04 NEST MONGODB POKEDEX

- Creo un nuevo proyecto

> nest new pokedex

- En esta sección vamos a trabajar también con **Pipes personalizados** y **Exception Filter**, además de la **conexión con la DB**
- Borro el archivo .spec, el app.service y el app.controller (no los necesito)
- Borro sus referencias en el app.module, dejo el modulo sin dependencias

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


@Module({
  imports: []
})
export class AppModule {}
```

Servir contenido estático

- Normalmente es en la raíz dónde quiero servir un sitio web (una app de React o lo que sea)
- Creo una carpeta en la raíz llamada public
- Dentro creo un index.html y dentro de la carpeta css/styles.css
- Pongo un h1 en el body del html y configuro algunos paddings y font-size en el css
- **Para servir contenido estático** uso **ServeStaticModule**
- Para ello instalo npm i @nestjs/serve-static
- Cuando veas la palabra módulo **siempre va en los imports**
- Las importaciones de Node van al inicio, en este caso uso join del paquete *path*
- app.module.ts

```
import { join } from 'path';
import { Module } from '@nestjs/common';
import {ServeStaticModule} from '@nestjs/serve-static'


@Module({
  imports: [
    ServeStaticModule.forRoot({
      rootPath: join(__dirname, '..', 'public')
    })
  ]
})
export class AppModule {}
```

- En la creación de la API REST nos vamos a basar en la API de Pokemon

Global Prefix

- Creo la API (res de resource)

```
nest g res pokemon --no-spec
```

- Si voy a app.module veo que en imports tengo **PokemonModule**
- Puedo especificar un segmento como prefijo de la url con **setGlobalPrefix** en el main

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

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

  app.setGlobalPrefix('api/v2')

  await app.listen(3000);
}
bootstrap();
```

Docker - Docker Compose - MongoDB

- Para el desarrollo es aconsejable usar docker
- Creo el archivo **docker-compose.yaml** en la raíz del proyecto
- Uso la versión de la imagen de mongo (5)
- Puedo obviar el restart:always para que no incie el contenedor con el SO
- Conecto el puerto 270127 de **mi computadora** con el 27017 **del contenedor** (solo ese puerto está expuesto)
- Configuro unas **variables de entorno** cómo dice la documentación
- Para que la data sea persistente pese a que borre el contenedor uso los **volumenes**
  - Viene a ser como el puerto solo que ahora va a ser **una carpeta del file system**
  - Creo la carpeta mongo y la conecto con el comando *./mongo:/data/db,* donde data/db está en la imagen que estoy montando
- De momento no le coloco password (se hará en el despliegue)

```
version: '3'

services:
  db:
    image: mongo:5
    restart: always
    ports:
```

```
      - 27017:27017
    environment:
    - MONGODB_DATABASE=nest-pokemon
    volumes:
      - ./mongo:/data/db
```

- En VSCode escribo el comando up para levantarlo, -d para que corra desligada de esta instancia de la terminal

> docker-compose up -d

- Si no existe la imagen la descarga
- Esto me crea también la carpeta mongo en la raíz de mi proyecto
- Si miro en docker tengo la imagen de **mongo 5** y en containers tengo **pokedex**
- Puedo borrar la imagen y volver a usar el comando para levantar la DB
- Creo la conexión con este string usando TablePlus

> mongodb://localhost:27017/nest-pokemon

- **NOTA**: Para evitar **problemas** detengo el container con docker-compose down y paro mi servicio de mongo en windows con

> net stop MongoDB

- Hago un test de conexión con TablePlus con el contenedor de docker UP, todo OK

Creo un README.md

- Uso el README de Nest, borro todo menos el log (porque me gusta)

````
<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
````
````

```
```

Stack Usado
- MongoDB
- Nest

Conectar Nest con Mongo

- Instalo mongoose y los conectores de nest

```
npm i @nestjs/mongoose mongoose
```

- En app.module uso **MongooseModel.forRoot**, tengo que especificarle la url de la DB

```
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';


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

- Debo tener docker abierto, subir la DB con docker-compose e iniciar nest con nom run start:dev

---

Crear esquemas y modelos

- Se recomienda que **la entidad sea una clase** para poder definir reglas de negocio
- Cada registro en la DB de mi entidad será una nueva instancia de la clase
- Voy a tener **3 identificadores únicos**:
    - El nombre del pokemon
    - El número del pokemon
    - El mongoID (no lo tengo que especificar porque mongo me lo da automaticamente)
- Mongoose se va a encargar de ponerle la "s" a la clase Pokemon(s)
- Hago que la clase **herede de Document** de mongoose
- Necesito especificarle un decorador para decir que es un esquema
- Le añado unas propiedades con el **decorador Props**
- Lo exporto

```
import {Document} from 'mongoose'
import {Schema, SchemaFactory, Prop} from '@nestjs/mongoose'


@Schema()
export class Pokemon extends Document{

    @Prop({
        unique: true,
        index: true
    })
    name: string

    @Prop({
        unique: true,
        index: true
    })
    no: number
}


export const PokemonSchema = SchemaFactory.createForClass( Pokemon )
```

- Hay que conectar esta entidad con la DB
- **Los módulos siempre van en los imports**
- Voy a usar **el módulo mongoose PERO NO VA A SER forRoot**
- Pokemon.name, el name sale **de la herencia del Document**, no es la propiedad
- También debo indicarle el **schema**
- **NOTA**: el forFeature es **un arreglo** de objetos
- pokemon.module

```
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';


@Module({
  imports:[
    MongooseModule.forFeature([
      {
      name: Pokemon.name,
      schema: PokemonSchema
      }
    ])
  ],
  controllers: [PokemonController],
  providers: [PokemonService]
```

```
})
export class PokemonModule {}
```

- Hay veces que aparece la tabla directamente en TablePlus (en este caso pokemons)
- Si no, aparecerá con la primera inserción. Mientras no haya ningún error está bien
- Se puede relacionar este modelo fuera de este módulo, **se verá más adelante**

POST - Recibir y validar data

- Primero voy al createPokemonDto
- Instalo class-validator y class-transformer

```
npm i class-validator class-transformer
```

- el dto

```
import { IsPositive, IsInt, IsString, Min, MinLength } from "@nestjs/class-
validator"

export class CreatePokemonDto {

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

    @IsString()
    @MinLength(3)
    name: string

}
```

- Para que las validaciones sean efectivas debo hacer la validación global en el main (**Recuerda!! Lo del whitelist!!**)

```
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
```

```
    })
  )

  await app.listen(3000);
}
bootstrap();
```

- Con el PartialType del updatePokemonDto tengo configurado el dto del update, ya que hereda del createPokemonDto y hace las propiedades opcionales
- Debo hacer validaciones en el servicio para no hacer duplicados

---

Crear Pokemon en base de datos

- Vohy a insertar este dto en la base de datos
- Paso el nombre a minúsculas
- Voy a necesitar hacer la **inyección de dependencias** en el constructor de mi **entity** pasándoselo al **Model de mongoose como genérico**
- Si quiero inyectarlo debo ponerle el decorador **@InjectModel() de @nestjs/mongoose** y pasarle el nombre (**Pokemon.name**)

```
import { Injectable } from '@nestjs/common';
import {InjectModel} from '@nestjs/mongoose'
import { CreatePokemonDto } from './dto/create-pokemon.dto';
import { UpdatePokemonDto } from './dto/update-pokemon.dto';
import { Model } from 'mongoose';
import { Pokemon } from './entities/pokemon.entity';

@Injectable()
export class PokemonService {

  constructor(
    @InjectModel(Pokemon.name) //le pongo el decorador y le paso el .name de la
entity
    private readonly pokemonModel: Model<Pokemon>){}//importo Model de mongoose y
le paso como genérico la entity

  create(createPokemonDto: CreatePokemonDto) {

    createPokemonDto.name = createPokemonDto.name.toLowerCase()

    return createPokemonDto;
  }

  findAll() {
    return `This action returns all pokemon`;
  }

  findOne(id: number) {
    return `This action returns a #${id} pokemon`;
```

```
  }

  update(id: number, updatePokemonDto: UpdatePokemonDto) {
    return `This action updates a #${id} pokemon`;
  }

  remove(id: number) {
    return `This action removes a #${id} pokemon`;
  }
}
```

- Vamos a hacer una inserción. Dejemos a un lado las validaciones, solo para crear algo facilmente
- Como las inserciones a las DB son asíncronas coloco el async

```
async create(createPokemonDto: CreatePokemonDto) {

    createPokemonDto.name = createPokemonDto.name.toLowerCase()
    const pokemon = await this.pokemonModel.create(createPokemonDto)
    return pokemon;
  }
```

- Hago una petición con ThunderClient a http://localhost:3000/api/v2/pokemon con un numero y un nombre en el body
- Refresco TablePlus con *Ctrl+R*
- Si intento volver a hacer la misma inserción me va a decir "*Internal Server Error*", por un duplicate key
- Esto vendría a suponer que desde el frontend nos digan que es error del backend, siendo **una validación** que nos hizo falta
- Cuantas **menos consultas** se hagan a la **base de datos mucho mejor**
- Vamos a manejarlo de una mejor manera

Responder un error específico

- Meto la inserción en un **try y un catch** y hago la **validación**
- Sé que si me devuelve el error 11000 es **un error de clave duplicada**. Puedo verlo con **un console.log del error**
- Puedo usar eso para **no hacer otra consulta a la base de datos**, si no tendría que usar una consulta para comprobar el número y otra para el nombre
- Si **no es un error 11000** hago un console.log del error y lanzo (ahora si) un error del server
- **NOTA**: Cuando lanzo un error **con throw new Error no hace falta colocar un return después**

```
async create(createPokemonDto: CreatePokemonDto) {

    createPokemonDto.name = createPokemonDto.name.toLowerCase()

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

```
      return pokemon;

    } catch (error) {
      if(error.code === 11000) throw new BadRequestException(`Pokemon exixts in db ${JSON.stringify(error.keyValue)}`)

      console.log(error)
      throw new InternalServerErrorException("Can't create Pokemon - Check server Logs")
    }
  }
```

- Cuando se trata de un nombre o número duplicado (error 11000), ahora puedo ver en el message que el error es del nombre duplicado o del número
- Para borrar en TablePlus seleccionar la fila, apretar **Supr** y luego **Ctrl+S** para aplicar el commit
- Ctrl+R para recargar
- Si quisiera poner otro código en lugar de un 201, poner un 200, hago uso del decorador **@HttpCode()** en el controlador
- pokemon.controller

```
@Controller('pokemon')
export class PokemonController {
  constructor(private readonly pokemonService: PokemonService) {}

  @Post()
  @HttpCode(200)
  create(@Body() createPokemonDto: CreatePokemonDto) {
    return this.pokemonService.create(createPokemonDto);
  }
///...rest of code...
```

- También tengo HttpStatus que me ofrecen los códigos de error con autocompletado

```
  @Post()
  @HttpCode(HttpStatus.OK) //HttpStatus.OK === 200
  create(@Body() createPokemonDto: CreatePokemonDto) {
    return this.pokemonService.create(createPokemonDto);
  }
```

---

findOne - Buscar

- Tenemos 3 identificadores: el nombre, el número y el id de mongo
- Si se le pasa un id y no es un mongoId va a dar un error de base de datos (hay que hacer esa validación)
- Si el nombre no existe también devuelve error

- Por id siempre vamos a recibir un string, ya que me puede enviar mongoId, el nombre y si me da un número lo recibo como string
- Le **quito el + del +id** del controlador findOne para no parsear el id a número
- Lo mismo en el servicio,**tipo el id a string**
- En el servicio, declaro la variable pokemon **de tipo Pokemon** (**entity**)
- Uso la negación con NaN para decir : si es un número
- Es un método async porque voy a consultar la DB
- pokemon.service

```
async findOne(id: string) {

  let pokemon:Pokemon

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

  return pokemon
}
```

- Si le paso un número que no existe me devuelve un status 200 aunque no encontró nada. Obviamente no queremos eso

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

- Para el mongoId tengo que balidar que sea un mongoId válido
- Para ello tengo **isValidObjectId de mongoose**
- Agrego la condición para que lo busque si no tengo un pokemon por id
- Si no lo encuentra por id voy a intentar encontrarlo por el nombre. Uso **.trim** para eliminar posibles espacios en blanco
- Y si no lo encuentra lanza el error
- Luego se optimizará este código
- pokemon.service

```
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
  }
```

Actualizar Pokemon

- Si miramos en la entity vemos que el nombre está indexado y el número está indexado, por lo que es igaul de rápido que lo busquemos por nombre, por número...
- Hago **los mismos retoques con el id** para tiparlo como string
- El updatePokemonDto está heredando las propiedades (ahora opcionales) de createPokemonDto gracias a **PartialType**
- Uso el método **findOne** creado antes para encontrar el objeto pokemon (de mongo) y verificar el id
- Si viene el nombre lo paso a **lowerCase**
- Hago el update. Este **pokemon es un objeto de mongo**, por lo que tiene todos los métodos y propiedades.
- Le pongo new en true para que me devuelva el objeto actualizado, pero debo guardarlo en una variable para retornarlo
- **Pero aunque lo haga de esta manera no me devuelve el objeto actualizado si no el retorno del update como operación**

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

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

        const updatedPokemon = await pokemon.updateOne(updatePokemonDto, {new:
true}) //aún así no me devuelve el objeto actualizado

        return updatedPokemon
  }
```

- Esparzo todas las propiedades que tiene el pokemon con el **spread.toJSON** y las sobreescribo con el spread de updatedPokemonDto

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

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

```
        }

        await pokemon.updateOne(updatePokemonDto)

        return {...pokemon.toJSON(), ...updatePokemonDto}
}
```

- Ahora **hay un problema**
- Si intento actualizar el número de un pokemon que ya existe (el 1, por ejemplo, y es bulbasur) con otro nombre me devuelve **error 11000** (de valor duplicado)

Validar valores únicos

- Meto la actualización en un try catch
- En caso de que el error sea 11000 lanzo un *BadRequestException*
- Si no imprimo el error en un console.log para debuggear y lanzo un InternalError

```
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) {
          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")
        }
}
```

- Voy a crear un método en el servicio para manejar los errores
- Voy a tipar el error como any para dejarlo abierto

```
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")
  }
```

- Sustituyo el código de error por el método
- Siempre regresa un error (siempre hay que hacer un throw)

```
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';

@Injectable()
export class PokemonService {

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

  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() {
    return this.pokemonModel.find() ;
  }

  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)
        }
  }

  remove(id: number) {
    return `This action removes a #${id} pokemon`;
  }

  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")
  }
}
```

Eliminar un pokemon

- Hago los retoques del tipado del id como string

```
async remove(id: string) {
  const pokemon = await this.findOne(id)
  await pokemon.deleteOne()
}
```

- Pero yo quiero implementar la lógica que **para borrar el pokemon se tenga que usar el id de mongo**
- Para ello creo un customPipe

CustomPipe - parseMongoIdPipe

- Quiero asegurarme de que el parámetro que le paso a la url sea un mongoId
- Hay una **estructura de módulo recomendada**
    - src
        - common
            - decorators
            - dtos
            - filter
            - guards
            - interceptors
            - middlewares
            - pipes
        - common.controller.ts
        - common.module.ts
        - common.service.ts
- Voy a usar el **CLI** para generar un nuevo módulo llamado common
- Por defecto viene sin servicio ni nada más que el .module

> nest g mo common

- Para crear la carpeta pipes tambien uso el CLI, le digo la carpeta dónde lo quiero (crea la carpeta pipes) y el nombre del pipe
- No le coloco Pipe al final porque lo crea nest directamente

> nest g pi common/pipes/parseMongoId

- Me crea esto.
- Implementa la interfaz de PipeTransform
- Tipo el value a string
- Hago un console.log del value y de la data

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

@Injectable()
export class ParseMongoIdPipe implements PipeTransform {
  transform(value: string, metadata: ArgumentMetadata) {

    console.log({value, metadata})
  }
}
```

- Para poder obervar el *console.log* del **ParseMongoIdPipe** uso el Pipe en el controlador del delete (borro el código del servicio y dejo solo un *console.log* del id)

- pokemon.service

```
async remove(id: string) {
  //const pokemon = await this.findOne(id)
  //await pokemon.deleteOne()

  console.log({id})
}
```

- En el controlador

```
@Delete(':id')
remove(@Param('id', ParseMongoIdPipe) id: string) {
  return this.pokemonService.remove(id);
}
```

- Hago una llamada al endpoint desde ThunderClient para observar el *console.log* del value y la metadata
- Como id en la url le paso un 1. Me devuelve esto por consola

```
{
  value: '1',
  metadata: { metatype: [Function: String], type: 'param', data: 'id' }
}
```

- Los Pipes transforman la data
- Si coloco en el *return* value.toUpperCase() me devuelve el id en mayúsculas (al poner un string)
- En consola me devolverá en minúsculas porque en el momento de hacer el *console.log* no le he aplicado el toUpperCase

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

@Injectable()
export class ParseMongoIdPipe implements PipeTransform {
  transform(value: string, metadata: ArgumentMetadata) {
    console.log({value, metadata})  //bulbasur    (pasado como id)

    return value.toUpperCase();     //BULBASUR
  }
}
```

- Puedo usar la metadata para hacer validaciones
- Uso el *isValidObjectId* de mongoose para hacer la validación
- Si pasa la validación retorno el value (que es el mongoId que he pasado por parámetro)

```
import { ArgumentMetadata, Injectable, PipeTransform, BadRequestException } from
'@nestjs/common';
import { isValidObjectId } from 'mongoose';

@Injectable()
export class ParseMongoIdPipe implements PipeTransform {
  transform(value: string, metadata: ArgumentMetadata) {

    if(!isValidObjectId(value))
        throw new BadRequestException(`{value} is not a valid MongoId`)

    return value;
  }
}
```

- En el servicio escribo la lógica de negocio

```
async remove(id: string) {

  const pokemon=  await this.pokemonModel.findByIdAndDelete(id)
    return pokemon
  }
```

- De esta manera obtengo el pokemon eliminado. Pero **hay un problema** al hacerlo así
- Si desde el frontend me envían un mongoId válido pero que no existe, el status que me devuelve el remove es un *200*
- Pero la verdad es que no encontró el pokemon
- Quiero **evitar hacer otra consulta a la db**

Validar y eliminar en una sola consulta

```
async remove(id: string) {

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

   return result
  }
```

- Esto me devuelve un valor **"deletedCount"** en 0 si no ha borrado ningún registro, y el acknowledged (boolean) si realizó el procedimiento
- Puedo **desestructurarlo** de la llamada a la db

```
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
  }
```

- Podríamos envolver esta llamada de eliminación en un try catch
- Más adelante se creará un *ExceptionFilter* para filtrar todos los endpoints
- Falta el *findAll*, donde haremos paginación y búsqueda mediante expresiones regulares
- También haremos el SEED, sacaremos la data de PokeAPI

> https://pokeapi.co/api/v2/pokemon?limit=500

- Le establezco un límite a la data de 500 pokemon