NestJS Notebook

- A Student Notebook Covering Fundamentals, Real-world Use Cases, and Interview Preparation

---

Introduction

❓ What is NestJS?

NestJS is a progressive Node.js framework used for building efficient and scalable server-side applications. It is built with **TypeScript** and takes a lot of inspiration from **Angular**; so if you have worked with Angular before, you will feel right at home here.

One of the things that makes NestJS stand out is its **modular architecture**, which basically means your code is organized into neat, self-contained pieces. This makes it a lot easier to manage as your project grows.

❓ Why Do We Even Need NestJS?

The short answer; to make backend development less chaotic.

NestJS gives us a **structured, modern way** to build applications that are easy to scale and test. It also solves a problem that developers kept running into with traditional Express apps: the lack of structure.

NestJS vs ExpressJS *(Common Interview Question)*

This is something that comes up a lot in interviews, so it is worth understanding clearly.

ExpressJS is what is called an ***unopinionated*** framework. But what does "unopinionated" actually mean?

It means Express gives you **zero rules** on how to structure your project. You are free to do things however you like; which sounds great at first, but when your project starts growing, things get messy fast. There is no enforced architecture, so different developers on the same team might organize things completely differently.

NestJS, on the other hand, is "***opinionated***". It comes with a clear structure and a bunch of built-in tools right out of the box means full TypeScript support available immediately after installation; things like:

- **Dependency Injection**; a design pattern that manages how different parts of your app are connected
- **Middleware** support
- **Database integration**
- **REST API** support
- **GraphQL** support
- And more...

📌 So, **in short**: Express is great for small, flexible projects. But for large, production-grade applications, NestJS wins because it keeps everything organized and maintainable.

Feature	ExpressJS	NestJS
Language	JavaScript	TypeScript
Architecture	Unopinionated (no structure)	Opinionated (structured)
Scalability	Harder to scale	Built for scalability
Built-in tools	Minimal	Rich (DI, GraphQL, etc.)
Learning Curve	Lower	Moderate

💰 Benefits of NestJS

- ☑ Full TypeScript support available immediately after installation (no extra configuration required)
- ☑ Built-in Dependency Injection system
- ☑ Easy integration with databases, WebSockets, GraphQL, and microservices
- ☑ Scalable and maintainable codebase
- ☑ Active and growing community

❓ What We will Cover in This Notebook

- Beginner to Advanced NestJS concepts
- Interview question preparation
- Real-world scenarios and use cases
- Source code included for every topic

Let us get started! 🎉

Installation

🛠️ Step 1: **Install Node.js**

Head over to the official Node.js website and download the **LTS (Long Term Support)** version:

🔗 https://nodejs.org/en/download

Tip: Always go with the LTS version; it is the most stable one and recommended for most projects.

Once downloaded, run the installer and just click **Next → Next → Next → Finish**. Make sure **no extra checkboxes are checked** during the installation process.

❓ But wait; why do we need Node.js in the first place?

JavaScript was originally designed to run only inside browsers. That was fine for frontend stuff, but it meant you couldn't use JavaScript to build backend/server-side applications.

Node.js changed that. It is a runtime environment that lets JavaScript run *anywhere*; not just in a browser. So now we can use JavaScript (and TypeScript) to build full backend applications, APIs, and more.

Verify Node.js installation:

```
node - v
# Expected output example: v20.11.0
```

🛠️ Step 2: **npm** (Node Package Manager)

The good news? When you install Node.js, npm comes(install) along with it **automatically.** You don't need to install it separately.

❓ What is npm and why do we need it?

npm is basically an **online store** for **code packages**. Instead of writing everything from scratch, you can install pre-built packages (libraries/tools) that other developers have shared. NestJS itself is installed through npm.

Verify npm installation:

```
npm - v       # Expected output example: 10.2.4
```

Step 3: Install NestJS

Now that Node.js and npm are ready, let us install the **NestJS CLI** (Command Line Interface) globally on your machine. The -g flag means it will be available everywhere, not just in one project folder.

```
npm i - g @nestjs/cli
```

Step 4: Create a New NestJS Project

Navigate to the folder(or, `mkdir projectfolder # to create the folder)` where you want to create your project, then run:

```
cd projectfolder
nest new project-name
```

During setup, NestJS will ask:

```
Which package manager would you ❤ to use ?
👉 Select npm
```

Step 5: Open and Run the Project

```
mkdir project-name   # Creating a new folder.
cd project-name      # Navigate into your newly created project
code.                # Open it in VS Code
npm run start        # Start the development server
```

Once it is running, open your browser and go to: http://localhost:3000/

You should see **"Hello World!"**; that means everything is working perfectly!

Installation done! Now let's dig into the actual code. 🎉

File & Folder Structure

When you create a new NestJS project, it automatically generates a bunch of files and folders. At first it looks overwhelming, but once you understand what each piece does, it all makes sense. Let's break it down:

```
project-name /
├── dist /              # Build output folder. After running`npm run build`, TypeScript files from src / are compiled into JavaScript and stored here for execution.
├── node_modules /         # Stores all installed npm packages and their dependencies.Created automatically when `npm install` is executed.
├── src /                  # Main development folder containing the application's TypeScript source code
│   ├── app.controller.spec.ts  # Unit test file for app.controller.ts, written with Jest to test controller functionality
│   ├── app.controller.ts       # Controller that handles incoming HTTP requests and sends responses to the client
│   ├── app.module.ts           # Root module that organizes the application by registering controllers and services
│   ├── app.service.ts          # Service class where business logic and data processing are implemented
│   └── main.ts                 # Application entry point that bootstraps the NestJS app and starts the HTTP server
├── test /          # Contains end - to - end tests that simulate real user requests and verify complete application behavior
│   ├── app.e2e - spec.ts
│   └── jest - e2e.json
├── .gitignore            # Lists files and folders that Git should ignore(e.g.node_modules and build files)
├── .prettierrc           # Configuration file for Prettier to keep code formatting consistent across the project
├── eslint.config.mjs     # ESLint configuration used to detect coding errors and enforce coding standards
├── nest - cli.json         # Configuration file used by the NestJS CLI for project generation, build, and development tasks
├── package - lock.json     # Automatically generated file that records the exact versions of installed dependencies
├── package.json        # Core project configuration file containing project info, dependencies, and npm scripts
├── README.md             # Documentation explaining the project, setup steps, and usage instructions
├── tsconfig.build.json   # TypeScript configuration specifically used when building the production version
└── tsconfig.json         # Main TypeScript configuration that controls how TypeScript code is compiled
```

Quick Summary of the Most Important Parts

The folder you will spend most of your time in is **src/** — that is where all your actual application code lives.

- **main.ts**: this is where everything starts. It boots up the app.
- **app.module.ts**: think of this as the *root organizer*. It ties everything together.
- **app.controller.ts**: handles incoming requests (like when someone hits an API endpoint).
- **app.service.ts**: where the actual logic lives (calculations, database calls, etc.).

The rest of the files are mostly config files that you will rarely need to touch, especially when you are just starting out.

Interview Questions

Q1. What is the difference between package.json and package-lock.json?

package.json is the main configuration file of your project. It holds:

- Your project name, version, and description
- The list of dependencies your project needs
- npm scripts like npm run start, npm run build, etc.

Think of it as the blueprint of your project; it tells npm which packages your project needs and the allowed version range, but the exact version that gets installed is recorded in package-lock.json.

package-lock.json goes one step deeper. It **locks** the exact version of every single package (and every sub-dependency of those packages) that was installed. It is auto-generated by npm and you never edit it manually.

So, in simple words:

- package.json → *"I need Express version 4 or above"*
- package-lock.json → *"I installed Express version 4.18.2 specifically"*

This ensures that when someone else clones your project and runs npm install, they get the **exact same versions** you used; no surprises.

Q2. Which is most important among node_modules/, package-lock.json, and package.json? Also, what is npm i and why do we use it?

package.json is the most important one. Here is why:

- node_modules/ is a **generated** folder. It can be deleted and recreated anytime.
- package-lock.json is also **auto-generated** by npm.
- But package.json? That is the one you actually write and maintain. Without it, npm has no idea what to install.

What is npm i (npm install)?

```
npm i
# or the full version:
npm install
```

When you run this command, npm reads your package.json, looks at all the listed dependencies, and **downloads and installs all of them** into the node_modules/ folder. Also creates package-lock.json if it does not already exist.

❓ Why do we use it?

Because node_modules/ is never pushed to GitHub (it is listed in .gitignore; it is too large and unnecessary to share). So, whenever someone clones your project, the first thing they do is run **npm i** to *restore all the packages locally. And we should push package-lock.json to GitHub.*

In short:

You share the recipe (package.json), not the ingredients (node_modules/). npm i goes and fetches the ingredients and package-lock.json for you.

Structure understood! Now let's look at what's inside these files. 🗃

NestJS Controllers

❓ What are Controllers?

Controllers are basically the **front door** of your application. Whenever someone sends an HTTP request to your server, such as GET, POST, PUT, DELETE, etc the controller is the first part of the application that receives it.

Think of it this way:

- The **client** (browser, mobile app, Postman) sends a request.
- The **controller** catches it and figures out what to do.
- It then talks to the **service** (where the actual business logic lives).
- And sends a response back to the client.

So, controllers act as the **bridge between the client and your business logic**.

❓ Why Are Controllers Important?

- They keep your API endpoints(A specific URL where a client sends a request to interact with a server resource) **organized and modular**; each controller handles one specific area (users, products, orders, etc.).
- They enforce **separation of concerns**; routing is the controller's job, logic is the service's job.
- They make your code **scalable and readable**; easy to find what you're looking for, easy to add new endpoints.

❓ What are Decorators?

Before we look at the actual controller code, we need to understand **decorators**; because NestJS uses them *everywhere*.

A decorator is a **special function that starts with @** and attaches metadata to a class or method. It basically tells NestJS:

> *"Hey, treat this class/method in this specific way."*

For example:

- @Controller() → tells NestJS *"this class is a controller"*
- @Get() → tells NestJS *"this method handles GET requests"*
- @Post() → handles POST requests, and so on...

Decorators are used for routing, dependency injection, validation, and a lot more. Once you get used to them, they feel very natural.

💣 Creating a Controller

Inside the src/ controllerNameFolder, controller files follow this naming convention:

controllerName.controller.ts

🕷️ Generate a Controller Using the NestJS CLI

You *could* create the file manually, but the **recommended way** is to use the NestJS CLI:

```
nest g controller user # nest generate controller user
```

❓ Why use the CLI instead of creating the file manually?

Because the CLI doesn't just create the file; it also **automatically registers the controller** in **`app.module.ts`**. And that registration step is critical. If a controller isn't registered in the module, NestJS simply won't recognize it and it will not work.

Running the above command will generate **two** files:

src/user/user.controller.spec.ts ← unit test file (auto-generated)

src/user/user.controller.ts ← your actual controller file

☠️ Controller Code

src/user/user.controller.ts

```
// Importing the necessary decorators from the NestJS common package.
// @Controller defines this class as a controller.
// @Get defines a route handler that responds to GET requests.
import { Controller, Get } from '@nestjs/common';

// @Controller('user') tells NestJS that this controller will handle
// all incoming requests that start with the '/user' route.
// For example: http://localhost:3000/user
@Controller('user')
export class UserController {

    // @Get() tells NestJS that this method should be triggered
    // specifically when a GET request is made to the '/user' endpoint.
    @Get()
    getUser() {
        // For now, we're just returning a plain string as the response.
        // Later, this is where we'd call a service to fetch real data.
        return 'This is the user endpoint';
    }
}
```

Running the App

```
npm run start # Start the server normally
```

```
# Then open your browser and visit: http://localhost:3000/user

# You should see: This is the user endpoint


       npm run start:dev # Start in development/watch mode (recommended during development)

# This watches for any file changes and automatically restarts the server.

# Super useful — no need to stop and restart manually every time you edit something.
```

❓ **How it all connects** — Quick Visual

```
Client Request(GET / user)
        ↓
UserController        ← @Controller('user') + @Get()
        ↓
(calls service)       ← business logic lives here
        ↓
Response sent back to client
```

Controllers done! Next up; Services, where the real logic happens. 🔧

NestJS Services

❓ What are Services?

A service is just a **TypeScript class** where you write your actual logic; things like fetching data, doing calculations, calling external APIs, etc. Instead of stuffing all that logic inside your controller, you move it into a service to keep things clean and reusable.

Services are marked with the **@Injectable()** decorator, which is how NestJS knows it can manage and inject them wherever they are needed.

📌 Quick Note on Two Important Terms

@Injectable() and Providers:

- @Injectable() marks a class as a **provider**
- A **provider** is simply a class that NestJS manages and can automatically *inject* into other classes; that is the whole idea behind Dependency Injection

Dependency Injection (DI):

- DI means NestJS automatically **creates**(object) **and supplies** a class wherever it is needed
- You don't have to manually write new ProductsService() anywhere; NestJS handles that for you behind the scenes

❓ Why Do We Use Services?

- To **separate logic from controllers**; controllers handle routing, services handle logic
- Makes code **modular, clean, and testable**
- Services can be **reused** across multiple controllers

- Keeps your app **organized and scalable** as it grows

📌 Important Points to Remember

- Always use @Injectable() on your service classes
- Inject services into controllers using **constructor injection**
    - Constructor Injection = receiving a service inside the controller's constructor() so NestJS can automatically provide it
- Services are a core part of the **Dependency Injection (DI)** system
- All your heavy lifting, data fetching, calculations, API calls; goes inside services

💣 Creating a Service

Inside the src/ serviceNameFolder, service files follow this naming convention:

serviceName.service.ts

🕷️ Generate a Service Using the NestJS CLI

```
nest g s products # nest generate service products
```

This will generate **two** files:

src/products/products.service.spec.ts ← unit test file (auto-generated)

src/products/products.service.ts ← your actual service file

☠️ Service Code

src/products/products.service.ts

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

// @Injectable() marks this class as a provider/service.
// This tells NestJS it can manage this class and inject it into other classes automatically.
@Injectable()
export class ProductsService {

    // 'private' means this data is only accessible inside this class — not from outside.
    // This is a simple in-memory array acting as our data source for now.
    private products = [
        { id: 1, name: 'Raspberry Pi', price: 15400 },
        { id: 2, name: 'BW16', price: 1220 },
        { id: 3, name: 'Arduino Uno', price: 500 },
    ];
    // Each object is a product with three key-value pairs:
    // id    → unique identifier for the product
    // name  → name of the product
    // price → price of the product
    //
    // Example: products[2].name → 'Arduino Uno'

    // Returns the entire products array
    getAllProducts() {
        return this.products; // 'this' refers to the current instance of this class
    }

    // Accepts an id (number) and finds the matching product
    getProductById(id: number) {
        return this.products.find(product => product.id === id);
        // .find() loops through the array and returns the first match
        // Return type: { id: number; name: string; price: number } | undefined
```

```
        // → returns the product object if found, or 'undefined' if not found
    }
}
```

Generate a Controller Using the NestJS CLI

```
nest generate controller products
```

This will generate **two** files:

src/products/products.controller.spec.ts ← unit test file (auto-generated)

src/products/products.controller.ts ← your actual controller file

Controller Code

src/products/products.controller.ts

```
import { Controller, Get, Param } from '@nestjs/common';
// @Controller  → marks this class as a controller
// @Get         → handles GET HTTP requests
// @Param       → extracts route parameters from the URL

import { ProductsService } from './products.service';
// Importing our ProductsService so we can use its methods inside this controller

@Controller('products')
// Sets the base route for this controller → all routes here start with /products
export class ProductsController {

    // Constructor Injection — NestJS automatically creates and provides
    // an instance of ProductsService here. We don't need to do 'new ProductsService()' manually.
    // 'private'  → this property is only accessible within this class
    // 'readonly' → this property cannot be reassigned after it's set
    constructor(private readonly productsService: ProductsService) { }

    @Get()
    // Handles GET requests to → /products
    getProducts() {
        return this.productsService.getAllProducts();
        // Calls the service method and returns the full list of products
    }

    @Get(':id')
    // Handles GET requests to → /products/1, /products/2, etc.
    // The ':' means this part of the URL is a dynamic route parameter (a variable)
    getProduct(@Param('id') id: string) {
        // @Param('id') extracts the 'id' value from the URL
        // URL parameters are always strings by default, so we convert it to a number
        return this.productsService.getProductById(Number(id));
        // Return type: { id: number; name: string; price: number } | undefined
    }
}
```

Running the App & Expected Output

```
npm run start: dev
```

http://localhost:3000/products

```
[
    { "id": 1, "name": "Raspberry Pi", "price": 15400 },
    { "id": 2, "name": "BW16", "price": 1220 },
    { "id": 3, "name": "Arduino Uno", "price": 500 }
]
```

http://localhost:3000/products/2

```
{ "id": 2, "name": "BW16", "price": 1220 }
```

Interview Questions: NestJS Services

Q1. What is a Service in NestJS?

A service is a class decorated with @Injectable() that contains the business logic of your application; things like data processing, calculations, or database calls. It keeps logic separated from controllers, making the code clean and reusable.

Q2. What is the @Injectable() decorator?

@Injectable() marks a class as a **provider**, meaning NestJS can manage its lifecycle and automatically inject it into any class that needs it; without you having to manually create an instance/object.

Q3. What is Dependency Injection and how does NestJS use it?

Dependency Injection (DI) is a design pattern where a class receives its dependencies from the outside rather than creating them itself. In NestJS, you simply declare the service in the constructor, and NestJS automatically creates and injects the instance for you.

```
// NestJS handles the creation of ProductsService automatically
constructor(private readonly productsService: ProductsService) { }
```

private means the service is only accessible inside the class. It also allows TypeScript to automatically create and assign the class property from the constructor parameter.

Q4. What is the difference between a Controller and a Service?

	Controller	Service
Job	Handle HTTP requests & routes	Contain business logic
Decorator	@Controller()	@Injectable()
Should have logic?	No, keep it thin	Yes, all logic goes here
Can be reused?	Not typically	Yes, across multiple controllers

Q5. Why should we never write business logic inside a Controller?

Controllers are only meant for **routing**, receiving requests and sending responses. If you put logic there, the code becomes hard to test, maintain, and reuse. Services are designed specifically to hold that logic, keeping everything clean and separated.

Services down! Next up; Modules, the glue that holds everything together.

Modules in NestJS

What are Modules?

A module is basically a **container** that groups together everything related to a specific feature; it is controllers, services, and providers. It is a core part of how NestJS is structured, and every NestJS application has **at least one module**.

✈ Real-World Analogy **(The Restaurant)**

Think of it like a restaurant:

Real World	NestJS Equivalent
🏠 Restaurant (Container)	Module
🤵 Waiter	Controller
👨‍🍳 Kitchen Chef	Service

The **waiter** (controller) takes the customer's order (HTTP request) and passes it to the **kitchen chef** (service) who prepares the food (processes the logic). Both the waiter and the kitchen work together inside the **restaurant** (module); that is the container that holds them.

In NestJS, that container is the **Module**.

📌 **The Parent Module** (app.module.ts)

src/app.module.ts

This is the **root module**; the main container of your entire application. Every other module you create should be **imported** here to become part of the app.

Best Practice: Divide your application into separate **feature** modules (e.g., EmployeeModule, ProductsModule) and import them all into the parent app.module.ts. This keeps your code organized, maintainable, reusable, and scalable.

💣 **Creating a Module**

Inside the src/ moduleNameFolder, module files follow this naming convention:

moduleName.module.ts

🕷 **Generate a Module Using the NestJS CLI**

```
nest g module employee
```

This creates a new folder and module file:

src/employee/ ← new feature folder

🕷 **Generate the Controller and Service for the Same Feature**

Now create the controller and service for the employee feature using the **same name**:

```
nest g controller employee
nest g service employee
```

❓ **Why use the same name?**

Because NestJS CLI is smart enough to detect the existing `employee/` folder and place all files inside it automatically; keeping everything for one feature neatly together in one place.

Your folder will now look like this:

```
src /
└── employee /
```

```
    ├── employee.module.ts          ← module(container)
    ├── employee.controller.ts      ← handles HTTP requests
    ├── employee.controller.spec.ts ← auto - generated test file
    ├── employee.service.ts         ← business logic
    └── employee.service.spec.ts    ← auto - generated test file
```

And the CLI also **automatically imports EmployeeModule into app.module.ts** — so you don't have to do it manually. Without this import, the feature simply won't work in the application.

☠️ Controller Code

src/employee/employee.controller.ts

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

// Sets the base route for this controller → /employee
@Controller('employee')
export class EmployeeController {

    // Handles GET requests to → /employee
    @Get()
    getAllEmployees() {
        // Returning a simple string for now just to test the feature is working
        return 'Employee data fetched successfully';
    }
}
```

🚢 Running the App & Expected Output

npm run start:dev

http://localhost:3000/employee

Output: Employee data fetched successfully

❓ How It All Fits Together

```
AppModule(app.module.ts)   ← Parent module, root of everything
    │
    └── EmployeeModule     ← Feature module, imported into AppModule
            │
            ├── EmployeeController  ← Handles / employee routes
            └── EmployeeService     ← Business logic for employees
```

Every feature you build in NestJS follows this same pattern; **Module → Controller → Service**. Once you get this flow, everything else starts to click.

🫀 Interview Questions (NestJS Modules)

Q1. What is a Module in NestJS?

A module is a class decorated with @Module() that acts as a container for a specific feature of the application. It groups related controllers, services, and providers together, making the codebase organized and modular.

Q2. Why do we need Feature Modules instead of putting everything in AppModule?

If everything was crammed into one module, the application would become impossible to maintain as it grows. Feature modules let you isolate each concern (employees, products, orders) into its own container. This makes the code reusable, testable, and scalable; and it is considered the standard best practice in NestJS.

Q3. What happens if you forget to import a feature module into AppModule?

The feature simply will not work. NestJS won't recognize any of the controllers or services inside that module because they were never registered in the application's module tree. Always make sure every feature module is imported into app.module.ts (the CLI does this automatically when you use nest g module).

Modules covered! You now understand the full NestJS triangle; Module → Controller → Service. ⏢

NestJS Architecture

Overview

NestJS follows a **modular architecture**; meaning for every feature you build, you create a separate module to contain it. This keeps your codebase clean, reusable, and easy to scale as the project grows.

📌 The golden rule:

> **One feature = One module** → with its own controller, service, and providers.

💣 **The Big Picture** (How a Request Flows)

Here is what happens from the moment a user makes a request to the moment they get a response back:

```
Client(User)
      |
      |  HTTP Request(e.g., GET / products)
      ▼
Controller          ← receives the request, calls the right service method
      |
      ▼
Service             ← processes the business logic
      |
      ▼
Provider / DB       ← data is fetched or processed
      |
      ▼
Controller          ← gets the result back from service
      |
      |  HTTP Response(e.g., JSON data)
      ▼
Client(User)
```

🛠️ **Important Terminologies**

1. 🥷 Client (User): *Generates the Request*

The client is whoever or whatever is sending a request to your application; could be a browser, a mobile app, or a tool like Postman.

- Triggers the whole process by hitting an endpoint [like /products, @Controller('products')]
- Doesn't know or care about what happens behind the scenes; it just waits for a response

2. 📋 Controller: *The Receptionist*

Think of the controller as a **receptionist at a front desk**. It doesn't solve your problem itself; it listens to what you need and directs you to the right person.

- Receives the client's HTTP request

- Figures out which service method to call
- Sends the response back to the client
- Does **not** contain any business logic; that is the service's job

3. ⚙️ **Service:** *The Worker*

This is where the **actual work happens**. The service contains all the business logic; calculating things, applying rules, fetching data, and so on.

- Does not deal with HTTP requests or responses directly
- Just focuses on logic and data processing
- Can be reused across multiple controllers

4. 🔌 **Provider:** *The Injectable Helper*

A provider is any class that can be **injected and reused** across the application. Services are the most common type of provider, but you can also have custom utility classes, factories, repositories, etc.

- Registered inside the module
- Made available to other classes via **Dependency Injection**

5. 📦 **Module:** *The Container*

The module is the **container** that groups everything for a specific feature together; it is controller, service, and providers.

- Organizes your app into self-contained features (e.g., ProductModule, UserModule)
- Every feature gets its own module
- All feature modules are imported into the root AppModule
- Keeps the app clean, scalable, and well-structured

6. 💉 **Dependency Injection (DI):** *The Auto-Supplier*

Dependency Injection is one of the most powerful concepts in NestJS. Instead of manually creating instances of services with new ServiceName(), NestJS **automatically creates and provides them** wherever they are needed.

- You just declare what you need in the constructor
- NestJS handles the rest
- Makes your code much easier to test and reuse

```
// You declare it — The object is created by NestJS and provides it automatically
constructor(private readonly productsService: ProductsService) { }
// The constructor does not create the object manually. It only declares that the class needs
ProductsService, and NestJS automatically creates and provides the instance when the class is
initialized.
```

7. 🎨 Decorators: The Metadata Labels

Decorators are **special functions that start with `@`** and tell NestJS how to treat a class, method, or parameter. They are used absolutely everywhere in NestJS.

Decorator	Purpose
@Module()	Marks a class as a module
@Controller()	Marks a class as a controller
@Injectable()	Marks a class as a provider/service
@Get(), @Post()	Defines HTTP route handlers
@Param(), @Body()	Extracts data from requests

🕷️ Putting It All Together:

```
AppModule(Root)
│
├── ProductModule
│       ├── ProductController(@Controller)   ← handles / products routes
│       ├── ProductService(@Injectable)      ← business logic
│       └── (Providers, if any)
│
├── UserModule
│       ├── UserController(@Controller)      ← handles / users routes
│       └── UserService(@Injectable)         ← business logic
│
└── EmployeeModule
        ├── EmployeeController(@Controller)  ← handles / employee routes
        └── EmployeeService(@Injectable)     ← business logic
```

In NestJS, each module focuses on a specific feature and keeps its related components together. Controllers handle incoming requests and routing, services contain the business logic, and NestJS connects everything automatically using Dependency Injection. This structured approach is the core design philosophy of NestJS.

🦞 **Interview Questions** (NestJS Architecture)

Q1. Describe the NestJS request lifecycle in simple terms.

When a client sends an HTTP request, it first hits the **Controller**, which acts like a receptionist; it receives the request and calls the appropriate **Service** method. The service processes the business logic (and may interact with a database or other providers), then returns the result back to the controller, which sends it as a response to the client.

Q2. What is the difference between a Provider and a Service in NestJS?

A **Service** is a specific type of **Provider**. All services are providers, but not all providers are services. A provider is any class decorated with @Injectable() that NestJS can manage and inject; this includes services, repositories, factories, and custom utility classes.

Q3. Why does NestJS use Dependency Injection(DI) instead of manually creating instances?

If you manually create objects using new, your classes become **tightly** connected(not flexible) to specific implementations. This tight coupling makes the code harder to test, harder to modify, and difficult to replace components when the application grows.

NestJS uses Dependency Injection (DI) to solve this problem. Instead of creating objects manually, a class simply declares the dependencies it needs, and NestJS automatically creates and provides those instances.

As a result, the code becomes **loosely** connected, easier to test, more modular, and easier to maintain as the application scales.

Q4. Can a NestJS application work without modules?

No. Every NestJS application must have at least one module; the root AppModule. Modules are fundamental to how NestJS organizes and bootstraps the application. Without them, there is no application.

Architecture done! You now have a solid mental model of how NestJS works end to end. 🏛

Dependency Injection in NestJS

❓ What is Dependency Injection?

Dependency Injection (DI) is a mechanism where NestJS automatically provides the required dependencies to your classes; without you having to create them manually yourself.

In plain terms: instead of writing **new SomeService**() every time you need a service, you just *declare* that you need it, and NestJS handles the rest.

By default, NestJS services are **singleton**, meaning NestJS creates one instance and reuses it everywhere. So, one constructor injection is enough to access that shared instance.

If you really need multiple instances

NestJS can create different instances using **special scopes** like:

- Scope.REQUEST (new instance per request)
- Scope.TRANSIENT (new instance every time it is injected)

But for most applications, **the default singleton instance is enough**.

❓ Why NestJS does this

Using a single shared instance:

- Saves memory
- Keeps state consistent
- Improves performance

❓ Why is DI Important?

- Makes code **reusable and clean**; one instance, used everywhere it is needed
- Makes **testing easier**; you can swap out real services for mock ones effortlessly
- Promotes **loose coupling**; classes don't tightly depend on each other's internal details
- Improves **readability and maintainability**; less boilerplate, clearer intent

Less boilerplate: Less repetitive setup code that developers would otherwise need to write manually.

Real-World Analogy (The Water Supply System)

This is probably the clearest way to understand DI:

Without NestJS (no DI)**:** Every time you need water, you grab a bucket, walk all the way to the water tank, fill it up, and carry it back yourself. It works; but it is exhausting, repetitive, and messy.

With NestJS (with DI)**:** A WaterServiceProvider installs a pipe system *once*. After that, water flows automatically from the tank straight to your tap; you just open it.

```
WaterTank(CategoryService)
        |
        |   NestJS IoC Container sets up the pipes for you
        |
        v
      Pipe(Dependency Injection)
        |
        v
Your Tap(CategoryController)
```

Just open the tap; you never touch the tank directly. NestJS pipes the Service into your Controller automatically.

Water System	NestJS Equivalent
Carrying a bucket manually	new CategoryService()
Water Tank	CategoryService
Pipe Installer	NestJS IoC Container
Pipe	Dependency Injection
Your Tap	CategoryController

Note: NestJS IoC Container is the internal system of NestJS that automatically creates, manages, and delivers dependencies (like Services) wherever they are needed.

Practical Example

Generate the Feature Using NestJS CLI

```
nest g module category
nest g controller category
nest g service category
```

Your folder will look like this:

```
src /
└── category /
    ├── category.module.ts
    ├── category.controller.ts
    ├── category.controller.spec.ts
    ├── category.service.ts
    └── category.service.spec.ts
```

Service Code

src/category/category.service.ts

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

// @Injectable() registers this class as a provider.
```

```
// This tells NestJS: "I'm available — inject me wherever I'm needed."
@Injectable()
export class CategoryService {

    // A simple method that returns a list of category names.
    // In a real app, this would fetch data from a database.
    getCategories() {
        return ['Microcontroller', 'Microprocessor', 'Display'];
    }

}
```

☠️ Controller Code

src/category/category.controller.ts

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

// Importing CategoryService so we can use it inside this controller.
// We don't create it manually — NestJS will inject it for us.
import { CategoryService } from './category.service';

@Controller('category')
// All routes in this controller are prefixed with /category
export class CategoryController {

    // Dependency Injection happening right here in the constructor.
    // We declare that we need a CategoryService, and NestJS automatically
    // creates and provides one — we never write 'new CategoryService()'.
    constructor(private categoryService: CategoryService) { }

    @Get()
    // Handles GET requests to → /category
    getAllCategories() {
        // Calling the service method through the injected instance.
        // The controller doesn't care HOW the data is fetched — that's the service's job.
        return this.categoryService.getCategories();
    }
}
```

🚀 Running the App & Expected Output

```
npm run start: dev
```

http://localhost:3000/category

```
[
    "Microcontroller",
    "Microprocessor",
    "Display"
]
```

❓ How DI Works Behind the Scenes

```
1. NestJS reads @Injectable() on CategoryService
        ↓
2. NestJS registers it in the IoC Container (the pipe system)
        ↓
3. CategoryController declares it needs CategoryService in its constructor
        ↓
4. NestJS looks it up in the container and automatically injects it
        ↓
5. Controller uses the service — no manual 'new' required, ever.
```

IoC Container (Inversion of Control); this is the internal system NestJS uses to keep track of all providers and figure out where to inject them. You never interact with it directly; NestJS manages it for you.

🐓 Interview Questions (Dependency Injection)

Q1. What is Dependency Injection in NestJS?

DI is a design pattern where NestJS automatically creates and provides required dependencies (like services) to classes that need them. Instead of manually instantiating a service with new, we declare it in the constructor and NestJS injects it automatically through its IoC Container.

Q2. What is an IoC Container?

IoC stands for **Inversion of Control**. The IoC Container is NestJS's internal system that manages the creation and lifecycle of all providers. When a class needs a dependency, the container looks it up and injects it automatically; you never create or manage instances manually.

Q3. What is the difference between private and private readonly in constructor injection?

```
// 'private' → accessible only within this class, but can be reassigned
constructor(private categoryService: CategoryService) { }

// 'private readonly' → accessible only within this class AND cannot be reassigned
constructor(private readonly categoryService: CategoryService) { }
```

Using readonly is the recommended approach because it protects the injected instance from being accidentally overwritten anywhere in the class.

Q4. What is loose coupling and why does DI promote it?

Loose coupling means classes don't directly depend on the concrete implementation of other classes — they just declare what they need. Because NestJS injects the dependency, you can easily swap out one implementation for another (like a mock service during testing) without changing the controller code at all.

Q5. What is the difference between public and private in constructor injection?

```
// 'private' → accessible ONLY within this class
constructor(private categoryService: CategoryService) { }

// 'public' → accessible from OUTSIDE the class as well
constructor(public categoryService: CategoryService) { }
```

	private	public
Accessible inside class	✅	✅
Accessible outside class	❌	✅
Recommended for services	✅	❌

📌 **Rule of thumb:** Always use private for injected services. Your controller's dependencies are internal implementation details; the outside world has no business accessing them directly. Use public only when you intentionally need to expose a property outside the class.

Full comparison of all modifiers:

```
constructor(private categoryService: CategoryService) { }          // internal only
constructor(private readonly categoryService: CategoryService) { } // internal + protected from
reassignment ☑ best practice
constructor(public categoryService: CategoryService) { }           // exposed outside
constructor(public readonly categoryService: CategoryService) { }  // exposed + protected
```

Dependency Injection covered! The plumbing of NestJS is now crystal clear. 🚀

REST API & HTTP Methods + Postman

❓ What is an API?

API stands for **Application Programming Interface**. In simple words, it is a way for two applications to talk to each other.

Think of it as a **bridge**; the frontend (what the user sees) sends a request across the bridge, and the backend (your NestJS server) sends a response back.

❓ What is a REST API?

REST stands for **Representational State Transfer**. It is not a tool or a library; it is a **set of rules** that defines how a client and server should communicate over the internet.

A REST API:

- Uses standard **HTTP methods** (GET, POST, PUT, PATCH, DELETE).
- Is **stateless**; each request is independent, the server doesn't remember previous requests.
- Is **simple and scalable**; easy to understand, easy to grow.

❓ HTTP Methods; What Do They Mean?

These methods tell the server **what kind of action** the client wants to perform:

Method	Purpose	Example
GET	Read or fetch existing data	Get all products
POST	Create new data	Add a new product
PUT	**Completely** update existing data	Replace a product's full info
PATCH	**Partially** update existing data	Update only the product price
DELETE	Remove data	Delete a product

📌 A simple way to remember this is **CRUD**; **C**reate (POST), **R**ead (GET), **U**pdate (PUT/PATCH), **D**elete (DELETE).

❓ Why is REST API Important?

- It **organizes** how clients interact with your server in a predictable, standard way
- Keeps your backend code **clean, structured, and reusable**
- Makes your backend work like a **service** that any client; browser, mobile app, another server; can consume

❓ We Have Controllers; Then Why Do We Need REST API?

A controller is just **NestJS code sitting inside your server**. It exists on the backend. But the outside world; your frontend, mobile app, or Postman; needs a **standard, agreed-upon way** to talk to it. That is exactly what REST API provides.

REST API is the **common language** between client and server.

```
Frontend / Mobile App
        |
        |  REST API — the agreed rules & language
        ▼
   Your Controller(NestJS)
```

Here is a good analogy to lock this in:

🍽 The **controller is the chef**. The **REST API is the menu**. Without a menu, customers don't know what to order or how to ask for it. The menu (REST API) defines what is available and how to request it; the chef (controller) then handles it.

❓ What is Postman?

Postman is an **API testing tool** that lets you send any HTTP request; GET, POST, PUT, PATCH, DELETE; to your server and see the response. And you don't need a frontend at all to do it.

This is super useful during development because you can test your endpoints(e.g., /products) immediately without building a UI first.

🚘 Browser vs Postman

HTTP Method	Browser	Postman
GET	☑	☑
POST	❌	☑
PUT	❌	☑
PATCH	❌	☑
DELETE	❌	☑

The browser can **only** send GET requests to test (when you type a URL and hit enter). Postman can send **all** HTTP methods; which is why we use it for API testing.

🔍 Download & Install Postman

🔗 https://www.postman.com/downloads/

Install it the usual way — Next → Next → Finish. Once it is open, you are ready to start testing your NestJS endpoints.

🫀 Interview Questions (REST API & HTTP Methods)

Q1. What is the difference between PUT and PATCH?

Both are used for updating data, but the key difference is:

- **PUT** replaces the **entire** resource. If we don't include a field, it gets wiped.
- **PATCH** updates only the **specified fields**, leaving everything else untouched.

For example, if a product has name, price, and stock:

- PUT → you must send all fields, even if only one changed
- PATCH → you send only one, and the rest stays as-is

Q2. What does "stateless" mean in REST?

Stateless means the server **does not remember** anything about previous requests. Every request must contain all the information needed to process it. The server treats each request as if it is seeing that client for the very first time.

Q3. What is the difference between an API and a REST API?

An **API** is a general concept; any interface that allows two systems to communicate. A **REST API** is a specific *type* of API that follows the REST architectural rules; using HTTP methods, being stateless, and returning data (usually JSON) in a structured way.

REST API concepts locked in! Now let's start actually building and testing endpoints. 🚀

Create REST APIs (GET, POST, PUT, PATCH, DELETE)

Generate the Feature Using NestJS CLI

```
nest g module student
nest g controller student
nest g service student
```

Your folder will now look like this:

```
src /
└── student /
    ├── student.module.ts
    ├── student.controller.ts
    ├── student.controller.spec.ts
    ├── student.service.ts
    └── student.service.spec.ts
```

Service Code

src/student/student.service.ts

```ts
import { Injectable, NotFoundException } from '@nestjs/common';
// NotFoundException is a built-in NestJS exception that returns a 404 HTTP error

@Injectable()
export class StudentService {

    // In-memory array acting as our temporary data source
    // In a real app, this would be replaced by a database
    private students = [
        { id: '22-47929-2', name: 'Sagar Biswas', dept: 'BSc in CSE' }, // index 0
        { id: '22-47930-2', name: 'Anik Das', dept: 'BSc in EEE' }, // index 1
        { id: '22-47931-2', name: 'Rafiul Islam', dept: 'BSc in CSE' }, // index 2
    ];

    // ── GET ──────────────────────────────────────────────────────

    // Returns the full list of students
    getAllStudents() {
        return this.students;
```

```
}

// Finds and returns one student by their ID
getStudentById(id: string) {
    const student = this.students.find(student => student.id === id);

    if (!student) {
        // If no student matches the given ID, throw a 404 error
        throw new NotFoundException('Student not found');
    }

    return student;
}

// ── POST ──────────────────────────────────────────────────────────

/*
// Option A: Auto-generate the ID (no id needed in the request body)
addStudent(student: { name: string, dept: string }) {
    const newStudent = {
        id: Date.now().toString(), // Unique ID based on current timestamp
        ...student                 // Spread operator: copies name & dept into newStudent
    };
    this.students.push(newStudent);
    return newStudent;
}
*/

// Option B: Client provides the ID in the request body
addStudent(student: { id: string, name: string, dept: string }) {
    this.students.push(student); // Adds the new student object to the array
    return student;              // Returns the newly added student
}

// ── PUT ───────────────────────────────────────────────────────────

// Completely replaces a student's data (both name and dept must be provided)
updateStudent(id: string, updatedStudent: { name: string, dept: string }) {

    // findIndex() returns the position of the matching element, or -1 if not found
    const studentIndex = this.students.findIndex(student => student.id === id);

    if (studentIndex === -1) {
        throw new NotFoundException('Student not found');
    }

    // Replace the entire student object at that index
    // We keep the original id and overwrite everything else
    this.students[studentIndex] = { id, ...updatedStudent };

    return this.students[studentIndex]; // Return the updated student
}

// ── PATCH ─────────────────────────────────────────────────────────

// Updates only the fields provided — everything else stays the same
// Partial<{}> makes all properties optional so we can send just name, just dept, or both
patchStudent(id: string, updatedFields: Partial<{ name: string, dept: string }>) {

    const student = this.getStudentById(id); // Reuse existing method (throws 404 if not found)

    // Object.assign(target, source)
    // Copies only the provided fields from updatedFields into the existing student object
    // This is what gives PATCH its "partial update" behavior
    Object.assign(student, updatedFields);

    return student; // Return the patched student
}

// ── DELETE ────────────────────────────────────────────────────────

deleteStudent(id: string) {

    const studentIndex = this.students.findIndex(student => student.id === id);
```

```
        if (studentIndex === -1) {
            throw new NotFoundException('Student not found');
        }

        // splice(startIndex, deleteCount)
        // Removes 1 element at the given index and returns it as an array
        const deletedStudent = this.students.splice(studentIndex, 1);

        return {
            message: 'Student deleted successfully',
            student: deletedStudent[0] // splice returns an array, so we grab the first (and only)
element
        };
    }
}
```

☠ Controller Code

src/student/student.controller.ts

```
import { Controller, Get, Post, Body, Put, Param, Patch, Delete } from '@nestjs/common';
// Importing all the decorators we need for defining routes and extracting request data

import { StudentService } from './student.service';
// Importing the service that holds all our business logic

@Controller('student')
// All routes in this controller are prefixed with /student
export class StudentController {

    // NestJS automatically injects StudentService here via Dependency Injection
    constructor(private readonly studentService: StudentService) { }

    // ─── GET ──────────────────────────────────────────────────────────

    @Get()
    // Handles → GET /student
    getAll() {
        return this.studentService.getAllStudents();
    }

    @Get(':id')
    // Handles → GET /student/22-47929-2
    // ':id' makes this part of the URL a dynamic route parameter
    getById(@Param('id') id: string) {
        // @Param('id') extracts the 'id' value from the URL and passes it here
        return this.studentService.getStudentById(id);
    }

    // ─── POST ─────────────────────────────────────────────────────────

    @Post()
    // Handles → POST /student
    create(@Body() student: { id: string, name: string, dept: string }) {
        // @Body() extracts the entire request body and maps it to the 'student' parameter
        return this.studentService.addStudent(student);
    }

    // ─── PUT ──────────────────────────────────────────────────────────

    @Put(':id')
    // Handles → PUT /student/22-47931-2
    // Expects both 'name' and 'dept' in the body — full replacement
    update(
        @Param('id') id: string,
        @Body() updatedStudent: { name: string, dept: string }
    ) {
        return this.studentService.updateStudent(id, updatedStudent);
    }

    // ─── PATCH ────────────────────────────────────────────────────────
```

```
    @Patch(':id')
    // Handles → PATCH /student/22-47931-2
    // Partial<{}> means any combination of fields is acceptable — partial update
    patch(
        @Param('id') id: string,
        @Body() updatedStudent: Partial<{ name: string, dept: string }>
    ) {
        return this.studentService.patchStudent(id, updatedStudent);
    }

    // ─── DELETE ──────────────────────────────────────────────────────────

    @Delete(':id')
    // Handles → DELETE /student/22-47930-2
    delete(@Param('id') id: string) {
        return this.studentService.deleteStudent(id);
    }
}
```

💣 Testing in Postman

```
npm run start: dev
```

🕷️ GET; Fetch All Students

GET http://localhost:3000/student/

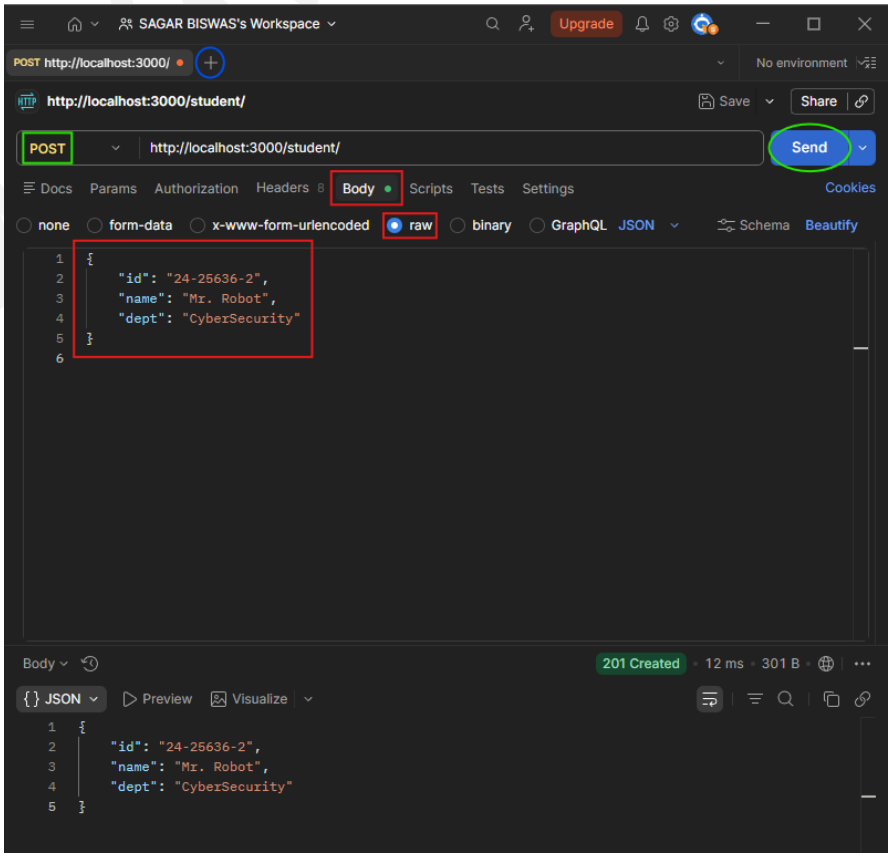
```
[
    { "id": "22-47929-2", "name": "Sagar Biswas", "dept": "BSc in CSE" },
    { "id": "22-47930-2", "name": "Anik Das", "dept": "BSc in EEE" },
    { "id": "22-47931-2", "name": "Rafiul Islam", "dept": "BSc in CSE" }
]
```

🕷️ GET; Fetch One Student by ID

GET http://localhost:3000/student/22-47929-2

```
{ "id": "22-47929-2", "name": "Sagar Biswas", "dept": "BSc in CSE" }
```

POST; Add a New Student



🕷 **POST** http://localhost:3000/student/

In Postman: Body → raw → JSON

```
{
    "id": "24-25636-2",
    "name": "Mr. Robot",
    "dept": "CyberSecurity"
}
```

Response:

```
{ "id": "24-25636-2", "name": "Mr. Robot", "dept": "CyberSecurity" }
```

🕷 **PUT; Fully Update a Student**

PUT http://localhost:3000/student/22-47931-2

In Postman: Body → raw → JSON

```
{ "name": "Rafiul Islam", "dept": "BSc in Networking" }
```

⚠ PUT replaces the **entire** record — always send all fields, otherwise missing fields get wiped.

🕷 **PATCH; Partially Update a Student**

PATCH http://localhost:3000/student/24-25636-2

In Postman: Body → raw → JSON

```
{ "dept": "BSc in Cybersecurity" }
```

Response:

```
{ "id": "24-25636-2", "name": "Mr. Robot", "dept": "BSc in Cybersecurity" }
```

☑ PATCH only updates what you send — name stays untouched.

🕷 **DELETE; Remove a Student**

DELETE http://localhost:3000/student/22-47930-2

Response:

```
{
    "message": "Student deleted successfully",
    "student": { "id": "22-47930-2", "name": "Anik Das", "dept": "BSc in EEE" }
}
```

🔍 **Final Check; GET All After All Operations**

GET http://localhost:3000/student/

```
[
    { "id": "22-47929-2", "name": "Sagar Biswas", "dept": "BSc in CSE" },
    { "id": "22-47931-2", "name": "Rafiul Islam", "dept": "BSc in Networking" },
    { "id": "24-25636-2", "name": "Mr. Robot", "dept": "BSc in Cybersecurity" }
]
```

Anik Das (index 1) was deleted ☑, Rafiul's dept was updated via PUT ☑, Mr. Robot's dept was patched via PATCH ☑

🚘 Quick Decorator Reference

Decorator	What it does
@Get()	Handles GET requests
@Post()	Handles POST requests
@Put(':id')	Handles PUT requests with a route param
@Patch(':id')	Handles PATCH requests with a route param
@Delete(':id')	Handles DELETE requests with a route param
@Param('id')	Extracts a value from the URL (:id)
@Body()	Extracts the full request body

Full CRUD REST API done! You can now create, read, update, and delete data like a proper backend developer. 🖋

---------- X ----------