**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**

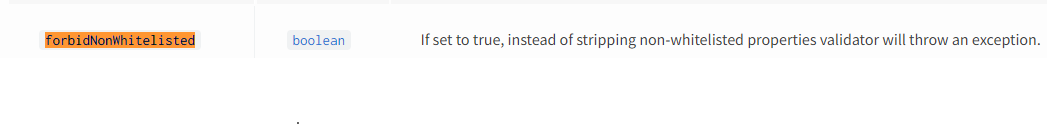
Our ValidationPipe can also filter out properties that should not be received by the method handler. In this case, we can **whitelist** the acceptable properties, and any property not included in the whitelist is automatically stripped from the resulting object. For example, if our handler expects email 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.

1. **Add PostgreSql with TypeOrm**
2. **Docker**

Docker is a popular platform and tool used for developing, shipping, and running applications within containers. It enables developers to package their applications and dependencies into a standardized unit known as a container. These containers are portable, lightweight, and isolated environments that can run on any system that supports the Docker platform, ensuring consistency across different environments, from development to production.

Key components of Docker include:

**Docker Engine**: The core of Docker that enables the creation and management of containers. It consists of the Docker daemon, which runs in the background, and the Docker CLI used for interacting with the daemon.

**Containers**: These are lightweight, portable, and self-sufficient units that package an application and its dependencies. Containers are isolated from one another and from the underlying infrastructure, ensuring consistency and reproducibility across different environments.

**Images**: Docker images are read-only templates used to create containers. An image includes the application code, runtime, libraries, dependencies, and configuration needed to run the application. Images are built from a Dockerfile, which specifies the steps to create the image.

**Dockerfile**: A text file that contains instructions for building a Docker image. It defines the environment and steps required to create a containerized application, including base image, dependencies, environment variables, and more.

Docker is widely used in software development and deployment due to its advantages, including:

**Portability**: Containers can run consistently across different environments, from development to testing to production.

**Isolation**: Containers isolate applications and their dependencies, preventing conflicts between different applications running on the same host.

**Resource Efficiency**: Containers share the host system's kernel and use fewer resources compared to virtual machines.

**Rapid Deployment**: Containers can be easily and quickly deployed, scaled, and managed.

Docker has become an integral part of modern software development workflows, facilitating DevOps practices, enabling microservices architectures, and streamlining the process of building, shipping, and running applications.

1. **Yml**

The **.yml** file extension is commonly used to denote files that contain YAML (YAML Ain't Markup Language) data. YAML files use this extension to indicate that the content within follows the YAML syntax and structure.

These **.yml** files can store various types of data structures, including configurations, settings, data models, and more. They are often used in software development for defining settings, specifying configurations for applications, storing data in a readable format, and exchanging data between different systems or programming languages.

For instance, in a project, you might find **.yml** files used for configurations in tools like:

* **Docker**: To define container configurations in **docker-compose.yml** files.
* **Continuous Integration/Continuous Deployment (CI/CD)**: Configuration of pipelines and workflows in **.yml** files, such as **.gitlab-ci.yml** or **.github/workflows/main.yml**.
* **Configuration files**: Settings for applications or services, like **config.yml** or **settings.yml**.

These files follow the YAML syntax rules, utilizing indentation and key-value pairs to define data structures. The **.yml** extension simply denotes that the content within the file should adhere to YAML formatting conventions.

* 1. **Runing PostgreSql**

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

Description générée automatiquement**

And then we execute the cmd:

**docker-compose up -d**

1. **TypeOrm Module**

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:

### Setting Up TypeORM:

1. **Install TypeORM**: Install TypeORM and the required database driver using npm (Node Package Manager):

**npm i @nestjs/typeorm typeorm pg**

**Set Up Configuration**:

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

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

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

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

import { TypeOrmModule } from '@nestjs/typeorm';

@Module({

  imports: [

    CoffeesModule,

    TypeOrmModule.forRoot({

      type: 'postgres',

      host: 'localhost',

      port: 5432,

      username: 'postgres',

      password: 'pass123',

      database: 'postgres',

      autoLoadEntities: true,

      synchronize: true,

    }),

  ],

  controllers: [AppController],

  providers: [AppService],

})

export class AppModule {}

**Define Entities**: Create TypeScript classes that represent your database tables/entities. An entity class typically represents a table, and its properties represent table columns.

Example of an entity class:

### Example Entity in NestJS with TypeORM:

import { Column, Entity, PrimaryGeneratedColumn } from 'typeorm';

@Entity() // sql table === 'coffee'

export class Coffee {

  @PrimaryGeneratedColumn()

  id: number;

  @Column()

  name: string;

  @Column()

  brand: string;

  @Column('json', { nullable: true })

  flavors: string[];

}

### Integrating Entity with NestJS:

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

import { TypeOrmModule } from '@nestjs/typeorm';

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

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

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

@Module({

  imports: [TypeOrmModule.forFeature([Coffee])],

  controllers: [CoffesController],

  providers: [CoffeesService],

})

export class CoffeesModule {}

### Explanation of TypeOrmModule.forFeature([User]):

* **TypeOrmModule.forFeature([User])**: This method is provided by **@nestjs/typeorm** and is used to specify which entities will be registered and managed by TypeORM within the current module context. It's essentially telling TypeORM to load the **User** entity into the current module, making it available for use in services, controllers, and other components within this module.

In a NestJS application with TypeORM, you can use repositories to interact with your database entities. Repositories provide a way to perform various database operations like querying, inserting, updating, and deleting records.

To access the database using repositories, you typically follow these steps:

### 1. Inject Repository into a Service or Controller:

Inject the repository for the specific entity you want to work with into a NestJS service or controller using **@InjectRepository()** decorator.

For example, considering a **UserService** that interacts with the **User** entity:

### 2. Performing Database Operations:

Once the repository is injected, you can use its methods to perform CRUD operations or other database-related tasks:

import {

  HttpException,

  HttpStatus,

  Injectable,

  NotFoundException,

} from '@nestjs/common';

import { InjectRepository } from '@nestjs/typeorm';

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

import { Repository } from 'typeorm';

import { CreateCoffeeDto } from './dto/create-coffee.dto';

import { UpdateCoffeeDto } from './dto/update-coffee.dto';

@Injectable()

export class CoffeesService {

  constructor(

    @InjectRepository(Coffee)

    private readonly cofeeRepository: Repository<Coffee>,

  ) {}

  findAll() {

    return this.cofeeRepository.find();

  }

  findOne(id: string) {

    const coffee = this.cofeeRepository.findOneBy({ id: +id });

    if (!coffee) {

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

    }

    return coffee;

  }

  create(createCoffeeDto: CreateCoffeeDto) {

    const coffee = this.cofeeRepository.create(createCoffeeDto);

    return this.cofeeRepository.save(coffee);

  }

  async update(id: string, updateCoffeeDto: UpdateCoffeeDto) {

    const coffee = await this.cofeeRepository.preload({

      id: +id,

      ...updateCoffeeDto,

    });

    if (!coffee) {

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

    }

    return this.cofeeRepository.save(coffee);

  }

  async remove(id: string) {

    const coffe = await this.findOne(id);

    return this.cofeeRepository.remove(coffe);

  }

}

### Summary:

By injecting a TypeORM repository into a service or controller in NestJS, you can conveniently interact with your database entities using methods provided by the repository. This approach enables you to perform various database operations while maintaining separation of concerns within your application.

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

Description générée automatiquement**

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

Description générée automatiquement**

**Retrieve Entities with Relations**:

Relations are not loaded by default so we need to use TypeORM's **find** or **findOne** method along with the **relations** option to eagerly load related entities:

findAll() {

    return this.cofeeRepository.find({

      relations: ['flavors'],

    });

  }

  findOne(id: string) {

    const coffee = this.cofeeRepository.find({

      where: { id: +id },

      relations: ['flavors'],

    });

    if (!coffee) {

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

    }

    return coffee;

  }

**Using Cascading inserts and updates**

In TypeORM, when working with many-to-many relationships, the concept of cascading inserts and updates involves automatically persisting changes to related entities through a join (relation) table. This behavior allows you to propagate modifications from one entity to another when performing operations like inserts, updates, or deletes.

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

Description générée automatiquement**

Certainly! The **@ManyToMany** decorator in TypeORM is used to define a many-to-many relationship between entities. Here's an explanation of the parameters used within the **@ManyToMany** decorator:

1. **(type) => Flavor**:
   * **(type)** is the argument used to define the target entity type.
   * **Flavor** is the target entity (the entity on the other side of the relationship). In this case, the relationship is established with the **Flavor** entity.
2. **(flavor) => flavor.coffees**:
   * **(flavor)** is a function used to specify the field in the target entity (**Flavor**) that represents the inverse side of the relationship.
   * **flavor.coffees** represents the property in the **Flavor** entity that relates to the current entity (where this **@ManyToMany** decorator is used).
3. **{ cascade: true }**:
   * **cascade** is an option that specifies whether operations performed on one side of the relationship should cascade (be propagated) to the other side.
   * **cascade: true** indicates that operations such as insert, update, and remove should cascade from one entity to the other. For example, if a coffee flavor is removed, it will automatically remove the association from the **Flavor** entity.

Putting it all together, this **@ManyToMany** decorator sets up a many-to-many relationship between the current entity (where this decorator is applied) and the **Flavor** entity. The **Flavor** entity has a property called **coffees** that represents the inverse side of this relationship. When a cascade is enabled (**cascade: true**), operations like insert, update, or remove on one side of the relationship will be automatically propagated to the other side, ensuring data consistency between the entities involved in the many-to-many relationship.

import {

  Column,

  Entity,

  PrimaryGeneratedColumn,

  JoinTable,

  ManyToMany,

} from 'typeorm';

import { Flavor } from './flavor.entity';

@Entity() // sql table === 'coffee'

export class Coffee {

  @PrimaryGeneratedColumn()

  id: number;

  @Column()

  name: string;

  @Column()

  brand: string;

  @JoinTable()

  @ManyToMany((type) => Flavor, (flavor) => flavor.coffees, {

    cascade: true,

  })

  flavors: Flavor[];

}

import {

  HttpException,

  HttpStatus,

  Injectable,

  NotFoundException,

} from '@nestjs/common';

import { InjectRepository } from '@nestjs/typeorm';

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

import { Repository } from 'typeorm';

import { CreateCoffeeDto } from './dto/create-coffee.dto';

import { UpdateCoffeeDto } from './dto/update-coffee.dto';

import { Flavor } from './entities/flavor.entity';

@Injectable()

export class CoffeesService {

  constructor(

    @InjectRepository(Coffee)

    private readonly cofeeRepository: Repository<Coffee>,

    @InjectRepository(Flavor)

    private readonly flavorRepository: Repository<Flavor>,

  ) {}

  findAll() {

    return this.cofeeRepository.find({

      relations: ['flavors'],

    });

  }

  findOne(id: string) {

    const coffee = this.cofeeRepository.find({

      where: { id: +id },

      relations: ['flavors'],

    });

    if (!coffee) {

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

    }

    return coffee;

  }

  async create(createCoffeeDto: CreateCoffeeDto) {

    const flavors = await Promise.all(

      createCoffeeDto.flavors.map((name) => this.preloadFlavorByName(name)),

    );

    const coffee = this.cofeeRepository.create({ ...createCoffeeDto, flavors });

    return this.cofeeRepository.save(coffee);

  }

  async update(id: string, updateCoffeeDto: UpdateCoffeeDto) {

    const flavors = await Promise.all(

      updateCoffeeDto.flavors &&

        (await Promise.all(

          updateCoffeeDto.flavors.map((name) => this.preloadFlavorByName(name)),

        )),

    );

    const coffee = await this.cofeeRepository.preload({

      id: +id,

      ...updateCoffeeDto,

      flavors,

    });

    if (!coffee) {

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

    }

    return this.cofeeRepository.save(coffee);

  }

  async remove(id: string) {

    const coffe = await this.findOne(id);

    return this.cofeeRepository.remove(coffe);

  }

  private async preloadFlavorByName(name: string): Promise<Flavor> {

    const existingFlavor = await this.flavorRepository.find({

      where: { name: name },

    })[0];

    if (existingFlavor) {

      return existingFlavor;

    }

    return this.flavorRepository.create({ name });

  }

}

**Pagination**

****

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

import { NestFactory } from '@nestjs/core';

import { AppModule } from './app.module';

async function bootstrap() {

  const app = await NestFactory.create(AppModule);

  app.useGlobalPipes(

    new ValidationPipe({

      whitelist: true,

      transform: true,

      forbidNonWhitelisted: true,

      // For pagination Dto en desous

      transformOptions: {

        enableImplicitConversion: true,

      },

    }),

  );

  await app.listen(3000);

}

bootstrap();

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

Description générée automatiquement**

import { IsOptional, IsPositive } from 'class-validator';

export class PaginationQueryDto {

  @IsOptional()

  @IsPositive()

  limit: number;

  @IsOptional()

  @IsPositive()

  offset: number;

}

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

Description générée automatiquement**

  findAll(paginationQuery: PaginationQueryDto) {

    const { limit, offset } = paginationQuery;

    return this.cofeeRepository.find({

      relations: ['flavors'],

      skip: offset,

      take: limit,

    });

  }

****

 @Get('')

  findAll(@Query() paginationQuery: PaginationQueryDto) {

    return this.coffesService.findAll(paginationQuery);

  }

**Use Transaction**

**Ba9i**

In NestJS, there isn't a specific **useTransactions** method or function provided by the framework itself. The concept of transactions in NestJS primarily revolves around the interaction with databases or any other data stores.

Transactions in a NestJS application are typically managed using the features and capabilities of the database library or ORM (Object-Relational Mapping) tool being used, such as TypeORM, Sequelize, Prisma, etc.

The need for transactions in a NestJS application, or any application dealing with a database, arises from the requirement to ensure data integrity and consistency when performing multiple operations on the database.

Here's why transactions are important:

1. **Atomicity:** Transactions ensure that a group of operations either complete successfully as a whole or are fully rolled back if any one of them fails. This maintains the integrity of the data.
2. **Consistency:** Transactions help maintain consistency in the database. When multiple operations need to be executed together, either they all execute successfully, or none of them apply, preventing the database from being in an inconsistent state.
3. **Isolation:** Transactions provide isolation from other concurrent transactions. They allow operations to be carried out independently until they are committed, ensuring that other transactions do not interfere with the data being manipulated during the transaction.
4. **Durability:** Once a transaction is committed, its changes are permanently saved in the database, ensuring durability.

To use transactions in NestJS, you generally work with the methods provided by your chosen database library or ORM. These methods usually involve starting a transaction, executing operations within that transaction, and then either committing or rolling back the transaction based on the success or failure of those operations.

For example, as shown in the previous example using TypeORM, the **transaction** method of the **EntityManager** is used to encapsulate multiple database operations within a single transaction.

It's crucial to handle transactions properly to ensure data consistency and integrity in applications dealing with databases, and NestJS, in conjunction with database libraries or ORMs, provides tools and mechanisms to manage these transactions effectively.

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

Description générée automatiquement**

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

Description générée automatiquement**

import { Column, Entity, PrimaryGeneratedColumn } from 'typeorm';

@Entity()

export class Event {

  @PrimaryGeneratedColumn()

  id: number;

  @Column()

  type: string;

  @Column()

  name: string;

  @Column()

  payload: Record<string, any>;

}

**Une image contenant texte, capture d’écran, carte de visite, Police

Description générée automatiquement**

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

import { TypeOrmModule } from '@nestjs/typeorm';

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

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

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

import { Flavor } from './entities/flavor.entity';

import { Event } from 'src/events/entities/event.entity/event.entity';

@Module({

  imports: [TypeOrmModule.forFeature([Coffee, Flavor, Event])],

  controllers: [CoffesController],

  providers: [CoffeesService],

})

export class CoffeesModule {}

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

Description générée automatiquement**

import {

  Column,

  Entity,

  PrimaryGeneratedColumn,

  JoinTable,

  ManyToMany,

} from 'typeorm';

import { Flavor } from './flavor.entity';

@Entity() // sql table === 'coffee'

export class Coffee {

  @PrimaryGeneratedColumn()

  id: number;

  @Column()

  name: string;

  @Column()

  brand: string;

  @Column({ default: 0 })

  recommendations: number;

  @JoinTable()

  @ManyToMany((type) => Flavor, (flavor) => flavor.coffees, {

    cascade: true,

  })

  flavors: Flavor[];

}

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

Description générée automatiquement**

import {

  HttpException,

  HttpStatus,

  Injectable,

  NotFoundException,

  Query,

} from '@nestjs/common';

import { InjectRepository } from '@nestjs/typeorm';

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

import { Connection, Repository } from 'typeorm';

import { CreateCoffeeDto } from './dto/create-coffee.dto';

import { UpdateCoffeeDto } from './dto/update-coffee.dto';

import { Flavor } from './entities/flavor.entity';

import { PaginationQueryDto } from 'src/common/dto/pagination-query.dto/pagination-query.dto';

import { Event } from 'src/events/entities/event.entity/event.entity';

@Injectable()

export class CoffeesService {

  constructor(

    @InjectRepository(Coffee)

    private readonly cofeeRepository: Repository<Coffee>,

    @InjectRepository(Flavor)

    private readonly flavorRepository: Repository<Flavor>,

    private readonly connection: Connection,

  ) {}

  findAll(paginationQuery: PaginationQueryDto) {

    const { limit, offset } = paginationQuery;

    return this.cofeeRepository.find({

      relations: ['flavors'],

      skip: offset,

      take: limit,

    });

  }

  findOne(id: string) {

    const coffee = this.cofeeRepository.find({

      where: { id: +id },

      relations: ['flavors'],

    });

    if (!coffee) {

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

    }

    return coffee;

  }

  async create(createCoffeeDto: CreateCoffeeDto) {

    const flavors = await Promise.all(

      createCoffeeDto.flavors.map((name) => this.preloadFlavorByName(name)),

    );

    const coffee = this.cofeeRepository.create({ ...createCoffeeDto, flavors });

    return this.cofeeRepository.save(coffee);

  }

  async update(id: string, updateCoffeeDto: UpdateCoffeeDto) {

    const flavors = await Promise.all(

      updateCoffeeDto.flavors &&

        (await Promise.all(

          updateCoffeeDto.flavors.map((name) => this.preloadFlavorByName(name)),

        )),

    );

    const coffee = await this.cofeeRepository.preload({

      id: +id,

      ...updateCoffeeDto,

      flavors,

    });

    if (!coffee) {

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

    }

    return this.cofeeRepository.save(coffee);

  }

  async remove(id: string) {

    const coffe = await this.findOne(id);

    return this.cofeeRepository.remove(coffe);

  }

  async recommendCoffe(coffe: Coffee) {

    const queryRunner = this.connection.createQueryRunner();

    await queryRunner.connect();

    await queryRunner.startTransaction();

    try {

      coffe.recommendations++;

      const recommendEvent = new Event();

      recommendEvent.name = 'recommend\_coffee';

      recommendEvent.type = 'coffee';

      recommendEvent.payload = { coffeeId: coffe.id };

      await queryRunner.manager.save(coffe);

      await queryRunner.manager.save(recommendEvent);

      await queryRunner.commitTransaction();

    } catch (err) {

      // since we have errors lets rollback the changes we made

      await queryRunner.rollbackTransaction();

    } finally {

      // you need to release a queryRunner which was manually instantiated

      await queryRunner.release();

    }

  }

  private async preloadFlavorByName(name: string): Promise<Flavor> {

    const existingFlavor = await this.flavorRepository.find({

      where: { name: name },

    })[0];

    if (existingFlavor) {

      return existingFlavor;

    }

    return this.flavorRepository.create({ name });

  }

}

**Adding indexes Injection**

**Ba9i**

**Setting Up migration**

**Ba9i**

Setting up migrations in a NestJS application often involves using an ORM (Object-Relational Mapping) tool like TypeORM to manage database schemas and perform migrations. TypeORM allows generating migrations based on changes in your entity classes and applying these changes to your database.

Here are the steps to set up migrations using TypeORM in a NestJS application:

1. **Install TypeORM and Database Driver:**

Ensure you have TypeORM and the necessary database driver installed. For example, if you're using PostgreSQL, you would install TypeORM and the PostgreSQL driver like this:

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

Description générée automatiquement**

1. Replace **pg** with the appropriate driver for your database (e.g., **mysql**, **sqlite**, **mssql**, etc.).
2. **Configuration in ormconfig.ts:**

Set up the TypeORM module in your NestJS application by importing the **TypeOrmModule** and configuring it with your database connection options. Modify the **app.module.ts** or the appropriate module file as follows:

module.exports = {

      type: 'postgres', // Replace with your database type

      host: 'localhost',

      port: 5432, // Replace with your database port

      username: 'postgres',

      password: 'pass123',

      database: 'postgres',

      entities: ['dist/\*\*/\*.entity{.ts,.js}'],

      migrations: ['dist/migrations/\*{.ts,.js}'],

      cli: {

        migrationsDir: 'src/migrations',

      },

};

**Creating Entities:**

Define your TypeORM entities that represent your database tables. An example of an entity file (**your-entity.entity.ts**):

**Generating Migrations:**

Once you have defined your entities, generate migrations based on the changes to these entities. Use the TypeORM CLI to create a migration:

1. **Dependency Injection**
2. **Understand dependency Injection.**
   1. **Design Pattern**

One of the fundamental design patterns in software engineering is the **Model-View-Controller (MVC)** pattern. MVC is a widely adopted architectural pattern used to structure software applications by separating concerns into three interconnected components:

1. **Model (M):**
   * Represents the application's data and business logic.
   * Handles the data manipulation, validation, and processing.
   * It notifies views of changes in the data (through observers or events).
   * Typically doesn't directly interact with the user interface.
2. **View (V):**
   * Represents the presentation layer or the user interface.
   * Renders the data from the model into a user-readable format.
   * Observes changes in the model and updates the interface accordingly.
   * Often multiple views can exist for the same data.
3. **Controller (C):**
   * Acts as an intermediary between the model and the view.
   * Receives user input and initiates actions on the model.
   * Updates the view when the model changes.
   * Handles user interactions and modifies the model accordingly.

Key benefits of the MVC pattern include:

* **Separation of Concerns:** Dividing an application into distinct components helps maintainability, as each component has its own responsibility.
* **Modifiability:** Changes in one component do not affect the others, allowing easier modifications and updates.
* **Reusability:** Components can often be reused in different parts of the application or in other applications.
* **Testability:** Easier to test individual components in isolation, aiding in unit testing.

Apart from MVC, other important design patterns in software engineering include:

* **Factory Pattern:** Used for creating objects without specifying their exact class.
* **Singleton Pattern:** Restricts the instantiation of a class to a single instance.
* **Observer Pattern:** Defines a one-to-many dependency between objects, so that when one object changes state, all its dependents are notified and updated automatically.
* **Dependency Injection (DI):** Technique where one object supplies the dependencies of another object, reducing coupling between components.
* **Decorator Pattern:** Allows behavior to be added to individual objects dynamically.
* **Strategy Pattern:** Enables selecting an algorithm at runtime from a family of algorithms.
* **Facade Pattern:** Provides a simplified interface to a larger body of code, making it easier to use.

Understanding these design patterns and when to apply them is crucial for creating robust, maintainable, and scalable software systems. They provide proven solutions to common software design problems and promote best practices in software development.

* 1. **Understand dependency injection.**

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

Description générée automatiquement**

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

Description générée automatiquement**

Certainly! Dependency Injection (DI) is a fundamental design pattern in software engineering that NestJS heavily utilizes. It's a technique where one object supplies the dependencies of another object, thus reducing the coupling between components and making the code more modular, maintainable, and testable.

In NestJS, Dependency Injection is implemented using TypeScript decorators and a built-in Inversion of Control (IoC) container.

Here's a breakdown of how Dependency Injection works in NestJS:

### 1. **Decorators:**

NestJS uses decorators like **@Injectable()**, **@Inject()**, **@Controller()**, **@InjectRepository()**, etc., to denote the various types of dependencies and their injection points.

* **@Injectable():** This decorator marks a class as a provider that can be injected into other components. Services, repositories, or any class that holds business logic or acts as a singleton should be decorated with **@Injectable()**.
* **@Inject():** Use **@Inject()** to specify the dependency you want to inject into a class constructor or method.

### 2. **IoC Container:**

NestJS has an Inversion of Control (IoC) container, responsible for managing the application's dependencies. It maintains a registry of providers (classes decorated with **@Injectable()**) and resolves dependencies based on their registered providers.

### 3. **Constructor Injection:**

The most common form of dependency injection in NestJS is Constructor Injection. Here's an example:

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

Description générée automatiquement**

In this example, **ConsumerService** requires an instance of **DataService**. When **ConsumerService** is created, NestJS automatically resolves the dependency and injects an instance of **DataService** into its constructor.

### 4. **Module Providers:**

In NestJS, dependencies are usually provided within modules using the **providers** property. Modules define the context for how dependencies are scoped and managed.

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

Description générée automatiquement**

### 5. **Scopes:**

NestJS supports different scopes for providers like Singleton (default), Request, etc. You can specify the scope using the **@Injectable({ scope: ScopeType })** decorator or the **@Injectable()** default scope.

### 6. **Custom Injection Tokens:**

You can create custom injection tokens to define and inject unique dependencies manually using **@Inject()**.

### Benefits of Dependency Injection in NestJS:

* **Modularity:** Components are loosely coupled, making it easier to replace or modify them without affecting the entire application.
* **Testability:** Easier unit testing by mocking or substituting dependencies during tests.
* **Readability:** Clearer code by explicitly declaring dependencies at their injection points.
* **Reusability:** Promotes the reuse of components in different parts of the application.

By leveraging Dependency Injection, NestJS enables the creation of scalable and maintainable applications with clear separation of concerns. It's a key aspect of building robust and efficient NestJS applications.

1. **Control NestJS Module Encapsulation**

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

Description générée automatiquement**

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

Description générée automatiquement**

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

Description générée automatiquement**

**It doesn’t work because because we don’t export the coffesService in our coffee module:**

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

Description générée automatiquement**

**Solution :**

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

Description générée automatiquement**

1. **Non-class-based Provider Tokens**
   1. **useClass**

In NestJS, providers are essentially classes decorated with **@Injectable()** that can be injected into other components using dependency injection. However, there might be scenarios where you want to use non-class-based tokens as providers.

NestJS supports non-class-based tokens as providers using the **useValue**, **useFactory**, and **useExisting** properties in the **provide** array of a module's configuration.

Here's how you can use non-class-based tokens as providers:

### Using useValue:

You can provide a value directly without using a class.

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

Description générée automatiquement**

@Module({

  imports: [TypeOrmModule.forFeature([Coffee, Flavor])],

  controllers: [CoffesController],

  providers: [

    CoffeesService,

    { provide: 'COFFEE\_BRANDS', useValue: ['buddy brew', 'nescafe'] },

  ],

  exports: [CoffeesService],

})

export class CoffeesModule {}

****

@Injectable()

export class CoffeesService {

  constructor(

    @InjectRepository(Coffee)

    private readonly cofeeRepository: Repository<Coffee>,

    @InjectRepository(Flavor)

    private readonly flavorRepository: Repository<Flavor>,

    // private readonly connection: Connection,

    @Inject('COFFEE\_BRANDS') coffeBrands: string[],

  ) {

    console.log('Non-class-based Provider Tokens', coffeBrands);

  }

* 1. **useFactory**

In NestJS, **useFactory** is a property used within the **providers** array of a module to define and provide dependencies using a factory function. This factory function generates and returns the value or instance that will be injected as a dependency.

The **useFactory** property allows for dynamic creation and customization of dependencies at runtime, enabling more flexibility in managing and providing instances based on certain conditions or configurations.

Here's an example demonstrating the usage of **useFactory**:

Suppose you have a factory function to create and configure a connection to a database:

Suppose you have a factory function to create and configure a connection to a database:

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

Description générée automatiquement

In this example:

* **connectionFactory** is a factory provider.
* **provide** specifies the token used for injection.
* **useFactory** contains the factory function responsible for creating and returning a database connection.
* The factory function (**useFactory**) returns a **Promise<Connection>** as it's creating a connection asynchronously (**createConnection** typically returns a promise).

This **connectionFactory** can be registered within a module's **providers** array to make the database connection available for injection across the application:

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

Description générée automatiquement

Now, whenever the **'CONNECTION'** token is requested for injection, NestJS will execute the **useFactory** function to generate and provide the database connection.

### Dynamic Configurations:

The **useFactory** property is also helpful for dynamically creating instances based on conditions or configurations. For instance, you could incorporate environment variables or other runtime conditions inside the **useFactory** function to create different instances based on the environment or specific conditions.

This approach allows NestJS applications to dynamically generate and inject dependencies, facilitating adaptability and customization based on various runtime scenarios.

Nest Project example:

@Module({

  imports: [TypeOrmModule.forFeature([Coffee, Flavor])],

  controllers: [CoffesController],

  providers: [

    CoffeesService,

    {

      provide: ConfigService,

      useClass:

        process.env.NODE\_ENV === 'developement'

          ? DevelopmentConfigService

          : ProductionConfigService,

    },

    { provide: 'COFFEE\_BRANDS', useFactory: () => ['buddy brew', 'nescafe'] },

  ],

  exports: [CoffeesService],

})

export class CoffeesModule {}

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

Description générée automatiquement

* 1. **Leverage Async Providers**

In NestJS, the **useFactory** property within a provider's configuration allows for the creation of asynchronous dependencies using an asynchronous factory function. This is particularly useful when dealing with operations that return promises or require asynchronous initialization.

Let's consider an example where you need to asynchronously fetch configuration settings from an external service during the creation of a provider. You 'll use **useFactory** with an asynchronous factory function to achieve this:Une image contenant texte, Appareils électroniques, capture d’écran, logiciel

Description générée automatiquement

Now, let's create an asynchronous provider that uses **useFactory** with an asynchronous function to fetch configuration settings during initialization:

Une image contenant texte, capture d’écran

Description générée automatiquement

In this example:

* **ConfigService** is a service responsible for fetching configuration settings.
* The **useFactory** property defines an asynchronous factory function that returns the result of **configService.getConfig()**.
* **inject** specifies the dependencies that need to be injected into the factory function (in this case, **ConfigService**).

Now, you can inject the **'ASYNC\_CONFIG'** token into other components across the application, and it will provide the resolved configuration settings asynchronously.

This approach allows for the asynchronous initialization of dependencies by leveraging **useFactory** with an asynchronous function, enabling NestJS applications to handle asynchronous operations during the creation of providers.

1. **useClass**

In NestJS, when defining providers within modules, the **useClass** property is used to specify a class provider. The **useClass** property allows you to associate an abstract token (the provider) with a concrete class implementation, allowing NestJS to resolve dependencies and perform dependency injection.

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

import { TypeOrmModule } from '@nestjs/typeorm';

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

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

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

import { Flavor } from './entities/flavor.entity';

import { Event } from 'src/events/entities/event.entity/event.entity';

class ConfigService {}

class DevelopmentConfigService {}

class ProductionConfigService {}

@Module({

  imports: [TypeOrmModule.forFeature([Coffee, Flavor])],

  controllers: [CoffesController],

  providers: [

    CoffeesService,

    {

      provide: ConfigService,

      useClass:

        process.env.NODE\_ENV === 'developement'

          ? DevelopmentConfigService

          : ProductionConfigService,

    },

    { provide: 'COFFEE\_BRANDS', useValue: ['buddy brew', 'nescafe'] },

  ],

  exports: [CoffeesService],

})

export class CoffeesModule {}

In the provided NestJS **CoffeesModule** example, the **useClass** property is used within the **providers** array to conditionally select a configuration service (**ConfigService**) implementation based on the current environment (**NODE\_ENV**).

Here's a breakdown of the code:

1. **Environment-Specific Configuration Services:**

You have three classes: **ConfigService**, **DevelopmentConfigService**, and **ProductionConfigService**. Each of these classes represents a different configuration service implementation tailored for different environments (development and production).

1. **Module Configuration:**

The **CoffeesModule** is defined as a NestJS module using the **@Module** decorator. Here's what's happening inside this module:

* + **Imports:**
    - **TypeOrmModule.forFeature([Coffee, Flavor])**: Imports TypeORM modules for the **Coffee** and **Flavor** entities, allowing their usage within this module.
  + **Controllers:**
    - **CoffesController**: Specifies the controller(s) associated with this module.
  + **Providers:**
    - **CoffeesService**: Registers the **CoffeesService** as a provider within the module.
    - **ConfigService**: Specifies a provider for a configuration service. Here, the **useClass** property is employed conditionally to determine which configuration service implementation to use based on the **NODE\_ENV** environment variable.
      * If **NODE\_ENV** is set to **'development'**, the **DevelopmentConfigService** class is used.
      * If **NODE\_ENV** is set to **'production'**, the **ProductionConfigService** class is used.
    - **'COFFEE\_BRANDS'**: Registers a provider for the **'COFFEE\_BRANDS'** token using **useValue**, providing an array of coffee brands as a static value.
  + **Exports:**
    - **CoffeesService**: Specifies that **CoffeesService** should be available for import by other modules.

### Use of useClass:

The use of **useClass** within the **providers** array conditionally selects a specific class to act as the **ConfigService** provider based on the environment variable **NODE\_ENV**.

This conditional registration of the **ConfigService** allows NestJS to use an appropriate configuration service (**DevelopmentConfigService** or **ProductionConfigService**) depending on the environment, enabling environment-specific configurations to be utilized in the application.

By utilizing **useClass** in this context, the application can adapt its behavior and settings based on the environment it is running in, ensuring different configurations for development and production environments.

1. **Create a Dynamic Module**

In NestJS, dynamic modules allow for the creation of modules at runtime. This is particularly useful when you want to dynamically configure and register providers based on certain conditions or configurations.

Here's an example demonstrating how to create a dynamic module in NestJS:

Let's assume you have a **LoggerService** and **LoggerConfigService**, and you want to create a module that dynamically registers the **LoggerService** based on configurations provided by **LoggerConfigService**.

First, define the services:

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

Description générée automatiquement**

Now, let's create a dynamic module that registers the **LoggerService** based on the configuration from **LoggerConfigService**:

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

Description générée automatiquement**

In this example:

* **LoggerModule** is a dynamic module created using the **forRoot** static method.
* Inside the **forRoot** method, **LoggerConfigService** is instantiated to access its configuration.
* The **useFactory** property within the **providers** array conditionally creates an instance of **LoggerService** based on the configuration retrieved from **LoggerConfigService**.
* If the logger is enabled according to the configuration, **LoggerService** is provided as a provider; otherwise, it returns **null**.
* **LoggerConfigService** is also registered as a provider within the module.

To use this dynamic module, you can import it in your application's main module (**AppModule**):

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

Description générée automatiquement**

This **LoggerModule** can now be imported into other modules, and the **LoggerService** will be provided based on the configuration settings retrieved from **LoggerConfigService** at runtime.

1. **Control providers scope**

Bien sûr, voici une explication des scopes des fournisseurs (providers) dans NestJS :

Dans NestJS, la portée des fournisseurs détermine comment les instances de ces fournisseurs sont créées et gérées au sein de l'application. NestJS prend en charge plusieurs portées de fournisseurs qui définissent le cycle de vie et le partage des instances dans toute l'application.

### Différentes portées des fournisseurs dans NestJS :

1. **Singleton (par défaut) :**
   * La portée par défaut dans NestJS.
   * Une seule instance du fournisseur est créée et partagée dans toute l'application.
   * La même instance est injectée partout où le fournisseur est demandé.

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

Description générée automatiquement**

**Transient :**

* Chaque injection crée une nouvelle instance du fournisseur.
* Une nouvelle instance est fournie pour chaque module, contrôleur ou service qui en fait la demande.

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

Description générée automatiquement**

1. **Request (Requête) :**

* Une nouvelle instance est créée par requête HTTP entrante.
* Elle est associée au contexte de la requête actuelle et est disponible tout au long du cycle de vie de la requête.

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

Description générée automatiquement**

Bien sûr, voici une explication des scopes des fournisseurs (providers) dans NestJS :

Dans NestJS, la portée des fournisseurs détermine comment les instances de ces fournisseurs sont créées et gérées au sein de l'application. NestJS prend en charge plusieurs portées de fournisseurs qui définissent le cycle de vie et le partage des instances dans toute l'application.

### Différentes portées des fournisseurs dans NestJS :

1. **Singleton (par défaut) :**
   * La portée par défaut dans NestJS.
   * Une seule instance du fournisseur est créée et partagée dans toute l'application.
   * La même instance est injectée partout où le fournisseur est demandé.

typescriptCopy code

@Injectable() // La portée singleton est appliquée par défaut export class ServiceSingleton { // Logique du service }

1. **Transient :**
   * Chaque injection crée une nouvelle instance du fournisseur.
   * Une nouvelle instance est fournie pour chaque module, contrôleur ou service qui en fait la demande.

typescriptCopy code

@Injectable({ scope: Scope.TRANSIENT }) export class ServiceTransient { // Logique du service }

1. **Request (Requête) :**
   * Une nouvelle instance est créée par requête HTTP entrante.
   * Elle est associée au contexte de la requête actuelle et est disponible tout au long du cycle de vie de la requête.

typescriptCopy code

@Injectable({ scope: Scope.REQUEST }) export class ServiceRequete { // Logique du service }

### Comment contrôler la portée des fournisseurs :

* **Utilisation du décorateur @Injectable() :**
  + Par défaut, les fournisseurs sont des singletons. Ajouter **@Injectable()** sans spécifier une portée différente résulte en un fournisseur singleton.
* **Spécification de la portée dans @Injectable() :**
  + Utilisez l'option **scope** dans **@Injectable()** pour définir une portée différente pour le fournisseur.
* **Portée au niveau du module :**
  + Par défaut, les fournisseurs sont enregistrés au niveau du module. Lorsque vous importez un module dans un autre module, ses fournisseurs sont disponibles dans le module important.
* **Personnalisation de la portée en utilisant des fournisseurs personnalisés :**
  + Vous pouvez mettre en œuvre une logique de portée personnalisée en créant des fournisseurs personnalisés ou en utilisant des factories pour contrôler comment les instances sont créées et partagées.

### Quand utiliser différentes portées :

* **Singleton :** Utilisez lorsque vous souhaitez une seule instance partagée dans toute l'application.
* **Transient :** Utilisez lorsque vous avez besoin d'une nouvelle instance pour chaque module ou injection.
* **Requête :** Utilisez lorsque vous voulez une nouvelle instance pour chaque requête HTTP entrante (généralement utilisé pour manipuler des données par requête).

En comprenant et en appliquant les différentes portées des fournisseurs, vous pouvez gérer efficacement le cycle de vie et le partage des instances dans votre application NestJS en fonction des besoins spécifiques de chaque service ou composant.

1. **Application configuration**
2. **Introduction to the config module**

**Une image contenant texte, capture d’écran, Graphique, Caractère coloré

Description générée automatiquement**

The **@nestjs/config** module in NestJS is a powerful utility that simplifies the management and access to configuration variables and environment-specific settings within a NestJS application. It provides a clean and structured way to handle environment variables, configuration files, and other settings needed for the application's runtime behavior.

Key features and functionalities of the **@nestjs/config** module include:

1. **Environment Variables:** It allows you to easily access environment variables across your application. These variables are typically used for sensitive or environment-specific configuration settings like database credentials, API keys, etc.
2. **Configuration File Parsing:** The module supports the parsing of various types of configuration files such as **.env**, **.json**, **.yaml**, **.properties**, etc., making it flexible to use different formats based on your preferences or project requirements.
3. **Hierarchical Configuration:** NestJS Config module supports hierarchical configurations, enabling you to organize settings into different environments (development, production, testing, etc.) or sections within your configuration files.
4. **Typed Configuration:** It provides a way to define and validate typed configurations using TypeScript interfaces or classes, ensuring type safety when working with configuration values across the application.

To utilize the **@nestjs/config** module in your NestJS application, you typically perform the following steps:

**Installation:** Install the module and its dependencies via npm or yarn:

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

Description générée automatiquementModule Registration:** Import the **ConfigModule** into your application root module (**AppModule**) to make the configuration service available throughout the application.

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

Description générée automatiquement**

**Usage:** Access configuration values within your services, controllers, or other components using the **ConfigService** provided by NestJS.

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

Description générée automatiquement**

**Configuration File(s):** Optionally, create and manage configuration files (such as **.env**, **.json**, etc.) to store environment-specific settings or sensitive information.

By using the **@nestjs/config** module, developers can maintain a clean and manageable approach to handling configurations, ensuring flexibility, type safety, and easy access to environment variables or application settings across the NestJS application.

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

Description générée automatiquement**

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

Description générée automatiquement**

1. **Custom Environment file path**

ConfigModule.*forRoot*({  
 envFilePath: '.environment',  
}),

The snippet **ConfigModule.forRoot({ envFilePath: '.environment' })** is a configuration setup within the **@nestjs/config** module used in NestJS applications. Let's break down this specific part:

* **ConfigModule.forRoot()** is a method provided by the **@nestjs/config** module specifically used to configure and initialize the configuration module for the NestJS application.
* The method accepts an object with configuration options as its argument.
* Within this object, **envFilePath** is one of the configuration options specified.
* **envFilePath** is used to indicate the path to the environment file where environment-specific variables and configuration settings are stored.
  + In this case, **envFilePath: '.environment'** suggests that the configuration module will look for an environment file named **.environment**. The file path is relative to the root directory of the NestJS application.
  + This file, if found, will contain environment-specific variables such as database connection strings, API keys, and other settings required for different environments (e.g., development, testing, production).

By providing this configuration (**envFilePath: '.environment'**), the **@nestjs/config** module is instructed to load and parse configuration settings from the specified environment file during the application's initialization, making those settings available throughout the application where the **ConfigService** from **@nestjs/config** is used.

ConfigModule.*forRoot*({  
 ignoreEnvFile: true,  
}),

When **ignoreEnvFile** is set to **true** as shown (**ignoreEnvFile: true**), it instructs the **@nestjs/config** module to ignore the environment file during the application's initialization.

1. **Schema validation**

**Joi** is a powerful schema description language and data validator for JavaScript. It's often used for validating and sanitizing input data, especially in applications where ensuring data integrity and validation is crucial.

In the context of a NestJS application, **@nestjs/config** integrates with **Joi** to allow validation of configuration settings loaded through environment files or other sources. Here's an example of using **Joi** for validation within **@nestjs/config**:

1. **Setting up Joi validation in NestJS with @nestjs/config:**

**npm I @hapi/joi**

**npm i --save-dev @types/hapi\_\_joi**

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

Description générée automatiquement**

**Joi** is a powerful schema description language and data validator for JavaScript. It's often used for validating and sanitizing input data, especially in applications where ensuring data integrity and validation is crucial.

In the context of a NestJS application, **@nestjs/config** integrates with **Joi** to allow validation of configuration settings loaded through environment files or other sources. Here's an example of using **Joi** for validation within **@nestjs/config**:

1. **Setting up Joi validation in NestJS with @nestjs/config:**

typescriptCopy code

import { ConfigModule } from '@nestjs/config'; import \* as Joi from 'joi'; @Module({ imports: [ ConfigModule.forRoot({ validationSchema: Joi.object({ // Define your validation rules for configuration settings DB\_HOST: Joi.string().required(), DB\_PORT: Joi.number().default(5432), // Add more configuration keys and their validation rules as needed }), }), // other modules... ], // other module configurations... }) export class AppModule {}

In this example:

* **ConfigModule.forRoot()** accepts an object with **validationSchema** property.
* **validationSchema** is defined using **Joi.object()** to create a schema for validating configuration settings.
* Each configuration key (e.g., **DB\_HOST**, **DB\_PORT**) is associated with its respective validation rule (e.g., **Joi.string().required()**, **Joi.number().default(5432)**).

1. **Explanation of Joi Validation Rules:**

* **Joi.string().required()**: Specifies that the configuration value for **DB\_HOST** should be a string and it is required (cannot be empty or undefined).
* **Joi.number().default(5432)**: Specifies that the configuration value for **DB\_PORT** should be a number and sets a default value of **5432** if the **DB\_PORT** environment variable is not defined.

1. **Handling Validation Errors:**

When the application starts and the configuration is loaded, **@nestjs/config** will automatically validate the configuration against the defined Joi schema. If a configuration value doesn't match the defined validation rules, it will throw a validation error, preventing the application from starting.

Validating configuration using Joi within **@nestjs/config** helps ensure that the loaded configuration adheres to predefined rules, preventing potential runtime errors due to missing or incorrect configuration settings.

ConfigModule.*forRoot*({  
 validationSchema: Joi.object({  
 DATABASE\_HOST: Joi.required(),  
 DATABASE\_PORT: Joi.number().default(5432),  
 }),  
}),

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

Description générée automatiquement**

1. **Using the config service**

Once the configuration has been loaded and validated using Joi through **@nestjs/config**, you can access the validated configuration values throughout your application using the **ConfigService** provided by NestJS. Here's an example of how to use the **ConfigService** to access validated configuration settings:

Assuming you've defined and validated your configuration settings using **ConfigModule.forRoot()** and **validationSchema** as explained in the previous examples:

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

Description générée automatiquement

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

Description générée automatiquement

We can access and also define it if we don’t have It :

const dataBase = this.configService.get<string>(  
 'DATABASE\_HOST',  
 'localhost2',  
);

Explanation:

1. **@Injectable()** decorator is used to make the **YourService** class injectable.
2. **ConfigService** is injected into the service via its constructor.
3. Inside the service method (**getDatabaseConfig()** in this case), **this.configService.get()** is used to retrieve configuration settings by their keys (in this example, **DB\_HOST** and **DB\_PORT**).

By using the **ConfigService**, you can easily access validated configuration settings throughout your NestJS application. It handles the retrieval of configuration values based on the keys defined in your configuration schema, ensuring type safety and providing default values if specified in the schema.

Remember, the **ConfigService** methods (**get()**, **getInt()**, **getBoolean()**, etc.) are used to retrieve configuration values, and their return types are inferred based on the defined schema in the **ConfigModule.forRoot()** configuration. This helps maintain type safety and prevent runtime errors related to missing or incorrect configuration keys.

1. **Custom configuration Files**

NestJS typically relies on configuration files (such as **.env**, **.json**, **.yaml**, etc.) or modules like **@nestjs/config** to manage configurations. However, if you prefer a configuration file like **app.config.js**, you can create a custom configuration approach and load values from this file.

Here's an example of how you can achieve custom configuration using an **app.config.js** file:

**Create the app.config.js File:**

Create a file named **app.config.js** at the root of your NestJS project. This file will hold your custom configuration.

App.config.ts

export default () => ({  
 environments: *process*.env.NODE\_ENV || 'development',  
 database: {  
 host: *process*.env.DATABASE\_HOST,  
 port: parseInt(*process*.env.DATABASE\_PORT, 10) || 5432,  
 },  
});

**Use ConfigModule.forRoot() with the Loader Function:**

Incorporate the **loadAppConfig** function into the **ConfigModule.forRoot()** method to load the configurations.

app.module.ts



**Access Configuration Values in Your Service or Controller:**

Inject and use the configuration values as needed in your services or controllers.

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

Description générée automatiquement

This approach uses the **ConfigModule.forRoot()** method with a custom loader function to load configurations from the **app.config.js** file. The configurations become available through the **ConfigService** provided by the **@nestjs/config** module. Adjust the paths and structure according to your project setup to ensure the correct loading of the **app.config.js** file.

1. Configuration namespace and Partial Registration

In NestJS, you can use configuration namespaces and partial registration to manage different sets of configuration values for specific modules or sections of your application. This allows you to organize and encapsulate configurations neatly.

In NestJS, the **registerAs** feature is part of the **@nestjs/config** module, which allows you to register configurations under specific names or aliases. This is useful when you want to load multiple configurations or configurations with different names, and you need to access them using these aliases.

Here's an example of how to use **registerAs**:

Let's assume you have separate configurations for different services or parts of your application.

1. **Create Configuration Files:**

Coffes.config.ts

import { registerAs } from '@nestjs/config';  
  
export default registerAs('coffees', () => ({  
 foo: 'bar',  
}));

**2-Register Configurations :**

@Module({  
 imports: [  
 TypeOrmModule.*forFeature*([Coffee, Flavor]),  
 ConfigModule.*forFeature*(*coffesConfig*),  
 ],

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

Description générée automatiquement**

**3-Accessing Configurations by Alias:**

In your services or controllers, access the configurations by their registered aliases.

export class CoffeesService {  
 constructor(  
 @InjectRepository(Coffee)  
 private readonly cofeeRepository: Repository<Coffee>,  
 @InjectRepository(Flavor)  
 private readonly flavorRepository: Repository<Flavor>,  
 // private readonly connection: Connection,  
 @Inject('COFFEE\_BRANDS') coffeBrands: string[],  
 private readonly configService: ConfigService,  
 @Inject(*coffesConfig*.KEY)  
 private readonly coffeesConfiguration: ConfigType<typeof *coffesConfig*>,  
 ) {  
 *console*.log('Non-class-based Provider Tokens', coffeBrands);  
 // Access validated configuration settings using the ConfigService  
 const dataBase = this.configService.get<string>(  
 'database.host',  
 'localhost',  
 );  
 *console*.log('ConfigService', dataBase);  
 const coffesConfig0 = this.configService.get('coffees');  
 *console*.log('coffesConfig0', coffesConfig0);  
 *console*.log('coffesConfig1', coffeesConfiguration);  
 }

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

Description générée automatiquement**

The line **@Inject(coffeesConfig.KEY) private readonly coffeesConfiguration: ConfigType<typeof coffeesConfig>** is a way to inject a strongly-typed configuration directly into a NestJS service or controller. It uses the **@Inject()** decorator to inject the configuration defined in a configuration file or module.

Let's break down this line:

1. **@Inject(coffeesConfig.KEY)**: The **@Inject()** decorator is used to inject dependencies into a class constructor. In this case, it injects the configuration defined by **coffeesConfig.KEY**. This allows for direct injection of the configuration object into the service or controller.
2. **private readonly coffeesConfiguration: ConfigType<typeof coffeesConfig>**: This line declares a private readonly property named **coffeesConfiguration** of type **ConfigType<typeof coffeesConfig>**. This type represents the strongly-typed structure of the configuration defined in the **coffeesConfig** file.

Using **@Inject(coffeesConfig.KEY)** allows you to directly inject and access the configuration object within a specific service or controller, providing direct access to configuration values. However, it is an alternative approach to using the **ConfigService** provided by the **@nestjs/config** module.

On the other hand, **this.configService** refers to an instance of the **ConfigService** provided by NestJS' configuration module (**@nestjs/config**). It is a more general approach that allows accessing configurations globally throughout the application.

Both methods have their use cases:

* Using **@Inject(coffeesConfig.KEY)** for specific services/controllers allows for more direct access to a strongly-typed configuration object within that class.
* Using **this.configService** gives you access to the configuration service throughout the application, allowing you to fetch configurations wherever needed, providing more flexibility and global accessibility.

Choose the approach that fits best with your application's design and the level of granularity you need for accessing configurations within different parts of your NestJS application.

1. Configuration namespace and Partial Registration

**App.module.ts**

TypeOrmModule.*forRootAsync*({  
 useFactory: () => ({  
 type: 'postgres',  
 host: process.env.DATABASE\_HOST,  
 port: +process.env.DATABASE\_PORT,  
 username: process.env.DATABASE\_USER,  
 password: process.env.DATABASE\_PASSWORD,  
 database: process.env.DATABASE\_NAME,  
 autoLoadEntities: true,  
 synchronize: true,  
 }),  
}),

Certainly! The line you've provided configures a TypeORM module in NestJS using the **forRootAsync()** method. This method is used when you want to provide the configuration for a module asynchronously, typically because the configuration depends on values that are resolved at runtime, such as environment variables or values obtained from external sources.

Explanation of each part:

1. **TypeOrmModule.forRootAsync()**: This method is used to asynchronously configure the TypeORM module in NestJS.
2. **useFactory**: It is a property of **forRootAsync()** that expects a factory function. This function is called to create and return the configuration object for the TypeORM module.
3. Configuration Object: Inside the **useFactory** function, you are constructing the TypeORM configuration object. Here's a breakdown of the properties:
   * **type**: Specifies the database type (in this case, 'postgres' for PostgreSQL).
   * **host**, **port**, **username**, **password**, **database**: These properties are set based on environment variables (**process.env**) for the database connection details. They can be set in your environment or config files.
   * **autoLoadEntities**: A boolean indicating whether entities (database models) should be loaded automatically. Setting this to **true** means TypeORM will automatically load entities from the specified directories.
   * **synchronize**: A boolean flag indicating whether TypeORM should automatically synchronize the database schema with the entity metadata. Be cautious using this in production as it can cause data loss. It's usually recommended for development purposes.

This configuration approach allows you to dynamically configure the TypeORM module based on environment variables or any other logic that resolves values at runtime. The factory function is called when the module is initialized, and it generates the TypeORM configuration object based on the provided logic, which enables dynamic and flexible configuration of the database connection in your NestJS application.