Build an e-commerce app with **NestJS**

Documentation:

NestJS is one of the best Node frameworks for building server-side applications. we'll explore how to build a simple NestJS ecommerce app, demonstrating many of Nest's major features along the way.

We'll cover:

- Getting started with our NestJS ecommerce app
- Creating the NestJS ecommerce store product feature
- Creating the user management feature
- Creating user authentication and authorization
- Creating the store cart feature for our NestJS ecommerce app

Getting started with our NestJS ecommerce app:

Modules:

Definition: Modules are used to organize and structure a Nest.js application. At least one root module is required to create an app.

Composition: Modules can contain controllers, services, and even other modules.

Dependency Injection: Nest uses the dependency injection pattern to connect modules with their dependencies.

Controllers:

Responsibility: Controllers handle incoming HTTP requests, validate parameters, and return responses to the client.

Clean and Simple: Controllers should be kept clean and simple, with most of the complex logic delegated to services.

Services:

Role: Services hold the business logic and application functionality. Complex logic should be implemented within services.

Provider: Services are a type of class known as providers.

Dependency Injection: Services can be injected into controllers or other services.

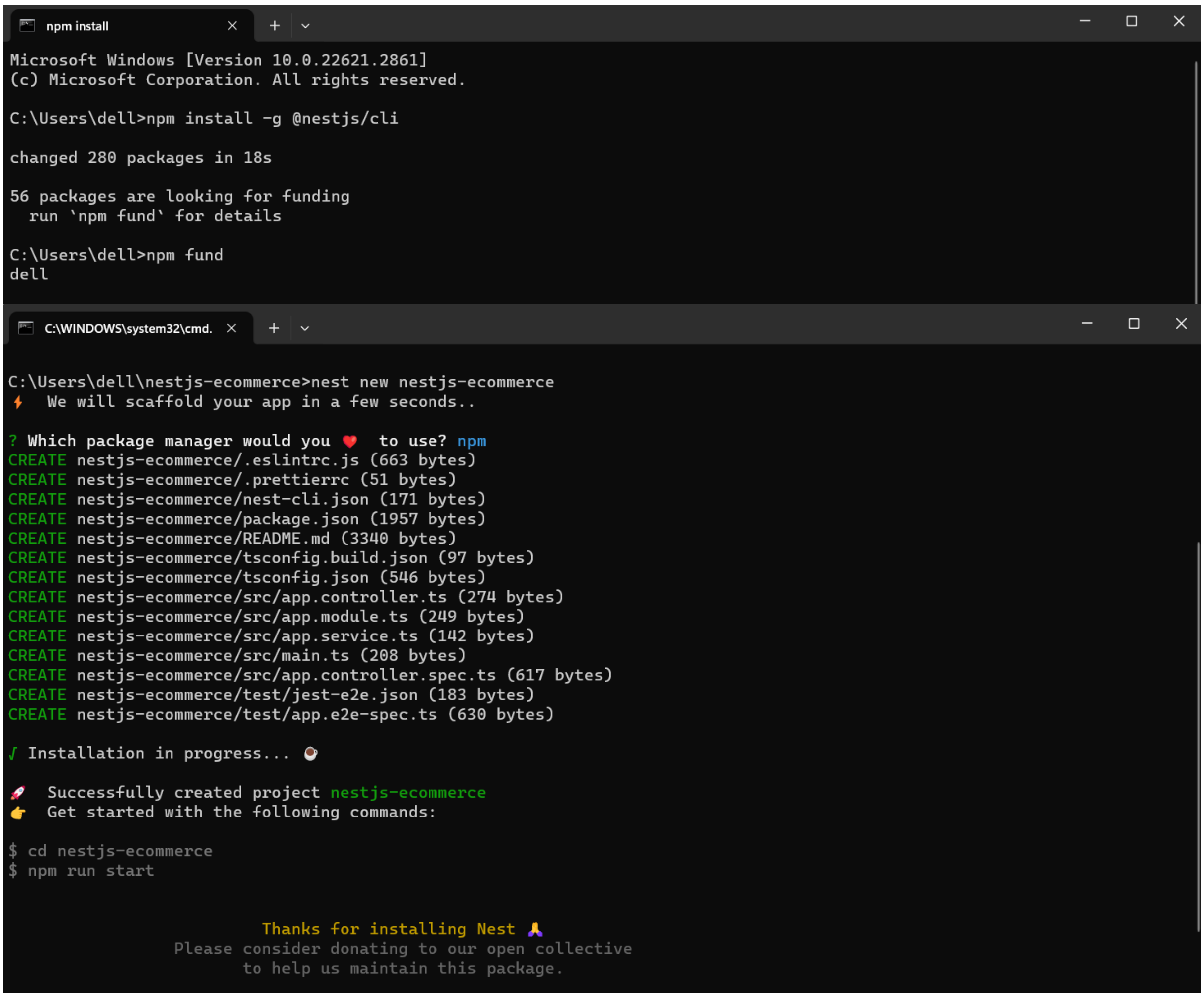
Providers:

Definition: Providers are classes that can be injected as dependencies.

Types: Besides services, providers can include classes like repositories, factories, helpers, etc.

let’s initialize a new Nest project. First, we’ll install Nest CLI. Then, we will create a new project:

```
npm install -g @nestjs/cli
nest new nestjs-ecommerce
```



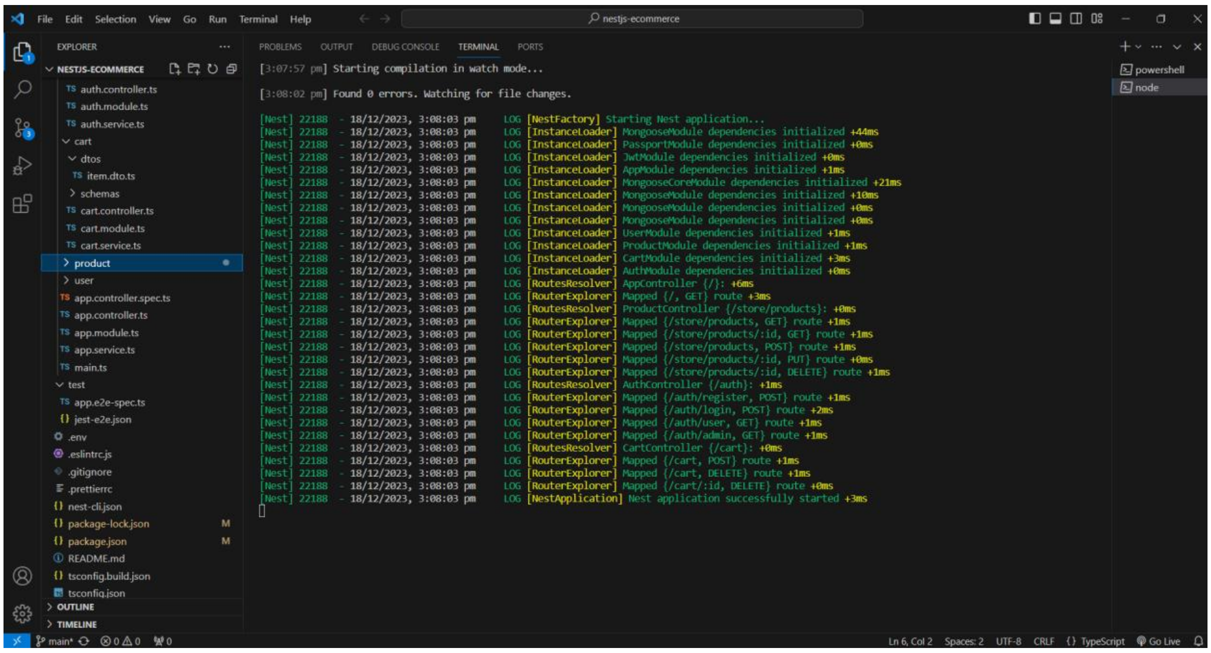
```
Microsoft Windows [Version 10.0.22621.2861]
(c) Microsoft Corporation. All rights reserved.

C:\Users\dell>npm install -g @nestjs/cli

changed 280 packages in 18s

56 packages are looking for funding
  run `npm fund` for details

C:\Users\dell>npm fund
dell
```

```
C:\Users\dell\nestjs-ecommerce>nest new nestjs-ecommerce
⚡  We will scaffold your app in a few seconds..

? Which package manager would you ❤️  to use? npm
CREATE nestjs-ecommerce/.eslintrc.js (663 bytes)
CREATE nestjs-ecommerce/.prettierrc (51 bytes)
CREATE nestjs-ecommerce/nest-cli.json (171 bytes)
CREATE nestjs-ecommerce/package.json (1957 bytes)
CREATE nestjs-ecommerce/README.md (3340 bytes)
CREATE nestjs-ecommerce/tsconfig.build.json (97 bytes)
CREATE nestjs-ecommerce/tsconfig.json (546 bytes)
CREATE nestjs-ecommerce/src/app.controller.ts (274 bytes)
CREATE nestjs-ecommerce/src/app.module.ts (249 bytes)
CREATE nestjs-ecommerce/src/app.service.ts (142 bytes)
CREATE nestjs-ecommerce/src/main.ts (208 bytes)
CREATE nestjs-ecommerce/src/app.controller.spec.ts (617 bytes)
CREATE nestjs-ecommerce/test/jest-e2e.json (183 bytes)
CREATE nestjs-ecommerce/test/app.e2e-spec.ts (630 bytes)

√ Installation in progress... ☕

🚀  Successfully created project nestjs-ecommerce
👉  Get started with the following commands:

$ cd nestjs-ecommerce
$ npm run start


                 Thanks for installing Nest 🙏
          Please consider donating to our open collective
                 to help us maintain this package.
```

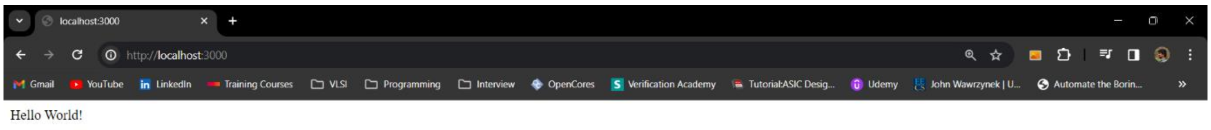
After installation is complete, navigate to the project and start it:

```
cd nestjs-ecommerce
npm run start:dev
```

You can then launch the app in your browser by visiting **http://localhost:3000/.** You should see a nice “Hello World!” message. The app will reload automatically after any changes you make. If you want to restart the app manually, use **npm run start** command instead.



```
[3:07:57 pm] Starting compilation in watch mode...

[3:08:02 pm] Found 0 errors. Watching for file changes.

[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [NestFactory] Starting Nest application...
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [InstanceLoader] MongooseModule dependencies initialized +44ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [InstanceLoader] PassportModule dependencies initialized +0ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [InstanceLoader] JwtModule dependencies initialized +0ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [InstanceLoader] AppModule dependencies initialized +1ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [InstanceLoader] MongooseCoreModule dependencies initialized +21ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [InstanceLoader] MongooseModule dependencies initialized +10ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [InstanceLoader] MongooseModule dependencies initialized +0ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [InstanceLoader] MongooseModule dependencies initialized +0ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [InstanceLoader] UserModule dependencies initialized +1ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [InstanceLoader] ProductModule dependencies initialized +1ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [InstanceLoader] CartModule dependencies initialized +3ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [InstanceLoader] AuthModule dependencies initialized +0ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [RoutesResolver] AppController {/}: +6ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [RouterExplorer] Mapped {/, GET} route +3ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [RoutesResolver] ProductController {/store/products}: +0ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [RouterExplorer] Mapped {/store/products, GET} route +1ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [RouterExplorer] Mapped {/store/products/:id, GET} route +1ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [RouterExplorer] Mapped {/store/products, POST} route +1ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [RouterExplorer] Mapped {/store/products/:id, PUT} route +0ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [RouterExplorer] Mapped {/store/products/:id, DELETE} route +1ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [RoutesResolver] AuthController {/auth}: +1ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [RouterExplorer] Mapped {/auth/register, POST} route +1ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [RouterExplorer] Mapped {/auth/login, POST} route +2ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [RouterExplorer] Mapped {/auth/user, GET} route +1ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [RouterExplorer] Mapped {/auth/admin, GET} route +1ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [RoutesResolver] CartController {/cart}: +0ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [RouterExplorer] Mapped {/cart, POST} route +1ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [RouterExplorer] Mapped {/cart, DELETE} route +1ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [RouterExplorer] Mapped {/cart/:id, DELETE} route +0ms
[Nest] 22188  - 18/12/2023, 3:08:03 pm     LOG [NestApplication] Nest application successfully started +3ms
```



Hello World!

Now we’re ready to start creating the store features.

Creating the NestJS ecommerce store product feature

In this section, we’ll focus on product management. The store product feature will allow us to retrieve store products, add new ones, and edit or delete them.

Creating our product resources

Let’s start by creating the needed resources. To create them, run the following commands:

```
nest g module product
nest g service product --no-spec
nest g controller product --no-spec
```

The first command generates a product module and puts it in its own directory with the same name.

The next two commands generate service and controller files and import them automatically in the **product** module. The **--no-spec** argument tells Nest that we don’t want to generate additional test files.

After running the above commands, we’ll get a new **product** directory containing the following files: **product.module.ts, product.service.ts,** and **product.controller.ts.**

Now we have a basic structure for the NestJS ecommerce store product feature. Before we move on, we need to set up our database.

Configuring the MongoDB database:

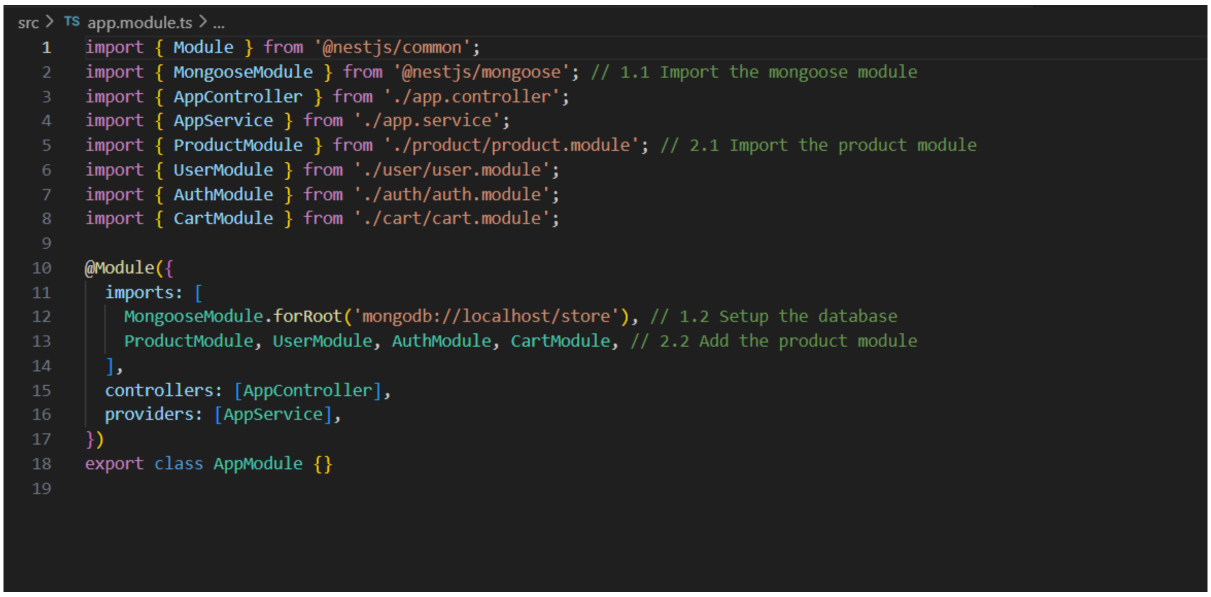
As we are using MongoDB as a database, we’ll need to install **mongoose** and **@nestjs/mongoose** packages.

```
npm install --save @nestjs/mongoose mongoose
```

After the installation is complete, open **app.module.ts**

Follow along using my numbered notes:

- First, we imported the **MongooseModule** (1.1) and used it to set up a new **store** database (1.2)
- Second, we imported the **ProductModule** (2.1) and added it to the **imports** array (2.2)

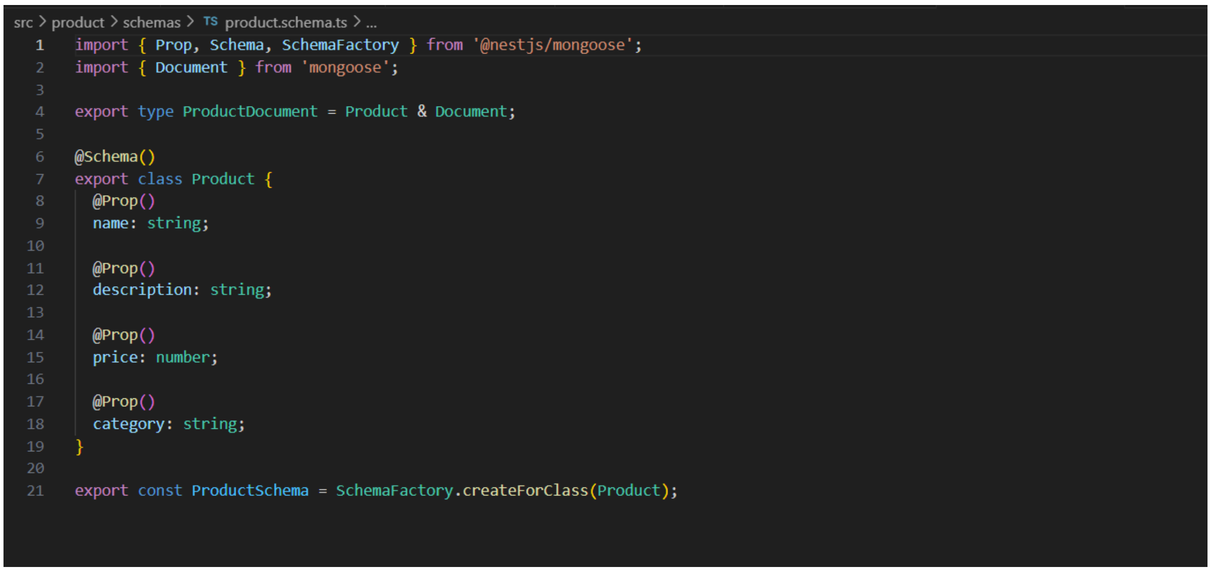

src > app.module.ts > ...

```typescript
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose'; // 1.1 Import the mongoose module
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ProductModule } from './product/product.module'; // 2.1 Import the product module
import { UserModule } from './user/user.module';
import { AuthModule } from './auth/auth.module';
import { CartModule } from './cart/cart.module';

@Module({
  imports: [
    MongooseModule.forRoot('mongodb://localhost/store'), // 1.2 Setup the database
    ProductModule, UserModule, AuthModule, CartModule, // 2.2 Add the product module
  ],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

Our next step is to create a database schema for our product model.

Creating a product model schema:

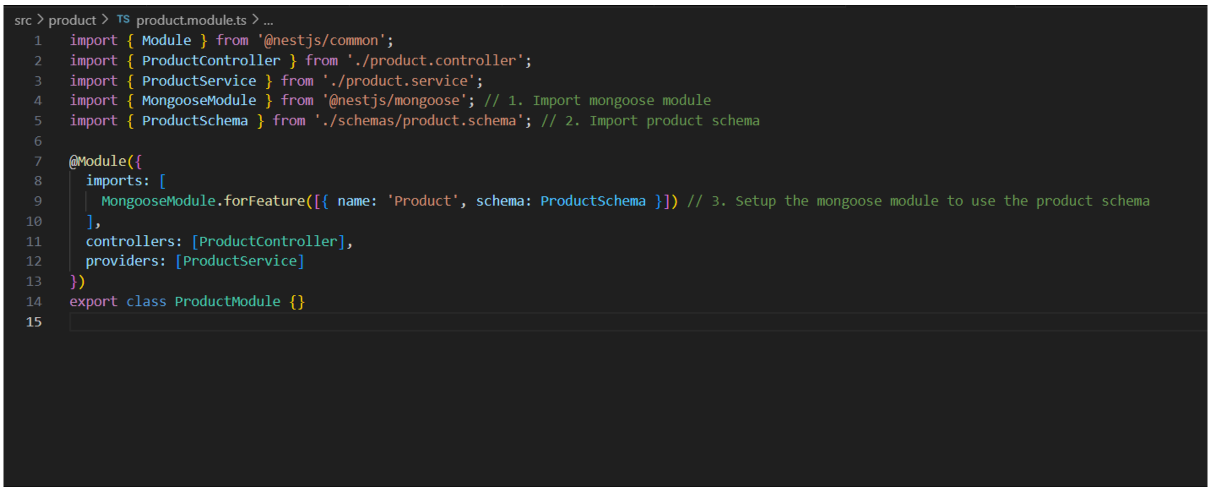
In the **product** directory, create a new **schemas** directory. Put a **product.schema.ts** file in the new directory with the following content:


src > product > schemas > product.schema.ts > ...

```typescript
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import { Document } from 'mongoose';

export type ProductDocument = Product & Document;

@Schema()
export class Product {
  @Prop()
  name: string;

  @Prop()
  description: string;

  @Prop()
  price: number;

  @Prop()
  category: string;
}

export const ProductSchema = SchemaFactory.createForClass(Product);
```

The code above creates a schema for our product with **name, description, price, and category** properties.

Here is the **product.module.ts** file:



src > product > TS product.module.ts > ...

```typescript
import { Module } from '@nestjs/common';
import { ProductController } from './product.controller';
import { ProductService } from './product.service';
import { MongooseModule } from '@nestjs/mongoose'; // 1. Import mongoose module
import { ProductSchema } from './schemas/product.schema'; // 2. Import product schema

@Module({
  imports: [
    MongooseModule.forFeature([{ name: 'Product', schema: ProductSchema }]) // 3. Setup the mongoose module to use the product schema
  ],
  controllers: [ProductController],
  providers: [ProductService]
})
export class ProductModule {}

```

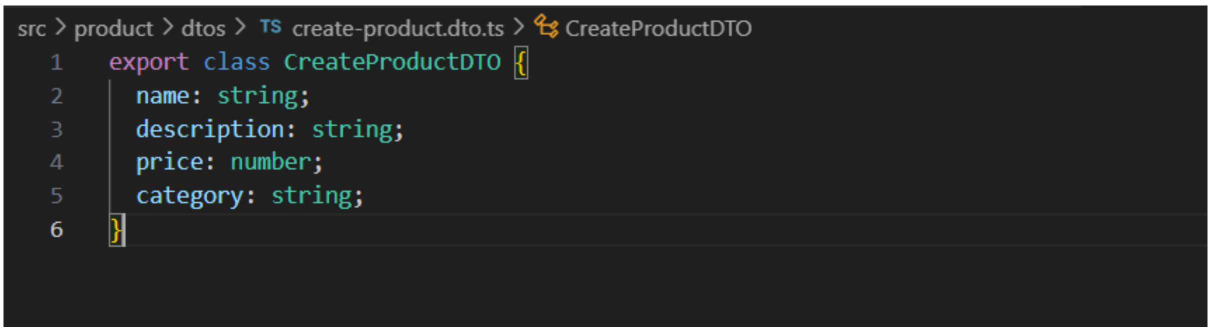
As you can see from my numbered notes, in the code above, we imported the **MongooseModule** (1) **ProductModule** (2), then set the **ProductSchema** to be used for our product model (3).

Creating product DTO files:

In addition to the product schema, we'll also need two Data Transfer Object (DTO) files for our NestJS ecommerce app. A DTO file defines the data which will be received from a form submission, a search query, and so on.

We need one DTO for product creation and another for product filtering. Let's create them now.

In the product directory, create a new **dtos** directory. Put a create**product.dto.ts** file in this new directory with the following content:

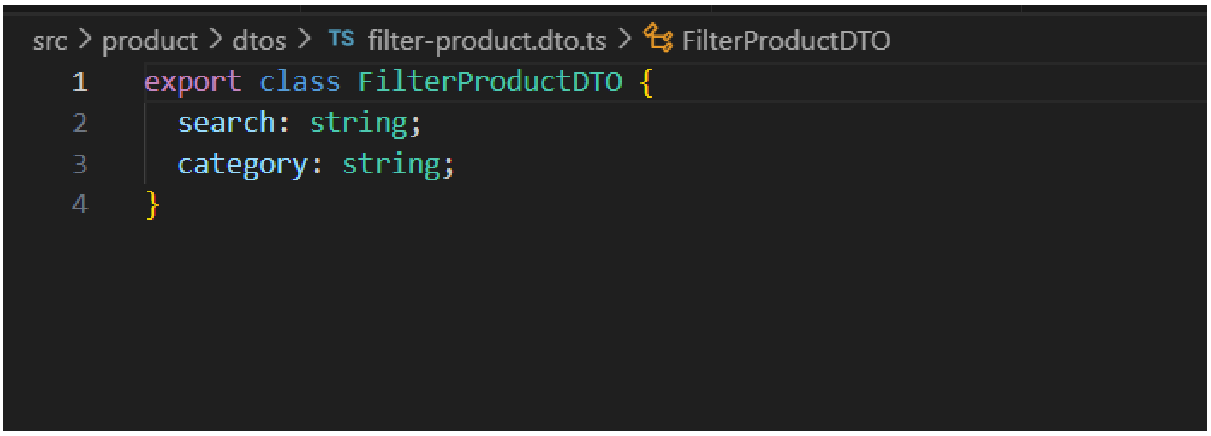


src > product > dtos > TS create-product.dto.ts > CreateProductDTO

```typescript
export class CreateProductDTO {
  name: string;
  description: string;
  price: number;
  category: string;
}
```

The above DTO defines a product object with the necessary properties for new product creation.

Then, in the same directory, create a filter-**product.dto.ts** file with the following content:



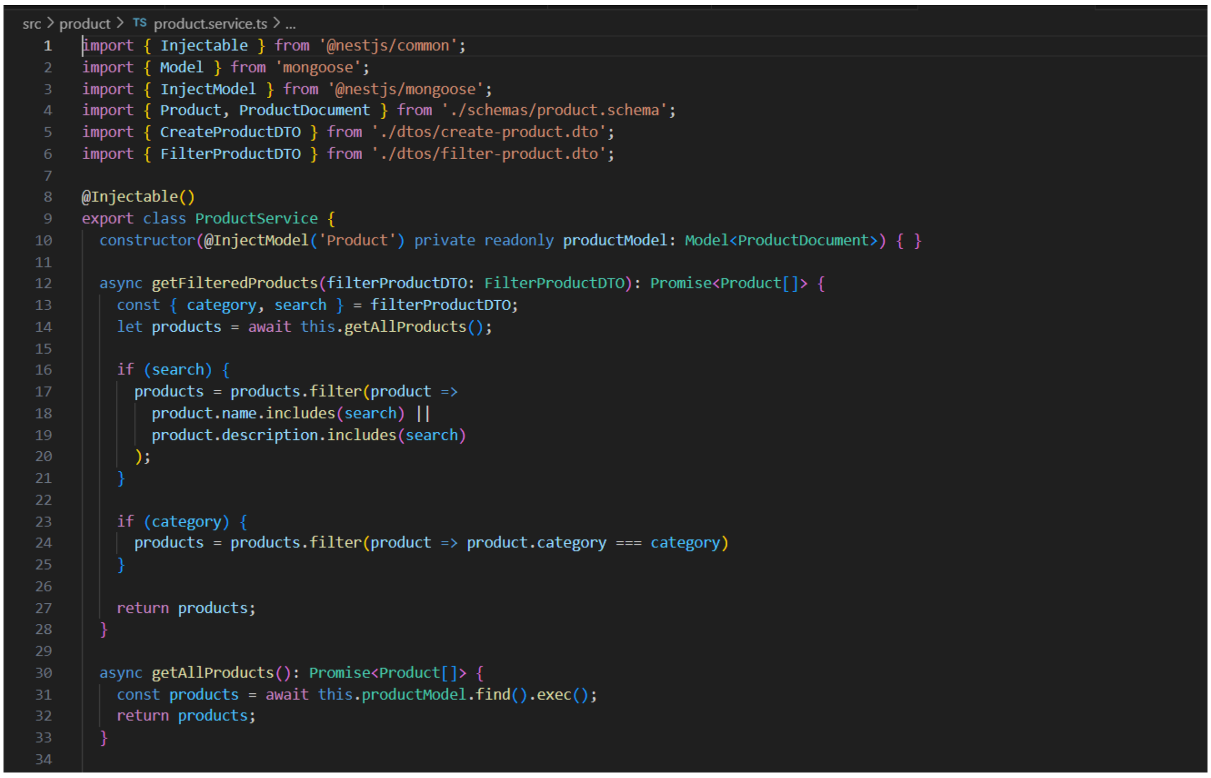
src > product > dtos > TS filter-product.dto.ts > FilterProductDTO

```typescript
export class FilterProductDTO {
  search: string;
  category: string;
}
```

This second DTO defines a filter object, which we'll use to filter the store products by search query, category, or both.

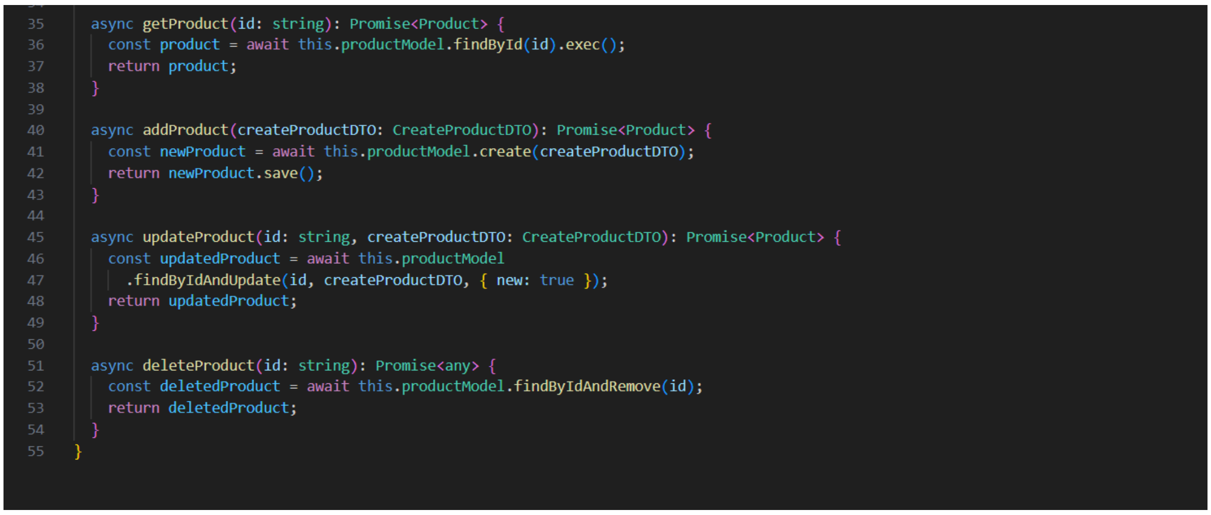
Creating product service methods:

Now let's create the actual code for product management.Open the **product.service.ts** file.



src > product > TS product.service.ts > ...

```typescript
import { Injectable } from '@nestjs/common';
import { Model } from 'mongoose';
import { InjectModel } from '@nestjs/mongoose';
import { Product, ProductDocument } from './schemas/product.schema';
import { CreateProductDTO } from './dtos/create-product.dto';
import { FilterProductDTO } from './dtos/filter-product.dto';

@Injectable()
export class ProductService {
  constructor(@InjectModel('Product') private readonly productModel: Model<ProductDocument>) { }

  async getFilteredProducts(filterProductDTO: FilterProductDTO): Promise<Product[]> {
    const { category, search } = filterProductDTO;
    let products = await this.getAllProducts();

    if (search) {
      products = products.filter(product =>
        product.name.includes(search) ||
        product.description.includes(search)
      );
    }

    if (category) {
      products = products.filter(product => product.category === category)
    }

    return products;
  }

  async getAllProducts(): Promise<Product[]> {
    const products = await this.productModel.find().exec();
    return products;
  }

```



```
  async getProduct(id: string): Promise<Product> {
    const product = await this.productModel.findById(id).exec();
    return product;
  }

  async addProduct(createProductDTO: CreateProductDTO): Promise<Product> {
    const newProduct = await this.productModel.create(createProductDTO);
    return newProduct.save();
  }

  async updateProduct(id: string, createProductDTO: CreateProductDTO): Promise<Product> {
    const updatedProduct = await this.productModel
      .findByIdAndUpdate(id, createProductDTO, { new: true });
    return updatedProduct;
  }

  async deleteProduct(id: string): Promise<any> {
    const deletedProduct = await this.productModel.findByIdAndRemove(id);
    return deletedProduct;
  }
}
```

Let’s examine the code block above piece by piece.

It injects the needed dependencies (the product model) by using the **@InjectModel** decorator.

The method **getAllProducts** is for getting all products. The second method **getProduct** is for getting a single product. We use standard Mongoose methods to achieve these actions.

The method **getFilteredProducts** below returns filtered products. Products can be filtered by search query, by category, or by both.

The next method **addProduct** below creates a new product. **addProduct** achieves this by using the class from the create-**product.dto.ts** file and saving it to the database.

The final two methods are **updateProduct** and **deleteProduct.** Using these methods, you can find a product by ID and either update it or remove it from the database.

Creating product controller methods:

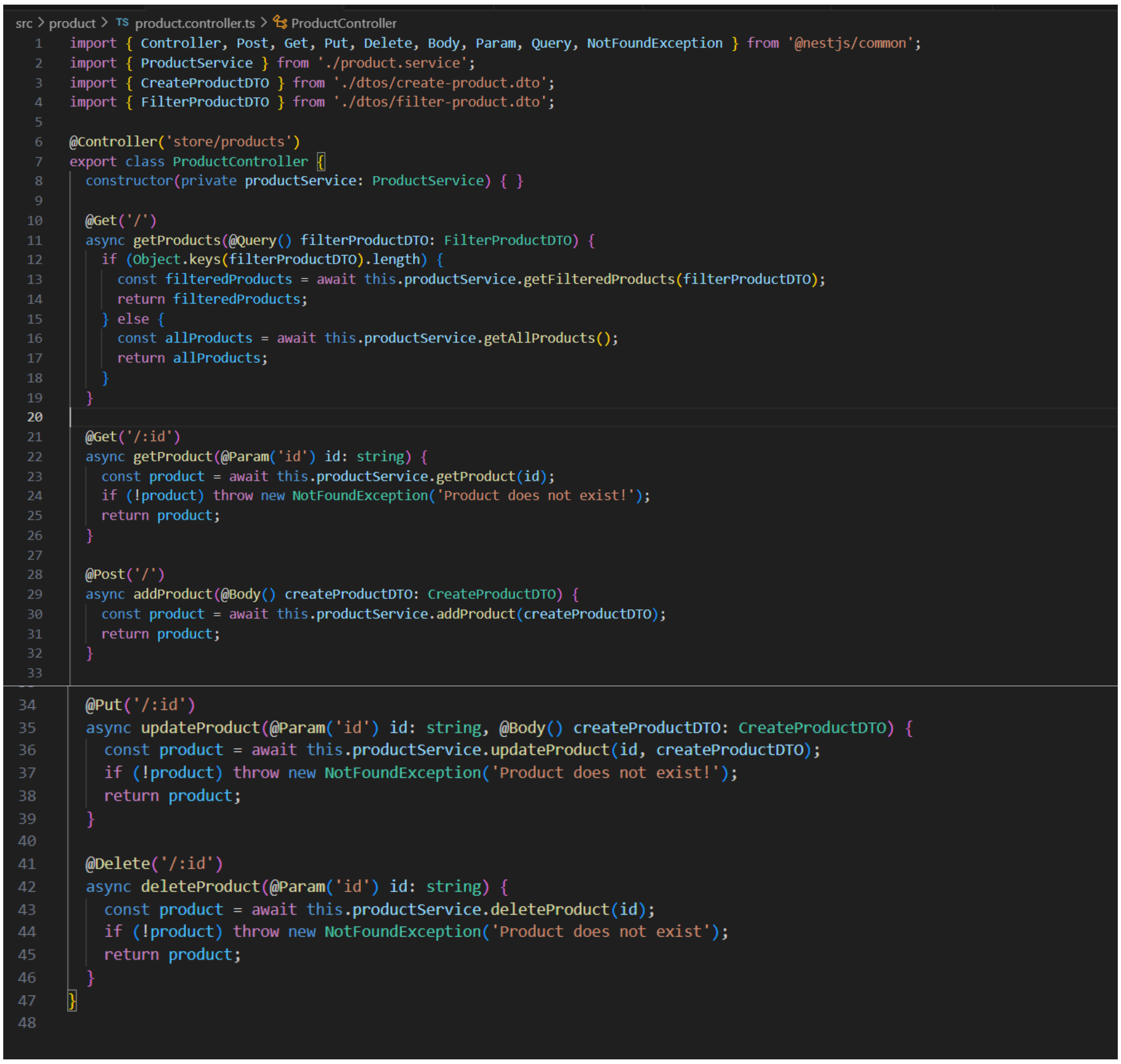
The final step for the product module is to create the API endpoints.

We’ll create the following API endpoints:

- POST **store/products/** — add new product
- GET **store/products/** — get all products
- GET **store/products/:id** — get single product
- PUT **store/products/:id** — edit single product

- DELETE **store/products/:id** — remove single product

Open the **product.controller.ts** file:


src > product > TS product.controller.ts > ProductController

```typescript
import { Controller, Post, Get, Put, Delete, Body, Param, Query, NotFoundException } from '@nestjs/common';
import { ProductService } from './product.service';
import { CreateProductDTO } from './dtos/create-product.dto';
import { FilterProductDTO } from './dtos/filter-product.dto';

@Controller('store/products')
export class ProductController {
  constructor(private productService: ProductService) { }

  @Get('/')
  async getProducts(@Query() filterProductDTO: FilterProductDTO) {
    if (Object.keys(filterProductDTO).length) {
      const filteredProducts = await this.productService.getFilteredProducts(filterProductDTO);
      return filteredProducts;
    } else {
      const allProducts = await this.productService.getAllProducts();
      return allProducts;
    }
  }

  @Get('/:id')
  async getProduct(@Param('id') id: string) {
    const product = await this.productService.getProduct(id);
    if (!product) throw new NotFoundException('Product does not exist!');
    return product;
  }

  @Post('/')
  async addProduct(@Body() createProductDTO: CreateProductDTO) {
    const product = await this.productService.addProduct(createProductDTO);
    return product;
  }

  @Put('/:id')
  async updateProduct(@Param('id') id: string, @Body() createProductDTO: CreateProductDTO) {
    const product = await this.productService.updateProduct(id, createProductDTO);
    if (!product) throw new NotFoundException('Product does not exist!');
    return product;
  }

  @Delete('/:id')
  async deleteProduct(@Param('id') id: string) {
    const product = await this.productService.deleteProduct(id);
    if (!product) throw new NotFoundException('Product does not exist');
    return product;
  }
}
```

NestJS provides a full set of JavaScript decorators to work with HTTP requests and responses (Get, Put, Body, Param, etc.), handle errors (NotFoundException), define controllers (Controller), and so on.

We imported the ones we need from **@nestjs/common** at the beginning of the file. We also import all the other files we've already created and we need: **ProductService, CreateProductDTO, and FilterProductDTO**.

We also inject the product service in the class constructor in the code above.

Next, we define the following endpoint by using the **@Get** decorator.

After defining the endpoint, we use @Query decorator in the **getProducts()** method and the object from filter-**product.dto.ts** to get the query parameters from a request.

If the query parameters from a request exist, we use **getFilteredProduct()** method from the product service. If there are no such parameters, we use the regular **getAllProducts()** method instead.

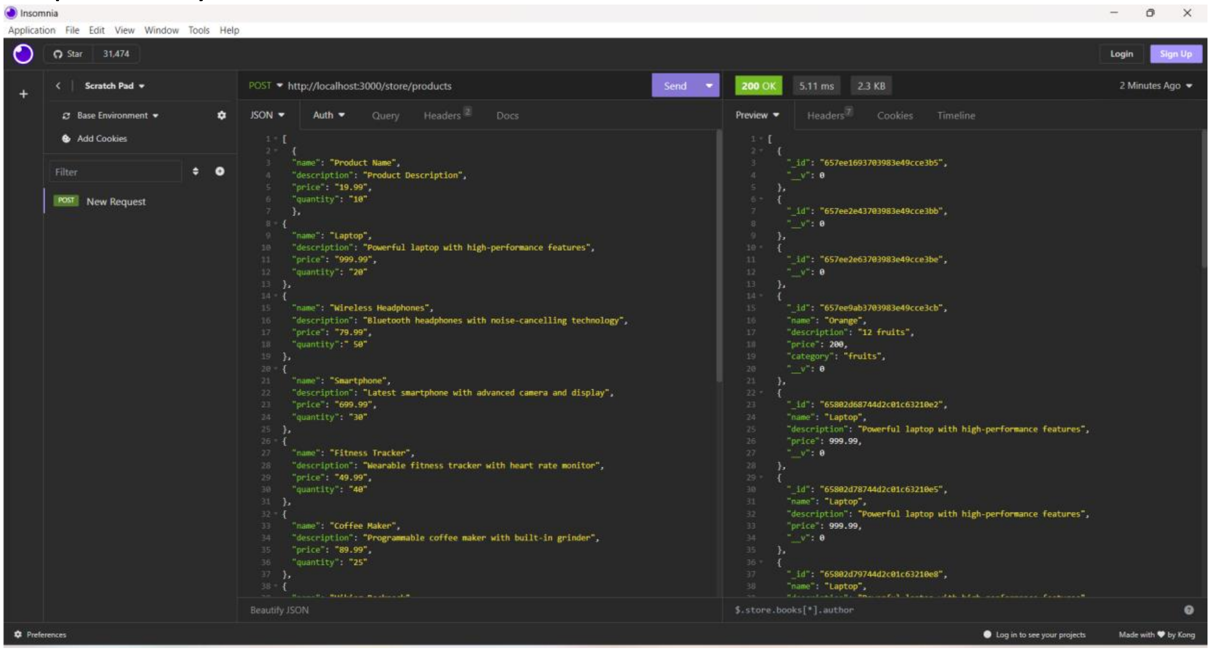
we use the @Body decorator to get the needed data from the request body and then pass it to the **addProduct()** method

we use the **@Param** decorator to get the product ID from the URL.

We then use the appropriate method from the product service to get, edit, or delete a product. If a product is not found, we use the **NotFoundException** to throw an error message.

Result:

Product Management:
Create Product
Endpoint: POST http://localhost:3000/store/products
Description: Creates a new product.
Request Body:



POST http://localhost:3000/store/products

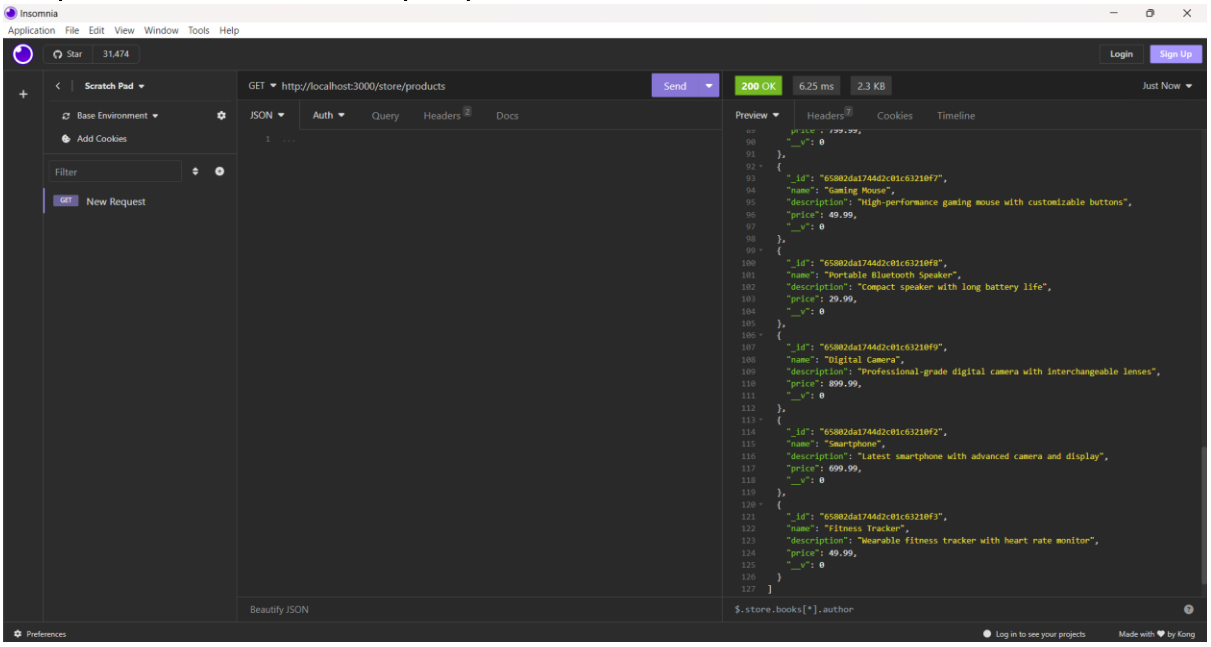
```
[
  {
  "name": "Product Name",
  "description": "Product Description",
  "price": "19.99",
  "quantity": "10"
  },
{
  "name": "Laptop",
  "description": "Powerful laptop with high-performance features",
  "price": "999.99",
  "quantity": "20"
},
{
  "name": "Wireless Headphones",
  "description": "Bluetooth headphones with noise-cancelling technology",
  "price": "79.99",
  "quantity":" 50"
},
{
  "name": "Smartphone",
  "description": "Latest smartphone with advanced camera and display",
  "price": "699.99",
  "quantity": "30"
},
{
  "name": "Fitness Tracker",
  "description": "Wearable fitness tracker with heart rate monitor",
  "price": "49.99",
  "quantity": "40"
},
{
  "name": "Coffee Maker",
  "description": "Programmable coffee maker with built-in grinder",
  "price": "89.99",
  "quantity": "25"
},
{
```

```
[
  {
    "_id": "657ee1693703983e49cce3b5",
    "__v": 0
  },
  {
    "_id": "657ee2e43703983e49cce3bb",
    "__v": 0
  },
  {
    "_id": "657ee2e63703983e49cce3be",
    "__v": 0
  },
  {
    "_id": "657ee9ab3703983e49cce3cb",
    "name": "Orange",
    "description": "12 fruits",
    "price": 200,
    "category": "fruits",
    "__v": 0
  },
  {
    "_id": "65802d68744d2c01c63210e2",
    "name": "Laptop",
    "description": "Powerful laptop with high-performance features",
    "price": 999.99,
    "__v": 0
  },
  {
    "_id": "65802d78744d2c01c63210e5",
    "name": "Laptop",
    "description": "Powerful laptop with high-performance features",
    "price": 999.99,
    "__v": 0
  },
  {
    "_id": "65802d79744d2c01c63210e8",
    "name": "Laptop",
```

$.store.books[*].author

Response: Returns the created product.

Get All Products
Endpoint: GET http://localhost:3000/store/products
Description: Retrieves a list of all products.
Response: Returns an array of products.



GET http://localhost:3000/store/products

Send

200 OK 6.25 ms 2.3 KB Just Now

JSON Auth Query Headers Docs

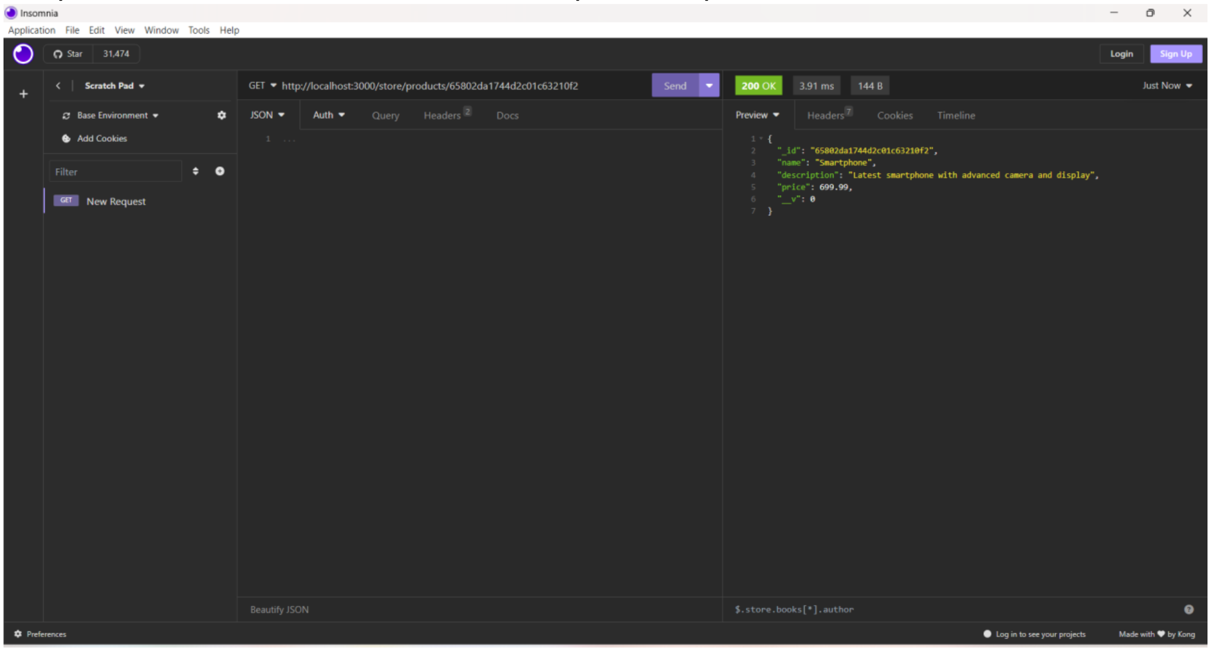
Preview Headers Cookies Timeline

```
      "price": 799.99,
      "__v": 0
   },
   {
      "_id": "65802da1744d2c01c63210f7",
      "name": "Gaming Mouse",
      "description": "High-performance gaming mouse with customizable buttons",
      "price": 49.99,
      "__v": 0
   },
   {
      "_id": "65802da1744d2c01c63210f8",
      "name": "Portable Bluetooth Speaker",
      "description": "Compact speaker with long battery life",
      "price": 29.99,
      "__v": 0
   },
   {
      "_id": "65802da1744d2c01c63210f9",
      "name": "Digital Camera",
      "description": "Professional-grade digital camera with interchangeable lenses",
      "price": 899.99,
      "__v": 0
   },
   {
      "_id": "65802da1744d2c01c63210f2",
      "name": "Smartphone",
      "description": "Latest smartphone with advanced camera and display",
      "price": 699.99,
      "__v": 0
   },
   {
      "_id": "65802da1744d2c01c63210f3",
      "name": "Fitness Tracker",
      "description": "Wearable fitness tracker with heart rate monitor",
      "price": 49.99,
      "__v": 0
   }
]
```

Beautify JSON

$.store.books[*].author

Get Single Product
Endpoint: GET http://localhost:3000/store/products/{productId}
Description: Retrieves details of a specific product.
Response: Returns the details of the specified product.



GET http://localhost:3000/store/products/65802da1744d2c01c63210f2

Send

200 OK 3.91 ms 144 B Just Now

JSON Auth Query Headers Docs

Preview Headers Cookies Timeline

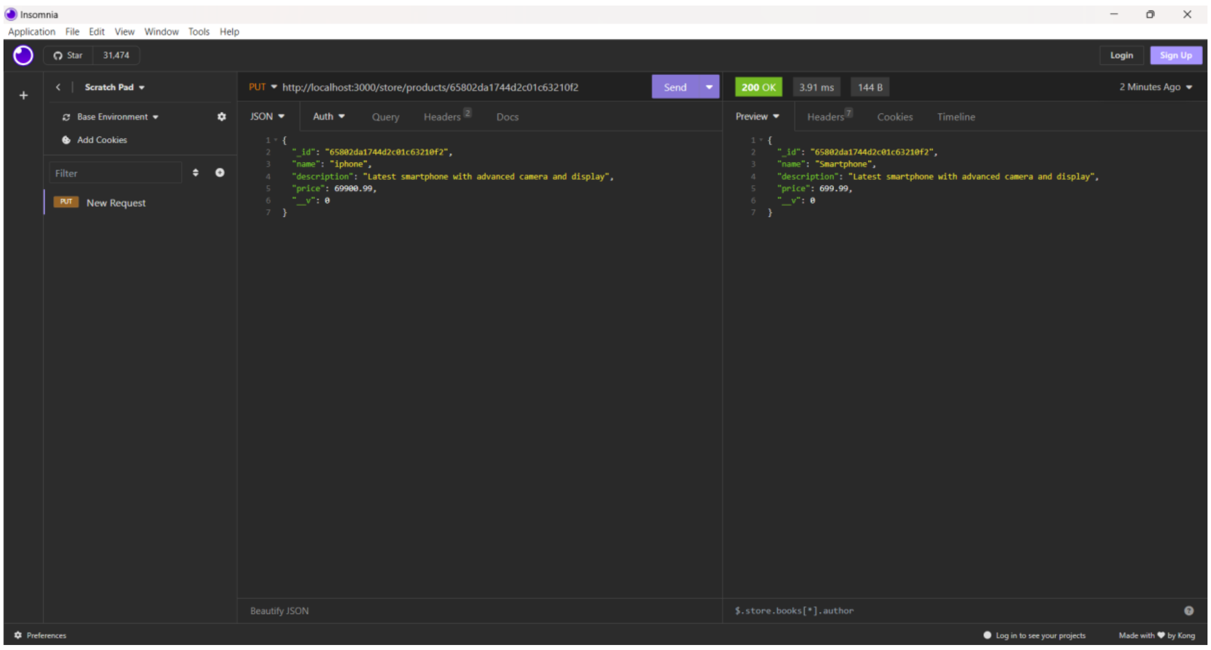
```
{
   "_id": "65802da1744d2c01c63210f2",
   "name": "Smartphone",
   "description": "Latest smartphone with advanced camera and display",
   "price": 699.99,
   "__v": 0
}
```

Beautify JSON

$.store.books[*].author

Update Product
Endpoint: PUT http://localhost:3000/store/products/{productId}

Description: Updates details of a specific product.
Request Body:



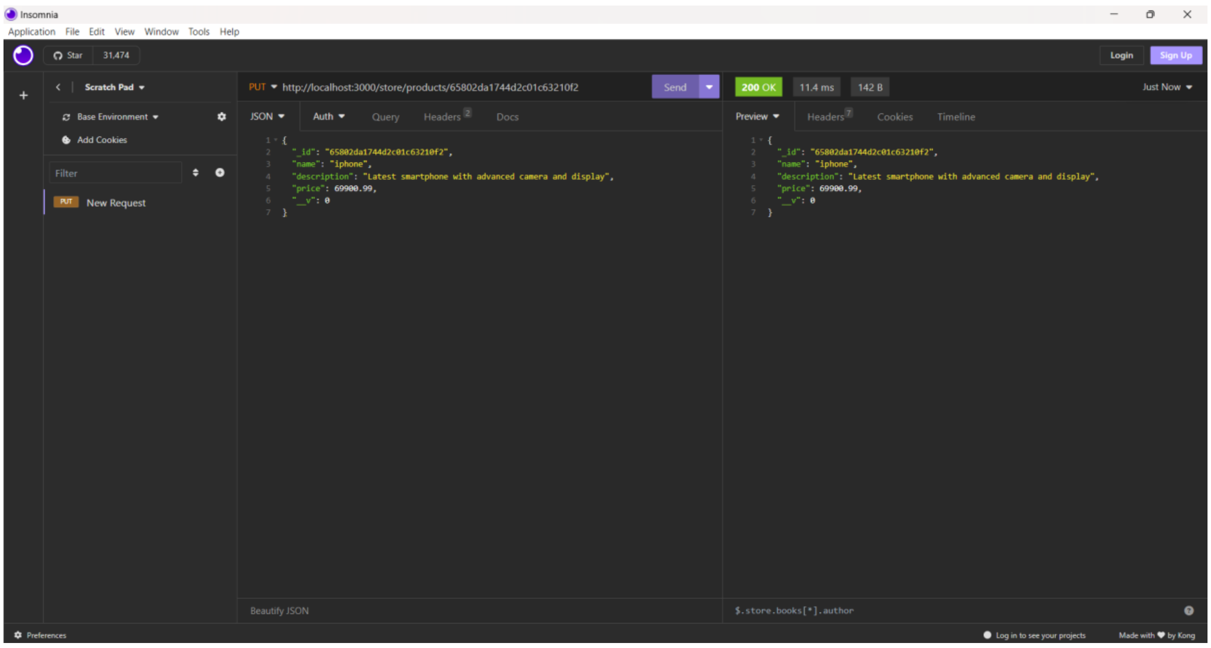
PUT http://localhost:3000/store/products/65802da1744d2c01c63210f2

```
{
  "_id": "65802da1744d2c01c63210f2",
  "name": "iphone",
  "description": "Latest smartphone with advanced camera and display",
  "price": 69900.99,
  "__v": 0
}
```

200 OK 3.91 ms 144 B

```
{
  "_id": "65802da1744d2c01c63210f2",
  "name": "Smartphone",
  "description": "Latest smartphone with advanced camera and display",
  "price": 699.99,
  "__v": 0
}
```

In the above picture, You can see the name of the product is “smart phone”. Now, we edit the product name to “Iphone”. Let’s see



PUT http://localhost:3000/store/products/65802da1744d2c01c63210f2

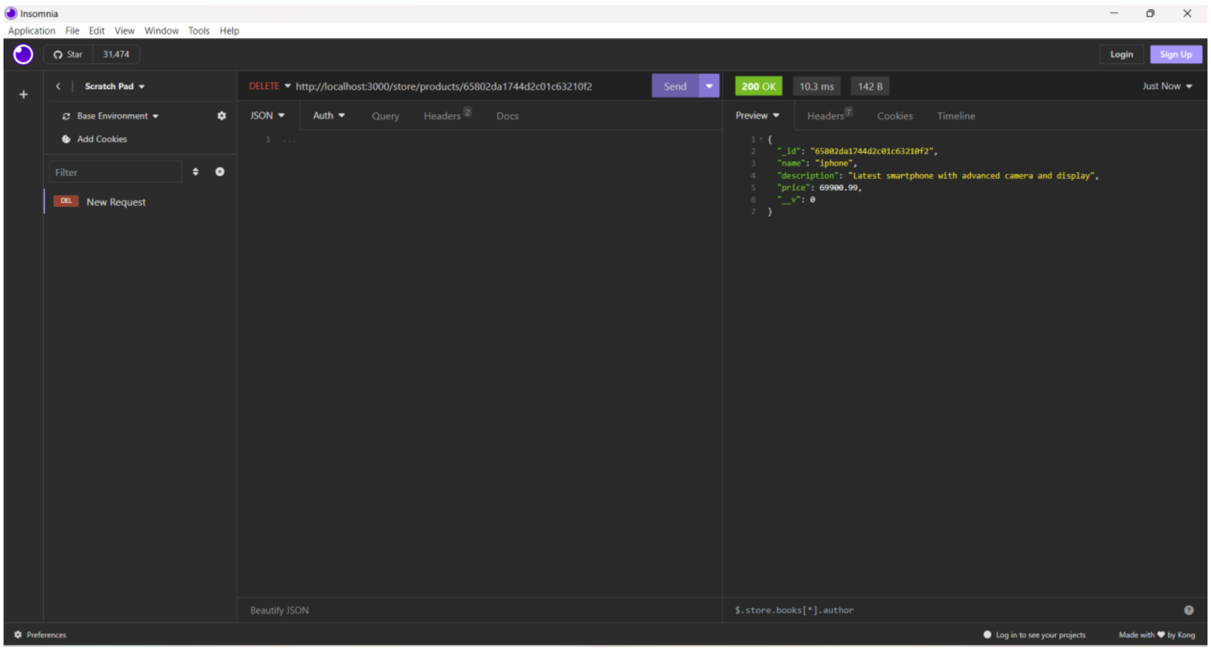
```
{
  "_id": "65802da1744d2c01c63210f2",
  "name": "iphone",
  "description": "Latest smartphone with advanced camera and display",
  "price": 69900.99,
  "__v": 0
}
```

200 OK 11.4 ms 142 B

```
{
  "_id": "65802da1744d2c01c63210f2",
  "name": "iphone",
  "description": "Latest smartphone with advanced camera and display",
  "price": 69900.99,
  "__v": 0
}
```

Response: Returns the updated product.

Delete Product
Endpoint: DELETE http://localhost:3000/store/products/{productId}
Description: Deletes a specific product.
Response: Returns a success message.

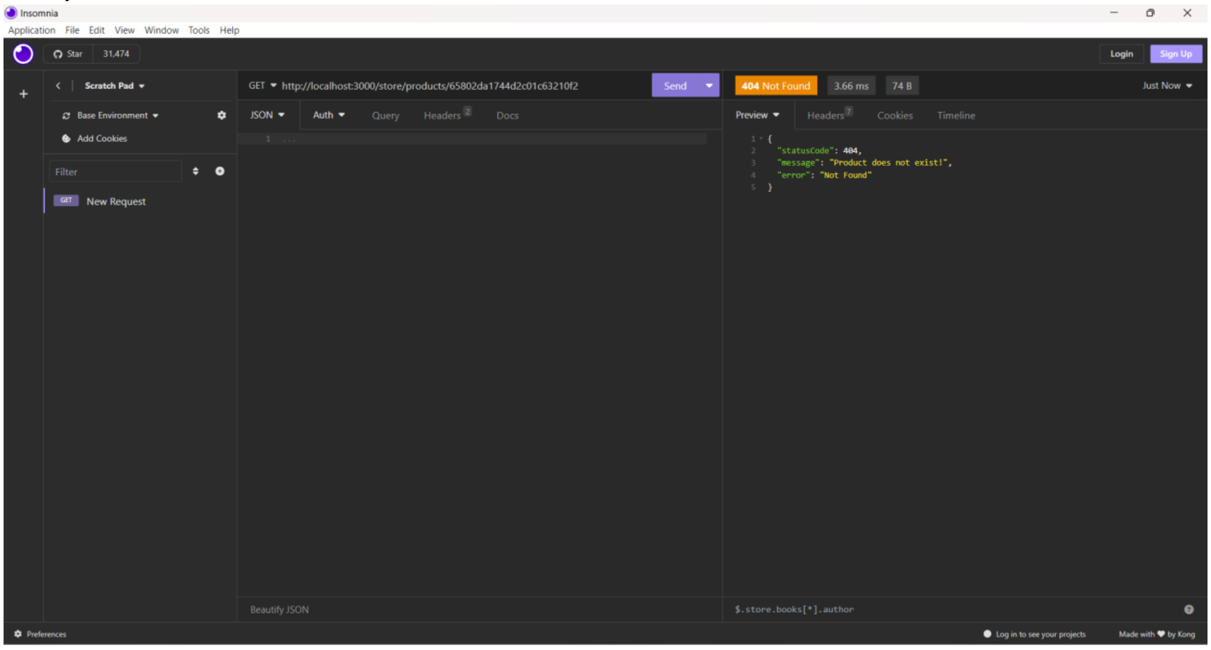


```
DELETE http://localhost:3000/store/products/65802da1744d2c01c63210f2
```

200 OK 10.3 ms 142 B

```
{
  "_id": "65802da1744d2c01c63210f2",
  "name": "iphone",
  "description": "Latest smartphone with advanced camera and display",
  "price": 69900.99,
  "__v": 0
}
```

Here, we are deleted this product. Now, will check using GET method whether the product is deleted or not.



```
GET http://localhost:3000/store/products/65802da1744d2c01c63210f2
```

404 Not Found 3.66 ms 74 B

```
{
  "statusCode": 404,
  "message": "Product does not exist!",
  "error": "Not Found"
}
```

Here is the result, Product doesn't exist.

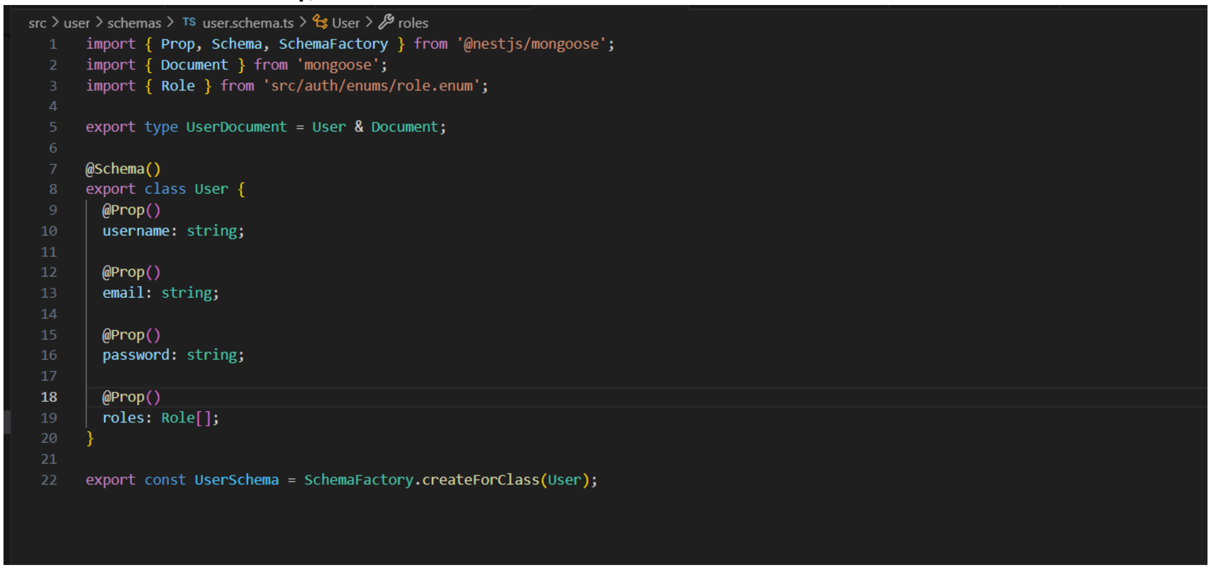
Creating the user management feature:

Run the following commands to generate the necessary module and service:

nest g module user

nest g service user --no-spec

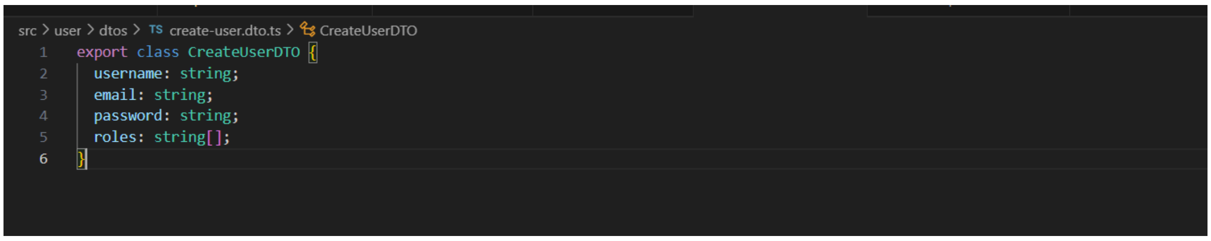
In the user directory, create a schemas folder and add a **user.schema.ts** file:



src > user > schemas > TS user.schema.ts > User > roles

```typescript
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import { Document } from 'mongoose';
import { Role } from 'src/auth/enums/role.enum';

export type UserDocument = User & Document;

@Schema()
export class User {
  @Prop()
  username: string;

  @Prop()
  email: string;

  @Prop()
  password: string;

  @Prop()
  roles: Role[];
}

export const UserSchema = SchemaFactory.createForClass(User);
```

Next, create a dtos folder in the user directory and add a create-**user-dto.ts** file:

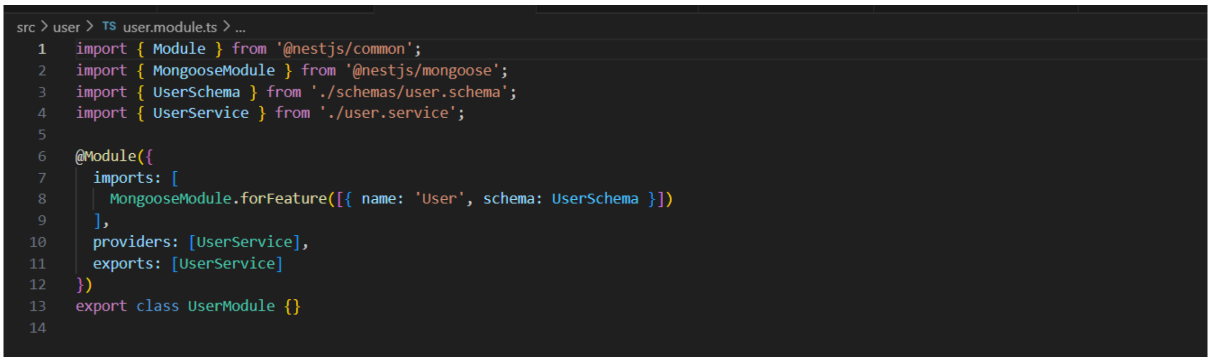


src > user > dtos > TS create-user.dto.ts > CreateUserDTO

```typescript
export class CreateUserDTO {
  username: string;
  email: string;
  password: string;
  roles: string[];
}
```

Configuring Resources

Open **user.module.ts** and configure the schema:



src > user > TS user.module.ts > ...

```typescript
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';
import { UserSchema } from './schemas/user.schema';
import { UserService } from './user.service';

@Module({
  imports: [
    MongooseModule.forFeature([{ name: 'User', schema: UserSchema }])
  ],
  providers: [UserService],
  exports: [UserService]
})
export class UserModule {}
```

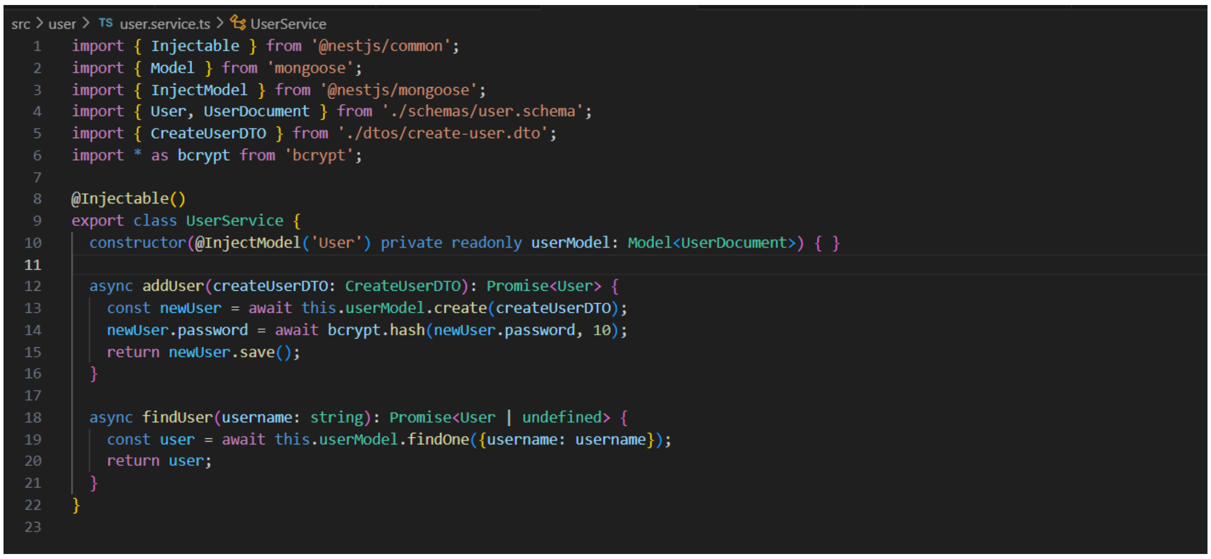
We'll also need to install two additional packages: **bcrypt** and **@types/bcrypt**:

```
npm install bcrypt
npm install -D @types/bcrypt
```

These packages enable us to keep the password saved, which we will work on in the next section.

Creating user service methods

Now let's add the logic for the user management. Open the **user.service.ts** file and replace its content with the following:



src > user > TS user.service.ts > UserService

```
import { Injectable } from '@nestjs/common';
import { Model } from 'mongoose';
import { InjectModel } from '@nestjs/mongoose';
import { User, UserDocument } from './schemas/user.schema';
import { CreateUserDTO } from './dtos/create-user.dto';
import * as bcrypt from 'bcrypt';

@Injectable()
export class UserService {
  constructor(@InjectModel('User') private readonly userModel: Model<UserDocument>) { }

  async addUser(createUserDTO: CreateUserDTO): Promise<User> {
    const newUser = await this.userModel.create(createUserDTO);
    newUser.password = await bcrypt.hash(newUser.password, 10);
    return newUser.save();
  }

  async findUser(username: string): Promise<User | undefined> {
    const user = await this.userModel.findOne({username: username});
    return user;
  }
}

```

We have added two methods in the code above. The **addUser()** method creates a new user, encrypts the new user's password by using **bcrypt.hash()**, and then saves the user to the database.

The **findUser()** method finds a particular user by the username.

Creating User Authentication and Authorization

Authentication:

Install Passport packages and dotenv:

npm install --save @nestjs/passport passport passport-local

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

npm install dotenv

Create the authentication resources:

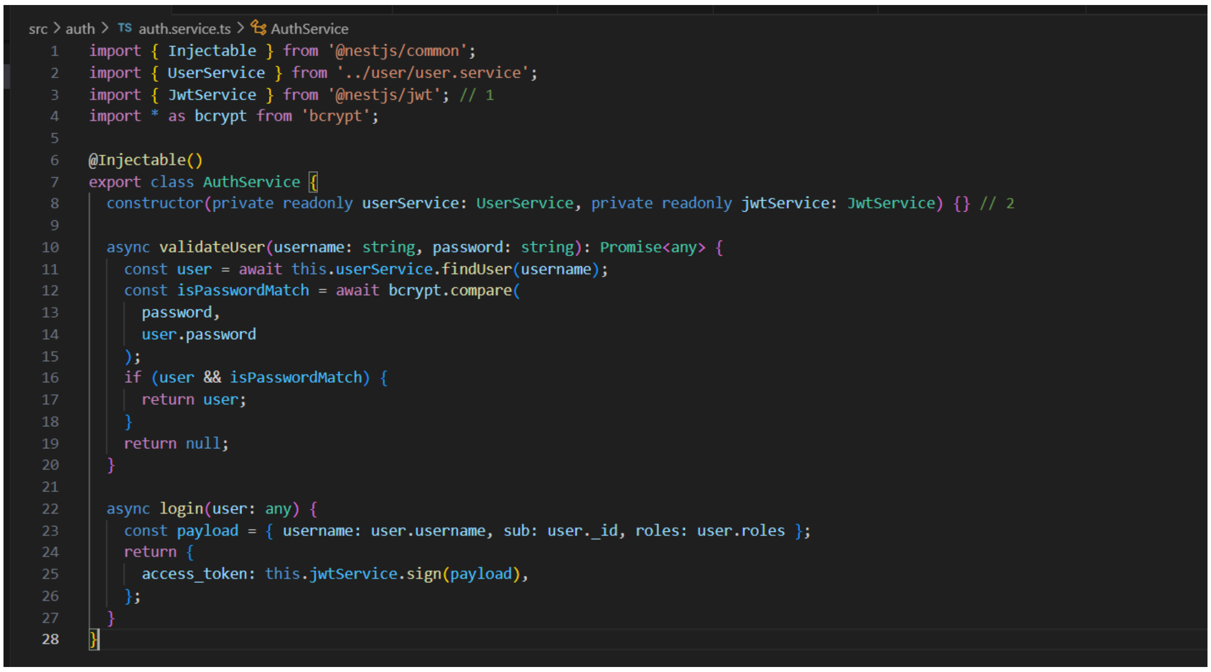
As usual, let's start by creating the needed resources for our auth feature:

nest g module auth

nest g service auth --no-spec

nest g controller auth --no-spec

Open the **auth.service.ts** file and replace its content with the following:



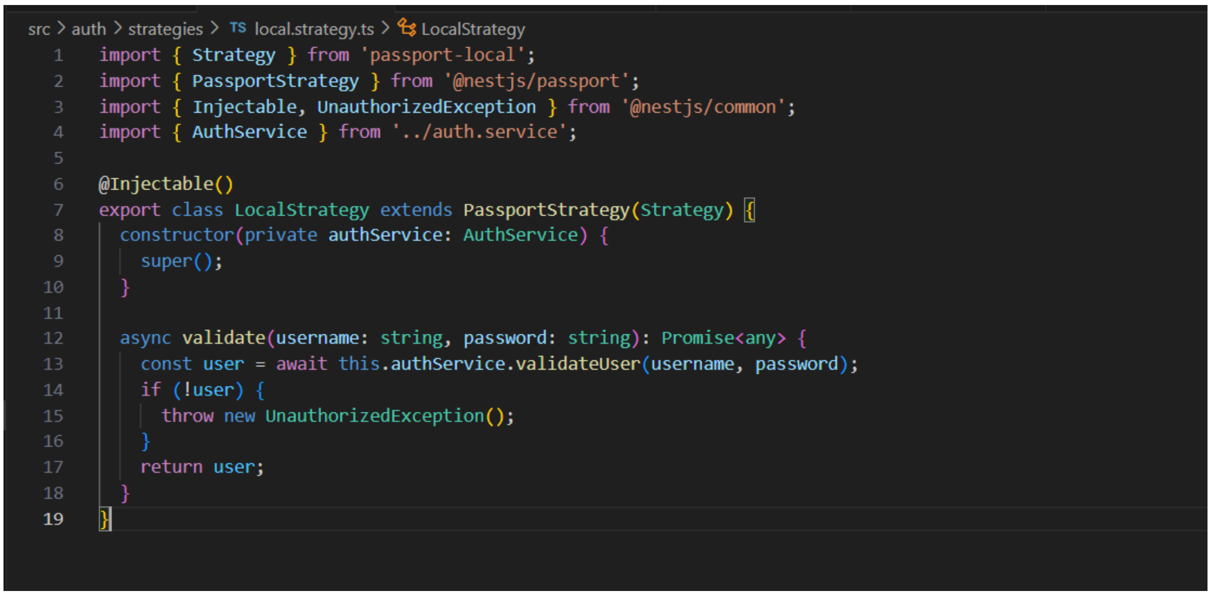
```typescript
import { Injectable } from '@nestjs/common';
import { UserService } from '../user/user.service';
import { JwtService } from '@nestjs/jwt'; // 1
import * as bcrypt from 'bcrypt';

@Injectable()
export class AuthService {
  constructor(private readonly userService: UserService, private readonly jwtService: JwtService) {} // 2

  async validateUser(username: string, password: string): Promise<any> {
    const user = await this.userService.findUser(username);
    const isPasswordMatch = await bcrypt.compare(
      password,
      user.password
    );
    if (user && isPasswordMatch) {
      return user;
    }
    return null;
  }

  async login(user: any) {
    const payload = { username: user.username, sub: user._id, roles: user.roles };
    return {
      access_token: this.jwtService.sign(payload),
    };
  }
}
```

The code above gives us a user validation method, which retrieves the user and verifies the user's password.

Imported the **JwtService** (see //1)
Added **JwtService** to the constructor (see //2).
We then used the **login()** method to sign a JWT.

Local Authentication Strategy:

Create **local.strategy.ts** in the strategies folder:


src > auth > strategies > local.strategy.ts > LocalStrategy

```
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();
  }

  async validate(username: string, password: string): Promise<any> {
    const user = await this.authService.validateUser(username, password);
    if (!user) {
      throw new UnauthorizedException();
    }
    return user;
  }
}
```

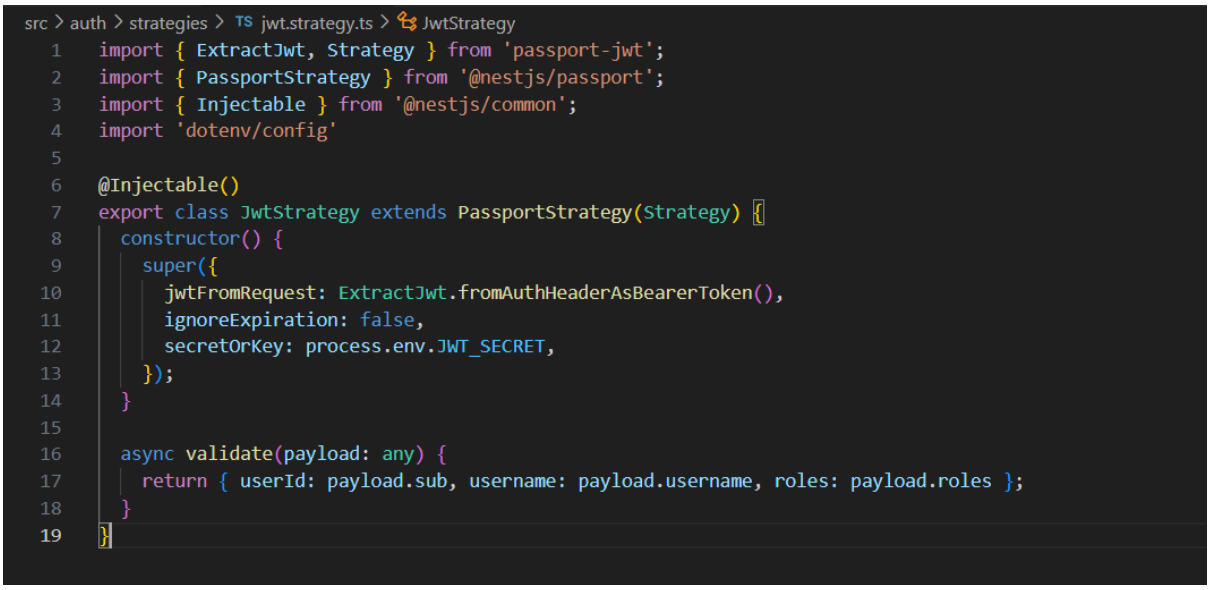
JWT Authentication Strategy

Install JWT packages:

Let's install the necessary packages:

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

Next, in the strategies directory, create a **jwt.strategy.ts** file with the following content:


src > auth > strategies > jwt.strategy.ts > JwtStrategy

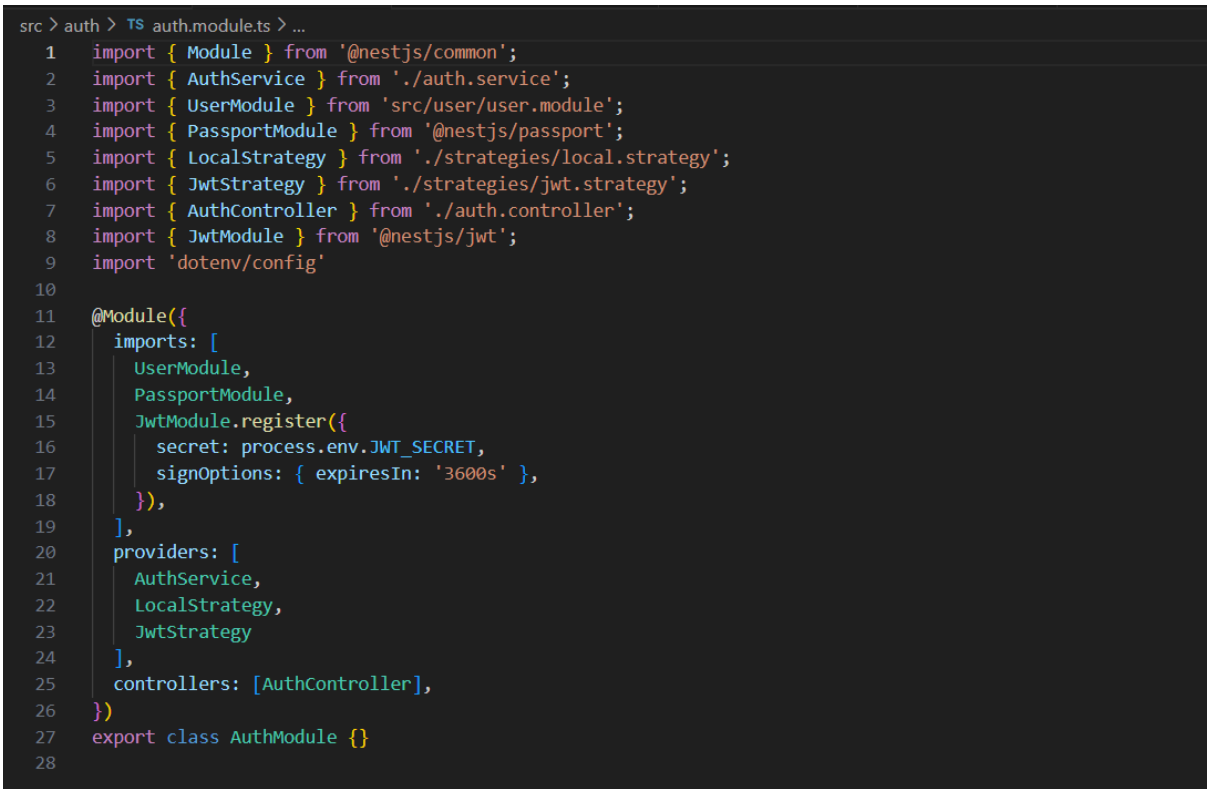
```
import { ExtractJwt, Strategy } from 'passport-jwt';
import { PassportStrategy } from '@nestjs/passport';
import { Injectable } from '@nestjs/common';
import 'dotenv/config'

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor() {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      secretOrKey: process.env.JWT_SECRET,
    });
  }

  async validate(payload: any) {
    return { userId: payload.sub, username: payload.username, roles: payload.roles };
  }
}
```

In the code above, we set an options object with the following properties:

jwtFromRequest tells the Passport module how JWT will be extracted from the request (in this case, as a bearer token)

ignoreExpiration set to false means the responsibility of ensuring that a JWT has not expired is delegated to the Passport module
secretOrKey is used to sign the token
The **validate()** method returns a payload, which is the JWT decoded as JSON. We then use this payload to return a user object with the necessary properties.

we need to update the **auth.module.ts:**



src > auth > TS auth.module.ts > ...

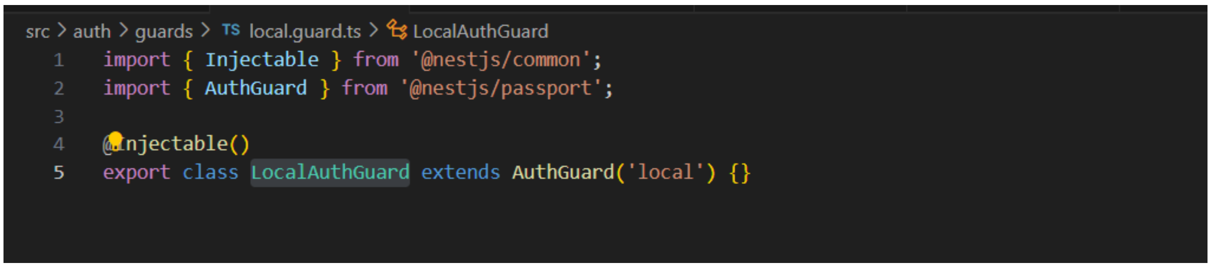
```
import { Module } from '@nestjs/common';
import { AuthService } from './auth.service';
import { UserModule } from 'src/user/user.module';
import { PassportModule } from '@nestjs/passport';
import { LocalStrategy } from './strategies/local.strategy';
import { JwtStrategy } from './strategies/jwt.strategy';
import { AuthController } from './auth.controller';
import { JwtModule } from '@nestjs/jwt';
import 'dotenv/config'

@Module({
  imports: [
    UserModule,
    PassportModule,
    JwtModule.register({
      secret: process.env.JWT_SECRET,
      signOptions: { expiresIn: '3600s' },
    }),
  ],
  providers: [
    AuthService,
    LocalStrategy,
    JwtStrategy
  ],
  controllers: [AuthController],
})
export class AuthModule {}
```

In the code above, we added **UserModule, PassportModule, and JwtModule** in the **imports** array.
We also used the **register()** method to provide the necessary options:
the **secret** key and **signOptions** object, which set the token expiration to `3600s`, or 1 hour.
Finally, we added **LocalStrategy** and **JwtStrategy** in the **providers** array.

Creating local and JWT guards:

To use the strategies we've just created, we'll need to create Guards.

In auth directory, create a new guards folder. Add a **local.guard.ts** file to this new folder with the following content:



```
src > auth > guards > TS local.guard.ts > LocalAuthGuard
import { Injectable } from '@nestjs/common';
import { AuthGuard } from '@nestjs/passport';

@Injectable()
export class LocalAuthGuard extends AuthGuard('local') {}
```

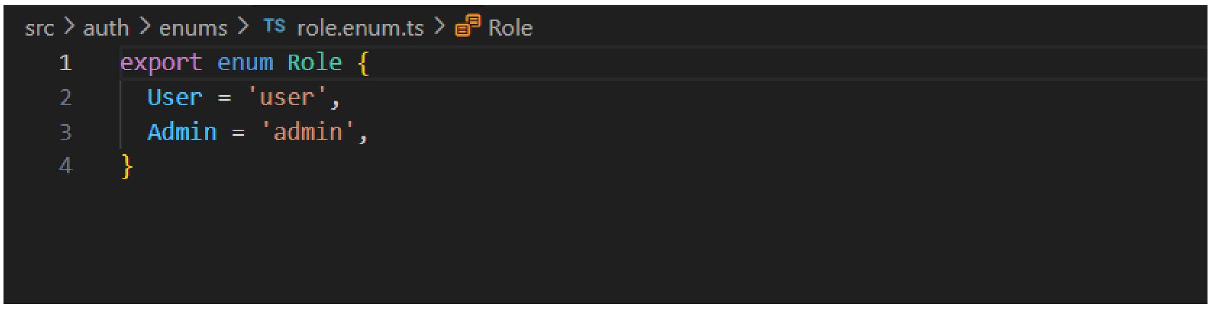
let's create the user authorization functionality.

Creating user roles management:

To implement this feature in our NestJS ecommerce app, we'll use role-based access control.

For this feature, we'll need three files: **role.enum.ts, roles.decorator.ts,** and **roles.guard.ts.** Let's start with the **role.enum.ts** file.

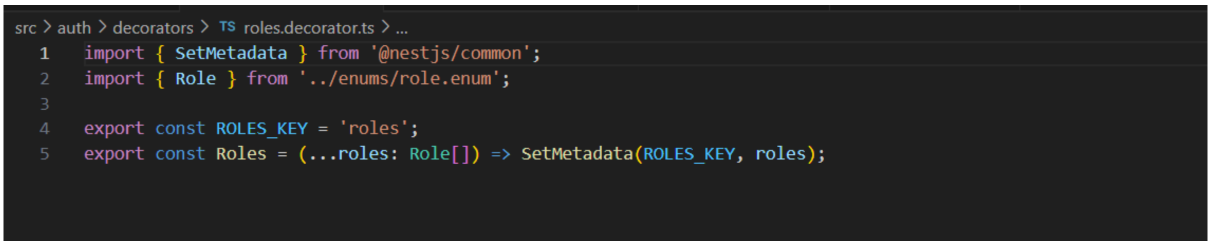
In the auth directory, create a new **enums** folder. Add a **role.enum.ts** file in this new folder with the following content:
User Management:



```
src > auth > enums > TS role.enum.ts > Role
export enum Role {
  User = 'user',
  Admin = 'admin',
}
```

This represents the available roles for registered users.

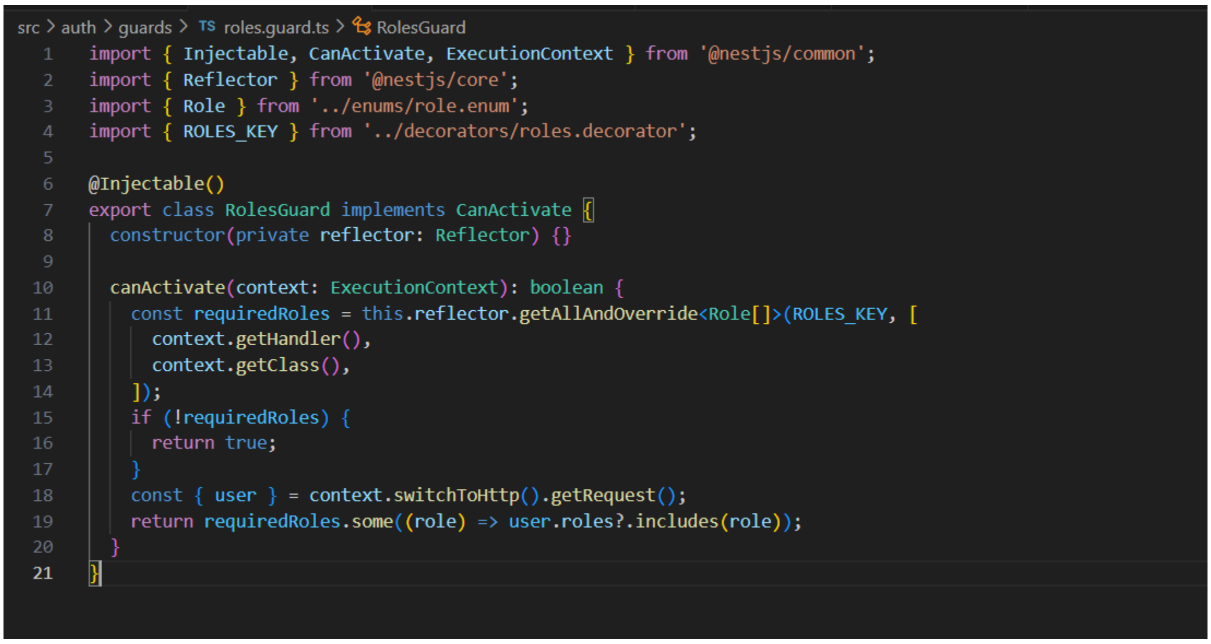
Next, in the **auth** directory, create a new **decorators** folder. Add a **roles.decorator.ts** file in this new folder with the following content:



```
src > auth > decorators > TS roles.decorator.ts > ...
import { SetMetadata } from '@nestjs/common';
import { Role } from '../enums/role.enum';

export const ROLES_KEY = 'roles';
export const Roles = (...roles: Role[]) => SetMetadata(ROLES_KEY, roles);
```

In the code above, we used `SetMetadata()` to create the decorator.

Finally, in the `guards` directory, create a **`roles.guard.ts`** file with the following content:

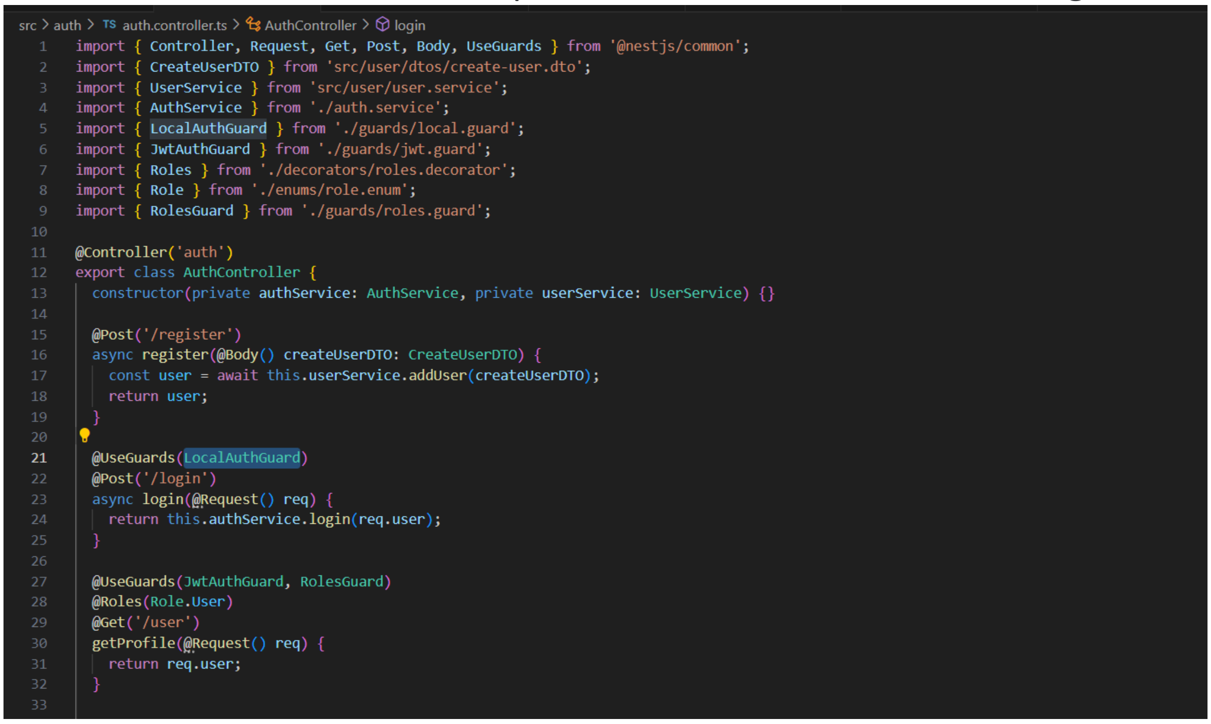


```
src > auth > guards > TS roles.guard.ts > RolesGuard
import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { Role } from '../enums/role.enum';
import { ROLES_KEY } from '../decorators/roles.decorator';

@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const requiredRoles = this.reflector.getAllAndOverride<Role[]>(ROLES_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (!requiredRoles) {
      return true;
    }
    const { user } = context.switchToHttp().getRequest();
    return requiredRoles.some((role) => user.roles?.includes(role));
  }
}
```

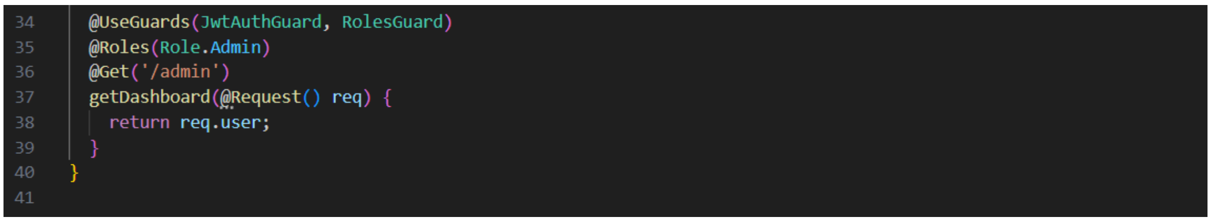
In the code above, we used the Reflector helper class to access the route's roles. We also switched the execution context to HTTP with switchToHttp() to get the user details using getRequest(). Finally, we returned the user's roles.

Controller methods:

Our last step in this section is to create the controller methods. Open the **auth.controller.ts** file and replace its content with the following:



```
src > auth > TS auth.controller.ts > AuthController > login
import { Controller, Request, Get, Post, Body, UseGuards } from '@nestjs/common';
import { CreateUserDTO } from 'src/user/dtos/create-user.dto';
import { UserService } from 'src/user/user.service';
import { AuthService } from './auth.service';
import { LocalAuthGuard } from './guards/local.guard';
import { JwtAuthGuard } from './guards/jwt.guard';
import { Roles } from './decorators/roles.decorator';
import { Role } from './enums/role.enum';
import { RolesGuard } from './guards/roles.guard';

@Controller('auth')
export class AuthController {
  constructor(private authService: AuthService, private userService: UserService) {}

  @Post('/register')
  async register(@Body() createUserDTO: CreateUserDTO) {
    const user = await this.userService.addUser(createUserDTO);
    return user;
  }

  @UseGuards(LocalAuthGuard)
  @Post('/login')
  async login(@Request() req) {
    return this.authService.login(req.user);
  }

  @UseGuards(JwtAuthGuard, RolesGuard)
  @Roles(Role.User)
  @Get('/user')
  getProfile(@Request() req) {
    return req.user;
  }

```



```
  @UseGuards(JwtAuthGuard, RolesGuard)
  @Roles(Role.Admin)
  @Get('/admin')
  getDashboard(@Request() req) {
    return req.user;
  }
}
```

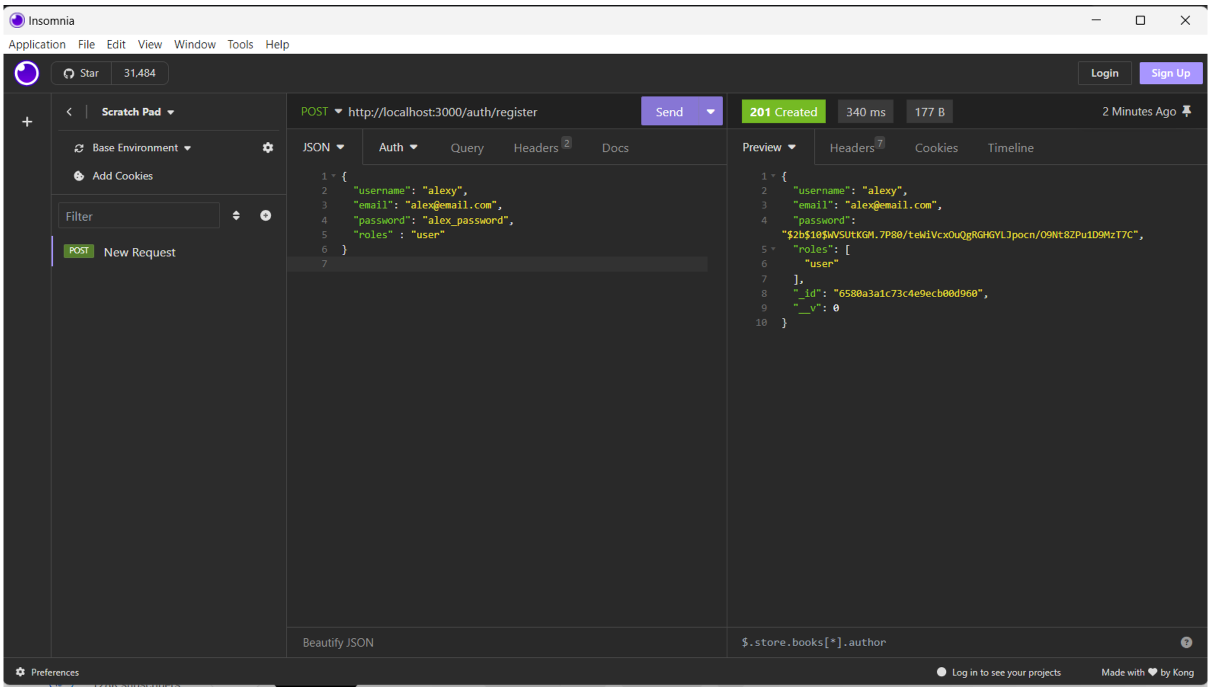
We have four endpoints in the code above:

- POST **auth/register** is used to create a new user
- POST **auth/login** is used to log in a registered user
  - To verify the user, we use the LocalAuthGuard
- GET **auth/user** is used to access the user's profile
  - We used JwtGuard to authenticate the user
  - We used RolesGuard plus @Roles decorator to provide the appropriate authorization depending on the user's roles
- GET **auth/admin** is used to access the admin dashboard
  - We also used JwtGuard and RolesGuard as done in the previous endpoint

Results:

Create a New User (Registration):

Method: POST
URL: http://localhost:3000/auth/register
Payload: Send a POST request to this URL with the necessary user details in the request body (e.g., username, email, password, roles).



POST http://localhost:3000/auth/register

201 Created 340 ms 177 B 2 Minutes Ago

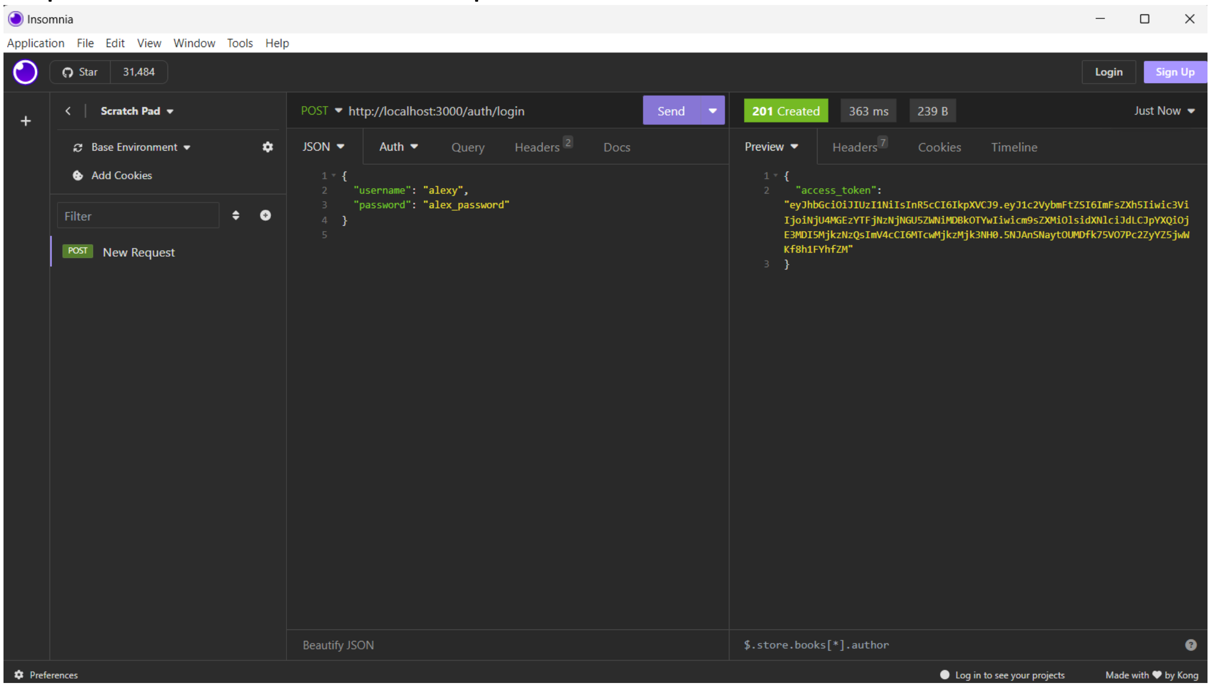
```
{
  "username": "alexy",
  "email": "alex@email.com",
  "password": "alex_password",
  "roles" : "user"
}
```

```
{
  "username": "alexy",
  "email": "alex@email.com",
  "password":
"$2b$10$WVSUtKGM.7P80/teWiVcxOuQgRGHGYLJpocn/O9Nt8ZPu1D9MzT7C",
  "roles": [
    "user"
  ],
  "_id": "6580a3a1c73c4e9ecb00d960",
  "__v": 0
}
```

Here I posted some data in the JSON format { Username, Email, Password }.

Now Let's try to login:

Method: POST
URL: http://localhost:3000/auth/login
Payload: Send a POST request to this URL with the username and password of the registered user in the request body.
Note: This will return an access token that you need to include in subsequent requests for authenticated endpoints.



POST http://localhost:3000/auth/login

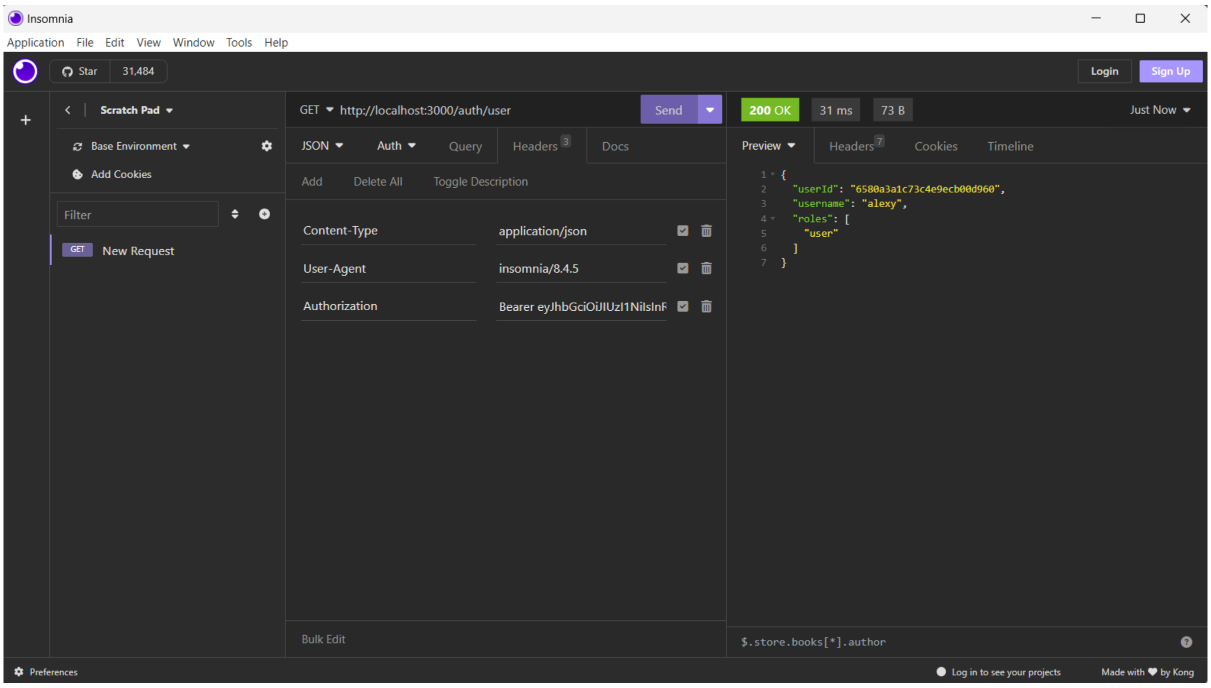
```
{
  "username": "alexy",
  "password": "alex_password"
}
```

201 Created 363 ms 239 B Just Now

```
{
  "access_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImFsZXh5Iiwic3ViIjoiNjU4MGEzYTFjNzNjNGU5ZWNiMDBkOTYwIiwicm9sZXMiOlsidXNlciJdLCJpYXQiOjE3MDI5MjkzNzQsImV4cCI6MTcwMjkzMjk3NH0.5NJAnSNaytOUMDfk75VO7Pc2ZyYZ5jwWKf8h1FYhfZM"
}
```

Access User's Profile:

Method: GET
URL: http://localhost:3000/auth/user
Headers: Include the access token obtained during the login in the Authorization header as a Bearer token.
Note: This endpoint requires authentication using JWT (Json Web Token) and is protected using JwtAuthGuard.


GET http://localhost:3000/auth/user

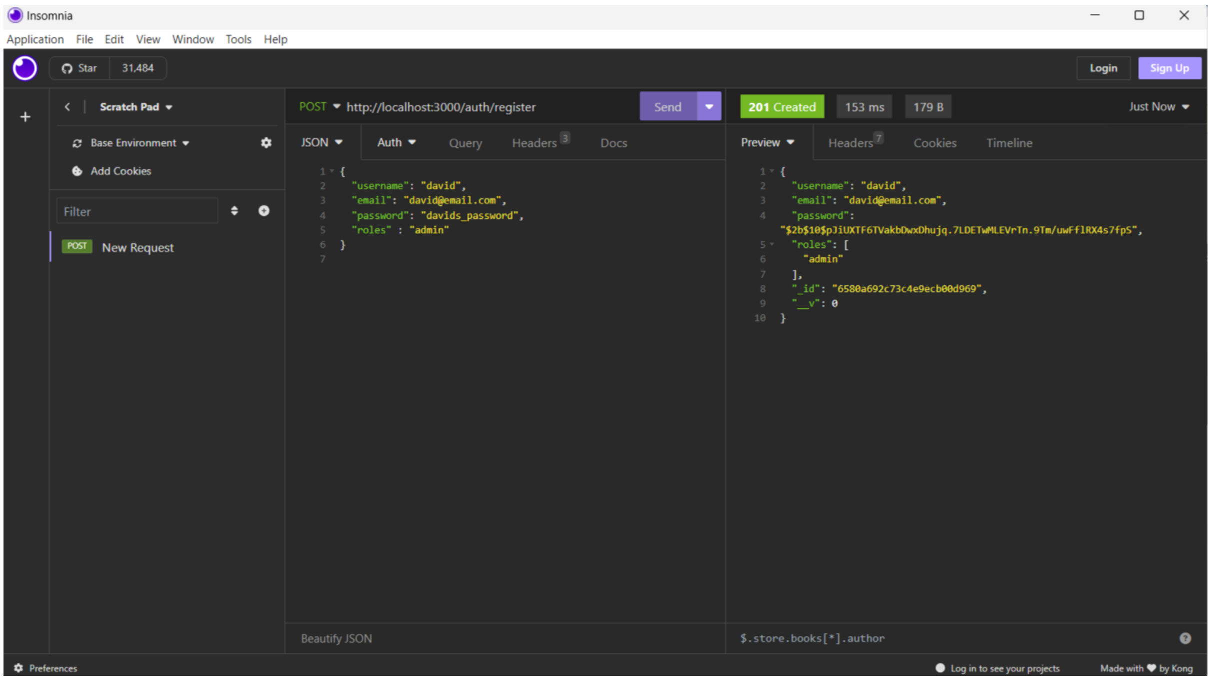
Content-Type	application/json
User-Agent	insomnia/8.4.5
Authorization	Bearer eyJhbGciOiJIUzI1NiIsInF

200 OK 31 ms 73 B

```
{
  "userId": "6580a3a1c73c4e9ecb00d960",
  "username": "alexy",
  "roles": [
    "user"
  ]
}
```

Access Admin Dashboard:

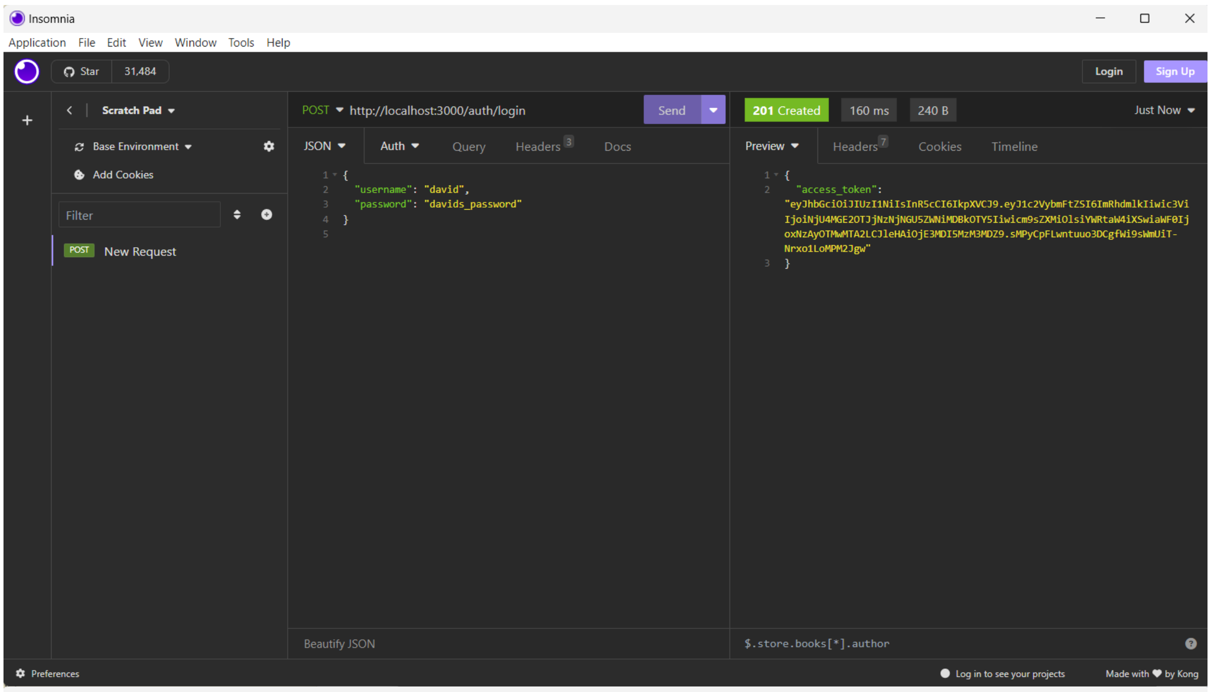
Method: GET
URL: http://localhost:3000/auth/admin
Headers: Include the access token obtained during the login in the Authorization header as a Bearer token.
Note: This endpoint requires authentication using JWT and is also protected using JwtAuthGuard. Additionally, it requires the user to have the "admin" role, which is checked by the RolesGuard and the @Roles decorator.


POST http://localhost:3000/auth/register

```
{
  "username": "david",
  "email": "david@email.com",
  "password": "davids_password",
  "roles" : "admin"
}
```

201 Created 153 ms 179 B

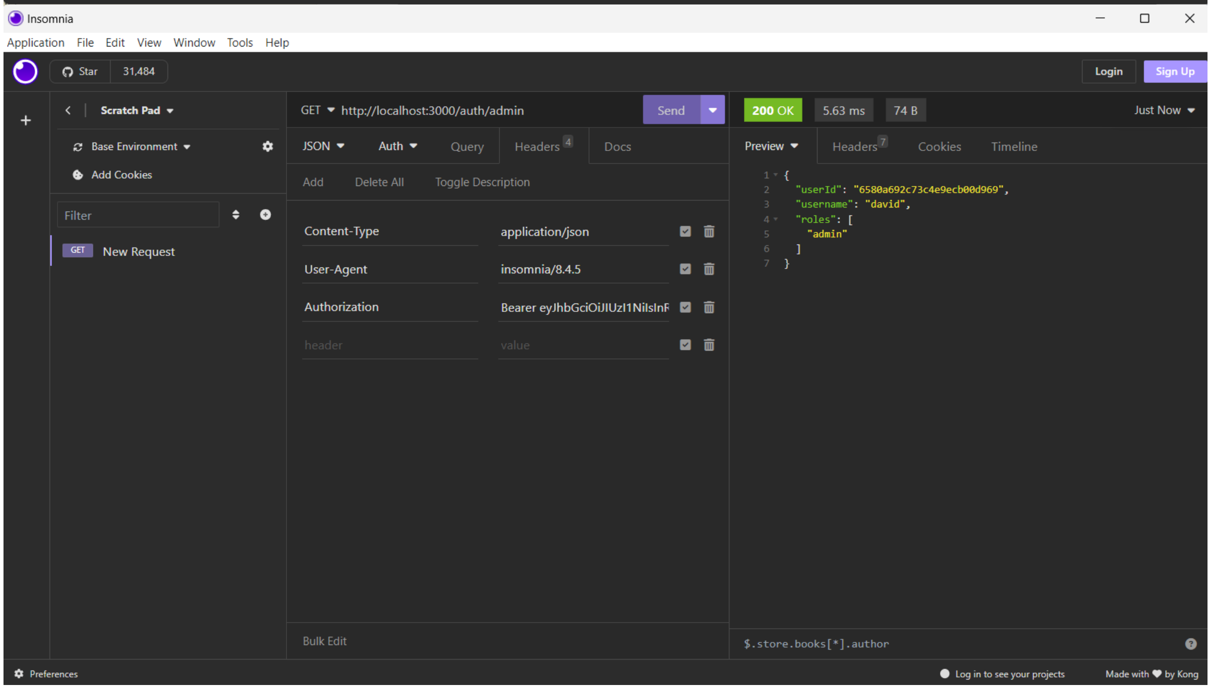
```
{
  "username": "david",
  "email": "david@email.com",
  "password":
"$2b$10$pJiUXTF6TVakbDwxDhujq.7LDETwMLEVrTn.9Tm/uwFflRX4s7fpS",
  "roles": [
    "admin"
  ],
  "_id": "6580a692c73c4e9ecb00d969",
  "__v": 0
}
```


POST http://localhost:3000/auth/login

```
{
  "username": "david",
  "password": "davids_password"
}
```

201 Created 160 ms 240 B

```
{
  "access_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImRhdmlkIiwic3ViIjoiNjU4MGE2OTJjNzNjNGU5ZWNiMDBkOTY5Iiwicm9sZXMiOlsiYWRtaW4iXSwiaWF0IjoxNzAyOTMwMTA2LCJleHAiOjE3MDI5MzM3MDZ9.sMPyCpFLwntuuo3DCgfWi9sWmUiT-Nrxo1LoMPM2Jgw"
}
```


GET http://localhost:3000/auth/admin

Content-Type	application/json
User-Agent	insomnia/8.4.5
Authorization	Bearer eyJhbGciOiJIUzI1NiIsInR

200 OK 5.63 ms 74 B

```
{
  "userId": "6580a692c73c4e9ecb00d969",
  "username": "david",
  "roles": [
    "admin"
  ]
}
```

Creating the store cart feature for our NestJS ecommerce app:

The last feature we'll add to our project is a basic cart functionality.

Creating our store cart resources

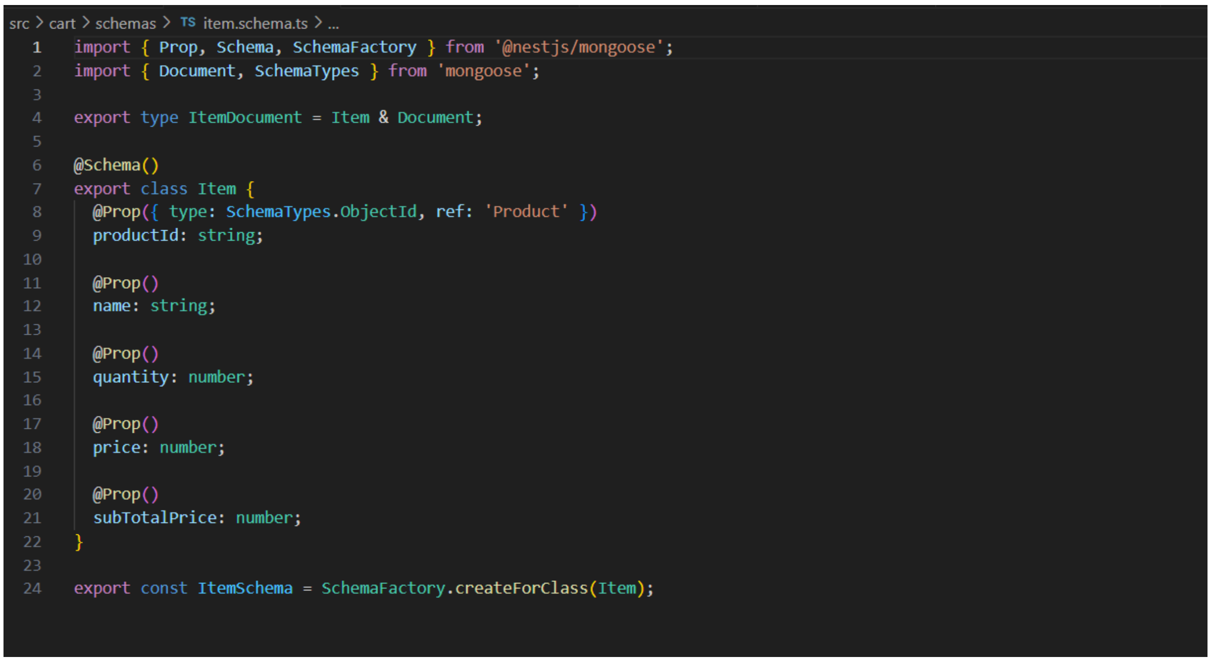
Let's create the resources we need for this next section:

```
nest g module cart
nest g service cart --no-spec
nest g controller cart --no-spec
```

Creating the schemas and DTOs:

For the store cart feature, we'll need two schemas: one describing the products in the cart, and one describing the cart itself.

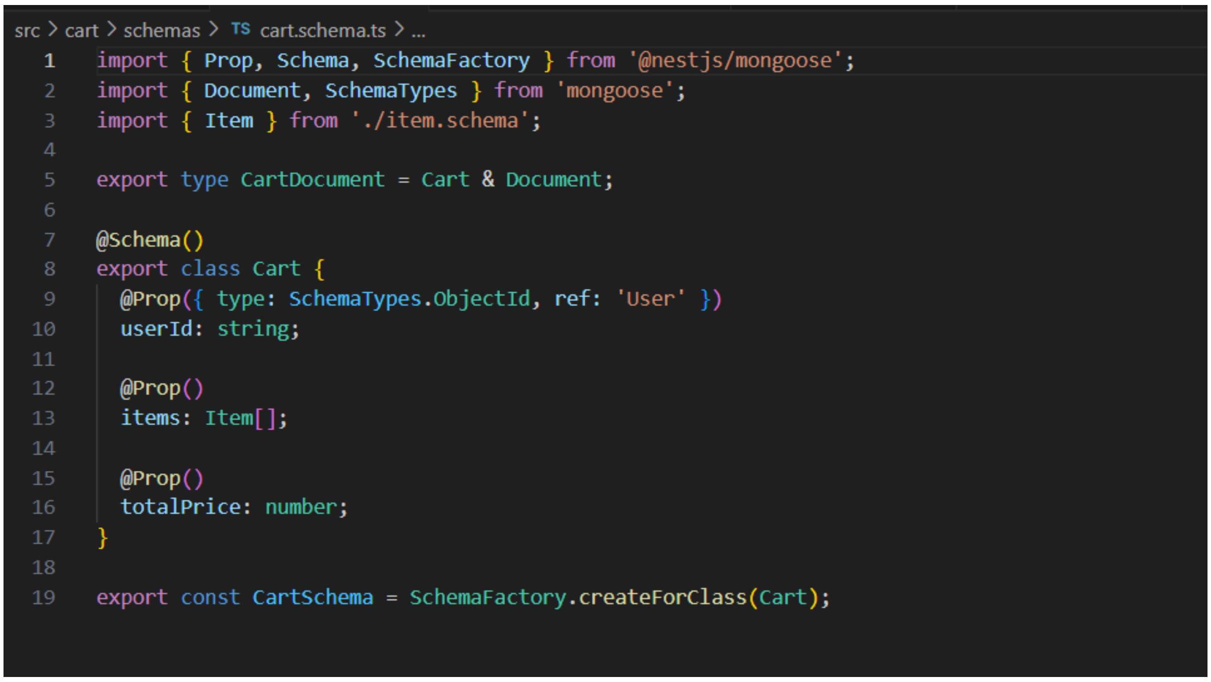
As usual, in the cart directory, create a new schemas folder. Add a **item.schema.ts** file in this new folder.



```
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import { Document, SchemaTypes } from 'mongoose';

export type ItemDocument = Item & Document;

@Schema()
export class Item {
  @Prop({ type: SchemaTypes.ObjectId, ref: 'Product' })
  productId: string;

  @Prop()
  name: string;

  @Prop()
  quantity: number;

  @Prop()
  price: number;

  @Prop()
  subTotalPrice: number;
}

export const ItemSchema = SchemaFactory.createForClass(Item);
```

In the code above, in the @Prop decorator for the **productId** property, we defined an object id schema type and added a reference to the product. This means that we will use the id of the product for the productId value.

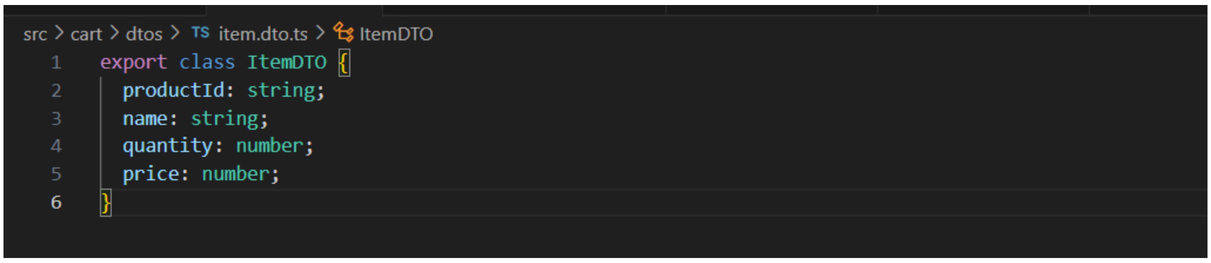
The next schema is for the cart. In the schemas directory, create a **cart.schema.ts** file with the following content:


src > cart > schemas > TS cart.schema.ts > ...

```ts
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import { Document, SchemaTypes } from 'mongoose';
import { Item } from './item.schema';

export type CartDocument = Cart & Document;

@Schema()
export class Cart {
  @Prop({ type: SchemaTypes.ObjectId, ref: 'User' })
  userId: string;

  @Prop()
  items: Item[];

  @Prop()
  totalPrice: number;
}

export const CartSchema = SchemaFactory.createForClass(Cart);
```

Here, we use the same technique for the userId property which will get as a value the user's id. For the items property we use the our Item schema to define an array of items with type of Item.

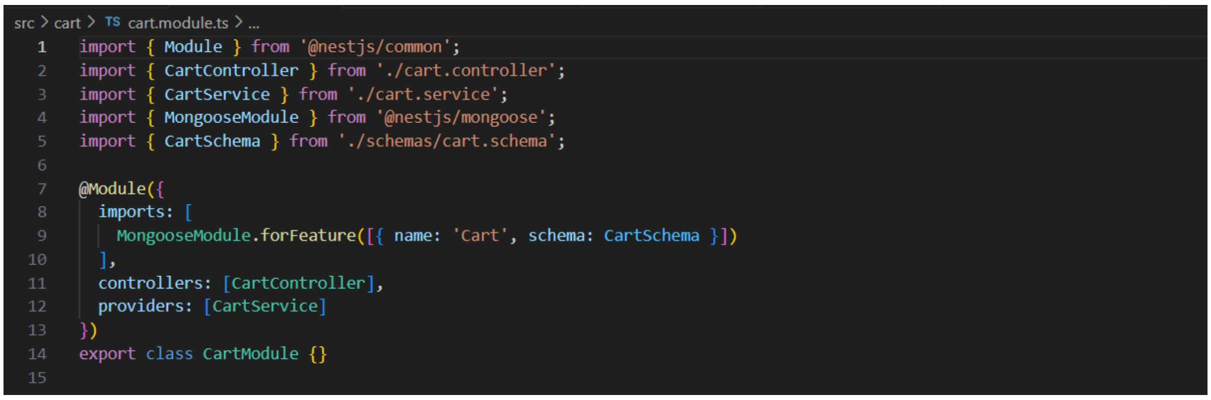
And lastly, let's create the item DTO. In the user directory, create a new dtos folder and add an **item.dto.ts** file with the following content:


src > cart > dtos > TS item.dto.ts > ItemDTO

```ts
export class ItemDTO {
  productId: string;
  name: string;
  quantity: number;
  price: number;
}
```

Configuring the cart module:

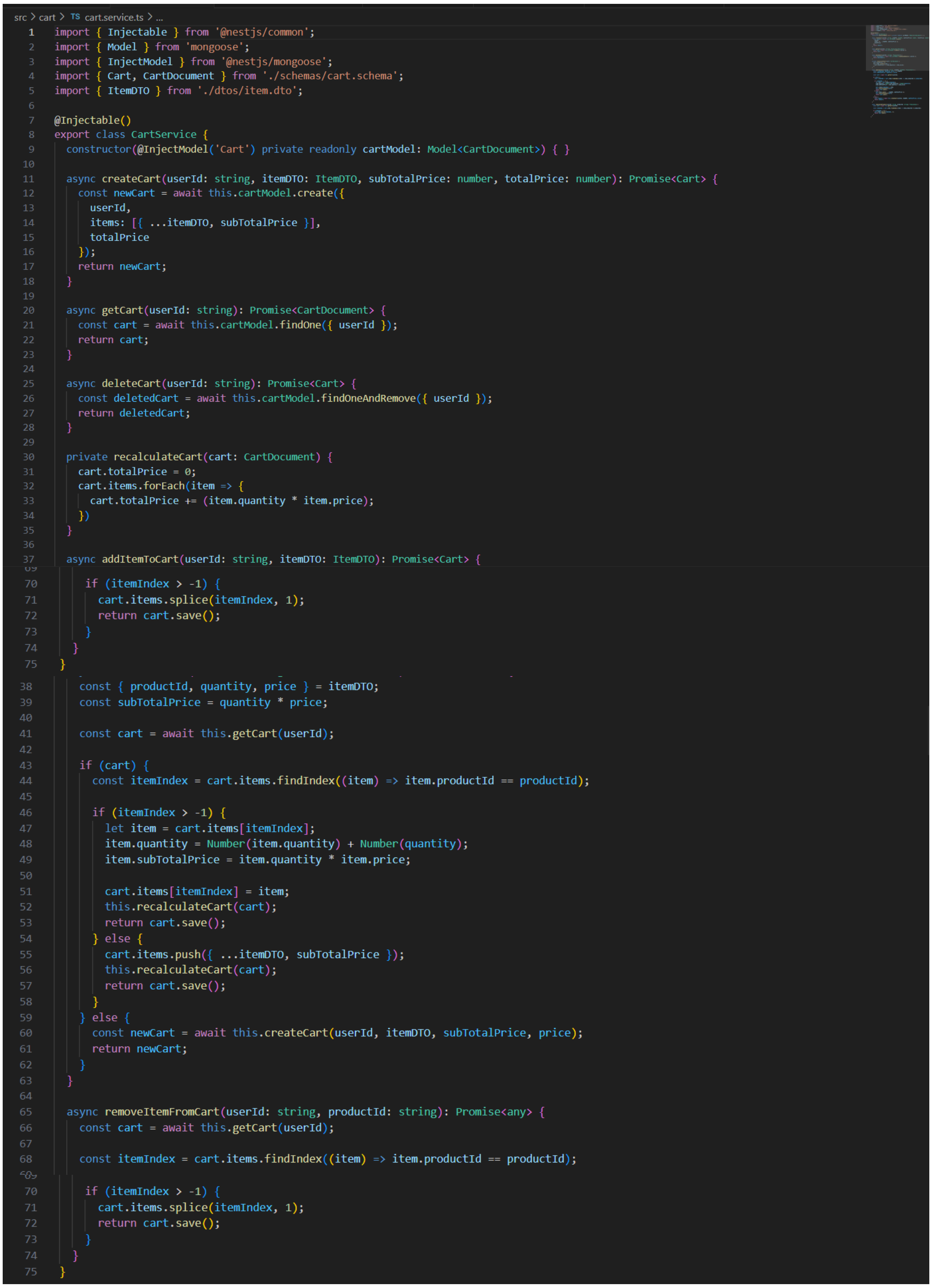
Before we move to the business logic, we need to add the cart schema to the cart module. Open the **cart.module.ts** file and configure it to use the cart schema as follows:


src > cart > TS cart.module.ts > ...

```
import { Module } from '@nestjs/common';
import { CartController } from './cart.controller';
import { CartService } from './cart.service';
import { MongooseModule } from '@nestjs/mongoose';
import { CartSchema } from './schemas/cart.schema';

@Module({
  imports: [
    MongooseModule.forFeature([{ name: 'Cart', schema: CartSchema }])
  ],
  controllers: [CartController],
  providers: [CartService]
})
export class CartModule {}
```

Creating cart service methods:

Now let’s create the cart management logic. Open the **cart.service.ts** file and replace its content with the following:


src > cart > TS cart.service.ts > ...

```typescript
import { Injectable } from '@nestjs/common';
import { Model } from 'mongoose';
import { InjectModel } from '@nestjs/mongoose';
import { Cart, CartDocument } from './schemas/cart.schema';
import { ItemDTO } from './dtos/item.dto';

@Injectable()
export class CartService {
  constructor(@InjectModel('Cart') private readonly cartModel: Model<CartDocument>) { }

  async createCart(userId: string, itemDTO: ItemDTO, subTotalPrice: number, totalPrice: number): Promise<Cart> {
    const newCart = await this.cartModel.create({
      userId,
      items: [{ ...itemDTO, subTotalPrice }],
      totalPrice
    });
    return newCart;
  }

  async getCart(userId: string): Promise<CartDocument> {
    const cart = await this.cartModel.findOne({ userId });
    return cart;
  }

  async deleteCart(userId: string): Promise<Cart> {
    const deletedCart = await this.cartModel.findOneAndRemove({ userId });
    return deletedCart;
  }

  private recalculateCart(cart: CartDocument) {
    cart.totalPrice = 0;
    cart.items.forEach(item => {
      cart.totalPrice += (item.quantity * item.price);
    })
  }

  async addItemToCart(userId: string, itemDTO: ItemDTO): Promise<Cart> {
```

```typescript
    if (itemIndex > -1) {
      cart.items.splice(itemIndex, 1);
      return cart.save();
    }
  }
}
```

```typescript
    const { productId, quantity, price } = itemDTO;
    const subTotalPrice = quantity * price;

    const cart = await this.getCart(userId);

    if (cart) {
      const itemIndex = cart.items.findIndex((item) => item.productId == productId);

      if (itemIndex > -1) {
        let item = cart.items[itemIndex];
        item.quantity = Number(item.quantity) + Number(quantity);
        item.subTotalPrice = item.quantity * item.price;

        cart.items[itemIndex] = item;
        this.recalculateCart(cart);
        return cart.save();
      } else {
        cart.items.push({ ...itemDTO, subTotalPrice });
        this.recalculateCart(cart);
        return cart.save();
      }
    } else {
      const newCart = await this.createCart(userId, itemDTO, subTotalPrice, price);
      return newCart;
    }
  }

  async removeItemFromCart(userId: string, productId: string): Promise<any> {
    const cart = await this.getCart(userId);

    const itemIndex = cart.items.findIndex((item) => item.productId == productId);

    if (itemIndex > -1) {
      cart.items.splice(itemIndex, 1);
      return cart.save();
    }
  }
}
```

There are many methods here. Let’s examine them one by one.
The first one is for creating a new cart for the current user:
The next two methods are for getting or deleting a particular user’s cart

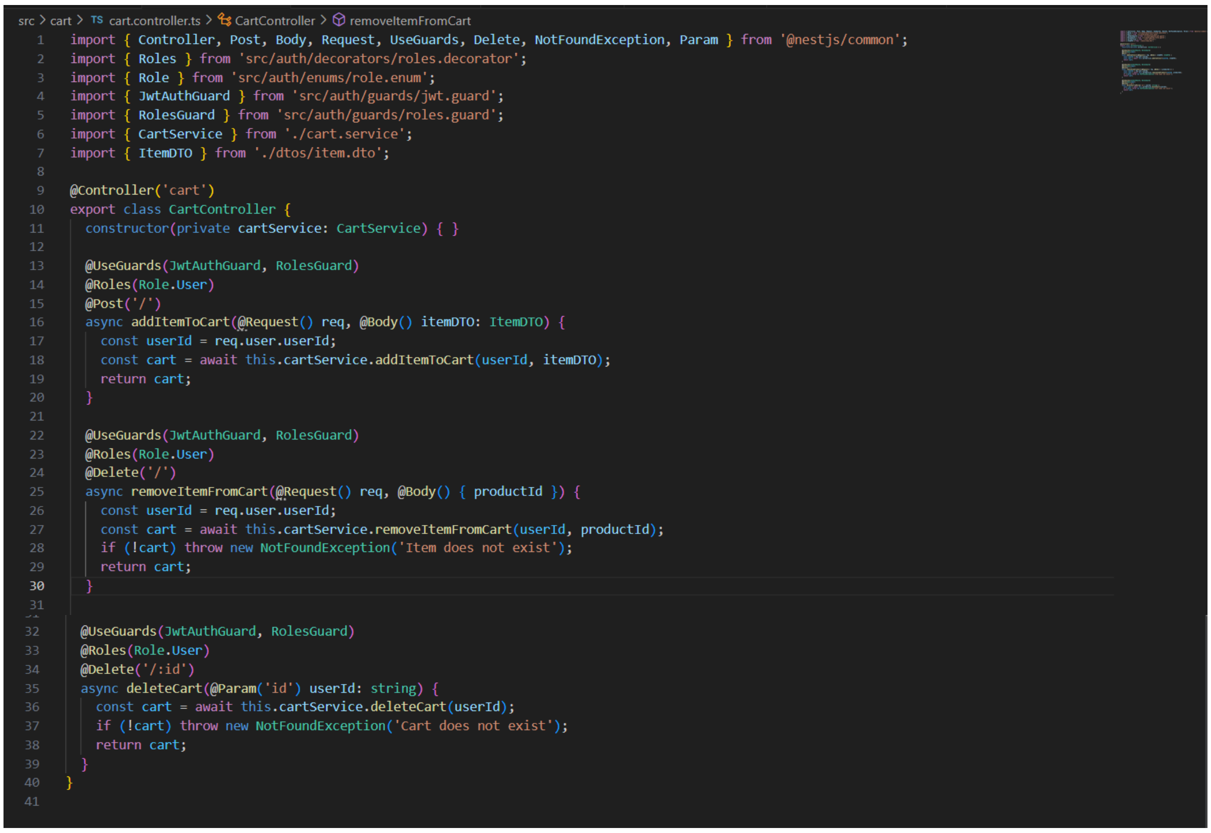
The next method is for recalculating the cart total when an item is added or removed, or when an item’s quantity is changed
The next method is for adding items to the cart
In the method above, if the cart exists, there are two options:

The product exists, so we need to update its quantity and subtotal price
The product doesn’t exist, so we need to add it
Either way, we need to run the recalculateCart() method to update the cart appropriately. If the cart doesn’t exist, we need to create a new one.

The last method is for removing an item from the cart.
Similarly to the previous method, in the method above, we run **recalculateCart()** to update the cart correctly after an item is removed.

Creating cart controller methods:

Our final step to finish this NestJS ecommerce app project is to add the cart controller methods.


src > cart > TS cart.controller.ts > CartController > removeItemFromCart

```
import { Controller, Post, Body, Request, UseGuards, Delete, NotFoundException, Param } from '@nestjs/common';
import { Roles } from 'src/auth/decorators/roles.decorator';
import { Role } from 'src/auth/enums/role.enum';
import { JwtAuthGuard } from 'src/auth/guards/jwt.guard';
import { RolesGuard } from 'src/auth/guards/roles.guard';
import { CartService } from './cart.service';
import { ItemDTO } from './dtos/item.dto';

@Controller('cart')
export class CartController {
  constructor(private cartService: CartService) { }

  @UseGuards(JwtAuthGuard, RolesGuard)
  @Roles(Role.User)
  @Post('/')
  async addItemToCart(@Request() req, @Body() itemDTO: ItemDTO) {
    const userId = req.user.userId;
    const cart = await this.cartService.addItemToCart(userId, itemDTO);
    return cart;
  }

  @UseGuards(JwtAuthGuard, RolesGuard)
  @Roles(Role.User)
  @Delete('/')
  async removeItemFromCart(@Request() req, @Body() { productId }) {
    const userId = req.user.userId;
    const cart = await this.cartService.removeItemFromCart(userId, productId);
    if (!cart) throw new NotFoundException('Item does not exist');
    return cart;
  }

  @UseGuards(JwtAuthGuard, RolesGuard)
  @Roles(Role.User)
  @Delete('/:id')
  async deleteCart(@Param('id') userId: string) {
    const cart = await this.cartService.deleteCart(userId);
    if (!cart) throw new NotFoundException('Cart does not exist');
    return cart;
  }
}
```

In the code above, we used **@UseGuards** and **@Roles** decorators for the three methods. This instructs the app that a customer must be logged in and must have a user role assigned to add or remove products.

Results:

Add Items to Cart:

URL: http://localhost:3000/cart/
Method: POST
Description: Adds items to the user's cart.
Authentication: Bearer Token (Include the token obtained during login)
Request Body:

```
{
     "Product ID" : "P001",
     "Name" : "Laptop",
     "Quantity" : "10",
     "Price": "1200.00"
}
```

Response: Returns the updated cart.

Remove Item from Cart:
Endpoint: DELETE http://localhost:3000/cart/
Description: Removes an item from the user's cart.
Authorization Header: Bearer Token (Include the token obtained during login)
Request Body:

```
{
  "productId": "product_id"
}
```

Response: Returns the updated cart.

Delete Cart:
Endpoint: DELETE http://localhost:3000/cart/{userId}
Description: Deletes the entire cart of a user.
Authorization Header: Bearer Token (Include the token obtained during login)
Response: Returns a success message.

---------THANK YOU--------