Zaawansowany typescript - NestJS

Szymon Zych, Sandra Góra,
Damian Zyznar, Jakub Stec

TypeScript

TypeScript to język programowania, który jest rozwinięciem języka JavaScript. Jest to język typowany statycznie, co oznacza, że w przeciwieństwie do JavaScriptu, w TypeScript zmiennym i wartościom przypisywanym do zmiennych można przypisać określony typ, a kompilator TypeScript może wykrywać błędy typów podczas kompilacji. TypeScript jest zwykle używany w dużych projektach, które wymagają zwiększonej jakości kodu, łatwiejszego debugowania i utrzymania.

TypeScript jest rozwijany przez Microsoft i jest w pełni zgodny z językiem JavaScript, co oznacza, że kod TypeScript może być bezpośrednio uruchamiany w środowiskach uruchomieniowych JavaScript. TypeScript dodaje wiele funkcji, takich jak interfejsy, klasy, dziedziczenie, wyliczenia, typy generyczne i wiele innych, które nie są dostępne w JavaScript.

NestJS

NestJS to platforma aplikacji serwerowej (backend) dla Node.js, oparta na architekturze MVC (Model-View-Controller), zorientowana na moduły i rozwijana z myślą o tworzeniu skalowalnych i wydajnych aplikacji webowych. Jest to framework TypeScript, który wykorzystuje typowanie statyczne i pozwala na łatwiejsze tworzenie i utrzymywanie kodu.

NestJS zapewnia wiele gotowych modułów, narzędzi i bibliotek, takich jak moduł obsługi żądań HTTP, moduł autentykacji, moduł bazy danych i wiele innych, które ułatwiają tworzenie zaawansowanych aplikacji. Ponadto, NestJS wspiera architekturę mikrousług i pozwala na łatwe tworzenie API REST oraz gniazd WebSocket.

NestJS jest zainspirowany Angular i wykorzystuje podobne koncepcje, takie jak moduły, wstrzykiwanie zależności i dekoratory. Dzięki temu, programiści z doświadczeniem w Angularze mogą szybko nauczyć się NestJS i wykorzystać swoją wiedzę w tworzeniu aplikacji serwerowych.

Implementacja

Przedstawiamy aplikacją webową - backendową napisaną w języku TypeScript z wykorzystaniem frameworka NestJS. Aplikacja składa się z kilku modułów: `AppModule`, `UsersModule` oraz kilku usług, kontrolerów i modeli.

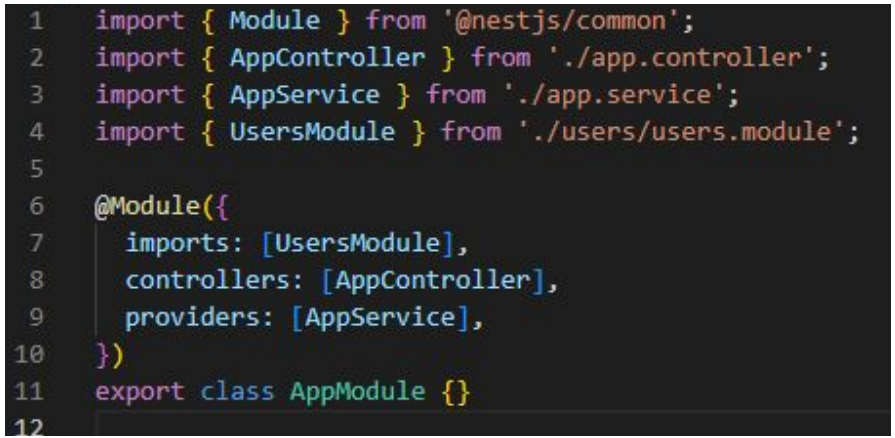
`AppModule` jest głównym modułem aplikacji, który importuje i agreguje pozostałe moduły oraz definiuje kontrolery i usługi dostępne na poziomie aplikacji. W tym module znajduje się kontroler `AppController`, który obsługuje jedno proste zapytanie GET i zwraca napis "Hello World!".

`UsersModule` to moduł, który obsługuje operacje na użytkownikach. Zawiera usługę `UsersService`, która implementuje logikę biznesową związaną z użytkownikami, a także kontroler `UsersController`, który definiuje punkty końcowe API do obsługi operacji CRUD (tworzenie, odczyt, aktualizacja i usuwanie) użytkowników.

Ponadto, w projekcie znajduje się model `User`, który definiuje strukturę danych użytkownika, a także testy jednostkowe dla `UsersService`.

Plik konfiguracyjny

1. app.module.ts - plik konfiguracyjny modułu głównego aplikacji, zawierający konfigurację kontrolerów, dostawców (provider) i modułów zaimportowanych do aplikacji. W tym pliku importowane są moduły zawierające logikę biznesową aplikacji, takie jak moduł użytkowników.



```
1  import { Module } from '@nestjs/common';
2  import { AppController } from './app.controller';
3  import { AppService } from './app.service';
4  import { UsersModule } from './users/users.module';
5
6  @Module({
7    imports: [UsersModule],
8    controllers: [AppController],
9    providers: [AppService],
10 })
11 export class AppModule {}
12
```

Plik kontrolera

app.controller.ts - plik kontrolera aplikacji, zawierający endpointy, czyli miejsca, gdzie aplikacja odbiera żądania HTTP i zwraca odpowiedzi. W tym pliku zdefiniowany jest endpoint, który zwraca wiadomość "Hello World!".

```
import { Controller, Get } from '@nestjs/common';
import { AppService } from './app.service';

@Controller()
export class AppController {
  constructor(private readonly appService: AppService) {}

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

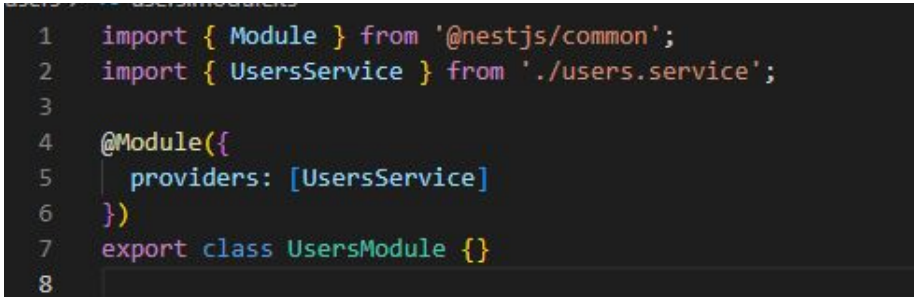
Plik obsługi aplikacji

app.service.ts - plik usługi aplikacji, zawierający logikę biznesową endpointów kontrolera aplikacji. W tym pliku zdefiniowana jest metoda getHello() zwracająca "Hello World!".

```
1  import { Injectable } from '@nestjs/common';
2
3  @Injectable()
4  export class AppService {
5    getHello(): string {
6      return 'Hello World!';
7    }
8  }
```

Plik modułu użytkownika

user.module.ts - plik modułu użytkowników, zawierający konfigurację dostawców (provider) i eksportowanych usług, które będą wykorzystywane przez kontrolery.



```
1  import { Module } from '@nestjs/common';
2  import { UsersService } from './users.service';
3
4  @Module({
5    providers: [UsersService]
6  })
7  export class UsersModule {}
8
```

Plik kontrolera użytkownika

user.controller.ts - plik kontrolera użytkowników, zawierający endpointy związane z operacjami na użytkownikach, takimi jak tworzenie, pobieranie, aktualizacja i usuwanie użytkowników.

```typescript
import { Controller, Get, Post, Body, Param, Put, Delete } from '@nestjs/common';
import { UsersService } from './users.service';
import { User } from './user.model';

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

  @Post()
  create(@Body() user: User): User {
    return this.usersService.create(user);
  }

  @Get()
  findAll(): User[] {
    return this.usersService.findAll();
  }

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

  @Put(':id')
  update(@Param('id') id: string, @Body() user: User): User {
    return this.usersService.update(+id, user);
  }

  @Delete(':id')
  remove(@Param('id') id: string): void {
    return this.usersService.remove(+id);
  }
}
```

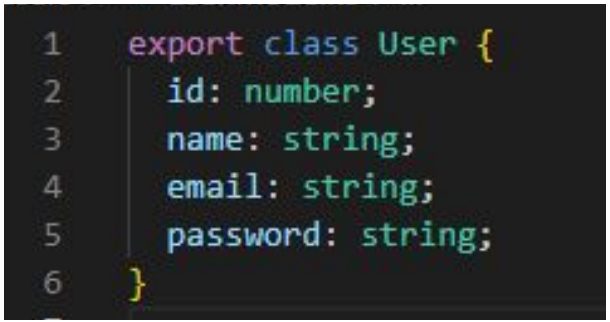
Plik user.service

user.service.ts - plik usługi użytkowników, zawierający logikę biznesową operacji na użytkownikach, takie jak dodawanie, pobieranie, aktualizowanie i usuwanie użytkowników.

```typescript
import { Injectable } from '@nestjs/common';
import { User } from './user.model';

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

  create(user: User): User {
    user.id = this.users.length + 1;
    this.users.push(user);
    return user;
  }

  findAll(): User[] {
    return this.users;
  }

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

  update(id: number, user: User): User {
    const index = this.users.findIndex(user => user.id === id);
    this.users[index] = { id, ...user };
    return this.users[index];
  }

  remove(id: number): void {
    const index = this.users.findIndex(user => user.id === id);
    this.users.splice(index, 1);
  }
}
```

Plik user.model

user.model.ts - plik modelu użytkownika, zawierający definicję struktury danych użytkownika.



Pliki testów jednostkowych

app.controller.spec.ts oraz user.service.spec.ts - pliki testów jednostkowych kontrolera aplikacji oraz usługi użytkowników. Są to pliki zawierające testy jednostkowe, które zapewniają poprawne działanie aplikacji i wykrywają błędy przed wdrożeniem.

```ts
import { Test, TestingModule } from '@nestjs/testing';
import { UsersService } from './users.service';

describe('UsersService', () => {
  let service: UsersService;

  beforeEach(async () => {
    const module: TestingModule = await Test.createTestingModule({
      providers: [UsersService],
    }).compile();

    service = module.get<UsersService>(UsersService);
  });

  it('should be defined', () => {
    expect(service).toBeDefined();
  });
});
```

```ts
import { Test, TestingModule } from '@nestjs/testing';
import { AppController } from './app.controller';
import { AppService } from './app.service';

describe('AppController', () => {
  let appController: AppController;

  beforeEach(async () => {
    const app: TestingModule = await Test.createTestingModule({
      controllers: [AppController],
      providers: [AppService],
    }).compile();

    appController = app.get<AppController>(AppController);
  });

  describe('root', () => {
    it('should return "Hello World!"', () => {
      expect(appController.getHello()).toBe('Hello World!');
    });
  });
});
```

Zalety TypeScript w połączeniu z NestJS

1. Typowanie - TypeScript to język oparty na typach, co oznacza, że można zapewnić większą pewność co do poprawności kodu i zmniejszyć ryzyko błędów podczas tworzenia aplikacji.
2. Intellisense - Dzięki typowaniu TypeScript i silnikowi IntelliSense, IDE może udzielić lepszych podpowiedzi co do tego, jakie funkcje i właściwości są dostępne w danym kontekście, co znacznie zwiększa produktywność programisty.
3. Czysty kod - NestJS pozwala na pisanie modularnego i czystego kodu, dzięki czemu aplikacja jest łatwiejsza do utrzymania i rozwijania w dłuższej perspektywie czasowej.
4. Rozszerzalność - NestJS oferuje wiele wbudowanych funkcjonalności i dodatków, co znacznie ułatwia rozwój aplikacji. Ponadto, dzięki architekturze opartej na modułach, aplikacje można łatwo rozbudowywać i dostosowywać do potrzeb użytkownika.
5. Popularność - NestJS zyskuje na popularności wśród programistów dzięki swojej wydajności, łatwości użycia i modułowości. Dzięki temu możemy spodziewać się wielu społecznościowych projektów, dodatków i narzędzi, które pomogą w tworzeniu aplikacji.

Dziękujemy za uwagę