**NestJs**

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

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

Key features of NestJS include:

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

Reasons to use NestJS:

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

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

1. **What is the Nest CLI**

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

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

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

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

1. **Getting Started**
   1. **First Step**

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

Description générée automatiquement

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

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

Description générée automatiquement

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

Description générée automatiquement

* 1. **Controllers**

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

Description générée automatiquement**

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

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

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

For example:

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

Description générée automatiquement**

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

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

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

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

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

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

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

Description générée automatiquement

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

Description générée automatiquement

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

Description générée automatiquement

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

Description générée automatiquement**

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

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

Description générée automatiquement

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

Description générée automatiquement

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

Description générée automatiquement

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

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

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

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

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

Description générée automatiquement

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

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

Description générée automatiquement

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

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

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

Description générée automatiquement

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

Description générée automatiquement

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

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

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

Description générée automatiquement