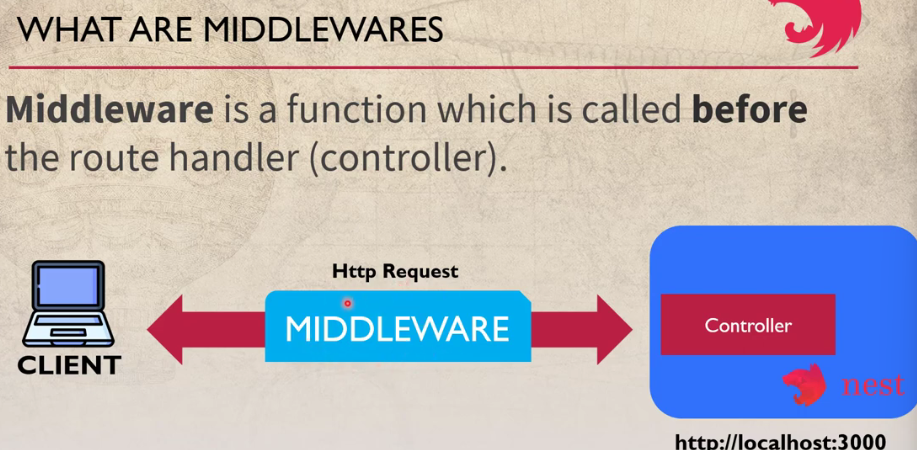
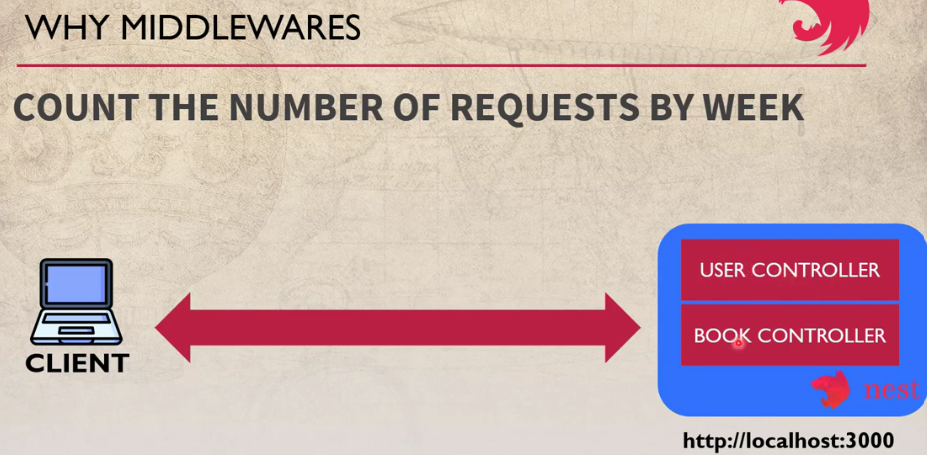
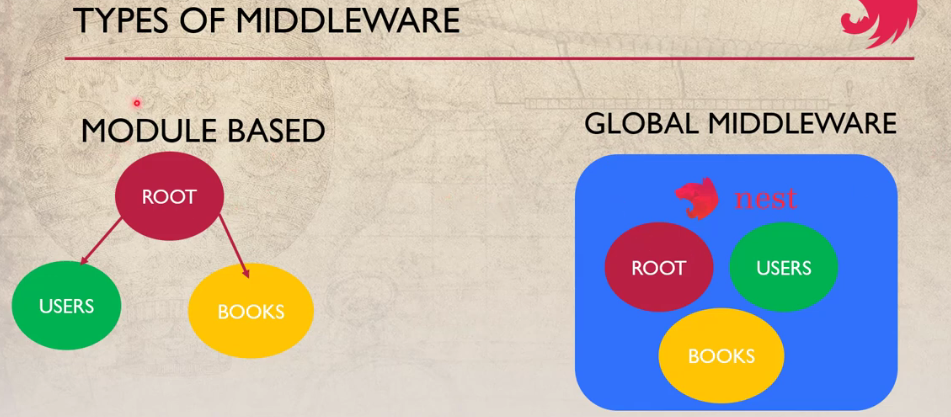
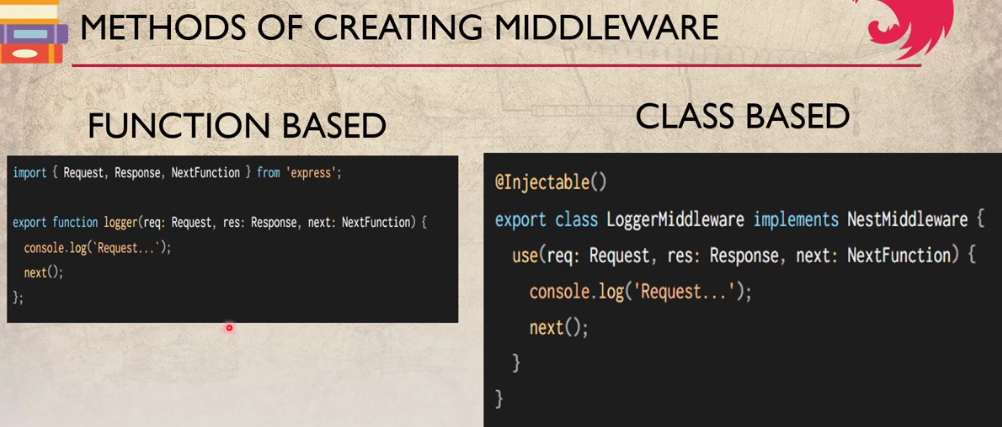
Middlewares

* Module based
* Global – only through function based. Class based cannot be used

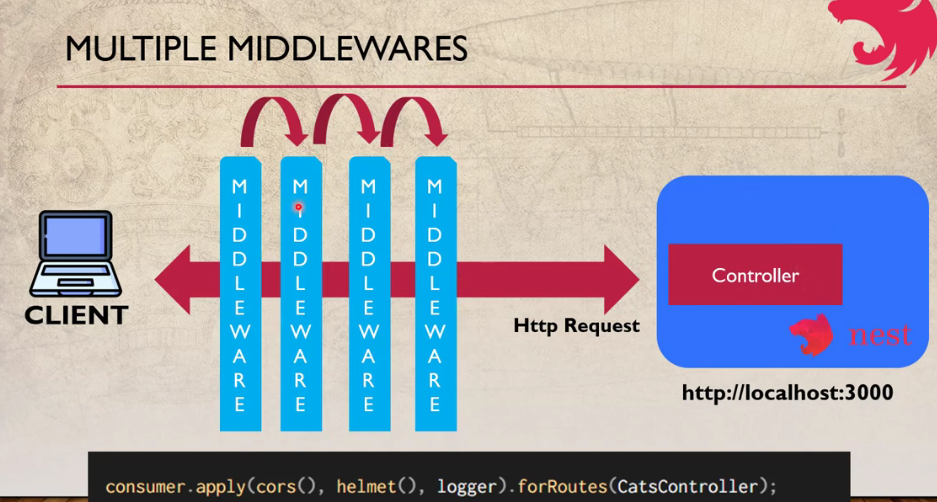




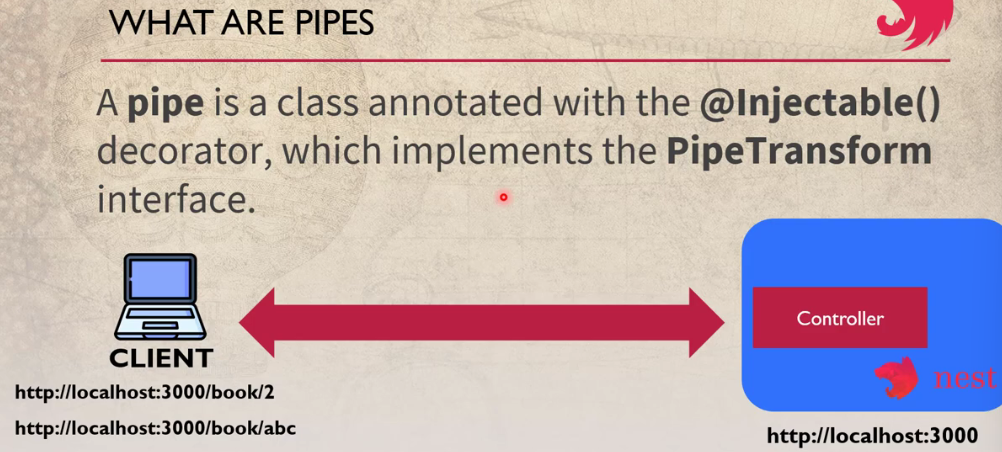








Pipes



**Built-in pipes**[**#**](https://docs.nestjs.com/pipes#built-in-pipes)

Nest comes with nine pipes available out-of-the-box:

* ValidationPipe
* ParseIntPipe
* ParseFloatPipe
* ParseBoolPipe
* ParseArrayPipe
* ParseUUIDPipe
* ParseEnumPipe
* DefaultValuePipe
* ParseFilePipe

**Binding pipes**[**#**](https://docs.nestjs.com/pipes#binding-pipes)

To use a pipe, we need to bind an instance of the pipe class to the appropriate context. In our ParseIntPipe example, we want to associate the pipe with a particular route handler method, and make sure it runs before the method is called. We do so with the following construct, which we'll refer to as binding the pipe at the method parameter level:

@Get(':id')

async findOne(@Param('id', ParseIntPipe) id: number) {

return this.catsService.findOne(id);

}

This ensures that one of the following two conditions is true: either the parameter we receive in the findOne() method is a number (as expected in our call to this.catsService.findOne()), or an exception is thrown before the route handler is called.

For example, assume the route is called like:

GET localhost:3000/abc

Nest will throw an exception like this:

{

"statusCode": 400,

"message": "Validation failed (numeric string is expected)",

"error": "Bad Request"

}

The exception will prevent the body of the findOne() method from executing.

In the example above, we pass a class (ParseIntPipe), not an instance, leaving responsibility for instantiation to the framework and enabling dependency injection. As with pipes and guards, we can instead pass an in-place instance. Passing an in-place instance is useful if we want to customize the built-in pipe's behavior by passing options:

@Get(':id')

async findOne(

@Param('id', new ParseIntPipe({ errorHttpStatusCode: HttpStatus.NOT\_ACCEPTABLE }))

id: number,

) {

return this.catsService.findOne(id);

}

Binding the other transformation pipes (all of the **Parse\*** pipes) works similarly. These pipes all work in the context of validating route parameters, query string parameters and request body values.

For example with a query string parameter:

@Get()

async findOne(@Query('id', ParseIntPipe) id: number) {

return this.catsService.findOne(id);

}

**Custom pipes**[**#**](https://docs.nestjs.com/pipes#custom-pipes)

As mentioned, you can build your own custom pipes. While Nest provides a robust built-in ParseIntPipe and ValidationPipe, let's build simple custom versions of each from scratch to see how custom pipes are constructed.

We start with a simple ValidationPipe. Initially, we'll have it simply take an input value and immediately return the same value, behaving like an identity function.

validation.pipe.ts

**JS**

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

@Injectable()

export class ValidationPipe implements PipeTransform {

transform(value: any, metadata: ArgumentMetadata) {

return value;

}

}

**Hint**PipeTransform<T, R> is a generic interface that must be implemented by any pipe. The generic interface uses T to indicate the type of the input value, and R to indicate the return type of the transform() method.

Every pipe must implement the transform() method to fulfill the PipeTransform interface contract. This method has two parameters:

* value
* metadata

The value parameter is the currently processed method argument (before it is received by the route handling method), and metadata is the currently processed method argument's metadata. The metadata object has these properties:

export interface ArgumentMetadata {

type: 'body' | 'query' | 'param' | 'custom';

metatype?: Type<unknown>;

data?: string;

}

These properties describe the currently processed argument.

|  |  |
| --- | --- |
| type | Indicates whether the argument is a body @Body(), query @Query(), param @Param(), or a custom parameter (read more [**here**](https://docs.nestjs.com/custom-decorators)). |
| metatype | Provides the metatype of the argument, for example, String. Note: the value is undefined if you either omit a type declaration in the route handler method signature, or use vanilla JavaScript. |
| data | The string passed to the decorator, for example @Body('string'). It's undefined if you leave the decorator parenthesis empty. |

Class Validator

Decoration based validation