NEST VARIABLES DE ENTORNO - DEPLOY

- Levanto la db

```
docker-compose up -d
```

- Levanto el servidor

```
npm run start:dev
```

- Para sembrar la base de datos hago una petición GET al endpoint

```
http://localhost:3000/api/v2/seed
```

- Todo listo!

Configuración de variables de entorno

- El string de conexión con la db y el puerto dónde escucha el servidor deben ser variables de entorno
- Creo el archivo .env en la raíz
- Lo añado al .gitignore para no darle seguimiento

```
PORT=3000
MONGODB=mongodb://localhost:27017/nest-pokemon
```

- Node ya tiene sus variables de entorno
- Puedo colocar un console.log en el constructor de AppModule para visualizarlas

```
import { join } from 'path';
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
  ]
```

```
})
export class AppModule {
  constructor(){
    console.log(process.env)
  }
}
```

- Hay un monton! Pero **no aparecen las que yo he creado**
- Para decirle a Nest dónde están las variables de interno hago la **instalación**

> npm i @nestjs/config

- Hay que importar en el app.module el **ConfigModule**
- La posición dónde se coloca es importante. Lo coloco al inicio

```
import { join } from 'path';
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';
import { ConfigModule } from '@nestjs/config';


@Module({
  imports: [
    ConfigModule.forRoot(),
    ServeStaticModule.forRoot({
      rootPath: join(__dirname, '..', 'public')
    }),
    MongooseModule.forRoot('mongodb://localhost:27017/nest-pokemon'),
    PokemonModule,
    CommonModule,
    SeedModule
  ]
})
export class AppModule {
  constructor(){
    console.log(process.env)
  }
}
```

- Ahora puedo observar en el console.log que las variables que yo he creado están disponibles
- Para usar las variables PORT y MONGODB solo tengo que escribir **process.env.PORT**, **process.env.MONGODB** donde corresponde
- **OJO! Que PORT aparece como un string en la variable de entorno**. Todas las variables de entorno **son strings**
    - Debo parsearlo a número con **+**

Configuration Loader

- El ConfigModule ofrece un servicio que permite inyectar las variables de entorno
- Para probarlo escribo una nueva variable de entorno que será el limit de la paginación

```
default_limit=5
```

- Si agrego un console.log(process.env.DEFAULT_LIMIT) en el constructor del PokemonService lo imprime en consola

```
@Injectable()
export class PokemonService {

  constructor(
    @InjectModel(Pokemon.name)
    private readonly pokemonModel: Model<Pokemon>){
      console.log(process.env.DEFAULT_LIMIT)
    }
```

- Si agrego directamente la variable de entorno, podría ser que viniera undefined y esto crasheara mi app
- pokemon.service

```
async findAll(paginationDto: PaginationDto) {

const {limit=+process.env.DEFAULT_LIMIT, offset=0}= paginationDto

return await this.pokemonModel.find()
.limit(limit)
.skip(offset)
.sort({
    no:1
}) ;
}
```

- En este caso funciona porque resulta un NaN y JavaScript lo considera un número, lo que no hace el límite y me muestra todos los pokemons
- Pero no es un comportamiento deseable
- Lo que se enseña a continuación vale para la mayoría de casos
- Más adelante se enseñará como ser más estricto y que devuelva un error si no está bien configurado
- Creo en /src la carpeta config con app.config.ts
- Voy a **exportar una función** que va a **mapear mis variables de entorno**
- Regreso **un objeto** entre paréntesis (return implícito)
- En el caso que no esté **NODE_ENV** (que siempre va a estar) lo voy a colocar como 'dev' (desarrollo)
- Coloco el MONGODB como una propiedad de un objeto por lo que lo coloco en lowerCase
- app.config.ts

```
export const EnvConfiguration = ()=>({
    environment: process.env.NODE_ENV || 'dev',
    mongodb: process.env.MONGODB,
    port: process.env.PORT || 3001,
    defaultLimit: process.env.DEFAULT_LIMIT || 5
})
```

- Tengo que decirle a Nest que va a usar este archivo (carga esto!)
- Para ello voy a ConfigModule en app.module

```
import { join } from 'path';
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';
import { ConfigModule } from '@nestjs/config';
import { EnvConfiguration } from './config/app.config';


@Module({
  imports: [
    ConfigModule.forRoot({
      load: [EnvConfiguration]
    }),
    ServeStaticModule.forRoot({
      rootPath: join(__dirname, '..', 'public')
    }),
    MongooseModule.forRoot(process.env.MONGODB),
    PokemonModule,
    CommonModule,
    SeedModule
  ]
})
export class AppModule {
  constructor(){
    console.log(process.env)
  }
}
```

- No vamos a usar las variables de entorno mediante el process.env
- Las vamos a usar mediante un servicio que proporciona el ConfigModule

---

ConfigurationService

- Inyecto en el constructor del pokemon.service

```
import {ConfigService} from '@nestjs/config'

@Injectable()
export class PokemonService {

  constructor(
    @InjectModel(Pokemon.name)
    private readonly pokemonModel: Model<Pokemon>,
    private readonly configService: ConfigService
    ){}
 }
```

- Esto por si solo da error. Dice que la primera dependencia si puede resolverla, pero no la segunda (en el índice 1)

```
Nest can't resolve dependencies of the PokemonService (PokemonModel, ?). Please
make sure that the argument ConfigService at index [1] is available in the
PokemonModule context.

Potential solutions:
- Is PokemonModule a valid NestJS module?
- If ConfigService is a provider, is it part of the current PokemonModule?
- If ConfigService is exported from a separate @Module, is that module imported
within PokemonModule?
  @Module({
    imports: [ /* the Module containing ConfigService */ ]
  })
```

- El error nos dice que hay que importar el módulo que contenga el ConfigService
- Lo importo de **@nestjs/config**

```
import { Module } from '@nestjs/common';
import { PokemonService } from './pokemon.service';
import { PokemonController } from './pokemon.controller';
import { MongooseModule } from '@nestjs/mongoose';
import { Pokemon, PokemonSchema } from './entities/pokemon.entity';
import {ConfigModule} from '@nestjs/config'

@Module({
  imports:[
    ConfigModule,
    MongooseModule.forFeature([
      {
      name: Pokemon.name,
      schema: PokemonSchema
      }
    ])
  ],
```

```
  controllers: [PokemonController],
  providers: [PokemonService],
  exports:[MongooseModule]
})
export class PokemonModule {}
```

- Para usar las variables de entorno en *PokemonService* uso la inyección de dependencia configService
- Me da dos métodos **get** y **getOrThrow** (si no lo obtiene le decimos que lance un error)
    - Con getOrThrow, si le paso una key que no existe me lanzará un error. Ex: getOrThrow('jwt-seed)

```
async findAll(paginationDto: PaginationDto) {

const {limit=this.configService.get('defaultLimit') , offset=0}= paginationDto

return await this.pokemonModel.find()
.limit(limit)
.skip(offset)
.sort({
    no:1
}) ;
}
```

- Si hago un console.log del this.configService.get('defaultLimit') puedo ver que lo obtengo **como un número**
- configService.get es de tipo genérico, le puedo especificar que es de tipo number y guardarlo en una variable (o directamente en la desestructuración)
    - **OJO que esto no hace ninguna conversión, es solo para decírselo a TypeScript**
- Para que quede más claro, declaro la propiedad como private arriba de todo y la inicializo en el constructor

```
import { BadRequestException, Injectable, InternalServerErrorException,
NotFoundException } from '@nestjs/common';
import {InjectModel} from '@nestjs/mongoose'
import { CreatePokemonDto } from './dto/create-pokemon.dto';
import { UpdatePokemonDto } from './dto/update-pokemon.dto';
import { Model, isValidObjectId } from 'mongoose';
import { Pokemon } from './entities/pokemon.entity';
import { PaginationDto } from 'src/common/dto/pagination.dto';
import {ConfigService} from '@nestjs/config'

@Injectable()
export class PokemonService {

  private defaultLimit: number     //declaro la propiedad

  constructor(
    @InjectModel(Pokemon.name)
    private readonly pokemonModel: Model<Pokemon>,
```

```
    private readonly configService: ConfigService
    ){

      this.defaultLimit= configService.get<number>('defaultLimit') //la inicializo
    }


  async create(createPokemonDto: CreatePokemonDto) {

    createPokemonDto.name = createPokemonDto.name.toLowerCase()

    try {
      const pokemon = await this.pokemonModel.create(createPokemonDto)
      return pokemon;

    } catch (error) {
      this.handleExceptions(error)
    }
  }

  async findAll(paginationDto: PaginationDto) {
    const {limit=this.defaultLimit , offset=0}= paginationDto

    return await this.pokemonModel.find()
    .limit(limit)
    .skip(offset)
    .sort({
      no:1
    }) ;
  }

  async findOne(id: string) {

    let pokemon:Pokemon

    if(!isNaN(+id)){
      pokemon = await this.pokemonModel.findOne({no:id})
    }

    if(!pokemon && isValidObjectId(id)){
      pokemon = await this.pokemonModel.findById(id)
    }

    if(!pokemon){
      pokemon = await this.pokemonModel.findOne({name: id.toLowerCase().trim()})
    }

    if(!pokemon) throw new NotFoundException("Pokemon not found")

    return pokemon
  }

  async update(id: string, updatePokemonDto: UpdatePokemonDto) {
        let pokemon = await this.findOne(id)
```

```
        if(updatePokemonDto.name){
          updatePokemonDto.name = updatePokemonDto.name.toLowerCase()
        }

        try {
          await pokemon.updateOne(updatePokemonDto)
          return {...pokemon.toJSON(), ...updatePokemonDto}

        } catch (error) {
          this.handleExceptions(error)
        }
  }

  async remove(id: string) {

   const {deletedCount} = await this.pokemonModel.deleteOne({_id: id})

   if(deletedCount === 0){
    throw new BadRequestException(`Pokemon with id ${id} not found`)
   }

   return
  }

  private handleExceptions(error: any){
    if(error.code === 11000){
      throw new BadRequestException(`Pokemon exists in db
${JSON.stringify(error.keyValue)}`)
    }
    console.log(error)
    throw new InternalServerErrorException("Can't update Pokemon. Check server
logs")
  }
}
```

- Si la variable de entorno no está definida, **tomará el valor que puse en app.config**
- Si no tengo definida un string de conexión en MONGODB, Nest va a intentar conectarse varias veces sin éxito hasta lanzar un error definitivo
- Deberíamos decirle a la persona que está tratando de levantar la aplicación que tiene que configurar la variable de entorno
- También hay otras formas de establecer valores por defecto
- En el main **no puedo hacer inyección de dependencias** para obtener el PORT, ya que esta fuera del building block
- En este punto puedo usar el **process.env.PORT**
- main.ts

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { ValidationPipe } from '@nestjs/common';
```

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  app.setGlobalPrefix('api/v2')

  app.useGlobalPipes(
    new ValidationPipe({
      whitelist: true,
      forbidNonWhitelisted: true,
      transform: true,
      transformOptions:{
        enableImplicitConversion: true
      }
    })
  )

  await app.listen(process.env.PORT); //uso process.env
}
bootstrap();
```

- Si no tuviera definida la variable PORT daría *undefined*. No sirve la configuración de app.config porque **no está en el building block de Nest**, por lo que no puedo usar el servicio
- En la siguiente lección vamos a establecer unas reglas de validación mediante un Validation Schema, para poder lanzar un error si una de las variables de entorno falla

---

Joi ValidationSchema

- Cuando queremos ser más estrictos y que lance errores en el caso de que un tipo de dato no venga o no sea el esperado podemos usar joi

> npm i joi

- Creo en src/config/joi.validation.ts
- Debo importarlo de esta manera porque solo importando joi no funciona

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

- Básicamente quiero crear un Validation Schema

```
import * as Joi from 'joi'


export const joiValidationSchema = Joi.object({
    MONGODB: Joi.required(),
    PORT: Joi.number().default(3005), //le establezco el puerto 3005 por defecto
    DEFAULT_LIMIT: Joi.number().default(5)
})
```

- **Dónde uso este joiValidationSchema?**
- El ConfigurationLoader puede trabajar de la mano de joi
- En ConfigModule (en app.module)

```
import { join } from 'path';
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';
import { ConfigModule } from '@nestjs/config';
import { EnvConfiguration } from './config/app.config';
import { joiValidationSchema } from './config/joi.validation';


@Module({
  imports: [
    ConfigModule.forRoot({
      load: [EnvConfiguration],
      validationSchema: joiValidationSchema
    }),
    ServeStaticModule.forRoot({
      rootPath: join(__dirname, '..', 'public')
    }),
    MongooseModule.forRoot(process.env.MONGODB),
    PokemonModule,
    CommonModule,
    SeedModule
  ]
})
export class AppModule {}
```

- Pueden trabajar **conjuntamente** el EnvConfiguration con joinValidationSchema
- Prevalece el joinValidationSchema sobre EnvConfiguration. Es decir:
    - Si le doy un valor por default al Schema y pongamos que no viene la variable de entorno, este la setea y para cuando llega a EnvConfiguration, ya esta seteada por el Schema
- Esto hace que trabaje la variable de entorno **como string** (porque está seteada en process.env.VARIABLE) y las variables de entorno **siempre son strings**
- Por ello parseo (por si acaso) la variable

```
export const EnvConfiguration = ()=>({
    environment: process.env.NODE_ENV || 'dev',
    mongodb: process.env.MONGODB,
    port: +process.env.PORT || 3001,
    defaultLimit: +process.env.DEFAULT_LIMIT || 5
})
```

ENV Template README

- Se aconseja que si la app tiene variables de entorno no le dejemos todo el trabajo al nuevo desarrollador o que las adivine
- Debo especificar las variables de entorno necesarias
- Copio el .env a .env.template
- Si hubiera cosas como el SECRET_KEY no lo llenaríamos, solo dejaríamos la definición
- El .env.template si va a estar en el repositorio
- Lo describo en el README. Añado los pasos 5,6,7

````
<p align="center">
  <a href="http://nestjs.com/" target="blank"><img
src="https://nestjs.com/img/logo-small.svg" width="200" alt="Nest Logo" /></a>
</p>

# Ejecutar en desarrollo

1. Clonar el repositorio
2. Ejecutar
```
npm i
```

3. Tener el Nest CLI instalado
```
npm i -g @nestjs/cli
```
4. Levantar la base de datos
```
docker-compose up -d
```
5. Clonar el archivo .env.template y renombrar la copia a .env

6. LLenar las variables de entorno definidas en el .env

7. Ejecutar la app en dev:
```
npm run start:dev
```

8. Reconstruir la base de datos con la semilla
```
http://localhost:3000/api/v2/seed
```

## Stack Usado

- MongoDB
- Nest
````

- **NOTA**: Para desplegar la aplicación crear la base de datos en MongoDBAtlas. Cambiar la cadena de conexión en .env y en TablePlus. Para desplegar ejecutar el script build. Después la app se ejecuta con start:prod. Así lo ejecuta ya como javascript
- Plataformas como Heroku ejecutan **directamente el comando build y luego el start**
- Por ello cambio los archivos start así

```
"start": "node dist/main",
"start:prod": "nest start"
```

- Debo asegurarme de que el puerto sea process.env.PORT para que le asigne el 3000
- Ingreso en Heroku
- Hago click en New (crear nueva app)
- Le pongo el nombre, el País
- La manera más fácil de subir el código es usar Heroku Git
- Para ello debo usar el Heroku CLI (busco la instalación en la web)
- Para saber la versión:

> heroku -v

- me situo en la carpeta de proyecto en la terminal

> heroku login

- Para subir el código escribo esta linea con el nombre que le puse a la app de heroku
- Debo haber hecho git init, con el último commit y estar en la carpeta de proyecto

> heroku git:remote -a pokedex-migue

> git add . git commit -am "commit a heroku" git push heroku main

- A veces da error el package-lock.json (colócalo en el .gitignore!)
- Para ver los logs (y los posibles errores)

> heroku logs --tail

- En Heroku logeado, en el proyecto, Settings, Reveal Config Varsç
- Aquí puedo setear las variables de entorno
- Ahora puedo hacer una petición al endpoint que me dio heroku, por ejemplo un GET

> https://pokedex-migue.herokuapp.com/api/v2/pokemon

- Debo hacer el seed para llenar la db de MongoDBAtlas de data!!!
- Si hago algún cambio, puedo hacer otra vez el proceso de git add .
- Si quiero hacer un redespliegue sin ninguna modificación

> git commit --allow-empty -m "Heroku redeploy" git push heroku main

- Es main o master, se recomienda cambiar la branch master a main