nest generate controller –dry-run

nest g co

nest g module coffees

@Params() params

@Params(‘id’) id:string

@Body() body

@Body(‘name’) name

@HttpCode(HttpStatus.GONE)

As we are using express, we can get the response object as follows @Res() response; using this to send data, we loose the nestjs features like interceptors and http code decorators; Use nest standard response instead of this

PUT (SUPPOSE TO) replaces the entire resource and PATCH (SUPPOSE TO) replace only partially of the resource

Normally we use path parameters to identify specific resource and query parameters for filtration and sort

@Query() paginationQuery

Throw new HttpException(`Not Found`, HttpsStatus.NOT\_FOUND)

Throw new NotFoundException(‘Not found’); // automatically handles the response code and error like {statusCode: 404, “message”: “Coffee not found”, “error”: “Not Found”}

Use DTO’s for transferring data, nest g class coffees/sto/create-coffee.dto –no-spec

Its good practice to mark all properties in DTO readonly

Make different dtos for create, update, patch etc; reuse the properties if needed from already made dtos

To use validators and transofrmers, install class-validators and class-transformer package and register the validators in main.ts as app.useGlobalPiper(new ValidationPipe()); After adding this you can directly import the validators in DTO’s and use them there

We can reuse already made DTO’s using mappedTypes package. Install the mapped Type package and use already made DTO’s as export class UpdateCoffeeDTO extends PartialType(CreateCoffeeDTO){}; This inherits all the validation rules of CreateCoffeeDTO and also applied additional rule of optional (@IsOptional()) on all the inherited fields

With parameters to ValidationPipe({whitelist:true}) we can strip all the extra properties that were sent in the request, we can also add ValidationPipe({whitelist:true, forbidWhitelisted:true}) to return error if extra properties are sent in requests

If we pass forbidNonWhitelisted: true in validation pipe, it would give an error if extra properties are sent in the request

Transform:true passing in the validation pipe help transform the objects that are sent in the request. By Default all data is sent in a request as JSON, passing transform flag will enable validation pipe to transform JSON data into required type (or primitive type) mentioned in the controller. This helps us to transform the body, params etc into actual types that are mentioned at the controller level.

Other Building Blocks by example:

There are some other building blocks in NestJs.

1. Exception Filters
2. Pipes
3. Guards
4. Interceptors

Exception Filters: Used to handle and process some unhandled exception that might occur in our application. They let us control the exact flow of responses that we send back to the client

Pipes: These are used mainly for 2 things. (i) Transformation, meaning to transform the input data to desired output (ii) Validation

Guards: Determines if a given request needs certain conditions like Authorization, Roles, Authentication etc. If condition is met then the request is allowed to access the route.

Interceptors: These have many useful capabilities like bind extra logic before or after method execution, transform the result returned by a method, extend basic method behavior etc.

These building blocks are available to bind in following

1. Global scope
2. Controller Scope
3. Method Scope
4. Params Scope (Pipe only)

Catch Exceptions with Filters:

NestJS comes with a build in exceptions layer, responsible for processing all unhandled exceptions across our application. When an exception is not handled by our application, its automatically caught by this layer that sends the appropriate user friendly response. Out of the box, this action is performed by a built in global exception filter. While this basic exception filter may handle many uses cases for us, we may want full control over it. For example we may want to implement exception logging or return error in different JSON schema.

Protect routes with Guards:

Guards have a single responsibility which is to determine if the given request is allowed access to something.

If the request meets certain conditions mentioned in the guard, it will be allowed to the route else it wont be allowed and error would be thrown .

Add Pointcuts with Interceptors:

Interceptors make it possible for us to

1. Bind extra logic before or after method execution
2. Transform the result returned from a method
3. Transform the exception thrown from a method
4. Extend basic method behavior
5. Or completely overwriting a method depending on a specific condition

Pipes:

Pipes have 2 typical use cases. Transformation and Validation

Middleware:

Middleware is called before the route handler or any other building blocks are processed, this includes inceptors, guards and pipes. Middleware functions have access to the request and response objects and are not tied to any specific method, rather they are tied to a route path.

Middleware functions can perform the following tasks.

1. Executing code
2. Making changes to request and response objects
3. Ending the request and response cycle
4. Calling the next middleware function in the callstack

Middleware must call next() method if its not ending the request response cycle. Otherwise the request will be hanged.

Custom Next middleware can be implemented in either a function or a class. Function middleware is stateless; it doesn’t have access to dependencies or the DI container. Class middlewares can inject the dependencies.