičajeno koristimo za prevođenje programskog koda direktno u strojni jezik. Iako se u službenoj dokumentaciji koristi riječ "compile" za prevođenje TypeScripta u JavaScript, zapravo se radi o postupku kojeg zovemo "transcompile", odnosno prevođenju iz jednog programskog jezika u drugi programski jezik, iako se obje riječi na hrvatski jezik prevode kao "prevođenje".
Prava snaga TypeScripta dolazi iz toga što je to pravi objektno-orijentirani jezik i pomoću njega na nestrukturirani JavaScript kod možemo primjenjivati objektno-orijentiranu paradigmu i time aplikaciju dići na višu razinu. Naime, tako aplikacije postaju puno lakše održive i skalabilne.

2.1 Dekoratori

Dekoratori su koncept u objektno-orijentiranom programiranju koji nam omogućuje da klasama, svojstvima klase, metodama ili funkcijama te parametrima metoda ili funkcija pridružimo meta-podatke. Te meta-podatke možemo iskoristit da tijekom prevođenja promijenimo ponašanje objekta na koji smo primijenili dekorator, a da ne mijenjamo samu definiciju objekta. Na taj način dekoratori nam omogućuju da određene stvari napravimo na vrlo elegantan način unutar samog programskog koda u što ćemo se uvjeriti u poglavlju 6, tijekom izrade aplikacije.

TypeScript kao objektno-orijentirani programski jezik također podržava dekoratore. U TypeScriptu su dekoratori obične funkcije koje unutar definicije koriste posebne alate za dodavanje meta-podataka. Nakon što napravimo dekorator ili preuzmemo neki već postojeći, možemo ga primijeniti na način da ga navedemo neposredno prije deklaracije objekta na kojeg ga primjenjujemo, ali mu dodamo prefiks @. Ako na primjer imamo klasu *MojaKlasa* i dekorator *MojDekorator*, možemo ga primijeniti kao u Kodu 1.

```
@MojDekorator()
class MojaKlasa{
    .
    .
    .
}
```

Kod 1: Primjena dekoratora

3 Node.js

Node.js je predstavljen 2009. godine kao platforma na kojoj se mogao izvršavati JavaScript kod izvan internet preglednika. Za izvršavanje JavaScript koda Node koristi Google V8 engine koji je dio Google Chrome preglednika.

Node.js omogućuje da pomoću JavaScript-a razvijamo sve vrste aplikacija, a to nam omogućuje da JavaScript sada koristimo kao jezik za izradu poslužiteljskih aplikacija. Kako je JavaScript asinkroni jezik, to poslužiteljima baziranima na Node.js-u daje određenu prednost nad sinkronim poslužiteljima.

3.1 Npm

Filozofija Node.js-a je da sama jezgra bude minimalna, odnosno da sadrži samo osnovne funkcionalnosti. Ako želimo neku dodatnu funkcionalnost moramo ju sami razviti i uključiti u jezgru ili možemo koristiti već razvijene pakete. Kako bi se sam proces uključivanja novih paketa olakšao, predstavljen je upravitelj paketima za Node.js koji se zove Npm (*Node package manager*). Npm je zapravo alat koji služi za cjelokupno upravljanje Node.js projektom pa tako osim što pomoću njega možemo uključivati pakete, omogućuje primjerice generiranje novog projekta i izvršavanje ručno zadanih skripti.

4 Express.js

Express.js je okvir (*framework*) za izradu poslužiteljskih aplikacija na Node.js platformi koristeći JavaScript jezik. Express je zapravo omotač (*wrapper*), odnosno sloj apstrakcije oko HTTP modula koji se nalazi u jezgri Node.js-a. Rad s osnovnim HTTP modulom je vrlo težak, jer je za osnovne stvari potrebno jako puno konfiguracije. Express.js taj cijeli proces olakšava pa nam je za izradu osnovne poslužiteljske aplikacije potrebno svega nekoliko linija koda.

Express je dostupan kao npm paket i vrlo se lako uključuje u Node projekt. Okvir prati filozofiju same Node platforme te mu je jezgra minimalna, ali zato postoji puno paketa koji mu proširuju funkcionalnost i također su dostupni kao npm paketi.

5 HTTP

HTTP (*HyperText Transfer Protocol*) je protokol koji se koristi za razmjenu podataka na *World wide web* mreži. Protokol je baziran na klijent/poslužitelj arhitekturi u kojoj postoje čvorovi koji nešto zahtijevaju i njih zovemo klijenti te čvorovi koji te zahtjeve ispunjavaju i njih zovemo poslužitelji. Poslužitelj stalno osluškuje nove zahtjeve pa kada klijent uputi zahtjev poslužitelj ga obradi i klijentu vrati odgovor koji može biti i pozitivan i negativan.

5.1 HTTP metode

Jedan poslužitelj je sposoban izvršavati različite zahtjeve. Kako bi on znao što točno mora napraviti klijent mu to mora nekako specificirati. Jedna specifikacija je putanja do resursa za kojeg klijent traži zahtjev. Tu putanju zovemo URI. No klijent mora biti još specifičniji, jer s jednim resursom možemo raditi razne radnje. Sljedeća razina specifikacije je HTTP metoda. Postoji mnogo metoda, a najčešće su:

- GET - metoda koja služi za dohvaćanje resursa
- POST - metoda koja služi za stvaranje novog resursa
- PATCH i PUT - metode koje služe za ažuriranje resursa
- DELETE - metoda koja služi za brisanje resursa

5.2 Status

Uobičajeno je da poslužitelj uz odgovor priloži i status odgovora. Status se označava troznamenkastim brojem. Svaki broj označava točno jedan status, a najčešći su:

- 200 - nosi poruku OK, odnosno označava da je sve prošlo u redu
- 201 - nosi poruku CREATED, označava da je resurs kreiran
- 404 - nosi poruku NOT FOUND, označava da resurs nije pronađen

6 NestJS

NestJS ili samo Nest je okvir za izradu poslužiteljskih aplikacija na Node.js platformi. Napisan je u TypeScript-u te je preporučeni jezik za razvoj aplikacija pomoću Nesta također TypeScript, iako je omogućeno i razvijanje pomoću JavaScripta. NestJS je zapravo sloj apstrakcije oko već postojećih okvira za razvoj poslužiteljskih aplikacija na Node.js-u. Preporučeni i pretpostavljeni okvir na kojem Nest radi je Express.js iako se može konfigurirati da radi i na nekom drugom okviru. Razvoj samih Nest aplikacija ne ovisi o okviru u pozadini jer je sučelje koje pruža jednako.

Nest je okvir koji "ima mišljenje", odnosno tvorci NestJS-a nameću način na koji se nešto treba napraviti. To je zapravo dobro, jer će u konačnici sve Nest aplikacije biti izgrađene na vrlo sličan način te ako smo prije razvijali neku Nest aplikaciju, vrlo lako se možemo prilagoditi ako se uključimo u razvoj neke druge aplikacije. Ako uzmemo u obzir to i da je Nest razvijen u TypeScriptu, aplikacije razvijene pomoću ovog okvira su modularne, održive i skalabilne.

Okvir radi na način da nam pruža nekoliko gradivnih blokova, koji se slažu i time čine aplikaciju. U ovom radu ćemo proći kroz te gradivne blokove te ćemo u tu svrhu izgraditi jednu malu aplikaciju za vođenje bilješki. Njene mogućnosti bit će: registracija korisnika, koji će onda moći izrađivati, dohvaćati, uređivati i brisati svoje bilješke te dodavanje posebnih korisnika koji će imati mogućnost dohvaćanja, ažuriranja i brisanja drugih korisnika. Sva pohrana odvijat će se u radnoj memoriji poslužitelja, a ne u bazi podataka.

6.1 Nest CLI

Okvir dolazi s naredbenim sučeljem koje se naziva Nest CLI (*Command Line Interface*). Ono nam omogućava generiranje novog Nest projekta te njegovo upravljanje. Također omogućuje generiranje osnovnih gradivnih blokova tako što stvori datoteke s neophodnim kodom.

Nest CLI dostupan je kao npm paket pa na računalu moramo imati instaliran Node.js. Ako je taj uvjet ispunjen, CLI možemo dohvatiti na način da u terminal ili komandnu liniju na našem računalu upišemo

```
npm install -global @nestjs/cli
```

Ta će naredba Nest CLI instalirati globalno te će nam on uvijek biti dostupan, neovisno o projektu. CLI ćemo korisiti pri izradi aplikacije pa ćemo onda nešto više reći o samim funkcionalnostima.

6.2 Opskrbljivači

6.2.1 Ubrizgavanje ovisnosti

Ubrizgavanje ovisnosti (*Dependency injection*) koncept je objektno orijentiranog programiranja koji nam govori da aplikacije trebamo razvijati na način da sve ovisnosti neke klase (ovisnost je zapravo neka druga klasa) instanciramo negdje izvan te ih ubrizgamo u konstruktor klase kao parametre umjesto da ih instanciramo unutar konstruktora. Kako bi bilo

jasnije konstruirajmo 2 primjera. U prvom primjeru (Kod 2) nećemo koristiti ubrizgavanje ovisnosti, a u drugom (Kod 3) hoćemo.

```
class MyClass{
    private instance1: Class1;
    private instance2: Class2;
    constructor(){
        this.instance1 = new Class1();
        this.instance2 = new Class2();
    }
}
```

Kod 2: Klasa bez ubrizgavanja ovisnosti

```
class MyClass{
    private instance1: Class1;
    private instance2: Class2;
    constructor(instance1: Class1, instance2: Class2){
        this.instance1 = instance1;
        this.instance2 = instance2;
    }
}
```

Kod 3: Klasa s ubrizgavanjem ovisnosti

Aplikacije koje koriste ubrizgavanje ovisnosti mogu se puno lakše testirati i održavati. To dolazi do velikog izražaja kod kompleksnijih aplikacija. Zato je ovaj koncept široko rasprostranjen među svim tipovima aplikacija i svim programskim jezicima.

Ako neka klasa ima puno ovisnosti, očito bi jako kompleksno bilo ručno unositi sve ovisnosti. Zato svaki ozbiljan okvir u nekom obliku postupak instanciranja automatizira. Koriste se takozvana skladišta ovisnosti (*Dependency container*) gdje se nalaze instance svih ovisnosti koje su prethodno na neki način registrirane. Nešto o tome kako to obavlja NestJS reći ćemo u potpotpoglavlju 6.2.2 te potpoglavlju 6.3.

6.2.2 Opskrbiljivači

Opskrbljivači (*Providers*) su jedna skupina gradivnih blokova Nest okvira. Predstavljaju osnovni koncept za izradu Nest aplikacija. Opskrbljivač je blok koji može ubrizgavati ovisnosti. Konkretno, to je obična TypeScript klasa dekorirana `@Injectable()` dekoratorom koji je dio *@nestjs/common* paketa.

```
import { Injectable } from '@nestjs/common';

@Injectable()
export class MyProvider{
    constructor(dependency1: Dependency1, dependency2: Dependency2){}
}
```

Kod 4: Primjer opskrbljivača

Iz Koda 4 vidimo da je to jednostavna klasa koja ima svoje ovisnosti. Kako bi NestJS okvir mogao te ovisnosti ubrizgati umjesto nas moramo klasu dekorirati dekoratorom `@Injectable()`. Kao što smo naveli u potpotpoglavlju 6.2.1, Nest koristi skladište ovisnosti,

gdje se nalaze sve potrebne instance ovisnosti koje treba ubrizgati. Taj će dekorator prilikom prevođenja klasi dodijeliti meta-podatke koji sadržavaju informaciju o tipu ovisnosti na svakoj poziciji unutar parametara konstruktora koji će poslužiti kao identifikator ovisnosti i okviru će omogućiti da zaključi koju ovisnost iz skladišta treba ubrizgati. Kada taj dekorator ne bismo koristili, informacija o tipovima podataka bi se izgubila što bi onemogućilo okviru da ubrizga ovisnosti umjesto nas. Ključnu riječ `export` koristimo kako bismo klasu kasnije mogli uključiti negdje drugdje kao ovisnost.

6.3 Moduli

Moduli (*Modules*) su osnovna gradivna jedinica u NestJS okviru. Služe nam da kod organiziramo u logičke cjeline. Svi gradivni blokovi koji čine jednu logičku cjelinu zajedno čine modul. Kako bismo sve te blokove povezali koristimo TypeScript klasu koja ih registrira, odnosno uključuje u projekt. Zato kada se referiramo na modul, zapravo se referiramo na tu klasu i kada govorimo o Nest modulu, govorimo o jednoj takvoj klasi. Svaka Nest aplikacija mora imati barem jedan modul kako bi se aplikacija mogla inicijalizirati. Taj modul zovemo korijenski modul koji zajedno s ostalim modulima čini strukturu povezanoga grafa.

Modul je obična TypeScript klasa dekorirana `@Module()` dekoratorom koji je dio *@nestjs/common* paketa. U Kodu 5 vidljiva je osnovna struktura modula.

```
1 import { Module } from '@nestjs/common';
2
3 @Module({
4     providers: [],
5     imports: [],
6     exports: [],
7     controllers: []
8 })
9 export class MyModule{}
```

Kod 5: Struktura NestJS modula

Iz Koda 5 vidimo da je modul zapravo prazna klasa, koja služi samo da bi se mogli referirati na modul unutar ostatka aplikacije. Sve ostalo odvija se unutar `@Module()` dekoratora koji će klasi dodijeliti meta-podatke o blokovima koje koristimo unutar modula te će se tijekom prevođenja konstruirati potrebna klasa. To je dekorator koji kao parametar prima objekt koji ima četiri opcionalna svojstva koja su zapravo polja.

6.3.1 Polje providers

Kada smo govorili o ubrizgavanju ovisnosti i o skladištu ovisnosti, rekli smo da sve ovisnosti na neki način moraju biti registrirane ako želimo da okvir bude u mogućnosti ubrizgati ih umjesto nas. Za to nam služi polje `providers`. U njemu navodimo sve opskrbljivače koji su dio modula i moramo ih negdje ubrizgati te ih na taj način registriramo unutar okvira. Uočimo da i ovisnosti tipa `Dependency1` i `Dependency2` koje se pojavljuju u Kodu 4 također prethodno moraju biti registrirane kako bi ih okvir znao ubrizgati.

Osim klasičnih opskrbiljivača o kojima smo govorili, možemo registrirati i prilagođene opskrbljivače kojima identifikator neće biti tip podatka, odnosno ime klase, nego proizvoljno zadani niz znakova (*string*). Možemo ih registrirati na način da u polje `providers` dodamo objekt oblika

```
{provide: 'dependency1', useClass: Dependency1}
```

Da bismo ubrizgali takav opskbljivač moramo koristiti dekorator parametra funkcije `@Inject()` koji je dio *@nestjs/common* paketa. Kod 4 bi tada postao Kod 6.

```
1 import { Injectable } from '@nestjs/common';
2
3 @Injectable()
4 export class MyProvider{
5     constructor(
6         @Inject('dependency1') dependency1: Dependency1,
7         dependency2: Dependency2
8     ){}
9 }
```

Kod 6: Ubrizgavanje prilagođenog opskrbljivača

Umjesto svojstva `useClass` možemo koristiti `useValue` ukoliko želimo ubrizgati neku konkretnu vrijednost ili objekt, `useFactory` ukoliko želimo ovisnost instancirati pomoću funkcije ili `useExisting` ukoliko već postojećem opskrbljivaču želimo dodati još jedno ime.

6.3.2 Polja imports i exports

Pri razvoju kompleksnije aplikacije može postojati potreba da opskrbljivač koji je dio jednog modula koristimo u nekom drugom modulu. Kako bi to bilo moguće koriste se polja `imports` i `exports`. Ukoliko postoji potreba korištenja opskrbljivača izvan modula, navodimo ga unutar polja `exports`. Kako bismo ga mogli koristiti u nekom drugom modulu, pomoću polja `imports` uvozimo cijeli modul u kojem se željeni opskrbljivač nalazi. Također, pomoću polja `exports` možemo izvesti module koje smo prethodno uvezli.

Na primjer, ako imamo modul `A` koji sadrži opskrbljivač `AProvider`, modul `B` koji sadrži opskrbljivač `BProvider`, a kojem je za rad potreban `AProvider` te modul `C` koji ima opskrbljivač `CProvider` kojem su za rad potrebni i `AProvider` i `BProvider`. Kako bi Nest bio u mogućnosti ubrizgati sve ovisnosti, u modul `B` moramo uključiti modul `A`, ali prije toga unutar modula `A` moramo izvesti `AProvider`. U modul `C` moramo uključiti i modul `A` i modul `B` iz kojeg izvozimo `BProvider`. Postoji i mogućnost da iz modula `B` uz `BProvider` izvezemo i modul `A` kojeg smo prethodno uvezli. Tada u modul `C` moramo uvesti samo modul `B`. Ako pretpostavimo da su opskrbljivači definirani na opisani način i da su nam dostupni, u Kodu 7 prikazano je kako bi to izgledalo.

```
1 import { Module } from '@nestjs/common';
2
3 @Module({
4     providers: [AProvider],
5     imports: [],
6     exports: [AProvider],
7     controllers: []
8 })
```

```
9  export class A{}
10
11 @Module({
12     providers: [BProvider],
13     imports: [A],
14     exports: [A, BProvider],
15     controllers: []
16 })
17 export class B{}
18
19 @Module({
20     providers: [CProvider],
21     imports: [B],
22     exports: [],
23     controllers: []
24 })
25 export class C{}
```

Kod 7: Uvoz i izvoz unutar modula

Kada `CProvider` ne bi direktno ovisio o `AProvider` opskrbljivaču, modul `A` ne bismo morali ni na koji način uvoziti u modul `C` jer je za instanciranje `BProvider` opskrbljivača, koji ovisi o `AProvider`, odgovoran modul `B`.

6.3.3 Polje controllers

Polje `controllers` je zaduženo za registraciju kontrolera (*controller*) koji su također jedan gradivni blok NestJS okvira, ali nešto više o njima reći ćemo u potpoglavlju 6.4.

6.3.4 Aplikacija

Predstavili smo osnovni gradivni blok pa sada to možemo primijeniti i krenuti s izradom aplikacije. Krenut ćemo s dijelom za izradu bilješki, a korisnike ćemo dodati kasnije.

Prvo moramo generirati Nest projekt. To ćemo napraviti pomoću Nest CLI alata, odnosno u terminal ćemo upisati: `nest new notes`

"notes" je ime aplikacije, a umjesto toga tu možemo upisati što god želimo. Nakon što izvršimo ovu naredbu, CLI će nas pitati koji upravitelj paketima želimo koristiti, mi ćemo odabrati npm. Nakon toga će CLI generirati projekt. Ako se pozicioniramo unutar `src` direktorija, pronaći ćemo 5 datoteka. Od njih su bitne samo `app.module.ts` i `main.ts`. Ostale datoteke ćemo obrisati jer ih CLI generira uz pretpostavku da će aplikacija biti vrlo jednostavna. Nakon što ih obrišemo, potrebno je prilagoditi datoteku `app.module.ts` kao što je prikazano u Kodu 8.

`src/app.module.ts`

```
1 import { Module } from '@nestjs/common';
2
3 @Module({
4   imports: [],
5   controllers: [],
6   providers: [],
7 })
8 export class AppModule {}
```

Kod 8: Prilagođeni korijenski modul

Kod unutar `main.ts` zadužen je za instanciranje aplikacije, a unutar `app.module.ts` nalazi se korijenski modul. On neće imati svoje opskrbljivače, nego će samo uvoziti preostale module kako bi ih povezao.

Sustav za upravljanje bilješkama će biti jedna cjelina pa ima smisla da svu logiku vezanu uz njih smjestimo u jedan modul. Modul možemo generirati pomoću CLI alata koji će stvoriti datoteku s predloškom za modul te će ga uključiti unutar korijenskog modula, ali ovaj modul ćemo napraviti sami.

Unutar `src` direktorija napravimo novi direktoriji `notes`, a unutar tog novog direktorija napravimo datoteku `notes.module.ts`. U toj datoteci napišimo kod prikazan u Kodu 9, a koji će predstavljati modul. Običaj je da svakom gradivnom bloku u ime dodamo sufiks koji govori o kojem bloku se radi pa zato ovaj modul nazivamo `NotesModule`.

src/notes/notes.module.ts

```
1 import { Module } from '@nestjs/common';
2
3 @Module({
4     providers: [],
5     imports: [],
6     exports: [],
7     controllers: []
8 })
9 export class NotesModule{}
```

Kod 9: Kreiranje modula za bilješke

Sada ovaj modul još moramo uključiti u korijenski modul kako bismo ga mogli koristiti. Prilagodimo korijenski modul da izgleda kao Kod 10.

src/app.module.ts

```
1 import { Module } from '@nestjs/common';
2 import { NotesModule } from './notes/notes.module';
3
4 @Module({
5   imports: [NotesModule],
6   controllers: [],
7   providers: [],
8 })
9 export class AppModule {}
```

Kod 10: Uključivanje modula za bilješke u korijenski modul

6.4 Kontroleri

Kontroleri (*Controllers*) su gradivne jedinice unutar NestJS okvira koje služe za komunikaciju s klijentom. Oni zaprimaju zahtjeve, obrađuju ih ili ih prosljeđuju na obradu te šalju odgovore. Kada smo govorili o HTTP-u rekli smo da klijent specificira svoj zahtjev tako što ga šalje na određeni URI, odnosno rutu i s određenom metodom. Svaki će kontroler biti zadužen za jednu rutu, a unutar njega ćemo definirati njegovo ponašanje ovisno o metodi zahtjeva.

U Nestu, kontroler je TypeScript klasa dekorirana `@Controller()` dekoratorom koji je dio *@nestjs/common* paketa. Primjer kontrolera možemo vidjeti u Kodu 11.

```
1 import { Controller, Get } from '@nestjs/common';
2
3 @Controller('myroute')
4 export class MyController {
5     @Get()
6     getSomething(): string{
7         return 'something';
8     }
9 }
```

Kod 11: Primjer kontrolera

Iz Koda 11 vidimo da `@Controller()` dekorator kao parametar prima string koji predstavlja rutu za koju je kontroler zadužen. Ruta je zapisana relativno u odnosu na sjedište naše aplikacije.

Zahtjeve obrađujemo funkcijama unutar klase dekoriranim odgovarajućim dekoratorom. Nest nam za svaku HTTP metodu pruža odgovarajući dekorator koji se nalazi u *@nestjs/common* paketu. U ovom slučaju ako pošaljemo HTTP GET zahtjev na rutu `/myroute` dobit ćemo odgovor u obliku stringa, konkretno string `'something'`.

Dekoratori metoda imaju opcionalni parametar, koji proširuje rutu. Ako neku funkciju klase `MyController` dekoriramo dekoratorom `@Get('something')` ona će biti odgovorna za sve zahtjeve koje dođu na rutu `/myroute/something` HTTP metodom GET.

6.4.1 Parametrizirane rute

Često želimo rutu proširiti dinamički, odnosno da jedan njen dio bude parametar. To možemo postići tako da dijelu stringa kojeg prosljeđujemo dekoratoru, a za kojeg želimo da bude dinamičan damo prefiks : (dvotočka).

Kako bismo taj parametar mogli iskoristiti u funkciji moramo ga primiti kao parametar funkcije, ali dekoriran dekoratorom `@Param()` koji je također dio *@nestjs/common* paketa. Taj dekorator prima opcionalni parametar tipa `string` koji označava koji točno parametar želimo izvući iz rute, a identifikator mu je string koji smo označili s :. Ukoliko mu ne proslijedimo niti jedan identifikator, bit će pokupljeni svi parametri u obliku JavaScript objekta, odnosno TypeScript klase ako smo specificirali tip parametra funkcije. Kako bi bilo jasnije pogledajmo primjere koji su prikazani u Kodovima 12 i 13.

```
1 import { Controller, Get, Param } from '@nestjs/common';
2
3 @Controller('myroute')
4 export class MyController {
5     @Get(':id')
6     getById(@Param('id') idOfSomething: number): number{
7         return idOfSomething;
8     }
9 }
```

Kod 12: Primjer kontrolera koji vraća parametar rute

Ako u Kodu 12 pošaljemo HTTP GET zahtjev na rutu npr. `/myroute/1`, dobit ćemo odgovor broj `1`.

```
import { Controller, Get, Param } from '@nestjs/common';

@Controller('myroute')
export class MyController {
    @Get(':id')
    getById(@Param() params: {id: number}): {id: number}{
        return params;
    }
}
```

Kod 13: Primjer kontrolera koji vraća sve parametre rute

Ako u Kodu 13, pošaljemo HTTP GET zahtjev na rutu npr. `/myroute/1`, dobit ćemo odgovor objekt `{id: 1}`.

Neke od preostalih mogućnosti kontrolera upoznat ćemo kroz izradu aplikacije u potpotpoglavlju 6.4.2.

6.4.2 Aplikacija

Upoznali smo se Nest kontrolerima, a sada ćemo ih primijeniti na našu aplikaciju. Napravit ćemo kontroler pomoću kojeg ćemo moći kreirati nove bilješke, dohvaćati ih, uređivati te brisati. Kako bismo napravili kontroler koristit ćemo Nest CLI koji će za nas generirati datoteku s predloškom za kontroler koji će dekoratoru već proslijediti ime rute koje želimo. Dodatno, CLI će uključiti kontroler u `NotesModule` jer ćemo mu dodijeliti isto ime.

U terminalu izvršimo komandu `nest generate controller notes --no-spec`. Zastavica `--no-spec` spriječit će CLI u generiranju datoteka koje nećemo koristiti, a odnose se na testiranje. Unutar `src/notes` direktorija napravimo novu datoteku `note.model.ts` koja će predstavljati model naše bilješke. Bilješka će za sada imati identifikator koji ćemo zvati `id` i sadržaj `content`. Sadržaj `note.model.ts` datoteke prikazan je u Kodu 14.

`src/notes/note.model.ts`

```
export class Note{
    id: number;
    content: string;
}
```

Kod 14: Model bilješke

Sada ćemo u klasu koja predstavlja kontroler enkapsulirati polje bilješki, koje će biti skladište bilješki te broj `currentId` koji služi za praćenje broja napravljenih bilješki, kako bismo im mogli dodijeliti pravilan identifikator. U Kodu 15 prikazano je kako to postići.

`src/notes/notes.controller.ts`

```
import { Controller } from '@nestjs/common';
import { Note } from './note.model';

@Controller('notes')
export class NotesController {
    private notes: Note[] = [];
```

```
7       private currentId: number = 0;
8   }
```

Kod 15: Enkapsulacija skladišta i brojača unutar kontrolera

Prvo ćemo implementirati metodu koja će kreirati nove bilješke, a zahtjeve će primati metodom POST. Zahtjev će imati tijelo (*body*) koje će sadržavati sadržaj bilješke. Kako bismo dohvatili tijelo, slično kao kada smo dohvaćali parametre, koristit ćemo `@Body()` dekorator koji radi na isti način kao i `@Param()` dekorator. Nakon što napravimo novu bilješku, kao odgovor ćemo ju vratiti klijentu. Nest za uspješno izvršene POST zahtjeve, automatski dodjeljuje status 201, odnosno "kreirano". Ne želimo da se naša aplikacija zamrzne dok ne ispuni zahtjev pa ćemo metode naših konrolera napraviti asinkronima. U klasu kontrolera dodajemo metodu prikazanu u Kodu 16.

```
1   @Post()
2   async create(@Body('content') content: string): Promise<Note>{
3       const note: Note = new Note();
4       note.id = this.currentId++;
5       note.content = content;
6       this.notes.push(note);
7       return note;
8   }
```

Kod 16: Metoda za kreiranje bilješki unutar kontrolera

Zatim ćemo implementirati metodu koja će vraćati sve bilješke, a zahtjeve će primati GET metodom. Ona će klijentu vratiti polje svih bilješki. Dodatno želimo mogućnost pretrage. Klijent će u zahtjev moći uključiti riječ koju bilješka mora sadržavati da bi bila dio odgovora. Klijent će tu riječ uključiti pomoću takozvanih "query" parametara. Kako bismo dohvatili te parametre, koristit ćemo `@Query()` dekorator koji radi na isti način kao i `@Param()` i `@Body()` dekoratori. U klasu kontrolera dodajemo metodu prikazanu u Kodu 17.

```
1   @Get()
2   async get(@Query('search') term?: string): Promise<Note[]>{
3       if(term){
4           return this.notes.filter((note: Note) => note.content.includes(
                term));
5       }
6       return this.notes;
7   }
```

Kod 17: Metoda za dohvaćanje svih bilješki unutar kontrolera

Dodatno, želimo dohvaćati određenu bilješku pomoću njenog indentifikatora. To ćemo napraviti koristeći parametriziranu rutu, o čemu smo već govorili u potpotpoglavlju 6.4.1. Klijent će u zahtjev uključiti identifikator, a odgovor će sadržavati odgovarajuću bilješku. Ukoliko bilješka s tim identifikatorom ne postoji, vratit ćemo grešku, a status odgovora će biti 404, odnosno "nije pronađeno". Kako bismo to postigli, dovoljno je da "bacimo" iznimku (*throw exception*) `NotFoundException` koja je dio *@nestjs/common* paketa. Nestov sustav rukovanja iznimkama će sve iznimke koje do vraćanja odgovora nisu uhvaćene uhvatiti i

vratiti lijepo formatiranu i smislenu poruku te odgovarajući status. U kontroler dodajemo metodu prikazanu u Kodu 18.

```
@Get(':id')
async getById(@Param('id') id: number): Promise<Note>{
    const note: Note = this.notes.find((note: Note) => note.id == id);
    if(!note){
        throw new NotFoundException;
    }
    return note;
}
```

Kod 18: Metoda za dohvaćanje bilješke pomoću identifikatora unutar kontrolera

Sljedeće što ćemo implementirati je metoda za ažuriranje postojeće bilješke metodom PATCH. Klijent će kao parametar proslijediti identifikator, a u tijelo zahtjeva će staviti novi sadržaj. Odgovor će biti ažurirana bilješka ili, slično kao kod dohvaćanja, greška ukoliko takva bilješka ne postoji. U kontroler dodajemo metodu prikazanu u Kodu 19.

```
@Patch(':id')
async update(@Param('id') id: number, @Body('content') content: string):
   Promise<Note>{
    const note: Note = this.notes.find((note: Note) => note.id == id);
    if(!note){
        throw new NotFoundException;
    }
    note.content = content;
    return note;
}
```

Kod 19: Metoda za ažuriranje bilješke unutar kontrolera

Preostalo je još implementirati metodu za brisanje. Klijent će poslati zahtjev metodom DELETE te će kao parametar proslijediti identifikator. Ukoliko bilješka, postoji metoda će ju obrisati, a inače se neće dogoditi ništa. Odgovor će uvijek biti prazan pa je prikladan status 204, odnosno "nema sadržaja". Kako bismo primijenili taj status koristit ćemo `HttpCode()` dekorator. U kontroler dodajemo metodu prikazanu u Kodu 20. U konačnici `notes.controller.ts` datoteka treba izgledati kako je prikazano u Kodu 21.

```
@Delete(':id')
@HttpCode(204)
async delete(@Param('id') id: number): Promise<void>{
    this.notes = this.notes.filter((note: Note) => note.id != id);
}
```

Kod 20: Metoda za brisanje bilješke unutar kontrolera

`src/notes/notes.controller.ts`

```
import { Controller, Post, Body, Get, Param, Query, NotFoundException,
   Patch, Delete, HttpCode } from '@nestjs/common';
import { Note } from './note.model';

@Controller('notes')
export class NotesController {
    private notes: Note[] = [];
    private currentId: number = 0;
```

```
8
9       @Post()
10      async create(@Body('content') content: string): Promise<Note>{
11          const note: Note = new Note();
12          note.id = this.currentId++;
13          note.content = content;
14          this.notes.push(note);
15          return note;
16      }
17
18      @Get()
19      async get(@Query('search') term?: string): Promise<Note[]>{
20          if(term){
21              return this.notes.filter((note: Note) => note.content.includes
                    (term));
22          }
23          return this.notes;
24      }
25
26      @Get(':id')
27      async getById(@Param('id') id: number): Promise<Note>{
28          const note: Note = this.notes.find((note: Note) => note.id == id);
29          if(!note){
30              throw new NotFoundException;
31          }
32          return note;
33      }
34
35      @Patch(':id')
36      async update(@Param('id') id: number, @Body('content') content: string
            ): Promise<Note>{
37          const note: Note = this.notes.find((note: Note) => note.id == id);
38          if(!note){
39              throw new NotFoundException;
40          }
41          note.content = content;
42          return note;
43      }
44
45      @Delete(':id')
46      @HttpCode(204)
47      async delete(@Param('id') id: number): Promise<void>{
48          this.notes = this.notes.filter((note: Note) => note.id != id);
49      }
50  }
```

Kod 21: Prva verzija kontrolera bilješki

6.5 Servisi

Servisi (*Services*) su gradivni blokovi u kojima se nalazi poslovna logika naše aplikacije. Oni obrađuju podatke koje im prosljeđuje kontroler (odnosno klijent) te ih pohranjuju na odgovarajući način. Oni su posrednik između kontrolera i dijela aplikacije koji je zadužen za pohranu podataka u bazu podataka, iako samo servisi mogu poslužiti kao skladište podataka.

U NestJS okviru servisi pripadaju skupini opskrbljivača. Konkretno, oni su obični ops-

krbljivači koji nemaju nikakvih dodatnih zahtjeva niti novih specifičnosti. Primjer servisa prikazan je u Kodu 22.

```
import { Injectable } from '@nestjs/common';

@Injectable()
export class MyService{
    private names: string[];
    constructor(){}
    create(name: string){
        this.names.push(name);
    }
}
```

Kod 22: Primjer servisa

6.5.1 Aplikacija

Trenutno se sva logika naše aplikacije nalazi unutar kontrolera, što nije dobra praksa. Kontroler bi trebao samo zaprimati zahtjeve i slati odgovore, a sva logika bi se trebala odvijati negdje drugdje. Stoga ćemo tu logiku premjestiti u servis. Takav način izrade aplikacije zovemo slojevita arhitektura jer razne zadaće unutar aplikacije smještamo u posebne slojeve koji su poredani hijerarhijski.

Pomoću CLI alata generirat ćemo servis. CLI će ga automatski uključiti u polje `imports` unutar modula `NotesModule` te ga tako registrirati kao opskrbljivača. U terminalu izvršimo sljedeću naredbu: `nest generate service notes --no-spec`. U stvorenu datoteku ubacit ćemo Kod 23. Nakon toga možemo prilagoditi kontroler kao što je prikazano u Kodu 24.

src/notes/notes.service.ts

```
import { Injectable, NotFoundException } from '@nestjs/common';
import { Note } from './note.model';

@Injectable()
export class NotesService {
    private notes: Note[] = [];
    private currentId: number = 0;

    async create(content: string): Promise<Note>{
        const note: Note = new Note();
        note.id = this.currentId++;
        note.content = content;
        this.notes.push(note);
        return note;
    }

    async get(term?: string): Promise<Note[]>{
        if(term){
            return this.notes.filter((note: Note) => note.content.includes
                (term));
        }
        return this.notes;
    }

    async getById(id: number): Promise<Note>{
```

```
25          const note: Note = this.notes.find((note: Note) => note.id == id);
26          if(!note){
27              throw new NotFoundException;
28          }
29          return note;
30      }
31
32      async update(id: number, content: string): Promise<Note>{
33          const note: Note = await this.getById(id);
34          note.content = content;
35          return note;
36      }
37
38      async delete(id: number): Promise<void>{
39          this.notes = this.notes.filter((note: Note) => note.id != id);
40      }
41  }
```

Kod 23: Servis bilješki

U Kodu 23 uočimo ključnu riječ `await` u metodi `update`. Kako je metoda `getById` asinkrona, ključnom riječi `await` govorimo da želimo sačekati da se ta radnja izvrši. To ipak neće blokirati našu aplikaciju jer je metoda `update` asinkrona pa metoda koja ju je pozvala neće blokirati izvođenje.

src/notes/notes.controller.ts

```
1  import { Controller, Post, Body, Get, Param, Query, Patch, Delete,
       HttpCode } from '@nestjs/common';
2  import { Note } from './note.model';
3  import { NotesService } from './notes.service';
4
5  @Controller('notes')
6  export class NotesController {
7      constructor(private notesService: NotesService){}
8
9      @Post()
10     async create(@Body('content') content: string): Promise<Note>{
11         return this.notesService.create(content);
12     }
13
14     @Get()
15     async get(@Query('search') term?: string): Promise<Note[]>{
16         return this.notesService.get(term);
17     }
18
19     @Get(':id')
20     async getById(@Param('id') id: number): Promise<Note>{
21         return this.notesService.getById(id);
22     }
23
24     @Patch(':id')
25     async update(@Param('id') id: number, @Body('content') content: string
           ): Promise<Note>{
26         return this.notesService.update(id, content);
27     }
28
29     @Delete(':id')
```

```
30      @HttpCode(204)
31      async delete(@Param('id') id: number): Promise<void>{
32          await this.notesService.delete(id);
33      }
34  }
```

Kod 24: Druga verzija kontrolera bilješki

U Kodu 24 uočimo da kod metoda u kojima direktno vraćamo rezultat iz servisa ne koristimo ključnu riječ `await` jer metode kontrolera vraćaju tip podatka `Promise`, a to je upravo ono što vraćaju i metode servisa. Naime, kada koristimo `await` na nekoj asinkronoj metodi (koja onda mora vraćati tip podatka `Promise`) tip podatka će se promijeniti u osnovni tip podatka koji je bio omotan unutar `Promise` tipa.

6.6 Cijevi

Cijevi (*Pipes*) su gradivni blokovi koji dolaze iz skupine opskrbljivača. U Nest aplikacijama one nam služe za validaciju ili transformaciju podataka. Primjenjuju se isključivo na parametre metoda unutar kontrolera, odnosno na podatke koje nam dostavlja klijent i to netom prije nego se sama metoda počne izvršavati.

Cijev je opskrbljivač koji mora implementirati `PipeTransform` sučelje koje je dio *@nestjs/common* paketa, odnosno mora implementirati funkciju `transform` koja prima 2 parametra, `value` i `metadata`. `value` je vrijednost na koju primjenjujemo cijev, a `metadata` sadrži meta-podatke metode na koju primjenjujemo cijev. Primjer cijevi prikazan je u Kodu 25. Kako samo ime cijevi sugerira, ona ne radi ništa, već vraća nepromijenjenu vrijednost.

```
1  import { PipeTransform, Injectable, ArgumentMetadata } from '@nestjs/
       common';
2
3  @Injectable()
4  export class DoNothingPipe implements PipeTransform {
5
6    transform(value: any, metadata: ArgumentMetadata) {
7      return value;
8    }
9  }
```

Kod 25: Primjer cijevi

Cijev se može primijeniti na razini cijelog kontrolera, na razini metode te na razini parametra metode. Ako ju primijenimo na razini kontrolera ona će djelovati na svaki parametar svake metode tog kontrolera. Ako ju primijenimo na razini metode djelovat će na sve parametre te metode, a na razini parametra će djelovati samo na taj parametar. Kako bismo cijev primijenili na kontroler ili metodu moramo ju dekorirati `@UsePipes()` dekoratorom, koji je dio *@nestjs/common* paketa, a kao parametre prima proizvoljan broj cijevi koje će se izvršavati redoslijedom kako su navedene. Parametar tog dekoratora može biti instanca cijevi, a može biti i referenca na klasu cijevi, a u tom slučaju će ju okvir instancirati sam. Ako želimo primijeniti cijev na razini parametra metode, na identičan način ju proslijedimo kao

parametar dekoratora `@Param()`, `@Query()` ili `@Body()`. Primjer primjene cijevi prikazan je u Kodu 26.

```
import { Controller, Get, Query UsePipes } from '@nestjs/common';

@Controller('myroute')
@UsePipes(DoNothingPipe)
export class MyController {
    @Get()
    getSomething(@Query() query: any): string{
        return 'something';
    }
}
```

Kod 26: Primjena cijevi na razini kontrolera

NestJS okvir dolazi s nekoliko već definiranih cijevi koje se nalaze u *@nestjs/common* paketu. Cijev `ValidationPipe` koja se koristi za validaciju parametara te cijevi `ParseIntPipe`, `ParseBoolPipe`, `ParseArrayPipe` i `ParseUUIDPipe` koje služe za transformaciju, odnosno parsiranje u određeni tip podatka jer svi parametri rute i "query" parametri u metodu ulaze kao tip `string` te `DefaultValuePipe` koji `null` i `undefined` parametrima dodjeljuje zadanu vrijednost u ovisnosti o tipu podatka.

6.6.1 Aplikacija

U potpoglavlju 6.6 je rečeno da je tip svih parametara rute unutar aplikacije `string`, no identifikator bilješke je broj, a aplikacija i dalje radi ono što treba. Pogledajmo `getById` metodu u servisu bilješki (Kod 27).

src/notes/notes.service.ts

```
async getById(id: number): Promise<Note>{
    const note: Note = this.notes.find((note: Note) => note.id == id);
    if(!note){
        throw new NotFoundException;
    }
    return note;
}
```

Kod 27: getById metoda unutar servisa bilješki

Razlog zbog kojeg aplikacija radi na valjani način je taj što općenito u JavaScript-u pa i u TypeScript-u vrijedi `'1' == 1` je istinito. Primjena `==` vrijedi i općenito – odnosno `string` reprezentacija broja jednaka je `number` reprezentaciji tog istog broja. Kada bismo usporedbu radili s `===` tada te reprezentacije ne bi bile jednake. U ovako jednostavnoj aplikaciji nam to ne pravi problem, ali u nekim većim aplikacijama bi moglo. Zato je dobra praksa parsirati parametare u odgovarajući tip. Iako je tip podatka parametra `number`, ovo se odvija u vrijeme izvođenja aplikacije, kada je naš kod zapravo pretvoren u JavaScript gdje informacija o tipu nije sačuvana pa je moguće da taj parametar bude `string` ukoliko je kao takav proslijeđen.

To ćemo popraviti tako što ćemo u naš kontroler dodati cijev za parsiranje u broj i to na razini parametra metode. Kontroler treba prilagoditi kao što je prikazano u Kodu 28.

Nakon toga možemo prilagoditi metode `getById` i `delete` unutar servisa, odnosno dodati im pravilnu usporedbu, kao što je prikazano u Kodu 29.

src/notes/notes.controller.ts

```
import { Controller, Post, Body, Get, Param, Query, Patch, Delete,
    HttpCode, ParseIntPipe } from '@nestjs/common';
import { Note } from './note.model';
import { NotesService } from './notes.service';

@Controller('notes')
export class NotesController {
    constructor(private notesService: NotesService){}

    @Post()
    async create(@Body('content') content: string): Promise<Note>{
        return this.notesService.create(content);
    }

    @Get()
    async get(@Query('search') term?: string): Promise<Note[]>{
        return this.notesService.get(term);
    }

    @Get(':id')
    async getById(@Param('id', ParseIntPipe) id: number): Promise<Note>{
        return this.notesService.getById(id);
    }

    @Patch(':id')
    async update(@Param('id', ParseIntPipe) id: number, @Body('content')
        content: string): Promise<Note>{
        return this.notesService.update(id, content);
    }

    @Delete(':id')
    @HttpCode(204)
    async delete(@Param('id', ParseIntPipe) id: number): Promise<void>{
        await this.notesService.delete(id);
    }
}
```

Kod 28: Treća verzija kontrolera bilješki

src/notes/notes.service.ts

```
async getById(id: number): Promise<Note>{
    const note: Note = this.notes.find((note: Note) => note.id === id);
    if(!note){
        throw new NotFoundException;
    }
    return note;
}

async delete(id: number): Promise<void>{
    this.notes = this.notes.filter((note: Note) => note.id !== id);
}
```

Kod 29: Prilagođene getById i delete metode unutar servisa bilješki

6.7 Presretači

Presretači (*Interceptors*) su gradivni blokovi iz skupine opskrbljivača koji omogućuju presretanje i modificiranje zahtjeva i odgovora. Oni su opskrbljivači koji implementiraju `NestInterceptor` sučelje, odnosno metodu `intercept()`. Metoda `intercept()` prima 2 parametra – `ExecutionContext` te `CallHandler`. `ExecutionContext` sadrži informacije o parametrima koji su proslijeđeni metodi koja je zadužena za obradu zahtjeva te sam zahtjev. `CallHandler` sadrži metodu `handle()` koja će izvršiti originalnu metodu zaduženu za obradu zahtjeva (onu iz kontrolera). U nekom trenutku moramo pozvati metodu `handle()` ako želimo da se zahtjev izvrši do kraja.

Kako bismo mogli modificirati odgovor, koristi se `rxjs`, biblioteka za reaktivno programiranje u JavaScriptu. Metoda `handle()` vraća objekt tipa `Observable` iz `rxjs` biblioteke na koji se proizvoljno mogu ulančavati i ostali operatori s dodatnom logikom koji će se izvršiti nakon što metoda zadužena za zahtjev vrati odgovor.

Presretače, slično kao i cijevi, možemo primijeniti na razini kontrolera i metode koristeći `@UseInterceptors()` dekorator iz *@nestjs/common* paketa, koji radi na identičan način kao i `@UsePipes()` dekorator. Njihov rad vidjet ćemo u sljedećem potpotpoglavlju 6.7.1.

6.7.1 Aplikacija

Prije nego li počnemo primjenjivati presretače na našu aplikaciju kreirat ćemo cijeli modul za korisnike.

Krenimo s generiranjem potrebnih datoteka. U terminalu ćemo izvršiti sljedeće 3 naredbe ovim redoslijedom:

```
nest generate module users
nest generate controller users --no-spec
nest generate service users --no-spec
```

Time smo generirali potrebne datoteke i registrirali ih unutar modula.

Korisnike ćemo moći kreirati, dohvaćati, uređivati, prijavljivati te brisati. Kasnije ćemo neke mogućnosti ograničiti na samo neke korisnike. Prijava korisnika će se odvijati na način da će klijent poslati korisničko ime i lozinku te ako su oni točni, vratit ćemo korisnikov identifikator kako bi ga klijent mogao uključiti u buduće zahtjeve i potvrditi svoj identitet. Aplikacije koje se nalaze u produkciji koriste složenije identifikatore jer je to radnja koja zahtjeva veliku razinu sigurnosti. Mi ćemo pak u tu svrhu koristiti općeniti identifikator korisnika unutar skladišta korisnika. Kreirajmo prvo model korisnika u datoteci `src/users/user.model.ts` koristeći Kod 30.

`src/users/user.model.ts`

```
1 export class User{
2     id: number;
3     username: string;
4     password: string;
5     role: 'admin' | 'user';
6 }
```

Kod 30: Model korisnika

Svojstvo `role`, odnosno uloga korisnika, koristit će nam kasnije kada budemo željeli ograničiti pristup nekim rutama. Lozinku ćemo pohranjivati u onom obliku u kojem ju dobijemo, ali to nije preporučeno za aplikacije koje se nalaze u produkciji, već ju ondje šifriranu pohranjujemo.

Kreirajmo servis za korisnike tako što ćemo u odgovarajuću datoteku ubaciti Kod 31, nakon čega možemo implementirati i kontroler kao što je prikazano u Kodu 32.

src/users/users.service.ts

```
1  import { Injectable, NotFoundException, UnauthorizedException } from '
       @nestjs/common';
2  import { User } from './user.model';
3
4  @Injectable()
5  export class UsersService {
6      private users: User[] = [];
7      private currentId: number = 0;
8
9      async create(username: string, password: string): Promise<User>{
10         const user: User = new User();
11         user.id = this.currentId++;
12         user.username = username;
13         user.password = password;
14         this.users.push(user);
15         return user;
16     }
17
18     async get(): Promise<User[]>{
19         return this.users;
20     }
21
22     async getById(id: number): Promise<User>{
23         const user: User = this.users.find((user: User) => user.id === id)
               ;
24         if(!user){
25             throw new NotFoundException;
26         }
27         return user;
28     }
29
30     async update(id: number, username?: string, password?: string):
           Promise<User>{
31         const user: User = await this.getById(id);
32         if(username){
33             user.username = username;
34         }
35         if(password){
36             user.password = password;
37         }
38         return user;
39     }
40
41     async delete(id: number): Promise<void>{
42         this.users = this.users.filter((user: User) => user.id !== id);
43     }
44
45     async login(username: string, password: string): Promise<number>{
```

```
46          const user: User = this.users.find((user: User) => user.username
               === username);
47          if(!user || user.password !== password){
48              throw new UnauthorizedException;
49          }
50          return user.id;
51      }
52  }
```

Kod 31: Servis korisnika

src/users/controller.model.ts

```
1   import { Controller, Post, Body, Get, Param, ParseIntPipe, Patch, Delete,
       HttpCode } from '@nestjs/common';
2   import { UsersService } from './users.service';
3   import { User } from './user.model';
4
5   @Controller('users')
6   export class UsersController {
7       constructor(private usersService: UsersService){}
8
9       @Post()
10      async create(@Body('username') username: string, @Body('password')
           password: string): Promise<User>{
11          return this.usersService.create(username, password);
12      }
13
14      @Get()
15      async get(): Promise<User[]>{
16          return this.usersService.get();
17      }
18
19      @Get(':id')
20      async getById(@Param('id', ParseIntPipe) id: number): Promise<User>{
21          return this.usersService.getById(id);
22      }
23
24      @Patch(':id')
25      async update(@Param('id', ParseIntPipe) id: number, @Body() body: {
           username?: string, password?: string}): Promise<User>{
26          return this.usersService.update(id, body.username, body.password);
27      }
28
29      @Delete(':id')
30      @HttpCode(204)
31      async delete(@Param('id', ParseIntPipe) id: number): Promise<void>{
32          await this.usersService.delete(id);
33      }
34
35      @Post('login')
36      async login(@Body('username') username: string, @Body('password')
           password: string): Promise<number>{
37          return this.login(username, password);
38      }
39  }
```

Kod 32: Kontroler korisnika

U Kodu 32 vidimo da za prijavljivanje korisnika također koristimo HTTP POST metodu. Koristimo ju iz sigurnosnih razloga, jer bi koristeći HTTP GET lozinka bila vidljiva u URL-u zahtjeva.

Trenutno se u svakom odgovoru koji vraća korisnika, vraća i njegova lozinka iako to nije preporučeno čak niti kada je ona šifrirana. Kako bismo lozinku uklonili iz odgovora, koristit ćemo presretač. Za odgovor u kojem vraćamo polje korisnika, napravit ćemo poseban presretač. Kreirajmo datoteku `src/users/remove-password-from-user.interceptor.ts` i u nju dodajmo Kod 33.

`src/users/remove-password-from-user.interceptor.ts`

```
1  import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from
       '@nestjs/common';
2  import { Observable } from 'rxjs';
3  import { map } from 'rxjs/operators';
4  import { User } from './user.model';
5
6  @Injectable()
7  export class RemovePasswordFromUserInterceptor implements NestInterceptor
       {
8    intercept(context: ExecutionContext, next: CallHandler): Observable<any>
         {
9      return next
10       .handle()
11       .pipe(map((user: User) => ({id: user.id, username: user.username,
           role: user.role})));
12   }
13 }
```

Kod 33: Presretač za brisanje lozinke iz odgovora jednog korisnika

Pomoću funkcije `pipe` ulančavamo naš odgovor. Koristimo operator `map` iz `rxjs` biblioteke koji nam služi kako bismo preslikali korisnika u novi objekt koji neće sadržavati njegovu lozinku.

Na sličan način možemo napraviti presretač za polje korisnika. Kreiramo datoteku `src/users/remove-password-from-users.interceptor.ts` i u nju dodamo Kod 34. Nakon što smo to napravili možemo prilagoditi kontroler korisnika kako je prikazano u Kodu 35.

`src/users/remove-password-from-users.interceptor.ts`

```
1  import { Injectable, NestInterceptor, ExecutionContext, CallHandler } from
       '@nestjs/common';
2  import { Observable } from 'rxjs';
3  import { map } from 'rxjs/operators';
4  import { User } from './user.model';
5
6  @Injectable()
7  export class RemovePasswordFromUsersInterceptor implements NestInterceptor
       {
8    intercept(context: ExecutionContext, next: CallHandler): Observable<any>
         {
9      return next
10       .handle()
11       .pipe(map((users: User[]) => users.map((user: User) => ({id: user.id
           , username: user.username, role: user.role}))));
12   }
13 }
```

Kod 34: Presretač za brisanje lozinke iz odgovora polja korisnika

src/users/users.controller.ts

```
1  import { Controller, Post, Body, Get, Param, ParseIntPipe, Patch, Delete,
       HttpCode, UseInterceptors } from '@nestjs/common';
2  import { UsersService } from './users.service';
3  import { User } from './user.model';
4  import { RemovePasswordFromUserInterceptor } from './remove-password-from-
       user.interceptor';
5  import { RemovePasswordFromUsersInterceptor } from './remove-password-from
       -users.interceptor';
6
7  @Controller('users')
8  export class UsersController {
9      constructor(private usersService: UsersService){}
10
11     @Post()
12     @UseInterceptors(RemovePasswordFromUserInterceptor)
13     async create(@Body('username') username: string, @Body('password')
           password: string): Promise<User>{
14         return this.usersService.create(username, password);
15     }
16
17     @Get()
18     @UseInterceptors(RemovePasswordFromUsersInterceptor)
19     async get(): Promise<User[]>{
20         return this.usersService.get();
21     }
22
23     @Get(':id')
24     @UseInterceptors(RemovePasswordFromUserInterceptor)
25     async getById(@Param('id', ParseIntPipe) id: number): Promise<User>{
26         return this.usersService.getById(id);
27     }
28
29     @Patch(':id')
30     @UseInterceptors(RemovePasswordFromUserInterceptor)
31     async update(@Param('id', ParseIntPipe) id: number, @Body() body: {
           username?: string, password?: string}): Promise<User>{
32         return this.usersService.update(id, body.username, body.password);
33     }
34
35     @Delete(':id')
36     @HttpCode(204)
37     async delete(@Param('id', ParseIntPipe) id: number): Promise<void>{
38         await this.usersService.delete(id);
39     }
40
41     @Post('login')
42     async login(@Body('username') username: string, @Body('password')
           password: string): Promise<number>{
43         return this.usersService.login(username, password);
44     }
45 }
```

Kod 35: Prilagođeni kontroler korisnika

Presretač ćemo još iskoristiti kako bismo presreli zahtjeve klijenta prilikom manipulacije bilješkama. U ovom će slučaju klijent morati priložiti identifikator koji je dodijeljen prilikom

prijave. On će uz prefiks "Bearer " biti vrijednost polja "Authorization". Pomoću njega ćemo odrediti kojem ćemo korisniku dodijeliti novokreiranu bilješku kako bismo vratili samo one bilješke koje pripadaju tom korisniku. Presretač će taj identifikator izvući iz zaglavlja, provjeriti je li valjan te ga, ukoliko je, priložiti tijelu zahtjeva.

Kako još nismo uključili korisnike u bilješke, prvo ćemo to napraviti. Prilagodimo model bilješki kako je prikazano u Kodu 36. Zatim ćemo implementirati opisani presretač tako što ćemo kreirati datoteku `src/notes/auth.interceptor.ts` i u nju dodati Kod 37.

`src/notes/note.model.ts`

```
export class Note{
    id: number;
    content: string;
    userId: number;
}
```

Kod 36: Prilagođeni model bilješki

`src/notes/auth.interceptor.ts`

```
import { Injectable, NestInterceptor, ExecutionContext, CallHandler,
    UnauthorizedException } from '@nestjs/common';
import { Observable } from 'rxjs';
import { Request } from 'express';
import { UsersService } from '../users/users.service';
import { User } from '../users/user.model';

@Injectable()
export class AuthInterceptor implements NestInterceptor {
    constructor(private usersService: UsersService){}
    async intercept(context: ExecutionContext, next: CallHandler): Promise
        <Observable<any>> {
         const request: Request = context.switchToHttp().getRequest();
         let user: User;
         try{
             const id: number = Number(request.headers.authorization.split(
                 ' ')[1]);
             user = await this.usersService.getById(id);
             request.body.user = user;
         }catch(error){
             throw new UnauthorizedException;
         }
         return next.handle()
    }
}
```

Kod 37: Presretač za autorizaciju

Kako je rečeno u potpoglavlju 6.7, koristeći `ExecutionContext` dohvaćamo zahtjev unutar čijeg polja "Authorization" se nalazi identifikator korisnika. Ukoliko tog identifikatora nema, ili ga ne uspijemo povezati s korisnikom, vratit ćemo grešku i obavijestiti klijenta da nije autoriziran. Ukoliko korisnik postoji, priložit ćemo ga u tijelo zahtjeva.

Za funkcioniranje presretača nam je potreban servis korisnika pa ga moramo izvesti iz modula korisnika te taj modul uvesti u modul bilješki, što je prikazano u Kodovima 38 i 39.

src/users/users.module.ts

```
import { Module } from '@nestjs/common';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';

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

Kod 38: Prilagođeni modul korisnika

src/notes/notes.module.ts

```
import { Module } from '@nestjs/common';
import { NotesController } from './notes.controller';
import { NotesService } from './notes.service';
import { UsersModule } from '../users/users.module';

@Module({
    providers: [NotesService],
    imports: [UsersModule],
    exports: [],
    controllers: [NotesController]
})
export class NotesModule{}
```

Kod 39: Prilagođeni modul bilješki

Sada moramo prilagoditi servis bilješki kako bismo mogli dohvatiti bilješke koje pripadaju određenom korisniku. Prilagođeni servis prikazan je u Kodu 40.

src/notes/notes.service.ts

```
import { Injectable, NotFoundException } from '@nestjs/common';
import { Note } from './note.model';

@Injectable()
export class NotesService {
    private notes: Note[] = [];
    private currentId: number = 0;

    async create(userId: number, content: string): Promise<Note>{
        const note: Note = new Note();
        note.id = this.currentId++;
        note.content = content;
        note.userId = userId;
        this.notes.push(note);
        return note;
    }

    async get(userId: number, term?: string): Promise<Note[]>{
        if(term){
            return this.notes.filter((note: Note) => note.content.includes
                (term) && note.userId === userId);
        }
        return this.notes.filter((note: Note) => note.userId === userId);
```

```
23      }
24
25      async getById(userId: number, id: number): Promise<Note>{
26          const note: Note = this.notes.find((note: Note) => note.id === id
                && note.userId === userId);
27          if(!note){
28              throw new NotFoundException;
29          }
30          return note;
31      }
32
33      async update(userId: number, id: number, content: string): Promise<
            Note>{
34          const note: Note = await this.getById(userId, id);
35          note.content = content;
36          return note;
37      }
38
39      async delete(userId: number, id: number): Promise<void>{
40          this.notes = this.notes.filter((note: Note) => !(note.id === id &&
                note.userId === userId));
41      }
42  }
```

Kod 40: Prilagođeni servis bilješki

Još je preostalo prilagoditi kontroler bilješki. Budući da će svaka metoda unutar kontrolera koristiti naš presretač, cijeli kontroler ćemo dekorirati `@UseInterceptors()` dekoratorom s odgovarajućim presretačem kao parametrom. Tada će nam u tijelu zahtjeva biti dostupan prijavljeni korisnik pa ga u svakoj metodi možemo izvući pomoću `@Body()` dekoratora. Prilagođeni kontroler prikazan je u Kodu 41.

src/notes/notes.controller.ts

```
1   import { Controller, Post, Body, Get, Param, Query, Patch, Delete,
        HttpCode, ParseIntPipe, UseInterceptors } from '@nestjs/common';
2   import { Note } from './note.model';
3   import { NotesService } from './notes.service';
4   import { AuthInterceptor } from './auth.interceptor';
5   import { User } from '../users/user.model';
6
7   @Controller('notes')
8   @UseInterceptors(AuthInterceptor)
9   export class NotesController {
10      constructor(private notesService: NotesService){}
11
12      @Post()
13      async create(@Body('user') user: User, @Body('content') content:
            string): Promise<Note>{
14          return this.notesService.create(user.id, content);
15      }
16
17      @Get()
18      async get(@Body('user') user: User, @Query('search') term?: string):
            Promise<Note[]>{
19          return this.notesService.get(user.id, term);
20      }
21
```

```
22      @Get(':id')
23      async getById(@Body('user') user: User, @Param('id', ParseIntPipe) id:
            number): Promise<Note>{
24          return this.notesService.getById(user.id, id);
25      }
26
27      @Patch(':id')
28      async update(@Body('user') user: User, @Param('id', ParseIntPipe) id:
            number, @Body('content') content: string): Promise<Note>{
29          return this.notesService.update(user.id, id, content);
30      }
31
32      @Delete(':id')
33      @HttpCode(204)
34      async delete(@Body('user') user: User, @Param('id', ParseIntPipe) id:
            number): Promise<void>{
35          await this.notesService.delete(user.id, id);
36      }
37  }
```

Kod 41: Prilagođeni kontroler bilješki

6.8 Čuvari

Čuvari (*Guards*) su gradivni blokovi iz skupine opskrbljivača, a ujedno su i posljednji gradivni blokovi koje ćemo obraditi. Čuvari se izvode prije svih presretača i cijevi pa onda, jasno, i prije metode koja obrađuje zahtjev. Odlučuju hoćemo li zahtjev ispuniti ili nećemo u ovisnosti o nekim uvjetima.

Čuvar je opskrbljivač koji implementira `CanActivate` sučelje, odnosno metodu `canActivate` koja prima jedan parametar `ExecutionContext` koji je jednak kao `ExecutionContext` parametar koji smo vidjeli kod presretača u potpoglavlju 6.7. Metoda vraća `boolean` vrijednost. Ako je vraćena vrijednost `true` zahtjev će se izvršiti, ako je `false` neće se izvršiti te će se vratiti greška sa statusom 403, "zabranjeno". Ako želimo vratiti neku drugu grešku, moramo ručno "baciti" iznimku koju želimo.

Kako bismo čuvara primijenili, slično kao kod presretača koristimo dekorator `@UseGuards()` iz *@nestjs/common* paketa.

Kako čuvari rade vidjet ćemo u sljedećem potpotpoglavlju 6.8.1.

6.8.1 Aplikacja

U poglavlju 6 je rečeno da će dohvaćati, brisati te uređivati korisnike moći samo posebni korisnici koji će imati ulogu administratora. Kako bismo to postigli, koristit ćemo čuvare.

Prvo ćemo dodati jednog korisnika unutar skladišta koji će uvijek biti tu i koji će imati ulogu administratora. Uredit ćemo `create` metodu unutar servisa tako da svim novim korisnicima dodijeli običnu ulogu te ćemo urediti `update` metodu kako bismo korisniku mogli promijenti ulogu. Prilagodimo servis korisnika kako je prikazano u Kodu 42. Nakon toga napravimo datoteku src/users/role.guard.ts te u nju dodajmo Kod 43. Implementacija

čuvara bit će vrlo slična implementaciji presretača.

src/users/users.service.ts

```
@Injectable()
export class UsersService {
    private users: User[] = [
        {
            id: 0,
            username: 'admin',
            password: 'password',
            role: 'admin'
        }
    ];
    private currentId: number = 1;

    async create(username: string, password: string): Promise<User>{
        const user: User = new User();
        user.id = this.currentId++;
        user.username = username;
        user.password = password;
        user.role = 'user';
        this.users.push(user);
        return user;
    }
    .
    .
    .
    async update(id: number, username?: string, password?: string, role?:
        'user' | 'admin'): Promise<User>{
        const user: User = await this.getById(id);
        if(username){
            user.username = username;
        }
        if(password){
            user.password = password;
        }
        if(role){
            user.role = role;
        }
        return user;
    }
    .
    .
    .
}
```

Kod 42: Prilagođeni servis korisnika

src/users/role.guard.ts

```
import { Injectable, CanActivate, ExecutionContext, UnauthorizedException
    } from '@nestjs/common';
import { User } from './user.model';
import { UsersService } from './users.service';
import { Request } from 'express';

@Injectable()
export class RoleGuard implements CanActivate {
    constructor(private usersService: UsersService){}
```

```
9       async canActivate(context: ExecutionContext):  Promise<boolean>{
10          const request: Request = context.switchToHttp().getRequest();
11          let user: User;
12          try{
13              const id: number = Number(request.headers.authorization.split(
                    ' ')[1]);
14              user = await this.usersService.getById(id);
15          }catch(error){
16              throw new UnauthorizedException;
17          }
18          if(user.role !== 'admin'){
19              throw new UnauthorizedException;
20          }
21          return true;
22      }
23  }
```

Kod 43: Čuvar prema ulogama

Preostalo je još prilagoditi kontroler korisnika kako je prikazano u Kodu 44 i aplikacija je završena.

src/users/users.controller.ts

```
1   import { Controller, Post, Body, Get, Param, ParseIntPipe, Patch, Delete,
        HttpCode, UseInterceptors, UseGuards } from '@nestjs/common';
2   import { UsersService } from './users.service';
3   import { User } from './user.model';
4   import { RemovePasswordFromUserInterceptor } from './remove-password-from-
        user.interceptor';
5   import { RemovePasswordFromUsersInterceptor } from './remove-password-from
        -users.interceptor';
6   import { RoleGuard } from './role.guard';
7
8   @Controller('users')
9   export class UsersController {
10      constructor(private usersService: UsersService){}
11
12      @Post()
13      @UseInterceptors(RemovePasswordFromUserInterceptor)
14      async create(@Body('username') username: string, @Body('password')
            password: string): Promise<User>{
15          return this.usersService.create(username, password);
16      }
17
18      @Get()
19      @UseGuards(RoleGuard)
20      @UseInterceptors(RemovePasswordFromUsersInterceptor)
21      async get(): Promise<User[]>{
22          return this.usersService.get();
23      }
24
25      @Get(':id')
26      @UseGuards(RoleGuard)
27      @UseInterceptors(RemovePasswordFromUserInterceptor)
28      async getById(@Param('id', ParseIntPipe) id: number): Promise<User>{
29          return this.usersService.getById(id);
30      }
31
```

```
32      @Patch(':id')
33      @UseGuards(RoleGuard)
34      @UseInterceptors(RemovePasswordFromUserInterceptor)
35      async update(@Param('id', ParseIntPipe) id: number, @Body() body: {
            username?: string, password?: string, role?: 'user' | 'admin'}):
            Promise<User>{
36          return this.usersService.update(id, body.username, body.password,
                body.role);
37      }
38
39      @Delete(':id')
40      @UseGuards(RoleGuard)
41      @HttpCode(204)
42      async delete(@Param('id', ParseIntPipe) id: number): Promise<void>{
43          await this.usersService.delete(id);
44      }
45
46      @Post('login')
47      async login(@Body('username') username: string, @Body('password')
            password: string): Promise<number>{
48          return this.usersService.login(username, password);
49      }
50  }
```

Kod 44: Posljednja verzija kontrolera korisnika

6.9 Pokretanje aplikacije

Kako bismo pokrenuli našu aplikaciju, možemo koristiti Nest CLI alat. Ako izvršimo naredbu `nest start`, kod naše aplikacije će se prevesti te će se aplikacija, ukoliko je prevođenje uspješno, pokrenuti. Ako želimo da se aplikacija automatski ponovno prevede i pokrene kada spremimo novu promjenu, možemo koristiti zastavicu `--watch`. Kako bismo našu aplikaciju izgradili za produkciju možemo koristiti naredbu `nest build` koja će napraviti paket sa svim datotekama koje su potrebne za njen rad.

Nakon što pokrenemo aplikaciju, ona će nove zahtjeve osluškivati na portu 3000 koji je moguće promijeniti po potrebi tako što ćemo unutar datoteke `src/main.ts` metodi `app.listen` kao parametar proslijediti port koji želimo.

Literatura

[1] A. Mardan, *Pro Express.js*, Apress, 2014.

[2] J. Bell, G. Magolan, D. Guijarro, A. de Peretti, P. Housley , *Nest.js: A Progressive Node.js Framework* , Bleeding Edge Press, 2018.

[3] K. Mysliwiec, *Demystifying Dependency Injection*, NG-DE Conference, 2019., `https://youtu.be/vYFhHVMetPg`

[4] M. Haverbeke, *Eloquent JavaScript*, No Starch Press, 2018.

[5] *NestJS dokumentacija*, `https://docs.nestjs.com`

[6] *NestJS izvorni kod*, `https://github.com/nestjs/nest`

[7] *Node.js*, `https://nodejs.org`

[8] *TypeScript*, `https://www.typescriptlang.org/`