Zaawansowany Przewodnik po NestJS

Kacper Renkel

August 11, 2024

Contents

1 Wprowadzenie

NestJS to framework do budowania aplikacji serwerowych w Node.js, który wspiera TypeScript i korzysta z dekoratorów oraz architektury opartej na modułach. Ułatwia tworzenie skalowalnych aplikacji dzieki wbudowanym mechanizmom, takim jak Dependency Injection i obsługa middleware.

2 Kontrolery

Kontrolery w NestJS sa odpowiedzialne za obsługe żadań HTTP i delegowanie zadań do serwisów. Każdy kontroler odpowiada za jedna grupe funkcjonalności aplikacji.

2.1 Przykład Kontrolera

```
import { Controller, Get, Post, Body, Param } from '@nestjs/
   common';
import { UsersService } from './users.service';
import { CreateUserDto } from './dto/create-user.dto';

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

  @Get(':id')
  async getUser(@Param('id') id: string) {
    return this.usersService.getUser(id);
  }

  @Post()
  async createUser(@Body() createUserDto: CreateUserDto) {
    return this.usersService.createUser(createUserDto);
  }
}
```

Listing 1: Przykład kontrolera w NestJS

3 Serwisy

Serwisy w NestJS sa klasami, które zawieraja logike biznesowa aplikacji i moga komunikować sie z baza danych lub innymi źródłami danych. Serwisy sa wstrzykiwane do kontrolerów i innych serwisów za pomoca Dependency Injection.

3.1 Przykład Serwisu

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

```
import { Repository } from 'typeorm';
import { User } from './user.entity';
import { CreateUserDto } from './dto/create-user.dto';

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

  async getUser(id: string): Promise<User> {
    return this.userRepository.findOneBy({ id });
  }

  async createUser(createUserDto: CreateUserDto): Promise<
     User> {
    const user = this.userRepository.create(createUserDto);
    return this.userRepository.save(user);
  }
}
```

Listing 2: Przykład serwisu w NestJS

4 Repozytoria

Repozytoria w NestJS zarzadzaja interakcja z baza danych i sa zwykle tworzone przy użyciu ORM, takiego jak TypeORM.

4.1 Przykład Repozytorium

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

@EntityRepository(User)
export class UserRepository extends Repository<User> {
  // Dodatkowe metody repozytorium mog  by  zdefiniowane
     tutaj
}
```

Listing 3: Przykład repozytorium w NestJS

5 JWT (JSON Web Token)

JWT jest używany w NestJS do autoryzacji i uwierzytelniania. Używa sie biblioteki '@nestjs/jwt' do generowania i weryfikowania tokenów.

5.1 Przykład Generowania i Weryfikowania Tokenu

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

@Injectable()
export class AuthService {
  constructor(private readonly jwtService: JwtService) {}

  async generateToken(payload: any) {
    return this.jwtService.sign(payload);
  }

  async verifyToken(token: string) {
    return this.jwtService.verify(token);
  }
}
```

Listing 4: Przykład użycia JWT w NestJS

6 Hashowanie Haseł

Do hashowania haseł w NestJS można użyć biblioteki ‘bcrypt‘.

6.1 Przykład Hashowania Haseł

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

export class AuthService {
  async hashPassword(password: string): Promise<string> {
    const salt = await bcrypt.genSalt();
    return bcrypt.hash(password, salt);
  }

  async comparePassword(password: string, hash: string):
     Promise<boolean> {
    return bcrypt.compare(password, hash);
  }
}
```

Listing 5: Przykład hashowania haseł przy użyciu bcrypt

7 Middleware

Middleware w NestJS to klasy implementujace interfejs ‘NestMiddleware‘, które moga modyfikować żadania i odpowiedzi.

7.1 Przykład Middleware

```
import { Injectable, NestMiddleware } from '@nestjs/common';
import { Request, Response, NextFunction } from 'express';

@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: NextFunction) {
    console.log(`Request made to: ${req.url}`);
    next();
  }
}
```

Listing 6: Przykład middleware w NestJS

8 Guardy

Guardy w NestJS służa do autoryzacji i decyduja, czy żadanie może przejść do kontrolera. Implementuja interfejs 'CanActivate'.

8.1 Przykład Guarcu

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

@Injectable()
export class AuthGuard implements CanActivate {
  canActivate(context: ExecutionContext): boolean {
    const request = context.switchToHttp().getRequest();
    const token = request.headers['authorization'];
    return token === 'valid-token';
  }
}
```

Listing 7: Przykład guardu w NestJS

9 Interceptory

Interceptory w NestJS moga modyfikować żadania i odpowiedzi oraz realizować dodatkowe operacje, takie jak logowanie czy modyfikacja danych odpowiedzi.

9.1 Przykład Interceptora

```
import { Injectable, NestInterceptor, ExecutionContext,
   CallHandler } from '@nestjs/common';
import { Observable } from 'rxjs';
```

```
import { tap } from 'rxjs/operators';

@Injectable()
export class ResponseInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler):
     Observable<any> {
    console.log('Before handling request');
    return next.handle().pipe(
      tap(() => console.log('After handling request'))
    );
  }
}
```

Listing 8: Przykład interceptora w NestJS

10 Protected Routes

Protected routes w NestJS sa zabezpieczane za pomoca guardów, które sprawdzaja autoryzacje przed dostepem do chronionych zasobów.

10.1 Przykład Chronionej Trasy

```
import { Module } from '@nestjs/common';
import { AuthGuard } from './auth.guard';
import { UsersController } from './users.controller';

@Module({
  controllers: [UsersController],
  providers: [AuthGuard],
})
export class AppModule {}
```

Listing 9: Przykład chronionej trasy w NestJS

11 Łaczenie sie z Baza Danych

NestJS wspiera wiele bibliotek ORM i ODM, takich jak TypeORM i Mongoose.

11.1 TypeORM

TypeORM to popularny ORM dla TypeScript i JavaScript. Aby połaczyć sie z baza danych przy użyciu TypeORM, należy skonfigurować 'TypeOrmModule' w module głównym.

11.1.1 Przykład Konfiguracji TypeORM

```
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './user.entity';

@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'mysql',
      host: 'localhost',
      port: 3306,
      username: 'root',
      password: 'password',
      database: 'test',
      entities: [User],
      synchronize: true,
    }),
    TypeOrmModule.forFeature([User]),
  ],
  // inne konfiguracje
})
export class AppModule {}
```

Listing 10: Konfiguracja TypeORM w NestJS

11.2 Mongoose

Mongoose to popularne ODM dla MongoDB. Aby połaczyć sie z MongoDB przy użyciu Mongoose, należy skonfigurować 'MongooseModule' w module głównym.

11.2.1 Przykład Konfiguracji Mongoose

```
import { MongooseModule } from '@nestjs/mongoose';
import { User, UserSchema } from './schemas/user.schema';

@Module({
  imports: [
    MongooseModule.forRoot('mongodb://localhost/nest'),
    MongooseModule.forFeature([{ name: User.name, schema:
       UserSchema }]),
  ],
  // inne konfiguracje
})
export class AppModule {}
```

Listing 11: Konfiguracja Mongoose w NestJS

12 Inne Ważne Aspekty

12.1 Konfiguracja Bazy Danych

NestJS obsługuje różne bazy danych przez TypeORM, Mongoose i inne ORM/ODM. Ważne jest, aby dobrze skonfigurować połaczenie z baza danych i zarzadzać migracjami oraz schematami.

12.2 Wstrzykiwanie Zależności

NestJS używa Dependency Injection (DI) do zarzadzania zależnościami, co ułatwia testowanie i modularność aplikacji. DI jest kluczowym elementem architektury NestJS i umożliwia tworzenie luźno powiazanych komponentów.