### **Module 1: Introduction to NestJS0**

* What is NestJS?
* Why use NestJS for backend development?
* Real-world use cases
* Overview of the E-Commerce Application we'll build (Products, Categories, Users, Orders, Payments, etc.)

### **Module 2: Environment Setup**

* Prerequisites: Node.js, PostgreSQL, Package Manager (npm/yarn), IDE (VSCode)
* Installing Nest CLI
* Creating your first NestJS project
* Understanding NestJS folder structure and CLI tools

### **Module 3: Understanding NestJS Architecture**

* Core Principles: Modularity, Dependency Injection, Providers
* The role of main.ts and AppModule
* Project structure best practices for large applications
* Intro to Clean Architecture concepts in NestJS

### **Module 4: Key NestJS Terminologies**

* **Modules**: Organizing features like Users, Products, Orders
* **Controllers**: Handling HTTP requests (GET, POST, etc.)
* **Services**: Business logic and data handling
* **Decorators**: @Module, @Injectable, @Controller, etc.
* **Providers**: Services, factories, repositories, and custom providers
* Middlewares, Pipes, Interceptors, and Guards (brief overview)

### **Module 5: Creating Your First API Endpoint**

* Defining routes with controllers
* Connecting controllers to services
* Testing endpoints using Postman or Swagger
* Example use-case: GET /products

### **Module 6: Setting Up PostgreSQL**

* Database installation and setup
* Creating the e-commerce database and user
* NestJS database configuration using environment variables
* Using a configuration module for scalable setups

### **Module 7: Introduction to ORM**

* What is an ORM and why use one?
* TypeORM overview
* Other ORM alternatives in NestJS (Prisma, Sequelize) – quick mention

### **Module 8: Integrating TypeORM with NestJS**

* Configuring TypeORM with MySQL
* Introduction to Entity and Repository patterns
* Structuring the project by modules (Users, Products, Orders, etc.)
* Mapping entities to tables

### **Module 9: Designing E-Commerce Entities**

* Core entities:  
  + **User**
  + **Product**
  + **Category**
  + **Order**
  + **Order\_products (pivot table)**
  + **Payment**
  + **Role**
  + **Permission\_roles (pivot\_table)**
  + **Permission**

Relationships: Briefly explain

* + One-to-One (order ↔ Payment)
  + One-to-Many (Category → Products). ( Role→ users)
  + Many-to-Many (Products ↔ Orders ), (Roles → Permissions)

### **Module 10: Performing Queries**

* Using Repository methods for:  
  + Creating records
  + Fetching by ID or filters
  + Updating and deleting records
* Query Builder vs Repository pattern
* Pagination, Sorting, and Searching (e.g., Product list)

### **Module 11: Seeders**

* Generating and running database migrations
* Updating schema without losing data
* Writing seed scripts for:  
  + Sample users
  + Demo products and categories
* Organizing and automating migrations and seeds

### **Module 12: Authentication with Passport**

* Overview of Passport.js in NestJS
* Implementing local strategy for login
* Securing user passwords with hashing
* Login and registration flows

### **Module 13: Token & Refresh Token System**

* Generating JWT access tokens
* Using refresh tokens for session persistence
* Token expiration and renewal strategy
* Where to store and how to manage refresh tokens securely

### **Module 14: Authorization & Role-Based Access**

* Protecting routes with JWT Guards
* Implementing role-based access (Admin, Customer)
* Custom decorators and guards (@Roles())
* Ownership checks (e.g., update only your own order)

### **Module 15: File Upload Functionality**

* Accepting product images or user profile pictures
* Validating file size, type, and storage location
* Structuring folders for local or cloud storage (S3, Cloudinary intro)

### **Module 16: Sending Emails**

* Configuring an email provider (SMTP, Mailgun, or SendGrid)
* Email service module
* Templating emails (order confirmation, password reset, etc.)
* Sending transactional emails on events

### **Module 17: Queues and Background Jobs**

* Why use queues? (Decoupling, performance)
* Integrating Redis with Bull (Queue manager)
* Job lifecycle: queue → process → completed/failed
* Background use-cases:  
  + Sending emails
  + Invoice generation
  + Order processing

### **Module 18: Project Finalization**

* Environment separation (Dev, Staging, Prod)
* Swagger API documentation
* Logging and error handling
* Deployment strategies (Heroku, Docker, Railway, etc.)

## **Module 1: Introduction to NestJS**

### **What is NestJS?**

NestJS is a **progressive Node.js framework** built with and fully supporting **TypeScript**. It is heavily inspired by Angular in terms of architecture and design patterns, making it ideal for building **scalable, testable, and maintainable server-side applications**.

Key characteristics:

* **Modular**: Encourages splitting code into reusable modules.
* **TypeScript-first**: Offers type safety and modern JS features.
* **Extensible**: Integrates easily with libraries like TypeORM, Passport, and more.
* **Powerful Dependency Injection (DI)**: Promotes clean, testable code.

### **🚀 Why Use NestJS for Backend Development?**

1. **Scalability** NestJS is designed for building both monolithic and microservices architectures. You can scale your application in modules.
2. **Type Safety** Thanks to TypeScript, you get auto-completion, compile-time checks, and better documentation through types.
3. **Built-in Support for Modern Features** Like decorators, interceptors, guards, and middleware—no need to manually wire them.
4. **Seamless Integration** Works perfectly with ORMs (like TypeORM, Prisma), WebSockets, GraphQL, queues (like Bull), and authentication tools (like Passport).
5. **Familiarity** If you know Angular or any MVC structure, NestJS feels intuitive and familiar.
6. **Robust CLI Tools** Helps generate modules, controllers, and services quickly with consistent boilerplate.

### **🌍 Real-World Use Cases of NestJS**

* **E-Commerce Platforms**: Managing users, inventory, orders, and payments.
* **Microservices & Event-Driven Systems**: Using Redis, Kafka, or RabbitMQ.
* **Admin Dashboards & APIs**: With robust access control and role-based systems.
* **Real-time Applications**: Chat apps, live dashboards with WebSocket integration.
* **Enterprise Applications**: Modular architecture helps large teams scale quickly.

Companies using NestJS include Adidas, Autodesk, Roche, and many early-stage startups due to its maintainability and dev experience.

### **🛒 Overview of the E-Commerce Application We'll Build**

In this course, you’ll build a production-ready **E-Commerce API** using NestJS and PostgreSQL. Here's a high-level view of the core features and entities:

#### **🧑‍🤝‍🧑 User Management**

* Customer registration & login
* Admin and role-based access
* Token & refresh token authentication

#### **📦 Products**

* CRUD operations
* Image uploads
* Stock & pricing management

#### **🗂️ Categories**

* Nested categories (e.g., Electronics → Phones)
* Products linked to one or multiple categories

#### **🛒 Shopping Cart**

* Add/remove/update products in cart
* Calculate total price and quantity

#### **📑 Orders**

* Place an order from the cart
* Manage order status: pending, shipped, completed, cancelled

#### **💳 Payments**

* Simulate or integrate with third-party gateways (e.g., Paystack, Stripe)
* Save transaction history

#### **📬 Notifications & Emails**

* Send order confirmation and status update emails
* Use background jobs (queues) to process email sending

#### **📁 File Upload**

* Upload product images or user profile pictures

#### **🧰 Admin Controls**

* Manage all products, categories, users, and orders from backend APIs

## **Module 2: Environment Setup**

### **✅ 1. Prerequisites**

Before diving into NestJS development, make sure the following tools are installed on your machine:

#### **🔹 Node.js**

* Required to run NestJS and manage dependencies.
* Install the **LTS version** from<https://nodejs.org>

#### **🔹 PostgreSQL**

* The relational database we’ll use in this course.
* You can install it locally or use tools like Docker, pgAdmin, or hosted services (e.g., Supabase, Railway).

#### **🔹 Package Manager**

* Either:  
  + npm – comes bundled with Node.js
  + yarn – install with npm install -g yarn

#### **🔹 IDE**

* Use **Visual Studio Code (VSCode)** for the best experience, especially with TypeScript and NestJS plugins (e.g., ESLint, Prettier, REST Client, etc.).

### **🛠️ 2. Installing Nest CLI**

The Nest CLI helps you scaffold and manage NestJS applications efficiently.

#### **📦 Installation command:**

bash

npm install -g @nestjs/cli

#### **🔍 Verify installation:**

bash

nest --version

### **🚀 3. Creating Your First NestJS Project**

Use the Nest CLI to create a new project.

#### **🏗️ Command:**

bash

nest new ecommerce-api

You’ll be prompted to:

* Choose a package manager (npm, yarn, or pnpm)
* Wait while it installs dependencies and sets up the initial structure

## **Steps to Start a NestJS App After Installation**

### **✅ 1. Navigate to the Project Folder**

After running nest new ecommerce-api, go into the project directory:

bash

cd ecommerce-api

### **✅ 2. Install Dependencies (if not already done)**

If you skipped automatic installation during setup, run:

bash

npm install

Or

yarn install

### **✅ 3. Start the Development Server**

To run your app in development mode with automatic reloads:

bash

npm run start:dev

Or

yarn start:dev

This uses [**Webpack Hot Module Replacement**] to reload the app when you change the code.

### **✅ 4. Open in Browser or API Client**

By default, the app runs on:

arduino

http://localhost:3000

You can test the default route (GET /) by visiting:

arduino

http://localhost:3000/

Or use an API tool like [Postman](https://www.postman.com/) or [Insomnia](https://insomnia.rest/) to test the API.

* **Install Dependencies for casl (Authorization Role and permission):**
* Install nest-casl: npm install nest-casl
  + Install CASL: npm install @casl/ability
  + Install TypeORM: npm install --save @nestjs/typeorm typeorm

### **✅ 5. Build for Production (Optional)**

To build the app for production:

npm run build

The compiled files will be in the dist/ folder.

To run the production build:

node dist/main

### **✅ 6. Useful Additional Commands**

| **Action** | **Command** |
| --- | --- |
| Start normally | npm run start |
| Watch & reload on changes | npm run start:dev |
| Debug mode | npm run start:debug |
| Run tests | npm run test |
| Format code | npm run format |
| Lint code | npm run lin |

**🧬 4. Understanding NestJS Folder Structure**

After generating the project, you'll see the following structure:

ecommerce-api/

├── src/

│ ├── app.controller.ts # Handles incoming HTTP requests

│ ├── app.service.ts # Business logic

│ ├── app.module.ts # Root application module

│ └── main.ts # Application entry point

├── test/ # Unit and e2e tests

├── package.json # Project dependencies and scripts

├── tsconfig.json # TypeScript configuration

└── nest-cli.json # CLI configuration

#### **🗂️ Key Folders and Files:**

* src/: Main source folder where all code lives  
  + app.module.ts: Root module of the application
  + app.controller.ts: Default controller for handling routes
  + app.service.ts: Provides logic for the controller
  + main.ts: Entry point of the application (bootstraps NestJS app)
* test/: Default folder for test files
* node\_modules/: Installed packages (auto-managed)
* package.json: Dependency management and scripts
* tsconfig.json: TypeScript compiler settings

’

### **🧰 5. Nest CLI Generators – Syntax Guide**

The CLI helps generate files for you. Here's a quick reference:

| **Purpose** | **CLI Command** |
| --- | --- |
| Generate a Module | nest g module <name> or nest generate module <name> |
| Generate a Controller | nest g controller <name> |
| Generate a Service | nest g service <name> |
| Generate a Resource | nest g resource <name> (interactive: CRUD setup) |
| Generate a Class | nest g class <name> |
| Generate a DTO | nest g class <name>.dto --no-spec |
| Generate an Interface | nest g interface <name> |
| Generate a Pipe | nest g pipe <name> |
| Generate a Guard | nest g guard <name> |
| Generate an Interceptor | nest g interceptor <name> |

Example:

nest g module products # Creates a products module

nest g controller products # Creates products.controller.ts

nest g service products # Creates products.service.ts

## **3. NestJS Architecture and Structure**

### **🔹 1. Entry Point: main.ts**

The main.ts file is the **entry point** of every NestJS application. It's where the app is bootstrapped using the NestFactory.

* It creates an instance of the application using the root module (typically AppModule)
* You can configure global middleware, pipes, guards, exception filters, etc. here
* It starts the HTTP server and listens on a specified port

This file is similar to index.js in Node.js or main() in other languages.

### **🔹 2. App Module: app.module.ts**

The **AppModule** is the **root module** that connects all feature modules. It's the starting point of your app's module tree.

In @Module() metadata, you define:

* imports: other modules to import
* controllers: HTTP route handlers
* providers: services or any injectable classes
* exports: what this module shares with others

Think of it as the “brain” that wires up everything in the app.

## **🧭 Core Components of NestJS**

### **🔹 3. Controllers (Full Explanation)**

Controllers handle **incoming HTTP requests** and return **responses**. They act as the **entry point to your app’s routes**.

* A controller is a class annotated with the @Controller() decorator
* Inside, methods are mapped to HTTP verbs using decorators like:  
  + @Get(), @Post(), @Put(), @Delete(), @Patch()
* Controllers do **not contain business logic**; they delegate to services

**Example usage in e-commerce:**

* ProductsController handles routes like GET /products, POST /products
* AuthController handles login, register, etc.

Controllers should stay slim. Keep heavy logic in services.

### **🔹 4. Providers (Services, Repositories, etc.)**

Providers are **classes that can be injected into other components** (via DI) and perform specific tasks like data fetching, calculations, or external calls.

#### **💡 Most common types of providers:**

* **Services**: Business logic (e.g., calculating order totals, sending emails)
* **Repositories**: Data access (e.g., interacting with TypeORM for entities)
* **Helpers/Utilities**: For shared logic across the app

Defined using the @Injectable() decorator.

In e-commerce: ProductsService manages product-related logic, OrdersService handles checkout and order placement.

### **🔹 5. Modules (Full Explanation)**

Modules are the **building blocks of a NestJS app**. They group related code into cohesive units.

Every NestJS app has at least one module: AppModule. But large apps are divided into **feature modules**, like:

* UsersModule
* ProductsModule
* AuthModule
* OrdersModule

A module is a class decorated with @Module(), where you define:

* imports: other modules this one depends on
* controllers: request handlers
* providers: services and other injectable dependencies
* exports: services to share with other modules

Modules help maintain:

* Separation of concerns
* Reusability
* Scalable structure

## **⚙️ Advanced System Components**

### **🔹 6. Pipes**

Pipes are used to **transform input data** or **validate** it before it reaches your controllers or services.

Use cases:

* Input validation (e.g., class-validator + DTOs)
* Data transformation (e.g., converting strings to numbers)

Built-in pipes:

* ValidationPipe, ParseIntPipe, DefaultValuePipe, etc.

You can also create **custom pipes**.

### **🔹 7. Guards**

Guards are used for **authorization and route access control**.

They run **before the route handler**, and can allow or deny access based on:

* Roles
* Permissions
* User ownership

Example:

* RolesGuard checks if a user has the right role to access a route.
* JwtAuthGuard checks for a valid token before proceeding.

Implemented using @CanActivate() and applied with @UseGuards().

### **🔹 8. Interceptors**

Interceptors are used to **transform responses**, **log requests**, **handle caching**, or **wrap methods** in extra logic.

They are similar to middleware but work **around method execution**.

Use cases:

* Logging request duration
* Wrapping responses (e.g., adding a status field to API responses)
* Error transformation

Defined with @Injectable() and @UseInterceptors().

### **🔹 9. Middleware**

Middleware functions are executed **before the route handler**, similar to Express middleware.

Use cases:

* Logging requests
* Checking request headers
* Setting CORS or rate limiting

They can be applied globally or to specific routes.

For example: Log all requests to /api/\*, or verify API keys before hitting routes.

### **🔹 10. Dependency Injection (DI) – Full Explanation**

Dependency Injection is a **design pattern** where an object receives its dependencies from an external source, instead of creating them itself.

In NestJS:

* DI is handled automatically using TypeScript decorators like @Injectable()
* You simply add a service in a class constructor, and Nest injects it

Benefits:

* **Testability**: Easily mock dependencies during unit tests
* **Reusability**: Services can be shared across components
* **Decoupling**: Components don’t create their dependencies, leading to cleaner architecture

#### **Example:**

A ProductsController doesn’t need to create a ProductsService — it declares it in its constructor, and NestJS injects it automatically.

ts

constructor(private readonly productsService: ProductsService) {}

## **✅ Summary**

| **Concept** | **Responsibility** |
| --- | --- |
| main.ts | Bootstraps the application |
| AppModule | Root module of the app |
| Controllers | Handle incoming HTTP requests |
| Providers | Contain logic, interact with DB, reusable code |
| Modules | Group related features, organize app |
| Pipes | Validate and transform input data |
| Guards | Handle access control |
| Interceptors | Wrap methods for logging, transforming responses |
| Middleware | Execute logic before requests hit controllers |
| DI | Automatically injects classes for decoupling |

## **Module 4: Key NestJS Terminologies**

### **🔹 1. Modules – Organizing Features**

Modules are the **core building blocks** in NestJS. They help group and encapsulate related functionalities.

* Each feature (Users, Products, Orders, etc.) should have its own module.
* Promotes modularity, reusability, and easier maintenance.

📦 **E-Commerce Example:**

* UsersModule handles registration, login, profiles.
* ProductsModule handles listing, adding, editing products.
* OrdersModule handles order creation, status updates.

Every NestJS app must have at least one root module (AppModule).

### **🔹 2. Controllers – Handling HTTP Requests**

Controllers define **routes and handle incoming requests**.

* Decorated with @Controller()
* Methods inside are mapped to HTTP verbs: @Get(), @Post(), @Put(), @Delete(), etc.
* Should only focus on **request-response logic**, and **delegate processing to services**

📦 **E-Commerce Example:**

* ProductsController may expose routes like:  
  + GET /products
  + POST /products
  + GET /products/:id

### **🔹 3. Services – Business Logic & Data Handling**

Services contain the core logic of your application.

* Decorated with @Injectable()
* Used to abstract complex operations from controllers
* Can be injected into other components (thanks to DI)

📦 **E-Commerce Example:**

* ProductsService handles fetching products from the database
* OrdersService handles calculating totals and placing orders

### **🔹 4. Decorators – Metadata & Behavior Enhancers**

Decorators are special **annotations** used throughout NestJS to attach **metadata** or modify behavior.

Common decorators include:

| **Decorator** | **Purpose** |
| --- | --- |
| @Module() | Declares a Nest module |
| @Controller() | Marks a class as a controller |
| @Injectable() | Marks a class as a service or provider |
| @Get() | Maps GET HTTP requests to a method |
| @Post() | Maps POST requests to a method |
| @Param() | Extracts route parameters |
| @Body() | Extracts request body |
| @Query() | Extracts query parameters |
| @UseGuards() | Attaches a guard to a controller/route |

📦 **E-Commerce Example:**

* @Controller('products') to route requests to product endpoints

### **🔹 5. Providers – Services, Repositories, and More**

In NestJS, a provider is **any class that can be injected** into another class via Dependency Injection.

Common providers:

* **Services** (business logic)
* **Repositories** (data access layer)
* **Factories** (for generating config or instances)
* **Custom providers** (e.g., logging or caching logic)

Defined in the providers array of a module.

📦 **E-Commerce Example:**

* A StripePaymentProvider could be a custom provider for handling Stripe payments.

### **🔹 6. Middleware, Pipes, Interceptors, and Guards (Quick Overview)**

These are **advanced tools** used to control request/response flow and enforce policies.

#### **🔸 Middleware**

* Runs **before** the route handler
* Used for logging, request manipulation, authentication tokens, etc.

#### **🔸 Pipes**

* Validate and transform data
* Commonly used with DTOs for input validation

#### **🔸 Interceptors**

* Wrap around method execution
* Used for logging, response transformation, caching, etc.

#### **🔸 Guards**

* Determine if a request can be processed (used for **authorization**)
* Example: Only allow admins to access certain endpoints

📦 **E-Commerce Example:**

* Use a **Guard** to allow only users with role admin to create products
* Use a **Pipe** to validate product data before saving

## **Module 5: Creating Your First API Endpoint**

### **🔹 1. Defining Routes with Controllers**

Controllers handle routing and are the entry point for HTTP requests in NestJS.

* Decorated with @Controller('products') to route requests to /products
* Use method decorators to define route types:  
  + @Get() – Retrieve data
  + @Post() – Create new data
  + @Put() – Update existing data
  + @Delete() – Remove data

📦 **E-Commerce Example – Product Routes:**

| **Method** | **Route** | **Purpose** |
| --- | --- | --- |
| GET | /products | Get all products |
| POST | /products | Add a new product |
| PUT | /products/:id | Update a product |
| DELETE | /products/:id | Delete a product |

### **🔹 2. Connecting Controllers to Services**

Controllers delegate actual business logic to **services**.

* Use Dependency Injection to inject ProductsService into ProductsController
* The controller handles request data and response, the service handles processing

**Workflow:**

* POST /products → controller receives product data
* controller → calls productsService.createProduct(data)
* service → saves product to database
* controller → returns success response

### **🔹 3. Testing Endpoints Using Postman or Swagger**

#### **✅ Postman**

* Create and test requests manually
* Set headers, request bodies, and auth tokens
* Monitor and inspect responses

#### **✅ Swagger**

* Auto-generates API documentation
* Use @nestjs/swagger module
* Interact with all endpoints through the Swagger UI (http://localhost:3000/api)

Benefits:

* Easy for teams, QA, and 3rd parties to understand and test your API

### **🔹 4. CRUD Example Use-Case: Products API**

#### **🟢 GET /products**

* Fetch all available products
* Returns a list of product objects

#### **🔵 POST /products**

* Create a new product
* Requires product data in the request body

Example payload:  
  
 json  
  
{

"name": "Wireless Mouse",

"price": 3500,

"description": "Ergonomic wireless mouse",

"categoryId": 2

}

#### **🟡 PUT /products/1**

* Update product with ID 1
* Requires updated data in request body

#### **🔴 DELETE /products/1**

* Deletes product with ID 1
* Responds with success message or error if not found

### **✅ Summary Table**

| **Endpoint** | **Method** | **Description** |
| --- | --- | --- |
| /products | GET | List all products |
| /products | POST | Create new product |
| /products/:id | PUT | Update product by ID |
| /products/:id | DELETE | Delete product by ID |

## **Module 6: Setting Up PostgreSQL for NestJS E-Commerce App**

### **🔹 1. Database Installation and Setup**

PostgreSQL is a powerful, open-source, relational database system. We’ll use it to store all the structured data in our e-commerce app (e.g., products, users, orders, categories, etc.).

#### **✅ Installation Steps**

**Windows:**

* Download installer from the official PostgreSQL website:<https://www.postgresql.org/download/windows/>
* Follow the wizard, set a superuser password (typically for the postgres user)

**macOS:**

brew install postgresql

brew services start postgresql

**Linux (Debian/Ubuntu):**

sudo apt update

sudo apt install postgresql postgresql-contrib

sudo service postgresql start

### **🔹 2. Creating the E-Commerce Database and User**

It's a best practice to create a dedicated database and user for your application, rather than using the default postgres superuser.

#### **✅ Access the PostgreSQL CLI:**

bash

psql -U postgres

Enter your password if prompted.

#### **✅ Run SQL Commands:**

sql

-- Create the database

CREATE DATABASE ecommerce\_db;

-- Create a dedicated user

CREATE USER ecommerce\_user WITH PASSWORD 'secure\_password';

-- Grant all privileges on the database to the user

GRANT ALL PRIVILEGES ON DATABASE ecommerce\_db TO ecommerce\_user;

✅ Tip: Replace 'secure\_password' with a strong password. Never use 1234 or admin.

### **🔹 3. NestJS Database Configuration Using Environment Variables**

Storing database credentials directly in code is a security risk. Instead, we use **environment variables** to manage sensitive data

npm install dotenv

#### **✅ Create a .env file at the root of your project:**

env

DB\_HOST=localhost

DB\_PORT=5432

DB\_USERNAME=ecommerce\_user

DB\_PASSWORD=secure\_password

DB\_NAME=ecommerce\_db

Add .env to your .gitignore file so it’s not pushed to version control (GitHub, GitLab, etc.)

### **🔹 4. Using a Configuration Module for Scalable Setups**

NestJS provides @nestjs/config for handling app configuration in a clean and scalable way. It reads from .env files and makes values accessible throughout your app.

#### **✅ Install the package:**

bash

npm install @nestjs/config

#### **✅ Configure in app.module.ts:**

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

import { ConfigModule } from '@nestjs/config';

@Module({

imports: [

ConfigModule.forRoot({

isGlobal: true, // makes it available app-wide

}),

],

})

export class AppModule {}

#### **✅ Access Configuration in Services:**

import { ConfigService } from '@nestjs/config';

constructor(private configService: ConfigService) {

const dbHost = this.configService.get<string>('DB\_HOST');

const dbPort = this.configService.get<number>('DB\_PORT');

}

This allows dynamic configuration for different environments:

* .env for development
* .env.staging for staging
* .env.production for production

### **🔹 5. Next Step: Integrating PostgreSQL with TypeORM**

In the next module, we’ll install **TypeORM** and link it to our PostgreSQL setup using the variables we just created.

### **✅ Module Summary**

| **Task** | **Description** |
| --- | --- |
| Install PostgreSQL | Install locally via installer or CLI |
| Create database and user | Use SQL to create ecommerce\_db and ecommerce\_user |
| Configure .env file | Store database settings securely |
| Install @nestjs/config | Load config from .env throughout the application |
| Use ConfigService | Inject and use configuration data dynamically |

# **Module 7: Introduction to ORM**

## **🧠 1. What is an ORM and Why Use One?**

### **🔍 Definition**

An **ORM (Object-Relational Mapper)** is a tool that lets developers interact with a database using object-oriented programming concepts. Instead of writing raw SQL queries, you define models (classes) that map to database tables.

### **💡 How It Works**

An ORM:

* Maps **tables** to **classes**
* Maps **rows** to **objects**
* Maps **columns** to **object properties**

This lets you do things like:

ts

const user = await userRepository.findOne({ where: { email: 'test@example.com' } });

Instead of writing:

SELECT \* FROM users WHERE email = 'test@example.com';

### **✅ Why Use an ORM?**

| **Benefit** | **Explanation** |
| --- | --- |
| 🔄 Abstraction | Focus on business logic instead of SQL syntax. |
| 📈 Productivity | Less boilerplate = faster development. |
| 🔧 Migrations | Easily manage schema changes over time. |
| 🤝 Relationships | Handle one-to-many, many-to-many with decorators or definitions. |
| 🧪 Type Safety | Autocomplete and compile-time checks with TypeScript. |
| ♻️ Reusability | Create reusable models and repositories. |

## **⚙️ 2. TypeORM Overview**

### **🏷 What is TypeORM?**

* A **TypeScript-based ORM** that integrates deeply with NestJS.
* Supports **Active Record** and **Data Mapper** patterns.
* Allows easy use of decorators to define entities and relationships.
* Handles **migrations**, **transactions**, **eager/lazy loading**, etc.

### **🔨 Key Features**

* TypeScript-first
* Works seamlessly with NestJS
* Entities as classes with decorators
* Supports PostgreSQL, MySQL, SQLite, MSSQL, and more

### **📄 Example: Defining an Entity**

ts

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

@Entity()

export class Product {

@PrimaryGeneratedColumn()

id: number;

@Column()

name: string;

@Column('decimal')

price: number;

@Column({ default: true })

isActive: boolean;

}

### **🔎 Example: Using a Repository**

ts

@Injectable()

export class ProductService {

constructor(

@InjectRepository(Product)

private productRepository: Repository<Product>,

) {}

async findAll(): Promise<Product[]> {

return this.productRepository.find();

}

async create(data: CreateProductDto): Promise<Product> {

const product = this.productRepository.create(data);

return this.productRepository.save(product);

}

}

### **🛠 TypeORM Patterns**

| **Pattern** | **Description** |
| --- | --- |
| **Active Record** | Entities contain both data and methods (like .save()) |
| **Data Mapper** | Logic is in services or repositories, not entities (preferred in NestJS) |

## **Module 8: Integrating TypeORM with NestJS**

### **1. ✅ Configuring TypeORM with PostgreSQL**

This sets up the database connection in NestJS using TypeORM and PostgreSQL.

**Install dependencies:**

npm install --save @nestjs/typeorm typeorm pg

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

@Module({

imports: [

TypeOrmModule.forRoot({

type: 'postgres',

host: 'localhost',

port: 5432,

username: 'postgres',

password: 'your\_password',

database: 'your\_db',

entities: [\_\_dirname + '/\*\*/\*.entity{.ts,.js}'],

migrations: [\_\_dirname + '/migrations/\*{.ts,.js}'],

migrationsTableName: 'migrations',

synchronize: false,

}),

],

})

export class AppModule {}

**Also add a ormconfig.ts or extend CLI options in package.json:**

ts

// ormconfig.ts (or `data-source.ts` in newer setups)

import { DataSource } from 'typeorm';

export default new DataSource({

type: 'postgres',

host: 'localhost',

port: 5432,

username: 'postgres',

password: 'your\_password',

database: 'your\_db',

entities: ['src/\*\*/\*.entity.ts'],

migrations: ['src/migrations/\*.ts'],

});

**Configure TypeOrmModule in app.module.ts:**synchronize: true

🔒 Do **not** use in production. We’ll use migrations instead.

ts

### **2. 🏗️ Introduction to Entity and Repository Patterns**

#### **Entity Pattern:**

* An Entity is a TypeScript class decorated with @Entity() and mapped to a database table.
* Use @PrimaryGeneratedColumn(), @Column(), and relationship decorators.

#### **Repository Pattern (Without Custom Class):**

* TypeORM provides a default repository which is injected into services using @InjectRepository().
* This repository exposes methods like find(), create(), save(), etc.

### **3. 🧩 Structuring the Project by Modules**

Each feature (like Users, Products, Orders) is isolated into a module. This improves **modularity, separation of concerns, and scalability.**

**Structure:**

arduino

src/

├── users/

│ ├── user.entity.ts

│ ├── users.service.ts

│ ├── users.controller.ts

│ └── users.module.ts

├── products/

├── orders/

Each module uses:

ts

TypeOrmModule.forFeature([User])

To register entities for repository injection.

### **4. 🗺️ Mapping Entities to Tables**

Use decorators to map class properties to table columns.

ts

@Entity()

export class Product {

@PrimaryGeneratedColumn()

id: number;

@Column()

name: string;

@Column('decimal')

price: number;

@Column({ default: true })

isActive: boolean;

}

Use @OneToMany, @ManyToOne, @JoinColumn etc. to define relationships between entities.

### **5. 📦 Managing Migrations in TypeORM**

**Why Migrations?** Migrations help manage schema changes **safely** and **version-controlled** across environments.

#### **📁 Setup CLI Script for Migrations**

Add this to your package.json:

json

"scripts": {

"typeorm": "ts-node -r tsconfig-paths/register ./node\_modules/typeorm/cli.js",

"migration:generate": "npm run typeorm -- migration:generate src/migrations/MigrationName",

"migration:run": "npm run typeorm -- migration:run",

"migration:revert": "npm run typeorm -- migration:revert"

}

Ensure you're using tsconfig-paths and ts-node:

bash

npm install --save-dev ts-node tsconfig-paths

#### **📥 Create a Migration**

bash

npm run migration:generate -- -d ormconfig.ts src/migrations/InitialSchema

This inspects your entities and generates SQL code to sync the database.

#### **🚀 Run Migrations**

bash

npm run migration:run

#### **⏪ Revert a Migration**

bash

npm run migration:revert

**Module 9: Designing E-Commerce Entities.**

## RDBMS Relationships — Overview

In Relational Database Management Systems (RDBMS), relationships define how tables are logically connected. There are three main types:

One to One (1:1)

One to many (1:M)

Many to Many (M:M)

## **Core Entities & Relationships**

### **1. User**

* Represents a customer or admin.
* Fields: id, name, email, password, phone , address , roleId (FK)

### **2. Role**

* Defines access levels: admin, customer, manager, etc.
* Fields: id, name
* **One-to-Many** → Role → Users

### **3. Permission**

* Represents a specific action (e.g., "can\_view\_orders", "can\_edit\_products").
* Fields: id, name

### **4. PermissionRoles (Pivot Table)**

* Resolves **many-to-many** between Roles and Permissions
* Each row links a roleId to a permissionId

✅ **Many-to-Many:** Roles ↔ Permissions

### **5. Category**

* Groups products (e.g., Electronics, Clothing)
* Fields: id, name, description

✅ **One-to-Many:** Category → Products

### **6. Product**

* Represents items for sale.
* Fields: id, name, price, description, colour, size, categoryId(FK)

✅ **Many-to-One:** Product → Category

✅ **Many-to-Many:** Product ↔ Orders (via OrderProducts)

### **7. Order**

* Placed by a user.
* Fields: id, userId, tracking\_no, total, payment\_status, createdAt, updatedAt, deletedAt,

✅ **One-to-One:** Order ↔ Payment  
 ✅ **Many-to-Many:** Order ↔ Products (via OrderProducts)

### **8. OrderProducts (Pivot Table)**

* Links products to orders, with quantity.
* Fields: id, orderId, productId, quantity, amount

✅ **Many-to-Many:** Orders ↔ Products

### **9. Payment**

* Stores payment details of an order.
* Fields: id, orderId, amount, pay\_ref, txn\_id, txn\_ref, invoice, status, gateway, txn\_date, createdAt, updatedAt, deletedAt

✅ **One-to-One:** Order ↔ Payment  
 (Each order has exactly one payment record)

## **🔗 Relationship Summary (with explanations)**

| **Relationship** | **Entities** | **Explanation** |
| --- | --- | --- |
| **One-to-One** | Order ↔ Payment | One order has one payment, one payment belongs to one order |
| **One-to-Many** | Category → Products | A category has many products, each product belongs to one category |
| **One-to-Many** | Role → Users | A role can have many users |
| **Many-to-Many** | Products ↔ Orders | A product can belong to many orders, and an order can contain many products (via pivot) |
| **Many-to-Many** | Roles ↔ Permissions | A role can have many permissions and vice versa (via pivot) |

# **Module 10: Performing Queries**

**Tech Stack**: NestJS · TypeORM · MySQL  
 **Goal**: Learn how to effectively interact with the database using Repository methods and Query Builder in TypeORM for common data operations.

https://orkhan.gitbook.io/typeorm/docs/find-options#basic-options

## **🔨 1. Using Repository Methods**

TypeORM’s Repository pattern provides built-in methods for standard CRUD operations.

### **✅ Creating Records**

ts

const product = productRepository.create({ name: 'Phone', price: 299 });

await productRepository.save(product);

* create() instantiates an entity
* save() persists to DB

### **✅ Fetching by ID or Filters**

ts

// By ID

const user = await userRepository.findOneBy({ id: 1 });

// With filters

const activeProducts = await productRepository.find({

where: { status: 'active' },

relations: ['category'],

});

* findOneBy and find allow filtering by fields.
* relations loads related entities (e.g., Category → Products)

### **✅ Updating Records**

ts

await productRepository.update({ id: 1 }, { price: 350 });

* Or load + save:

ts

const product = await productRepository.findOneBy({ id: 1 });

product.price = 350;

await productRepository.save(product);

### **✅ Deleting Records**

ts

await productRepository.delete({ id: 1 });

* For **soft deletes** (when using @DeleteDateColumn):

ts

await productRepository.softDelete({ id: 1 });

## **⚖️ 2. Query Builder vs Repository Pattern**

| **Feature** | **Repository** | **Query Builder** |
| --- | --- | --- |
| Syntax | Simple, high-level | SQL-like, flexible |
| Use case | Standard CRUD | Complex joins, filters, aggregations |
| Performance | Optimized, abstracted | Tuned queries, fine-grained control |
| Example | repo.find({ where: { name } }) | createQueryBuilder().where(...).getMany() |

### **🔍 Query Builder Example**

ts

const products = await dataSource

.getRepository(Product)

.createQueryBuilder('product')

.leftJoinAndSelect('product.category', 'category')

.where('product.price > :minPrice', { minPrice: 100 })

.orderBy('product.createdAt', 'DESC')

.getMany();

Use when:

* You need custom SQL
* You want multiple joins, dynamic filters

## **🔎 3. Pagination, Sorting, and Searching**

### **✅ Pagination**

ts

const page = 1;

const limit = 10;

const [products, total] = await productRepository.findAndCount({

skip: (page - 1) \* limit,

take: limit,

});

### **✅ Sorting**

ts

await productRepository.find({

order: {

price: 'ASC', // or DESC

},

});

### **✅ Searching (e.g., by name)**

ts

await productRepository.find({

where: {

name: ILike(`%${searchTerm}%`), // case-insensitive

},

});

## **🧪 Suggested Exercises**

1. Create a new product with a related category
2. Fetch a paginated product list, sorted by createdAt
3. Search for products by name using a query parameter
4. Compare the performance of a repository query vs query builder

# **Module 11: Seeders in NestJS (with TypeORM + Faker)**

**Goal**: Automate database population using seeders for development/testing.  
 **Tools**: typeorm-extension, faker, TypeORM, NestJS

## **🔧 1. Why Use Seeders?**

* Populate test data (users, products, categories)
* Improve development and QA workflow
* Bootstrap demo content for frontend testing

## **Step-by-Step Guide to Seeding with typeorm-extension**

### **📦 1. Install Required Packages**

bash

CopyEdit

npm install typeorm-extension @faker-js/faker

Also ensure:

bash

CopyEdit

npm install --save-dev ts-node typescript

### **📁 2. Directory Structure for Seeds**

pgsql

CopyEdit

src/

├── seeds/

│ ├── category.seed.ts

│ ├── product.seed.ts

│ ├── user.seed.ts

├── database/

│ └── data-source.ts

### **⚙️ 3. Configure data-source.ts for Seeders**

Update your data-source.ts to include the seeder and factory paths:

ts

CopyEdit

// src/database/data-source.ts

import 'reflect-metadata';

import { DataSource } from 'typeorm';

import { User } from '../entities/user.entity';

import { Product } from '../entities/product.entity';

import { Category } from '../entities/category.entity';

export const AppDataSource = new DataSource({

type: 'mysql',

host: 'localhost',

port: 3306,

username: 'root',

password: 'yourpassword',

database: 'yourdb',

entities: [User, Product, Category],

migrations: ['src/migrations/\*.ts'],

synchronize: false,

// TypeORM Extension config

seeds: ['src/seeds/\*\*/\*{.ts,.js}'],

});

✅ If you're using ormconfig.ts, add the seeds property there instead.

### **🧬 4. Create a Seeder File**

Example: src/seeds/user.seed.ts

ts

CopyEdit

import { Seeder } from 'typeorm-extension';

import { DataSource } from 'typeorm';

import { User } from '../entities/user.entity';

import { faker } from '@faker-js/faker';

export default class UserSeeder implements Seeder {

public async run(dataSource: DataSource): Promise<void> {

const repo = dataSource.getRepository(User);

for (let i = 0; i < 10; i++) {

const user = repo.create({

name: faker.person.fullName(),

email: faker.internet.email(),

password: 'hashedPassword',

});

await repo.save(user);

}

}

}

Repeat similarly for product.seed.ts and category.seed.ts.

### **🚀 5. Run the Seeder**

Use the typeorm-extension CLI:

bash

CopyEdit

npx typeorm-extension seed

This will:

* Automatically load all \*.seed.ts files
* Execute their run() methods

### **✅ 6. Optional: Add a NPM Script**

In package.json:

json

CopyEdit

"scripts": {

"seed": "ts-node src/seeds/user.seed.ts", // single file run

"seed:all": "typeorm-extension seed" // run all seeders

}

Run with:

bash

CopyEdit

npm run seed:all

### **📌 Notes:**

* typeorm-extension automatically detects and runs seed files that **default export a class implementing Seeder**.
* If you need ordered seeding, you can create a main.seed.ts that runs each manually.

# **🧩 Module 12: Authentication with Passport in NestJS**

## **🎯 Goal**

Build a secure authentication system in NestJS using Passport.js, implement local strategy for login, secure passwords, and structure login/registration flows.

## **📌 1. What is Authentication?**

**Authentication** is the process of verifying the identity of a user.

In most apps, this involves:

* Users registering (creating credentials)
* Logging in (providing credentials)
* Receiving a token (session or JWT)
* Accessing protected routes or data

## **🔐 2. Overview of Passport.js in NestJS**

### **✅ What is Passport.js?**

[Passport.js](http://www.passportjs.org/) is a flexible middleware for authentication in Node.js.  
 In NestJS, it integrates with the @nestjs/passport module.

### **🔧 How NestJS uses it:**

* Strategies (local, jwt, etc.)
* Guards to protect routes
* Custom validation logic

### **📦 Required Packages**

bash

CopyEdit

npm install @nestjs/passport passport passport-local bcrypt

npm install --save-dev @types/passport-local @types/bcrypt

## **🔐 3. Securing Passwords with Hashing**

Use bcrypt to:

* Hash passwords before storing
* Compare hashed password during login

### **🔐 Hash Password**

ts

CopyEdit

import \* as bcrypt from 'bcrypt';

const hashedPassword = await bcrypt.hash(password, 10);

### **✅ Compare During Login**

ts

CopyEdit

const isMatch = await bcrypt.compare(inputPassword, storedHash);

## **🔐 4. Local Strategy (Username/Password Login)**

### **✨ Local Strategy Setup**

#### **local.strategy.ts**

ts

CopyEdit

import { Strategy } from 'passport-local';

import { PassportStrategy } from '@nestjs/passport';

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

import { AuthService } from './auth.service';

@Injectable()

export class LocalStrategy extends PassportStrategy(Strategy) {

constructor(private authService: AuthService) {

super(); // defaults: username + password

}

async validate(username: string, password: string) {

const user = await this.authService.validateUser(username, password);

if (!user) {

throw new UnauthorizedException('Invalid credentials');

}

return user; // attaches to request.user

}

}

### **🛡️ Guard: local-auth.guard.ts**

ts

CopyEdit

import { AuthGuard } from '@nestjs/passport';

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

@Injectable()

export class LocalAuthGuard extends AuthGuard('local') {}

## **🔐 5. Auth Service Logic**

#### **auth.service.ts**

ts

CopyEdit

@Injectable()

export class AuthService {

constructor(private usersService: UsersService) {}

async validateUser(username: string, password: string) {

const user = await this.usersService.findByUsername(username);

if (user && await bcrypt.compare(password, user.password)) {

const { password, ...result } = user;

return result;

}

return null;

}

async login(user: any) {

return {

message: 'Login successful',

user,

};

}

}

## **🔐 6. Login & Registration Flow**

### **👤 Registration Controller**

ts

CopyEdit

@Post('register')

async register(@Body() dto: CreateUserDto) {

const hashed = await bcrypt.hash(dto.password, 10);

return this.usersService.create({ ...dto, password: hashed });

}

### **🔐 Login Controller**

ts

CopyEdit

@UseGuards(LocalAuthGuard)

@Post('login')

async login(@Request() req) {

return this.authService.login(req.user);

}

## **🧠 Authentication Flow Summary**

scss

CopyEdit

[User Submits Login Form]

↓

[LocalAuthGuard]

↓

[LocalStrategy.validate()]

↓

[authService.validateUser()]

↓

bcrypt.compare() → Success or fail

↓

[req.user set] → Controller gets user

# **🔐 Module 13: Token & Refresh Token System in NestJS**

This module teaches how to build a **secure, stateless session system** using JWT access tokens and refresh tokens in NestJS.

## **🧩 Why Use JWT + Refresh Tokens?**

### **🔑 JWT Access Token**

* Short-lived
* Stored on client side (e.g., in memory or Authorization header)
* Used to access protected resources

### **♻️ Refresh Token**

* Long-lived
* Used to get a new access token without forcing login
* Should be stored securely (e.g., HttpOnly cookie or secure storage)

## **📦 Required Packages**

bash

CopyEdit

npm install @nestjs/jwt passport-jwt

npm install --save-dev @types/passport-jwt

## **🔐 Step 1: JWT Module Configuration**

### **auth.module.ts**

ts

CopyEdit

import { JwtModule } from '@nestjs/jwt';

JwtModule.register({

secret: process.env.JWT\_SECRET,

signOptions: { expiresIn: '15m' }, // Access token duration

});

Use environment variables for secrets.

## **🔐 Step 2: Generate Access and Refresh Tokens**

### **auth.service.ts**

ts

CopyEdit

@Injectable()

export class AuthService {

constructor(

private jwtService: JwtService,

private usersService: UsersService,

) {}

async generateTokens(user: any) {

const payload = { sub: user.id, email: user.email };

const accessToken = await this.jwtService.signAsync(payload, {

expiresIn: '15m',

});

const refreshToken = await this.jwtService.signAsync(payload, {

expiresIn: '7d',

secret: process.env.JWT\_REFRESH\_SECRET,

});

// Optionally save hashed refreshToken in DB

await this.usersService.updateRefreshToken(user.id, refreshToken);

return { accessToken, refreshToken };

}

}

## **✅ Step 3: Secure Refresh Token Storage**

### **Best practice options:**

| **Location** | **Security** |
| --- | --- |
| **HttpOnly Cookie** | ✅ Secure, not accessible via JS |
| **Secure localStorage** | ❌ Vulnerable to XSS (not ideal) |
| **Encrypted DB column** | ✅ Safe for server-side revocation |

You should:

* **Hash** refresh tokens before saving
* Invalidate old refresh tokens on new login/logout

## **🔁 Step 4: Refresh Token Flow**

### **auth.controller.ts**

ts

CopyEdit

@Post('refresh')

async refresh(@Body() body: { refreshToken: string }) {

const user = await this.authService.verifyRefreshToken(body.refreshToken);

if (!user) throw new UnauthorizedException();

return this.authService.generateTokens(user);

}

### **auth.service.ts**

ts

CopyEdit

async verifyRefreshToken(token: string) {

try {

const payload = await this.jwtService.verifyAsync(token, {

secret: process.env.JWT\_REFRESH\_SECRET,

});

const user = await this.usersService.findById(payload.sub);

if (!user || user.storedRefreshToken !== token) {

return null;

}

return user;

} catch (err) {

return null;

}

}

## **🧠 Expiration Strategy**

| **Token Type** | **Duration** | **Strategy** |
| --- | --- | --- |
| Access Token | ~15 mins | Used for requests |
| Refresh Token | ~7 days | Used for renewal |

* Access token expires quickly to reduce risk if stolen
* Refresh token stays valid longer but should be invalidated on logout or rotation

## **🚨 Security Best Practices**

1. ✅ **HttpOnly Cookies** for refresh tokens
2. ✅ **Rotate refresh tokens** every time it’s used
3. ✅ **Blacklist or expire old tokens** on logout
4. ✅ **Encrypt/Hash** tokens stored in DB
5. ❌ **Never expose secrets in frontend**

## **🧪 Optional: Guard Routes with JWT**

ts

CopyEdit

@Injectable()

export class JwtAuthGuard extends AuthGuard('jwt') {}

Use @UseGuards(JwtAuthGuard) to protect your endpoints.

## **✅ Summary**

| **Feature** | **Implemented** |
| --- | --- |
| Access Token | ✅ 15 mins expiry |
| Refresh Token | ✅ 7 days expiry |
| Secure storage | ✅ Encrypted DB or cookies |
| Token renewal endpoint | ✅ /auth/refresh |
| Logout and revocation | ✅ Clear refresh token |

# **Module 14: Authorization & Role-Based Access Using CASL (NestJS, TypeORM, MySQL)**

This module integrates **CASL** for rich, flexible access control and combines it with JWT Guards, roles, and ownership-based logic.

## **🛠 Prerequisites**

Install required packages:

bash

CopyEdit

npm install @casl/ability @casl/nestjs

## **✅ 1. Protecting Routes with JWT Guards**

Use JwtAuthGuard to ensure only authenticated users can access protected routes:

ts

CopyEdit

@UseGuards(JwtAuthGuard)

@Get('me')

getProfile(@Request() req) {

return req.user;

}

## **🔐 2. Define Roles and Permissions with CASL**

### **🔖 Create ability types**

ts

CopyEdit

// ability.factory.ts

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

import {

AbilityBuilder,

AbilityClass,

ExtractSubjectType,

InferSubjects,

PureAbility,

} from '@casl/ability';

import { User } from 'src/users/entities/user.entity';

import { Order } from 'src/orders/entities/order.entity';

type Subjects = InferSubjects<typeof Order | typeof User> | 'all';

export type Actions = 'manage' | 'create' | 'read' | 'update' | 'delete';

export type AppAbility = PureAbility<[Actions, Subjects]>;

@Injectable()

export class AbilityFactory {

defineAbility(user: User) {

const { can, cannot, build } = new AbilityBuilder<AppAbility>(PureAbility);

if (user.role === 'admin') {

can('manage', 'all');

} else {

can('read', 'all');

can('update', Order, { userId: user.id });

can('delete', Order, { userId: user.id });

}

return build({

detectSubjectType: (item) =>

typeof item === 'string' ? item : item.constructor,

});

}

}

## **🎯 3. Create @CheckAbilities() Decorator & Guard**

### **🧩 check-abilities.decorator.ts**

ts

CopyEdit

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

import { AppAbility } from './ability.factory';

export interface RequiredRule {

action: AppAbility['can'][0];

subject: Parameters<AppAbility['can']>[1];

}

export const CHECK\_ABILITY = 'check\_ability';

export const CheckAbilities = (...rules: RequiredRule[]) =>

SetMetadata(CHECK\_ABILITY, rules);

### **🛡 abilities.guard.ts**

ts

CopyEdit

import {

CanActivate,

ExecutionContext,

Injectable,

ForbiddenException,

} from '@nestjs/common';

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

import { CHECK\_ABILITY, RequiredRule } from './check-abilities.decorator';

import { AbilityFactory } from './ability.factory';

@Injectable()

export class AbilitiesGuard implements CanActivate {

constructor(

private reflector: Reflector,

private abilityFactory: AbilityFactory,

) {}

canActivate(context: ExecutionContext): boolean {

const rules = this.reflector.get<RequiredRule[]>(

CHECK\_ABILITY,

context.getHandler(),

);

if (!rules) return true;

const request = context.switchToHttp().getRequest();

const user = request.user;

const ability = this.abilityFactory.defineAbility(user);

for (const rule of rules) {

if (ability.cannot(rule.action, rule.subject)) {

throw new ForbiddenException('Access denied');

}

}

return true;

}

}

## **🧪 4. Apply Ability Checks in Controller**

ts

CopyEdit

@UseGuards(JwtAuthGuard, AbilitiesGuard)

@CheckAbilities({ action: 'update', subject: Order })

@Put('orders/:id')

updateOrder(@Param('id') id: number, @Body() dto: UpdateOrderDto) {

return this.ordersService.update(id, dto);

}

## **🔐 5. Ownership Check (Order ↔ User)**

ts

CopyEdit

// In ability.factory.ts

can('update', Order, { userId: user.id });

This ensures a user can only update their **own** orders. CASL checks this condition against the resource's properties.

## **✅ Summary**

| **Feature** | **Implemented ✅** |
| --- | --- |
| JWT route protection | ✅ JwtAuthGuard |
| Role-based access (admin/customer) | ✅ via AbilityFactory |
| Custom decorator for abilities | ✅ @CheckAbilities() |
| Ownership-based permission | ✅ (e.g., Order.userId === user.id) |