**NestJs**

1. **Getting Started**
2. **Introduction to NestJs**

NestJS is a framework for building efficient, reliable, and scalable server-side applications with Node.js. It's built with TypeScript and heavily inspired by Angular's architecture, which promotes modularity and uses decorators, dependency injection, and strong typing to create robust and maintainable applications.

Key features of NestJS include:

1. **Architecture:** NestJS enforces a structured and modular architecture that is heavily inspired by Angular. This architecture helps in organizing code into modules, controllers, services, etc., making it easier to manage and maintain larger applications. If you prefer a more opinionated and structured approach to development, NestJS can be advantageous.
2. **Modularity:** NestJS encourages a modular structure, allowing developers to organize code into separate modules, each responsible for specific features or functionalities. This makes the codebase more manageable and easier to maintain.
3. **Dependency Injection:** It leverages the concept of dependency injection, making it simpler to manage the components' dependencies and facilitating testing by allowing for easy mocking and substitution of dependencies.
4. **Middleware:** Middleware support enables the creation of reusable components to handle tasks like logging, authentication, error handling, and more, simplifying code and promoting reusability.
5. **Built-in support for TypeScript:** TypeScript is the default language for NestJS, providing strong typing, enhanced developer tooling, and improved code quality through type checking.
6. **Scalability:** NestJS supports scalable architectures and facilitates the development of large-scale applications by offering built-in support for microservices, enabling communication between various components.
7. **Robust HTTP Server:** It comes with a robust HTTP server, based on Express.js, but offers compatibility with other HTTP platforms if needed.
8. **CLI (Command Line Interface):** NestJS provides a powerful CLI tool to generate modules, controllers, services, etc., which speeds up development and ensures consistency across the application.

Reasons to use NestJS:

1. **TypeScript Support:** If you prefer a strongly typed language and enjoy the benefits it offers in terms of catching errors during development, NestJS is an excellent choice.
2. **Scalability:** NestJS provides a solid foundation for scalable applications, especially with its support for microservices architecture.
3. **Maintainability:** Its modular structure, dependency injection, and use of decorators promote clean, maintainable code, which is easier to understand and update.
4. **Community and Ecosystem:** NestJS has an active community and growing ecosystem with various plugins and modules available, helping developers to extend its functionality.
5. **Familiarity for Angular Developers:** Developers experienced with Angular will find NestJS familiar due to its similar architecture, making the learning curve less steep.

Overall, NestJS is a robust framework that combines the power of TypeScript with modern architectural patterns, facilitating the creation of scalable, maintainable, and efficient server-side applications in Node.js.

1. **What is the Nest CLI**

The "Nest CLI" (Command Line Interface) is a powerful tool provided by NestJS to streamline the development process and automate various tasks when creating and managing NestJS applications.

Here are some key functions and features of the Nest CLI:

1. **Project Scaffolding:** The Nest CLI allows developers to quickly generate the basic structure of a NestJS application, including modules, controllers, services, middleware, and more. By using simple commands like **nest new** followed by the project name, it creates a new NestJS project with a predefined directory structure.
2. **Code Generation:** Developers can use commands like **nest generate** or its shorthand **nest g** to create new components within the NestJS application. For instance, it can generate controllers, modules, services, filters, guards, interceptors, and other files with predefined boilerplate code, helping in maintaining a consistent codebase structure.
3. **Running the Application:** It provides commands to start the NestJS application locally for development or testing purposes. The **nest start** command launches the application and monitors changes in the codebase, automatically restarting the server when files are modified, improving the development workflow.
4. **Plugin and Module Management:** The Nest CLI assists in installing, updating, or removing NestJS plugins and dependencies via simple commands like **nest add** or **nest update**. These commands help manage the application's dependencies and integrate additional features seamlessly.
5. **Configuration and Environment Management:** It aids in managing environment variables and configuration files, making it easier to handle different settings for development, testing, and production environments.
6. **Execution of Custom Scripts:** It allows for executing custom scripts and tasks by integrating them into the NestJS application's workflow, enhancing automation and productivity.

The Nest CLI simplifies the development process by providing a set of commands that automate routine tasks, standardize the project structure, and improve the overall development experience for NestJS applications. It's a valuable tool for both beginners and experienced developers working on NestJS projects.

1. **Getting Started**

Une image contenant texte, capture d’écran, diagramme, Police

Description générée automatiquement

* 1. **First Step**

Une image contenant texte, capture d’écran, logiciel, Page web

Description générée automatiquement

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

Une image contenant texte, capture d’écran, logiciel, Page web

Description générée automatiquement

Une image contenant texte, capture d’écran, Police, Page web

Description générée automatiquement

* 1. **Controllers**

**Une image contenant texte, capture d’écran, Police, conception

Description générée automatiquement**

In NestJS, a controller is a fundamental building block that handles incoming requests, processes them, and sends back responses to the client. Controllers are responsible for defining the request handling logic and act as intermediaries between the incoming HTTP requests and the business logic of the application.

Here are some key characteristics and functionalities of controllers in NestJS:

1. **Routing:** Controllers are responsible for defining routes and mapping them to specific endpoints (URL paths) within your application. Using decorators provided by NestJS, such as **@Controller** and **@Get**, **@Post**, **@Put**, **@Delete**, etc., you can define HTTP methods and their corresponding endpoints.

For example:

**Une image contenant texte, capture d’écran, multimédia, logiciel

Description générée automatiquement**

In this example, the **UsersController** defines a GET endpoint at **/users** that returns a message when accessed.

1. **Request Handling:** Each method within a controller is responsible for handling a specific endpoint. These methods are often referred to as handler methods or actions. These actions can perform necessary operations, interact with services, manipulate data, and return the response.
2. **Dependency Injection:** NestJS allows the use of dependency injection within controllers. By using decorators like **@Inject** or **@Injectable**, you can inject services or other dependencies into the controller, making it easier to maintain and test code.
3. **Middleware Integration:** Controllers can incorporate middleware functions to execute code before or after processing the request. These middleware functions can handle tasks like logging, authentication, error handling, and more.
4. **Separation of Concerns:** NestJS encourages a modular structure where controllers handle routing and request handling logic separately from the business logic. This separation of concerns makes code more organized and maintainable.

Controllers in NestJS are a crucial part of creating RESTful APIs or web applications. They provide a clean and organized way to manage incoming requests, define endpoints, and handle various operations associated with those endpoints.

The @Get() HTTP request method decorator before the findAll() method tells Nest to create a handler for a specific endpoint for HTTP requests. The endpoint corresponds to the HTTP request method (GET in this case) and the route path. What is the route path? The route path for a handler is determined by concatenating the (optional) prefix declared for the controller, and any path specified in the method's decorator. Since we've declared a prefix for every route ( cats), and haven't added any path information in the decorator, Nest will map GET /cats requests to this handler. As mentioned, the path includes both the optional controller path prefix **and** any path string declared in the request method decorator. For example, a path prefix of cats combined with the decorator @Get('breed') would produce a route mapping for requests like GET /cats/breed.

In our example above, when a GET request is made to this endpoint, Nest routes the request to our user-defined findAll() method. Note that the method name we choose here is completely arbitrary. We obviously must declare a method to bind the route to, but Nest doesn't attach any significance to the method name chosen.

This method will return a 200 status code and the associated response, which in this case is just a string. Why does that happen? To explain, we'll first introduce the concept that Nest employs two **different** options for manipulating responses:

Une image contenant texte, capture d’écran, Police, document

Description générée automatiquement

Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

Description générée automatiquement

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

**Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

Description générée automatiquement**

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement

**Request payloads**[**#**](https://docs.nestjs.com/controllers#request-payloads)

In NestJS, handling request payloads involves extracting and processing data sent by clients as part of an HTTP request. Request payloads typically contain data from forms, query parameters, JSON bodies, or other sources, and NestJS provides various mechanisms to access and utilize this incoming data.

Here's how NestJS handles different types of request payloads:

1. **Query Parameters:** To access query parameters sent in the URL, you can use the **@Query()** decorator in controller methods. For example:

Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement

1. In this example, the **id** query parameter is accessed within a GET request handler.
2. **Request Body (JSON Payloads):** When working with POST, PUT, or PATCH requests containing JSON payloads in the request body, you can use the **@Body()** decorator to extract data. Here's an example:

Une image contenant texte, Appareils électroniques, capture d’écran, affichage

Description générée automatiquement

1. Here, **CreateUserDto** represents a DTO (Data Transfer Object) defining the structure of the expected request body.
2. **Request Headers:** Accessing request headers is possible using the **@Headers()** decorator. It allows extraction of specific headers from the incoming request.
3. **File Uploads:** For handling file uploads, NestJS can use packages like **multer** or **@nestjs/platform-express** to process multipart/form-data and handle file uploads within controller methods.
4. **Route Parameters:** Route parameters are parts of the URL path that can be captured using dynamic route segments. These parameters can be accessed using the **@Param()** decorator in controller methods.
5. **Validation and Transformation:** NestJS provides built-in validation and transformation mechanisms, often using DTOs with decorators like **@IsString()**, **@IsInt()**, etc., to validate and transform incoming data.

Handling request payloads in NestJS involves using these decorators and mechanisms to access and process data sent by clients as part of their requests. These mechanisms provide a clean and structured way to handle and validate incoming data, enhancing the robustness and maintainability of NestJS applications.

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement

#### Library-specific approach[#](https://docs.nestjs.com/controllers#library-specific-approach)

So far we've discussed the Nest standard way of manipulating responses. The second way of manipulating the response is to use a library-specific [**response object**](https://expressjs.com/en/api.html#res). In order to inject a particular response object, we need to use the @Res() decorator. To show the differences, let's rewrite the CatsController to the following:

Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

Description générée automatiquement

* 1. **Controllers**

**Une image contenant texte, capture d’écran, diagramme, conception

Description générée automatiquement**

In NestJS, a provider is a fundamental concept used to define a class that can be managed by the NestJS dependency injection (DI) container. Providers in NestJS are typically annotated with the **@Injectable()** decorator, which allows them to be injected as dependencies into other classes.

Providers in NestJS can serve various purposes, including:

1. **Services:** They encapsulate business logic, perform data manipulation, interact with databases, and handle various tasks within the application.
2. **Repositories:** In applications following the repository pattern, providers can act as repositories that abstract away the database operations from the rest of the application.
3. **Helpers or Utilities:** Providers can contain helper functions, utilities, or any shared functionality that can be used across different parts of the application.

To create a provider in NestJS, you can define a class and annotate it with **@Injectable()**. For instance:

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

import { Cat } from './interfaces/cat.interface';

@Injectable()

export class CatsService {

private readonly cats: Cat[] = [];

create(cat: Cat) {

this.cats.push(cat);

}

findAll(): Cat[] {

return this.cats;

}

}

Once a class is annotated with **@Injectable()**, it can be injected into other components (controllers, other services, modules, etc.) by specifying it as a dependency in their constructors. NestJS's dependency injection mechanism handles the instantiation and management of these injected dependencies.

import { Controller, Get, Post, Body } from '@nestjs/common';

import { CreateCatDto } from './dto/create-cat.dto';

import { CatsService } from './cats.service';

import { Cat } from './interfaces/cat.interface';

@Controller('cats')

export class CatsController {

constructor(private catsService: CatsService) {}

@Post()

async create(@Body() createCatDto: CreateCatDto) {

this.catsService.create(createCatDto);

}

@Get()

async findAll(): Promise<Cat[]> {

return this.catsService.findAll();

}

}

The CatsService is **injected** through the class constructor. Notice the use of the private syntax. This shorthand allows us to both declare and initialize the catsService member immediately in the same location.

Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, Police, logiciel

Description générée automatiquement

Une image contenant texte, capture d’écran, logiciel, Page web

Description générée automatiquement

1. **Modules**

In NestJS, a module is a fundamental building block used to organize and encapsulate different parts of an application. Modules play a crucial role in structuring the application by grouping related components, such as controllers, services, providers, and other modules, into cohesive units. They help to organize the codebase, promote reusability, and enable better maintainability and scalability of the application.

**Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement**

**Une image contenant texte, capture d’écran, Police, document

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, Page web

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, Page web

Description générée automatiquement**

**Une image contenant texte, capture d’écran, Police, conception

Description générée automatiquement**

**Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, Site web

Description générée automatiquement**

**Une image contenant texte, capture d’écran, Police, logiciel

Description générée automatiquement**

1. **Middleware**

In NestJS, middleware functions are functions that have access to the request and response objects within the application's request-response cycle. These functions can manipulate the request or response objects, execute additional logic, modify incoming requests, or perform operations before passing control to the next middleware or route handler.

**Une image contenant texte, capture d’écran, Police, blanc

Description générée automatiquement**

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

**Une image contenant texte, Appareils électroniques, capture d’écran, affichage

Description générée automatiquement**

**Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

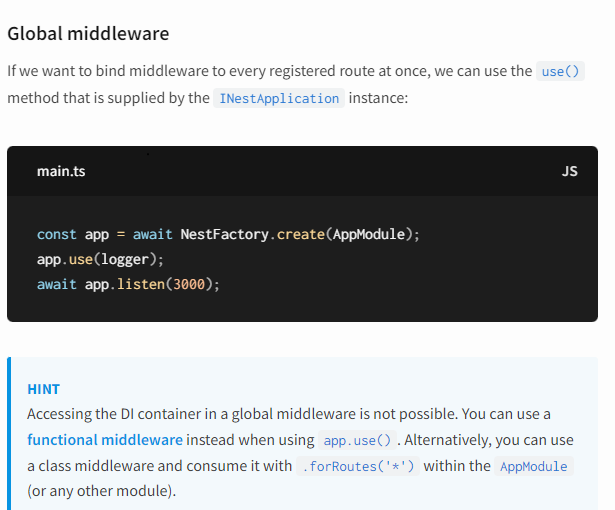
Description générée automatiquement**

**Une image contenant texte, Appareils électroniques, capture d’écran, affichage

Description générée automatiquement**

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

****

1. **Exception Filters**

Nest comes with a built-in **exceptions layer** which is responsible for processing all unhandled exceptions across an application. When an exception is not handled by your application code, it is caught by this layer, which then automatically sends an appropriate user-friendly response.

Une image contenant capture d’écran, texte, conception

Description générée automatiquement

Out of the box, this action is performed by a built-in **global exception filter**, which handles exceptions of type HttpException (and subclasses of it). When an exception is **unrecognized** (is neither HttpException nor a class that inherits from HttpException), the built-in exception filter generates the following default JSON response:

Une image contenant texte, Police, ligne, capture d’écran

Description générée automatiquement

* 1. **Throwing standard exceptions**

Nest provides a built-in HttpException class, exposed from the @nestjs/common package. For typical HTTP REST/GraphQL API based applications, it's best practice to send standard HTTP response objects when certain error conditions occur.

For example, in the CatsController, we have a findAll() method (a GET route handler). Let's assume that this route handler throws an exception for some reason. To demonstrate this, we'll hard-code it as follows:

**Une image contenant texte, multimédia, logiciel, Logiciel multimédia

Description générée automatiquement**

When the client calls this endpoint, the response looks like this:

**Une image contenant Police, capture d’écran, texte

Description générée automatiquement**

The HttpException constructor takes two required arguments which determine the response:

* The response argument defines the JSON response body. It can be a string or an object as described below.
* The status argument defines the [**HTTP status code**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status).

By default, the JSON response body contains two properties:

* statusCode: defaults to the HTTP status code provided in the status argument
* message: a short description of the HTTP error based on the status

To override just the message portion of the JSON response body, supply a string in the response argument. To override the entire JSON response body, pass an object in the response argument. Nest will serialize the object and return it as the JSON response body.

The second constructor argument - status - should be a valid HTTP status code. Best practice is to use the HttpStatus enum imported from @nestjs/common.

There is a **third** constructor argument (optional) - options - that can be used to provide an error [**cause**](https://nodejs.org/en/blog/release/v16.9.0/#error-cause). This cause object is not serialized into the response object, but it can be useful for logging purposes, providing valuable information about the inner error that caused the HttpException to be thrown.

Here's an example overriding the entire response body and providing an error cause:

Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

Description générée automatiquement

* 1. **Custom exceptions**

In many cases, you will not need to write custom exceptions, and can use the built-in Nest HTTP exception, as described in the next section. If you do need to create customized exceptions, it's good practice to create your own **exceptions hierarchy**, where your custom exceptions inherit from the base HttpException class. With this approach, Nest will recognize your exceptions, and automatically take care of the error responses. Let's implement such a custom exception:

**Une image contenant texte, capture d’écran, logiciel, Page web

Description générée automatiquement**

* 1. **Built-in HTTP exceptions**

Nest provides a set of standard exceptions that inherit from the base HttpException. These are exposed from the @nestjs/common package, and represent many of the most common HTTP exceptions:

**Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement**

* 1. **Exception filters**

While the base (built-in) exception filter can automatically handle many cases for you, you may want **full control** over the exceptions layer. For example, you may want to add logging or use a different JSON schema based on some dynamic factors. **Exception filters** are designed for exactly this purpose. They let you control the exact flow of control and the content of the response sent back to the client.

Let's create an exception filter that is responsible for catching exceptions which are an instance of the HttpException class, and implementing custom response logic for them. To do this, we'll need to access the underlying platform Request and Response objects. We'll access the Request object so we can pull out the original url and include that in the logging information. We'll use the Response object to take direct control of the response that is sent, using the response.json() method.

Une image contenant texte, capture d’écran, logiciel, Système d’exploitation

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

The @Catch(HttpException) decorator binds the required metadata to the exception filter, telling Nest that this particular filter is looking for exceptions of type HttpException and nothing else. The @Catch() decorator may take a single parameter, or a comma-separated list. This lets you set up the filter for several types of exceptions at once.

* 1. **Arguments host**

Let's look at the parameters of the catch() method. The exception parameter is the exception object currently being processed. The host parameter is an ArgumentsHost object. ArgumentsHost is a powerful utility object that we'll examine further in the [**execution context chapter**](https://docs.nestjs.com/fundamentals/execution-context)\*. In this code sample, we use it to obtain a reference to the Request and Response objects that are being passed to the original request handler (in the controller where the exception originates). In this code sample, we've used some helper methods on ArgumentsHost to get the desired Request and Response objects. Learn more about ArgumentsHost[**here**](https://docs.nestjs.com/fundamentals/execution-context).

\*The reason for this level of abstraction is that ArgumentsHost functions in all contexts (e.g., the HTTP server context we're working with now, but also Microservices and WebSockets). In the execution context chapter we'll see how we can access the appropriate [**underlying arguments**](https://docs.nestjs.com/fundamentals/execution-context#host-methods) for **any** execution context with the power of ArgumentsHost and its helper functions. This will allow us to write generic exception filters that operate across all contexts.

* 1. **Binding filters**

Let's tie our new HttpExceptionFilter to the CatsController's create() method.

**Une image contenant texte, capture d’écran, Police, logiciel

Description générée automatiquement**

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

We have used the @UseFilters() decorator here. Similar to the @Catch() decorator, it can take a single filter instance, or a comma-separated list of filter instances. Here, we created the instance of HttpExceptionFilter in place. Alternatively, you may pass the class (instead of an instance), leaving responsibility for instantiation to the framework, and enabling **dependency injection**.

**Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement**

In the example above, the HttpExceptionFilter is applied only to the single create() route handler, making it method-scoped. Exception filters can be scoped at different levels: method-scoped of the controller/resolver/gateway, controller-scoped, or global-scoped.  
For example, to set up a filter as controller-scoped, you would do the following:

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

This construction sets up the HttpExceptionFilter for every route handler defined inside the CatsController.

To create a global-scoped filter, you would do the following:

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

Global-scoped filters are used across the whole application, for every controller and every route handler. In terms of dependency injection, global filters registered from outside of any module (with useGlobalFilters() as in the example above) cannot inject dependencies since this is done outside the context of any module. In order to solve this issue, you can register a global-scoped filter **directly from any module** using the following construction:

**Une image contenant texte, capture d’écran, logiciel, Page web

Description générée automatiquement**

* 1. **Catch everything**

**i**n order to catch **every** unhandled exception (regardless of the exception type), leave the @Catch() decorator's parameter list empty, e.g., @Catch().

In the example below we have a code that is platform-agnostic because it uses the [**HTTP adapter**](https://docs.nestjs.com/faq/http-adapter) to deliver the response, and doesn't use any of the platform-specific objects (Request and Response) directly:

**Une image contenant texte, capture d’écran

Description générée automatiquement**

**Une image contenant texte, Police, ligne, capture d’écran

Description générée automatiquement**

* 1. **Inhertance**

Typically, you'll create fully customized exception filters crafted to fulfill your application requirements. However, there might be use-cases when you would like to simply extend the built-in default **global exception filter**, and override the behavior based on certain factors.

In order to delegate exception processing to the base filter, you need to extend BaseExceptionFilter and call the inherited catch() method.

**Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement**

**Une image contenant texte, Police, ligne, capture d’écran

Description générée automatiquement**

The above implementation is just a shell demonstrating the approach. Your implementation of the extended exception filter would include your tailored **business** logic (e.g., handling various conditions).

Global filters **can** extend the base filter. This can be done in either of two ways.

The first method is to inject the HttpAdapter reference when instantiating the custom global filter:

**Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement**

The second method is to use the APP\_FILTER token [**as shown here**](https://docs.nestjs.com/exception-filters#binding-filters).

1. **Pipes**

In NestJS, pipes are a powerful feature that helps with data transformation, validation, and handling incoming parameters in a predictable manner. They provide a way to intercept data before it reaches the route handler in controllers, allowing manipulation, validation, and transformation of incoming data.

Pipes have two typical use cases:

* **transformation**: transform input data to the desired form (e.g., from string to integer)
* **validation**: evaluate input data and if valid, simply pass it through unchanged; otherwise, throw an exception
  1. **Binding Pipes**

**Une image contenant texte, capture d’écran, Police, logiciel

Description générée automatiquement**

**Une image contenant texte, Police, logiciel, capture d’écran

Description générée automatiquement**

* 1. **Custom pipes**

Creating custom pipes in NestJS allows you to implement specific data transformation or validation logic tailored to your application's requirements. Custom pipes enable you to encapsulate reusable logic for transforming or validating data within your NestJS application.

Here's an example of how you can create a custom pipe in NestJS:

**Create a Custom Pipe Class**:

**Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

Description générée automatiquement**

* **EmailValidationPipe**: This custom pipe uses the **validator** library to check if the provided value is a valid email address using the **isEmail** method.
* If the provided value is not a valid email address, it throws a **BadRequestException**.

**Usage of Custom validation Pipe**:

Apply the custom pipe to a route handler in a controller:

**Une image contenant Appareils électroniques, texte, capture d’écran, multimédia

Description générée automatiquement**

* + **@UsePipes(EmailValidationPipe)**: Decorator applied to the **createUser** route handler to use the **EmailValidationPipe** for validating the **email** field in the request body.
  + **@Body('email') email: string**: Specifies that the **email** field from the request body will be validated by the **EmailValidationPipe**.

**Effect of the Custom Pipe**:

When a POST request is made to **/users/create** with a JSON body containing an **email** field, the **EmailValidationPipe** will validate if the provided value is a valid email address. If it's not valid, a **BadRequestException** will be thrown, preventing further execution of the route handler.

This custom pipe showcases how to perform data validation using NestJS. You can create custom pipes for various data validation needs by implementing specific logic within the **transform** method based on your application's requirements.

* 1. **Schema based validation**

**Une image contenant texte, capture d’écran, logiciel, Page web

Description générée automatiquement**

* 1. **Object schema validation**

There are several approaches available for doing object validation in a clean, [**DRY**](https://en.wikipedia.org/wiki/Don%27t_repeat_yourself) way. One common approach is to use **schema-based** validation. Let's go ahead and try that approach.

The [**Zod**](https://zod.dev/) library allows you to create schemas in a straightforward way, with a readable API. Let's build a validation pipe that makes use of Zod-based schemas.

Start by installing the required package:

**Une image contenant texte, capture d’écran, logiciel, Site web

Description générée automatiquement**

* 1. **Binding validation pipes**

**Une image contenant texte, capture d’écran, logiciel, Page web

Description générée automatiquement**

**Une image contenant texte, logiciel, Logiciel multimédia, capture d’écran

Description générée automatiquement**

* 1. **Class-validator**

In NestJS, **class-validator** is a powerful library used for object schema validation. It integrates seamlessly with NestJS and enables you to apply validation rules to your DTOs (Data Transfer Objects) or plain JavaScript/TypeScript objects using decorators.

Here's an example of how to use **class-validator** in a NestJS application:

1. **Installation**:

Start by installing **class-validator** and **class-transformer**:

**Une image contenant texte, Police, capture d’écran

Description générée automatiquement**

1. **Create a DTO with Validation Decorators**:

Define a DTO (Data Transfer Object) with validation decorators from **class-validator**:

**Une image contenant texte, capture d’écran, multimédia, logiciel

Description générée automatiquement**

* **IsNotEmpty()**: Ensures that the **name** property is not empty.
* **IsEmail()**: Validates that the **email** property is a valid email address.

1. **Controller Implementation**:

Use the DTO in your controller with the **ValidationPipe** from NestJS:

**Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement**

* + **ValidationPipe**: Pipe from **@nestjs/common** used for validation and transformation.
  + **@UsePipes**: Decorator applied to the **createUser** route handler to utilize the **ValidationPipe**.
  + **@Body() createUserDto: CreateUserDto**: Using the **CreateUserDto** for automatic validation and transformation of the incoming request body.

1. **Effect**:

When a POST request is made to **/users/create** with a JSON body containing **name** and **email** fields, the **ValidationPipe** will automatically validate the request body according to the rules defined in the **CreateUserDto**. If the validation fails based on the decorators specified in the DTO, it will throw a **BadRequestException**.

**class-validator** provides a wide range of decorators for different validation scenarios, allowing you to define validation rules for properties in your DTOs easily. This helps in ensuring that the incoming data adheres to the specified validation criteria, enhancing the reliability and safety of your application.

* 1. **Global scoped pipes**

In NestJS, global-scoped pipes are used to apply a pipe globally across all routes or controllers within an application. These pipes intercept incoming data before it reaches the route handlers and can be used for tasks such as data transformation, validation, logging, and more.

Here's how you can set up a global-scoped pipe in NestJS:

1. **Create a Custom Pipe**:

Start by creating a custom pipe that implements the **PipeTransform** interface:

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

1. Replace the **/\* validation fails \*/** comment with your actual validation logic. This could include checks based on data type, content, custom rules, etc.
2. **Set up a Global Pipe**:

Configure the pipe to be used globally in your NestJS application. You can do this in the **main.ts** file or the module where your application is bootstrapped (**AppModule** by default):

Une image contenant texte, capture d’écran, logiciel, Système d’exploitation

Description générée automatiquement

* + **app.useGlobalPipes(new ValidationPipe())**: The **ValidationPipe** from NestJS used for automatic validation.
  + **app.useGlobalPipes(new CustomValidationPipe())**: The custom pipe (**CustomValidationPipe**) added as a global pipe.

1. **Effect**:

Once set up globally, the **CustomValidationPipe** will be applied to all incoming requests, regardless of the route or controller. It intercepts the incoming data, applies your custom validation logic, and throws a **BadRequestException** if the validation fails.

By using global-scoped pipes, you can ensure consistent data validation or transformation across your entire application, reducing code duplication and ensuring that certain logic is enforced universally.

Haut du formulaire

* 1. **Providing defaults**

Parse\* pipes expect a parameter's value to be defined. They throw an exception upon receiving null or undefined values. To allow an endpoint to handle missing querystring parameter values, we have to provide a default value to be injected before the Parse\* pipes operate on these values. The DefaultValuePipe serves that purpose. Simply instantiate a DefaultValuePipe in the @Query() decorator before the relevant Parse\* pipe, as shown below:

**Une image contenant capture d’écran, texte

Description générée automatiquement**

1. **Guards**

**Une image contenant texte, capture d’écran, logo, Police

Description générée automatiquement**

A guard is a class annotated with the @Injectable() decorator, which implements the CanActivate interface.

Guards have a **single responsibility**. They determine whether a given request will be handled by the route handler or not, depending on certain conditions (like permissions, roles, ACLs, etc.) present at run-time. This is often referred to as **authorization**. Authorization (and its cousin, **authentication**, with which it usually collaborates) has typically been handled by [**middleware**](https://docs.nestjs.com/middleware) in traditional Express applications. Middleware is a fine choice for authentication, since things like token validation and attaching properties to the request object are not strongly connected with a particular route context (and its metadata).

But middleware, by its nature, is dumb. It doesn't know which handler will be executed after calling the next() function. On the other hand, **Guards** have access to the ExecutionContext instance, and thus know exactly what's going to be executed next. They're designed, much like exception filters, pipes, and interceptors, to let you interpose processing logic at exactly the right point in the request/response cycle, and to do so declaratively. This helps keep your code DRY and declarative.

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

* 1. **Authoriwation guard**

As mentioned, **authorization** is a great use case for Guards because specific routes should be available only when the caller (usually a specific authenticated user) has sufficient permissions. The AuthGuard that we'll build now assumes an authenticated user (and that, therefore, a token is attached to the request headers). It will extract and validate the token, and use the extracted information to determine whether the request can proceed or not.

Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

Description générée automatiquement

**HINT**If you are looking for a real-world example on how to implement an authentication mechanism in your application, visit [**this chapter**](https://docs.nestjs.com/security/authentication). Likewise, for more sophisticated authorization example, check [**this page**](https://docs.nestjs.com/security/authorization).

The logic inside the validateRequest() function can be as simple or sophisticated as needed. The main point of this example is to show how guards fit into the request/response cycle.

Every guard must implement a canActivate() function. This function should return a boolean, indicating whether the current request is allowed or not. It can return the response either synchronously or asynchronously (via a Promise or Observable). Nest uses the return value to control the next action:

* if it returns true, the request will be processed.
* if it returns false, Nest will deny the request.
  1. **Execution context**

The canActivate() function takes a single argument, the ExecutionContext instance. The ExecutionContext inherits from ArgumentsHost. We saw ArgumentsHost previously in the exception filters chapter. In the sample above, we are just using the same helper methods defined on ArgumentsHost that we used earlier, to get a reference to the Request object. You can refer back to the **Arguments host** section of the [**exception filters**](https://docs.nestjs.com/exception-filters#arguments-host) chapter for more on this topic.

By extending ArgumentsHost, ExecutionContext also adds several new helper methods that provide additional details about the current execution process. These details can be helpful in building more generic guards that can work across a broad set of controllers, methods, and execution contexts. Learn more about ExecutionContext[**here**](https://docs.nestjs.com/fundamentals/execution-context).

* 1. **Rôle-based authentication**

Let's build a more functional guard that permits access only to users with a specific role. We'll start with a basic guard template, and build on it in the coming sections. For now, it allows all requests to proceed:

**Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement**

* 1. **Binding guards**

Like pipes and exception filters, guards can be **controller-scoped**, method-scoped, or global-scoped. Below, we set up a controller-scoped guard using the @UseGuards() decorator. This decorator may take a single argument, or a comma-separated list of arguments. This lets you easily apply the appropriate set of guards with one declaration.

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

**HINT**The @UseGuards() decorator is imported from the @nestjs/common package.

Above, we passed the RolesGuard class (instead of an instance), leaving responsibility for instantiation to the framework and enabling dependency injection. As with pipes and exception filters, we can also pass an in-place instance:

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

The construction above attaches the guard to every handler declared by this controller. If we wish the guard to apply only to a single method, we apply the @UseGuards() decorator at the **method level**.

In order to set up a global guard, use the useGlobalGuards() method of the Nest application instance:

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

**Une image contenant texte, capture d’écran, Police, information

Description générée automatiquement**

Global guards are used across the whole application, for every controller and every route handler. In terms of dependency injection, global guards registered from outside of any module (with useGlobalGuards() as in the example above) cannot inject dependencies since this is done outside the context of any module. In order to solve this issue, you can set up a guard directly from any module using the following construction:

**Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

Description générée automatiquement**

* 1. **Setting roles per handler**

Our RolesGuard is working, but it's not very smart yet. We're not yet taking advantage of the most important guard feature - the [**execution context**](https://docs.nestjs.com/fundamentals/execution-context). It doesn't yet know about roles, or which roles are allowed for each handler. The CatsController, for example, could have different permission schemes for different routes. Some might be available only for an admin user, and others could be open for everyone. How can we match roles to routes in a flexible and reusable way?

This is where **custom metadata** comes into play (learn more [**here**](https://docs.nestjs.com/fundamentals/execution-context#reflection-and-metadata)). Nest provides the ability to attach custom **metadata** to route handlers through either decorators created via Reflector#createDecorator static method, or the built-in @SetMetadata() decorator.

For example, let's create a @Roles() decorator using the Reflector#createDecorator method that will attach the metadata to the handler. Reflector is provided out of the box by the framework and exposed from the @nestjs/core package.

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

The Roles decorator here is a function that takes a single argument of type string[].

Now, to use this decorator, we simply annotate the handler with it:

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

Here we've attached the Roles decorator metadata to the create() method, indicating that only users with the admin role should be allowed to access this route.

Alernatively, instead of using the Reflector#createDecorator method, we could use the built-in @SetMetadata() decorator. Learn more about [**here**](https://docs.nestjs.com/fundamentals/execution-context#low-level-approach).

* 1. **Putting it all together**

Let's now go back and tie this together with our RolesGuard. Currently, it simply returns true in all cases, allowing every request to proceed. We want to make the return value conditional based on the comparing the **roles assigned to the current user** to the actual roles required by the current route being processed. In order to access the route's role(s) (custom metadata), we'll use the Reflector helper class again, as follows:

**Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement**

This code is an example of a NestJS guard (**RolesGuard**) that implements the **CanActivate** interface. It is responsible for handling role-based authorization by checking whether the user making the request has the necessary roles to access a specific route or resource.

1. **@Injectable()**:
   * This decorator marks the **RolesGuard** class as injectable, allowing it to receive dependencies via its constructor, such as the **Reflector**.
2. **constructor(private reflector: Reflector) { }**:
   * The constructor injects the **Reflector** instance, which is provided by NestJS and used to access metadata associated with routes and handlers.
3. **canActivate(context: ExecutionContext): boolean { }**:
   * This method is part of the **CanActivate** interface that guards in NestJS must implement. It contains the logic to determine if the request should be allowed or denied access.
4. **const roles = this.reflector.get(Roles, context.getHandler());**:
   * It uses the **Reflector** to retrieve the roles defined for the current route handler. The **Roles** decorator (presumably used to annotate routes or controllers) is used to attach role information to specific routes.
5. **if (!roles) { return true; }**:
   * If no roles are defined for the route (or handler), it allows access by returning **true**.
6. **const request = context.switchToHttp().getRequest();**:
   * This line extracts the current HTTP request object from the execution context.
7. **const user = request.user;**:
   * Assuming that **request.user** holds information about the current user, such as their roles.
8. **return matchRoles(roles, user.roles);**:
   * The **matchRoles** function (not shown in the provided code snippet) is used to compare the roles required for accessing the route (**roles**) with the roles of the current user (**user.roles**).

In summary, this **RolesGuard** is used for role-based authorization in NestJS. It leverages metadata attached to route handlers via the **Roles** decorator to determine if the user making the request possesses the required roles for accessing a specific route. If the user has the necessary roles, access is granted; otherwise, access is denied.

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

Refer to the [**Reflection and metadata**](https://docs.nestjs.com/fundamentals/execution-context#reflection-and-metadata) section of the **Execution context** chapter for more details on utilizing Reflector in a context-sensitive way.

When a user with insufficient privileges requests an endpoint, Nest automatically returns the following response:

**Une image contenant texte, Police, capture d’écran

Description générée automatiquement**

**Une image contenant texte, capture d’écran, Police, Page web

Description générée automatiquement**

1. **Interceptors**

An interceptor is a class annotated with the @Injectable() decorator and implements the NestInterceptor interface.

**Une image contenant texte, capture d’écran

Description générée automatiquement**

Interceptors have a set of useful capabilities which are inspired by the [**Aspect Oriented Programming**](https://en.wikipedia.org/wiki/Aspect-oriented_programming) (AOP) technique. They make it possible to:

* bind extra logic before / after method execution
* transform the result returned from a function
* transform the exception thrown from a function
* extend the basic function behavior
* completely override a function depending on specific conditions (e.g., for caching purposes)
  + 1. **Basics**

Each interceptor implements the intercept() method, which takes two arguments. The first one is the ExecutionContext instance (exactly the same object as for [**guards**](https://docs.nestjs.com/guards)). The ExecutionContext inherits from ArgumentsHost. We saw ArgumentsHost before in the exception filters chapter. There, we saw that it's a wrapper around arguments that have been passed to the original handler, and contains different arguments arrays based on the type of the application. You can refer back to the [**exception filters**](https://docs.nestjs.com/exception-filters#arguments-host) for more on this topic.

* + 1. **Execition context**

By extending ArgumentsHost, ExecutionContext also adds several new helper methods that provide additional details about the current execution process. These details can be helpful in building more generic interceptors that can work across a broad set of controllers, methods, and execution contexts. Learn more about ExecutionContext[**here**](https://docs.nestjs.com/fundamentals/execution-context).

* + 1. **Call handler**

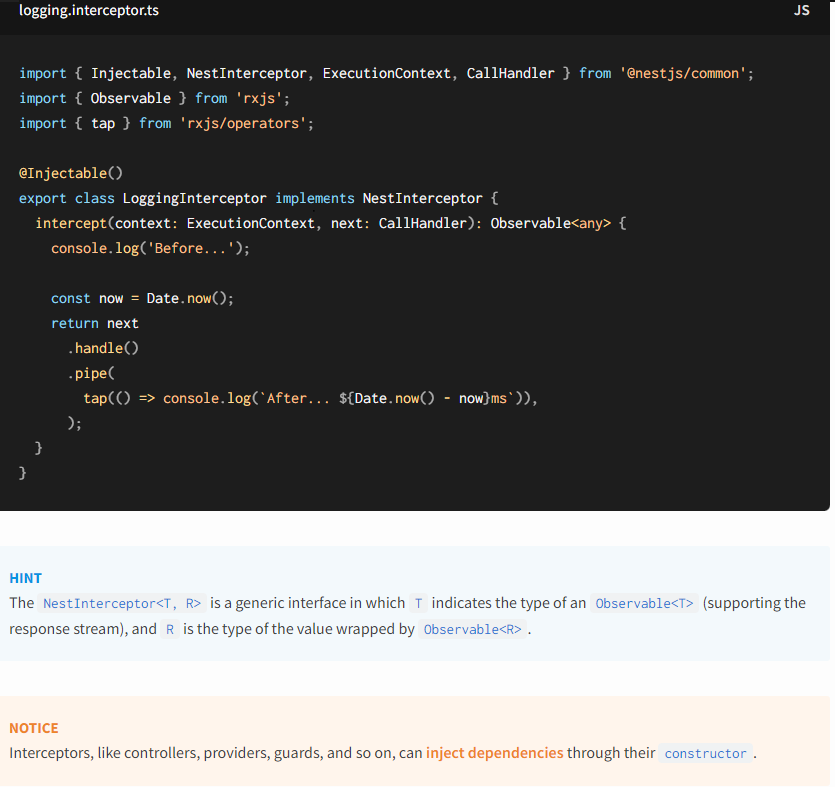
The second argument is a CallHandler. The CallHandler interface implements the handle() method, which you can use to invoke the route handler method at some point in your interceptor. If you don't call the handle() method in your implementation of the intercept() method, the route handler method won't be executed at all.

This approach means that the intercept() method effectively **wraps** the request/response stream. As a result, you may implement custom logic **both before and after** the execution of the final route handler. It's clear that you can write code in your intercept() method that executes **before** calling handle(), but how do you affect what happens afterward? Because the handle() method returns an Observable, we can use powerful [**RxJS**](https://github.com/ReactiveX/rxjs) operators to further manipulate the response. Using Aspect Oriented Programming terminology, the invocation of the route handler (i.e., calling handle()) is called a [**Pointcut**](https://en.wikipedia.org/wiki/Pointcut), indicating that it's the point at which our additional logic is inserted.

Consider, for example, an incoming POST /cats request. This request is destined for the create() handler defined inside the CatsController. If an interceptor which does not call the handle() method is called anywhere along the way, the create() method won't be executed. Once handle() is called (and its Observable has been returned), the create() handler will be triggered. And once the response stream is received via the Observable, additional operations can be performed on the stream, and a final result returned to the caller.

* + 1. **Aspect interception**

The first use case we'll look at is to use an interceptor to log user interaction (e.g., storing user calls, asynchronously dispatching events or calculating a timestamp). We show a simple LoggingInterceptor below:

****

Since handle() returns an RxJS Observable, we have a wide choice of operators we can use to manipulate the stream. In the example above, we used the tap() operator, which invokes our anonymous logging function upon graceful or exceptional termination of the observable stream, but doesn't otherwise interfere with the response cycle.

* + 1. **Binding interceptors**

In order to set up the interceptor, we use the @UseInterceptors() decorator imported from the @nestjs/common package. Like [**pipes**](https://docs.nestjs.com/pipes) and [**guards**](https://docs.nestjs.com/guards), interceptors can be controller-scoped, method-scoped, or global-scoped.

**Une image contenant texte, capture d’écran, logiciel, Page web

Description générée automatiquement**

In order to set up a global interceptor, we use the useGlobalInterceptors() method of the Nest application instance:

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

**Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

Description générée automatiquement**

* + 1. **Response napping**

We already know that handle() returns an Observable. The stream contains the value **returned** from the route handler, and thus we can easily mutate it using RxJS's map() operator.

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

Let's create the TransformInterceptor, which will modify each response in a trivial way to demonstrate the process. It will use RxJS's map() operator to assign the response object to the data property of a newly created object, returning the new object to the client.

**Une image contenant texte, capture d’écran, logiciel, Icône d’ordinateur

Description générée automatiquement**

**Une image contenant texte, capture d’écran, logiciel, Page web

Description générée automatiquement**

* + 1. **Exception mapping**

Another interesting use-case is to take advantage of RxJS's catchError() operator to override thrown exceptions:

**Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement**

* + 1. **Stream overriding**

There are several reasons why we may sometimes want to completely prevent calling the handler and return a different value instead. An obvious example is to implement a cache to improve response time. Let's take a look at a simple **cache interceptor** that returns its response from a cache. In a realistic example, we'd want to consider other factors like TTL, cache invalidation, cache size, etc., but that's beyond the scope of this discussion. Here we'll provide a basic example that demonstrates the main concept.

**Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

Description générée automatiquement**

* + 1. **More operators**

The possibility of manipulating the stream using RxJS operators gives us many capabilities. Let's consider another common use case. Imagine you would like to handle **timeouts** on route requests. When your endpoint doesn't return anything after a period of time, you want to terminate with an error response. The following construction enables this:

**Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

Description générée automatiquement**

* + 1. **Custom route decorators**

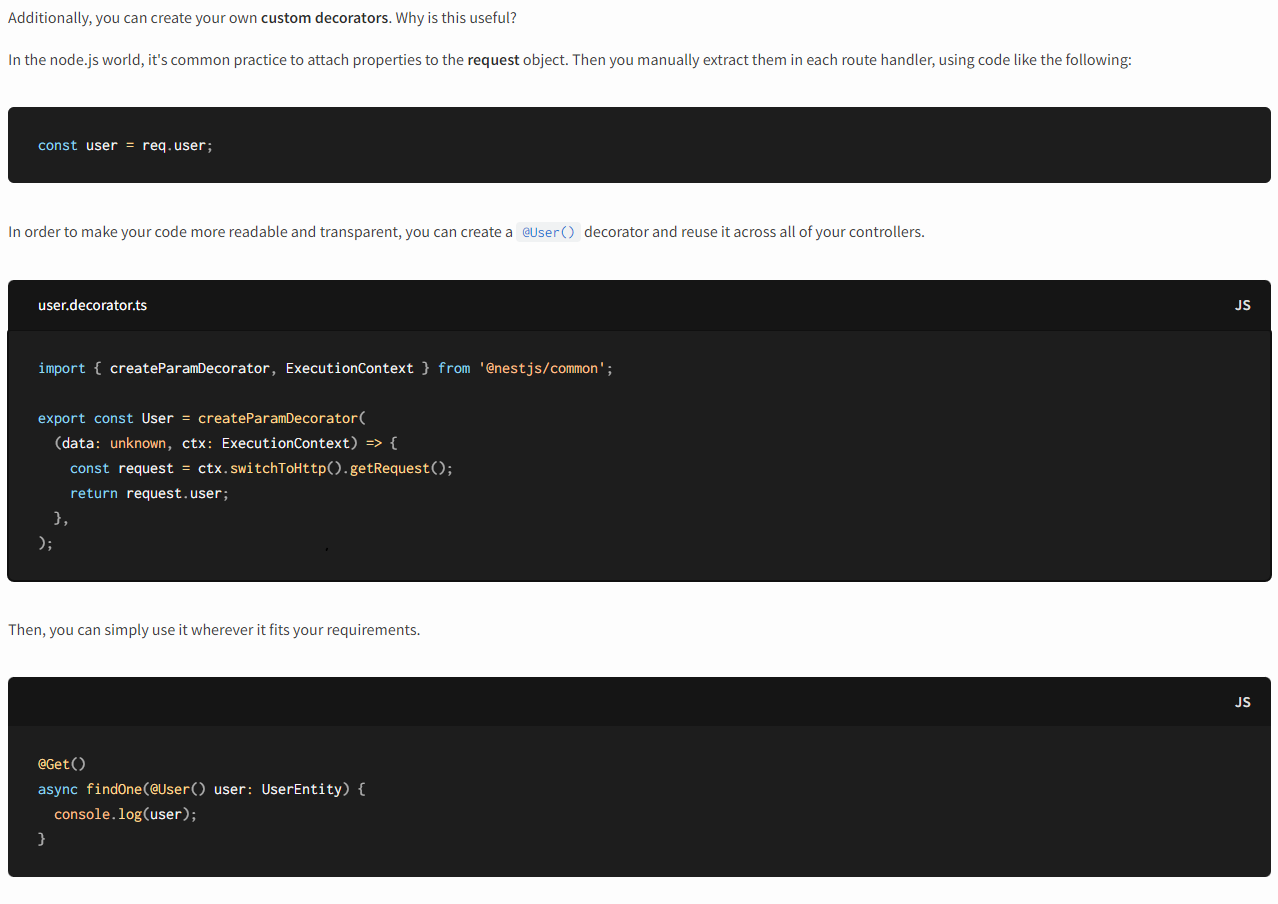
Nest is built around a language feature called **decorators**. Decorators are a well-known concept in a lot of commonly used programming languages, but in the JavaScript world, they're still relatively new. In order to better understand how decorators work, we recommend reading [**this article**](https://medium.com/google-developers/exploring-es7-decorators-76ecb65fb841). Here's a simple definition:

* + - 1. **Param decorators**

Nest provides a set of useful **param decorators** that you can use together with the HTTP route handlers. Below is a list of the provided decorators and the plain Express (or Fastify) objects they represent

**Une image contenant texte, capture d’écran, menu, document

Description générée automatiquement**

****

* + - 1. **Passing data**

When the behavior of your decorator depends on some conditions, you can use the data parameter to pass an argument to the decorator's factory function. One use case for this is a custom decorator that extracts properties from the request object by key. Let's assume, for example, that our [**authentication layer**](https://docs.nestjs.com/techniques/authentication#implementing-passport-strategies) validates requests and attaches a user entity to the request object. The user entity for an authenticated request might look like:

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Let's define a decorator that takes a property name as key, and returns the associated value if it exists (or undefined if it doesn't exist, or if the user object has not been created).

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

Here's how you could then access a particular property via the @User() decorator in the controller:

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

**Une image contenant texte, Police, ligne, capture d’écran

Description générée automatiquement**

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

**Une image contenant texte, Police, capture d’écran

Description générée automatiquement**

* + - 1. **Working with pipes**

Nest treats custom param decorators in the same fashion as the built-in ones (@Body(), @Param() and @Query()). This means that pipes are executed for the custom annotated parameters as well (in our examples, the user argument). Moreover, you can apply the pipe directly to the custom decorator:

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

**Une image contenant texte, Police, ligne, capture d’écran

Description générée automatiquement**

* + - 1. **Decorator composition**

Nest provides a helper method to compose multiple decorators. For example, suppose you want to combine all decorators related to authentication into a single decorator. This could be done with the following construction:

**Une image contenant texte, capture d’écran, Police

Description générée automatiquement**

You can then use this custom @Auth() decorator as follows:

**Une image contenant Police, texte, capture d’écran, typographie

Description générée automatiquement**

This has the effect of applying all four decorators with a single declaration.

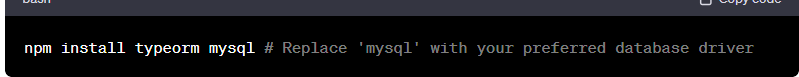
**Une image contenant texte, capture d’écran, Police, ligne

Description générée automatiquement**

1. **Recipes**
2. **TypeOrm**

TypeORM is an Object-Relational Mapping (ORM) library for TypeScript and JavaScript that simplifies database interaction by allowing developers to work with databases using object-oriented programming. It supports various database management systems like MySQL, PostgreSQL, SQLite, and others.

Here's an overview of using TypeORM:



The first step we need to do is to establish the connection with our database using new DataSource().initialize() class imported from the typeorm package. The initialize() function returns a Promise, and therefore we have to create an [**async provider**](https://docs.nestjs.com/fundamentals/async-components).

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement

Then, we need to export these providers to make them **accessible** for the rest of the application.

Une image contenant texte, capture d’écran, logiciel, multimédia

Description générée automatiquement

Now we can inject the DATA\_SOURCE object using @Inject() decorator. Each class that would depend on the DATA\_SOURCE async provider will wait until a Promise is resolved.

#### Repository pattern[#](https://docs.nestjs.com/recipes/sql-typeorm#repository-pattern)

The [**TypeORM**](https://github.com/typeorm/typeorm) supports the repository design pattern, thus each entity has its own Repository. These repositories can be obtained from the database connection.

But firstly, we need at least one entity. We are going to reuse the Photo entity from the official documentation.

Une image contenant texte, capture d’écran, logiciel, Système d’exploitation

Description générée automatiquement

The Photo entity belongs to the photo directory. This directory represents the PhotoModule. Now, let's create a **Repository** provider:

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

Description générée automatiquement

Now we can inject the Repository<Photo> to the PhotoService using the @Inject() decorator:

Une image contenant texte, capture d’écran, affichage, logiciel

Description générée automatiquement

The database connection is **asynchronous**, but Nest makes this process completely invisible for the end-user. The PhotoRepository is waiting for the db connection, and the PhotoService is delayed until repository is ready to use. The entire application can start when each class is instantiated.

Here is a final PhotoModule:

Une image contenant texte, capture d’écran, logiciel

Description générée automatiquement