Nest.js Overview

May 2018

Content

- Philosophy
- Features
- Core Concepts
- Module
- Controller
- Component
- Middlewares
- Pipes
- Exception Filters
- Interceptors
- Guards
- Websockets
- Microservices
- Unit Testing

About Project Author

- Kamil Myśliwiec
- kamilmysliwiec.com
- Poland

Introduction

Nest is a powerful web framework for Node.js, which helps you effortlessly build efficient, scalable applications. It uses modern JavaScript, is built with TypeScript and combines best concepts of both OOP (Object Oriented Progamming) and FP (Functional Programming).

Philosophy

The core concept of Nest is to provide an **architecture**.
Which helps developers to accomplish maximum separation of layers and increase abstraction in their applications.

Features

- Based on express.js and socket.io
- Dependency Injection
- Abstract Layers
  - Controllers
  - Exception Layer
  - Guards
  - Interceptors
  - Pipes
- Testing Utils
- Written in TypeScript
- Websockets
- Microservices

Core Concepts

- **Modules** - logical part of the application which lives within the same domain boundary
- **Controlers** - is responsible for handling incoming requests, and return a response to the client
- **Components** - almost everything is a component – service, repository, provider etc.

Setup Application

```
import { Module } from '@nestjs/common';

@Module({})
export class ApplicationModule { }
```

```
import { NestFactory } from '@nestjs/core';
import { ApplicationModule } from './app.module';

const app = await NestFactory.create(ApplicationModule);
app.listen(3000, () => console.log('Application is listening on port 3000'));
```

Controller

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

@Controller()
export class UsersController {

    @Get('users')
    getAllUsers(req, res, next) { }

    @Get('users/:id')
    getUser(req, res, next) { }

    @Post('users')
    addUser(req, res, next) { }
}
```

Endpoints

```
GET: users
GET: users/:id
POST: users
```

Endpoints

```
GET: users
GET: users/:id
POST: users
```

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

@Controller('users')
export class UsersController {

    @Get()
    getAllUsers(req, res, next) { }

    @Get(':id')
    getUser(req, res, next) { }

    @Post()
    addUser(req, res, next) { }
}
```

Adding to Module

```
import { Module } from '@nestjs/common';
import { UsersController } from './users.controller';

@Module({
    controllers: [UsersController],
})
export class ApplicationModule { }
```

Response Object

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

@Controller('users')
export class UsersController {

    @Get()
    getAllUsers(@Res() res) {
        res.json([]);
    }
}
```

Request Object

Nest Decorators	Express Object
`@Request()`	`req`
`@Response()`	`res`
`@Next()`	`next`
`@Session()`	`req.session`
`@Param(param?: string)`	`req.params / req.params[param]`
`@Body(param?: string)`	`req.body / req.body[param]`
`@Query(param?: string)`	`req.query / req.query[param]`
`@Headers(param?: string)`	`req.headers / req.headers[param]`

Custom Decorators

```
import { createRouteParamDecorator } from '@nestjs/common';

export const User = createRouteParamDecorator((data, req) => {
    return req.user;
});
```

Using in Controller

```
@Get()
async findOne(@User() user: UserEntity) {
    console.log(user);
}
```

Component

```
import { Component } from '@nestjs/common';

@Component()
export class UsersService {

    private users = [
        { id: 1, name: 'John Doe' },
        { id: 2, name: 'Alice Caeiro' },
        { id: 3, name: 'Who Knows' },
    ];

    getAllUsers() {
        return Promise.resolve(this.users);
    }

    addUser(user) {
        this.users.push(user);
        return Promise.resolve();
    }
}
```

Let's use it

```
@Controller('users')
export class UsersController {

    constructor(private usersService: UsersService) { }

    @Get()
    async getAllUsers() {
        const result = await this.usersService.getAllUsers();
        return result;
    }

    @Post()
    async addUser(req, res) {
        const result = await this.usersService.addUser(req.body.user);
        res.status(201).json(result);
    }
```

`req.body.user`

We are trying to extract request body (`req.body.user`) without `body-parser`

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import * as bodyParser from 'body-parser';

const express = express();
express.use(bodyParser.json({ strict: false }));

const app = await NestFactory.create(AppModule, express);
await app.listen();
```

Adding to Module

```
import { Module } from '@nestjs/common';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';

@Module({
    controllers: [UsersController],
    components: [UsersService],
})
export class ApplicationModule { }
```

Module

By default, modules **encapsulate** each dependency. It means, that it is not possible to use its components / controllers outside module.

```
import { Module } from '@nestjs/common';
import { UsersController } from './users.controller';
import { UsersService } from './users.service';

@Module({
    controllers: [UsersController],
    components: [UsersService],
    exports: [UsersService],
})
export class UsersModule { }
```

```
@Module({
    imports: [UsersModule],
})
export class ApplicationModule { }
```

Middlewares

Middleware is a function, which is called before route handler.
Middleware functions can perform the following tasks:

- Execute any code
- Make changes to the request and the response objects
- End the request-response cycle
- Call the next middleware function in the stack
- If the current middleware function does not end the request-response cycle, it must call next() to pass control to the next middleware function. Otherwise, the request will be left hanging

LoggerMiddleware

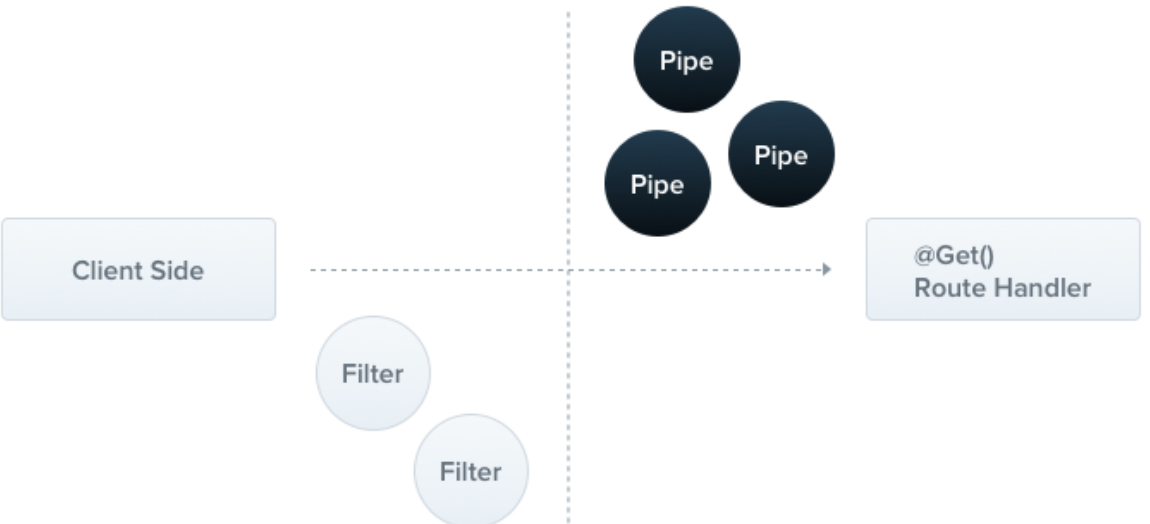
```
import { Middleware, NestMiddleware, ExpressMiddleware } from '@nestjs/common';

@Middleware()
export class LoggerMiddleware implements NestMiddleware {

    resolve(...args: any[]): ExpressMiddleware {
        return (req, res, next) => {
            console.log('Request...');
            next();
        };
    }

}
```

Where to put middlewares?

```
import { Module, NestModule, RequestMethod } from '@nestjs/common';

@Module({
    // ...
})
export class ApplicationModule implements NestModule {

    configure(consumer: MiddlewaresConsumer): void {
        consumer.apply(LoggerMiddleware).forRoutes(
            { path: '/users', method: RequestMethod.GET },
        );
        // Or...
        consumer.apply(LoggerMiddleware).forRoutes(UsersController);
    }

}
```

Pipes

A pipe transforms the input data to the desired output. Also, it could overtake the validation responsibility.



Pipes

```
@Post()
public async addUser(@Res() res: Response, @Body() createUser: CreateUserDto) {
    const result = await this.usersService.addUser(createUser);
    res.status(201).json(result);
}
```

```
class CreateUserDto {
    name: string;
}
```

Pipes

```
import * as Joi from 'joi';

export const userSchema = Joi.object().keys({
    name: Joi.string().alphanum().min(3).max(30).required(),
}).required();
```

Pipes

```
import { PipeTransform, Pipe, HttpStatus } from '@nestjs/common';
import { HttpException } from '@nestjs/common';
import * as Joi from 'joi';

@Pipe()
export class JoiValidatorPipe implements PipeTransform {

    constructor(private schema: Joi.ObjectSchema) { }

    public transform(value, metadata) {
        const { error } = Joi.validate(value, this.schema);
        if (error) {
            throw new HttpException(error, HttpStatus.BAD_REQUEST);
        }
        return value;
    }
}
```

Pipes usage

```
@Post()
@UsePipes(new JoiValidatorPipe(schema))
public async addUser( @Res() res: Response, @Body() user: CreateUserDto) {
    const result = await this.usersService.addUser(user)
    res.status(201).json(result);
}
```

Pipes usage

```
@UsePipes(new ValidationPipe(...))
class UsersController {
    // ...
}
```

```
async addUser(@Body(new ValidationPipe(...)) user: CreateUserDto) {
    this.usersService.addUser(user);
}
```

Exception Filters

In Nest there's an exceptions layer, which responsibility is to catch the unhandled exceptions and return the appropriate response to the end-user.

HttpException

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

@Controller()
export class UsersController {

    // ...

    @Get('users/:id')
    async findById(@Param('id') userId: number): User {
        const user = await this.usersService.findById(userId);
        if (!user) {
            throw new HttpException('User not found', HttpStatus.NOT_FOUND);
        }
        return user;
    }
}
```

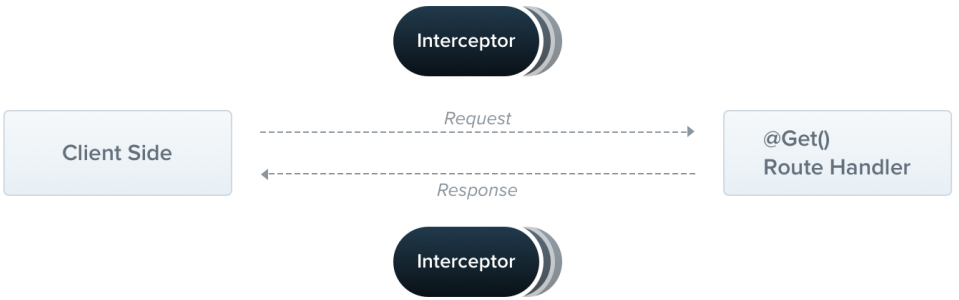
Exception Filter

```
import { ExceptionFilter, Catch, HttpException } from '@nestjs/common';

@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {

    catch(exception: HttpException, response) {
        const status = exception.getStatus();

        response
          .status(status)
          .json({
               statusCode: status,
               message: `It's a message from the exception filter`,
          });
    }
}
```

Filter usage

```
@Controller()
@UseFilters(new HttpExceptionFilter())
export class UsersController {
    // ...
}
```

Interceptors



- bind extra logic before / after method execution
- transform the result returned from the function
- transform the exception thrown from the function
- completely override the function depending on the chosen conditions (e.g. caching purposes)

Interceptor Example

```
import { Interceptor, NestInterceptor, ExecutionContext } from '@nestjs/common'
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/do';

@Interceptor()
export class LoggingInterceptor implements NestInterceptor {
  intercept(dataOrRequest, context: ExecutionContext, stream$: Observable<any>)
    console.log('Before...');
    const now = Date.now();

    return stream$.do(
      () => console.log(`After... ${Date.now() - now}ms`),
    );
  }
}
```

Guards

- Guards have a single responsibility
- They determine whether request should be handled by route handler or not

RolesGuard

```
import { Guard, CanActivate, ExecutionContext } from '@nestjs/common';
import { Observable } from 'rxjs/Observable';

@Guard()
export class RolesGuard implements CanActivate {
  canActivate(dataOrRequest, context: ExecutionContext)
          : boolean | Promise<boolean> | Observable<boolean> {
    // Validate user logic...
    return true;
  }
}
```

Guards Usage

```
@Controller('users')
@UseGuards(RolesGuard)
export class UsersController { }
```

```
const app = await NestFactory.create(ApplicationModule);
app.useGlobalGuards(new RolesGuard());
```

NestJS CLI

`@nestjs/schematics` Nestjs project and architecture element generation based on `@angular-devkit/schematics` engine.

- Application
- Controller
- Exception
- Guard
- Interceptor
- Middleware
- Module
- Pipe
- Service

Socket Gateway

```
import { WebSocketGateway, SubscribeMessage, WsResponse, WebSocketServer, WsExc
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/observable/from';
import 'rxjs/add/operator/map';

@WebSocketGateway()
export class EventsGateway {

  @WebSocketServer() server;

  @SubscribeMessage('events')
  onEvent(sender, data): Observable<WsResponse<number>> {
    const event = 'events';
    const response = [1, 2, 3];

    return Observable.from(response).map(res => ({ event, data: res }));
  }
}
```

Microservices

```
import { Controller, Get, UseInterceptors } from '@nestjs/common';
import { ClientProxy, Client, Transport, MessagePattern } from '@nestjs/microse
import { Observable } from 'rxjs/Observable';

@Controller()
export class MathController {

  @Client({ transport: Transport.TCP, port: 43210 })
  client: ClientProxy;

  @Get()
  call(): Observable<number> {
    const pattern = { cmd: 'sum' };
    const data = [1, 2, 3, 4, 5];
    return this.client.send<number>(pattern, data);
  }

  // In microservice
  @MessagePattern({ cmd: 'sum' })
  sum(data: number[]): number {
    return (data || []).reduce((a, b) => a + b);
  }
}
```

Unit Testing

```
import { Test } from '@nestjs/testing';
import { UserController } from './user.controller';
import { UserService } from './user.service';

const module = await Test.createTestingModule({
    controllers: [UserController],
    components: [
        { provide: UserService, useValue: mockUserService },
    ],
}).compile();

const userController = module.get<UserController>(UserController);
```

End of Overview

- https://nestjs.com/
- https://unlight.github.io/nestjs-talk/presentation.html
- https://videoportal.epam.com/video/PoAXPZa8
- https://kamilmysliwiec.com/nest-release-canditate-is-here-introduction-modern-node-js-framework
- https://github.com/juliandavidmr/awesome-nest
- https://angularcamp.tech/workshops/scalable-nodejs-with-nest/