**NESTJS RESEARCH OUTLINE**

TABLE OF CONTENTS

[1 SET UP 6](#_Toc61208250)

[2 CONTROLLER 8](#_Toc61208251)

[2.1 Definition of controller 8](#_Toc61208252)

[2.2 Request object 10](#_Toc61208253)

[2.3 Asynchronicity 11](#_Toc61208254)

[2.4 Full resource sample 12](#_Toc61208255)

[2.5 Getting up and running 13](#_Toc61208256)

[3 PROVIDERS 14](#_Toc61208257)

[3.1 Concepts 14](#_Toc61208258)

[3.2 Services 14](#_Toc61208259)

[3.2.1 Create a service using CLI: 14](#_Toc61208260)

[3.2.2 Injection copes: 14](#_Toc61208261)

[3.3 Provider Registration 15](#_Toc61208262)

[4 MODULES 18](#_Toc61208263)

[4.1 Concepts 18](#_Toc61208264)

[4.1.1 Metadata 19](#_Toc61208265)

[4.2 Features Modules 20](#_Toc61208266)

[4.3 Shared Modules 20](#_Toc61208267)

[4.4 Module Re-exporting 21](#_Toc61208268)

[4.5 Global Modules 21](#_Toc61208269)

[4.6 Dynamic modules 22](#_Toc61208270)

[5 MIDDLEWARE 25](#_Toc61208271)

[5.1 Concept 25](#_Toc61208272)

[5.2 Applying Middleware 26](#_Toc61208273)

[5.3 Middleware consumer 27](#_Toc61208274)

[5.4 Excluding route 28](#_Toc61208275)

[5.5 Global middleware 28](#_Toc61208276)

[6 PIPES 29](#_Toc61208277)

[6.1 Concepts: 29](#_Toc61208278)

[6.2 Built-in pipes 30](#_Toc61208279)

[6.3 Binding pipes 30](#_Toc61208280)

[6.4 Schema based validation 31](#_Toc61208281)

[6.4.1 Object based validation 31](#_Toc61208282)

[6.4.2 Class validator 33](#_Toc61208283)

[6.5 Global scoped pipes 35](#_Toc61208284)

TABLE OF FIGURES

[Figure 1. Set up project with command line 6](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208285)

[Figure 2. Default structure 6](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208286)

[Figure 3. Brief overview of default files 7](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208287)

[Figure 4. Core controllers' responsibility 8](#_Toc61208288)

[Figure 5. First example of controller 8](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208289)

[Figure 6. Promise for HTTP Handling 11](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208290)

[Figure 7. Observable for HTTP handling 11](#_Toc61208291)

[Figure 8. Full source sample for controller 12](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208292)

[Figure 9. Controller defined in Module 13](#_Toc61208293)

[Figure 10. Example of definition of a service 15](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208294)

[Figure 11. Example of consuming a service (provider) in a controller 16](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208295)

[Figure 12. Example of registration of a service in Module 17](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208296)

[Figure 13. Example of modules 18](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208297)

[Figure 14. Cat Module 19](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208298)

[Figure 15. Shared module 20](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208299)

[Figure 16. Module re-export 21](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208300)

[Figure 17. Global module 22](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208301)

[Figure 18. Dynamic database module 23](#_Toc61208302)

[Figure 19. Import dynamic module 24](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208303)

[Figure 20. Middleware 25](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208304)

[Figure 21. Example of middleware 25](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208305)

[Figure 22. Apply middleware 26](#_Toc61208306)

[Figure 23. Middleware consumer 27](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208307)

[Figure 24. Apply multiple middlewares 27](#_Toc61208308)

[Figure 25. Excluding route 28](#_Toc61208309)

[Figure 26. Global middleware 28](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208310)

[Figure 27. Pipe in the big picture 29](file:///E:\Projects\SELF_STUDY\NestJS_Projects\Docs\NestJS_Research.docx#_Toc61208311)

TABLE OF TABLES

[Table 1. Common decorator for HTTP Handler 9](#_Toc61208312)

[Table 2. Decorators for request object 10](#_Toc61208313)

[Table 3. Three types of injection copes 14](#_Toc61208314)

[Table 4. Metadata in @Module decorator 19](#_Toc61208315)

# SET UP

Figure 1. Set up project with command line

Default structure of generated project with CLI

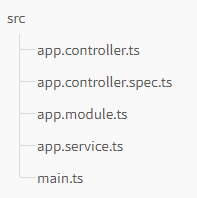
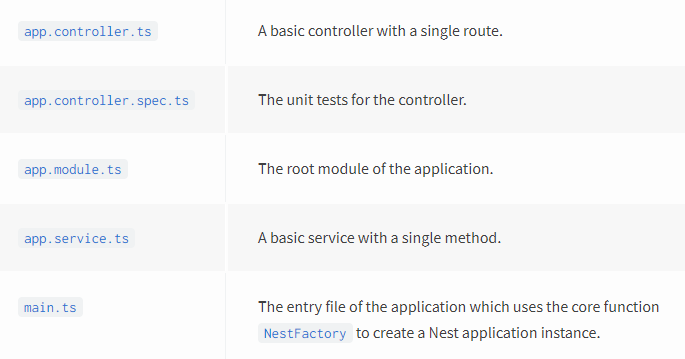


Figure 2. Default structure

There are two HTTP platforms supported: **express** (by default) and **fastify.**

Figure 3. Brief overview of default files



# CONTROLLER

## Definition of controller

Controllers are responsible for handling incoming requests and returning responses to the client.

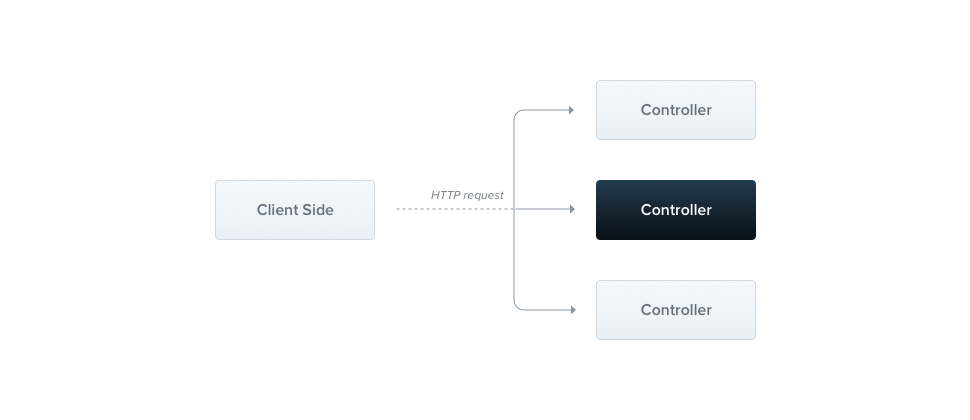


Figure 4. Core controllers' responsibility

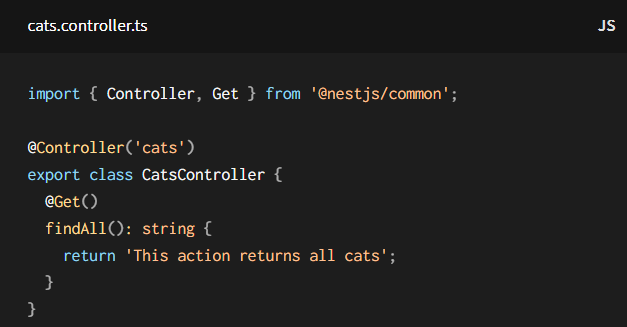
 Example:

Figure 5. First example of controller

To mark this call is controller we use @Controller decorator with the optional prefix routing. (In fig 5, we use prefix routing is cats).

Next, @Get decorator defines HTTP request method.

Additionally, there are other decorators supporting HTTP handling, below:

Table 1. Common decorator for HTTP Handler

|  |  |  |
| --- | --- | --- |
| Decorator name | Meaning | Example |
| @Get()  @Post()  @Put()  @Delete()  @Patch()  @Options()  @All() | HTTP methods decorator |  |
| @HttpCode() | Status code  (default is 200 for all but Post is 201) |  |
| @Header() |  | @Header('Cache-Control', 'none') |
| @Redirect() | Redirection |  |

## Request object

Table 2. Decorators for request object

|  |  |
| --- | --- |
| Syntax | Alias for |
| @Request(), @Req() | req |
| @Response(), @Res()\* | res |
| @Next() | next |
| @Session() | req.session |
| @Param(key?: string) | req.params / req.params[key] |
| @Body(key?: string) | req.body / req.body[key] |
| @Query(key?: string) | req.query / req.query[key] |
| @Headers(name?: string) | req.headers / req.headers[name] |
| @Ip() | req.ip |
| @HostParam() | req.hosts |
| @Request(), @Req() | req |
| @Response(), @Res()\* | res |
| @Next() | next |
| @Session() | req.session |
| @Param(key?: string) | req.params / req.params[key] |
| @Body(key?: string) | req.body / req.body[key] |
| @Query(key?: string) | req.query / req.query[key] |
| @Headers(name?: string) | req.headers / req.headers[name] |
| @Ip() | req.ip |
| @HostParam() | req.hosts |

## Asynchronicity

There are two approaches: PROMISE and OBSERVABLE

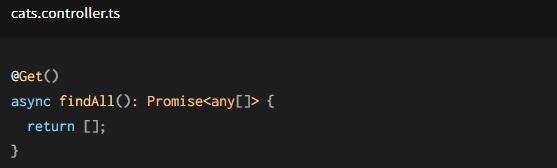


Figure 6. Promise for HTTP Handling

=> Every async function has to return the promise. This means that you can return a deferred value that the Nest will be able to resolve by itself.

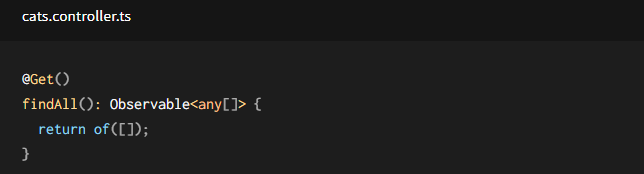


Figure 7. Observable for HTTP handling

=> Nest will automatically subscribe to the source underneath and take the last emitted value (once the stream is completed).

## Full resource sample

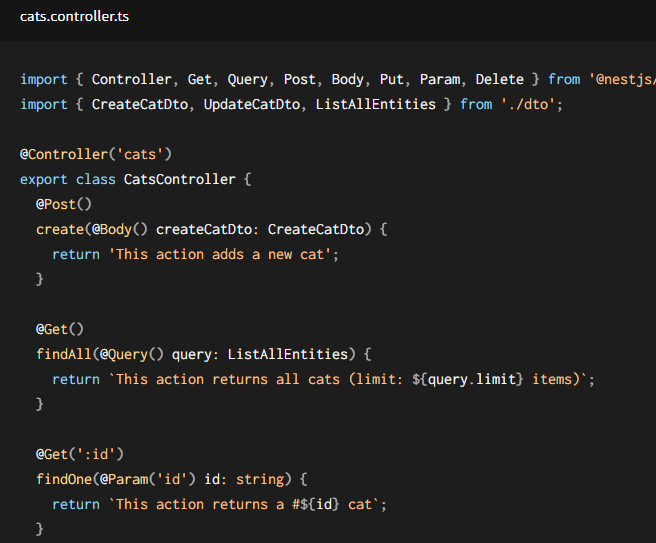
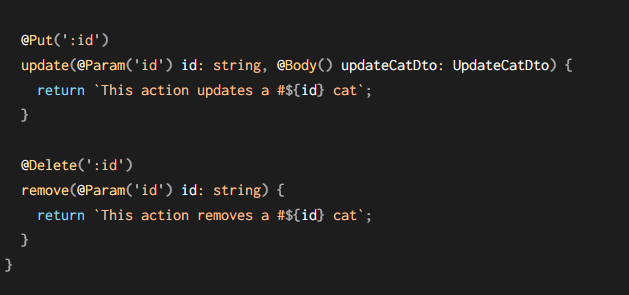


Figure 8. Full source sample for controller

## Getting up and running

Just like components in angular that are mounted at the Module. Similarly, controllers would be defined in Module or the Nest will never execute the controller in this case and others in other cases.



Figure 9. Controller defined in Module

# PROVIDERS

## Concepts

* Many of basic Nest classes may be treated as a provider (e.g. services, repositories, factories, helpers, …)
* Provider-defined classes indicate that they can inject dependencies
* A provider is simply a class annotated with an @Injectable() decorator.

## Services

### Create a service using CLI:

$ nest g service cats

### Injection copes:

Table 3. Three types of injection copes

|  |  |
| --- | --- |
| **DEFAULT** | A single instance of the provider is shared across the entire application. The instance lifetime is tied directly to the application lifecycle. Once the application has bootstrapped, all singleton providers have been instantiated. Singleton scope is used by default. |
| **REQUEST** | A new instance of the provider is created exclusively for each incoming request. The instance is garbage-collected after the request has completed processing. |
| **TRANSIENT** | Transient providers are not shared across consumers. Each consumer that injects a transient provider will receive a new, dedicated instance. |

**HINT:** Using singleton scope is recommended for most use cases. Sharing providers across consumers and across requests means that an instance can be cached and its initialization occurs only once, during application startup.

## Provider Registration

When we define a provider, normally, we will register the providers in the module that contains its classes using that provider.

EX:

Figure 10. Example of definition of a service



Figure 11. Example of consuming a service (provider) in a controller



Figure 12. Example of registration of a service in Module

# MODULES

## Concepts

* A module is a class annotated with @Module() decorator.
* The @Module() provides metadata to organize the application structure.
* Each application has at least one module – so called the root module.
* Most application, the resulting architecture will employ multiple modules.
* Each module will encapsulate a closely related set of capabilities.

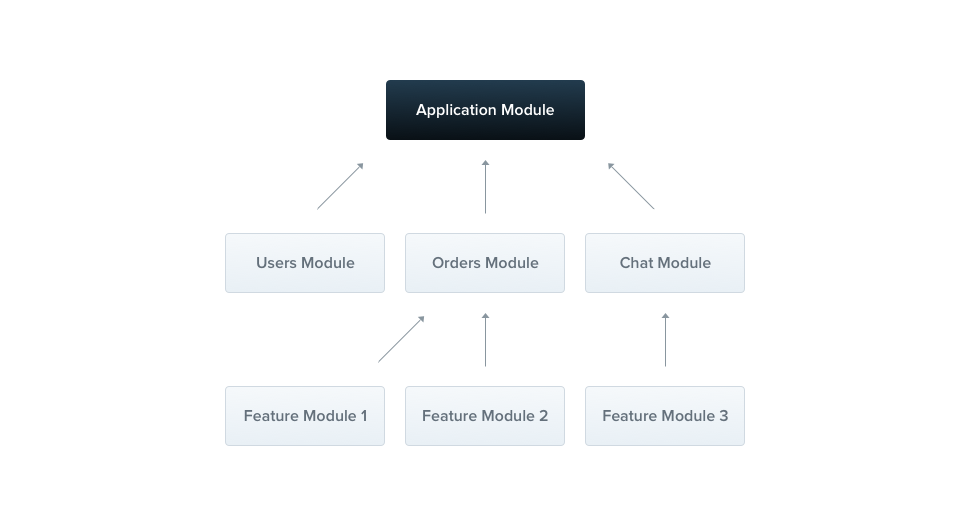


Figure 13. Example of modules

Hint: To create a module using the CLI, simply execute the *$ nest g module [module\_name]* command.

### Metadata

Table 4. Metadata in @Module decorator

|  |  |
| --- | --- |
| **Providers** | The providers that will be instantiated by the Nest injector and that may be shared at least across this module |
| **controllers** | The set of controllers defined in this module which have to be instantiated |
| **Imports** | The list of imported modules that export the providers which are required in this module |
| **Exports** | The subset of providers that are provided by this module and should be available in other modules which import this module |



Figure 14. Cat Module

## Features Modules

A feature module simply organizes code relevant for a specific feature, which helps us:

* Keep code organized and establishing clear boundaries
* Manage the complexity and follow the SOLID principles

There are already many aforementioned examples for this kind of module.

## Shared Modules

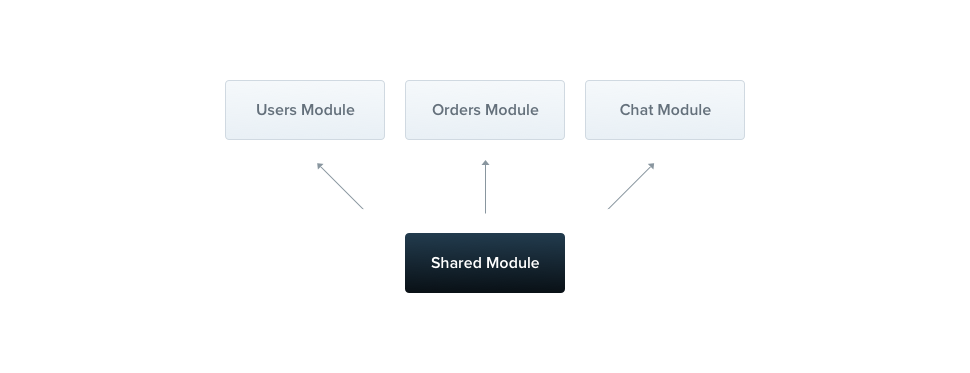
In Nest, Modules are singleton by default

Figure 15. Shared module

Every module is automatically a shared module. Once created, it can be reused by any module

## Module Re-exporting

Beside modules can export their internal providers, they also can re-export modules that they import.

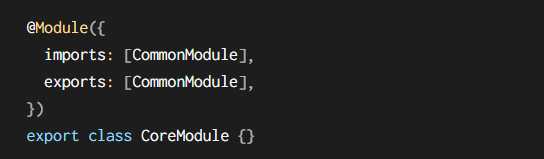


Figure 16. Module re-export

## Global Modules

Unlike in Angular, you can mark a provider with provided in root that makes the provider accessible everywhere in the application without register in the specific module to consume.

However, in the context of Nest App, providers are encapsulated inside the module scope. That means you can’t consume that providers elsewhere unless you import the module where the providers register.

Therefore, if you want to have some set of providers which are available everywhere out-of-the-box, simply make the module **GLOBAL** with the @Global() decorator



Figure 17. Global module

## Dynamic modules

Dynamic modules enable developers to register and configure the providers dynamically.

To make it clear, we will analyze the dynamic module in the example below:

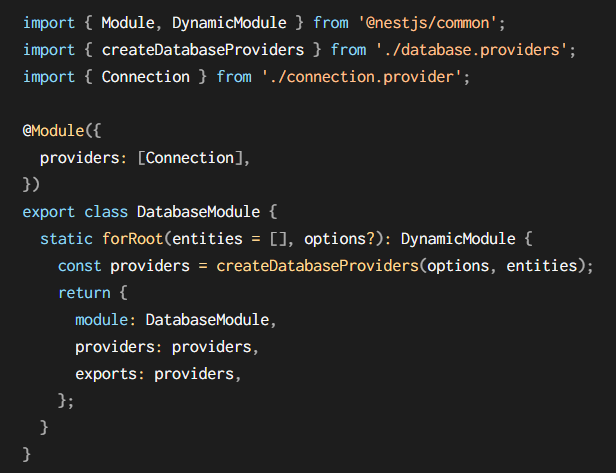


Figure 18. Dynamic database module

As we can see:

* The module defines Connection providers as default.
* Depending on the options and entities passed into the forRoot method, it will build up a set of providers and expose them.
* The DynamicMoule return is extend rather than override the base metadata defined in @Module() decorator.

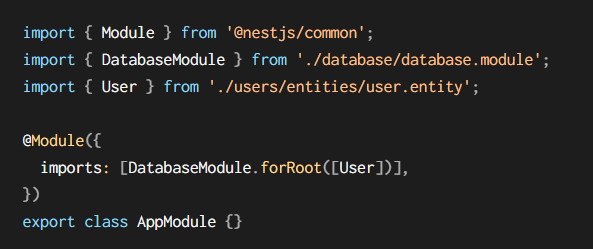
Then the module can be imported and configured in the following manner:

Figure 19. Import dynamic module

# MIDDLEWARE

## Concept

Is simply a function called before route handler.

Access to request and response objects.

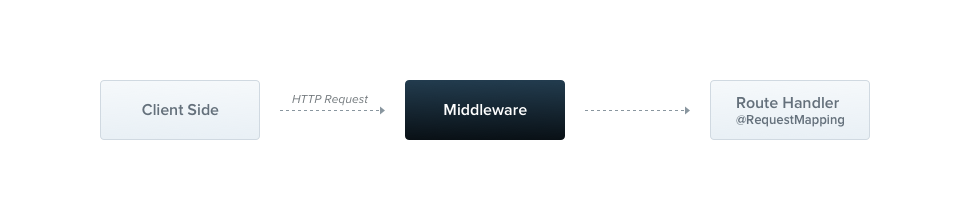


Figure 20. Middleware

In Nest, you can implement the middleware in either a function or a class, and middleware can inject in the module



Figure 21. Example of middleware

## Applying Middleware

We set up middleware by using configure() method in module class



Figure 22. Apply middleware

## Middleware consumer

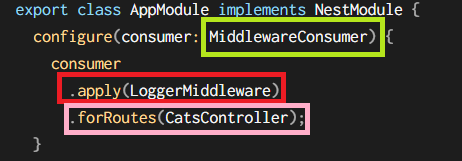


Figure 23. Middleware consumer

**Middleware Consumer:**

* The Middleware consumer is a helper class.
* Provide several built-in methods to manage middleware
* Can be chained in fluent style

**Apply method:**

Can take either a single middleware, or multiple middlewares



Figure 24. Apply multiple middlewares

**The forRoutes method:**

Take a signle string, multiple strings, a RoutInfo object, a controller class and even multiple controller classes

## Excluding route

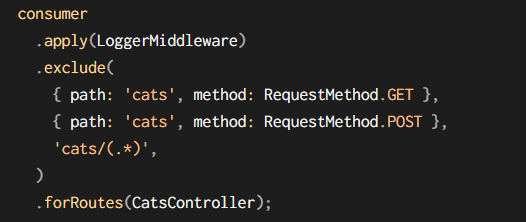


Figure 25. Excluding route

## Global middleware

When we want to bind middleware to every registered route at once

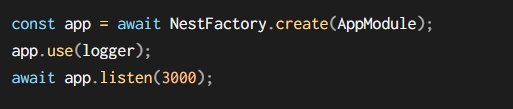


Figure 26. Global middleware

# PIPES

## Concepts:

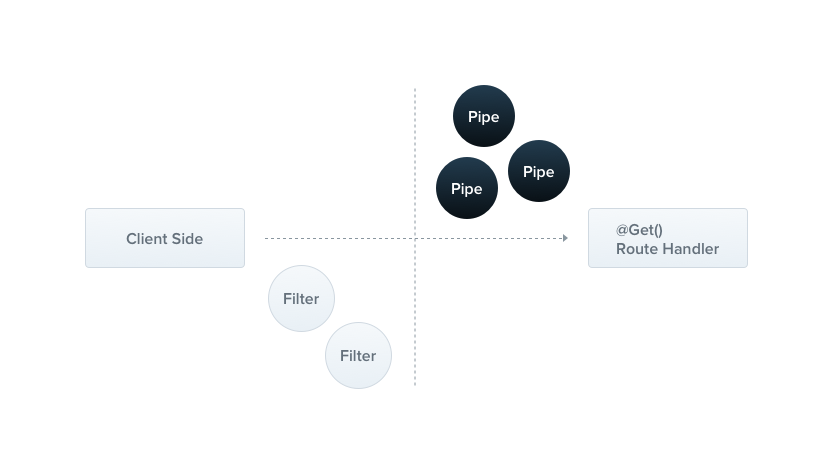


Figure 27. Pipe in the big picture

* Pipe is a class annotated with @Injectable() decorator (though angular has @pipe decorator)
* Implement from the PipeTransform interface
* Two typical use cases:
* Transformation
* Validation
* Pipe run insides exception zone => When pipes throw an exception, It is handled by exception layer.

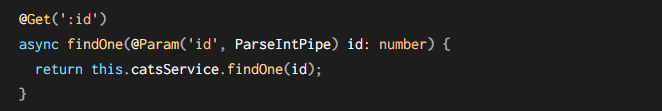
## Built-in pipes

There are six pipes available:

* ValidationPipe
* ParseIntPipe
* ParseBoolPipe
* ParseArrayPipe
* ParseUUIDPipe
* DefaultValuePipe

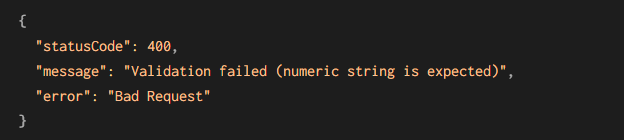
They're exported from the **@nestjs/common** package.

## Binding pipes

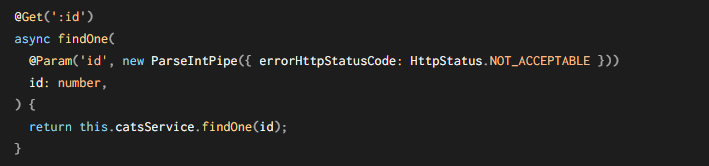
To use a pipe, we need to bind an instance of the pipe class to the appropriate context 

This will ensure the parameter we receive is the number or throw exception:

EX: *GET localhost:3000/abc*



We can change the default HTTP status code by adding like below:



## Schema based validation

We use validation pipe with some powerful libraries to implement this kind of validation

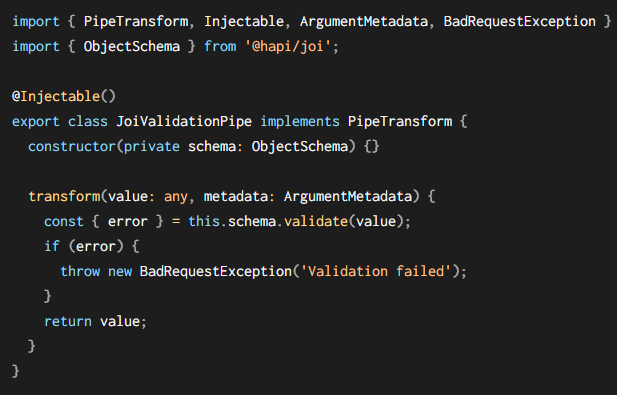
### Object based validation

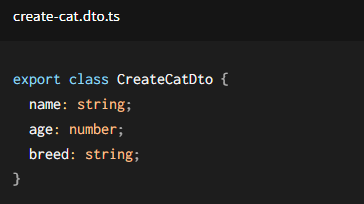
Use Joi library allows you to create schemas.

Start by installing the required package:

*$ npm install --save @hapi/joi*

*$ npm install --save-dev @types/hapi\_\_joi*

Then we create JoiValidationPipe:

Create cat dto:

Define the create cat schema:



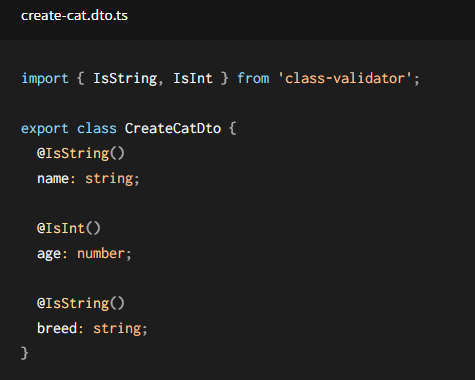
Binding the pipe to routing handler



### Class validator

Install class-validator library:

*npm i --save class-validator class-transformer*

**Add the decorator to the class:

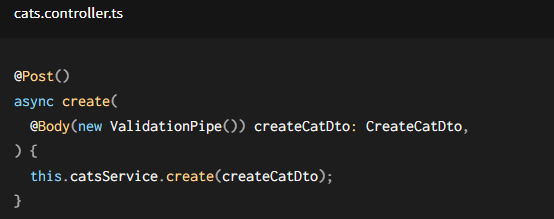
Create the validation pipe:

Extract the metatype field from ArgumentMetadata

The helper function toValidate(). It's responsible for bypassing the validation step when the current argument being processed is a native JavaScript type (these can't have validation decorators attached, so there's no reason to run them through the validation step).

the class-transformer function plainToClass() to transform our plain JavaScript argument object into a typed object so that we can apply validation

Then bind the validation pipe



## Global scoped pipes

It is applied to every route handler across the entire application.



# GUARDS

## Conceptors

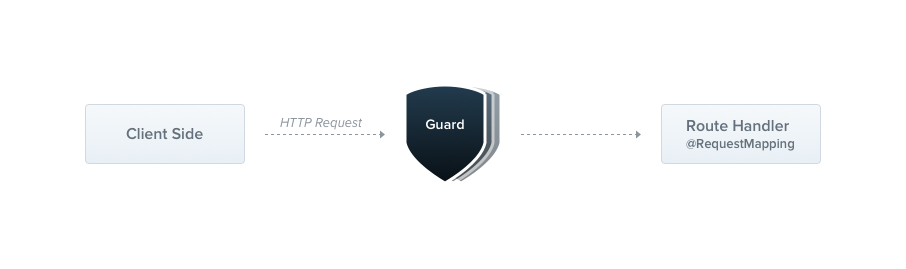


Figure 28. Guard in big picture

Guard is a class annotated with @Injectable() decorator

Guards should implement the CanActivate interface

Guards determine whether a given request will be handled by the route handler or not, depending on the certain conditions => it is often referred to as **Authorization**

**Comparation:**

|  |  |
| --- | --- |
| Authorization at **Middleware (traditional way)** | Authorization at **Guards  (modern way)** |
| Middleware is dumb => do not know which handler will be executed after calling the **next()** function | Guards have access to the **ExecutionContext** instance => know exactly what’s going to be executed next |

**NOTE:**

*Guards are executed* ***after*** *each middleware, but* ***before*** *any interceptor or pipe.*

## Authorization guard

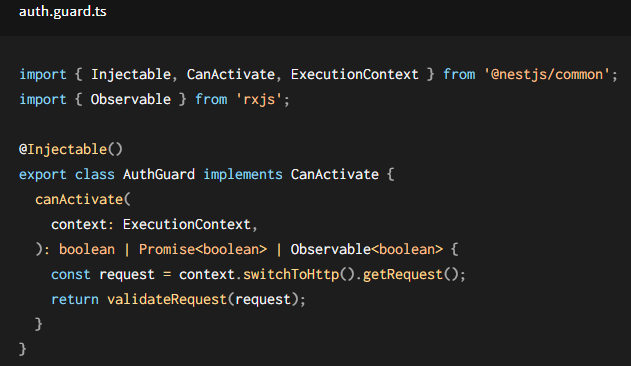


Figure 29. A simple guard

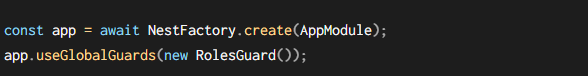
* The logic inside the **validateRequest()** function can be as simple or sophisticated as needed.
* Every guard must implement a **canActivate()** function => this function returns boolean
* if it returns true, the request will be processed.
* if it returns false, Nest will deny the request.
* Execution context:
* It inherits from ArgumentsHost
* Provide additional details about the current execution process
* Help in building more generic guards that can work across a broad set of controllers, methods, and execution contexts.

*Read more at* [*https://docs.nestjs.com/fundamentals/execution-context*](https://docs.nestjs.com/fundamentals/execution-context)

## Binding Guards

* Guards can be controller-scoped, method-scoped, or global-scoped.
* @UserGuars decorator may take a single argument, or a comma-separated list of arguments.

In order to set up a global guard, use the useGlobalGuards() method of the Nest application instance:



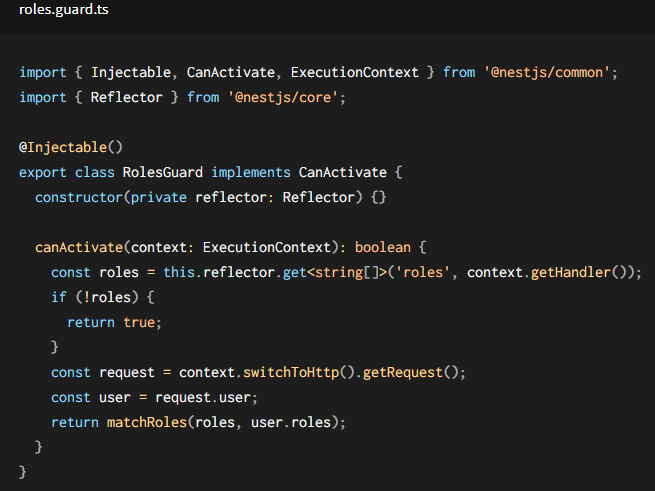
## Setting roles per handler

We will go through these steps (from simplicity to complexity) to make a complete Guard for permission-based authorization.

### Using custom metadata

Nest provides the ability to attach custom metadata to route handlers through the @SetMetadata() decorator.

### Using reflector and Guard



# INTERCEPTORS

## Concepts

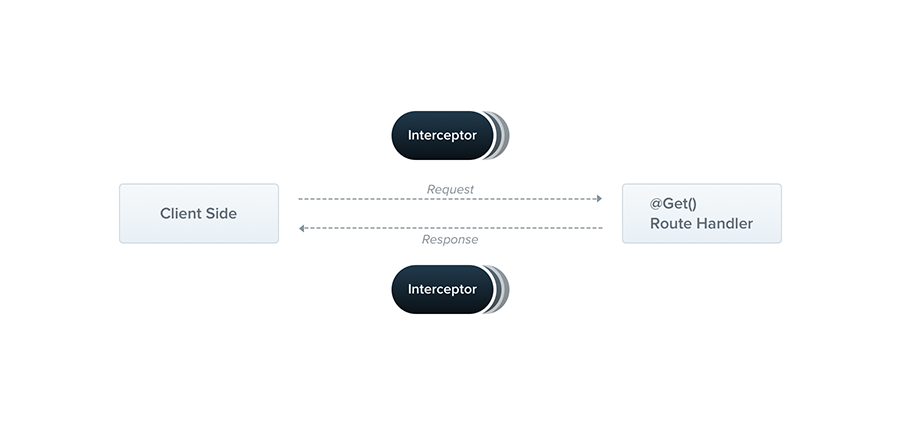


Figure 30. Interceptor in big picture

* An interceptor is a class annotated with the @Injectable() decorator.
* Interceptors should implement the NestInterceptor interface.
* Possibilities:
* bind extra logic before / after method execution
* transform the result returned from a function
* transform the exception thrown from a function
* extend the basic function behavior
* completely override a function depending on specific conditions (e.g., for caching purposes)