Nest.js

DELTA - Střední škola informatiky a ekonomie, s.r.o.

Ing. Luboš Zápotočný

02.02.2026

Úvod do Nest.js

Co je Nest.js?

Nest.js je progresivní Node.js framework pro budování **škálovatelných** serverových aplikací

- Postaven na **TypeScriptu** (s podporou čistého JS)
- Inspirován **Angular** architekturou
- Využívá **Express.js** (nebo Fastify) pod kapotou
- Podporuje **OOP**, **FP** (functional programming) i **FRP** (functional reactive programming) paradigmata
- Výborná podpora pro **testování**

Co je Nest.js?

Proč používat Nest.js?

- **Struktura** - jasná architektura od začátku
- **Modularita** - rozdělení aplikace do modulů
- **Dependency Injection** - vestavěný DI kontejner
- **Dekorátory** - čistý a čitelný kód
- **Dokumentace** - skvělá oficiální dokumentace
- **Ekosystém** - podpora pro GraphQL, WebSockets, mikroslužby, ...

Co je Nest.js?

Architektura Nest.js:

Porovnání s Express.js

Vlastnost	Express.js	Nest.js
Struktura	Volná	Definovaná
TypeScript	Volitelný	Nativní
DI	Manuální	Vestavěné
Dekorátory	Ne	Ano
Moduly	Manuální	Vestavěné
Testování	Složitější	Jednodušší

Instalace a první projekt

Instalace Nest.js CLI:

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

Vytvoření nového projektu:

```
nest new my-nest-app
```

Spuštění aplikace:

```
cd my-nest-app
npm run start:dev
```

Instalace a první projekt

Struktura projektu:

```
src/
├── app.controller.ts    # Controller
├── app.controller.spec.ts
├── app.module.ts        # Root module
├── app.service.ts       # Service
└── main.ts              # Entry point
```

Instalace a první projekt

Entry point - `main.ts`:

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';

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

Základní koncepty Nest.js

Moduly

Modul je třída označená dekorátorem `@Module()`

- Organizuje související komponenty
- Definuje **providery**, **controllery**, **importy** a **exporty**
- Každá aplikace má minimálně jeden **root modul**

Moduly

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

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

Moduly

Vlastnosti dekorátoru `@Module()`:

- `imports` - moduly, jejichž exportované providery potřebujeme
- `controllers` - controllery definované v tomto modulu
- `providers` - služby dostupné v tomto modulu
- `exports` - služby, které modul zpřístupňuje jiným modulům

Moduly

Root modul - `app.module.ts`:

```
import { Module } from '@nestjs/common';
import { UsersModule } from './users/users.module';
import { PostsModule } from './posts/posts.module';

@Module({
  imports: [UsersModule, PostsModule],
})
export class AppModule {}
```

Controllers

Controller je třída označená dekorátorem `@Controller()`

- Zpracovává **příchozí požadavky**
- Vrací **odpovědi** klientovi
- Používá **route dekorátory** (`@Get`, `@Post`, ...)

Controllers

```
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) {}
```

Controllers

```
@Get()
findAll() {
  return this.usersService.findAll();
}

@Get(':id')
findOne(@Param('id') id: string) {
  return this.usersService.findOne(+id);
}
```

Controllers

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

Controllers

Dekorátory pro HTTP metody:

- `@Get()` - GET požadavky
- `@Post()` - POST požadavky
- `@Put()` - PUT požadavky
- `@Patch()` - PATCH požadavky
- `@Delete()` - DELETE požadavky

Controllers

Dekorátory pro parametry:

- `@Param()` - route parametry (`/users/:id`)
- `@Query()` - query parametry (`?name=John`)
- `@Body()` - tělo požadavku
- `@Headers()` - HTTP hlavičky
- `@Req()` / `@Res()` - raw request/response objekty

Services a Providers

Provider je třída, která může být **injektována** jako závislost

Service je nejčastější typ provideru - obsahuje **business logiku**

Označuje se dekorátorem `@Injectable()`

Services a Providers

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

@Injectable()
export class UsersService {
  private users = [];

  findAll() {
    return this.users;
  }
```

Services a Providers

```
  findOne(id: number) {
    return this.users.find(user => user.id === id);
  }

  create(createUserDto: CreateUserDto) {
    const user = { id: Date.now(), ...createUserDto };
    this.users.push(user);
    return user;
  }
}
```

Dependency Injection

Dependency Injection (DI) je návrhový vzor, kdy třída přijímá závislosti zvenčí

- Nest.js má **vestavěný DI kontejner**
- Závislosti se injektují přes **konstruktor**
- Automatická správa životního cyklu instancí

Dependency Injection

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

  @Get()
  findAll() {
    return this.usersService.findAll();
  }
}
```

Dependency Injection

Scopes providerů:

- **DEFAULT** - singleton (jedna instance pro celou aplikaci)
- **REQUEST** - nová instance pro každý request
- **TRANSIENT** - nová instance při každé injekci

```
@Injectable({ scope: Scope.REQUEST })
export class UsersService {}
```

Data Transfer Objects (DTO)

DTO je objekt, který definuje strukturu přenášených dat

- Typová bezpečnost pro request body
- Základ pro validaci dat
- Dokumentace API struktury

Data Transfer Objects (DTO)

```ts
// src/users/dto/create-user.dto.ts
export class CreateUserDto {
  name: string;
  email: string;
  password: string;
}
```

Data Transfer Objects (DTO)

```ts
// src/users/dto/update-user.dto.ts
import { PartialType } from '@nestjs/mapped-types';
import { CreateUserDto } from './create-user.dto';

export class UpdateUserDto extends PartialType(CreateUserDto)
{}
```

`PartialType` vytvoří typ, kde jsou všechny vlastnosti **volitelné**

Validace s class-validator

Instalace validačních balíčků:

```
npm install class-validator class-transformer
```

Validace s class-validator

Povolení globální validace:

```
// main.ts
import { ValidationPipe } from '@nestjs/common';

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

Validace s class-validator

```
import { IsEmail, IsNotEmpty, MinLength }
  from 'class-validator';

export class CreateUserDto {
  @IsNotEmpty()
  name: string;
  @IsEmail()
  email: string;
  @MinLength(8)
  password: string;
}
```

Validace s class-validator

Často používané validační dekorátory:

- `@IsNotEmpty()` - nesmí být prázdné
- `@IsString()` / `@IsNumber()` / `@IsBoolean()`
- `@IsEmail()` - validní email
- `@MinLength()` / `@MaxLength()`
- `@Min()` / `@Max()` - pro čísla
- `@IsOptional()` - volitelné pole

Objektově-relační mapování (ORM)

Co je ORM?

ORM (Object-Relational Mapping) je technika pro mapování objektů na databázové tabulky

- Pracujeme s **objekty** místo SQL dotazů
- Automatická synchronizace mezi kódem a databází
- Abstrakce nad různými databázemi
- Typová bezpečnost při práci s daty

Co je ORM?

Porovnání přístupu:

```sql
-- Raw SQL
SELECT * FROM users WHERE id = 1;

INSERT INTO users (name, email) VALUES
  ('John', 'john@example.com');
```

Co je ORM?

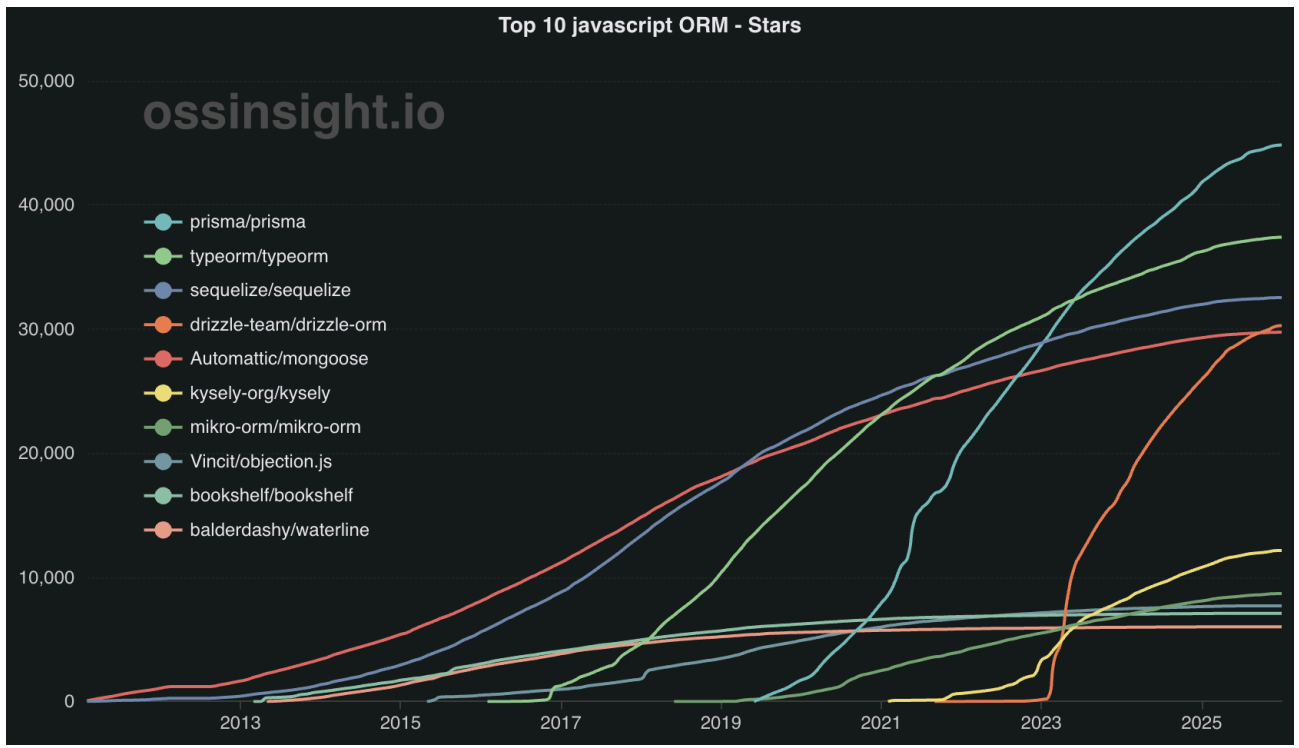
```
// ORM
const user = await userRepository.findOne({
  where: { id: 1 }
});

const newUser = await userRepository.save({
  name: 'John',
  email: 'john@example.com',
});
```

TypeORM

TypeORM je jeden z nejpopulárnějších ORM pro TypeScript a JavaScript

- Podpora pro MySQL, PostgreSQL, SQLite, MSSQL, …
- **Dekorátory** pro definici entit
- **Repository pattern**
- **Migrace** pro správu schématu
- **Relations** - vztahy mezi entitami

TypeORM



Popularita ORM v JavaScriptu ossinsight.io

TypeORM

Instalace TypeORM pro Nest.js:

```
npm install @nestjs/typeorm typeorm pg
```

- `@nestjs/typeorm` - integrace do Nest.js
- `typeorm` - samotné ORM
- `pg` - PostgreSQL driver (nebo `mysql2` pro MySQL)

Konfigurace TypeORM

Konfigurace v `app.module.ts`:

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';

@Module({
  imports: [
    TypeOrmModule.forRoot({
      type: 'postgres',
      host: 'localhost',
      port: 5432,
```

Konfigurace TypeORM

```
      username: 'user',
      password: 'password',
      database: 'mydb',
      entities: [__dirname + '/**/*.entity{.ts,.js}'],
      synchronize: true, // ONLY for development!
    }),
  ],
})
export class AppModule {}
```

Konfigurace TypeORM

Důležité: `synchronize: true` automaticky synchronizuje schéma databáze s entitami

- Výhodné pro **vývoj**
- **Nikdy** nepoužívejte v produkci!
- V produkci používejte **migrace**

Entity

Entita je třída, která reprezentuje databázovou tabulku

Definuje se pomocí dekorátorů z TypeORM

Entity

```
import { Entity, PrimaryGeneratedColumn, Column }
  from 'typeorm';

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

  @Column()
  name: string;
```

Entity

```
  @Column({ unique: true })
  email: string;

  @Column()
  password: string;

  @CreateDateColumn()
  createdAt: Date;
}
```

Registrace entity v modulu

```
import { Module } from '@nestjs/common';
import { TypeOrmModule } from '@nestjs/typeorm';
import { User } from './entities/user.entity';
import { UsersService } from './users.service';
import { UsersController } from './users.controller';
@Module({
  imports: [TypeOrmModule.forFeature([User])],
  providers: [UsersService],
  controllers: [UsersController],
})
export class UsersModule {}
```

Typy sloupců

Běžné typy sloupců:

- `@Column()` - běžný sloupec
- `@PrimaryGeneratedColumn()` - auto-increment ID
- `@PrimaryGeneratedColumn('uuid')` - UUID jako primární klíč
- `@CreateDateColumn()` - automaticky vyplněné datum vytvoření
- `@UpdateDateColumn()` - automaticky aktualizované datum

Možnosti sloupců

```
@Column({ type: 'varchar', length: 255 })
name: string;

@Column({ unique: true })
email: string;

@Column({ nullable: true })
bio: string;

@Column({ default: 0 })
points: number;
```

Repository Pattern

Repository je třída pro přístup k datům entity

- TypeORM poskytuje vestavěný `Repository<Entity>`
- Metody pro CRUD operace
- Query Builder pro složitější dotazy

Repository Pattern

Injektování repository do service:

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { User } from './entities/user.entity';
@Injectable()
export class UsersService {
  constructor(
    @InjectRepository(User)
    private usersRepository: Repository<User>,
  ) {}
```

Repository Pattern

```
findAll(): Promise<User[]> {
  return this.usersRepository.find();
}

findOne(id: number): Promise<User | null> {
  return this.usersRepository.findOneBy({ id });
}
```

Repository Pattern

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

  async remove(id: number): Promise<void> {
    await this.usersRepository.delete(id);
  }
}
```

Repository Pattern

Rozdíl mezi `create` a `save`:

- `create()` - vytvoří instanci entity **bez uložení** do DB
- `save()` - uloží entitu do databáze

Repository Pattern

```
// Correct approach

const user = this.usersRepository.create(dto);
// creates object

await this.usersRepository.save(user); // saves to DB

// Shorthand
await this.usersRepository.save(dto);  // creates and saves
```

Vztahy mezi entitami

TypeORM podporuje všechny typy vztahů:

- **One-to-One** - 1:1 (uživatel má jeden profil)
- **One-to-Many** / **Many-to-One** - 1:N (uživatel má více příspěvků)
- **Many-to-Many** - M:N (uživatel má více rolí, role má více uživatelů)

One-to-Many

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

  @Column()
  name: string;

  @OneToMany(() => Post, (post) => post.author)
  posts: Post[];
}
```

One-to-Many

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

  @Column()
  title: string;

  @ManyToOne(() => User, (user) => user.posts)
  author: User;
}
```

Možnosti načítání vztahů

Eager (dychtivé) načítání

```
// Eager loading - loads relations automatically
@OneToMany(() => Post, (post) => post.author, {
  eager: true
})
posts: Post[];
```

Možnosti načítání vztahů

Explicit (na vyžádání) načítání

```
// Explicit loading - loads relations on demand
const user = await this.usersRepository.findOne({
  where: { id },
  relations: ['posts'],
});
```

Možnosti načítání vztahů

Many-to-Many vztah:

```
@Entity()
export class User {
  @ManyToMany(() => Role, (role) => role.users)
  @JoinTable()
  roles: Role[];
}
```

`@JoinTable()` se uvádí na **jedné straně** vztahu - vytvoří vazební tabulku

Možnosti načítání vztahů

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

  @Column()
  name: string;

  @ManyToMany(() => User, (user) => user.roles)
  users: User[];
}
```

Migrace

Co jsou migrace?

Migrace jsou verzované změny databázového schématu

- Umožňují **trackovat** změny v databázi
- Lze je **aplikovat** a **vrátit zpět**
- Sdílení změn v **týmu**
- Bezpečný **deployment** do produkce

Co jsou migrace?

Proč nepoužívat `synchronize: true`?

- Může **smazat data** při změně schématu
- Žádná **kontrola** nad změnami
- Nelze **vrátit zpět** změny
- Nebezpečné pro **produkční** prostředí

Konfigurace migrací

Vytvoření konfiguračního souboru `data-source.ts`:

```
import { DataSource } from 'typeorm';

export const AppDataSource = new DataSource({
  type: 'postgres',
  host: 'localhost',
  port: 5432,
  username: 'user',
  password: 'password',
  database: 'mydb',
```

Konfigurace migrací

```
  entities: ['src/**/*.entity.ts'],
  migrations: ['src/migrations/*.ts'],
  synchronize: false,
});
```

Příkazy pro migrace

```
{
  "scripts": {
    "typeorm": "ts-node -r tsconfig-paths/register
      ./node_modules/typeorm/cli.js
      -d src/data-source.ts",
    "migration:generate": "npm run typeorm
migration:generate",
    "migration:run": "npm run typeorm migration:run",
    "migration:revert": "npm run typeorm migration:revert"
  }
}
```

Práce s migracemi

Generování migrace na základě změn v entitách:

```
npm run migration:generate -- src/migrations/CreateUser
```

TypeORM porovná entity s databází a vygeneruje migraci

Práce s migracemi

```
import { MigrationInterface, QueryRunner } from 'typeorm';
export class CreateUser1234567890 implements
MigrationInterface {
  public async up(queryRunner: QueryRunner): Promise<void> {
    await queryRunner.query(`
      CREATE TABLE "user" (
        "id" SERIAL PRIMARY KEY, "name" VARCHAR NOT NULL,
        "email" VARCHAR NOT NULL UNIQUE
      )
    `);
  }
```

Práce s migracemi

```
  public async down(queryRunner: QueryRunner): Promise<void>
  {
    await queryRunner.query(`DROP TABLE "user"`);
  }
}
```

- `up()` - aplikuje změny
- `down()` - vrátí změny zpět

Práce s migracemi

Spuštění migrací:

```
npm run migration:run
```

Vrácení poslední migrace:

```
npm run migration:revert
```

Práce s migracemi

Best practices pro migrace:

- **Nikdy** neměňte již aplikované migrace
- Vždy testujte `up()` i `down()` metody
- Používejte **popisné názvy** migrací
- Migrujte **postupně** - malé, atomické změny
- Zálohujte databázi před migrací v produkci

Architektura API

REST API

REST (Representational State Transfer) je architektonický styl pro webové API

Základní principy:

- **Stateless** - každý požadavek obsahuje všechny potřebné informace
- **Resource-based** - práce se zdroji přes URL
- **HTTP metody** - CRUD operace (GET, POST, PUT, DELETE)

REST API

RESTful routes v Nest.js:

```
@Controller('users')
export class UsersController {
  @Get()             // GET    /users
  findAll() {}

  @Get(':id')        // GET    /users/:id
  findOne() {}

  @Post()            // POST   /users
  create() {}
```

REST API

```
  @Patch(':id')     // PATCH  /users/:id
  update() {}

  @Delete(':id')    // DELETE /users/:id
  remove() {}
}
```

REST API

HTTP status kódy:

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

@Post()
@HttpCode(HttpStatus.CREATED)  // 201
create() {}

@Delete(':id')
@HttpCode(HttpStatus.NO_CONTENT)  // 204
remove() {}
```

REST API

Generátor CRUD resourceů:

```
nest generate resource users
```

Vytvoří kompletní strukturu:

- Controller s CRUD endpointy
- Service
- DTO (create, update)
- Entity
- Module

Swagger dokumentace

Swagger (OpenAPI) je standard pro dokumentaci REST API

Instalace:

```
npm install @nestjs/swagger
```

Swagger dokumentace

Konfigurace v `main.ts`:

```
import { SwaggerModule, DocumentBuilder } from '@nestjs/
swagger';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  const config = new DocumentBuilder()
    .setTitle('My API')
    .setDescription('API description')
    .setVersion('1.0')
    .build();
```

Swagger dokumentace

```
  const document = SwaggerModule.createDocument(app, config);
  SwaggerModule.setup('api', app, document);

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

Dokumentace je dostupná na `/api`

Swagger dokumentace

Dekorátory pro dokumentaci:

```
import { ApiTags, ApiOperation, ApiResponse } from '@nestjs/
swagger';
@ApiTags('users')
@Controller('users')
export class UsersController {
  @Get()
  @ApiOperation({ summary: 'Get all users' })
  @ApiResponse({ status: 200, description: 'List of users' })
  findAll() {}
}
```

Swagger dokumentace

Dokumentace DTO:

```
import { ApiProperty } from '@nestjs/swagger';

export class CreateUserDto {
  @ApiProperty({ example: 'John Doe' })
  name: string;

  @ApiProperty({ example: 'john@example.com' })
  email: string;
}
```

GraphQL

GraphQL je query jazyk pro API vyvinutý Facebookem

Rozdíly oproti REST:

- **Jeden endpoint** místo mnoha
- Klient **přesně specifikuje** jaká data chce
- Žádné **over-fetching** nebo **under-fetching**
- Silně **typovaný** systém

GraphQL

REST vs GraphQL:

```
// REST - multiple requests, fixed structure
GET /users/1
GET /users/1/posts
GET /users/1/followers
```

GraphQL

```
# GraphQL - single request, flexible structure
query {
  user(id: 1) {
    name
    posts { title }
    followers { name }
  }
}
```

GraphQL

Instalace GraphQL pro Nest.js:

```
npm install @nestjs/graphql @nestjs/apollo
npm install @apollo/server graphql
```

GraphQL

Konfigurace GraphQL:

```
// ... imports ...

@Module({
  imports: [
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver, autoSchemaFile: 'schema.gql',
    }),
  ],
})
export class AppModule {}
```

GraphQL - Code First

Nest.js podporuje dva přístupy k GraphQL:

- **Code First** - schéma se generuje z TypeScript kódu
- **Schema First** - píšeme GraphQL schéma ručně

Code First je preferovaný přístup pro TypeScript projekty

GraphQL - Code First

Definice ObjectType (entity pro GraphQL):

```
import { ObjectType, Field, Int } from '@nestjs/graphql';
@ObjectType()
export class User {
  @Field(() => Int)
  id: number;
  @Field()
  name: string;
  @Field()
  email: string;
}
```

Resolver

```
@Resolver(() => User)
export class UsersResolver {
  constructor(private usersService: UsersService) {}
  @Query(() => [User])
  users() {
    return this.usersService.findAll();
  }
  @Query(() => User)
  user(@Args('id', { type: () => Int }) id: number) {
    return this.usersService.findOne(id);
  }
```

Resolver

```
  @Mutation(() => User)
  createUser(
    @Args('createUserInput') createUserInput: CreateUserInput
  ) {
    return this.usersService.create(createUserInput);
  }
}
```

Resolver

```
import { InputType, Field } from '@nestjs/graphql';

@InputType()
export class CreateUserInput {
  @Field()
  name: string;
  @Field()
  email: string;
  @Field()
  password: string;
}
```

Resolver

GraphQL Playground je dostupný na `/graphql`:

```
# Query
query {
  users {
    id
    name
    email
  }
}
```

Resolver

```
# Mutation
mutation {
  createUser(createUserInput: {
    name: "John"
    email: "john@example.com"
    password: "secret"
  }) {
    id
    name
  }
}
```

REST vs GraphQL - kdy co použít?

REST je vhodný pro:

- **Jednoduché** CRUD operace
- **Cachování** (HTTP cache funguje out of the box)
- **Veřejná** API s fixní strukturou
- Týmy **zvyklé** na REST

REST vs GraphQL - kdy co použít?

GraphQL je vhodný pro:

- **Komplexní** datové struktury s mnoha vztahy
- **Mobilní** aplikace (šetří bandwidth)
- **Rapidní** vývoj frontendu
- Situace kdy potřebujeme **flexibilitu** v datech

Middleware a Guards

Middleware

Middleware v Nest.js funguje stejně jako v Express.js

- Zpracovává požadavky **před** route handlerem
- Může modifikovat request a response
- Může ukončit request-response cyklus

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(`[${new Date().toISOString()}] ${req.method}
${req.url}`);
    next();
  }
}
```

Middleware

Registrace middleware v modulu:

```
@Module({})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes('*');
  }
}
```

Guards

Guard je třída pro **autorizaci** požadavků

- Rozhoduje, zda požadavek **může pokračovat**
- Vrací `true` nebo `false`
- Běží **po** middleware, **před** interceptory

Guards

```
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 this.validateToken(token);
  }
```

Guards

```
  private validateToken(token: string): boolean {
    // Validate JWT token
    return token === 'valid-token';
  }
}
```

Guards

Použití guards:

```
// On controller
@Controller('users')
@UseGuards(AuthGuard)
export class UsersController {}

// On method
@Get('profile')
@UseGuards(AuthGuard)
getProfile() {}
```

```
// Globally in main.ts
app.useGlobalGuards(new AuthGuard());
```

Pipes

Pipe je třída pro **transformaci** a **validaci** dat

- Transformuje vstupní data
- Validuje vstupní data
- Může vyhodit výjimku při nevalidních datech

Pipes

Vestavěné pipes:

- `ValidationPipe` - validace pomocí class-validator
- `ParseIntPipe` - převod na číslo
- `ParseUUIDPipe` - validace UUID
- `ParseBoolPipe` - převod na boolean
- `DefaultValuePipe` - výchozí hodnota

Pipes

```
@Get(':id')
findOne(@Param('id', ParseIntPipe) id: number) {
  // id is guaranteed to be a number
  return this.usersService.findOne(id);
}

@Get()
findAll(@Query('limit', new DefaultValuePipe(10),
ParseIntPipe) limit: number) {
  return this.usersService.findAll(limit);
}
```

Exception Filters

Exception Filter je třída pro zpracování výjimek

- Zachytává výjimky z aplikace
- Transformuje je na HTTP odpovědi
- Jednotná struktura chybových odpovědí

Exception Filters

Nest.js má vestavěné HTTP výjimky:

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

@Get(':id')
findOne(@Param('id') id: string) {
  const user = this.usersService.findOne(+id);
  if (!user) {
    throw new NotFoundException('User not found');
  }
  return user;
}
```

Exception Filters

Vlastní exception filter:

```
import { ExceptionFilter, Catch, ArgumentsHost,
HttpException } from '@nestjs/common';

@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse();
    const status = exception.getStatus();
```

Exception Filters

```
    response.status(status).json({
      statusCode: status,
      timestamp: new Date().toISOString(),
      message: exception.message,
    });
  }
}
```

Praktický příklad

Kompletní CRUD modul

Struktura users modulu:

```
src/users/
├── dto/
│   ├── create-user.dto.ts
│   └── update-user.dto.ts
├── entities/
│   └── user.entity.ts
├── users.controller.ts
├── users.service.ts
└── users.module.ts
```

Kompletní CRUD modul

user.entity.ts:

```
import { Entity, PrimaryGeneratedColumn, Column,
CreateDateColumn } from 'typeorm';

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

  @Column()
  name: string;
```

Kompletní CRUD modul

```
  @Column({ unique: true })
  email: string;

  @Column()
  password: string;

  @Column({ default: true })
  isActive: boolean;
```

Kompletní CRUD modul

```
  @CreateDateColumn()
  createdAt: Date;
}
```

Kompletní CRUD modul

create-user.dto.ts:

```
import { IsEmail, IsNotEmpty, MinLength } from 'class-
validator';

export class CreateUserDto {
  @IsNotEmpty()
  name: string;

  @IsEmail()
  email: string;
```

```
  @MinLength(8)
  password: string;
}
```

Kompletní CRUD modul

users.service.ts:

```
// ...

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

Kompletní CRUD modul

```
  findAll(): Promise<User[]> {
    return this.usersRepository.find();
  }
  async findOne(id: number): Promise<User> {
    const user = await
this.usersRepository.findOneBy({ id });
    if (!user) {
      throw new NotFoundException(`User #${id} not found`);
    }
    return user;
  }
```

Kompletní CRUD modul

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

  async update(id: number, updateUserDto: UpdateUserDto):
Promise<User> {
    await this.usersRepository.update(id, updateUserDto);
    return this.findOne(id);
  }
```

Kompletní CRUD modul

```
  async remove(id: number): Promise<void> {
    const result = await this.usersRepository.delete(id);
    if (result.affected === 0) {
      throw new NotFoundException(`User #${id} not found`);
    }
  }
}
```

Kompletní CRUD modul

users.controller.ts:

```
// ...

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

Kompletní CRUD modul

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

  @Get()
  findAll() {
    return this.usersService.findAll();
  }
```

Kompletní CRUD modul

```
  @Get(':id')
  findOne(@Param('id', ParseIntPipe) id: number) {
    return this.usersService.findOne(id);
  }
```

Kompletní CRUD modul

```
  @Patch(':id')
  update(
    @Param('id', ParseIntPipe) id: number,
    @Body() updateUserDto: UpdateUserDto
  ) {
    return this.usersService.update(id, updateUserDto);
  }
```

Kompletní CRUD modul

```
  @Delete(':id')
  remove(@Param('id', ParseIntPipe) id: number) {
    return this.usersService.remove(id);
  }
}
```

Shrnutí

Shrnutí

Základy Nest.js:

- **Moduly** - organizace kódu
- **Controllers** - zpracování požadavků
- **Services** - business logika
- **Providers** - dependency injection

Shrnutí

TypeORM:

- **Entity** - mapování na databázovou tabulku
- **Repository** - přístup k datům
- **Relations** - vztahy mezi entitami
- **Migrace** - verzování schématu

Shrnutí

Architektura API:

- **REST** - resource-based API s HTTP metodami
- **GraphQL** - flexibilní query jazyk
- **Swagger** - dokumentace API

Shrnutí

Pokročilé koncepty:

- **Guards** - autorizace
- **Pipes** - validace a transformace
- **Exception Filters** - zpracování chyb
- **Middleware** - zpracování požadavků

Užitečné zdroje

Oficiální dokumentace

- Nest.js: https://docs.nestjs.com
- TypeORM: https://typeorm.io
- GraphQL: https://graphql.org

Další zdroje

- Nest.js Courses: https://courses.nestjs.com
- Awesome Nest.js: https://github.com/nestjs/awesome-nestjs

Užitečné zdroje

Doporučené balíčky:

- `@nestjs/config` - konfigurace z environment variables
- `@nestjs/passport` - autentizace (JWT, OAuth, ...)
- `@nestjs/cache-manager` - cachování
- `@nestjs/schedule` - cron jobs
- `@nestjs/bull` - job queues