

## Weekly Assignment-02

---

### [GITHUB LINK](#)

#### 1. Explain the role of `@Controller()` in NestJS. How does it differ from Express.js routes?

The `@Controller()` decorator in NestJS defines a class as a controller that handles incoming requests. It is responsible for processing requests and returning responses. Unlike Express.js routes, which are defined directly in the routing logic, NestJS controllers encapsulate route handling within classes, promoting better organization and separation of concerns. This structure allows for easier testing and maintenance of the application.

#### 2. What is the purpose of the `@Injectable()` decorator in NestJS?

The `@Injectable()` decorator marks a class as a provider that can be injected as a dependency. This is essential for implementing Dependency Injection (DI) in NestJS, allowing services to be reused across different parts of the application. By using `@Injectable()`, developers can manage the lifecycle of services and ensure that they are instantiated only when needed, improving performance and modularity.

#### 3. Describe how Dependency Injection (DI) works in NestJS with an example.

Dependency Injection (DI) in NestJS allows classes to receive their dependencies from an external source rather than creating them internally. For example, if a service class requires a repository, it can be injected via the constructor. This promotes loose coupling and makes testing easier, as dependencies can be mocked or replaced without modifying the class itself.

#### 4. Why does NestJS use Modules (`@Module()`)? How do they help in structuring applications?

NestJS uses modules to organize the application into cohesive blocks of functionality. Each module can encapsulate related components, services, and controllers, making the application more maintainable and scalable. The `@Module()` decorator defines a module, allowing developers to import and export components, which facilitates better separation of concerns and modular design.

#### 5. What are the differences between providers, controllers, and imports in a NestJS module?

In a NestJS module, providers are classes that can be injected as dependencies, typically services. Controllers handle incoming requests and define routes, while imports are other modules that a module depends on. This structure allows for clear organization of application logic, where providers encapsulate business logic, controllers manage request handling, and imports facilitate modularity.

#### 6. How do you extract route parameters (`/users/:id`) in a NestJS controller?

To extract route parameters in a NestJS controller, you can use the `@Param()` decorator. This decorator allows you to access the parameters defined in the route path. For example, in a method handling the route `/users/:id`, you can define a parameter in the method signature like this: `@Get('/:id') getUser(@Param('id') id: string)`, which will give you the value of the 'id' parameter.

## **7. What is the difference between @Body(), @Param(), and @Query() decorators?**

@Body() is used to extract the body of the request, typically for POST requests containing JSON data. @Param() is used to access route parameters defined in the URL path, while @Query() extracts query parameters from the URL. Each decorator serves a specific purpose in handling different parts of the request, allowing for flexible and organized request handling.

## **8. How would you handle a POST request with JSON payload validation in NestJS?**

To handle a POST request with JSON payload validation in NestJS, you can use the class-validator library along with DTOs (Data Transfer Objects). By defining a DTO with validation decorators, you can ensure that the incoming request body meets the required criteria. NestJS will automatically validate the payload and return errors if the validation fails, streamlining the request handling process.

## **9. Explain the difference between PATCH and PUT in RESTful NestJS APIs.**

PATCH and PUT are both HTTP methods used to update resources, but they differ in their approach. PUT is used to replace an entire resource, while PATCH is used to apply partial modifications. In NestJS, you would typically use PUT for complete updates and PATCH for updates that only change specific fields, allowing for more efficient data handling.

## **10. How can you restrict a route to accept application/json requests only using middleware?**

To restrict a route to accept only application/json requests in NestJS, you can create a middleware that checks the Content-Type header of incoming requests. If the header does not match 'application/json', the middleware can throw an error or return a response indicating that the request is not acceptable. This ensures that only valid requests are processed by the route.

## **11. What is middleware in NestJS? How is it different from Express.js middleware?**

Middleware in NestJS is a function that is executed during the request-response cycle, allowing for pre-processing of requests. While it serves a similar purpose to Express.js middleware, NestJS middleware is integrated into its modular architecture, allowing for better organization and dependency injection. This enables more structured and maintainable middleware implementations.

## **12. How would you create a global middleware vs. a route-specific middleware?**

To create a global middleware in NestJS, you can use the `app.use()` method in the main application file, applying it to all routes. For route-specific middleware, you can apply it directly to a controller or a specific route handler using the `@UseGuards()` or `@UseInterceptors()` decorators. This allows for flexibility in applying middleware based on the application's needs.

## **13. What is the execution order of middleware and pipes in NestJS?**

In NestJS, middleware is executed before the route handler, allowing for pre-processing of requests. After middleware, pipes are executed to validate and transform the request data before it reaches the route handler. This order ensures that requests are properly validated and processed before any business logic is applied.

## **14. Why should business logic be placed in a Service rather than a Controller?**

Business logic should be placed in a Service to promote separation of concerns and maintainability. Controllers should focus on handling requests and responses, while Services encapsulate the core application logic. This structure allows for easier testing, reusability, and adherence to the Single Responsibility Principle.

### 15. How do you share a service between multiple modules in NestJS?

To share a service between multiple modules in NestJS, you can export the service from its defining module and import it into other modules that require it. This is done by adding the service to the `providers` array in the module where it is defined and including it in the `exports` array. Other modules can then import the module to gain access to the shared service.

### 16. What is a singleton scope in NestJS providers? Are services singleton by default?

In NestJS, a singleton scope means that a provider is instantiated only once and shared across the entire application. By default, services in NestJS are singleton, meaning that the same instance is used whenever the service is injected. This behavior promotes efficient resource usage and consistent state management across the application.

### 17. How would you implement role-based access control (RBAC) in NestJS without a database using a custom decorator?

To implement role-based access control (RBAC) in NestJS without a database, you can create a custom decorator that checks the user's role against predefined roles. The decorator can be applied to route handlers, and it can use a simple in-memory structure to store user roles. If the user does not have the required role, the decorator can throw an exception, preventing access to the route.

### 18. Explain how Exception Filters work in NestJS. How are they different from Express error handlers?

Exception Filters in NestJS are used to handle errors thrown during the request-response cycle. They allow for centralized error handling and can format error responses consistently. Unlike Express error handlers, which are typically defined as middleware, NestJS Exception Filters are more structured and can be applied at various levels (global, controller, or route-specific), providing greater flexibility in error management.

## Practical Tasks:

### [GITHUB LINK](#)

#### 1. Dynamic Route Handling

Create a route `GET /users/:id/role/:role` where:

- If `role=admin`, return mock admin data.
- If `role=user`, return filtered user data.

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

- PS C:\Users\Jafar\Desktop\nodejs\js files\weekly\_assignment\_2> nest g service user  
CREATE src/user/user.service.ts (92 bytes)  
CREATE src/user/user.service.spec.ts (464 bytes)  
UPDATE src/app.module.ts (323 bytes)
- PS C:\Users\Jafar\Desktop\nodejs\js files\weekly\_assignment\_2> █

```
1 import { Injectable } from '@nestjs/common';
2
3 @Injectable()
4 export class UserService {
5
6   showUserDetails(id:number,role:string){
7     const users=[ {
8       "id": "1",
9       "email": "user1@example.com",
10      "name": "UserOne",
11      "password": "User12345",
12      "description": "This is mock user data for UserOne."
13    },
14    {
15      "id": "2",
16      "email": "user2@example.com",
17      "name": "UserTwo",
18      "password": "User67890",
19      "description": "This is mock user data for UserTwo."
20    },
21    {
22      "id": "3",
23      "email": "user3@example.com",
24      "name": "UserThree",
25      "password": "User11111",
26      "description": "This is mock user data for UserThree."
27    },
28    {
29      "id": "4",
30      "email": "user4@example.com",
31      "name": "UserFour",
32      "password": "User22222",
33      "description": "This is mock user data for UserFour."
34    }
35  ]
36   if(role==="admin"){
37     return {id,email:"adminuser@gmail.com",name:"AdminUser",password:"Admin123",description:"This is mock admin data."}
38   }
39   if(role==="user"){
40     return users.find((user)=>Number(user.id)==id)
41   }
42   return {id,role}
43 }
44
45 }
46
```

```

1  @Get("/users/:id/role/:role")
2  getUser(@Param('id')id:number,@Param('role')role:string){
3  return this.userService.showUserDetails(Number(id),role)
4  }
5  }

```

GET

Query Headers<sup>4</sup> Auth **Body<sup>1</sup>** Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content

Status: 200 OK Size: 130 Bytes Time: 5 ms

Response Headers<sup>6</sup> Cookies Results Docs

```

1  {
2    "id": "2",
3    "email": "user2@example.com",
4    "name": "UserTwo",
5    "password": "User67890",
6    "description": "This is mock user data for UserTwo."
7  }

```

GET

Query Headers<sup>4</sup> Auth **Body** Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content

Status: 200 OK Size: 120 Bytes Time: 15 ms

Response Headers<sup>6</sup> Cookies Results Docs

```

1  {
2    "id": 2,
3    "email": "adminuser@gmail.com",
4    "name": "AdminUser",
5    "password": "Admin123",
6    "description": "This is mock admin data."
7  }

```

## 2. Custom Decorator

Create a `@Timeout(delay: number)` decorator that cancels the request if it takes longer than `delay` ms.

- Use `setTimeout` + throw new `RequestTimeoutException()`.

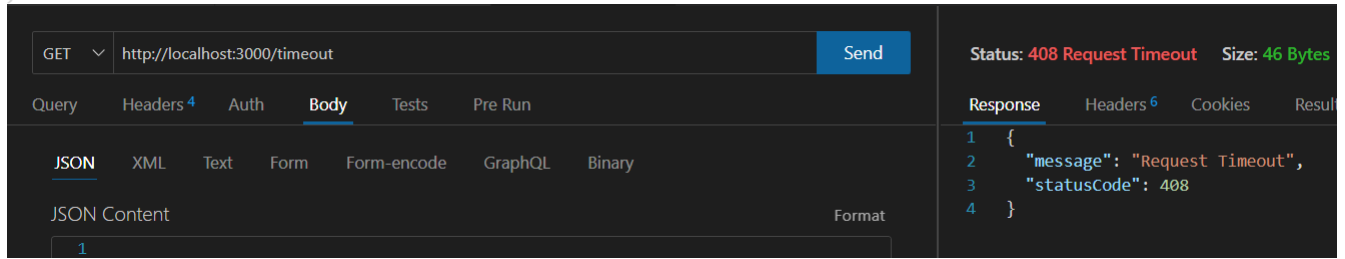
```

PS C:\Users\Jafar\Desktop\Nodejs\js_files\weekly_assignment_2> nest g decorator decorators/timeoutdelay
CREATE src/decorators/timeoutdelay/timeoutdelay.decorator.ts (136 bytes)
PS C:\Users\Jafar\Desktop\Nodejs\js_files\weekly_assignment_2>

```

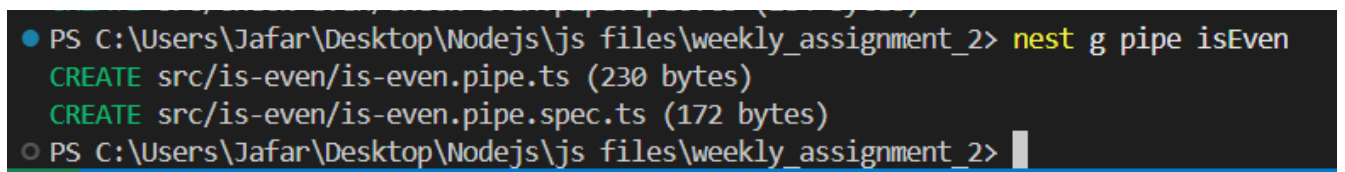
```
1 // timeout.decorator.ts
2 import { RequestTimeoutException } from '@nestjs/common';
3
4 export function TimeoutDelay(delay: number) {
5   return function (
6     _target: any,
7     _propertyKey: string,
8     descriptor: PropertyDescriptor,
9   ) {
10     const originalMethod = descriptor.value;
11
12     descriptor.value = async function (...args: any[]) {
13       return await Promise.race([
14         originalMethod.apply(this, args),
15         new Promise( (_, reject) =>
16           setTimeout(() => reject(new RequestTimeoutException()), delay),
17         ),
18       ]);
19     };
20   };
21 }
22
```

```
1
2 @Get("/timeout")
3 @TimeoutDelay(500)
4 async getData() {
5   await new Promise((res) => setTimeout(res, 5000));
6   return { message: 'done' };
7 }
8
9
```



### 3. Pipe for Custom Validation

Create a pipe that rejects numbers if not even (for routes like `GET /check-even/:num`).



GET http://localhost:3000/check-even/523 Send

Status: 400 Bad Request Size: 71 Bytes Time: 4 ms

Query Headers 4 Auth Body Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

1

Response Headers 6 Cookies Results Docs

```

1 {
2   "message": "Not an Even Number",
3   "error": "Bad Request",
4   "statusCode": 400
5 }

```

#### 4. Middleware for Rate Limiting

Implement middleware to allow only 3 requests/minute per IP (use Map to store IP counts).

```

• PS C:\Users\Jafar\Desktop\Nodejs\js files\weekly_assignment_2> nest g middleware rate-limit
CREATE src/rate-limit/rate-limit.middleware.ts (209 bytes)
CREATE src/rate-limit/rate-limit.middleware.spec.ts (208 bytes)

```

```

1 import { HttpException, Injectable, NestMiddleware } from '@nestjs/common';
2 import { Request, Response, NextFunction } from 'express';
3
4 @Injectable()
5 export class RateLimitMiddleware implements NestMiddleware {
6   private requestsMap = new Map<string, { count: number; timestamp: number }>();
7   private limit = 3;
8   private duration = 60 * 1000; // 1 minute
9
10  use(req: Request, res: Response, next: NextFunction) {
11    const ip = req.ip || '';
12    const currentTime = Date.now();
13
14    const requestData = this.requestsMap.get(ip);
15
16    if (!requestData || currentTime - requestData.timestamp > this.duration) {
17      this.requestsMap.set(ip, { count: 1, timestamp: currentTime });
18      return next();
19    }
20
21    if (requestData.count < this.limit) {
22      requestData.count += 1;
23      this.requestsMap.set(ip, requestData);
24      return next();
25    }
26    throw new HttpException('Rate limit exceeded. Try again later.', 429);
27  }
28 }
29

```



```

export class AppModule {
  configure(consumer: MiddlewareConsumer) {
    consumer.apply(RateLimitMiddleware).forRoutes('*');
  }
}

```

GET	http://localhost:3000/check-even/523	Send	Status: 429 Too Many Requests	Size: 68 Bytes	Time: 4 ms
Query	Headers 4	Auth	Body	Tests	Pre Run
JSON	XML	Text	Form	Form-encode	GraphQL Binary
JSON Content			Format		
			Response	Headers 6	Cookies Results Docs
			<pre> 1 { 2   "statusCode": 429, 3   "message": "Rate limit exceeded. Try again later." 4 } </pre>		

## 5. Mock E-Commerce Checkout Flow

- Simulate an entire checkout process (cart → payment → order confirmation)
- Use in-memory arrays for:
  - Cart items (/cart routes)
  - Mock payment processing (/payment route)
  - Order history (/orders route)
  - Throw custom errors (e.g., `InsufficientStockException`)

```

PS C:\Users\Jafar\Desktop\Nodejs\js files\weekly_assignment_2> nest g resource cart --no-spec
✓ What transport layer do you use? REST API
✓ Would you like to generate CRUD entry points? Yes
CREATE src/cart/cart.controller.ts (917 bytes)
CREATE src/cart/cart.module.ts (250 bytes)
CREATE src/cart/cart.service.ts (633 bytes)
CREATE src/cart/dto/create-cart.dto.ts (31 bytes)
CREATE src/cart/dto/update-cart.dto.ts (173 bytes)
CREATE src/cart/entities/cart.entity.ts (22 bytes)
UPDATE package.json (2152 bytes)
UPDATE src/app.module.ts (589 bytes)
✓ Packages installed successfully.
PS C:\Users\Jafar\Desktop\Nodejs\js files\weekly_assignment_2>

```

All carts:

GET

http://localhost:3000/cart/

Send

Query

Headers 4

Auth

Body 1

Tests

Pre Run

JSON

XML

Text

Form

Form-encode

GraphQL

Binary

JSON Content

Format

1

Status: 200 OK

Size: 365 Bytes

Time: 6 ms

Response

Headers 6

Cookies

Results

Docs

1

[

2

{

3

"productId": 1,

4

"name": "Laptop",

5

"quantity": 1,

6

"price": 499

7

},

8

{

9

"productId": 2,

10

"name": "Washing Machine",

11

"quantity": 2,

12

"price": 999

13

},

14

{

15

"productId": 3,

16

"name": "Smartphone",

17

"quantity": 3,

18

"price": 299

19

},

20

{

21

"productId": 4,

22

"name": "Headphones",

23

"quantity": 2,

24

"price": 99

25

},

26

{

27

"productId": 4,

Cart created:

POST

http://localhost:3000/cart/

Send

Query

Headers 4

Auth

Body 1

Tests

Pre Run

JSON

XML

Text

Form

Form-encode

GraphQL

Binary

JSON Content

Format

1

{

2

"productId": 4,

3

"name": "Headphones",

4

"quantity": 2,

5

"price": 99

6

}

Status: 201 Created

Size: 59 Bytes

Time: 6 ms

Response

Headers 6

Cookies

Results

Docs

1

{

2

"productId": 4,

3

"name": "Headphones",

4

"quantity": 2,

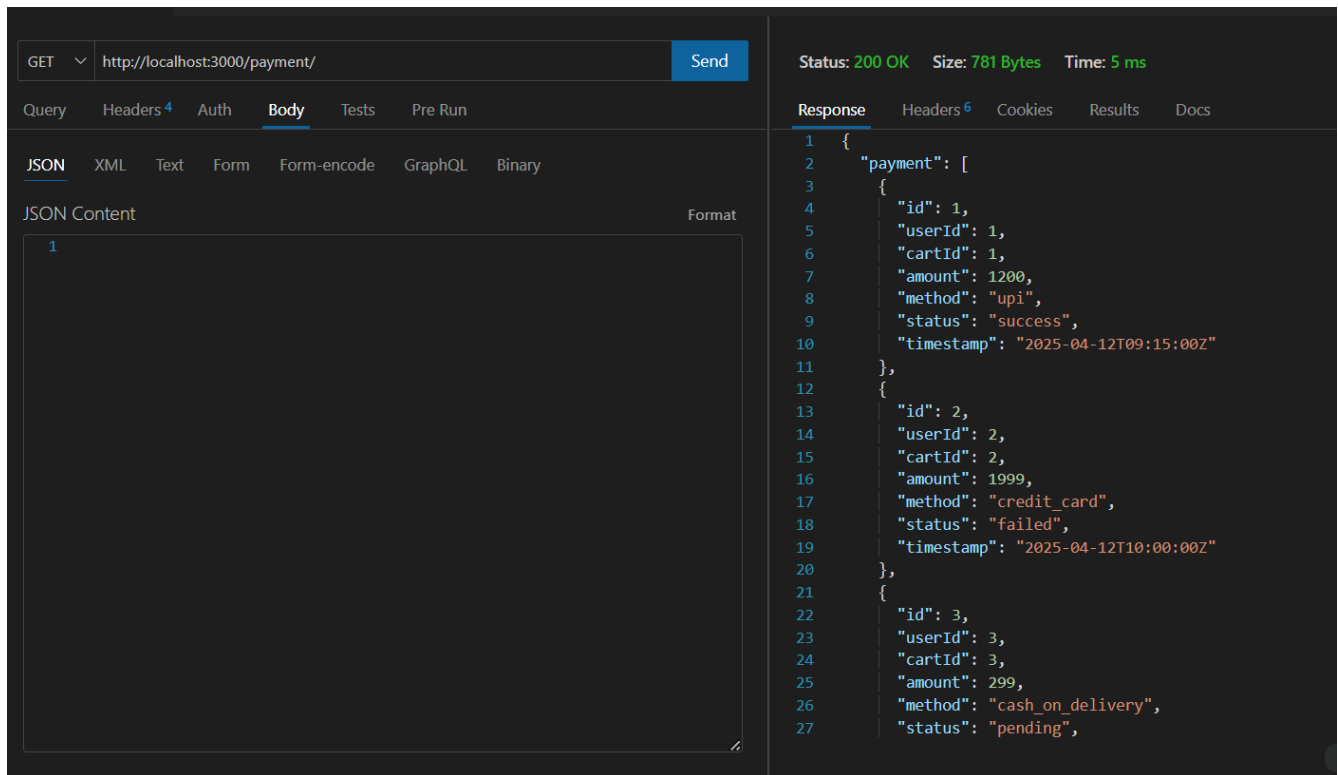
5

"price": 99

6

}

## All Payments :



GET http://localhost:3000/payment/ Send

Status: 200 OK Size: 781 Bytes Time: 5 ms

Query Headers<sup>4</sup> Auth **Body** Tests Pre Run

**JSON** XML Text Form Form-encode GraphQL Binary

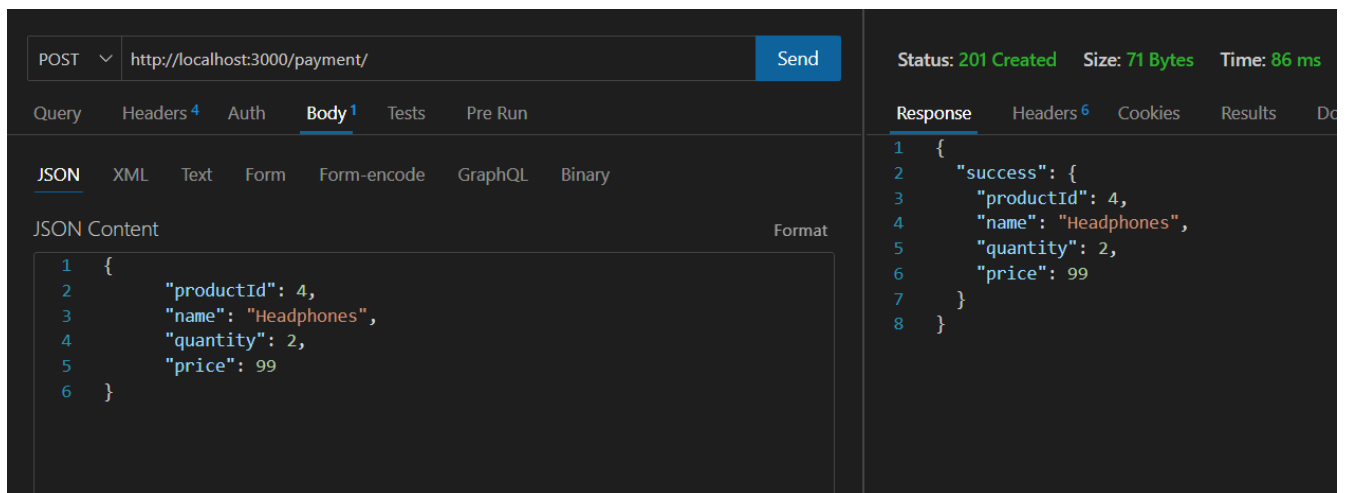
JSON Content Format

```
1
```

**Response** Headers<sup>6</sup> Cookies Results Docs

```
1 {
2   "payment": [
3     {
4       "id": 1,
5       "userId": 1,
6       "cartId": 1,
7       "amount": 1200,
8       "method": "upi",
9       "status": "success",
10      "timestamp": "2025-04-12T09:15:00Z"
11    },
12    {
13      "id": 2,
14      "userId": 2,
15      "cartId": 2,
16      "amount": 1999,
17      "method": "credit_card",
18      "status": "failed",
19      "timestamp": "2025-04-12T10:00:00Z"
20    },
21    {
22      "id": 3,
23      "userId": 3,
24      "cartId": 3,
25      "amount": 299,
26      "method": "cash_on_delivery",
27      "status": "pending",
```

## Payment added:



POST http://localhost:3000/payment/ Send

Status: 201 Created Size: 71 Bytes Time: 86 ms

Query Headers<sup>4</sup> Auth **Body**<sup>1</sup> Tests Pre Run

**JSON** XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {
2   "productId": 4,
3   "name": "Headphones",
4   "quantity": 2,
5   "price": 99
6 }
```

**Response** Headers<sup>6</sup> Cookies Results Docs

```
1 {
2   "success": {
3     "productId": 4,
4     "name": "Headphones",
5     "quantity": 2,
6     "price": 99
7   }
8 }
```

```
30 @Post('/make-payment/:id')
31 confirmPayment(@Param('id') id: string) {
32     const numericId = +id;
33     const removedCart = this.cartService.remove(numericId);
34     const paymentRecord = this.paymentService.makePayment(numericId);
35     const orderRecord = this.ordersService.confirmOrder(numericId);
36     return {
37         message: 'Payment and order confirmed successfully',
38         removedCart,
39         paymentRecord,
40         orderRecord,
41     };
42 }
```

POST http://localhost:3000/payment/make-payment/1 Send

Status: 201 Created Size: 803 Bytes Time: 7 ms

Query Headers 5 Auth Body Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

1

Response Headers 6 Cookies Results Docs

```
1 {
2   "message": "Payment and order confirmed successfully",
3   "removedCart": {
4     "cartId": 1,
5     "name": "Laptop",
6     "quantity": 1,
7     "price": 499
8   },
9   "paymentRecord": {
10    "success": [
11      {
12        "id": 1,
13        "userId": 1,
14        "cartId": 1,
15        "amount": 1200,
16        "method": "upi",
17        "status": "success",
18        "timestamp": "2025-04-12T09:15:00Z"
19      },
20      {
21        "id": 2,
22        "userId": 2,
23        "cartId": 2,
24        "amount": 1999,
25        "method": "credit_card",
26        "status": "failed",
27        "timestamp": "2025-04-12T10:00:00Z"
28      }
29    ]
30  }
```

Exceptions:

POST

http://localhost:3000/cart/

Send

Query

Headers 4

Auth

Body 1

Tests

Pre Run

JSON

XML

Text

Form

Form-encode

GraphQL

Binary

JSON Content

Format

```
1 {
2   "productId": 4,
3   "name": "Headphones",
4   "quantity": 20,
5   "price": 99
6 }
```

Status: 400 Bad Request

Size: 33 Bytes

Time: 4 ms

Response

Headers 6

Cookies

Results

Docs

```
1 {
2   "message": "Insufficient stock "
3 }
```

POST

http://localhost:3000/payment/make-payment/1

Send

Query

Headers 4

Auth

Body

Tests

Pre Run

JSON

XML

Text

Form

Form-encode

GraphQL

Binary

JSON Content

Format

```
1
```

Status: 404 Not Found

Size: 66 Bytes

Time: 15 ms

Response

Headers 6

Cookies

Results

Docs

```
1 {
2   "message": "Order not found",
3   "error": "Not Found",
4   "statusCode": 404
5 }
```

POST

http://localhost:3000/payment/make-payment/2

Send

Query

Headers 4

Auth

Body

Tests

Pre Run

JSON

XML

Text

Form

Form-encode

GraphQL

Binary

JSON Content

Format

```
1
```

Status: 400 Bad Request

Size: 50 Bytes

Time: 21 ms

Response

Headers 6

Cookies

Results

Docs

```
1 {
2   "message": "Order with ID 2 is already confirmed"
3 }
```

localhost:3000/ X

TS cart.service.ts U

TS orders.service.ts U

TS payment.service.ts U

TS payment.controller.ts U

GET

http://localhost:3000/cart/200

Send

Query

Headers 4

Auth

Body

Tests

Pre Run

JSON

XML

Text

Form

Form-encode

GraphQL

Binary

JSON Content

Format

```
1
```

Status: 404 Not Found

Size: 40 Bytes

Time: 4 ms

Response

Headers 6

Cookies

Results

Docs

```
1 {
2   "message": "Cart with ID 200 not found"
3 }
```

## 6. Validation Assignment

Create a post method for each DTO & throw errors.

DTO Structure: { firstName: string; // 2-50 chars lastName: string; // 2-50 chars email: string; // Valid email format age: number; // 18-65 } Tasks: 1. Implement class-validator decorators 2. Return simple error format: Like { "message": "Validation failed", "errors": ["Invalid email"] }



```
1 import { IsEmail, IsString, Length, IsInt, Min, Max } from 'class-validator';
2
3 export class CreateUserDto {
4   @IsString()
5   @Length(2, 50)
6   firstName: string;
7
8   @IsString()
9   @Length(2, 50)
10  lastName: string;
11
12  @IsEmail({}, { message: 'Invalid email' })
13  email: string;
14
15  @IsInt()
16  @Min(18)
17  @Max(65)
18  age: number;
19 }
20
```

```
1 import { Body, Controller, Post, UsePipes, ValidationPipe, BadRequestException } from '@nestjs/common';
2 import { CreateUserDto } from '../dto/create-user-dto';
3
4
5 @Controller('users')
6 export class UsersController {
7   @Post()
8   @UsePipes(new ValidationPipe({
9     exceptionFactory: (errors) => {
10       const messages = errors.map(err => Object.values(err)).flat();
11       return new BadRequestException({
12         message: 'Validation failed',
13         errors: messages
14       });
15     }
16   )))
17   createUser(@Body() userDto: CreateUserDto) {
18     return { message: 'User created', data: userDto };
19   }
20 }
21
```

POST http://localhost:3000/users/ Send

Status: 400 Bad Request Size: 168 Bytes Time: 6 ms

Query Headers 4 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {
2   "firstName": "John",
3   "lastName": "Doe",
4   "email": "not an email-",
5   "age": 30
6 }
7
```

Response Headers 6 Cookies Results Docs

```
1 {
2   "message": "Validation failed",
3   "errors": [
4     {
5       "firstName": "John",
6       "lastName": "Doe",
7       "email": "not an email-",
8       "age": 30
9     },
10    "not an email-",
11    "email",
12    [],
13    {
14      "isEmail": "Invalid email"
15    }
16  ]
17 }
```

## 7. DTO Structure for User Information

```
{ id: string; // UUID format address: { street: string; // 5-100 chars postalCode: string // Country-specific regex (US/UK/IN) }; education: { degree: string; // ["BSc", "MSc", "PhD"] year: number; // 1990-current year }[]; }
```

Tasks: 1. Add custom validator for postal code

2. Validate array of education objects

3. Return detailed errors:

```
1 import { IsIn, IsString, IsUUID, ValidateNested } from 'class-validator';
2 import { Type } from 'class-transformer';
3 import { AddressDto } from './address.dto';
4 import { EducationDto } from './education.dto';
5
6 export class MainDto {
7   @IsUUID()
8   id: string;
9
10  @ValidateNested()
11  @Type(() => AddressDto)
12  address: AddressDto;
13
14  @ValidateNested({ each: true })
15  @Type(() => EducationDto)
16  education: EducationDto[];
17 }
18
```





```
1 import { IsIn, IsInt, Min, Max } from 'class-validator';
2
3 export class EducationDto {
4   @IsIn(['BSc', 'MSc', 'PhD'])
5   degree: string;
6
7   @IsInt()
8   @Min(1990)
9   @Max(new Date().getFullYear())
10  year: number;
11 }
12
```



```
1 import { IsIn, IsString, Length, Validate } from 'class-validator';
2 import { PostalValidator } from 'src/validators/postal-validator.validator';
3
4
5 export class AddressDto {
6   @IsString()
7   @Length(5, 100)
8   street: string;
9   @IsString()
10  @IsIn(["US", "UK", "IN"], { message: 'Invalid country code' })
11  country: string;
12  @Validate(PostalValidator)
13  postalCode: string;
14 }
15
```

```
1 import { Body, Controller, Post, UsePipes, ValidationPipe, BadRequestException } from '@nestjs/common';
2 import { CreateUserDto } from '../dto/create-user-dto';
3 import { MainDto } from '../dto/address-education.dto';
4
5
6 @Controller('users')
7 export class UsersController {
8   @Post()
9   @UsePipes(new ValidationPipe({
10     exceptionFactory: (errors) => {
11       const messages = errors.map(err => Object.values(err)).flat();
12       return new BadRequestException({
13         message: 'Validation failed',
14         errors: messages
15       });
16     })
17   ))
18   createUser(@Body() userDto: CreateUserDto) {
19     return { message: 'User created', data: userDto };
20   }
21
22   @Post('address-education')
23   @UsePipes(new ValidationPipe())
24   create(@Body() dto: MainDto) { //combination of address and education DTO
25     return { message: 'Data submitted successfully', data: dto };
26   }
27
28
29 }
30
```

TS user.controller.ts U TS address-education.dto.ts U TS address.dto.ts U TS postal-validator.validator.ts U TC localhost:3000/\* X

POST http://localhost:3000/users/address-education Send

Status: 400 Bad Request Size: 82 Bytes Time: 7 ms

Query Headers 4 Auth Body 1 Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {
2   "id": "123e4567-e89b-12d3-a456-426614174000",
3   "country": "US",
4   "address": {
5     "street": "123 Main St",
6     "postalCode": "123456"
7   },
8   "education": [
9     {
10      "degree": "BSc",
11      "year": 2020
12    }
13  ]
14 }
```

Response Headers 6 Cookies Results Docs

```
1 {
2   "message": [
3     "address.INVALID_POSTAL_CODE"
4   ],
5   "error": "Bad Request",
6   "statusCode": 400
7 }
```

The screenshot shows a REST client interface with a POST request to `http://localhost:3000/users/address-education`. The request body is a JSON object containing user information. The response is a 201 status code with a JSON body indicating successful submission.

**Request:**

```

1 {
2   "id": "123e4567-e89b-12d3-a456-426614174000",
3   "country": "US",
4   "address": {
5     "street": "123 Main St",
6     "country": "US",
7     "postalCode": "12345"
8   },
9   "education": [
10    {
11      "degree": "BSc",
12      "year": 2020
13    },
14    {
15      "degree": "MSc",
16      "year": 2023
17    }
18  ]
19 }

```

**Response:**

```

1 {
2   "message": "Data submitted successfully",
3   "data": {
4     "id": "123e4567-e89b-12d3-a456-426614174000",
5     "country": "US",
6     "address": {
7       "street": "123 Main St",
8       "country": "US",
9       "postalCode": "12345"
10    },
11    "education": [
12      {
13        "degree": "BSc",
14        "year": 2020
15      },
16      {
17        "degree": "MSc",
18        "year": 2023
19      }
20    ]
21  }
22 }

```

## 8. DTO Structure for Employment Information

`{ employmentType: "full-time" | "contractor"; fullTimeDetails?: { benefits: string[]; // At least 1 if employmentType=full-time joiningDate: Date; // Must be future date }; contractorDetails?: { contractEnd: Date; // Must be > contractStart hourlyRate: number; // 20-100 for US, 15-80 for EU }; metadata: Record; // Validate keys/values } Tasks: 1. Implement cross-field validation: ○ Require fullTimeDetails if employmentType=full-time 2. Create dynamic rate validation based on locale header (X-Country-Code) 3. Validate metadata: ○ Keys must match /^[a-z0-9_]+$ ○ Values max 255 chars 4. Return hierarchical errors:`

## Invalid Metadata:

The screenshot shows a REST client interface with a POST request to `http://localhost:3000/employment`. The request body is a JSON object. The response is a 400 Bad Request with the following error details:

```
1 {
2   "errors": {
3     "metadata.department-name": {
4       "code": "INVALID_KEY_FORMAT",
5       "message": "Metadata keys must match /^[a-z0-9_]+$/"
6     }
7   }
8 }
```

The JSON Content of the request is as follows:

```
1 {
2   "employmentType": "full-time",
3   "fullTimeDetails": {
4     "benefits": ["health insurance"],
5     "joiningDate": "2025-06-01T00:00:00.000Z"
6   },
7   "metadata": {
8     "department-name": "finance",
9     "level": "mid"
10  }
11 }
```

## Hour Rate too Low for US:

The screenshot shows a REST client interface with a POST request to `http://localhost:3000/employment`. The request body is a JSON object. The response is a 400 Bad Request with the following error details:

```
1 {
2   "errors": {
3     "contractorDetails.hourlyRate": {
4       "code": "RATE_OUT_OF_RANGE",
5       "allowedRanges": {
6         "US": [
7           20,
8           100
9         ]
10      }
11    }
12  }
13 }
```

The JSON Content of the request is as follows:

```
1 {
2   "employmentType": "contractor",
3   "contractorDetails": {
4     "contractEnd": "2025-10-01T00:00:00.000Z",
5     "hourlyRate": 15
6   },
7   "metadata": {
8     "project": "short_term"
9   }
10 }
```

## Contractor Details missing:

POST  http://localhost:3000/employment

Query Headers 5 Auth **Body 1** Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {
2   "employmentType": "contractor",
3   "fullTimeDetails": {
4     "benefits": ["health insurance", "401k", "paid time off"],
5     "joiningDate": "2025-05-15T00:00:00.000Z"
6   },
7   "metadata": {
8     "department": "engineering",
9     "level": "senior",
10    "team_id": "backend_123"
11  }
12 }
```

Status: 400 Bad Request Size: 112 Bytes Time: 5 ms

Response Headers 6 Cookies Results Docs {}

```
1 {
2   "message": [
3     "Contractor details are required for contractor employment"
4   ],
5   "error": "Bad Request",
6   "statusCode": 400
7 }
```

POST  http://localhost:3000/employment

Query Headers 5 Auth **Body 1** Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {
2   "employmentType": "contractor",
3   "contractorDetails": {
4     "contractEnd": "2025-12-31T00:00:00.000Z",
5     "hourlyRate": 85
6   },
7   "metadata": {
8     "project": "api_redesign",
9     "manager": "john_doe"
10  }
11 }
```

Status: 201 Created Size: 237 Bytes Time: 6 ms

Response Headers 6 Cookies Results Docs {}

```
1 {
2   "success": true,
3   "message": "Employment created successfully",
4   "data": {
5     "employmentType": "contractor",
6     "contractorDetails": {
7       "contractEnd": "2025-12-31T00:00:00.000Z",
8       "hourlyRate": 85
9     },
10    "metadata": {
11      "project": "api_redesign",
12      "manager": "john_doe"
13    }
14  }
15 }
```

POST  http://localhost:3000/employment

Query Headers 5 Auth **Body 1** Tests Pre Run

JSON XML Text Form Form-encode GraphQL Binary

JSON Content Format

```
1 {
2   "employmentType": "full-time",
3   "fullTimeDetails": {
4     "benefits": ["health insurance", "401k", "paid time off"],
5     "joiningDate": "2025-05-15T00:00:00.000Z"
6   },
7   "metadata": {
8     "department": "engineering",
9     "level": "senior",
10    "team_id": "backend_123"
11  }
12 }
```

Status: 201 Created Size: 295 Bytes Time: 5 ms

Response Headers 6 Cookies Results Docs {}

```
1 {
2   "success": true,
3   "message": "Employment created successfully",
4   "data": {
5     "employmentType": "full-time",
6     "fullTimeDetails": {
7       "benefits": [
8         "health insurance",
9         "401k",
10        "paid time off"
11       ],
12       "joiningDate": "2025-05-15T00:00:00.000Z"
13     },
14     "metadata": {
15       "department": "engineering",
16       "level": "senior",
17       "team_id": "backend_123"
18     }
19   }
20 }
```

DTOs:

Employment Dto:

```
1 import { IsEnum, IsNotEmpty, IsDate, IsNumber, IsOptional, ValidateIf, ValidateNested, IsObject } from 'class-validator';
2 import { Type } from 'class-transformer';
3
4 export class FullTimeDetailsDto {
5   @IsNotEmpty({ message: 'Benefits are required for full-time employees' })
6   benefits: string[];
7
8   @IsDate()
9   @Type(() => Date)
10  joiningDate: Date;
11 }
12
13 export class ContractorDetailsDto {
14   @IsDate()
15   @Type(() => Date)
16   contractEnd: Date;
17
18   @IsNumber()
19   hourlyRate: number;
20 }
21
22 export class EmploymentDto {
23   @IsEnum(['full-time', 'contractor'], { message: 'Employment type must be either full-time or contractor' })
24   employmentType: 'full-time' | 'contractor';
25
26   @ValidateIf(o => o.employmentType === 'full-time')
27   @IsNotEmpty({ message: 'Full time details are required for full-time employment' })
28   @ValidateNested()
29   @Type(() => FullTimeDetailsDto)
30   fullTimeDetails?: FullTimeDetailsDto;
31
32   @ValidateIf(o => o.employmentType === 'contractor')
33   @IsNotEmpty({ message: 'Contractor details are required for contractor employment' })
34   @ValidateNested()
35   @Type(() => ContractorDetailsDto)
36   contractorDetails?: ContractorDetailsDto;
37
38   @IsObject()
39   metadata: Record<string, string>;
40 }
```

## Validators:

### Employment Validator:

```
1 import { Injectable } from '@nestjs/common';
2 import { EmploymentDto } from '../dto/employment.dto';
3
4 @Injectable()
5 export class EmploymentValidator {
6   private readonly HOURLY_RATE_RANGE = {
7     US: [20, 100],
8     EU: [15, 80],
9   };
10
11   private readonly DEFAULT_RANGE = [15, 100];
12
13   validateEmployment(dto: EmploymentDto, countryCode?: string): ValidationErrors | null {
14     const errors: ValidationErrors = {};
15
16     // validates fullTimeDetails when employmentType is full-time
17     if (dto.employmentType === 'full-time') {
18       if (!dto.fullTimeDetails) {
19         errors['fullTimeDetails'] = {
20           code: 'REQUIRED_FIELD',
21           message: 'Full time details are required for full-time employment'
22         };
23       } else {
24         // validate joining date is in the future
25         const now = new Date();
26         if (dto.fullTimeDetails.joiningDate <= now) {
27           errors['fullTimeDetails.joiningDate'] = {
28             code: 'INVALID_DATE',
29             message: 'Joining date must be in the future'
30           };
31         }
32       }
33
34       // validate at least one benefit
35       if (!dto.fullTimeDetails.benefits || dto.fullTimeDetails.benefits.length === 0) {
36         errors['fullTimeDetails.benefits'] = {
37           code: 'EMPTY_ARRAY',
38           message: 'At least one benefit must be provided'
39         };
40       }
41     }
42   }
43 }
```

```

1
2 // validate contractorDetails when employmentType is contractor
3 if (dto.employmentType === 'contractor') {
4   if (!dto.contractorDetails) {
5     errors['contractorDetails'] = {
6       code: 'REQUIRED_FIELD',
7       message: 'Contractor details are required for contractor employment'
8     };
9   } else {
10    // validate contract end date is after start date (assuming current date as start)
11    const now = new Date();
12    if (dto.contractorDetails.contractEnd <= now) {
13      errors['contractorDetails.contractEnd'] = {
14        code: 'INVALID_DATE',
15        message: 'Contract end date must be after the current date'
16      };
17    }
18
19    // validate hourly rate based on locale
20    const range = countryCode && this.HOURLY_RATE_RANGE[countryCode] ? this.HOURLY_RATE_RANGE[countryCode] : this.DEFAULT_RANGE;
21    if (dto.contractorDetails.hourlyRate < range[0] || dto.contractorDetails.hourlyRate > range[1]) {
22      errors['contractorDetails.hourlyRate'] = {
23        code: 'RATE_OUT_OF_RANGE',
24        allowedRanges: {
25          [countryCode || 'default']: range
26        }
27      };
28    }
29  }
30 }
31
32 // validate metadata
33 if (dto.metadata) {
34   Object.keys(dto.metadata).forEach(key => {
35     if (!key.match(/^[a-z0-9_]+$/)) {
36       errors['metadata.${key}'] = {
37         code: 'INVALID_KEY_FORMAT',
38         message: 'Metadata keys must match /^[a-z0-9_]+$/ '
39       };
40     }
41
42     if (dto.metadata[key] && dto.metadata[key].length > 255) {
43       errors['metadata.${key}'] = {
44         code: 'VALUE_TOO_LONG',
45         message: 'Metadata values must be at most 255 characters'
46       };
47     }
48   });
49 }
50
51 return Object.keys(errors).length > 0 ? { errors } : null;
52 }
53 }
54
55 export interface ValidationErrors {
56   [key: string]: any;
57 }

```



## Employment Controller:

```
1 import { Controller, Post, Body, Headers, HttpException, HttpStatus } from '@nestjs/common';
2 import { EmploymentDto } from '../dto/employment.dto';
3 import { EmploymentValidator } from '../validators/employment.validator';
4
5 @Controller('employment')
6 export class EmploymentController {
7   constructor(private readonly employmentValidator: EmploymentValidator) {}
8
9   @Post()
10  createEmployment(@Body() employmentDto: EmploymentDto, @Headers('X-Country-Code') countryCode: string): any {
11    const validationErrors = this.employmentValidator.validateEmployment(employmentDto, countryCode);
12
13    if (validationErrors) {
14      throw new HttpException(validationErrors, HttpStatus.BAD_REQUEST);
15    }
16
17    return {
18      success: true,
19      message: 'Employment created successfully',
20      data: employmentDto
21    };
22  }
23 }
24 }
```

## Main.ts :

```
1 import { NestFactory } from '@nestjs/core';
2 import { ValidationPipe } from '@nestjs/common';
3 import { AppModule } from './app.module';
4
5 async function bootstrap() {
6   const app = await NestFactory.create(AppModule);
7
8   app.useGlobalPipes(
9     new ValidationPipe({
10       transform: true,
11       whitelist: true,
12       forbidNonWhitelisted: true,
13     }),
14   );
15
16   await app.listen(3000);
17 }
18 bootstrap();
```