**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. **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. **Creating a Rest api application**
   1. **Prerequisite: Install Insomnia [Like Postman]**

Insomnia is an open source desktop application that takes the pain out of interacting with and designing, debugging, and testing APIs. Insomnia combines an easy-to-use interface with advanced functionality like authentication helpers, code generation, and environment variables.

* 1. **Running NestJs in dev mode**

**npm run start:dev**

it restart the server for us for every change

* 1. **Creating a basic controller**
  2. **Nest cli for creating a controller.**

**npx nest generate controller coffes || npx nest g co coffes 🡺** It will crerate for is coffes folder in src and inside we will have coffes controller and a file test for this controller

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

Description générée automatiquement**

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

Description générée automatiquement**

* 1. **Use Route parameters.**

In NestJS, you can use route parameters by using decorators provided by the framework. Route parameters allow you to extract variables from the URL in your NestJS application.

Here's an example of how you can use route parameters:

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

@Controller('coffes')

export class CoffesController {

  @Get(':id')

  findOne(@Param('id') id: string) {

    return `This action returns #${id} coffee`;

  }

}

* 1. **Handling Request Body / Payload**

Handling request body refers to the process in web development where a server-side application receives data sent as part of an HTTP request. This data is typically contained in the body of the request and could be in various formats like JSON, XML, form data, etc.

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

Description générée automatiquement**

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

Description générée automatiquement**

* 1. **Response status codes**

In NestJS, setting the response status code is a crucial part of handling HTTP responses. You can specify the status code in various ways, depending on your use case within your NestJS application.

Here are a few ways to set the response status code:

1. **Using the @HttpCode() decorator**:

You can use the **@HttpCode()** decorator provided by NestJS to set the HTTP status code directly on a specific route handler method.

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

Description générée automatiquement**

1. **Using the @Res() decorator and Node.js response object**:

You can also manually set the status code using the **@Res()** decorator to inject the Node.js **response** object and directly set the status code.

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

Description générée automatiquement**

* 1. **Handling Update and Delete Requests**
* **PATCH**: HTTP method used to partially update a resource identified by a specific URI (Uniform Resource Identifier) within a RESTful API.
* **DELETE**: HTTP method employed to remove or delete the resource specified by a unique URI from a RESTful API.

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

Description générée automatiquement**

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

Description générée automatiquement**

* 1. **Implement Pagination with Query Parameters**

Query parameters are elements appended to the end of a URL that modify or refine the data being requested from a server. They are typically used in HTTP requests, particularly in GET requests, to pass specific information to the server.

Here's an example of a URL with query parameters:

<https://www.example.com/api/products?category=electronics&page=2&sort=price>

Query parameters are elements appended to the end of a URL that modify or refine the data being requested from a server. They are typically used in HTTP requests, particularly in GET requests, to pass specific information to the server.

Here's an example of a URL with query parameters:

bashCopy code

https://www.example.com/api/products?category=electronics&page=2&sort=price

In this URL:

* **https://www.example.com/api/products** is the base URL.
* **?** marks the beginning of the query parameters.
* **category=electronics**, **page=2**, and **sort=price** are the query parameters.

Each query parameter consists of a key-value pair separated by an equals sign (**=**). Multiple query parameters are separated by ampersands (**&**).

Explanation of the query parameters in the example URL:

* **category=electronics**: This parameter specifies the category of products to retrieve (e.g., electronics).
* **page=2**: It indicates the page number of the results.
* **sort=price**: This parameter defines how the results should be sorted (e.g., by price).

Query parameters serve to customize and filter the content requested from a server, allowing for more specific retrieval of data without altering the base URL. They are commonly used in web APIs, allowing clients to retrieve and manipulate data by providing additional information in the URL.

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

Description générée automatiquement**

* 1. **Creating a basic Service**
     1. **Nest Cli to create a service**

**nest generate service coffees**

**or**

**nest g s coffees**

* + 1. **Nest service**

In NestJS, a service is a TypeScript class that encapsulates business logic, interacts with data sources (like databases or APIs), and performs various operations to handle specific tasks within an application. Services in NestJS are commonly used to separate concerns, keep code modular, and facilitate reusability.

Here's a general overview of a service in NestJS:

1. **Creation of a Service**:

Services in NestJS can be created using the **@Injectable()** decorator, making use of TypeScript classes. For instance:

export class Coffee {

    id:number;

    name:string;

    brand:string;

    flavors:string[];

}

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

import { Coffee } from './entities/coffe.entity';

@Injectable()

export class CoffeesService {

  private coffees: Coffee[] = [

    {

      id: 1,

      name: 'Shipwreck Roast',

      brand: 'Buddy Brew',

      flavors: ['chocolate', 'vanilla'],

    },

  ];

  findAll() {

    return this.coffees;

  }

  findOne(id: string) {

    return this.coffees.find((item) => item.id === +id);

  }

  create(createCoffeeDto: any) {

    this.coffees.push(createCoffeeDto);

  }

  update(id: string, updateCoffeeDto: any) {

    const existingCoffee = this.findOne(id);

    if (existingCoffee) {

      // update the existing entity

    }

  }

  remove(id: string) {

    const coffeeIndex = this.coffees.findIndex((item) => item.id === +id);

    if (coffeeIndex >= 0) {

      this.coffees.splice(coffeeIndex, 1);

    }

  }

}

1. **Usage of Services**:

Once a service is created, it can be injected into various components like controllers or other services using dependency injection provided by NestJS. For instance, injecting **UserService** into a controller:

import {

  Controller,

  Get,

  Post,

  Body,

  Param,

  HttpCode,

  HttpStatus,

  Res,

  Patch,

  Delete,

  Query,

} from '@nestjs/common';

import { CoffeesService } from 'src/coffees/coffees.service';

@Controller('coffes')

export class CoffesController {

  constructor(private readonly coffesService: CoffeesService) {}

  @Get('')

  findAll(@Query() paginationQuery) {

    // const { limit, offset } = paginationQuery;

    return this.coffesService.findAll();

  }

  @Get(':id')

  findOne(@Param('id') id: string) {

    return this.coffesService.findOne(id);

  }

  @Post()

  @HttpCode(HttpStatus.ACCEPTED)

  create(@Body() body) {

    return this.coffesService.create(body);

  }

  @Patch(':id')

  update(@Param('id') id: string, @Body() body) {

    return this.coffesService.update(id, body);

  }

  @Delete(':id')

  remove(@Param('id') id: string) {

    return this.coffesService.remove(id)

  }

}

1. **Purpose of Services**:
   1. Encapsulating business logic: Services contain methods that handle specific business logic, keeping it separate from controllers.
   2. Reusability: Services promote code reuse as the same service can be injected into multiple components.
   3. Interacting with data sources: Services often interact with databases or external APIs to perform CRUD operations or fetch data.
   4. Testing: Services can be easily tested in isolation by using unit tests, ensuring the functionality works as expected.

Services play a crucial role in the overall architecture of a NestJS application, aiding in the organization, separation of concerns, and maintainability of the codebase.

* 1. **Error Messages**

In NestJS, error messages are commonly managed and thrown using the built-in exception handling mechanism. NestJS provides various ways to handle errors and customize error messages within the application.

import { HttpException, HttpStatus, Injectable } from '@nestjs/common';

import { Coffee } from './entities/coffe.entity';

@Injectable()

export class CoffeesService {

 findOne(id: string) {

    const coffee = this.coffees.find((item) => item.id === +id);

    if (!coffee) {

      throw new HttpException(`Coffe #${id} not found`, HttpStatus.NOT\_FOUND);

    }

  }

}

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

Description générée automatiquement**

 findOne(id: string) {

    const coffee = this.coffees.find((item) => item.id === +id);

    if (!coffee) {

      throw new NotFoundException(`Coffe #${id} not found`);

    }

    return coffee;

  }

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

Description générée automatiquement**

* 1. **Module**
     1. **Nest CLI to create a module**

**npx nest generate module coffees**

* + 1. **Nest Module**

In NestJS, modules are a fundamental building block used to organize the application into cohesive units, encapsulating related components, services, controllers, and other features.

Here's an explanation of modules in NestJS:

1. **Definition of Modules**:
   * Modules are TypeScript classes annotated with the **@Module()** decorator from **@nestjs/common**.
   * They serve as a way to organize code by grouping related components and providing a context for dependency injection.

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

import { CoffesController } from 'src/coffes/coffes.controller';

import { CoffeesService } from './coffees.service';

@Module({

  controllers: [CoffesController],

  providers: [CoffeesService],

})

export class CoffeesModule {}

In this example, the **UsersModule** defines a module that includes the **UsersController** and **UsersService** as components related to managing users.

1. **Purpose of Modules**:
   * **Encapsulation and Separation of Concerns**: Modules encapsulate related functionality, promoting separation of concerns and maintainability by keeping related components together.
   * **Dependency Management**: Modules define a scope for dependency injection, allowing NestJS to manage the creation and sharing of instances of services and other components within the module.
   * **Organization and Reusability**: Modules facilitate code organization and reusability by grouping related features, making it easier to manage and extend the application.
2. **Features of Modules**:
   * **controllers**: Defines the controllers that handle incoming HTTP requests within the module.
   * **providers**: Contains the providers (services, repositories, etc.) that can be injected across the module.
   * **exports**: Specifies the components (services, controllers, etc.) that can be utilized by other modules when this module is imported.
   * **Imports**: Import another module and use ther controller and provider

**4. Usage of Modules**:

* Modules can be imported into other modules to make their components available for use.
* The root module (usually named **AppModule**) serves as the starting point of the application and imports other modules as needed.

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

import { AppController } from './app.controller';

import { AppService } from './app.service';

import { CoffeesModule } from './coffees/coffees.module';

@Module({

  imports: [CoffeesModule],

  controllers: [AppController],

  providers: [AppService],

})

export class AppModule {}

Here, the **AppModule** imports the **UsersModule**, allowing components from the **UsersModule** to be used within the **AppModule**.

In summary, modules in NestJS provide a way to structure applications, manage dependencies, and organize related components, contributing to better maintainability and scalability of the codebase.

* 1. **Data transfer objects**
     1. **Introduction**

In NestJS, DTO stands for Data Transfer Object. DTOs are plain TypeScript classes used to define and shape data structures exchanged between different parts of an application, primarily between the client and server or between different layers within the server-side code.

Here's what a DTO typically does in NestJS:

1. **Structure and Shape Data**:
   * DTOs define the structure of the data being transferred between different parts of the application, such as between a client and server or between different layers (like controllers and services).
   * They encapsulate data in a specific format, allowing for a clear definition of the payload's structure.
2. **Validation and Transformation**:
   * DTOs help in data validation, allowing for stricter control over the incoming data by defining the expected shape and types of data.
   * They can be used in combination with validation libraries (like class-validator) to automatically validate incoming data against specified rules.
3. **Clear Communication and Documentation**:
   * Using DTOs enhances communication among developers and teams, making it clear what data structures are expected in different parts of the application.
   * They act as a form of documentation, providing a clear contract for the shape of data expected by different endpoints or services.

Example of a DTO in NestJS:

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

Description générée automatiquement

In the context of a NestJS application, DTOs are often used as method parameters in controllers to define the shape of the incoming data when handling HTTP requests (such as POST or PUT requests). They help ensure that the incoming data matches the expected structure and types, enabling easier handling and validation of input data.

DTOs are an essential part of maintaining a clear and structured communication mechanism within a NestJS application, promoting clarity, maintainability, and type safety when dealing with incoming and outgoing data.

* **Nest Cli to create a DTO**
  + **npx nest g class coffees/dto/create-coffee.dto --no-spec**

export class CreateCoffeeDto {

  readonly name: string;

  readonly brand: string;

  readonly flavors: string[];

}

 @Patch(':id')

  update(@Param('id') id: string, @Body() UpdateCoffeeDto: UpdateCoffeeDto) {

    return this.coffesService.update(id, UpdateCoffeeDto);

  }

In TypeScript, the **readonly** keyword is used to define properties that can only be assigned a value during object creation and cannot be modified afterward. It is primarily used to create immutable properties within an object, preventing their values from being changed once they are set.

* + 1. **Validation Input Data with Data Transfer Objects**
  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. **Global scoped pipes**

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. **Mapped type**

In TypeScript, mapped types are a powerful feature that allows you to create new types based on the properties of an existing type. These types enable you to manipulate and transform existing types by applying modifications or constraints to their properties.

#### Partial[#](https://docs.nestjs.com/openapi/mapped-types#partial)

When building input validation types (also called DTOs), it's often useful to build **create** and **update** variations on the same type. For example, the **create** variant may require all fields, while the **update** variant may make all fields optional.

Nest provides the PartialType() utility function to make this task easier and minimize boilerplate.

The PartialType() function returns a type (class) with all the properties of the input type set to optional. For example, suppose we have a **create** type as follows:

* 1. **Example**

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

Description générée automatiquement**

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

Description générée automatiquement**

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

Description générée automatiquement**

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

Description générée automatiquement**

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

Description générée automatiquement**

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

Description générée automatiquement**

* 1. **Handling Malicious requests data**
     1. **Validation Stripping properties**

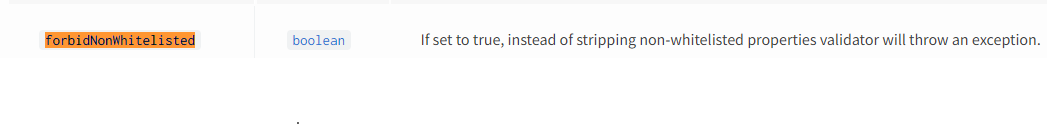
Our ValidationPipe can also filter out properties that should not be received by the method handler. In this case, we can **whitelist** the acceptable properties, and any property not included in the whitelist is automatically stripped from the resulting object. For example, if our handler expects email and password properties, but a request also includes an age property, this property can be automatically removed from the resulting DTO. To enable such behavior, set whitelist to true.

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

Description générée automatiquement**

When set to true, this will automatically remove non-whitelisted properties (those without any decorator in the validation class).

Alternatively, you can stop the request from processing when non-whitelisted properties are present, and return an error response to the user. To enable this, set the forbidNonWhitelisted option property to true, in combination with setting whitelist to true.

****

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

Description générée automatiquement**

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

Description générée automatiquement**

* + 1. **Auto-transform Payloads to DTO instances**

Payloads coming in over the network are plain JavaScript objects. The ValidationPipe can automatically transform payloads to be objects typed according to their DTO classes. To enable auto-transformation, set transform to true. This can be done at a method level:

@Post()

@UsePipes(new ValidationPipe({ transform: true }))

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

this.catsService.create(createCatDto);

}

To enable this behavior globally, set the option on a global pipe:

app.useGlobalPipes(

new ValidationPipe({

transform: true,

}),

);

With the auto-transformation option enabled, the ValidationPipe will also perform conversion of primitive types. In the following example, the findOne() method takes one argument which represents an extracted id path parameter:

@Get(':id')

findOne(@Param('id') id: number) {

console.log(typeof id === 'number'); // true

return 'This action returns a user';

}

By default, every path parameter and query parameter comes over the network as a string. In the above example, we specified the id type as a number (in the method signature). Therefore, the ValidationPipe will try to automatically convert a string identifier to a number.