NestJS Pipes and File Upload

Course Code: CSC 4182 Course Title: Advanced Programming In Web Technologies

Dept. of Computer Science
Faculty of Science and Technology

Lecture No:	**1**	**Week No:**	**03**	**Semester:**	
Lecturer:	*Sazzad Hossain; sazzad@aiub.edu*				

Lecture Outline

- ✓ NestJS Pipes
- ✓ Pipes Types
- ✓ Transformation Pipes
- ✓ Validation Pipes
- ✓ class-validator
- ✓ class-validator Decorators
- ✓ File Upload
- ✓ File Validation
- ✓ FileInterceptor

NestJS Pipes

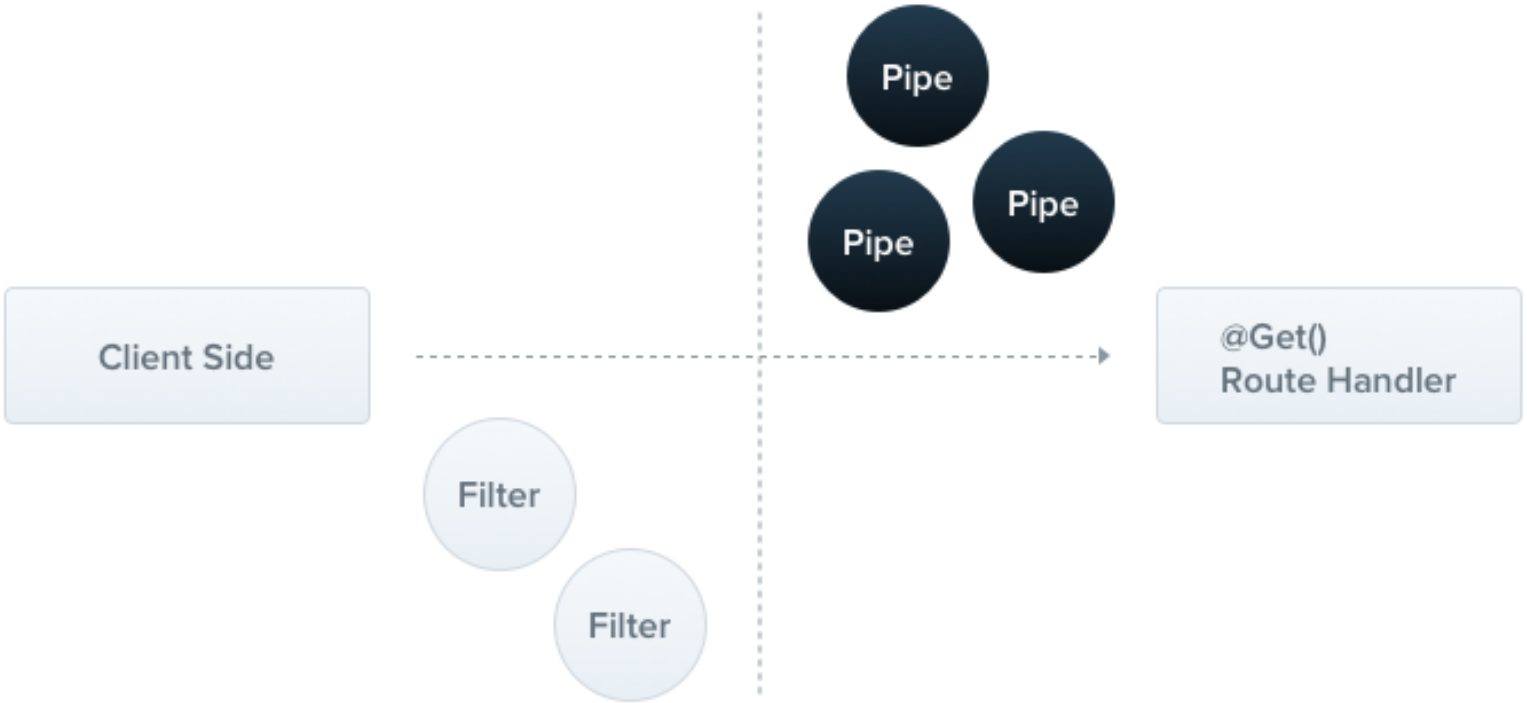
Pipes are a powerful feature that allows you to

- **transform,**
- **validate**
- **manipulate**

data as it flows through the application.

Pipes act as middleware components that can be applied to route handlers, controller methods, or globally to the entire application.

NestJS Pipes



Pipes Types

Transformation: transform input data to the desired form (e.g., from string to integer)

Validation: evaluate input data and if valid, simply pass it through unchanged; otherwise, throw an exception

Transformation Pipes

Following are the Pre-define Classes of Transformation Pipes;

- ParseIntPipe
- ParseFloatPipe
- ParseBoolPipe
- ParseArrayPipe
- ParseUUIDPipe
- ParseEnumPipe
- DefaultValuePipe
- ParseFilePipe

Transformation Pipes


```
import { Controller, Get, Param, ParseIntPipe } from
'@nestjs/common';

@Controller('users')
export class UserController {
  @Get(':id')
  getUser(@Param('id', ParseIntPipe) userId: number) {
    // userId will be automatically transformed to a number
    // ...
  }
}
```

Validation Pipes

- Nest uses **class-validator** library to validate data.
- **class-validator** library allows decorator-based validation.
- Details of class-validator can be found here; https://github.com/typestack/class-validator

class-validator

To perform validation using class-validator, use following command in your nestjs project folder

```
npm install class-validator class-transformer
```

Import the necessary modules in your DTO

```
import { IsString, IsEmail } from 'class-validator';
```

class-validator

The DTO file will look like below;

```
import { IsString, IsEmail } from 'class-validator';

export class CreateUserDto {
    @IsString()
    name: string;

    @IsEmail()
    email: string;

    @IsString()
    password: string;
  }
```

class-validator

Choose the route you want apply the validation

```
@Post('/addadmin’)
@UsePipes(new ValidationPipe())// Apply the validation
addAdmin(@Body() data:AdminDTO):string {
    console.log(data);
return this.adminService.addAdmin(data);
}
```

class-validator Decorators

Value Validation:

IsDefined: Checks if the value is defined (not undefined or null).
IsNotEmpty: Checks if the value is not empty.
IsEmpty: Checks if the value is empty.
Equals: Checks if the value is equal to a specified value.
NotEquals: Checks if the value is not equal to a specified value.
Contains: Checks if the value contains a specified substring.
NotContains: Checks if the value does not contain a specified substring.
IsIn: Checks if the value is one of the specified values.
IsNotIn: Checks if the value is not one of the specified values.
IsBoolean: Checks if the value is a boolean.
IsDate: Checks if the value is a valid date.
IsString: Checks if the value is a string.
IsNumber: Checks if the value is a number.
IsInt: Checks if the value is an integer.
IsPositive: Checks if the value is a positive number.
IsNegative: Checks if the value is a negative number.
IsArray: Checks if the value is an array.
IsEnum: Checks if the value is part of an enum.

class-validator Decorators

String Validation:
Length: Checks if the string length is within a specified range.
MinLength: Checks if the string length is greater than or equal to a specified minimum value.
MaxLength: Checks if the string length is less than or equal to a specified maximum value.
IsEmail: Checks if the value is a valid email address.
IsUrl: Checks if the value is a valid URL.
IsPhoneNumber: Checks if the value is a valid phone number.
Matches: Checks if it matches with regular express.

One Example for Matches given below;

```
@Matches(/^[A-Za-z]+$/)
firstName: string;
```

File Upload

- File Upload is handled by built-in module based on the **multer** middleware package for Express.
- **Multer** handles data posted in the multipart/form-data format, which is primarily used for uploading files via an **HTTP POST request.**
- installing **Multer typings** package:

```
$ npm i -D @types/multer
```

Import the necessary modules and decorators:

```
import { Controller, Post, UseInterceptors, UploadedFile }
from '@nestjs/common';
import { FileInterceptor } from '@nestjs/platform-express’;
import { MulterError, diskStorage } from "multer";
```

File Upload

To upload a single file, simply tie the **FileInterceptor()** interceptor to the route handler and extract file from the request using the **@UploadedFile()** decorator.

```
@Post('upload')
@UseInterceptors(FileInterceptor('file'))
uploadFile(@UploadedFile() file: Express.Multer.File)
{
  console.log(file);
}
```

File Validation


```
@Post('upload')
@UseInterceptors(FileInterceptor('file',
{ fileFilter: (req, file, cb) => {
  if (file.originalname.match(/^.*\.(jpg|webp|png|jpeg)$/))
    cb(null, true);
  else {
    cb(new MulterError('LIMIT_UNEXPECTED_FILE', 'image'), false);
  }
},
limits: { fileSize: 30000 },
  storage:diskStorage({
  destination: './uploads',
  filename: function (req, file, cb) {
    cb(null,Date.now()+file.originalname)
  },
})
}))
uploadFile(@UploadedFile() file: Express.Multer.File) {
  console.log(file);
}
```

FileInterceptor

The **FileInterceptor** is configured with the following options:

file: Specifies the field name in the request payload that contains the uploaded file.
fileFilter: A function that checks if the uploaded file meets the specified criteria. In this example, it uses a regular expression to match the file extensions (jpg, webp, png, jpeg).
limits: Sets limits for the uploaded file, such as the maximum file size (600000 bytes in this case).
storage: Specifies the destination directory and the file naming strategy. In this example, it uses diskStorage with the destination set to ./uploads and a custom filename generator function.

Fetching File


```
@Get('/getimage/:name')
    getImages(@Param('name') name, @Res() res) {
      res.sendFile(name,{ root: './uploads' })
    }
```

@Param('name'): The name parameter is obtained from the URL path using the @Param() decorator. It represents the image file name.

@Res() res: the response object using the @Res() decorator. It allows you to manipulate and send the HTTP response.

res.sendFile(name, { root: './uploads' }) statement is used to send the image file as the response.

References

1. **W3Schools Online Web Tutorials, URL: http://www.w3schools.com**
2. **Node.js, URL: https://nodejs.org/en/**
3. **Next.js, URL: https://nextjs.org/**
4. **TypeScript URL: https://www.typescriptlang.org/**
5. **MDN Web Docs URL: https://developer.mozilla.org/**


AMERICAN INTERNATIONAL UNIVERSITY
PRAESIDIUM
DISCIPLINA
CIVITATIS
1994
BANGLADESH

Thank You!