**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**
   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

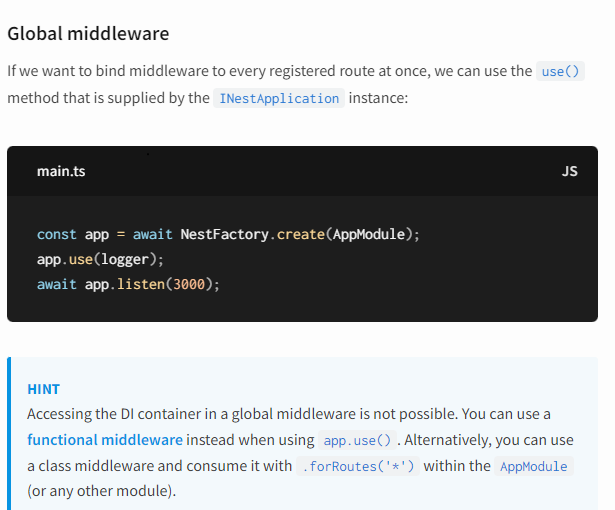
Description générée automatiquement**

**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. **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**.

1. **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**.

1. **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:

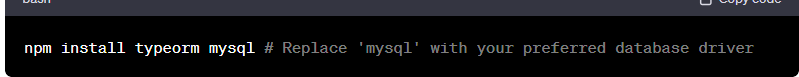
**Une image contenant capture d’écran, texte

Description générée automatiquement**

1. **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" \t "_blank)** 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