Nest

Componentes fundamentales de Nest

CertiDevs

Índice de contenidos

1. Módulos

Los **módulos** son una parte fundamental de la arquitectura de NestJS, ya que proporcionan una forma de organizar y encapsular diferentes aspectos de una aplicación en unidades separadas y reutilizables.

Un **módulo** es una **clase** decorada con `@Module()` que agrupa componentes relacionados, como controladores, proveedores, servicios, importaciones y exportaciones.

1.1. Creación de un módulo

Para crear un módulo, crea un nuevo archivo TypeScript y define una clase decorada con `@Module()`.

Por ejemplo, aquí hay un módulo básico llamado `CatModule`:

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Module({
controllers: [CatsController],
providers: [CatsService],
})
export class CatModule {}
```

En este ejemplo, `CatModule` agrupa un controlador `CatsController` y un proveedor `CatsService`.

1.2. Importación y exportación de módulos

Los módulos pueden importar otros módulos para utilizar sus funcionalidades. Para importar un módulo, utiliza la propiedad imports en el decorador @Module() y proporciona un array con los módulos que deseas importar. Por ejemplo, supongamos que necesitas importar el módulo DatabaseModule en CatModule:

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';
import { DatabaseModule } from '../database/database.module';

@Module({
    imports: [DatabaseModule],
    controllers: [CatsController],
    providers: [CatsService],
})
export class CatModule {}
```

Los **módulos** también pueden exportar funcionalidades para que otros módulos las utilicen.

Para hacerlo, utiliza la propiedad `exports` en el decorador `@Module()` y proporciona un array con los componentes que deseas exportar.

Por ejemplo, si deseas exportar `CatsService` desde `CatModule`:

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';
import { DatabaseModule } from '../database/database.module';

@Module({
    imports: [DatabaseModule],
    controllers: [CatsController],
    providers: [CatsService],
    exports: [CatsService],
})
export class CatModule {}
```

1.3. Módulo raíz

Cada aplicación NestJS tiene un **módulo raíz** que es responsable de organizar y coordinar todos los demás módulos en la aplicación.

Por defecto, este módulo se llama `AppModule`.

Cuando se crea una aplicación NestJS utilizando el CLI, se genera automáticamente un `AppModule` básico.

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module({
    imports: [],
    controllers: [AppController],
    providers: [AppService],
})
export class AppModule {}
```

A medida que tu aplicación crezca, puedes importar y organizar diferentes módulos en el módulo raíz para mantener una estructura limpia y modular.

2. Controladores

Los **controladores** son componentes clave en NestJS que se encargan de **manejar** las **solicitudes**

HTTP entrantes y devolver las respuestas a los clientes.

Son **clases** decoradas con el decorador `@Controller()` y contienen métodos que representan acciones o rutas específicas en tu aplicación.

2.1. Creación de un controlador

Para **crear un controlador**, crea un nuevo archivo TypeScript y define una **clase** decorada con `@Controller()`.

El decorador `@Controller()` acepta un argumento opcional que representa el prefijo de ruta para todas las rutas dentro del controlador.

Por ejemplo, aquí hay un **controlador** básico llamado `CatsController`:

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

@Controller('cats')
export class CatsController {

    @Get()
    findAll(): string {
        return 'This action returns all cats';
    }
}
```

En este ejemplo, `CatsController` maneja las solicitudes GET a la ruta `/cats`.

El método `findAll()` se ejecutará cuando se reciba una solicitud a esa ruta y devolverá una respuesta con el contenido "This action returns all cats".

2.2. Manejo de rutas y métodos HTTP

NestJS proporciona **decoradores** para manejar diferentes métodos HTTP, como `@Get()`, `@Post()`, `@Put()`, `@Delete()`, etc.

Estos **decoradores** se pueden aplicar a los **métodos** dentro de un **controlador** para especificar las **rutas** y **acciones** que manejan.

Por ejemplo, aquí hay un controlador que maneja varias rutas y métodos HTTP:

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

@Controller('cats')
export class CatsController {
    @Get()
    findAll(): string {
        return 'This action returns all cats';
```

```
    }

    @Get(':id')
    findOne(@Param('id') id: string): string {
        return `This action returns a cat with ID: ${id}`;
    }

    @Post()
    create(): string {
        return 'This action creates a new cat';
    }

    @Put(':id')
    update(@Param('id') id: string): string {
        return `This action updates a cat with ID: ${id}`;
    }

    @Delete(':id')
    remove(@Param('id') id: string): string {
        return `This action removes a cat with ID: ${id}`;
    }
}
```

2.3. Acceso a datos de la solicitud

Los **controladores** pueden acceder a los **datos** de la **solicitud**, como parámetros de ruta, parámetros de consulta y cuerpo de la solicitud, utilizando decoradores proporcionados por NestJS.

Algunos decoradores comunes incluyen `@Param()`, `@Query()`, `@Body()`, `@Headers()` y `@Req()`.

Por ejemplo:

```
import {
Controller,
Get,
Post,
Body,
Param,
Query,
Headers,
Req,
} from '@nestjs/common';
import { Request } from 'express';

@Controller('cats')
export class CatsController {
// ...

  @Post()
```

```
  create(@Body() createCatDto: CreateCatDto): string {
    // Access the request body
    return `This action creates a new cat with name: ${createCatDto.name}`;
  }

  @Get(':id')
  findOne(
    @Param('id') id: string,
    @Query('version') version: string,
    @Headers('authorization') authHeader: string,
    @Req() request: Request,): string {
// Access the route parameter
// Access the query parameter
// Access the authorization header
// Access the request object
return `This action returns a cat with ID: ${id}, version: ${version}, authHeader:
${authHeader}, userAgent: ${request.headers['user-agent']}`;
}
}
```

2.4. Inyección de dependencias en controladores

Los **controladores** pueden utilizar la inyección de dependencias para acceder a los servicios y otros componentes de la aplicación.

Para **inyectar un componente** en un **controlador**, simplemente inclúyelo como argumento en el constructor de la clase.

Por ejemplo, supongamos que tienes un **servicio** `CatsService` que deseas utilizar en `CatsController`:

```
import { Controller, Get, Post, Body } from '@nestjs/common';
import { CatsService } from './cats.service';
import { CreateCatDto } from './dto/create-cat.dto';
import { Cat } from './interfaces/cat.interface';

@Controller('cats')
export class CatsController {
    constructor(private readonly catsService: CatsService) {}

    @Get()
    findAll(): Cat[] {
        return this.catsService.findAll();
    }

    @Post()
    create(@Body() createCatDto: CreateCatDto): void {
        this.catsService.create(createCatDto);
    }
```

```
}
```

En este ejemplo, `CatsService` se inyecta en `CatsController` y se utiliza para manejar la lógica de negocio relacionada con las operaciones de creación y recuperación de gatos.

En resumen, los **controladores** en NestJS son componentes fundamentales que se encargan de manejar las solicitudes HTTP y devolver respuestas a los clientes.

Permiten organizar la lógica de enrutamiento y manejo de solicitudes en clases modulares y reutilizables, y pueden acceder a otros componentes de la aplicación a través de la inyección de dependencias.

2.5. Manejo de rutas

En NestJS, el **enrutamiento** y **manejo de solicitudes** se realiza mediante **controladores**.

Cada **controlador** puede tener **uno** o **varios manejadores de ruta**, que son funciones encargadas de manejar solicitudes para rutas específicas.

A continuación, se muestra cómo trabajar con enrutamiento y manejo de solicitudes en NestJS:

2.5.1. Crear un controlador:

Usa el decorador `@Controller()` para indicar que una clase es un controlador.

El argumento que se pasa al decorador es el prefijo de ruta para ese controlador.

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

@Controller('example')
export class ExampleController {}
```

En este ejemplo, se crea un controlador llamado `ExampleController` con un prefijo de ruta 'example'.

2.5.2. Agregar manejadores de ruta:

Dentro del **controlador**, puedes agregar **manejadores** de ruta utilizando los decoradores `@Get()`, `@Post()`, `@Put()`, `@Delete()`, `@Patch()`, y `@All()` que corresponden a los métodos HTTP.

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

@Controller('example')
export class ExampleController {
    @Get()
    getAll() {
    // manejar la solicitud GET /example
    }
```

```
    @Post()
    create() {
    // manejar la solicitud POST /example
    }
}
```

En este ejemplo, se agregan dos manejadores de ruta al controlador `ExampleController`: `getAll` maneja las solicitudes `GET` a la ruta `/example`, y `create` maneja las solicitudes `POST` a la misma ruta.

2.5.3. Vincular rutas a manejadores de ruta

Puedes vincular rutas específicas a los manejadores de ruta pasando una cadena de ruta al decorador correspondiente.

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

@Controller('example')
export class ExampleController {
    @Get('all')
    getAll() {
    // manejar la solicitud GET /example/all
    }

    @Post('create')
    create() {
    // manejar la solicitud POST /example/create
    }
}
```

En este ejemplo, se vincula la ruta `/example/all` al manejador de ruta `getAll` y la ruta `/example/create` al manejador de ruta create.

2.5.4. Acceder a los datos de la solicitud:

Para acceder a los datos de la solicitud, como parámetros de ruta, parámetros de consulta y cuerpo de la solicitud, puedes utilizar los decoradores `@Param()`, `@Query()` y `@Body()`.

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

@Controller('example')
export class ExampleController {
    @Get(':id')
    getById(@Param('id') id: string) {
    // manejar la solicitud GET /example/:id
    }

    @Get()
    getAll(@Query('filter') filter: string) {
```

```
    // manejar la solicitud GET /example?filter=...
    }

    @Post()
    create(@Body() data: any) {
    // manejar la solicitud POST /example
    }
}
```

En este ejemplo, se muestra cómo acceder a los **parámetros** de **ruta**, parámetros de consulta y cuerpo de la solicitud en los manejadores de ruta.

2.5.5. Devolver respuestas

Los manejadores de ruta pueden devolver valores que serán enviados como respuesta al cliente. Por defecto, NestJS serializará los objetos y arrays retornados en formato JSON.

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

@Controller('example')
export class ExampleController {
    @Get(':id')
    getById(@Param('id') id: string) {
        return { id, message: 'Item encontrado' };
    }

    @Get()
    getAll(@Query('filter') filter: string) {
        return [
          { id: 1, name: 'Item 1' },
          { id: 2, name: 'Item 2' },
        ];
    }

    @Post()
    create(@Body() data: any) {
        return { message: 'Item creado', data };
    }
}
```

En este ejemplo, los manejadores de ruta retornan objetos y arrays que serán enviados como respuestas JSON al cliente.

2.5.6. Personalizar las respuestas

Si deseas **personalizar la respuesta**, como establecer un código de estado HTTP específico o modificar los encabezados, puedes utilizar el decorador `@Res()` para inyectar la instancia de la respuesta de Express o Fastify.

```
import { Controller, Get, Post, Param, Query, Body, Res } from '@nestjs/common';
import { Response } from 'express';

@Controller('example')
export class ExampleController {
    @Get(':id')
    getById(@Param('id') id: string, @Res() res: Response) {
        res.status(200).json({ id, message: 'Item encontrado' });
    }

    @Get()
    getAll(@Query('filter') filter: string, @Res() res: Response) {
        res.status(200).json([
          { id: 1, name: 'Item 1' },
          { id: 2, name: 'Item 2' },
        ]);
    }

    @Post()
    create(@Body() data: any, @Res() res: Response) {
        res.status(201).json({ message: 'Item creado', data });
    }
}
```

Ten en cuenta que al usar `@Res()`, debes manejar la respuesta manualmente (por ejemplo, llamando a res.json()).

También debes tener en cuenta que al utilizar la instancia de respuesta directamente, tu código podría no ser compatible con diferentes adaptadores HTTP (Express o Fastify).

3. Proveedores y servicios

Los **proveedores** son componentes fundamentales en NestJS que encapsulan y proporcionan funcionalidades o servicios a otras partes de la aplicación.

Los **servicios** son un tipo específico de **proveedor** que se utiliza comúnmente para manejar la lógica de negocio y acceder a datos.

Los **proveedores** y **servicios** se pueden inyectar en otros **componentes**, como controladores y otros proveedores, mediante la inyección de dependencias.

3.1. Creación de un proveedor

Para crear un **proveedor**, define una clase TypeScript y decórala con el decorador @Injectable(). Por ejemplo, aquí hay un proveedor básico llamado CatsService:

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

```
import { Cat } from './interfaces/cat.interface';

@Injectable()
export class CatsService {
private readonly cats: Cat[] = [];

  findAll(): Cat[] {
    return this.cats;
  }

  create(cat: Cat): void {
    this.cats.push(cat);
  }
}
```

En este ejemplo, `CatsService` es un proveedor que maneja la lógica de negocio para crear y recuperar gatos.

Los datos de los gatos se almacenan en un array en la memoria.

3.2. Inyección de dependencias en proveedores

Los **proveedores** también pueden utilizar la inyección de dependencias para acceder a otros componentes de la aplicación. Para inyectar un componente en un proveedor, simplemente inclúyelo como argumento en el constructor de la clase. Por ejemplo, supongamos que tienes un proveedor DatabaseService que deseas utilizar en CatsService:

```
import { Injectable } from '@nestjs/common';
import { Cat } from './interfaces/cat.interface';
import { DatabaseService } from '../database/database.service';

@Injectable()
export class CatsService {
constructor(private readonly databaseService: DatabaseService) {}

  findAll(): Cat[] {
    return this.databaseService.query('SELECT * FROM cats');
  }

  create(cat: Cat): void {
    this.databaseService.query('INSERT INTO cats VALUES ?', [cat]);
  }
}
```

En este ejemplo, `DatabaseService` se inyecta en `CatsService` y se utiliza para interactuar con la base de datos y realizar operaciones de creación y recuperación de gatos.

3.3. Registro de proveedores en módulos

Para utilizar un **proveedor** en tu aplicación, debes **registrarlo** en un **módulo**.

Para hacerlo, añade el proveedor a la propiedad providers en el decorador `@Module()`.

Por ejemplo, para registrar `CatsService` en `CatModule`:

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Module({
controllers: [CatsController],
providers: [CatsService],
})
export class CatModule {}
```

Una vez registrado, el proveedor estará disponible para la inyección de dependencias en otros componentes del módulo.

4. Middleware

Los **middleware** en NestJS son componentes que se ejecutan **antes** de que lleguen las **solicitudes http** a los **controladores**.

Sirven para realizar **acciones** o **manipulaciones adicionales** en las solicitudes o respuestas, como *registro (logging)*, *validación*, *autenticación*, y otras tareas que no están relacionadas directamente con la lógica del negocio.

Los **middleware** son equivalentes a los middleware en Express, ya que NestJS está construido sobre **Express** por defecto (aunque también se puede utilizar con Fastify).

Los **middleware** en NestJS se implementan como **clases** que implementan la **interfaz** `NestMiddleware`.

Esta interfaz requiere que implementes un método llamado `use`, que toma **dos argumentos**: `req` (la solicitud) y `res` (la respuesta), y una función `next` que debe ser llamada para pasar el control al siguiente middleware o controlador.

A continuación, se muestra **cómo crear un middleware** simple de registro (logging) en NestJS:

4.1. Creación de un middleware

Crea un archivo llamado `logger.middleware.ts`:

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

```
@Injectable()
export class LoggerMiddleware implements NestMiddleware {

    use(req: any, res: any, next: () => void) {
        console.log('Solicitud realizada:', req.method, req.originalUrl);
        next();
    }
}
```

En este ejemplo, el **middleware** `LoggerMiddleware` simplemente registra el **método** y la URL de la solicitud antes de pasar el control al siguiente middleware o controlador.

4.2. Uso de un middleware

Aplica el middleware a las rutas en tu aplicación NestJS.

Para aplicar un **middleware**, primero debes **importarlo** en el **módulo** donde deseas utilizarlo.

Luego, implementa el método `configure` en la **clase** del **módulo** y usa el método `use` en la instancia de `MiddlewareConsumer` para aplicar el **middleware** a las rutas deseadas.

Por ejemplo, si deseas aplicar el **middleware** `LoggerMiddleware` a todas las rutas en tu aplicación, actualiza el archivo `app.module.ts` de la siguiente manera:

```
import { Module, MiddlewareConsumer, RequestMethod } from '@nestjs/common';
import { LoggerMiddleware } from './logger.middleware';

@Module({
// ...
})
export class AppModule {

    configure(consumer: MiddlewareConsumer) {
        consumer.apply(LoggerMiddleware).forRoutes({ path: '*', method: RequestMethod
.ALL });
    }
}
```

Aquí, estamos utilizando `forRoutes` para aplicar el middleware a todas las **rutas** (`path: ) y todos los *métodos` HTTP (`method: RequestMethod.ALL`).

También puedes aplicar el middleware a rutas específicas o métodos HTTP.

4.3. Cuándo utilizar middleware:

- Cuando necesites realizar **acciones adicionales** en las solicitudes o respuestas antes de que lleguen a los controladores, como registro (logging), validación, autenticación, modificación de

encabezados, etc.

- Cuando desees implementar **funcionalidades** que **no estén relacionadas directamente** con la lógica del negocio y puedan ser reutilizadas en diferentes partes de tu aplicación.
- Cuando necesites **interceptar** y **modificar solicitudes** y **respuestas** antes de que lleguen a los controladores o sean enviadas al cliente.

5. Guards

Los **guards** en NestJS son clases que implementan la **interfaz** `CanActivate` y permiten proteger rutas o controladores específicos en función de ciertas condiciones, como la autenticación del usuario o la autorización basada en roles.

Para crear un **guard**, define una clase que implemente `CanActivate` y proporciona una función `canActivate()` que devuelve `true` si la solicitud debe **continuar** o `false` si se debe **bloquear**.

Por ejemplo, aquí hay un **guard** básico llamado `AuthGuard`:

```
import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';

@Injectable()
export class AuthGuard implements CanActivate {
    canActivate(context: ExecutionContext): boolean {
        // Implement your authentication logic here
        // If the user is authenticated, return true
        // Otherwise, return false
        return true;
    }
}
```

Para aplicar un **guard**, añádelo como metadato a un controlador o método específico utilizando el decorador `@UseGuards()`.

Por ejemplo, para proteger el controlador `CatsController` con `AuthGuard`:

```
import { Controller, Get, UseGuards } from '@nestjs/common';
import { AuthGuard } from './common/guards/auth.guard';

@Controller('cats')
@UseGuards(AuthGuard)
export class CatsController {
    // ...
}
```

En este ejemplo, todas las rutas gestionadas por `CatsController` están protegidas por `AuthGuard`.

También puedes aplicar guards a métodos específicos en lugar de a todo el controlador.