NestJS Dasar

Nur Muhamad Ash Shidiqi

Kenalan Dulu 👋

- Nur Muhamad Ash Shidiqi (Read: **Diqi**)
- Husband, father, and sofware engineer
- 5+ years experiences
- Man behind Brain Dee Tech and Brain Dee Code

Get in Touch

- Instagram: @nurmuhamadas | @braindeecode | @braindeetech
- Facebook: Nur Muhamad Ash Shidiqi | Brain Dee Code | Brain Dee Tech
- Linkedin: Nur Muhamad Ash Shidiqi | Brain Dee Code | Brain Dee Tech
- YouTube: https://youtube.com/c/BrainDeeCode
- TikTok: https://www.tiktok.com/@braindeecode
- Email: braindeecode@gmail.com | braindtechid@gmail.com

Requirements

- JavaScript
- NodeJS
- TypeScript

🚀 Pengenalan NestJS

Apa itu NestJS?

- **Framework Node.js** progresif untuk membangun aplikasi sisi server (backend) yang efisien dan skalabel.
- Dibangun dengan dan sepenuhnya mendukung **TypeScript**.
- Tujuannya: Menyediakan **arsitektur aplikasi "out-of-the-box"** yang solid, memungkinkan developer fokus pada logika bisnis.
- Dibalik layar, NestJS menggunakan library populer seperti Express JS untuk HTTP handler, Winston untuk logging, dll

Kenapa NestJS Dibuat?

- **Masalah:** Ekosistem Node.js (seperti Express) memberikan kebebasan mutlak, yang seringkali berujung pada arsitektur yang tidak konsisten.
- **Solusi NestJS:**
  - Menyediakan **struktur standar** yang jelas (Modules, Controllers, Providers).
  - Mendorong **prinsip desain yang solid** seperti SOLID.
  - Memudahkan **skalabilitas dan maintenance** proyek dalam jangka panjang.

Keunggulan Utama NestJS

- **Arsitektur Terstruktur:** Kode lebih rapi, mudah dipahami, dan mudah dikelola.
- **Berbasis TypeScript:** Keamanan tipe data (Type Safety), auto-completion, dan mengurangi bug saat runtime.
- **Sangat Modular:** Aplikasi dipecah menjadi modul-modul yang reusable.
- **Dependency Injection (DI) Bawaan:** Membuat kode lebih mudah diuji (*testable*) dan tidak saling terikat erat (*loosely coupled*).
- **Ekosistem yang Kuat:** Dokumentasi lengkap dan integrasi mudah dengan berbagai teknologi (database, GraphQL, WebSocket, dll).

🛠️ NestJS CLI

Apa itu NestJS CLI?

- **CLI (Command Line Interface)** adalah alat bantu berbasis teks di terminal.
- NestJS CLI berfungsi untuk **mempercepat dan menstandarisasi** proses pengembangan aplikasi NestJS.
- **Fungsi Utama:**
  - Membuat proyek baru.
  - Scaffolding*: Membuat file-file dasar (module, controller, service) secara otomatis.
  - Menjalankan aplikasi untuk development.
  - Membangun aplikasi untuk produksi.

Praktik: Instalasi NestJS CLI

- Kita akan menginstal NestJS CLI secara **global** di komputer kita agar bisa digunakan di mana saja.
- Buka terminal dan jalankan perintah berikut:

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

- Library: https://github.com/nestjs/nest-cli
- Dokumentasi lengkap: https://docs.nestjs.com/cli/overview

🛠️ Inisialisasi Proyek

Membuat Proyek Pertama

- Kita bisa membuat proyek NestJS pertama kita menggunakan CLI yang sudah kita install sebelumnya.
- Perintah ini akan membuat folder baru, meng-install semua dependensi yang dibutuhkan, dan menyiapkan struktur proyek dasar.

```
nest new nama-proyek
```

📂 Struktur Folder

Membedah Struktur Folder Awal

- NestJS CLI membuatkan kita struktur folder yang standar dan rapi.
- `src/` : Folder utama tempat semua kode aplikasi kita berada.
- `main.ts` : Titik masuk aplikasi (entry point). Di sinilah aplikasi NestJS kita di-bootstrap dan dijalankan.
- `app.module.ts` : Modul utama (root module) dari aplikasi kita.
- `app.controller.ts` : Sebuah contoh controller untuk menangani request HTTP.
- `app.controller.spec.ts` : File testing untuk controller.

Membedah Struktur Folder Awal

- `app.service.ts` : Sebuah contoh service yang berisi logika bisnis sederhana.
- `test/` : Folder untuk menyimpan file testing End-to-End.

🎨 Decorator

Apa itu Decorator?

- Decorator adalah fitur TypeScript yang diawali dengan simbol `@`.
- Fungsi decorator adalah untuk "menghias" atau "menandai" sebuah class, method, atau properti untuk memberinya fungsi atau metadata tambahan.
- Analogi: Seperti memberi stiker pada kode kita untuk memberitahu NestJS, "Hei, class ini adalah sebuah Controller!" atau "Method ini untuk handle request GET!".
- Penjelasan selengkapnya: https://www.typescriptlang.org/docs/handbook/decorators.html

Contoh Decorator di NestJS

- `@Module()`: Menandai sebuah class sebagai Modul.
- `@Controller()`: Menandai sebuah class sebagai Controller.
- `@Injectable()`: Menandai sebuah class (seperti Service) agar bisa di-inject atau disediakan melalui Dependency Injection.
- `@Get()`, `@Post()`: Menandai method sebagai handler untuk request HTTP GET, POST, dll.
- `@Param()`, `@Body()`: Menandai parameter di dalam method untuk mengekstrak data dari URL atau body request.

📦 Modules

Apa itu Module?

- Sebuah class yang ditandai dengan decorator @Module().
- **Analogi**: Anggap saja seperti sebuah "kotak" atau "wadah" untuk mengelompokkan bagian-bagian aplikasi yang saling berhubungan.
- **Tugas Utama**:
  - Mengorganisir Controllers dan Providers.
  - Mengelola dependensi dan visibilitas antar bagian aplikasi.
- Dokumentasi lengkap: https://docs.nestjs.com/modules

Struktur Decorator `@Module`

`@Module()` menerima sebuah objek dengan beberapa properti penting:

- `providers: []` => Tempat kita mendaftarkan semua Service (atau Provider lain). NestJS perlu tahu tentang Service ini agar bisa menyediakannya nanti.
- `controllers: []` => Tempat kita mendaftarkan semua Controller yang menjadi bagian dari modul ini.
- `imports: []` => Tempat kita mengimpor modul lain yang providers-nya ingin kita gunakan.
- `exports: []` => Kebalikan dari imports. Jika kita ingin providers dari modul ini bisa digunakan oleh modul lain, kita harus 'mengekspos'-nya di sini.

Cara Membuat Module

- Kita bisa membuat module dengan cara menambahkan decorator `@Module()` pada class
- Lalu, kita import module tersebut ke dalam `AppModule` agar bisa digunakan oleh aplikasi
- Atau, kita juga bisa membuat module dengan menggunakan Nest CLI:

```
nest g module nama-module
```

- Nest akan membuatkan file module dalam folder `nama-module` dan otomatis meregistrasikannya ke dalam `AppModule`

Praktik: Membuat Books Module

- Kita akan membuat books module dengan Nest CLI

```
nest g module books
```

Controllers

Apa itu Controller?

- Sebuah *class* yang ditandai dengan decorator `@Controller()`.
- **Tugas Utama:** Menerima *request* HTTP yang masuk dan mengembalikannya dengan sebuah *response*.
- Controller bertindak sebagai **"Manajer Lalu Lintas"** yang menghubungkan dunia luar (klien) dengan logika bisnis di dalam aplikasi kita (Service).
- Dokumentasi lengkap: https://docs.nestjs.com/controllers

Cara Membuat Controller

- Untuk membuat controller, kita bisa membuat class dan menandainya dengan decorator `@controller`

```
@Controller() // <-- Menandai class ini sebagai Controller
export class AppController {
  // ...
}
```

- Agar controller dapat digunakan, kita perlu meregistrasikannya pada module

Cara Membuat Controller

- Kita juga bisa membuat controller secara otomatis dengan menggunakan Nest CLI:

```
nest g controller nama-controller
```

- Ketika kita membuat controller dengan menggunakan Nest CLI, kita akan dibuatkan file controller dan file testing nya sekaligus di dalam folder `nama-controller`.
- Nest juga otomatis meregistrasikan controller yang dibuat ke dalam module

Praktik: Membuat Controller

- Kita akan membuat books controller dengan menggunakan Nest CLI:

```
nest g controller books
```

Routing Dasar

- Untuk membuat routing di NestJS sangat sederhana, kita hanya perlu memberikan parameter di dalam decorator `@controller("/nama-router")`. Secara otomatis, NestJS akan membuatkan kita routing untuk path `/nama-router`
- Jika kita menggunakan Nest CLI, kita sudah dibuat router sesuai dengan nama controller yang kita buat

```
// books/books.controller.ts

@Controller("books") // Dibuatkan routing untuk path `/books`
export class BooksController {
  // ...
}
```

HTTP Method

HTTP Method

- Kita bisa memberi tahu NestJS method mana yang harus dieksekusi untuk menangani request dari user dengan menggunakan decorator
- Decorator yang paling umum:
  - `@Get()` : Untuk menangani HTTP Method GET.
  - `@Post()` : Untuk menangani HTTP Method POST.
  - `@Put()` : Untuk menangani HTTP Method PUT.
  - `@Delete()` : Untuk menangani HTTP Method DELETE.
  - `@Patch()` : Untuk menangani HTTP Method PATCH.

Praktik: Membuat Endpoint Pertama

```
// books/books.controller.ts

@Controller("books")
export class BooksController {
  @Get()
  getBooks(): string {
    return "This action returns all books";
  }
}
```

Sekarang kita bisa mengakses API nya di `localhost:3000/books` menggunakan browser atau postman

Praktik: Membuat Nested Routing

- Kita bisa menambahkan nested routing dengan memberikan parameter ke dalam method decorator

```
// books/books.controller.ts

@Controller("books")
export class BooksController {
  @Get("popular") // <-- hasil akhir routing: `/books/popular`
  getPopularBooks(): string {
    return "This action returns popular books";
  }
}
```

HTTP Request

HTTP Request

- Dalam membuat API, kita biasanya perlu untuk mengakses data yang dikirim oleh klien
- NestJS menyediakan cara yang mudah untuk mendapatkan request dari klien dengan menggunakan decorator `@Req`

```
@Controller("books")
export class BooksController {
  @Post()
  create(@Req() request: Request): string {
    console.log(request.body);
    return "This action adds a new book";
  }
}
```

HTTP Request

- Dalam prakteknya, kita akan jarang menggunakan decorator `@Req`
- NestJS menyediakan decorator khusus untuk mengambil data dari request:
  - `@Param` untuk mengambil data dari parameter URL.
  - `@Query` untuk mengambil data dari query parameter
  - `@Body` untuk mengambil data dari body request.
  - `@Headers` untuk mengambil data dari header request
  - Selengkapnya bisa dilihat di: https://docs.nestjs.com/controllers#request-object

Praktik: HTTP Request (Param)

```
@Get(":id")
getBookById(@Param() params: { id: string }): string {
  return `This method return book with id: ${params.id}`;
}
```

- Atau kita bisa mengambil value spesifik dari Param dengan menambahkan parameter ke decorator

```
@Get(":id")
getBookById(@Param("id") id: string): string {
  return `This action return a book with id: ${id}`;
}
```

Praktik: HTTP Request (Query)

```
@Get("/search")
search(@Query() query: object): string {
  return `This action return books with title: ${query.tilte}`;
}
```

- Atau kita bisa mengambil value spesifik dari Query dengan menambahkan parameter ke decorator

```
@Get("/search")
search(@Query("title") title?: string): string {
  return `This action return books with title: ${tilte}`;
}
```

HTTP Response

HTTP Response

- Secara default, NestJS sangat pintar. Apa pun yang kita return dari method controller akan otomatis dikirim sebagai response.
- `return 'sebuah string'` : Akan dikirim sebagai `text/html`.
- `return { key: 'value' }` : Akan dikirim sebagai `application/json`.
- NestJS juga secara otomatis mengatur status code yang sesuai (misalnya, 200 OK untuk `GET`, 201 Created untuk `POST`, dst).

Kustomisasi Response

- Kita bisa mengatur reponse secara manual jika dibutuhkan
- NestJS menyediakan decorator untuk mengubah response:
  - `@Header` untuk mengubah header response
  - `@HttpCode` untuk mengubah status code response
  - `@Redirect` untuk melakukan redirect

Praktik: HTTP Response (Http Code)

```
@Controller()
export class AppController {
  @HttpCode(204)
  @Get("custom-http-code")
  customHttpCode(): string {
    return "This action has no content";
  }
}
```

Praktik: HTTP Response (Redirect)

```
@Redirect("/books", 301)
@Get("redirect")
redirect() {}
```

- Kadang kita perlu melakukan redirect pada kondisi tertentu. Kita bisa melakukannya dengan mengembalikan `HttpRedirectResponse`

```
@Redirect()
@Get("redirect")
redirect(@Query('redirect_to') redirectTo?: string): HttpRedirectResponse {
  if (redirectTo) {
    return { url: redirectTo, statusCode: 301 };
  }

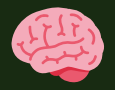
  return { url: '/books', statusCode: 301 };
}
```

Response Object

- NestJS memungkinkan kita mengakses object response dari library dengan menggunakan decorator `@Res`

```
@Get("get-with-res")
getWithRes(@Res() res: Response): void {
   res.status(HttpStatus.OK).json([]);
}
```

- Tapi kita harus hati-hati dalam menggunakannya. Karena jika salah dapat membuat API menjadi error

 Providers & Services

Apa itu Provider?

- **Masalah:** Controller seharusnya tidak berisi logika bisnis yang kompleks (seperti kalkulasi, akses database, atau memanggil API lain). Ini akan membuatnya "gendut" dan sulit diuji.
- **Solusi:** Kita pindahkan semua logika tersebut ke dalam sebuah **Provider**. Ia adalah sebuah *class* sederhana yang ditandai dengan decorator `@Injectable()`.
- **Service** adalah jenis Provider yang paling umum.

```
@Injectable() // <-- Menandai class ini agar bisa disediakan ke class lain
export class AppService {
  // ... Logika bisnis ada di sini
}
```

Kenapa Memisahkan Logika?

- Prinsip utamanya adalah **Separation of Concerns** (Pemisahan Tanggung Jawab).
- **Single Responsibility**: Setiap class punya satu tugas utama.
  - Controller: Menerima request & mengirim response.
  - Service: Menjalankan logika bisnis.
- **Reusability** (Dapat Digunakan Kembali): Service yang sama bisa digunakan oleh beberapa Controller atau Service lain.
- **Testability** (Mudah Diuji): Jauh lebih mudah untuk melakukan unit testing pada sebuah Service yang fokus pada logika, daripada menguji Controller yang terikat dengan HTTP.

Membuat Provider

- Untuk membuat provider, kita hanya perlu membuat class dan menandainya dengan decorator `@Injectable()`

```
@Injectable() // <-- Menandai class ini sebagai provider
export class AppService {
  // ... Logika bisnis ada di sini
}
```

- Agar dapat digunakan oleh class lain, kita perlu meregistrasikannya di module
- Kita juga bisa membuat provider secara otomatis dengan menggunakan Nest CLI:

```
nest g provider nama-provider
```

Membuat Service

- **Service** termasuk salah satu jenis provider
- NestJS CLI menyediakan cara khusus untuk membuat service:

```
nest g service nama-service
```

- NestJS akan membuatkan file service dan file testing dalam folder yang sesuai dan otomatis meregistrasikannya ke dalam module

Praktik: Membuat Service

- Kita akan membuat books service dengan menggunakan Nest CLI:

```
nest g service books
```

- Lalu kita buat method untuk menghandle logic untuk mengambil data buku

```
// books/books.service.ts

@Injectable()
export class BooksService {
  getBooks(): string {
    return "This method return all books from service";
  }
}
```

Praktik: Menggunakan Service

- Dalam controller, sekarang kita bisa menggunakan service dengan cara mendeklarasikannya di constructor

```
// books/books.controller.ts

@Controller("books")
export class BooksController {
  constructor(private readonly booksService: BooksService) {}
  // ...
}
```

🔗 Dependency Injection (DI)

Masalah: Bagaimana Controller & Service Berbicara?

- Kita punya `BooksController` dan `BooksService` yang keduanya sudah terdaftar di `BooksModule`
- Bagaimana cara `BooksController` menggunakan `BooksService` padahal kita tidak pernah membuat intansiasi dari `BooksService` secara manual?:

```
const service = new BooksService();
```

Solusi: Dependency Injection!

- **Definisi Sederhana**: Sebuah pola desain di mana sebuah class menerima dependensinya (misalnya, sebuah service) dari sumber eksternal, alih-alih membuatnya sendiri.
- **Peran NestJS**:
  - NestJS memiliki IoC (Inversion of Control) container. Ini adalah 'pabrik' pintar yang tahu cara membuat dan menyediakan semua providers.
  - Kita cukup "meminta" dependensi yang kita butuhkan melalui constructor class, dan NestJS akan menyediakannya secara ajaib.

```
constructor(private readonly booksService: BooksService) {}
```

Property-based Injection

- Selain menggunakan constructor, kita juga bisa melakukan inject dependensi pada property menambahkan decorator `@Inject()`

```
export class BooksController {
  @Inject()
  private readonly booksService: BooksService;
}
```

Alur Kerja DI

- AppService ditandai dengan @Injectable() agar bisa 'disediakan'.
- AppService didaftarkan di dalam array providers di AppModule.
- AppController "meminta" AppService di dalam constructor-nya.
- Saat aplikasi berjalan, NestJS melihat permintaan tersebut, mencari AppService di dalam "wadah"-nya, dan secara otomatis menyuntikkannya (injects) ke AppController.
- Selesai! Semuanya terhubung secara otomatis, bersih, dan mudah diuji.

Konsep Singleton

- Secara default, semua Provider (seperti `AppService`, `BooksService`, dll) yang kita daftarkan di NestJS adalah **Singleton**.
- Artinya, NestJS hanya membuat **SATU KALI INSTANCE** dari sebuah provider untuk keseluruhan siklus hidup aplikasi.
- Setiap kali dibutuhkan, NestJS akan meng-inject-kan objek yang sama ke semua service yang membutuhkan

📝 Data Transfer Object (DTO)

Masalah: Menerima Data "Mentah"

- Saat kita menerima data dari `@Body()`, secara default tipenya adalah `any`.
- **Ini menimbulkan masalah:**
  - **Tidak ada *Type Safety*:** Kita tidak tahu pasti apa saja properti yang ada. `body.name` bisa jadi `undefined` dan menyebabkan error.
  - **Tidak ada *Auto-completion*:** Editor kode tidak bisa membantu kita karena tidak tahu struktur datanya.
  - **Sulit Divalidasi:** Tidak ada cara standar untuk menerapkan aturan pada data yang masuk.

Solusi: Data Transfer Object (DTO)

- **DTO adalah sebuah `class`** yang kita buat untuk mendefinisikan **struktur data** yang seharusnya dikirim oleh klien.
- **Analogi:** Seperti "cetak biru" atau "kontrak" data yang disetujui antara frontend dan backend.
- **Keuntungan:**
  - **Keamanan Tipe (Type Safety):** Kode kita tahu persis properti apa saja yang ada dan tipe datanya.
  - **Dokumentasi Diri (Self-documenting):** Cukup dengan melihat DTO, developer lain tahu data apa yang dibutuhkan oleh sebuah *endpoint*.
  - **Validasi Terpusat:** (Akan dibahas di modul Validasi) Aturan validasi bisa ditempatkan langsung di dalam DTO.

Praktik: Membuat DTO Pertama

- Kita akan membuat file baru bernama `src/books/dto/create-book.dto.ts`.
- Definisikan `class` dengan properti yang diharapkan.

```
export class CreateBookDto {
  readonly title: string;
  readonly author: string;
  readonly price: number;
  readonly tags: string[];
}
```

- Menggunakan `readonly` adalah praktik yang baik untuk memastikan data tidak diubah di dalam aplikasi.

Praktik: Menggunakan DTO di Controller

- Sekarang kita gunakan DTO yang sudah dibuat di dalam method `create` di `BooksController`

```
@Controller("books")
export class BooksController {
  @Post()
  create(@Body() body: CreateBookDto): string {
    // Sekarang, TypeScript tahu bahwa CreateBookDto
    // memiliki properti name, author, price, dan tags.
    console.log(body);
    return `This action adds a new book with name: ${body.name}`;
  }
}
```

Mengelola Cookie

Apa itu Cookie?

- **Data kecil** yang disimpan di browser klien oleh server.
- **Tujuan:** Untuk menyimpan informasi antar *request*, seperti:
  - Status login (session ID)
  - Preferensi pengguna (misalnya, tema gelap/terang)
  - Isi keranjang belanja
- **Cara Kerja:** Browser akan otomatis mengirimkan kembali *cookie* tersebut ke server setiap kali membuat *request* baru ke domain yang sama.

Setup: Menggunakan `cookie-parser`

- NestJS tidak menangani *parsing cookie* secara default. Kita perlu *library* untuk ini.
- Library standar yang digunakan adalah `cookie-parser`.

Praktik: Instalasi cookie-parser

- Instalasi

```
npm install cookie-parser
npm install -D @types/cookie-parser
```

Praktik: Instalasi cookie-parser

- Registrasi di `main.ts`:

```
import * as cookieParser from "cookie-parser";

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  // Gunakan middleware cookie-parser
  app.use(cookieParser("Screet Key"));

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

Praktik: Membaca Cookie dari Request

- Kita menggunakan decorator `@Req()` untuk mendapatkan akses ke objek *request*.
- Setelah `cookie-parser` terpasang, objek `request` akan memiliki properti `cookies`.

```
export class BooksController {
  @Get("/get-cookie")
  getCookie(@Req() req: Request): any {
    // Akses semua cookie melalui req.cookies
    return { myCookie: req.cookies["my-cookie"] };
  }
}
```

Praktik: Mengirim (Set) Cookie ke Klien

- Kita menggunakan decorator @Res() untuk mendapatkan akses ke objek response.
- Penting: Gunakan opsi { passthrough: true } agar NestJS tetap bisa mengirimkan response body setelah kita memodifikasi response (misalnya, dengan res.cookie()).

```
export class BooksController {
  @Get("/set-cookie")
  setCookie(@Res({ passthrough: true }) res: Response) {
    res.cookie("my-cookie", "ini adalah nilai cookie saya", {
      maxAge: 300000, // Waktu kedaluwarsa dalam milidetik
    });
    return { message: "Cookie berhasil di-set!" };
  }
}
```

⏳ Asynchronous Programming

Asynchronous

- Kita bisa menggunakan asynchronous method pada controller untuk mengembalikan response setelah operasi yang memakan waktu selesai.

```
@Controller()
export class BooksController {
  constructor(private readonly booksService: BooksService) {}

  @Get()
  async getBooks(): Promise<string> {
    console.log("Mulai mengambil data...");
    const data = await this.booksService.getBooks(); // Menunggu service selesai
    console.log("Selesai mengambil data!");
    return data;
  }
}
```

Asynchronous

```
@Injectable()
export class BooksService {
  async getBooks(): Promise<string> {
    return new Promise((resolve) => {
      setTimeout(() => {
        resolve("This method return all books from service");
      }, 3000); // Tunggu 3 detik
    });
  }
}
```

 Request Lifecycle

Request Lifecycle

- Saat sebuah *request* dari klien masuk ke aplikasi NestJS kita, ia tidak langsung menuju ke *Controller*.
- Ia harus melewati serangkaian "pos pemeriksaan" dalam urutan yang sangat spesifik.
- **Analogi:** Anggap seperti alur pemeriksaan keamanan di bandara sebelum kita bisa sampai ke gerbang keberangkatan (Controller).
- **Mengapa ini penting?**
  - Membantu kita menempatkan logika di tempat yang tepat.
  - Memudahkan proses *debugging* saat terjadi masalah.

Diagram Request Lifecycle

Ini adalah urutan standar perjalanan sebuah request:

> Klien ➔ Request Masuk ➔ Middleware ➔ Guards ➔ Interceptor ➔ Pipes ➔ Controller & Service (Route Handler) ➔ Interceptor ➔ Response Keluar ➔ Klien

Tugas Setiap "Pos Pemeriksaan"

- Middleware
  - **Tugas:** Menjalankan kode **sebelum** *route handler* ditentukan. Bisa memodifikasi objek `request` dan `response`.
  - **Contoh:** `cookie-parser`, *logger* untuk setiap *request*.
  - **Ciri Khas:** Belum tahu *route* mana yang akan ditangani.
- Guards
  - **Tugas:** Menentukan apakah sebuah *request* **diizinkan (`true`)** atau **ditolak (`false`)** untuk mengakses *handler* tertentu.
  - **Contoh:** `AuthGuard` yang memeriksa apakah pengguna sudah login.
  - **Ciri Khas:** Fokus pada **izin dan otorisasi**.

Tugas Setiap "Pos Pemeriksaan" (Bagian 2)

- Interceptors
  - **Tugas:** Menjalankan logika **sebelum** dan **sesudah** *handler* dieksekusi. Bisa memodifikasi atau menimpa hasil dari *handler*.
  - **Contoh:** Mengubah format *response* JSON, mencatat durasi eksekusi.
  - **Ciri Khas:** "Mengikat" fungsionalitas tambahan di sekeliling *handler*.
- Pipes
  - **Tugas:** Melakukan **transformasi** (misal: mengubah string "123" menjadi angka 123) atau **validasi** pada argumen yang masuk ke *handler*.
  - **Contoh:** `ValidationPipe`, `ParseIntPipe`.
  - **Ciri Khas:** Fokus pada **data input** yang akan diterima *handler*.

Rangkuman Alur

Setiap komponen memiliki tanggung jawab yang sangat spesifik dan dieksekusi dalam urutan yang dapat diprediksi.

`Middleware` ➔ `Guards` ➔ `Interceptors` ➔ `Pipes` ➔ **`Controller`**

Di video-video selanjutnya, kita akan membahas setiap "pos pemeriksaan" ini satu per satu secara mendalam, dimulai dari **Middleware**.

🔌 Middleware

Apa itu Middleware?

- Sebuah **fungsi** yang dieksekusi **sebelum** *route handler* dipanggil. Ini adalah titik intervensi paling awal dalam *request lifecycle*.
- **Analogi:** "Satpam di gerbang utama" yang memeriksa setiap "tamu" (request) yang datang, sebelum tamu tersebut diarahkan ke ruangan tujuannya (controller).
- **Karakteristik Utama:**
  - Memiliki akses ke objek `request` (`req`) dan `response` (`res`).
  - Memiliki fungsi `next()` untuk meneruskan kontrol ke *middleware* atau *handler* selanjutnya.
  - Bisa menjalankan kode apapun, memodifikasi `req` & `res`, atau bahkan menghentikan siklus *request*.

Cara Membuat Middleware

- Kita bisa membuat middleware dengan membuat class yang implements ke `NestMiddleware` interface
- Atau kita bisa menggunakan Nest CLI:

  ```
  nest g middleware nama-middleware
  ```

  Kita akan dibuatkan file middleware dan file test nya di dalam folder `nama-middleware`
- Kita bisa memanfaatkan dependency injection di middleware jika kita perlu mengakses provider/service

Praktik: Membuat Middleware

- Kita akan membuat middleware sederhana untuk mencatat metode HTTP dan URL dari setiap request yang masuk ke konsol.

```
nest g middleware logger
```

Praktik: Membuat Middleware

- Lalu implementasikan fungsi `use` yang ada dalam 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... ${req.method} ${req.originalUrl}`);
    next(); // <-- Wajib dipanggil agar request diteruskan!
  }
}
```

- **PENTING!** Jika middleware tidak mengembalikan value, kita harus memanggil fungsi `next()`. Jika tidak, maka request tidak akan berhenti (Hang)

Cara Menggunakan Middleware

- Agar middleware yang kita buat berfungsi, kita perlu meregistrasikannya terlebih dahulu `AppModule`
- Akan tetapi, decorator `@module` tidak memiliki parameter bawaan yang menerima middleware
- Untuk meregistrasikannya, kita perlu implementasi `NestModule` di `AppModule`
- Dalam method `configure(MiddlewareConsumer)`, kita bisa meregistrasikan middleware yang kita buat dan menetapkan route mana saja middleware tersebut akan digunakan

Praktik: Menggunakan Middleware

Middleware diterapkan di dalam file **Module** (`app.module.ts`).

```
import { MiddlewareConsumer, Module, NestModule } from '@nestjs/common';
import { LoggerMiddleware } from './logger.middleware';
// ...

@Module({ ... })
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    consumer
      .apply(LoggerMiddleware)
      .forRoutes({
        path: '*', // Terapkan ke semua rute atau .forRoutes('/app'); // Hanya untuk rute /app
              // atau .forRoutes('/api/*'); // Hanya untuk rute yang berawalan API
        requestMethod: RequestMethod.ALL,
      });
  }
}
```

Guards

Apa itu Guard?

- Sebuah *class* yang mengimplementasikan *interface* `CanActivate`.
- **Analogi:** "Penjaga pintu ruangan" yang menentukan apakah seseorang punya izin masuk atau tidak.
- **Fungsi Utama:**
  - Memiliki satu method, `canActivate()`, yang **wajib** mengembalikan nilai `boolean` (atau `Promise<boolean>` / `Observable<boolean>`).
  - Jika `true`: Request diizinkan untuk melanjutkan ke *handler*.
  - Jika `false`: Request ditolak, dan NestJS secara otomatis melempar `403 ForbiddenException`.
- **Tujuan Utama: Authorization** (menentukan apakah pengguna ini *boleh* mengakses resource ini?).

Membuat Guard

- Untuk membuat guard, kita hanya perlu membuat class yang implements ke `CanActivate` interface
- Kita juga bisa menggunakan Nest CLI:

```
nest g guard nama-guard
```

- Kita akan dibuatkan file guard dan file test nya di dalam folder `nama-guard`
- Didalam class guard yang kita buat, kita perlu implementasi method `canActivate(ExecutionContext)` yang akan mengembalikan `boolean`

Praktik: Membuat Guard

- Kita akan membuat guard sederhana untuk memeriksa apakah role dari pengguna adalah `admin`.
- Jika request dari user memiliki *header* `X-ROLE` dengan value `admin`, maka request diizinkan. Sebaliknya, request ditolak.
- Kita gunakan Nest CLI dengan perintah:

```
nest g guard roles
```

Praktik: Membuat Guard

- Lalu, di dalam class `RolesGuard`, kita implementasikan fungsi `canActivate` sebagai berikut:

```
@Injectable()
export class RolesGuard implements CanActivate {
  canActivate(
    context: ExecutionContext
  ): boolean | Promise<boolean> | Observable<boolean> {
    const request = context.switchToHttp().getRequest<Request>();
    const role = request.headers["x-role"];

    return role === "admin";
  }
}
```

Praktik: Menerapkan Guard

- Kita bisa menerapkan Guard menggunakan decorator `@UseGuards()` langsung pada *method* spesifik:

```
@UseGuards(RolesGuard)
@Get('admin')
getAdminBooks(): string {
  return 'This method return admin books';
}
```

- Atau pada level *controller*:

```
@UseGuards(RolesGuard)
export class AdminController {
  // ...
}
```

Global Guards

- Kita bisa menerapkan Guard secara global dengan menggunakan `useGlobalGuards()` method dari NestJS app instance:

```
const app = await NestFactory.create(AppModule);
app.useGlobalGuards(new RolesGuard());
```

Д Pipes

Apa itu Pipe?

- Sebuah *class* yang mengimplementasikan *interface* `PipeTransform`.
- **Analogi:** "Inspektur dan konverter di mulut pipa saluran data" sebelum data masuk ke *controller*.
- **Dua Fungsi Utama:**
    i. **Transformasi:** Mengubah data input dari satu bentuk ke bentuk lain (misalnya, `string` "123" menjadi `number` 123).
    ii. **Validasi:** Memeriksa apakah data input valid. Jika tidak, *pipe* akan melempar *exception*.

Built-in Pipes

- NestJS menyediakan pipes bawaan yang bisa kita gunakan
- Salah satu contohnya adalah `ParseIntPipe` yang berguna untuk memastikan parameter input dari user adalah angka yang valid dan mengubahnya menjadi tipe `number`.

```
@Get(':id')
getBookById(@Param('id', ParseIntPipe) id: number) {
  // ...
}
```

- List selengkapnya ada di: https://docs.nestjs.com/pipes#built-in-pipes

Built-in Pipes

- Sama seperti Guard, kita bisa memberikan parameter berupa instance dari Pipe untuk memberikan options untuk mengubah behavior bawaan dari pipe

```
@Get(':id')
async getBookById(
  @Param('id', new ParseIntPipe({ errorHttpStatusCode: HttpStatus.BAD_REQUEST }))
  id: number,
) {
  // ...
}
```

- Kita bisa menerapkan Pipe di `@Param`, `@Body`, atau `@Query`

Custom Pipes

- Untuk membuat custom pipes, kita hanya perlu membuat class yang implements ke `PipeTransform` interface
- Kita juga bisa menggunakan Nest CLI:

```
nest g pipe nama-pipe
```

- Kita akan dibuatkan file pipe dan file test nya di dalam folder `nama-pipe`
- Didalam class, kita perlu mengimplementasikan method `transform(value: any, metadata: ArgumentMetadata)` yang mengembalikan value hasil transformasi atau throw error jika tidak sesuai

Praktik: Custom Pipes

- Kita akan membuat pipe sederhana yang mengubah nilai string apapun menjadi huruf kecil.

```
nest g pipe lowercase
```

- Lalu, kita implementasi method transform di dalamnya

```
export class LowercasePipe implements PipeTransform {
  // eslint-disable-next-line @typescript-eslint/no-unused-vars
  transform(value: any, metadata: ArgumentMetadata) {
    if (typeof value === "string") {
      return value.toLowerCase();
    }
    return value;
  }
}
```

Praktik: Menerapkan Custom Pipes

- Kita akan menerapkan Pipe tadi pada method `search` di `BooksController` yang sudah kita buat:

```
@Get("/search")
search(@Query("title", LowercasePipe) title: string): string {
  return `This action return books with title: ${title}`;
}
```

Global Pipe

- Kita bisa menerapkan Pipe pada semua parameter dari controller method dengan menggunakan `@UsePipes()` pada method:

```
@UsePipes(LowercasePipe)
@Get("/search")
search(@Query("title") title?: string, @Query("author") author?: string): string {
  return `This action return books with title: ${title}`;
}
```

Global Pipe

- Atau jika kita ingin menerapkan pipe pada semua method, kita bisa menerapkan `@UsePipes()` pada level Controller:

```
@UsePipes(new LowercasePipe())
@Controller()
export class BooksController {
  // ...
}
```

Global Pipe

- Kita juga bisa menerapkan Pipe secara global dengan menggunakan `useGlobalPipes()` method dari NestJS app instance:

```
const app = await NestFactory.create(AppModule);
app.useGlobalPipes(new LowercasePipe());
```

Validation Pipe

- Kita bisa menggunakan pipe sebagai validation untuk memastikan request body dari user memiliki type yang sesuai sebelum kita menjalankan method controller nya
- Untuk memudahkan dalam melakukan validation, kita akan memanfaatkan validation library yang popular di TypeScript yaitu `Zod`

```
npm install zod
```

- Dokumentasi lengkap tentang Zod dapat di akses di: https://zod.dev/

Praktik: Validation Pipe

- Kita akan membuat sebuah validation pipe bernama `ZodValidationPipe` dengan Nest CLI:

```
nest g pipe zod-validation
```

Praktik: Validation Pipe

- Lalu kita implementasikan method transform sebagai berikut:

```
export class ZodValidationPipe implements PipeTransform {
  constructor(private schema: z.ZodType) {}

  transform(value: unknown, metadata: ArgumentMetadata) {
    try {
      const parsedValue = this.schema.parse(value);
      return parsedValue;
    } catch (error) {
      console.log(error);
      throw new HttpException(`Validation Failed`, 400);
    }
  }
}
```

Praktik: Validation Pipes

- Kita akan buat skema object validation menggunakan zod:

```
export const createBookSchema = z
  .object({
    title: z.string(),
    author: z.string(),
    price: z.number(),
    tags: z.array(z.string()).default([]),
  })
  .required();

export type CreateBookDto = z.infer<typeof createBookSchema>;
```

Praktik: Validation Pipes

- Cara menggunakan validation pipes sama dengan cara kita menggunakan pipes biasa
- Kita bisa menggunakannya pada level method, class, atau global

```
@Post()
@UsePipes(new ZodValidationPipe(createBookSchema))
async createBook(@Body() value: CreateBookDto) {
  this.booksService.create(value);
}
```

Exception Filters

Exception Filters

- Sebuah *class* yang mengimplementasikan *interface* `ExceptionFilter`.
- Ditandai dengan decorator `@Catch()`.
  - `@Catch(HttpException)`: Hanya menangkap *error* yang merupakan turunan dari `HttpException`.
  - `@Catch()`: (Kosong) Menangkap semua jenis *error* yang terjadi.
- **Fungsi Utama:** Method `catch(exception, host)` memberi kita kontrol penuh untuk membuat dan mengirim respons *error* kustom.

Built-in Global Exceptions Filters

- NestJS secara default sudah menerapkan global exception filters yang menangani exception dengan type `HttpException` dan kelas turunannya
- Jika ada error yang tidak tertangani di aplikasi dan bukan `HttpException` atau turunanya, NestJS akan mengembalikan default response:

```
{
  "statusCode": 500,
  "message": "Internal server error"
}
```

Praktik: Menggunakan Standard Exception

- Kita bisa menggunakan standar HttpException bawaan dari NestJS:

```
@Get('standard-exception')
getStandardException(): string {
  throw new HttpException('You are not authorized', 403);
}
```

Custom Exception

- Pada banyak kasus, sebenarnya kita bisa menggunakan `HttpException` bawaan dari NestJS
- Tapi, terkadang kita ingin mengkustomisasi exceptionnya sesuai dengan kebutuhan kita dan menggunakannya dibanyak tempat
- Kita membuat exception turunan dari `HttpException`. Dengan begitu, kita bisa menggunakannya di banyak tempat dan bisa langsung ditangani oleh NestJS

```
export class CustomForbiddenException extends HttpException {
  constructor() {
    super("You are not authorized", HttpStatus.FORBIDDEN);
  }
}
```

Custom Exception

- Lalu, kita bisa menggunakannya di dalam program kita:

```
@Get("custom-http-exception")
async getCustomHttpException() {
  throw new CustomForbiddenException();
}
```

Built-in HTTP Exception

- NestJS sudah menyediakan standard exception turunan dari `HttpException` seperti
  - `BadRequestException`
  - `UnauthorizedException`
  - `NotFoundException`
  - `ForbiddenException`
- List selengkapnya dari built-in HTTP Exception: https://docs.nestjs.com/exception-filters#built-in-http-exceptions

Custom Exception Filter

- Mesktipun NestJS sudah menyediakan HTTP Exception bawaan, terkadang kita perlu membuat HTTP Exception dengan format yang kita inginkan
- Kita bisa membuat Exception kita sendiri dengan membuat *class* yang mengimplementasikan *interface* `ExceptionFilter` dan memberikan `@Catch` decorator pada class tersebut
- Kita bisa menentukan jenis error yang ingin di-handle dengan memberikan parameter pada `@Catch(ErrorType)`
- Jika kosong, semua jenis error akan ditangkap oleh exception tersebut

Praktik: Custom Exception Filter

- **Tujuan:** Membuat filter untuk menangkap semua `ZodError` dan mengirimkan respons JSON yang terstruktur.

```
nest g filter zod-validation
```

Praktik: Custom Exception Filter

- Lalu kita buat implementasinya:

```
@Catch(ZodError)
export class ZodValidationFilter implements ExceptionFilter<ZodError> {
  catch(exception: ZodError, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();

    response.status(400).json({
      statusCode: status,
      errors: exception.issues,
    });
  }
}
```

Menerapkan Custom Filter

- Kita bisa menerapkan Filter yang kita buat dengan menggunakan decorator `@UseFilters`
- Sama seperti Guard dan Pipe, kita bisa menerapkannya pada level method atau class controller

Praktik: Menerapkan Custom Filter

- Kita akan coba menerapkan validation yang kita buat pada `BooksController`:

```
@UseFilters(ZodValidationFilter)
@Controller("books")
export class BooksController {
  // ...
}
```

Global Filters

- Sama seperti Guard dan Pipe, kita bisa membuat Filter kita menangani error secara global
- Kita hanya perlu meregistrasikannya pada Nest Application Instance melalui method `useGlobalFilters()`

```
const app = await NestFactory.create(AppModule);
app.useGlobalFilters(new ValidationFilter());
```

✨ Interceptors

Apa itu Interceptor?

- Sebuah *class* yang mengimplementasikan *interface* `NestInterceptor`.
- **Analogi:** Seperti sebuah "bungkus" (*wrapper*) atau "agen pengamat" yang mengelilingi sebuah *route handler*.
- **Kekuatan Utama:**
  - Menjalankan logika **sebelum** *handler* dieksekusi.
  - Menjalankan logika **sesudah** *handler* dieksekusi.
  - **Mengubah (`transform`)** hasil yang dikembalikan oleh *handler*.
  - **Menimpa (`override`)** eksekusi *handler* sepenuhnya (misalnya, dengan mengembalikan data dari *cache*).

Konsep AOP (Aspect-Oriented Programming)

- **AOP** adalah sebuah paradigma pemrograman untuk memisahkan **cross-cutting concerns**.
- **Cross-cutting concerns** adalah logika yang dibutuhkan di banyak tempat dalam aplikasi, tetapi bukan bagian dari logika bisnis inti.
  - Contoh: **Logging**, **Caching**, **Transformasi Data**.
- **Interceptor adalah cara NestJS menerapkan AOP.** Kita bisa "menyisipkan" logika logging atau caching ke banyak *handler* tanpa harus mengubah kode di dalam *handler* itu sendiri.

Membuat Interceptor

- Untuk membuat interceptor, kita hanya perlu membuat *class* yang implement ke `NestInterceptor`
- Kita bisa menggunakan Nest CLI:

```
nest g interceptor nama-interceptor
```

RxJS (Reactive Extensions for JavaScript)

- Rxjs merupakan library javascript yang memungkinkan kita menerapakan reactive programming dalam code kita menggunakan tipe data observable.
- NestJS menggunakan RxJS dalam membuat Interceptor
- Dokumentasi selengkapnya: https://rxjs.dev/guide/overview

Praktik: Membuat Interceptor

- Kita akan membuat interceptor untuk membungkus semua response sukses ke dalam format:

```
{
  "statusCode": 200,
  "data": ...
}
```

- Kita gunakan perintah:

```
nest g interceptor transform
```

Praktik: Membuat Interceptor

- Lalu kita implementasikan method `intercept(context: ExecutionContext, next: CallHandler)`:

```
export class TransformInterceptor<T> implements NestInterceptor<T, any> {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    return next.handle().pipe(
      map((data) => ({
        statusCode: context.switchToHttp().getResponse().statusCode,
        data: data,
      }))
    );
  }
}
```

Praktik: Membuat Interceptor (2)

- Kita akan membuat interceptor untuk mencatat durasi request ke console:

```
nest g interceptor logging
```

```
export class LoggingInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    console.log("Sebelum request...");

    const now = Date.now();
    return next
      .handle()
      .pipe(
        tap(() => console.log(`Setelah request... ${Date.now() - now}ms`))
      );
  }
}
```

Menerapkan Interceptor

- Untuk mengimplementasikan interceptor, kita bisa menggunakan decorator `@UseInterceptors()`
- Sama seperti Guard, Pipe, dan Filter, kita bisa menggunakan interceptor pada level method, class, atau global

Praktik: Menerapkan Interceptor

- Level Global (di main.ts):

```
app.useGlobalInterceptors(new TransformInterceptor());
```

- Di level controller

```
@UseInterceptors(TransformInterceptor)
export class BooksController {
  // ...
}
```

✨ Custom Decorators

Custom Decorator

- Sebelum kita membahas custom decorator, kita akan membuat middleware baru dengan nama `AuthMiddleware` yang berguna untuk menangani autentikasi pengguna

```
@Injectable()
export class AuthMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: () => void) {
    const username = req.headers["x-username"] as string;
    const role = req.headers["x-role"] as string;

    if (!username || !role) {
      res.status(401).send("You are not authorized");
      return;
    }

    (req as ReqWithUser).user = { username, role };
    next();
  }
}
```

Custom Decorator

- Di controller, kita bisa mengakses object user yang sudah kita tambahkan sebelumnya

```
@Get('/profile')
getProfile(@Req() req: Request) {
  return req.user;
}
```

- **Masalah**:
  - **Repetitif**: Kita harus menulis @Req() dan req.user berulang kali.
  - **Kurang Deklaratif**: Kode menjadi kurang bersih dan tujuan utamanya sedikit kabur.

Custom Decorator

- Kita bisa mengatasi permasalahan tersebut dengan menggunakan custom decorator
- NestJS menyediakan fungsi `createParamDecorator()` untuk membuat decorator parameter kita sendiri.
- **Cara Kerja:**
  - Fungsi ini menerima sebuah callback.
  - Callback tersebut akan menerima data (opsional) dan context (konteks eksekusi).
  - Tugas kita di dalam callback adalah mengambil data yang diinginkan dari request dan mengembalikannya.

Praktik: Membuat Custom Decorator

- Kita akan membuat decorator `User()` untuk mengambil data user dari request:

```
nest g decorator user
```

- Lalu kita implementasikan sebagai berikut:

```
interface RequestWithUser extends Request {
  user: UserDto;
}

export const User = createParamDecorator(
  (data: unknown, ctx: ExecutionContext) => {
    // 1. Dapatkan object request dari context
    const request = ctx.switchToHttp().getRequest<RequestWithUser>();
    // 2. Ekstrak dan kembalikan properti user
    return request.user;
  }
);
```

Praktik: Menggunakan Custom Decorator

- Sekarang kita bisa menggunakan decorator @User di controller untuk kode yang jauh lebih bersih.

```
@Get('/profile')
getProfile(@User() user: any) {
  // 'user' di sini adalah hasil dari decorator kita
  // Tidak perlu lagi menulis req.user
  return user;
}
```

🔍 Reflector & Metadata

Masalah: Bagaimana Guard Tahu Aturan Spesifik?

- Sebelumnya kita sudah membuat `RolesGuard` untuk membatasi akses router hanya boleh di akses oleh admin
- Misal sekarang kita ingin `RolesGuard` bisa membatasi akses berdasarkan peran pengguna secara dinamis (misalnya, 'super-admin', 'admin', 'user', dll).
- Bagaimana cara `RolesGuard` tahu bahwa:
  - `getSuperAdmin()` hanya boleh diakses oleh `'super-admin'`?
  - `getAdmin()` hanya boleh diakses oleh `'super-admin'` dan `'admin'`?
  - `getProfile()` boleh diakses oleh `'super-admin'`, `'admin'` dan `'user'`?
- Menulis logika ini secara *hardcode* di dalam *Guard* sangat tidak efisien dan tidak skalabel.

Solusi: Metadata & Reflector

- `@SetMetadata('key', value)`
  - Sebuah *decorator* untuk **"menempelkan" data tambahan (metadata)** ke sebuah *controller* atau *route handler*.
  - Analogi: Memberi label atau catatan khusus pada sebuah method.
- `Reflector`
  - Sebuah *helper class* yang disediakan NestJS untuk **membaca kembali** metadata yang sudah kita tempelkan tadi dari dalam *Guard* atau *Interceptor*.

Alur Kerja Reflector

1. **Di Controller:** Kita menempelkan metadata peran yang dibutuhkan pada sebuah *route handler*.
   `@SetMetadata('roles', ['admin'])`
2. **Di Guard:** Kita meng-inject `Reflector`.
   `constructor(private reflector: Reflector) {}`
3. **Di Guard:** Kita menggunakan `reflector.get()` untuk membaca metadata 'roles' dari *handler* yang sedang diakses, lalu membandingkannya dengan peran pengguna saat ini.

Praktik: Membuat Decorator `@Roles`

- Untuk membuat `@SetMetadata` lebih mudah dibaca, kita bungkus dalam *decorator* kustom.

```
export const ROLES_KEY = "roles";
export const Roles = (...roles: string[]) => SetMetadata(ROLES_KEY, roles);
```

- Kita juga bisa membuat decorator dengan menggunakan `createDecorator()` static method dari class `Reflector`

```
export const OtherRoles = Reflector.createDecorator<string[]>();
```

Praktik: Memperbarui Guard dengan Reflector

```
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const requiredRoles = this.reflector.get<string[]>(
      ROLES_KEY,
      context.getClass()
    );
    if (!requiredRoles) {
      return true; // Jika tidak ada metadata roles, izinkan akses
    }
    const { user } = context.switchToHttp().getRequest();

    return requiredRoles.includes(user.role);
  }
}
```

Praktik: Implementasi `@Roles` di Controller

```
@UseGuards(RolesGuard)
@Roles("super-admin", "admin")
@Controller("admin")
export class AdminController {
  // ...
}
```

⚙ Manajemen Konfigurasi

Masalah: Hardcoding

- Seringkali kita menulis nilai-nilai konfigurasi langsung di dalam kode.
  - `app.listen(3000);`
  - `apiKey: 'abcdef123456'`
- **Ini buruk karena:**
  - **Tidak Aman:** Kredensial sensitif terekspos langsung di *source code*.
  - **Tidak Fleksibel:** Sulit mengubah konfigurasi untuk lingkungan yang berbeda (misalnya, *development* vs. *production*).

Solusi: Environment Variables & `@nestjs/config`

- **Environment Variables (.env):** Praktik standar untuk menyimpan konfigurasi di luar kode aplikasi.
- **`@nestjs/config`:** Modul resmi dari NestJS untuk memuat dan menggunakan variabel dari file `.env` dengan mudah.
  - Di belakang layar, ia menggunakan pustaka populer `dotenv`.

Praktik: Setup `@nestjs/config`

- Kita install library yang dibutuhkan:

```
npm install @nestjs/config
```

- Selanjutnya, kita buat file `.env` di folder utama proyek:

```
PORT=3000
DATABASE=mysql
```

Praktik: Setup `@nestjs/config`

- Impor `ConfigModule` di `app.module.ts`:

```
@Module({
  imports: [
    ConfigModule.forRoot({
      isGlobal: true, // <-- Membuat ConfigService tersedia di seluruh aplikasi
    }),
  ],
  // ...
})
```

Praktik: Implementasi `ConfigService`

- Setelah melakukan setup, kita bisa meng-inject-kan `ConfigService` untuk membaca nilai dari `.env`
- Misalnya di `main.ts`, kita ubah port-nya menjadi value dari `.env`:

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  // Inject ConfigService untuk mendapatkan nilai port
  const configService = app.get(ConfigService);
  const port = configService.get<number>("PORT");

  await app.listen(port);
  console.log(`Aplikasi berjalan di port ${port}`);
}
```

Praktik: Implementasi `ConfigService`

- Kita juga bisa menggunakan `ConfigService` di service.
- Kita akan buat `DatabaseModule` dan `DatabaseService` untuk menangani koneksi ke database:

```
export class DatabaseService {
  constructor(private readonly configService: ConfigService) {}
  getConnection(): string {
    return `Connected to ${this.configService.get<string>(
      "DATABASE"
    )} database.`;
  }
}
```

Lifecycle Events

Apa itu Lifecycle Events?

- "Hooks" yang disediakan NestJS untuk menjalankan logika pada fase-fase spesifik dari siklus hidup aplikasi.
- **Analogi:** Mirip seperti *event listener* `DOMContentLoaded` di web, atau `useEffect` di React.
- **Kegunaan Utama:**
  - Menjalankan proses inisialisasi (misalnya, koneksi ke database, *seeding* data awal).
  - Melakukan proses pembersihan (*cleanup*) sebelum aplikasi mati (misalnya, menutup koneksi database, mengirim log terakhir).

Alur Startup (Penyalaan) Aplikasi

Saat aplikasi NestJS kita dinyalakan, ia akan memicu *event* berikut secara berurutan:

1. `onModuleInit()`
    - Dipicu **setelah** semua dependensi dari sebuah modul berhasil di-*resolve* (terpenuhi).
    - Berguna untuk menjalankan kode yang bergantung pada *provider* lain di dalam modul yang sama.
2. `onApplicationBootstrap()`
    - Dipicu **setelah** semua modul ter-inisialisasi dan aplikasi siap menerima koneksi dari luar.
    - Ini adalah sinyal bahwa "aplikasi sudah siap sepenuhnya".

Alur Shutdown (Pematian) Aplikasi

Untuk mengaktifkan *event* ini, kita **wajib** memanggil `app.enableShutdownHooks()` di `main.ts`.

1. **`onModuleDestroy()`**
    - Dipicu saat Nest akan menghancurkan sebuah modul.
    - Berguna untuk membersihkan *resource* spesifik modul.
2. **`onBeforeApplicationShutdown()`**
    - Dipicu tepat **sebelum** koneksi aplikasi mulai ditutup.
3. **`onApplicationShutdown()`**
    - Dipicu **setelah** semua koneksi ditutup. Ini adalah kesempatan terakhir untuk menjalankan kode sebelum aplikasi benar-benar berhenti.

Praktik: Mengimplementasikan Hooks

Kita akan membuat *AppService* "mendengarkan" semua *event* ini dan mencatatnya ke konsol.

```
export class AppService implements OnModuleInit, OnApplicationBootstrap {
  onModuleInit() {
    console.log("1. AppService (Module) has been initialized.");
  }

  onApplicationBootstrap() {
    console.log("2. Aplikasi telah di-bootstrap sepenuhnya.");
  }

  // ...
}
```

Kapan Menggunakan Setiap Hook?

Hook	Kapan Digunakan?	Contoh Kasus
`onModuleInit`	Setelah dependensi modul siap	Melakukan caching data awal atau verifikasi koneksi internal
`onApplicationBootstrap`	Saat aplikasi siap sepenuhnya	Menjalankan cron job atau mengirim notifikasi "Aplikasi Online"
`onModuleDestroy`	Sebelum sebuah modul dihancurkan	Membersihkan cache atau koneksi spesifik modul
`onApplicationShutdown`	Sebelum aplikasi benar-benar mati	Membersihkan koneksi database, mengirim log terakhir

🌍 Global Modules

Masalah: Impor Berulang

- Sebelumnya kita sudah menggunakan `DatabaseModule` yang menyediakan `DatabaseService`.
- Bayangkan jika untuk menggunakan `DatabaseService` di dalam `UsersModule`, `OrdersModule`, dan `ProductsModule`, kita harus melakukan ini:

```
@Module({
  imports: [DatabaseModule], // <-- Impor di sini
  //...
})
export class UsersModule {}

@Module({
  imports: [DatabaseModule], // <-- Impor lagi di sini
  //...
})
export class OrdersModule {}
```

Solusi: Decorator `@Global()`

- `@Global()` adalah decorator yang ditempatkan di atas @Module().
- **Fungsi**: Membuat semua provider yang ada di dalam array exports dari modul tersebut tersedia secara otomatis di seluruh aplikasi.
- Kita tidak perlu lagi mengimpor modul tersebut di setiap modul lain yang membutuhkannya. Cukup impor sekali di modul utama (AppModule).

Praktik: Membuat Global Module

- Kita akan ubah `DatabaseModule` yang sudah kita buat menjadi global dengan menambahkan decorator `@Global()`

```
@Global()
@Module({
  providers: [DatabaseService],
  exports: [DatabaseService],
})
export class DatabaseModule {}
```

- Dengan begitu, kita bisa mengakses DatabaseService dari mana saja di aplikasi kita tanpa perlu mengimpornya di setiap module

Kapan Sebaiknya Menggunakan `@Global()`?

- ✅ **Gunakan untuk**:
  - Modul utilitas umum yang benar-benar dibutuhkan di banyak tempat.
  - Contoh: Modul Konfigurasi (@nestjs/config), Modul Koneksi Database (TypeOrmModule), Modul Logging.
- ❌ **Hindari untuk**:
  - Modul yang spesifik untuk sebuah fitur bisnis.
  - Contoh: UsersModule, OrdersModule, ProductsModule.
  - Alasan: Menggunakan @Global() secara berlebihan dapat mengaburkan dependensi antar modul dan membuat arsitektur sulit dilacak (mirip seperti bahaya global variable).

🔧 Dynamic Modules

Dynamic Modules

- Di materi-materi sebelumnya, sebenarnya kita sudah pernah melihat penggunaan Dynamic Module seperti `ConfigModule`:

```
@Module({
  imports: [
    // ...
    ConfigModule.forRoot({ isGlobal: true }),
    // ...
  ],
})
export class AppModule {}
```

Dynamic Modules

- **Dynamic Module** adalah sebuah modul yang bisa menerima konfigurasi dan mengembalikan definisi modul (ModuleDefinition) yang sudah disesuaikan.
- **Konvensi**: Kita membuat sebuah metode statis di dalam class modul, yang umumnya dinamai `forRoot()`.
- **Tugas `forRoot(options)`**:
  - Menerima sebuah objek options (konfigurasi).
  - Menggunakan options tersebut untuk membuat provider yang sudah terkonfigurasi.
  - Mengembalikan objek DynamicModule yang berisi provider tersebut, yang kemudian akan digunakan oleh NestJS.

Praktik: Membuat Dynamic Module

- Kita ingin `ValidationModule` yang bersifat dynamic:

```
nest g module validation
```

- Kita akan membuat `ValidationModule` menjadi dynamic module:

```
@Module({})
export class ValidationModule {
  static forRoot(options: { isGlobal: boolean }): DynamicModule {
    return {
      isGlobal: options.isGlobal,
      module: ValidationModule,
      providers: [DatabaseService],
      exports: [DatabaseService],
    };
  }
}
```

Praktik: Membuat Dynamic Module

- Lalu kita buat `ValidationService` nya:

```
nest g service validation
```

-

```
@Module({})
export class ValidationService {
  validate(schema: z.ZodType, data: unknown) {
    const parsedData = schema.parse(data);
    return parsedData;
  }
}
```

Praktik: Menggunakan Dynamic Module

- Kita import module yang sudah kita buat di `AppModule`:

```
@Module({
  imports: [
    //...
    ValidationModule.forRoot({
      isGlobal: true,
    }),
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

🏭 Custom Providers

Provider Standar: Sebuah "Pintasan"

- Saat kita menulis `providers: [BooksService]` di dalam sebuah modul, ini sebenarnya adalah sebuah "pintasan" (*shorthand*).
- **Bentuk lengkapnya adalah seperti ini:**

```
providers: [
  {
    provide: BooksService, // <-- Token
    useClass: BooksService, // <-- Implementasi
  },
];
```

- Setiap provider memiliki sebuah token (`provide`) yang digunakan untuk injeksi, dan sebuah definisi nilai (`useClass`, `useValue`, `useFactory`, dll.).

Jenis-Jenis Custom Provider

- NestJS menyediakan beberapa cara untuk mendefinisikan sebuah provider:
  - `useValue`
    Untuk menyediakan nilai statis (string, angka, objek konfigurasi).
  - `useClass`
    Untuk menggunakan class lain sebagai implementasi dari sebuah token.
  - `useExisting`
    Untuk membuat alias atau menunjuk ke provider lain yang sudah ada.
  - `useFactory`
    Paling powerful. Untuk membuat provider secara dinamis, di mana proses pembuatannya bergantung pada provider lain.

Praktik: Menggunakan `useValue`

- Misal kita mempunyai value / instance dari object `ConfigService` dan kita ingin menggunakannya di Module:

```
@Module({
  providers: [
    {
      provide: CustomConfigService,
      useValue: customConfigService,
    },
  ],
})
export class AppModule {}
```

Praktik: Menggunakan `useValue`

- Kita juga bisa menggunakan `useValue` untuk nilai statis lainnya seperti string atau objek:

```
const configObject = {
  apiKey: "XYZ-123",
  apiUrl: "https://api.example.com",
};

@Module({
  providers: [
    {
      provide: "CONFIG_OPTIONS", // <-- Token berupa string
      useValue: configObject,
    },
  ],
})
export class AppModule {}
```

Praktik: Menggunakan `useValue`

- Lalu kita bisa menggunakannya misal di `CustomConfigService`:

```
@Injectable()
export class CustomConfigService {
  constructor(@Inject("CONFIG_OPTIONS") private options: any) {
    console.log(this.options.apiKey); // -> 'XYZ-123'
  }

  getApiUrl(): string {
    return this.options.apiKey;
  }
}
```

Praktik: Menggunakan `useClass`

- Kita akan ubah kode dari `DatabaseService` untuk memisahkan koneksi dari tiap database menjadi:

```
export abstract class DatabaseService {
  getConnection(): string;
}
```

Praktik: Menggunakan `useClass`

- Kita buat class turunannya:

```
@Injectable()
export class MySQLService extends DatabaseService {
  getConnection(): string {
    return `Connected to MySQL database.`;
  }
}

@Injectable()
export class PostgresService extends DatabaseService {
  getConnection(): string {
    return `Connected to PostgreSQL database.`;
  }
}
```

Praktik: Menggunakan `useClass`

- Kita bisa ubah penggunaan `DatabaseService` di `AppModule`:

```
@Module({
  providers: [
    {
      provide: DatabaseService,
      useClass:
        process.env.DATABASE === "mysql" ? MySQLService : PostgresService,
    },
    ConfigService,
  ],
})
export class AppModule {}
```

Praktik: Menggunakan `useExisting`

- Kita bisa membuat provider dengan nilai provider lain yang sudah ada:

```
@Module({
  providers: [
    {
      provide: "OtherDatabaseService",
      useExisting: DatabaseService,
    },
  ],
})
export class AppModule {}
```

useFactory: Provider Dinamis

- **Kasus Penggunaan**: Saat pembuatan sebuah provider bergantung pada provider lain.
- Contoh: Membuat sebuah `ConnectionProvider` yang konfigurasinya didapat dari `ConfigService`.
- **Struktur `useFactory`**:
  - `factory`: Sebuah fungsi yang akan membuat dan mengembalikan nilai provider.
  - `inject`: Array berisi provider lain (seperti ConfigService) yang dibutuhkan oleh fungsi factory.

Praktik: Menggunakan `useFactory`

- Kita akan ubah `DatabaseService` agar menggunakan `useFactory`

```
export abstract class DatabaseService {
  abstract configService: ConfigService;

  abstract getConnection(): string;
}
```

Praktik: Menggunakan `useFactory`

```
@Injectable()
export class MySQLService extends DatabaseService {
  configService: ConfigService;

  getConnection(): string {
    return `Connected to MySQL database.`;
  }
}

@Injectable()
export class PostgresService extends DatabaseService {
  configService: ConfigService;

  getConnection(): string {
    return `Connected to PostgreSQL database.`;
  }
}
```

Praktik: Menggunakan `useFactory`

- Kita buat factory function untuk `DatabaseService`

```
export const createDatabaseService = (configService: ConfigService) => {
  const database = configService.get<string>("DATABASE");

  switch (database) {
    case "mysql":
      return MySQLService;
    case "postgres":
      return PostgresService;
    default:
      return PostgresService;
  }
};
```

Praktik: Menggunakan `useFactory`

- Lalu kita ubah implementasi di `DatabaseModule`:

```
@Module({
  providers: [
    // ...
    {
      provide: DatabaseService,
      useFactory: createDatabaseService,
      inject: [ConfigService],
    },
    // ...
  ],
})
export class AppModule {}
```

🔗 Module Reference (`ModuleRef`)

Masalah: Ketergantungan Dinamis

- **DI berbasis Constructor** sangat bagus, tapi bagaimana jika kita perlu memilih *service* mana yang akan digunakan berdasarkan sebuah kondisi saat *runtime*?
- **Contoh Skenario:**
  - Sebuah `NotificationService` harus memutuskan apakah akan menggunakan `EmailService` atau `SmsService` berdasarkan preferensi pengguna ('email' atau 'sms') yang didapat dari database.
  - Meng-inject semua kemungkinan *service* di *constructor* menjadi tidak efisien.

Solusi: `ModuleRef`

- `ModuleRef` adalah sebuah *provider* khusus yang bisa di-inject ke dalam *class* Anda.
- **Fungsi:** Memberikan kita akses langsung ke **DI Container** NestJS.
- **Analogi:** Seperti memiliki "kunci utama" untuk membuka seluruh "gudang *provider*" dan mengambil *provider* mana pun yang kita butuhkan, kapan pun kita mau.
- Dengan `ModuleRef`, kita bisa mengambil (*resolve*) sebuah *provider* secara manual berdasarkan **token**-nya.

Praktik: Menggunakan `ModuleRef`

- Kita akan membuat `NotificationService` yang berfungsi mengirimkan notification menggunkan `EmailService` atau `SmsService` berdasarkan preferensi pengguna:

```
@Injectable()
export class NotificationService {
  constructor(private moduleRef: ModuleRef) {}

  sendNotification(message: string, type: "email" | "sms") {
    if (type === "email") {
      const emailService = this.moduleRef.get(EmailService);
      return emailService.sendEmail(message);
    } else {
      const smsService = this.moduleRef.get(SmsService);
      return smsService.sendSms(message);
    }
  }
}
```

Praktik: Menggunakan `ModuleRef`

- ```
  // email/email.service.ts
  @Injectable()
  export class EmailService {
    sendEmail(message: string) {
      return `Email sent: ${message}`;
    }
  }

  // sms/sms.service.ts
  @Injectable()
  export class SmsService {
    sendSms(message: string) {
      return `SMS sent: ${message}`;
    }
  }
  ```
  ```

Praktik: Menggunakan `ModuleRef`

- Kita buat `NotificationController` yang akan menggunakan `NotificationService` untuk mengirimkan notification:

```
@Controller("notification")
export class NotificationController {
  constructor(private notificationService: NotificationService) {}

  @Get("send")
  sendNotification(
    @Query("message") message: string,
    @Query("type") type: "email" | "sms"
  ) {
    this.notificationService.sendNotification(message, type);
  }
}
```

Peringatan: Gunakan dengan Hati-Hati!

- `ModuleRef` adalah fitur tingkat lanjut.
- ✅ **Gunakan jika**:
  - Kita benar-benar membutuhkan resolusi provider yang dinamis.
  - Kita sedang membangun library atau framework yang kompleks di atas NestJS.
- ❌ **Hindari jika**:
  - Tujuan yang sama bisa dicapai dengan DI berbasis constructor biasa.
- **Alasan**: Menggunakan ModuleRef dapat membuat alur dependensi menjadi kurang eksplisit dan lebih sulit dilacak. Constructor-based injection adalah pilihan utama untuk 99% kasus karena lebih jelas dan deklaratif.

✅ Rangkuman & Langkah Selanjutnya

Perjalanan Kita di Modul Fondasi

Selamat! Kita telah menyelesaikan perjalanan mendalam tentang cara kerja internal NestJS. Mari kita lihat kembali apa saja yang telah kita kuasai:

- ✅ **Pilar Utama:** Kita tahu cara kerja `Controller`, `Service`, dan `Module` sebagai inti aplikasi.
- ✅ **Request Lifecycle:** Kita paham alur dari `Middleware` hingga `Interceptor` dan di mana harus menempatkan logika.
- ✅ **Pola Desain:** Kita mengerti `Dependency Injection`, pola `Singleton`, dan berbagai cara mendefinisikan *Providers*.
- ✅ **Arsitektur Lanjutan:** Kita sudah mengenal `Dynamic Modules`, `Global Modules`, dan cara membuat *decorator* kustom.

Memahami "Pikiran" NestJS

Poin terpenting dari modul ini adalah: Kita tidak lagi hanya "menggunakan" NestJS, tapi kita **mengerti cara kerjanya**.

- Kita tahu bagaimana *request* HTTP diproses **langkah demi langkah**.
- Kita tahu bagaimana *Dependency Injection* secara "ajaib" **menghubungkan semua komponen**.
- Kita tahu bagaimana cara **memodifikasi alur** *request* dan *response* di setiap tahap.
- Kita tahu bagaimana cara membangun **modul yang *reusable* dan fleksibel**.

Kita Sekarang Siap Untuk...

Semua pengetahuan ini adalah **fondasi** untuk membangun fitur-fitur nyata yang dibutuhkan dalam aplikasi modern.

Di playlist-playlist selanjutnya, kita akan menggunakan fondasi ini untuk:

- 📚 **Studi Kasus** untuk menerapkan pengetahuan kita di modul NestJS dasar
- 🗄️ **Menyimpan Data Permanen** dengan Database (TypeORM & Prisma).
- ✅ **Memvalidasi Input** secara profesional dengan `ValidationPipe`.
- 📊 **Studi Kasus Database** untuk menerapkan cara bekerja dengan database relasional.
- 🔐 **Mengamankan Aplikasi** dengan Autentikasi & Autorisasi.
- 📂 **Menangani File dan Upload** dengan cara yang aman dan scalable.
- 🧪 **Memastikan Kualitas Kode** dengan Unit & E2E Testing.
- Dst.

Terima Kasih!

Anda telah membangun fondasi yang sangat kuat.

Sampai jumpa di modul selanjutnya!

Happy Coding! 💻