Napredno softversko inženjerstvo

NestJs

Razvoj backend dela aplikacije

Studenti:
Nemanja Petrović 1499
Ivan Šušter 1548

Uvod

- Šta je NestJs i za šta se koristi
- Kako se instalira
- Kako se kreira projekat
- Struktura projekta
- CLI i generisanje servisa, kontrolera...
- Moduli
- Rad sa bazom
- Security

Šta je NestJs?

- Nest je Node.js framework
- Koristi se za kreiranje backend dela web aplikacija
- Koristi Typescript
- Modularan je
- Lak za učenje naročito za one koji znaju Angular

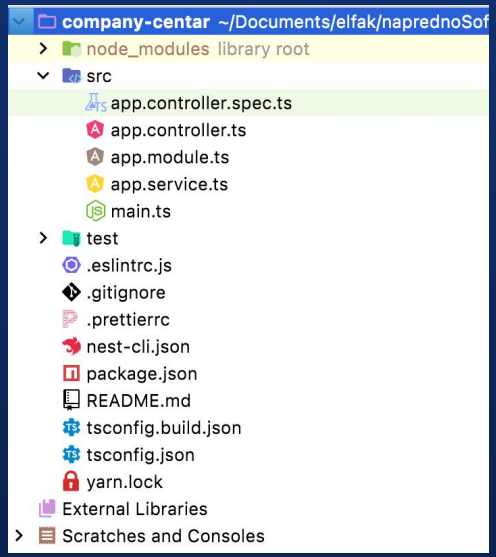
Instalacija

npm i -g @nestjs/cli

Kreiranje projekta

nest new company-centar

Struktura projekta



Najbitniji fajlovi/folderi

- `src folder` - Folder gde se nalazi kod aplikacije
- `package.json` - Fajl sa svim zavisnostima
- `src/main.ts` - Fajl gde se inicijalizuje Nest aplikacija

src/main.ts

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule, options: {cors: true});
  app.useGlobalPipes(new ValidationPipe());
  await app.listen( port: 3000);
}
bootstrap();
```

- `NestFactory` - core class
- `create()` - static method
- Startovanje HTTP listener-a

src/app.module.ts

```
4 usages  nemanja *
@Module( metadata: {
  imports: [],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

U ovom fajlu se definišu ostali fajlovi koje Nest koristi za kompajliranje aplikacije, odnosno fajlovi koji su neophodni za rad aplikacije

- `@Module` - dekorator koji Nest koristi da organizuje strukturu aplikacije u ovom slučaju potrebno je proslediti objekat
- `controllers` - skup kontrolera definisanih u modulu
- `providers` - skup servisa koje koriste kontroleri?

src/app.controller.ts

```
@Controller()
export class AppController {
  no usages  nemanja
  constructor(private readonly appService: AppService) {}

  1 usage  nemanja
  @Get()
  getHello(): string {
    return this.appService.getHello();
  }
}
```

Kontroler je odgovoran za rad sa zahtevima i odgovorima

- `routing` – kontroliše koji od kontrolera dobija zahtev
- Koriste se klase i dekoratori koji povezuju klase sa metadata kako bi Nest kreirao mapu rutiranja

src/app.service.ts

```
@Injectable()
export class AppService {
  1 usage  nemanja
  getHello(): string {
    return 'Hello World!';
  }
}
```

- Obično sadrže metode za upis, čitanje, brisanje i ažuriranje baze podataka odnosno biznis logiku za te pozive, dok se sam rad sa bazom radi u okviru repository klasa

Kreiranje modula

nest g module auth

Unutar ovog modula nalaziće se sve što se tiče korisnika i autorizacije

Šta se dešava kada kreiramo novi modul?

Kreiran je novi folder sa klasom koja predstavlja novi modul

```
4 usages  nemanja *
@Module( metadata: {
  imports: [AuthModule],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Kreirani modul je importovan u glavni AppModule

Entiteti

```
import {
  Column,
  CreateDateColumn,
  Entity,
  PrimaryGeneratedColumn,
  UpdateDateColumn,
} from 'typeorm';
import { RoleEnum } from './role.enum';

no usages  new *
@Entity()
export class User {
  @PrimaryGeneratedColumn( strategy: 'uuid')
  no usages  new *
  id: string;
  @Column( options: { unique: true })
  no usages  new *
  email: string;
  @Column()
  no usages  new *
  name: string;
  @Column()
  no usages  new *
  role: RoleEnum;
  @Column()
  no usages  new *
  password: string;
  @CreateDateColumn()
  no usages  new *
  createdAt: Date;
  @UpdateDateColumn()
  no usages  new *
  updatedAt: Date;
}
```

- Za rad sa bazom koristimo biblioteku **typeorm**
- Entiteti predstavlja objektnu reprezentaciju tabele iz relacione baze (u našem slučaju korismo PostgreSQL)
- Svaki entitet nosi dekorator **Entity**
- Ime entiteta je isto kao i ime tabele ali može i da se promeni ako se dekoratoru prosledi opcioni parametar
- Svaki entitet mora da ima ID koji predstavlja primarni ključ u bazi
- Entiteti podržavaju sve veze koje su podržane u relacionoj bazi

Šta se nam još treba da bi radili sa bazom?

```
import { EntityRepository, Repository } from 'typeorm';
import { User } from './user.entity';

no usages  new *
@EntityRepository(User)
export class UserRepository extends Repository<User> {}
```

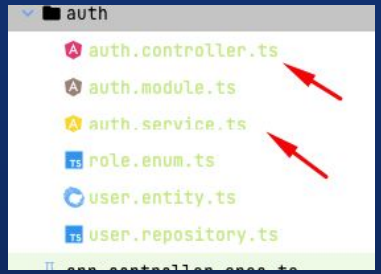
Kreirati repozitorijum

- Repozitorijum služi za komunikaciju sa bazom
- Svaki entitet mora da ima i svoj repozitorijum
- Repozitorijum nasleđuje TypeOrm repozitorijum i sadrži metode za CRUD operacije

```
2 usages  new *
@Module( metadata: {
  imports: [TypeOrmModule.forFeature( entities: [UserRepository])],
})
export class AuthModule {}
```

Importovati TypeORM u Auth modul

- Svaki modul koji sadrži entitete tj. Koji radi sa bazom mora da importuje TypeORM
- Prilikom importa navode se svi repozitorijumi koji se nalaze u tom modulu ili repozitorijumi iz eksternih modula ako ih koristimo




```
2 usages  new *
@Module( metadata: {
  imports: [TypeOrmModule.forFeature( entities: [UserRepository])],
  providers: [AuthService],
  controllers: [AuthController],
})
export class AuthModule {}
```

Kreiranje servisa i kontrolera

nest g service auth
nest g controller auth

Navedene komande kreiraju servis i kontroler klasu i automatski izmene modul fajl

Servis klasa koristi **@Injectable** dekorator što znači da ćemo moći da koristimo “dependency injection” i da umetnemo taj servis gde nam je potreban

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule,  options: {cors: true});
  app.useGlobalPipes(new ValidationPipe());
  await app.listen( port: 3000);
}
bootstrap();
```

```
export class CreateUserDto {
  @IsEmail()
  no usages  new *
  email: string;
  @IsString()
  no usages  new *
  name: string;
  @IsString()
  no usages  new *
  password: string;
}
```

DTO Klase

- DTO(Data Transfer Object) klase služe za prenos podataka.
- Dobra praksa kaže da se nikad ne koriste direktno klase koje predstavljaju entitete
- DTO klase se kreiraju za konkretne use case-ve
- U NestJs-u dto klase podržavaju validaciju koristeći dekoratore za validaciju (**IsString**, **IsEmail…**)
- Da bi aplikacija validirala podatke na osnovu tih validatora potrebno je da se korsiti globalni pipe - **ValidationPipe**

Kreiranje korisnika - repository

```
async createUser(createUserDto: CreateUserDto): Promise<User> {
  const { email, password, name } = createUserDto;
  const salt = await bcrypt.genSalt();
  const hashedPassword = await bcrypt.hash(password, salt);
  const user = this.create({
    email,
    name,
    role: RoleEnum.USER,
    password: hashedPassword
  });

  try {
    await this.save(user);
  } catch (error) {
    if (error.code === '23505') {
      throw new ConflictException( objectOrError: "Username not unique");
    } else {
      throw new InternalServerErrorException();
    }
  }
  return user;
}
```

- U repozitorijumu za User entitet kreiramo novu metodu kojoj prosleđujemo DTO kreiran za kreiranje korisnika
- Metoda je asinhrona jer je rad sa bazom asinhron
- Prvi korak je da lozinku koju je poslao korisnik heširamo, za to koristimo bcrypt
- U bazi čuvamo samo heširanu lozinku
- Nakon što popunimo sve podatke za entitet, pozivamo postojeću **save()** metodu
- Ako metoda ne vrata nikakav exception vraćamo nazad kreiranog korisnika
- Metoda može da baci grešku sa kodom 23505 što znači da korisnik sa tim korisničkim imenom već postoji, u našem slučaju gleda se email i tu grešku obrađujemo i poruku vraćamo nazad

Kreiranje korisnika - servis i kontroler

```
@Controller( prefix: 'auth')
export class AuthController {
  no usages  new *
  constructor(private  authService: AuthService) {
  }
  no usages  new *
  @Post()
  signUp(@Body() createUserDto: CreateUserDto): Promise<void> {
    return this.authService.create(createUserDto);
  }
```

```
@Injectable()
export class AuthService {
  no usages  new *
  constructor(
    no usages  new *
    @InjectRepository(UserRepository) private userRepository: UserRepository) {
  }
  1 usage  new *
  async create(createUserDto: CreateUserDto): Promise<void> {
    await this.userRepository.createUser(createUserDto);
  }
}
```

Kontroler

- Za kontroler koristimo dekorator **@Controller**
- Pošto servis ima dekorator **@Injectable** možemo da ga samo kroz konstruktor ubacimo bez da kreiramo objekat
- Za kreiranje korisnika imamo metodu koja ima dekorator **@Post** koji ozačava da je u pitanju POST Rest metoda
- Jedini posao kontrolera je da pozove servis

Servis

- Koristeći dekorator **@InjectRepository** u servis ubacujemo repozitorijum
- Imamo metodu create koja je asinhrona jer radi sa bazom koja radi asinhrono i čiji je jedini posao da pozove repozitorijum.
- Kad bi imali neku dodatnu biznis logiku ona bi se nalazila u servisu

Pokretanje projekta

yarn start

ALI
Pre toga moramo da podesimo bazu podataka

Šta nam je potrebno za rad sa bazom?

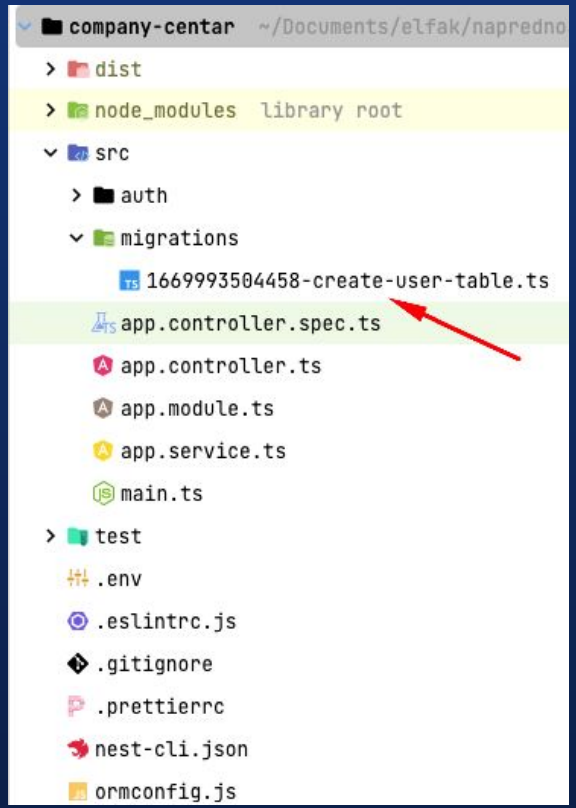
- Konfiguracija konekcije za bazu
- Konfiguracija migracija
- Kreiranje migracija
- Izvršavanje migracija
- Zašto migracije?
  Zato što je dobra praksa da se verzija baza (kreiranje tabela, izmena tabela...) radi uz pomoć nekih skripta

Konfiguracija konekcije i migracija

```
module.exports = {
  type: "postgres",
  host: process.env.DB_HOST,
  port: process.env.DB_PORT,
  username: process.env.DB_USERNAME,
  password: process.env.DB_PASSWORD,
  database: process.env.DB_DATABASE,
  synchronize: false,
  logging: false,
  entities: ["dist/**/*.entity{.ts,.js}"],
  migrations: ["dist/migrations/*.js"],
  migrationsRun: true,
  cli: {
    migrationsDir: "src/migrations",
    entitiesDir: "src/*.entity{.ts,.js}"
  }
}
```

- Kreiramo fajl ormconfig.js u root folderu projekta
- Tu podesimo sve parametre
- Parametri konekcije se uzimaju iz env varijabli
- U AppModulu importujemo TypeORM

```
@Module( metadata: {
  imports: [
    TypeOrmModule.forRootAsync( options: {
      useFactory: async () =>
        Object.assign(await getConnectionOptions(), source: {
          autoLoadEntities: true,
        }),
    }),
```



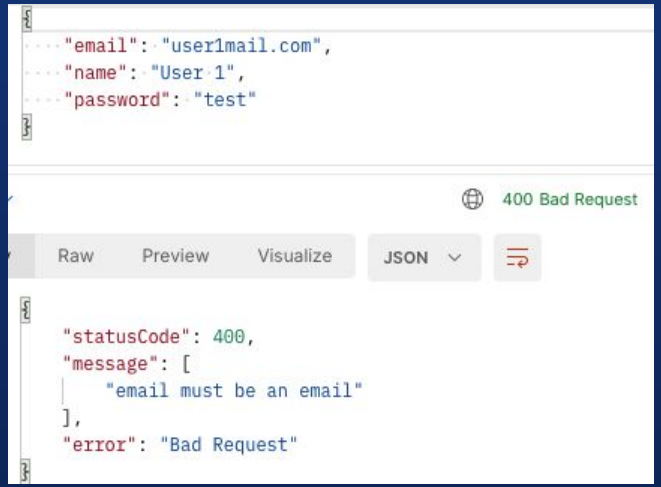
Kreiranje i izvršavanje migracija

- Komandom **npm run migration-generate -n ime_migracije** kreiramo migraciju
- TypeORM automatski nalazi sve entitete i gleda raziliku u odnosu na prethodnu migraciju i tako kreira novu migraciju
- Rezultat je fajl unutar foldera **migracije**

- Migracije izvršavamo komandom **Npm run migration-run**
- Tada TypeORM gleda tabelu **migrations** koja je automatski kreirana i izvršava sve migracije iz foldera **migrations** koje se ne nalaze u tabeli u samoj bazi

Testiranje kreiranja korisnika

- Kada smo kreirali i izvršili migracije možemo komandom **yarn start** da pokrenemo projekat
- Za testiranje endpointa koristimo **Postman**
- Slanje POST zahteva gde body izgleda kao i dto klasa za kreiranje korisnika, kreiramo korisnika



- Ako pošaljemo podatke koji nisu ispravni možemo da vidimo da dekoratori za validaciju rade i vraćaju nam greške

Generisanje dodatnog modula

- Cilj aplikacije je da se vodi evidencija o zaposlenima i timovima
- Kako bi maksimalno iskoristili modularnost NestJS-a kreiramo novi modul za timove (**nest g module team)**
- Novi modul će se koristiti za sve što ima veze sa timovima
- Sadržaće entitet za Tim, repozitorijum, servis, kontroler, DTO klase

Entit za Tim i veza sa User entitetom

```
@Entity()
export class Team {
  @PrimaryGeneratedColumn( strategy: 'uuid')
  no usages  new *
  id: string;
  @Column()
  no usages  new *
  name: string;
  @OneToMany( typeFunctionOrTarget: () => User,  inverseSide: user => user.team)
  no usages  new *
  users: User[]
}
```

- Entite Tim kreiramo na isti način kao što smo kreirali User entitet
- Kako bi podržali vezu jedan prema više - Jedan tim može da ima više korisnika potrebni je da u oba entiteta “podesimo” tu vezu
- Na strani Tim entiteta dodajemo niz korisnika i stavljamo dekorator **@OneToMany** jer jedan tim može da ima više korisnika
- Na strani User entiteta dodajemo polje tim, i stavljamo dekorator **@ManyToOne** jer više korisnika može da pripada jednom timu
- U oba dekoratora stavljamo i naziv polja sa druge strane veze

```
@ManyToOne( typeFunctionOrTarget: () => Team,  inverseSide: team => team.users,  options: {eager: true})
1 usage  new *
team: Team;
```

Repozitorijum i migracija za tim

```
@EntityRepository(Team)
export class TeamRepository extends Repository<Team> {

}
```

```
@Module( metadata: {
  imports: [TypeOrmModule.forFeature( entities: [TeamRepository])]
  controllers: [TeamController],
  providers: [TeamService]
})
export class TeamModule {}
```

- Kreiramo repozitorijum za entite Tim
- Importujemo TypeORM modul u modul za Tim I odmah definišemo i repozitorijum za Tim na istom mestu

- Nakon kreiranja repozitorijuma, potrebno je da kreiramo migraciju
- **npm run migration-generate -n create-team-table**
- Nakon što je migracija kreirana, potrebno je da je izvršimo kako bi se u bazi kreirala nova table i modifikovala tabela za korisnika kako bi imala strani ključ prema tabeli za tim
- **npm run migration-run**

Operacije nad timom

Potrebno je da nad timom obezbedimo sledeće operacije

- Kreiranje tima
- Izmena tima
- Brisanje tima
- Čitanje svih timova
- Čitanje jednog tima
- Dodavanje i uklanjanje korisnika iz tima

Kreiranje tima

Za kreiranje tima imamo par koraka:

- Prvo kreiramo dto klasu, istu klasu ćemo koristiti i za izmenu tima
- U repozitorijumu kreiramo metodu za kreiranje tima
- U servisu kreiramo metodu za kreiranje tima koja poziva repozitorijum
- Ostaje još da u kontroleru iskoristimo servisnu metodu ali to ćemo na kraju, kada završimo ostale funkcionalnosti na nivou repozitorijuma i servisa

```
no usages  new *
export class CreateUpdateTeamDto {
  @IsString()
  no usages  new *
  name: string;
}
```

```
@EntityRepository(Team)
export class TeamRepository extends Repository<Team> {
  no usages  new *
  async createTeam(dto: CreateUpdateTeamDto): Promise<Team> {
    const { name } = dto;
    const team = this.create({
      name
    });

    try {
      await this.save(team);
    } catch (error) {
      throw new InternalServerErrorException();
    }
    return team;
  }
}
```

```
@Injectable()
export class TeamService {
  no usages  new *
  constructor(
    no usages  new *
    @InjectRepository(TeamRepository) private teamRepository: TeamRepository) {
  }
  no usages  new *
  async create(dto: CreateUpdateTeamDto): Promise<Team> {
    return await this.teamRepository.createTeam(dto);
  }
}
```

Izmena tima

Izmena tima se sastoji od sledećih koraka

- Prvo po jedinstvenom id-u pronađemo tim, ako ne postoji vratimo odgovarajuću grešku
- Ako tim postoji, izmenimo mu podatke, u našem slučaju samo je ime moguće promeniti
- Sačuvamo tim koristeći već ugrađenu metodu iz repozitorijuma
- U ovom koraku nemamo nikakav dodatni kod unutar repozitorijuma, korisnimo samo stvari koje su već ugrađene u TypeORM, isti slučaj će biti i za čitanje i brisanje tima

```
async update(id: string, dto: CreateUpdateTeamDto): Promise<Team> {
  const team = await this.teamRepository.findOne(id);
  if (team) {
    team.name = dto.name;
    return await this.teamRepository.save(team);
  } else {
    throw new NotFoundException( objectOrError: "Team not found")
  }
}
```

Čitanje timova

Imamo dve vrste čitanja, čitanje jednog tima i svih timova

- Kod čitanja jednog tima imamo istu proceduru kao kod izmene, po id-u potražimo tim u bazi i ako postoji vratimo ga, ako ne postoji vratimo grešku
- Čitanje svih timova se radi metodom **find()** bez slanja parametara.
  Ako postoje timovi oni će biti vraćeni, ako ne postoje vratiće se prazna lista

```
async getOneTeam(id: string): Promise<Team> {
  const team = await this.teamRepository.findOne(id);
  if (team) {
    return team;
  } else {
    throw new NotFoundException( objectOrError: "Team not found")
  }
}
no usages  new *
async getAllTeams(): Promise<Team[]> {
  return await this.teamRepository.find();
}
```

Brisanje tima

Brisanje tima može da se obavi na dva načina.

- Prvi način je da se po id-u potraži tim, zatim ako postoji da se pozove **remove()** metoda a ako ne postoji da se vrati greška
- Drugi je da se odmah pozove **delete()** metoda i da joj se prosledi id

U prvom slučaju imamo dva poziva prema bazi dok u drugom ne znamo da li tim postoji ili ne a imamo jedan poziv prema bazi.
Ali, ako se pogelda rezultat **delete()** metode, možemo da zaključimo da li je neki red u bazi bio promenjen, ako jeste znači da tim postoji i da je obrisan, ako nije izmenjen ni jedan red, tim ne postoji i možemo da vratimo grešku.

Zaključak je da je optimalnije koristiti **delete()** metodu

```
async delete(id: string): Promise<void> {
  const team = await this.teamRepository.findOne(id);
  if (team) {
    await this.teamRepository.remove(team);
  } else {
    throw new NotFoundException( objectOrError: "Team not found")
  }
}
```

```
async delete(id: string): Promise<void> {
  const result = await this.teamRepository.delete(id);
  if (result.affected === 0) {
    throw new NotFoundException( objectOrError: "Team not found")
  }
}
```

Dodavanje korisnika u tim

Pre nego što možemo da radimo sa korisnicima u okviru modula za tim potrebno je da u taj modul dodamo korisnički repozitorijum.

Radi jednostavnosti ovog demo-a, ako je korisnik već u nekom timu, taj tim će biti zamenjen novim timom. Ovde naravno može da se vrati i greška da nije dozvoljeno da se korisnik doda u tim jer je već član drugog tima.

Za pretragu tima po id-u koristimo metodu koju smo napravili u prethonim koracima, dok smo za pretragu korisnika napravili istu takvu metodu

```
2 usages  nemanja *
@Module( metadata: {
  imports: [TypeOrmModule.forFeature( entities: [TeamRepository, UserRepository])],
  controllers: [TeamController],
  providers: [TeamService]
})
export class TeamModule {}
```

```
async addUserToTeam(teamId: string, userId: string): Promise<void> {
  const team = await this.getOneTeam(teamId);
  const user = await this.getUser(userId);
  user.team = team;
  await this.userRepository.save(user);
}
```

Brisanje korisnika iz tima

Kod brisanja korisnika iz tima koristimo iste one metode za pretragu tima i korisnika koje već posedujemo.

Ovde imamo dodatne validacije, ako korisnik nema tim, onda nemamo odakle da ga sklonimo pa vraćamo odgovarajuću grešku.
Takođe, ako korisnik ima tim ali taj tim nije onaj tim sa kojim trenutno manipulišemo, javljamo da ne možemo da ga sklonimo iz tog tima jer nije član tog tima

```
async removeUserFromTeam(teamId: string, userId: string): Promise<void> {
  const team = await this.getOneTeam(teamId);
  const user = await this.getUser(userId);
  if (user.team === null) {
    throw new BadRequestException( objectOrError: "Given user is not a member of any team");
  }
  if (user.team.id === team.id) {
    user.team = null;
  } else {
    throw new BadRequestException( objectOrError: "Given user is not a member of given team");
  }
  await this.userRepository.save(user);
}
```

TeamController

Do sada smo napravili svu biznis logiku. Kako bi to zapravo moglo da se koristi potrebno je da se izlože REST endpoint-i.
U kontroleru imamo sledeće metode

- **POST** metodu koja se koristi za kreiranje tima. Ona prihvata body i radi isto kao kod kreiranja korisnika što je već objašnjeno
- **PUT** metodu za izmenu tima. Ovde se koristi **PUT** a ne **POST** jer je praksa da se za izmene koristi **PUT** a za kreiranje **POST**.
  Metoda pored body parametra sadrži i jedan path parametar koji predstavlja id tima koji želimo da izmenimo. Path parametar se u NestJS-u označava sa :ime
  U konkretnom slučaju izgled url-a za ovu **PUT** metodu bi bio na primer **/teams/1** gde je **1** id tima tj path promenljiva

```
@Controller( prefix: 'team')
export class TeamController {
  no usages  new *
  constructor(private teamService: TeamService) {
  }
  no usages  new *
  @Post()
  create(@Body() dto: CreateUpdateTeamDto): Promise<Team> {
    return this.teamService.create(dto);
  }
  no usages  new *
  @Put( path: "/:id")
  update(@Param( property: 'id') id: string, @Body() dto: CreateUpdateTeamDto): Promise<Team> {
    return this.teamService.update(id, dto);
  }
```

TeamController

- **GET** metodu za čitanje jednog tima po id-u. Ovde smo takođe iskoristili path promenljivu za id
- **GET** metodu za čitanje svih timova
- **DELETE** metodu za brisanje tima po id-u. I ovde je takođe iskorišćena path promenljiva
- **PUT** metodu za dodavanje korisnika u tim. Ovde smo iskoristili dve path promenljive, jednu za id tima a drugu za id korisnika. Primer url-a bi bio: **/team/1/user/2** Gde je 1 id tima a 2 id korisnika
- **DELETE** metodu za uklanjanje korisnika iz tima. Kao i metoda iznad imamo dve path varijable i url bi isto izgledao

```
@Get( path: "/:id")
getOne(@Param( property: 'id') id: string): Promise<Team> {
  return this.teamService.getOneTeam(id);
}
no usages  new *
@Get()
getAll(): Promise<Team[]> {
  return this.teamService.getAllTeams();
}
no usages  new *
@Get( path: "/:id")
delete(@Param( property: 'id') id: string): Promise<void> {
  return this.teamService.delete(id);
}
no usages  new *
@Put( path: "/:teamId/user/:userId")
adUser(@Param( property: 'teamId') teamId: string,@Param( property: 'userId') userId: string ): Promise<void> {
  return this.teamService.addUserToTeam(teamId, userId);
}
no usages  new *
@Delete( path: "/:teamId/user/:userId")
removeUser(@Param( property: 'teamId') teamId: string,@Param( property: 'userId') userId: string ): Promise<void> {
  return this.teamService.removeUserFromTeam(teamId, userId);
}
```

Presek stanja

01

Kreiranje projekta

02

Rad sa bazom

03

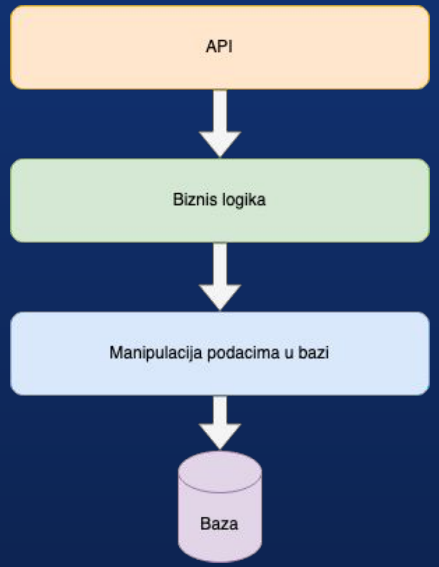
Rad sa modulima

04

Kreiranje servisa

05

Kreiranje kontrolera



Slojevita arhitektura

Problemi

Aplikacija nam nije zaštićena, tj:

- Svako može da kreira korisnika
- Svako može da kreira tim
- Svako može da vidi sve timove i sve članove
- Svako može da doda ili ukloni korisnike iz nekog tima

Cilj je da imamo dve korisničke role, USER i ADMIN.

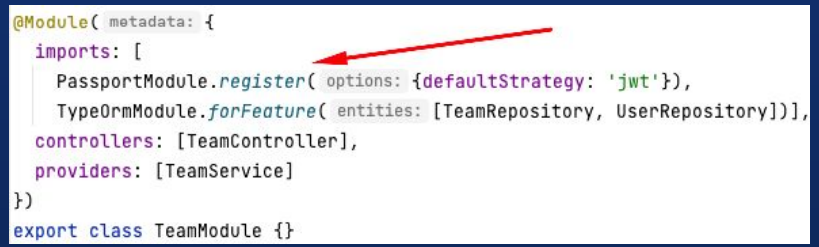
- Korisnik sa USER rolom bi trebalo da može samo da se prijavi i da pročita sve timove kao i da pročita neki tim po id-u
- Korisnik sa ADMIN rolom bi trebalo da može da kreira druge korisnike, kreira timove, dodaje i uklanja korisnike iz timova kao i da radi sve što može i korisnik sa USER rolom

Instalacija biblioteka za security

yarn add passport
yarn add passport-jwt
yarn add @nestjs/jwt
yarn add @nestjs/passport

Podešavanje modula za korišćenje security biblioteka

```
@Module( metadata: {
  imports: [
    PassportModule.register( options: {defaultStrategy: 'jwt'}),
    JwtModule.registerAsync(
      options: {
        useFactory: async () => ({
          secret: process.env.JWT_SECRET,
          signOptions: {
            expiresIn: 3600
          }
        }),
      },
    ),
    TypeOrmModule.forFeature( entities: [UserRepository])],
  providers: [AuthService],
  controllers: [AuthController],
})
export class AuthModule {}
```



```
@Module( metadata: {
  imports: [
    PassportModule.register( options: {defaultStrategy: 'jwt'}),
    TypeOrmModule.forFeature( entities: [TeamRepository, UserRepository])],
  controllers: [TeamController],
  providers: [TeamService]
})
export class TeamModule {}
```

Auth modul

- Potrebno je da registrujemo PassportModule i da mu kažemo koju strategiju koristimo, u našem slučaju to je JWT (JSON Web Token)
- Potrebno je takođe registrovati i JwtModule. Ovde podešamo stvari koje se tiču izdavanja i validacije tokena, tačnije podešavamo tajni ključ koji se koristi za potpisivanje i podešavamo trajanje tokena u sekundama

Tim modul

- I ovde registrujemo PassportModule i podešavamo ga za JWT strategiju.
  Ovaj modul je potrebno registrovati u svim modulima gde ćemo da imamo zaštićene endpointe
- U Tim modulu nema potrebe da registrujemo JwtModule zato što se Tim modul ne bavi izdavanjem i validacijom tokena

DTO klase za prijavu korisnika

```
export class LoginRequestDto {
  @IsEmail()
  no usages  new *
  email: string;
  @IsString()
  no usages  new *
  password: string;
}
```

LoginRequestDto

- Jednostavna klasa sa validacijom. Sadrži dva parametra koji su potrebni za prijavu na sistem, email i lozinku

```
export class LoginResponseDto {
  1 usage  new *
  token: string;
  1 usage  new *
  role: string;
  1 usage  new *
  name: string;
```

LoginResponseDto

- Ako su podaci poslati prilikom prijave ispravni, korisnik nazad dobija objekat koji sadrži podatke o tokenu, roli koju ima i imenu korisnika.
  Rola i ime su tu čisto radi primera, token je najbitniji jer taj token se šalje u svim pozivima koje korisnik poziva nakon prijave

```
export interface JwtPayload {
  2 usages  new *
  email: string;
}
```

JwtPayload

- Jednostavni objekat koji će biti potpisan u token koji se kreira. Ovde možemo da ubacimo šta god želimo od podataka ali sada za demonstraciju je dovoljan email korisnika

Prijava korisnika - izdavanje tokena

- Kreiramo **POST** metodu u klasi **AuthController**
  Metoda prihvata body u obliku dto-a koji smo kreirali u prethodnom koraku i poziva servis koji radi prijavljivanje korisnika
- Kreiramo **singIn** metodu u klasi **AuthService**
  A zatim radimo prijavu korisnika koja se sastoji iz sledećih koraka
  - Tražimo korisnika u bazi po emailu, za to koristimo već poznatu metodu **findOne** ali ovog puta joj šaljemo objekat umesto paremtra id, u objektu unosimo parametar po kom pretražujemo.
  - Zatim, ako korisnik postoji potrebno je da validiramo lozinku. Kada smo kreirali korisnika njegovu lozinku smo heširali koristeći **bcrypt** pa sada ne možemo samo da proverimo da li je ista kao i poslata lozinka već moramo da koristimo compare metodu iz bcrypt biblioteke koja za nas upoređuje heširanu vrednost i poslatu lozinku.
  - Na kraju ako je sve uspešno, potpisujemo i kreiramo token koristeći **jwtService**.
  - Ako nije prošla validacija lozinke ili korisnik ne postoji, vraćamo grešku

```
@Post( path: "/signin")
signIn(@Body() credentialsDto: LoginRequestDto): Promise<LoginResponseDto> {
  return this.authService.signIn(credentialsDto);
}
```

```
async signIn(credentialsDto: LoginRequestDto): Promise<LoginResponseDto> {
  const { email, password } = credentialsDto;
  const user = await this.userRepository.findOne( conditions: { email });

  if (user && (await bcrypt.compare(password, user.password))) {
    const payload: JwtPayload = {email};
    const accessToken = this.jwtService.sign(payload);
    await this.userRepository.save(user);
    return new LoginResponseDto(
      accessToken,
      user.role,
      user.name
    )
  } else {
    throw new UnauthorizedException( objectOrError: "Please check your login credentials");
  }
}
```

Validacija tokena

Kako bi prilikom poziva nekog endpointa mogli da validiramo token koji je poslat, potrebno je da uradimo par stvari:

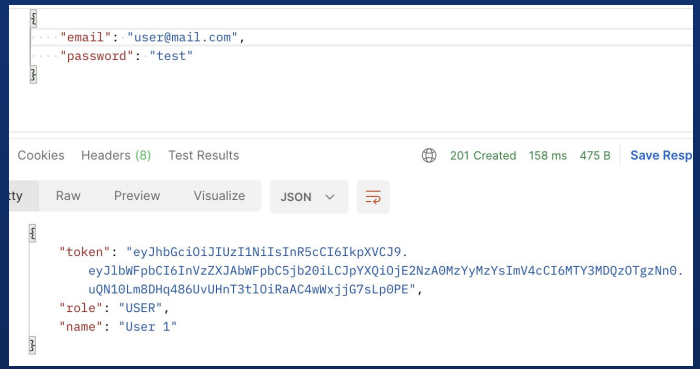
- Kreiramo klasu koja nasleđuje strategiju iz **Passport** biblioteke čiji je jedini posao da validira token.
- Izvlačenje i validacija tokena je skroz automatizovana i taj deo kompletno radimo u samom konstruktoru tako što navedemo ključ kojim je potpisan token prilikom kreiranja i način na koji da se pročita iz HTTP zahteva
- U našem slučaju kažemo da se token čita iz **headera** kao **BearerToken**, tj aplikacija očekuje da se token nalazi u headeru HTTP zahteva u formatu:
  Authorization: Bearer some_token
- Ako je token validiran da postoji i da je validan tj da je potpisan našim ključem, pravimo dodatnu validaciju u metodi validate, koja iz tokena uzima email korisnika i na osnovu emaila traži tok koisnika kod nas u bazi, ako korisnik postoji taj korisnik se vraća kao rezultat metode a ako ne postoji vraća se greška

```
@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  no usages  new *
  constructor(
    1 usage  new *
    @InjectRepository(UserRepository) private userRepository: UserRepository,
    1 usage  new *
    private configService: ConfigService
  ) {
    super(
      {
        secretOrKey: configService.get('JWT_SECRET'),
        jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      }
    );
  }
  no usages  new *
  async validate(payload: JwtPayload): Promise<User> {
    const { email } = payload;
    const user: User = await this.userRepository.findOne( conditions: { email });

    if (!user) {
      throw new UnauthorizedException();
    }

    return user;
  }
}
```

Testiranje prijave korisnika



- Slanjem **POST** zahteva u Postman-u sa podacima korisnika kog smo kreirali kada smo pravili kreiranje korisnika, dobijamo odgovor koji sadrži token
- Kada bi taj token dekodirali koristeći neki besplatni alat videli bi da on sadrži ono što smo upisali u njega tačnije email korisnika kao i podatke o tome kada je izdat i kada ističe

Obezbeđivanje endpointa

```
@Post()
@UseGuards(AuthGuard())
 createUser(@Body() createUserDto: CreateUserDto): Promise<void> {
  return this.authService.create(createUserDto);
}
```

```
@Controller( prefix: 'team')
@UseGuards(AuthGuard())
export class TeamController {
```

Endpointe obezbeđujemo kristeći dekorator iz NestJS-a @UseGuards() kome prosleđujemo neki Guard. Postoji već ugrađeni Guard u NestJs a to je AuthGuard, on proverava da li je u zahtevu prisutan validan token

Endpointe možemo obezbediti na dva načina:

- Možemo obezbediti konkretni endpoint, kao što smo uradili za kreiranje korisnika
- A možemo i staviti dekotrator na nivou celog kontrolera čime smo obezbedili sve endpointe u tom kontroleru

Obezbeđivanje na osnovu role

U prethodnom koraku smo obezbedili endpointe ali problem je to što i dalje svaki korisnik može sve da radi. Potrebno je da neke stvari zaštitimo na role korisnika, da bi to postigli raidmo sledeće

- Kreiramo naš Guard tako što implementiramo CanActivate interfejs iz NestJs-a i pravimo canActivate metodu koja radi sledeće:
  - Iz HTTP zahteva čita korisnika, on je tu jer smo kada smo validirali token vraćanjem korisnika zapravo njega upisali u HTTP zahtev
  - Uzima rolu tog korisnika i upoređuje je sa poslatom rolom
- Zatim taj kreirani Guard stavljamo iznad onih metoda u kontroleru koje želimo da zaštitimo, desno je dat primer na metodi za kreiranje tima ali isto to postavljamo i na ostalim metodama gde je potrebno

```
const RoleGuard = (role: RoleEnum): Type<CanActivate> => {
  1 usage  new *
  class RoleGuardMixin implements CanActivate {
    no usages  new *
    canActivate(context: ExecutionContext) {
      const { user } = context.switchToHttp().getRequest();
      return user.role === role;
    }
  }
  return mixin(RoleGuardMixin);
}
no usages  new *
export default RoleGuard;
```

```
@Post()
@UseGuards(RoleGuard(RoleEnum.ADMIN))
create(@Body() dto: CreateUpdateTeamDto): Promise<Team> {
  return this.teamService.create(dto);
}
```

Šta smo sve uradili

- Nakon svega što smo prošli imamo potpuno funkcionalni backend deo jedne web aplikacije
- Videli smo kako u NestJS-u napraviti sve što je potrebno da bi se dobila funkcionalna aplikacija
- Prošli smo kroz neke dobre prakse vezane za rad sa bazom, za razdvajanje aplikacije u module itd
- Nakon svega ovoga možemo da se osvrnemo i pogledamo prednosti i mane ove tehnologije

Prednosti

- Korišćenje TypeScript-a i vrlo lako učenje za nekog ko zna Angular
- Moćan CLI
- Dosta ugrađenih stvari i dostupnih biblioteka
- Dosta brz razvoj aplikacija jer je dosta stvari već ugrađen
- Modularna organizacija projekta
- Ugrađen Dependency Injection

Mane

- Relativno nova tehnologija pa je ne koristi veliki broj ljudi, što može da oteža rešavanje nekih kompleksnijih problema
- Kompleksna struktura za nekog ko dolazi iz Js sveta i ko nije navikao na TypeScript i Angular

Hvala