NestJS

Integrantes:

-Jonathan Ortega
-Mateo Once
-Belén Sarmiento

Asignatura: Programación Web

Docente: Ing. Priscila Cedillo PhD

Periodo: marzo-agosto 2024

Introducción: NestJS

- Nest.js es un framework de Node.js para construir aplicaciones backend.
- Eficiencia, escalabilidad y mantenimiento.
- TypeScript
- Opinionated

Introducción: NestJS

Curva de Aprendizaje

- **Facilidad de Aprendizaje y Dominio:** NestJS es particularmente accesible para desarrolladores que vienen del ecosistema de Angular. La familiaridad con los conceptos y la estructura de Angular permite una adaptación rápida y fluida a NestJS.
- **Curva de Aprendizaje para Desarrolladores No Familiarizados con Angular:** Para los desarrolladores que no tienen experiencia previa con Angular, la curva de aprendizaje puede ser más pronunciada. NestJS se inspira en gran medida en la sintaxis y los conceptos de Angular, lo que puede requerir un período de ajuste y aprendizaje para aquellos que provienen de otros marcos o entornos de desarrollo.

NestJS

Curva de Aprendizaje

- **Documentación y Comunidad:** NestJS cuenta con una documentación extensa y bien organizada, lo que es un recurso valioso tanto para principiantes como para desarrolladores experimentados. La comunidad activa también ofrece numerosos tutoriales, artículos y foros que pueden ayudar a resolver problemas y a aprender las mejores prácticas.
- **Principios y Convenciones del Marco:** NestJS es un marco con opiniones formadas, lo que significa que viene con un conjunto de convenciones y estructuras predeterminadas. Esto puede simplificar el desarrollo al proporcionar un camino claro, aunque podría requerir un ajuste para aquellos acostumbrados a marcos más flexibles.

NestJS

Arquitectura

- Nest.js usa principalmente un arquitectura modular.
- Organiza el código en módulos
- Cada módulo agrupa un conjunto de capacidades relacionadas.

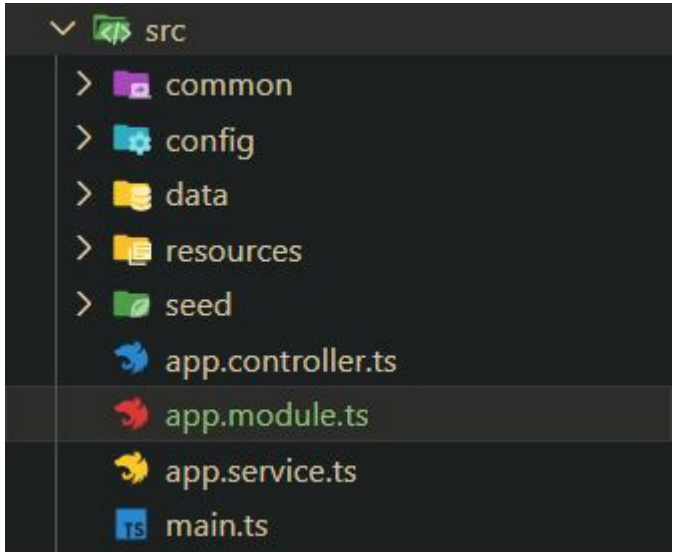
NestJS

Patrones de diseño

- **Patrón de inyección de dependencias**
  - Patrón sumamente utilizado. Permite inyectar dependencias automáticamente en los constructores de las clases.
- **Patrón de decoradores**
  - Los decoradores son parte esencial de Nest. Permiten asociar metadatos con clases y demás elementos.
  - Permite definir rutas, inyección de dependencias, etc.
- **Patrón de fachada (Facade)**
  - Cada módulo de Nest actúa como una fachada →Encapsula y exponen funcionalidades de manera controlada.
- **Patrón repositorio (Repository)**
  - Con el uso de TypeORM, se promueve el uso del patrón repositorio.
  - Capa limpia y desacoplada de acceso a los datos.

NestJS

Archivos del proyecto



app.controller.ts	Las rutas generales de la aplicación
app.module.ts	Módulo raíz del proyecto. Conexión a la base de datos
app.service.ts	Servicio raíz del proyecto
main.ts	Archivo de entrada de la aplicación. Configuración del servidor.

NestJS

Componentes de Nest.js

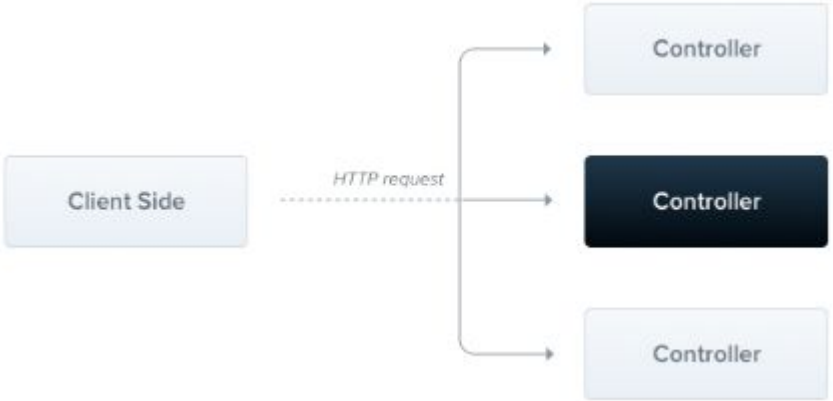
1. Controlador
2. Decorador
3. Módulo
4. Pipe
5. Servicio
6. Resource
7. DTO

NestJS

Controller

Controller

- Responsables de manejar solicitudes entrantes y emitir una respuesta al cliente.



NestJS

Controller

- Las solicitudes que recibe son específicas.
- El sistema de enrutamiento → Controla que controlador recibe qué solicitudes.
- Cada controlador puede tener varias rutas. → Varias acciones.
    - GET, POST, UPDATE, DELETE...
- Un controlador es una clase
- Con decoradores convertimos la clase en un controlador de Nest.

NestJS

Controller: ejemplo

```
import { Controller, Get, Post, Body, Param, Delete, Query, Put } from '@nestjs/common';
import { ApiOkResponse, ApiResponse, ApiTags, ApiNotFoundResponse } from '@nestjs/swagger';
import { Business } from './entities/business.entity';
import { FindBusinessByDto } from './dto'
@Controller('business')
export class BusinessController {
  constructor(
    private readonly businessService: BusinessService) { }

  @Post()
  @ApiResponse({ status: 202, description: 'Registro exitoso' })
  @ApiNotFoundResponse({ description: 'Uno o varios menus no existen' })
  create(@Body() createBusinessDto: CreateBusinessDto) {
    return this.businessService.create(createBusinessDto);
  }

}
```

NestJS

Controller: Creación con Nest CLI

```
nest g co <path/nombre>
```

NestJS

Decorador

Decorador

- Las solicitudes que recibe son específicas.
- Un decorador es una función especial
- Se aplica a clases, métodos, parámetros y propiedades.
- Añade metadatos a estos elementos → Permite a NestJS interpretar y manejar estos elementos de manera estructurada.
- Con esto se indica a NestJS el comportamiento que va a tener cada uno de estos elementos.
- Propios de TypeScript

NestJS

Decorador: ejemplo

```
import { Controller, Get, Post, Body, Param, Delete, Query, Put } from '@nestjs/common';
import { ApiOkResponse, ApiResponse, ApiTags, ApiNotFoundResponse } from '@nestjs/swagger';
import { Business } from './entities/business.entity';
import { FindBusinessByDto } from './dto'
@Controller('business')
export class BusinessController {
  constructor(
    private readonly businessService: BusinessService) { }

  @Post()
  @ApiResponse({ status: 202, description: 'Registro exitoso' })
  @ApiNotFoundResponse({ description: 'Uno o varios menus no existen' })
  create(@Body() createBusinessDto: CreateBusinessDto) {
    return this.businessService.create(createBusinessDto);
  }

}
```

NestJS

Module

Module

- Un módulo en NestJS se define utilizando el decorador @Module(). Este decorador toma un objeto de configuración que describe cómo se estructura el módulo. El objeto de configuración incluye propiedades como controllers, providers, imports y exports.

NestJS

Module: Ejemplo

```
import { Module } from '@nestjs/common';
import { ServeStaticModule } from '@nestjs/serve-static';
import { PokemonModule } from './pokemon/pokemon.module';
import { MongooseModule } from '@nestjs/mongoose';
import { CommonModule } from './common/common.module';
import { SeedModule } from './seed/seed.module'
@Module({
  imports: [
    ServeStaticModule.forRoot({
      rootPath: join(__dirname,'..','public'),
    }),
    MongooseModule.forRoot('mongodb://localhost:27017/nest-pokemon'),
    PokemonModule
    CommonModule
    SeedModule
  ],
  controllers: [],
  providers: [],
})
export class AppModule {}
```

NestJS

Module: Creación con Nest CLI

```
nest g mo <path/nombre>
```

NestJS

Pipes

Pipes

- Son clases que implementan la interfaz PipeTransform y se utilizan para transformar y validar los datos que pasan a través de ellos.
- Se usan para:
- Transformación de Datos: Modificar el formato o tipo de los datos de entrada.
- Validación de Datos: Verificar que los datos cumplan ciertos criterios y, en caso contrario, lanzar una excepción.
- Los pipes interceptan los datos antes de que lleguen a un controlador o a un método de controlador.
- Se aplican globalmente, a un controlador específico, a una ruta específica, o a parámetros.

NestJS

Pipes: Ejemplo

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

@Controller('users')
export class UserController {
  @Get(':id')
  getUserById(@Param('id', ParseIntPipe) id: number) {
    return `User ID is ${id}`;
  }
}
```

Pipe a nivel de parámetro

```
import { Controller, Post, Body, UsePipes, ValidationPipe }
from '@nestjs/common';
import { CreateUserDto } from './create-user.dto';

@Controller('users')
export class UserController {
  @Post()
  @UsePipes(ValidationPipe)
  createUser(@Body() createUserDto: CreateUserDto) {
    return `User created with name ${createUserDto.name}`;
  }
}
```

Pipe a nivel de método

NestJS

Pipes: Creación con Nest CLI

```
nest g pi <path/nombre>
```

NestJS

Services

Services

- Clases que contienen la lógica de negocio y la manipulación de la base de datos.
- Separan la lógica de lo demás mejorando la modularidad y la facilidad de mantenimiento.

NestJS

Services

```
@Injectable()
export class PokemonService {
  constructor(
    @InjectModel(Pokemon.name)
    private readonly pokemonModel: Model<Pokemon>
  ){}

  async create(createPokemonDto: CreatePokemonDto) {
    createPokemonDto.name = createPokemonDto.name.toLocaleLowerCase();
    try {
      const pokemon =  await this.pokemonModel.create(createPokemonDto);
      return pokemon;
    } catch (error) {
      this.handleExceptions(error);
    }

  }
}
```

NestJS

Services: Creación con Nest CLI

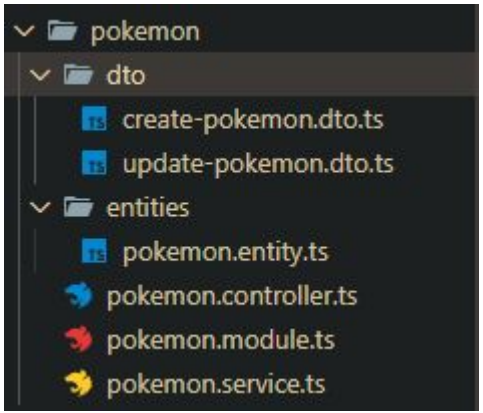
```
nest g s <path/nombre>
```

NestJS

Resources

Resources

- Un recurso es la representación completa de una entidad.
- Incluye todo lo necesario para gestionar una entidad,
  - Controlador
  - Servicio
  - Entidad
  - DTO’s



NestJS

Resources: Creación con Nest CLI

```
nest g resource <path/nombre>
```

NestJS

DTOs

DTOs

- DTO → Data Transfer Object
- Es un objeto que define cómo se transfieren los datos entre las capas de la aplicación.
- Utilizados para validar y tipar la información que se recibe de las solicitudes HTTP antes de que llegue a los servicios y controladores.
- Ayuda a mantener el código limpio y a asegurarse de que los datos que se manejan son correctos y completos.

NestJS

DTOs: Ejemplo

```ts
import { IsInt, IsPositive, IsString, Min, MinLength,
minLength } from "class-validator";

export class CreatePokemonDto {

  @IsInt()
  @IsPositive()
  @Min(1)
  no: number;

  @IsString()
  @MinLength(1)
  name: string;
}
```

```ts
@Post()
create(@Body() createPokemonDto: CreatePokemonDto) {
  return this.pokemonService.create(createPokemonDto);
}
```

NestJS