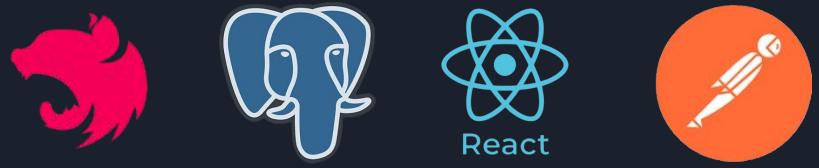
Full Stack Development BootCamp - Day 2

Build Real-World Apps from Backend to Frontend!



Trainer:

Muhammad Maaz Sheikh
Technical Team Lead

Tech Stack Covered:

- Nest Js
- PostgreSQL
- React Js
- Real world project development

Organized By:

Iqra University

Recap Day 1 & Implementation of EndPoint

Table of Content

Section A-2: Nest Js Advanced Features

- ➔ Postman Overview for testing APIs Endpoint
- ➔ Working with Data Transfer Objects (DTOs) and Validation Pipes.
- ➔ Understanding Controllers, Routes, and Request Handlers.
- ➔ Dependency Injection (Services & Providers)
- ➔ Middleware, Interceptors, Guards, and Pipes

Postman Overview for Testing APIs Endpoint

Postman for Testing APIs Endpoint

➔ What is Postman?

Postman is a collaboration platform for API development. It is a popular API client and it enables you to design, build, share, test, and document APIs.

Using the Postman tool, we can send HTTP/s requests to a service, as well as get their responses. By doing this we can make sure that the service is up and running.

Being originally a Chrome browser plugin, Postman now extends their solution with the native version for both Mac and Windows.

Postman for Testing APIs Endpoint

➔ Why Postman?

Postman has become a tool of choice for almost every user.

Free: It is free to download and use for teams of any size.

Easy: Just download it and send your first request in minutes.

APIs Support: You can make any kind of API call (REST, SOAP, or plain HTTP) and easily inspect even the largest responses.

Extensible: You can customize it for your needs with the Postman API.

Integration: You can easily integrate test suites into your preferred CI/CD service with Newman (command line collection runner)

Community & Support: It has a huge community forum

Postman for Testing APIs Endpoint

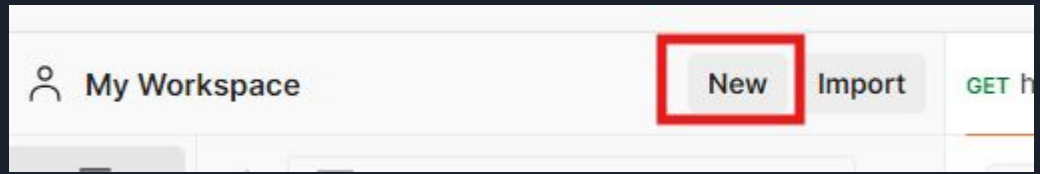
➔ Building blocks of Postman

Before testing an API, first we will see some building blocks of Postman Tool that are essential for every Postman operations.

- Requests
- Collections
- Environment

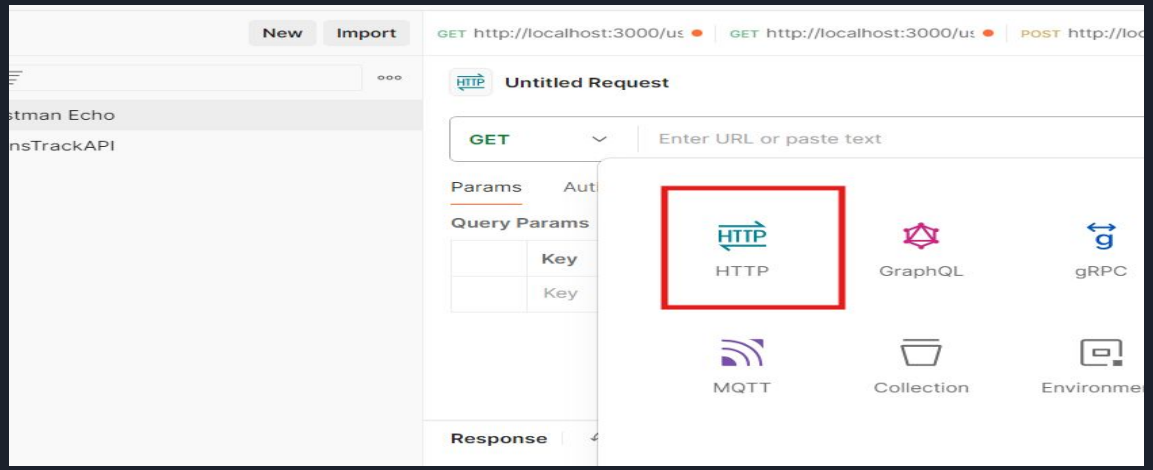
Requests: A request is a combination of the URL, HTTP headers, Body or Payload. In the postman tool, you can save your requests and use them in the future based on your needs.

Click on ***New – Request***



Postman for Testing APIs Endpoint

➔ Building blocks of Postman



You can make requests to APIs in Postman. An API request allows you to retrieve data from a data source, or to send data. APIs run on web servers, and expose endpoints to support the operations client applications use to provide their functionality.

Each API request uses an **HTTP** method.

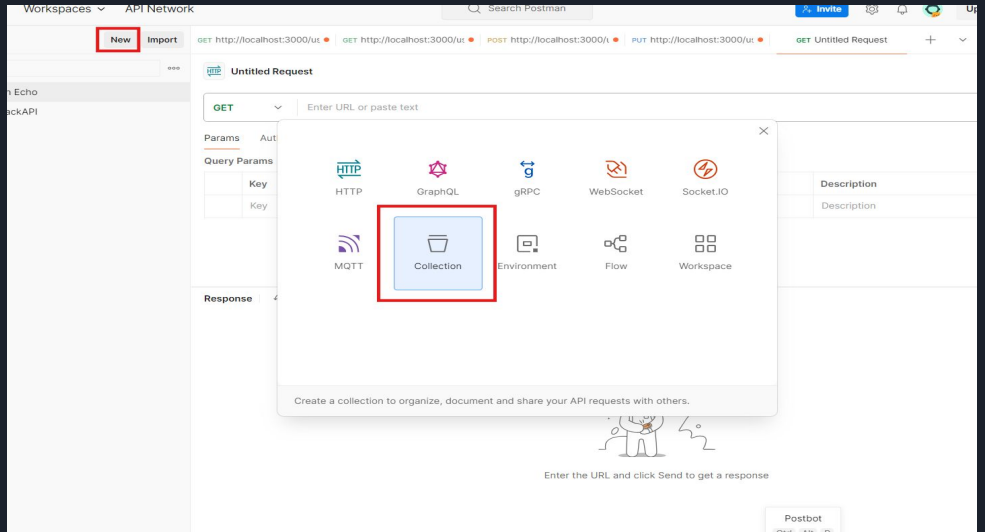
Postman for Testing APIs Endpoint

➔ Building blocks of Postman

Collections: Collections are a group of saved requests you can organize into folders. We can call it as a repository to save our requests.

How To Create Collections in Postman:

Click on ***New – Collection***



Postman for Testing APIs Endpoint

➔ Building blocks of Postman

Environment: Environments in Postman allow us to run requests and collections against different data sets. We could have different environments for Dev, QA & Production. Each of these environments will have different configurations such as URL, token's id and password, API keys etc., Environments are key-value pairs of variables. Each variable name represents its key. So whenever we reference a variable name then it allows us to access its corresponding value.

To create a new environment, we do as follows

Click on ***New – Environment***

Understanding Dtos (Data Transfer Objects)

Understanding Dtos (Data Transfer Objects)

In modern web application development, ensuring that the data passed between the client and server is structured, validated, and secure is paramount. One of the tools that help achieve this is the DTO (Data Transfer Object) design pattern. In NestJS, DTOs are frequently used to define the structure of request and response bodies, while also integrating data validation to ensure that the data meets specific requirements.

What is a DTO (Data Transfer Object)?

A Data Transfer Object (DTO) is a design pattern that is used to transfer data between different layers of an application. Typically, DTOs are used to encapsulate data that is sent over a network, whether it's in an API request or response. DTOs are especially helpful in:

- Defining the structure of the data 📊
- Performing data validation to ensure data integrity ✅
- Preventing issues like over-fetching or under-fetching of data 🔒

Understanding Dtos (Data Transfer Objects)

In NestJS, DTOs are usually classes with decorators that define the structure and validation rules for the data. They ensure that the incoming and outgoing data is well-formed, validated, and serialized properly.

Creating a Simple DTO in Nest JS?

In NestJS, you can create a DTO by defining a class. You can then use decorators provided by libraries like **class-validator** and **class-transformer** to define validation rules and transform the data.

1. **Install Class-Validator:** First, add **class-validator** and **class-transformer** to your project:

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

Understanding Dtos (Data Transfer Objects)

Example 1: Simple DTO for User Creation

Let’s say you are building a user registration API and need a DTO to validate the data the user sends.

```
import { IsString, IsEmail, IsInt, Min, Max } from 'class-validator';

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

    @IsEmail()
    email: string;

    @IsInt()
    @Min(18)
    @Max(120)
    age: number;
}
```

Understanding Dtos (Data Transfer Objects)

Example 1: Simple DTO for User Creation

In this DTO:

- The @**IsString**() decorator ensures that the name property is a string. ✍️
- The @**IsEmail**() decorator ensures that the email property is a valid email address 📧.
- The @**IsInt**(), @**Min**(18), and @**Max**(120) decorators validate that the age is an integer between 18 and 120.

Understanding Dtos (Data Transfer Objects)

Enabling Validation with the Validation Pipes

To use validation, you need to enable the **ValidationPipe** globally in your app. In the **main.ts** file, add the following code to activate the global validation pipe:

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

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

  // Enable global validation
  app.useGlobalPipes(new ValidationPipe());

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

This will ensure that every incoming request is validated according to the rules defined in your DTOs.

Understanding Dtos (Data Transfer Objects)

Using DTOs in Controller

Once the DTO is created and validation is enabled, you can use it in your controllers. For example, you might have a **POST** endpoint to register a new user, which would accept a **CreateUserDto**.

Example 2: Controller Using DTO

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

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

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

The @**Body**() decorator extracts the body of the incoming request and automatically maps it to the CreateUserDto.

Understanding Dtos (Data Transfer Objects)

Using DTOs for Responses

DTOs are not only useful for request validation but also for shaping the data that your application returns in response to the client. This helps prevent exposing unnecessary or sensitive data and ensures that the response format remains consistent.

Example 3: Response DTO for User Data

```
export class UserResponseDto {
  id: number;
  name: string;
  email: string;
}
```

Understanding Dtos (Data Transfer Objects)

Using DTOs for Responses

In your controller, you can use this response DTO to return a user object:

```
@Post()
createUser(@Body() userDto: UserDto) {
  const user = this.userService.createUser(userDto);
  return plainToInstance(UserResponseDto, user, {
    excludeExtraneousValues: true,
  });
}
```

For **plainToInstance** `import { plainToInstance } from 'class-transformer';`

In this case, the **UserResponseDto** ensures that only the id, name, and email properties are included in the response, excluding any other internal data (like passwords).

Understanding Dtos (Data Transfer Objects)

Partial DTOs for Updates

When dealing with updates (such as with a PATCH request), not all fields are required to be provided. NestJS allows you to define optional fields in your DTOs using the @**IsOptional**() decorator.

Example 4: DTO for Updating a User

```
import { IsString, IsEmail, IsOptional } from 'class-validator';

export class UpdateUserDto {
  @IsOptional()
  @IsString()
  name?: string;

  @IsOptional()
  @IsEmail()
  email?: string;
}
```

In this **UpdateUserDto**:
The @**IsOptional**() decorator allows fields to be omitted from the request body. If the client does not provide a name or email, they won't trigger validation errors.

Understanding Dtos (Data Transfer Objects)

Nesting DTOs

DTOs can be nested when you have complex data structures. For example, a user might have an address, which is itself a DTO.

Example 5: Nesting DTOs for User and Address

In nested DTOs the **CreateUserDto** includes an **AddressDto**. The @**Type**() decorator (from **class-transformer**) is used to ensure that the nested **AddressDto** is properly transformed and validated.

Understanding Dtos (Data Transfer Objects)

Nesting DTOs

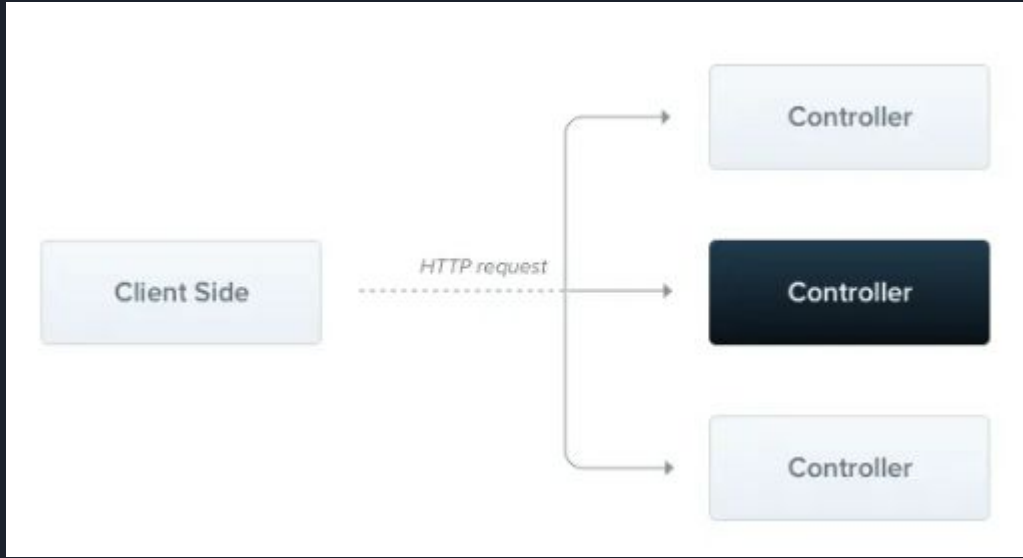
```
export class AddressDto {
  @IsString()
  street: string;

  @IsString()
  city: string;

  @IsString()
  postalCode: string;
}

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

  @IsEmail()
  email: string;

  @IsInt()
  age: number;

  @IsOptional()
  @Type(() => AddressDto)  // Use the Type decorator for nested objects
  address?: AddressDto;
}
```

Understanding Controllers, Routes, and Request

Understanding Controllers, Routes, and Request

Controllers:

Controllers are responsible for handling incoming requests and returning responses to the client:



Understanding Controllers, Routes, and Request

Controllers:

A controller's purpose is to receive specific requests for the application. The *routing* mechanism controls **which controller receives which requests**. Usually, each controller has more than one route, and different routes can perform different actions.

In order to create a basic controller, we use the @Controller() decorator. Decorators associate *classes* with required metadata and enable Nest to create a routing map (tie requests to the corresponding controllers):

Understanding Controllers, Routes, and Request

```
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();
  }
}
```

This is the basic controller you get when you load a new Nest project. The @**Get**() HTTP request method decorator before the getHello() method tells Nest to create a handler for a specific endpoint for HTTP requests. The endpoint corresponds to the HTTP request method (in this case GET) and the route path.

Understanding Controllers, Routes, and Request

Routing:

The next set of decorators connected to routing in the above controller are @Get(), @Post(), Delete(), and @Put(). They tell Nest to create a handler for a specific endpoint for HTTP requests. The above controller creates a following set of endpoints:

- ➔ GET/posts
  Returns all posts
- ➔ GET /posts/{id}
  Returns a post with a given id
- ➔ POST /posts
  Creates a new post
- ➔ PUT /posts/{id}
  Replaces a post with a given id
- ➔ DELETE /posts/{id}
  Removes a post with a given id

Understanding Controllers, Routes, and Request

Request Handling:

When we handle request in the controller, we also need to access the body of a request. By doing so, we can use it to populate our database.
In Nest.js, decorators are used to define metadata for various aspects of your application, including request handling. Let's break down each of the decorators for handling Nest.js Request.

1. @Param(key?: string)

Description: This decorator is used to extract parameters from the request URL.

Usage: It can be applied to method parameters in a controller class to capture specific parameters from the URL.

Example:

```
@Get('/param/:id')
 getParam(@Param('id') id: string) {
   return `Param ID: ${id}`;
 }
```

Understanding Controllers, Routes, and Request

2. @Body(key?: string)

Description: Used to extract data from the request body.

Usage: Apply it to a method parameter to receive data from the body of a POST or PUT request. This example expects a JSON object with a key data in the request body.

Example:

```
@Post()
postBody(@Body() body: any) {
  return `Body Data: ${JSON.stringify(body)}`;
}
```

3. @Query(key?: string)

Description: Extracts parameters from the query string in the URL.

Usage: Apply it to a method parameter to capture query parameters.

Example:

```
@Get()
getQuery(@Query() query: any) {
  return `Query Parameter:${JSON.stringify(query)}`;
}
```

Understanding Controllers, Routes, and Request

4. @Headers(name?: string)

Description: Extracts values from the request headers.

Usage: Apply it to a method parameter to get a specific header value.

Example:

```
 @Get('/header/')
getHeaders(@Headers() headers: any) {
  return `Header: ${JSON.stringify(headers)}`;
}
```

Dependency Injection (Services & Providers)

Dependency Injection (Services & Providers)

Services: What Are They and How Do They Work?

In the context of NestJS, services are a type of provider that encapsulates reusable business logic and data access operations. These services are responsible for performing specific tasks such as interacting with databases, making HTTP requests, or executing complex business logic. By centralizing these functionalities within services, NestJS promotes code reusability, modularity, and testability.

Here's a simple example of a service in NestJS:

In this example, **CatsService** is a basic service class with a **findAll**() method that returns an array of cat names. The @**Injectable**() decorator marks it as a provider that can be injected into other classes.

```
// cats.service.ts
import { Injectable } from '@nestjs/common'

@Injectable()
export class CatsService {
  findAll(): string[] {
    return ['Cat 1', 'Cat 2', 'Cat 3'];
  }
}
```

Dependency Injection (Services & Providers)

Dependency Injection: What Are They and How Do They Work?

Dependency Injection (DI) is a design pattern used to manage dependencies between different components of an application. In the context of NestJS, DI allows classes to declare their dependencies in their constructors, and the framework provides these dependencies when instantiating the class. This approach promotes loose coupling, making components easier to maintain, test, and replace.

Let's illustrate DI with a simple example:

```ts
// cats.controller.ts
import { Controller, Get } from '@nestjs/common';
import { CatsService } from './cats.service';

@Controller('cats')
export class CatsController {
  constructor(private readonly catsService: CatsService) {}

  @Get()
  findAll(): string[] {
    return this.catsService.findAll();
  }
}
```

In this example, **CatsController** depends on **CatsService**. The constructor of **CatsController** accepts an instance of **CatsService**, indicating its dependency.

When NestJS creates an instance of **CatsController**, it automatically injects an instance of **CatsService** into it.

Dependency Injection (Services & Providers)

Provider

Registering providers is a crucial step in configuring a **NestJS** application. Providers can be registered within modules using the providers array or decorators such as @**Module**, @**Controller**, or @**Injectable**. Additionally, providers can be scoped at different levels, including module scope, controller scope, or request scope.

Here's how you can register providers within a module:

In this example, **CatsService** is registered as a provider within the **CatsModule**. This makes **CatsService** available for injection into any component declared within the **CatsModule**.

```ts
// cats.module.ts
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Module({
  controllers: [CatsController],
  providers: [CatsService],
})
export class CatsModule {}
```

Middleware, Interceptors, Guards and Pipes

Middleware, Interceptors, Guards and Pipes

1. Middleware

Purpose:

- Middleware is used for pre-processing requests before they reach the route handlers (controllers).
- It works at the framework level (directly with the HTTP request and response objects).

Execution Order:

- Runs before the controller.

Use Case:

- Logging incoming requests.
- Adding or modifying headers in the request.
- Validating API keys.
- Parsing request data or handling CORS.

Middleware, Interceptors, Guards and Pipes

Middleware...

Key Points:

- Middleware does not handle responses after the controller logic.
- It operates globally or for specific routes.

Example:

```
@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: NextFunction) {
    console.log(`[${new Date().toISOString()}] ${req.method} ${req.url}`);
    next();
  }
}
```

Use:

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

Middleware, Interceptors, Guards and Pipes

Middleware...

How it Works:

- @Injectable()
  - Marks this class as injectable so NestJS can manage it using its Dependency Injection (DI) system.
- implements NestMiddleware
  - Ensures this class conforms to the NestJS middleware structure, which must include a use() method.
- use(req, res, next)
  - This is the middleware function.

Parameters:

req: Incoming HTTP request

res: Response object (Response)

next: Callback to pass control to the next middleware or route (**Without `next()`, the request would hang and not move to the next middleware or controller.)**

Middleware, Interceptors, Guards and Pipes

2. Guard

Purpose:

- Guards are used for authorization and access control.
- They decide whether a request is allowed to proceed to the controller.

Execution Order:

- Runs **before the controller** (after middleware)**.**

Use Case:

- Restrict access to routes based on roles (e.g., admin-only routes).
- Enforce authentication using tokens.
- Control access to sensitive endpoints.

Middleware, Interceptors, Guards and Pipes

Guard...

Key Points:

- Guards return true (allow access) or false (deny access).
- Can extract and validate user information (e.g., from tokens).

Example:

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

Middleware, Interceptors, Guards and Pipes

Guard...

How it Works:

- @Injectable()
  - Marks this class as a provider so Nest can inject it where needed
- implements CanActivate
  - Ensures the class has a canActivate() method.
- This method returns true (allow request) or false (deny request).
- canActivate(context: ExecutionContext): boolean
  - This is the core method.
- **context.switchToHttp().getRequest()** gets the raw request object.
  - It checks for an Authorization header

Middleware, Interceptors, Guards and Pipes

3. Interceptor

Purpose:

- Interceptors are used to wrap the entire request-response lifecycle.
- They allow pre-processing of requests and post-processing of responses.

Execution Order:

- Runs **before and after the controller**.

Use Case:

- Transform or modify incoming requests and outgoing responses.
- Add metadata to responses (e.g., timestamps, pagination info).
- Measure the execution time of requests.

Middleware, Interceptors, Guards and Pipes

Interceptor...

Key Points:

- Can modify both the request and the response.
- Useful for logging, caching, and response formatting.

Example:

```
@Injectable()
export class LoggingInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    const start = Date.now();
    return next.handle().pipe(
      tap(() => console.log(`Request completed in ${Date.now() - start}ms`))
    );
  }
}
```

Middleware, Interceptors, Guards and Pipes

Interceptor...

How it Works:

- @**Injectable**()
  - Tells NestJS this class can be injected as a provider.
- Implements NestInterceptor
  - Forces this class to have the intercept() method.

The intercept() Method:

intercept(context: ExecutionContext, next: CallHandler): Observable<any>

Parameters:

- context: Gives access to the execution context (like request/response, route metadata, etc.)

- next: Represents the next step in the request flow. Think of it like: *“Now go to the actual handler.”*

Middleware, Interceptors, Guards and Pipes

Interceptor...

How it Works:

- **Inside the Method:**

```
console.log('Before...');
```

Runs before the request is handled by the controller/method.

```
return next.handle().pipe(tap(() => console.log('After...')));
```

- **next.handle()** executes the actual route handler.
- **.pipe(tap(...))** runs logic after the route handler returns a value.
- t**ap()** comes from rxjs — it's used to peek into the response without modifying it.

Middleware, Interceptors, Guards and Pipes

4. Pipe

Purpose:

- Pipes are used for data **validation** and **transformation**.
- They transform or validate incoming data before it reaches the route handler.

Execution Order:

- Runs **before the controller method** (Parameter-level, route-level, or globally).

Use Case:

- Validate request parameters, query strings, or bodies.
- Convert data types (e.g., string to integer).
- Enforce specific rules for incoming data (e.g., ensuring a field is non-empty).

Middleware, Interceptors, Guards and Pipes

Pipe...

Key Points:

- Works at the parameter or route level.
- Returns transformed data or throws an error if validation fails.

Example:

```
@Injectable()
export class ParseIntPipe implements PipeTransform {
  transform(value: string): number {
    const parsedValue = parseInt(value, 10);
    if (isNaN(parsedValue)) {
      throw new BadRequestException('Validation failed');
    }
    return parsedValue;
  }
}
```

Middleware, Interceptors, Guards and Pipes

Pipe...

How it Works:

- **The transform() Method:**

```
transform(value: any)
```

Receives the raw input (like query params, route params, body, etc.), and does two things:

1. Transforms it into an integer using **parseInt**()
2. Validates that the result is indeed a number

Steps:

- const val = parseInt(value, 10);
- Converts string input to an integer (base 10)
- Throws a 400 Bad Request if input isn't a valid number
- If valid, returns the number for the controller to use