Contenido

[Video 1 1](#_Toc165936824)

[Fase 1 (Configuración del proyecto y conexión a bdd) 1](#_Toc165936825)

[Primeros pasos 1](#_Toc165936826)

[Configurar el inter 3](#_Toc165936827)

[Configurar base de datos (mongo) 8](#_Toc165936828)

[Fase 2 (Creación de los modelos y servicios) 9](#_Toc165936829)

[Tipos 9](#_Toc165936830)

[Modelos 10](#_Toc165936831)

[Servicios 11](#_Toc165936832)

[Fase 3 (Manejo de errores usando la librería BOOM) 15](#_Toc165936833)

[Fase 4 (Manejo de rutas) 22](#_Toc165936834)

[Pasos extra (min 2:07:00): 25](#_Toc165936835)

[Fase 5 (archivo ENV y libreria) 29](#_Toc165936836)

[Video 2 32](#_Toc165936837)

[Fase 1 (Autenticación por medio de json web token) 32](#_Toc165936838)

[Utilidades 32](#_Toc165936839)

[Tipo User versión 1: 32](#_Toc165936840)

[Modelo User versión 1: 33](#_Toc165936841)

[Servicio User versión 1: 34](#_Toc165936842)

[Router User versión 1: 36](#_Toc165936843)

[Video 3 39](#_Toc165936844)

[Encriptación de contraseña 39](#_Toc165936845)

[Video 4 45](#_Toc165936846)

[Apuntes Primer video 53](#_Toc165936847)

# Video 1

## Fase 1 (Configuración del proyecto y conexión a bdd)

### Primeros pasos

Se seguio el manual de digitalOcean:

[How To Set Up a Node Project With Typescript | DigitalOcean](https://www.digitalocean.com/community/tutorials/setting-up-a-node-project-with-typescript)

<https://www.digitalocean.com/community/tutorials/setting-up-a-node-project-with-typescript>

Para inicializar el proyecto se necesita del siguiente comando:  
  
npm init -y

Esto creara el package json

Texto

Descripción generada automáticamente

Después se tiene que configurar el compilador de typescrit, para ello se necesita de la dependencia de typescrit

npm i -D typescript

Crear un archivo llamado “tsconfig.json” y dentro de dicho archivo:

{

    "compilerOptions": {

        "module": "CommonJS", //Puede ser también ES6

        "esModuleInterop": true, //Este es para que pueda trabajar con varias versiones

        "target": "ES6",

        "moduleResolution": "node", //Resoluciones

        "sourceMap": true, //Proceso que ayudara a ver nuestro código, al momento de transpilar de typescript a javascript

        "outDir": "dist", //es para ver donde se guardará el proyecto

    },

    "lib": [

        "es2015"

    ],

}

------investigar las configuraciones realizadas, sobre todo sourceMap ---------

Después se debe instalar express:

npm i express

existen librerías que no tienen los tipos en su librería como tal y por ello se deben instalar tambien:

npm i -D @types/librería 🡪 npm i -D @types/express

Se debe crear una carpeta llamada “src” y dentro de dicha carpeta ya se debe trabajar el proyecto:

Se crea un archivo principal “app.ts” y dentro del mismo una aplicación sencilla:

import express from 'express';

//creo la aplicacion de express

const app = express();

const port = 3000;

app.get('/', (req, res) => {

    res.send('hello word');

});

app.listen(port, () => {

    console.log(`Server is running at http://localhost:${port}`);

});

------Investigar que son las dependencias de desarrollo y de producción-----

NOTAS

diferencia entre dependencias de desarrollo y dependencias que solo están marcadas como dependencia:

las de dependencia no se cargan en el proyecto cuando están en producción

Para saber que está corriendo:

npx tsc

el comando creará una carpeta llamada “dist” y dentro contendrá el código que realice, pero, más optimizado, además del archivo “.map” que se crea gracias a la configuración hecha: "sourceMap": true, dentro “de tsconfig.json”

después se inserta el siguiente comando:

node dist/app.js

dicho comando es el ejecutara la aplicación creada en el servidor

### Configurar el inter

Primeramente, se deben instalar unas dependencias de desarrollo:

npm install -D eslint@8

🡪 esta versión es compatible **actualmente** con typescript (¡¡verificar si no hay incompatibilidades!!)

Verificar si typescript es compatible con la versión más reciente, si lo es, ejecutar lo siguiente:

npm i -D eslint

Después daré init para que me pregunte **¿cómo quiero mi código?**, para ello se da lo siguiente:

npx eslint --init

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

javascript porque se esta utilizando import y export

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

En javascript porque permite configurar ciertas cosas, además para respetar la guía que ofrece digitalOcean

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Sin puntos y comas

Texto

Descripción generada automáticamente

Instalar las dependencias que faltan con el gestor de paquetes que se está utilizando “npm”.

Cuando se termine de instalar eslint, aparecerá un nuevo archivo, **.eslintrc.js**, dicho archivo exporta un objeto con todas las configuraciones hechas anteriormente.

Después se debe instalar la dependencia de nodemon (esta servirá para correr el servidor):

npm i -D nodemon

es posible que el sistema no reconozca ts-node, por lo que será necesario instalarlo:

npm install --save-dev ts-node

para correr la aplicación:

npx nodemon ./src/app.ts

Ahora bien en el archivo package.json insertaré los scripts necesarios para poder compilar y poder guardar cambios:

**ANTES**

  "main": "index.js",

  "scripts": {

    "test": "echo \"Error: no test specified\" && exit 1"

  },

**DESPUES**

  "main": "dist/app.js",

  "scripts": {

    "start": "tsc && node dist/app.js",

    "lint": "eslint . --ext .ts",

    "dev": "nodemon src/app.ts",

    "test": "echo \"Error: no test specified\" && exit 1"

  },

Con la configuración anterior se podrá ejecutar el siguiente comando que permite correr el servidor y a su vez refrescar los cambios de manera automática:

npm run dev

Ahora es necesario utilizar un estilo de código, para eso se necesita instalar prettier

npm i -D prettier

después crearé un archivo con la configuración del estilo, llamado “**.prettierrc**” y su contendo:

{

  "semi": false,

  "tabWidth": 2,

  "printWidth": 80,

  "singleQuote": true,

  "trailingComma": "none",

  "useTabs": false

}

Ahora se deben instalar los plugins:

npm i -D eslint-config-prettier

npm i -D eslint-plugin-prettier

Luego, configura ESLint para ello en el archivo “**.eslintrc.js”** en la sección “extends” y “plugins” se agrega priettier, de esta manera:

    "extends": [

        "eslint:recommended",

        "plugin:prettier/recommended",

        "plugin:@typescript-eslint/recommended"

    ],

    "plugins": [

        "@typescript-eslint",

        "prettier"

    ],

Al tener toda la configuración realizada el código ahora responderá a un solo estilo, así que al recargar cada línea de código **se debe ajustar al estilo prestablecido**

Para poder hacer esto posible, descargue la extensión de visual studio **Prettier - Code formatter** y después vi el siguiente video:

[How to use Prettier in VS Code - Code Formatting (youtube.com)](https://www.youtube.com/watch?v=__eiQumLOEo)

<https://www.youtube.com/watch?v=__eiQumLOEo>

### Configurar base de datos (mongo)

Primeramente, se necesita instalar Docker en el equipo

<https://www.docker.com/>

después se creará un archivo del tipo yamel-file, el cual se llamará “docker-compose.yml” y dentro de ella:

version: "3"

services:

  mongo:

    image: mongo:latest

    ports:

      - "27017:27017"

    volumes:

      - /mongo-bd-web-2/

Después se ejecuta el siguiente comando, que es usado para levantar contenedores Docker utilizando un archivo **docker-compose.yml**

docker-compose up -d

este comando hará un detached, Esto significa que los contenedores continúan ejecutándose incluso después de que la terminal desde la que se lanzó el comando se cierre o se desconecte

Información extra:

docker-compose up

leerá de docker compose y va a ejecutar lo que contenga

Ahora instalaré la api que permite conectar mongo con node.js

npm i mongoose

y en las dependencias los tipos:

npm i -D @types/mongoose

ahora en el código de app.ts, importaré mongoose y haré la conexión a la base de datos:

import mongoose from 'mongoose'

// const MONGO\_URI = 'mongodb://mongo:27017/pokedex' código original

const MONGO\_URI = 'mongodb://localhost:27017/pokedex' //pokedex será el nombre de la bdd

const connectDB = () => {

  mongoose

    .connect(MONGO\_URI)

    .then(() => {

      console.log('Connected to Database')

    })

    .catch((error) => {

      console.log('Could not connect to Database', error)

    })

}

app.listen(port, () => {

  console.log(`Server is running at http://localhost:${port}`)

  connectDB() //conecto la base de datos pokedex

})

## Fase 2 (Creación de los modelos y servicios)

### Tipos

En typescript a veces es necesario crear los tipos “type.ts”, para eso es necesario crear una carpeta con el nombre “types” dentro del directorio “src” y dentro de types, la estructura de sus archivos será:

**nombre.type.ts**

**Ejemplo:**

**pokemon.type.ts**

import type { Model } from 'mongoose'

// Existen dos "tipos" de datos (los cuales exporto)

// Datos o propiedades de un Pokemon

export type Pokemon = {

  id?: string //el signo de interrogación hace que el atributo id no sea obligatorio, sino opcional

  name: string

  type: string

  weight: string

  height: string

}

// Tambien existe el modelo del pokemon, (el modelo tendra más cosas,

// tendrá todo lo que mongo le asigne, como metodos para agregar

// información, consultar colecciónes, etc)

export type PokemonModel = Model<Pokemon>

### Modelos

Para continuar ahora me encargare de establecer los modelos como parte de la actividad:

Dentro del directorio “src” crearé otro directorio llamado “models”

Los modelos tendrán la siguiente estructura, en el nombre:

**nombre.model.ts**

**Ejemplo:**

Dentro de cada modelo lo primero que se tiene que hacer es importar el modelo y esquema de moongose “se crea la tabla o la colección”:

// importar el modelo y esquema de moongose “se crea la tabla o la colección”

import { Schema, model } from 'mongoose'

Después importo los tipos creados con anterioridad

// importo los tipos creados para este modelo

import { Pokemon, PokemonModel } from '../types/pokemon.type'

// nombre con el que voy a referir en el servicio

export const POKEMON\_REFERENCE = 'Pokemon'

//Para crear un modelo se pone en plural(Pokemons), por ser la colección de pokemons

// será igual a un nuevo esquema, y será del tipo Pokemon y el modelo PokemonModel.

// (dentro de "<" ">" se debe poner el tipo que se va a implementar y el modelo)

// Esto esta creando un nuevo objeto y la clase(Creó que la clase Schema), en su constructor

// recibe un objeto, donde tiene toda la definción de nuestro pokemon: name, type, weigth, heigth

// (id lo pone mongo)

const Pokemons = new Schema<Pokemon, PokemonModel>({

  //id no se pone porque eso lo pone moongo

  //Define la propiedad de la colección Pokemon

  name: {

    type: String, //tipo

    required: true, //es requerido

    unique: true, //es unica

    index: true, //al ser unica debe estar indexada

    trim: true //le quita los espacios al principio y al final

  },

  type: {

    type: String,

    required: true,

    index: true,

    trim: true

  },

  weight: {

    type: String,

    required: true,

    index: true,

    trim: true

  },

  height: {

    type: String,

    required: true,

    index: true,

    trim: true

  }

})

// Después de tener el squema hecho lo exporto, se crea un modelo a partir de dicho esquema

export default model(POKEMON\_REFERENCE, Pokemons)

// 'POKEMON\_REFERENCE' es el nombre al que me referiré a la colección en los servicios

// 'Pokemons' es el esquema que creé

### Servicios

Al tener hecho lo anterior, ya es posible hacer operaciones sobre la tabla (modelo) Pokemons

Para hacer operaciones, podría crear un servicio, para ello tendré que crear una carpeta de servicios es allí donde realizare las operaciones correspondientes, dentro del directorio “src” el cual he estado trabajando los modelos y tipos, crearé ahora el directorio “servicios” y dentro el archivo que manejara la lógica del servicio, a partir de ahora todos los servicios tendrán esta nomenclatura:

**nombre.service.ts**

**Ejemplo:**

// pokemon.service.ts

//Se necesita importar los Pokemons del modelo

import Pokemons from '../models/pokemon.model'

//Se necesita importar el tipo

import { Pokemon } from '../types/pokemon.type'

//Creo una clase llamada PokemonService, en ella agrego los

//metodos que utilizare para conectarme a la base de datos

class PokemonService {

  //INSERTAR

  //esta funcion recibira un objeto de tipo pokemon

  async create(pokemon: Pokemon) {

    //Ahora llamo al modelo

    //el AWAIT es para esperar a que termine ya que es una promesa

    //mando el pokemon que quiero crear

    //se pone un catch para mostrar el error

    const newPokemon = await Pokemons.create(pokemon).catch((error) => {

      console.log('Error while connecting to the DB', error)

    })

    //envio error en caso de no crear el pokemon

    if (!newPokemon) {

      throw boom.badRequest('Could not create pokemon')

    }

    return newPokemon

  } // En otras palabras....

  // la funcion create(pokemon: Pokemon) dice que al modelo de Pokemons agrega

  // un pokemon nuevo y si ocurre un error lo logeo o muestro con console.log

  //Metodo para que encuentre todo

  async findAll() {

    const pokemons = await Pokemons.find().catch((error) => {

      console.log('Could not find pokemons', error)

    })

    return pokemons

  }

} // Este metodo se encarga de encontrar todos los pokemons

//Exportar la clase para poder acceder desde fuera

export default PokemonService

Ahora, de momento para ingresar y recibir, en el archivo “app.ts”, puedo crear otro método post(ingresar), get(recibir).

Importacion para el metodo post:

//importacion para el metodo post

import { Pokemon } from './types/pokemon.type'

Después importo el servicio:

//importo el servicio

import PokemonService from './services/pokemon.service'

Invoco el middleware, **express.json()** que se encargará de parsear ese JSON, y lo convierte en un objeto JavaScript

//hacer que la app pueda recibir json con el middleware

app.use(express.json())

Para la ruta POST:

// Ruta Post guarda un pokemon en la base de datos

app.post('/pokemon', async (req, res) => {

  //para hacer la prueba introducire informacion crasheada

  //creo un pokemon, del tipo Pokemon, se importa el tipo

  const pokemon: Pokemon = {

    //Inserto datos del pokemon ejemplo

    name: 'Pikachu',

    type: 'Electrico',

    weight: '120kg',

    height: '.8m'

  }

  //se crea al pokemon, y se guarda en la instancia newPokemon

  const newPokemon = await pokemonService.create(pokemon)

  //Envió la respuesta al cliente con ‘res’ y envió una respuesta del objeto newPokemon en formato JSON con código de estado 201

  res.status(201).json(newPokemon)

})

Para la ruta GET:

// Ruta GET consulta pokemon

app.get('/pokemon', async (req, res) => {

  //busca todos lo pokemons

  const pokemons = await pokemonService.findAll()

  //200 por que busca si existen pokemons en el modelo

  res.status(200).json(pokemons)

})

Lo anterior es un ejemplo, ya que se están insertando datos “hardcodeados”, esto es dentro del código, exactamente esto sucede en el método POST.

Para hacer que estos provengan de manera dinámica como solicitud POST por parte del cliente, utilizare ‘req’, utilizando **req.body** para obtener los datos enviados en el cuerpo de la solicitud.

Por lo que en lugar de:

  const pokemon: Pokemon = {

    name: 'Pikachu',

    type: 'Electrico',

    weight: '120kg',

    height: '.8m'

  }

Haré esto:

const pokemon: Pokemon = req.body

Usando postman:

POST  
Captura de pantalla de computadora

Descripción generada automáticamente

GET

Captura de pantalla de computadora

Descripción generada automáticamente

## Fase 3 (Manejo de errores usando la librería BOOM)

Ahora bien, usare la librería boom para que me permita manejar los errores de **status** (200, 201, 404)una manera más elegante:

Para instalar la librería de boom

**npm i @hapi/boom**

a continuación, aplicaré los middlewares (funciones que se ejecutan cuando termina otra función)

ejemplo, cuando establezco una ruta como:

app.get('/',

tengo cierta función:

//Función

(req, res) => {

  res.send('Hola saludame, :)')

}

Pero dentro de la misma ruta creada puedo implementar otra función que puede ser “next”, permite ejecutar una función después de terminar con una anterior (se puede utilizar para muchas cosas diferentes)

)

Para trabajar con ellos, creé otro directorio dentro del directorio “src”, con nombre “middlewares” y dentro de la misma el archivo “error.handler.ts”

El concepto de los middlewares, estos son funciones que se ejecutan después de otra función. Para implementarla en el proyecto, dentro del directorio “src” se crea otro directorio “middleware” y dentro de ella el archivo error.handler.ts:

// se recibe el error, pero como es parte de la ejecución de rutas tambien se

// recibe el req y el res pero no se utilizaran, para no utilizar de manera

// obligatoria una variable definida en los parametros JavaScript y en TypeScript,

// se puede poner el prefijo '\_' de esta manera ya no marcara el error de que no

// se esta utlizando dicha variable

// este middleware simplemente registra el error en la consola y luego pasa el error al siguiente middleware con next(error).

const logErrors = (error, \_req, \_res, next) => {

  console.log(error) // loguear el error

  next(error) // despues manda a llamar lo que sea

} // meddleware

// Este es otro middleware de error que maneja los errores que no son de Boom

// (Boom es una librería para manejar errores de manera consistente).

// Si el error no es de Boom, devuelve un código de estado 500 (Internal Server Error)

// junto con el mensaje de error y la pila de llamadas (stack) en formato JSON.

const errorHandler = (error, \_req, res, \_next) => {

  if (!error.isBoom) {

    // internal server error

    res.status(500).json({

      message: error.message, // mostrar el error

      stack: error.stack // stack que mande el error

    })

  }

}

// Handler de BOOM

// Este middleware de error se ocupa de los errores que son de tipo Boom. Si el error es de Boom,

// extrae el objeto de salida (output) del error, que contiene el código de estado HTTP y el mensaje

// de error, y lo devuelve como respuesta en formato JSON.

const boomErrorHandler = (error, \_req, res, next) => {

  if (error.isBoom) {

    //Objeto output

    const { output } = error

    res.status(output.statusCode).json(output.payload)

  }

  next(error)

}

export { logErrors, errorHandler, boomErrorHandler }

Nota: BOOM es solo para errores de http, no para errores de mongo, pero se puede implementar para uso de http con mongodb

Antes de hacer uso de los middlewares creados con anterioridad, haré un ejemplo de donde se tienen que implementar un middleware (esto significa que **no** llamaré a los middlewares antes creados), en este caso utilizaré la ruta /pokemon con método GET del archivo ‘app.ts’, para ello la modifico de esta manera:

// Ruta GET consulta pokemon

app.get('/pokemon', async (req, res) => {

  try {

    //busca todos lo pokemons

    const pokemons = await pokemonService.findAll()

    //200 por que busca si existen pokemons en el modelo

    res.status(200).json(pokemons)

  } catch (error) {

    // Esto tambien lo ejecuta la funcion boomErrorHandlern de error.handler.ts

    if (error.isBoom) {

      res.status(error.output.statusCode).json(error.output.payload)

    }

    // Esto tambien lo ejecuta la funcion errorHandler de error.handler.ts

    res.status(500).json(error)

  }

})

Ahora bien, para forzar a que suceda el error y así mostrarlo en el middleware lo siguiente será modificar el archivo del servicio, específicamente el método que utiliza el método GET “findAll()”, de esta manera:

  async findAll() {

    throw boom.notFound('There are not pokemons')

    const pokemons = await Pokemons.find().catch((error) => {

      console.log('Could not find pokemons', error)

    })

    return pokemons

  }

Lo único que hice fue agregar la línea “throw boom.notFound('There are not pokemons')”

Esta al ejecutarse primero no dejara que lo de abajo se ejecute, provocando el error (se lanza el error “throw”), de esta manera guardando los cambios y a su vez probando el endpoint con Postman, se tiene lo siguiente:

Captura de pantalla de computadora

Descripción generada automáticamente

Como se aprecia en la imagen muestra de manera más elegante el error:

{

    "statusCode": 404,

    "error": "Not Found",

    "message": "There are not pokemons"

}

Ahora bien, para poder hacer uso de los middlewares (los que había creado con anterioridad en el archivo error.handler.ts), primeramente, es necesario importarlos al archivo “app.ts”:

// importacion de los middlewares

import {

  logErrors,

  errorHandler,

  boomErrorHandler

} from './middlewares/error.handler'

Para que la aplicación las utilice, tengo que declararlas de manera secuencial y ordenada a la estructura, para ello escribo el nombre de la aplicación que en este caso es ‘app’ seguido de lo que usara:

**app.use(lo que usará)**

Al usar **use** estoy metiendo a la aplicación una lista de actividades que debe realizar la misma, por lo que en el orden en el que las vaya declarando cada actividad importa.

El orden debe ser el siguiente:

app.use(logErrors) //error que se muestra en el servidor (en la terminal)

app.use(boomErrorHandler)

app.use(errorHandler) // error 500

y deben ser declarados hasta al final, por que primero se tienen que registrar las rutas y después los middlewares, si no se hace de esa forma no funcionará.

Para aplicar su uso en la ruta /pokemon con método GET, se agrega el parámetro next para que ejecute el middleware:

// Ruta GET consulta pokemon

app.get('/pokemon', async (req, res, next) => {

  try {

    //busca todos lo pokemons

    const pokemons = await pokemonService.findAll()

    //200 por que busca si existen pokemons en el modelo

    res.status(200).json(pokemons)

  } catch (error) {

    next(error)

  }

})

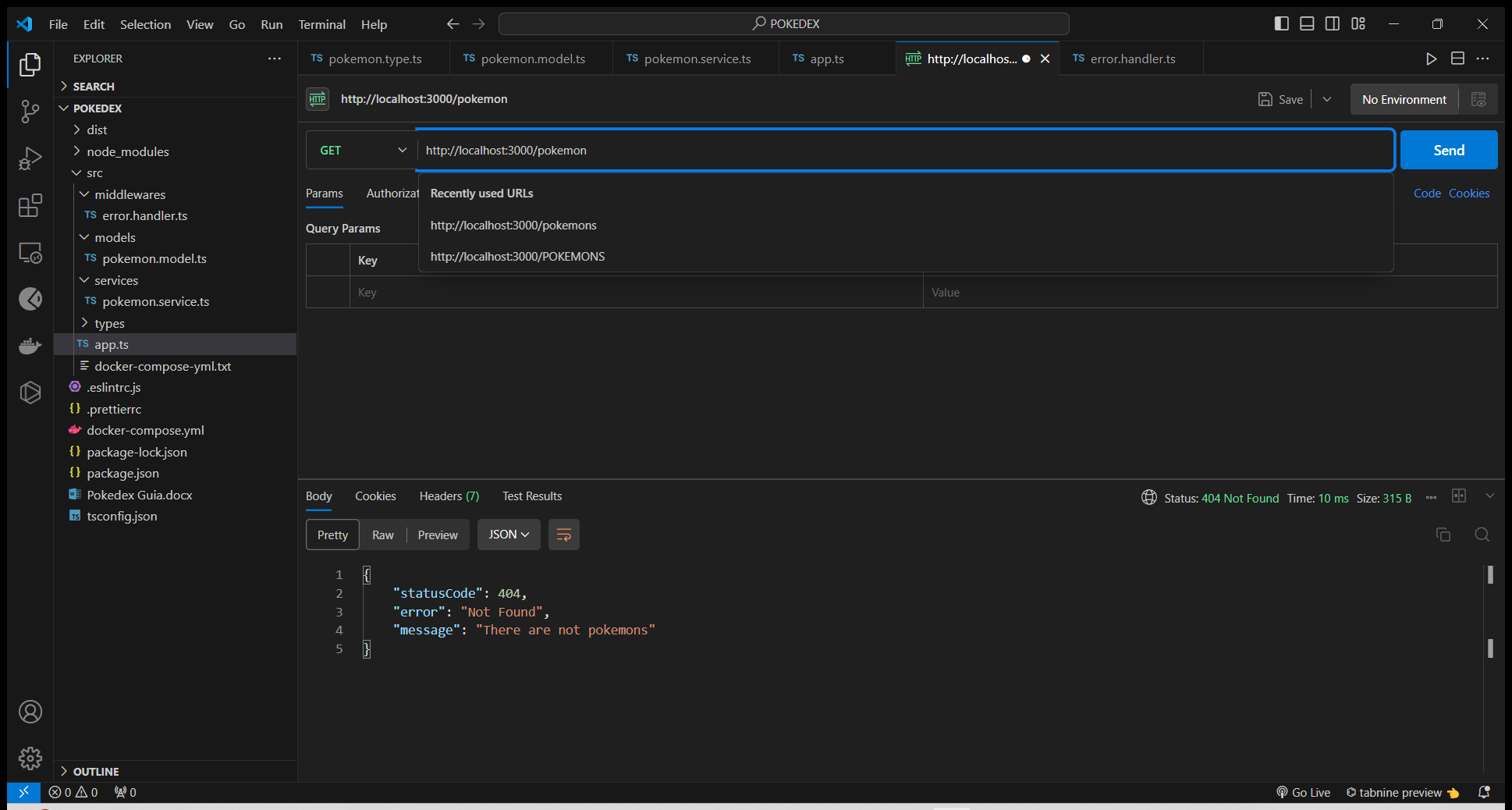
Con ello ya no es necesario decir explícitamente que middleware usar, simplemente en la cadena se van descartando los que no son.

Por otro lado, hay que seguir manteniendo el error forzado en el archivo de servicios para probar el funcionamiento de los middlewares:

    throw boom.notFound('There are not pokemons')

Si vuelvo a probar:

Lo que ejecutan los middlewares boomErrorHandler y errorHandler:



Lo que ejecuta el middleware logErrors

Texto

Descripción generada automáticamente

Para finalizar volveré a dejar como estaba el servicio de “pokemon.service.ts” que utilice para no afectar a futuras prácticas, únicamente agregare la sentencia que permita visualizar el error cuando este se presente, además de una modificación repentina en el primer manejo de error en el método de mongodb:

  //Metodo para que encuentre todo

  async findAll() {

    const pokemons = await Pokemons.find().catch((error) => {

      // Ahora este error sería de mongo

      console.log('Error while connecting to the DB', error)

    })

    // Si no existen pokemons manda el error

    if (!pokemons) {

      throw boom.notFound('There are not pokemons')

    }

    return pokemons

  }

Código completo en el archivo app.ts:

//app.ts

import express from 'express'

import mongoose from 'mongoose'

//importacion para el metodo post

import { Pokemon } from './types/pokemon.type'

//importo el servicio

import PokemonService from './services/pokemon.service'

// importacion de los middlewares

import {

  logErrors,

  errorHandler,

  boomErrorHandler

} from './middlewares/error.handler'

const MONGO\_URI = 'mongodb://localhost:27017/pokedex' //pokedex será el nombre de la bdd

//creo la aplicacion de express

const app = express()

//hacer que la app pueda recibir json con el middleware

app.use(express.json())

const port = 3000

const connectDB = () => {

  mongoose

    .connect(MONGO\_URI)

    .then(() => {

      console.log('Connected to Database :)')

    })

    .catch((error) => {

      console.log('Could not connect to Database', error)

    })

}

//creo una instancia del servicio

const pokemonService = new PokemonService()

//ruta normal

app.get('/', (req, res) => {

  res.send('Hola saludame, :)')

})

// Ruta Post guarda un pokemon en la base de datos

app.post('/pokemon', async (req, res) => {

  const pokemon: Pokemon = req.body

  //se crea al pokemon, y se guarda en la instancia newPokemon

  const newPokemon = await pokemonService.create(pokemon)

  //Envió la respuesta al cliente con ‘res’ y envió una respuesta del objeto newPokemon en formato JSON con código de estado 201

  res.status(201).json(newPokemon)

})

// Ruta GET consulta pokemon

app.get('/pokemon', async (req, res, next) => {

  try {

    //busca todos lo pokemons

    const pokemons = await pokemonService.findAll()

    //200 por que busca si existen pokemons en el modelo

    res.status(200).json(pokemons)

  } catch (error) {

    next(error)

  }

})

app.listen(port, () => {

  console.log(`Server is running at http://localhost:${port}`)

  connectDB() //conecto la base de datos pokedex

})

app.use(logErrors) //error que se muestra en el servidor (en la terminal)

app.use(boomErrorHandler)

app.use(errorHandler) // error 500

## Fase 4 (Manejo de rutas)

*En esta parte realizare algunas modificaciones en los archivos, entre ellas es la eliminación de los servicios POST y GET que manejan la ruta (/pokemon) en el archivo app.ts, dichos servicios serán utilizadas en otro archivo.*

Tener las rutas separadas por archivos es muy importante para estructurar de manera correcta el proyecto, lo que haré será crear un nuevo directorio dentro del archivo “src”, se llamará “routes” y dentro de ella el archivo de configuración de las rutas, la nomenclatura es la siguiente:

**nombre.route.ts**

**Ejemplo:**

//pokemon.route.ts

// importo express

import express from 'express'

// importo el tipo requerido para las rutas

import { Pokemon } from '../types/pokemon.type'

// importo el servicio requerido para las rutas

import PokemonService from '../services/pokemon.service'

//obtener el router de express, se necesita para crear una subruta en esta sección

const router = express.Router()

//creo una instancia del servicio en app.ts el nombre era pokemonSevice

const service = new PokemonService()

// Ruta Post guarda un pokemon en la base de datos

router.post('/', async (req, res) => {

  const pokemon: Pokemon = req.body

  //se crea al pokemon, y se guarda en la instancia newPokemon

  const newPokemon = await service.create(pokemon)

  //Envió la respuesta al cliente con ‘res’ y envió una respuesta del objeto newPokemon en formato JSON con código de estado 201

  res.status(201).json(newPokemon)

})

// Ruta GET consulta pokemon

router.get('/', async (req, res, next) => {

  try {

    //busca todos lo pokemons

    const pokemons = await service.findAll()

    //200 por que busca si existen pokemons en el modelo

    res.status(200).json(pokemons)

  } catch (error) {

    next(error)

  }

})

//exporto para que el main lo reciva y lo pueda ejecutar

export default router

Express maneja un patron de diseño llamado singleton, es una clase que solo puede tener una sola instancia

Un ejemplo es cuando creo la aplicación

//creo la aplicacion de express

const app = express()

esto crea la instancia express y esa instancia estará **compartida**, en todos los lugares donde lo importe.

Así que en lugar de app es router para las rutas:

router.post(…)

router.get(…)

Un punto a tomar en cuenta es que en el archivo “pokemon.route.ts” las rutas no se están poniendo completas, esto es, en vez de hacer esto:

router.post('/pokemon', async (req, res) => {

hago esto:

router.post('/', async (req, res) => {

tanto para el método POST como el GET, esto lo hago porque estableceré un archivo que ya defina la ruta predeterminada. A continuación, describo el proceso:

Dentro del directorio “routes” crearé un archivo “index.ts” donde establezca todas rutas que vaya a crear a futuro y en conjunto, de momento el contenido es el siguiente:

// index.ts

// importo express

import express from 'express'

// importo el router PokemonRouter

import PokemonRouter from './pokemon.route'

// funcion que recibe la aplicación

const routerApi = (app) => {

  //obtener el router de express

  const router = express.Router()

  // la aplicación va a utilzar una ruta "versionada" (versión 1)

  app.use('/api/v1', router)

  // subrouter que va utilizar

  router.use('/pokemons', PokemonRouter)

}

// exporto para que pueda comunicarse con el archivo app.ts

export default routerApi

Otro punto importante es que se tiene que exportar la función routerApi para que el archivo “app.ts” pueda hacer uso de la función de esta manera:

//importo los routers de ./routes/index.ts NO SE PONE index.ts

import routerApi from './routes'

// hacer que routerApi sepa que utlizando la aplicacion "app"

routerApi(app)

Con lo anterior realizado, ahora el endpoint para consultar o insertar debe ser el siguiente:

<http://localhost:3000/api/v1/pokemons>

donde “/api/v1/pokemons” es la ruta que establecí para esta sección en el archivo src/routes/index.ts

Haciendo la prueba:

Captura de pantalla de computadora

Descripción generada automáticamente

### Pasos extra (min 2:07:00):

Como desarrollo extra ahora, creare los servicios con su respectivas ruas para buscar por un tipo de dato, en este caso serán con dos por “id” y por “name”.

Primero empezare por los servicios (findById y findByName), cada uno tendrá como parámetro una variable de tipo string:

  //Metodo para que encuentre por ID

  async findById(id: string) {

    const pokemon = await Pokemons.findById(id).catch((error) => {

      console.log('Error while connecting to the DB', error)

    })

    // Si no existe el id del pokemon manda el error

    if (!pokemon) {

      throw boom.notFound('Pokémon not found with id: ' + id)

    }

    return pokemon

  }

  //Metodo para que encuentre por Nombre

  async findByName(name: string) {

    // findOne porque en el modelo el nombre es único, si es find retorna un arreglo

    // entre parentesis mando un objeto con las propiedades que quiero buscar

    // en este caso se busca por el atributo 'name'

    // naturalmente la manera de buscar por el metodo seria: findOne({ name: name })

    // el primer 'name' representa el atributo del objeto guardado en el modelo y el segundo 'name'

    // la variable en el parametro, pero ya que ambas 'por asi decirlo' se llaman igual la declaración en la

    // función se puede dejar asi: findOne({ name })

    const pokemon = await Pokemons.findOne({ name }).catch((error) => {

      console.log('Error while connecting to the DB', error)

    })

    // Si no existe el nombre del pokemon, se lanza el error

    if (!pokemon) {

      throw boom.notFound('Pokémon not found with name: ' + name)

    }

    return pokemon // Devolver el Pokémon encontrado

  }

Con respecto a las rutas, la estructura será parecida a las ya establecidas, pero con un poco de diferencia:

router.get('/id/:id', async (req, res, next) => {

  try {

    //busca a un pokemon por id

    const pokemon = await service.findById(req.params.id)

    //200 por que busca si existe el pokemon en el modelo

    res.status(200).json(pokemon)

  } catch (error) {

    // sino lo encuentra se pasa al middleware

    next(error)

  }

})

router.get('/name/:name', async (req, res, next) => {

  try {

    // busca un Pokémon por nombre

    const pokemon = await service.findByName(req.params.name)

    // si se encuentra el Pokémon, devolverlo

    res.status(200).json(pokemon)

  } catch (error) {

    // si hay un error, pasarlo al middleware

    next(error)

  }

})

Como se aprecia, las rutas nuevas ahora contienen una subruta cada una, esto se debe a que al buscar un cierto parámetro debe de estar especificado en rutas diferentes, esto es, si quiero buscar “name” utilizando una ruta, no puedo usar esa misma ruta para buscar “id”, esto puede hacer que las funciones choquen entre si porque se está buscando un parámetro en una misma ruta. El error se puede presentar si en lugar de lo anterior se hace lo siguiente:

router.get('/:id', async (req, res, next) => {…})

router.get('/:name', async (req, res, next) => {…})

en este ejemplo el error es evidente dos funciones están buscando un parámetro del mismo tipo (string) en la misma ruta, por lo que si busco un “name” en el endpoint activara la primera función que busca id, lo que no debe pasar ya que solo se busca el “name” el cual para eso le corresponde la segunda función:

**“/”** que representa: <http://localhost:3000/api/v1/pokemons/parametro>

En resumen, se genera confusión en Express porque el enrutador interpretará el parámetro **:id** o **:name** como el mismo.

la forma correcta (“/id/”, “/name/”) debe ser así:

<http://localhost:3000/api/v1/pokemons/id/>parametro

<http://localhost:3000/api/v1/pokemons>/name/parametro

de esta manera los métodos establecidos no chocan entre si

probando de la manera correcta por id:

Captura de pantalla de computadora

Descripción generada automáticamente

Probando de manera correcta por name:  
Captura de pantalla de computadora

Descripción generada automáticamente

Buscar por query usando únicamente la función findById, método implementado en el video:

Una captura de pantalla de una computadora

Descripción generada automáticamente

0Captura de pantalla de un celular

Descripción generada automáticamente

## Fase 5 (archivo ENV y libreria)

Como siguiente paso, creare un archivo que guarde mis variables de entorno este se alojará en la raíz del proyecto y tendrá el nombre de “.env”, a parte crearé también uno con nombre “.env.example”

Para aclarar el archivo “.env” no se guarda en un repositorio debido a que contiene credenciales o información importante, la cual no debe compartirse, sin embargo el archivo “.env.example” si se guarda dentro del repositorio, este se encarga de visualizar la estructura que tiene el archivo “.env” visualizar la información de lo que se debe poner en las credenciales o datos que requiere el proyecto para que este funcione.

Dentro del archivo “.env”:

PORT=3000

MONGO\_URI='mongodb://localhost:27017/pokedex'

Dentro del archivo “.env.example”

PORT=

MONGO\_URI=

Por consiguiente, debo crear un archivo de configuración que me permita leer el contenido (en este caso **variables de entorno**) del archivo “.env”

Para ello crearé un nuevo directorio como es de costumbre dentro de “src”, este se llamará “config” y dentro del mismo el archivo” config.ts”, en este archivo estableceré las funciones de necesarias, pero antes, debo instalar un servicio necesario para esta configuración:

**npm i dotenv**

// servicio para leer el archivo .env

import dotenv from 'dotenv'

// leer el archivo.env (ejecuto el servicio)

dotenv.config()

// guardo la configuracion y la exporto

export const config = {

  // ambiente de desarrollo

  env: process.env.NODE\_ENV || 'development',

  // ver si estamos en producción

  isProd: process.env.NODE\_ENV === 'production',

  // busca el puerto y sino encuentra informacion hará fallback a 3000

  port: process.env.PORT || 3000,

  // busca la url de mongo en la variable MONGO\_URI y sino encuentra informacion hará fallback a vacio

  mongoUri: process.env.MONGO\_URI || ''

}

Lo siguiente por hacer es que la aplicación (el archivo “app.ts”), reciba la configuración:

// importo la configuración de config.ts

import { config } from './config/config'

const {mongoUri, port} = config

const MONGO\_URI = mongoUri

la variable “port” la recibe la función listener:

app.listen(port, () => {

  console.log(`Server is running at http://localhost:${port}`)

  connectDB() //conecto la base de datos pokedex

})

Archivo “app.ts” actualizado:

// app.ts

import express from 'express'

import mongoose from 'mongoose'

// importacion de los middlewares

import {

  logErrors,

  errorHandler,

  boomErrorHandler

} from './middlewares/error.handler'

// importo los routers de ./routes/index.ts NO SE PONE index.ts

import routerApi from './routes'

// importo la configuración de config.ts

import { config } from './config/config'

// establesco las variables para configuraciones

const { mongoUri, port } = config

const MONGO\_URI = mongoUri

//creo la aplicacion de express

const app = express()

//hacer que la app pueda recibir json con el middleware

app.use(express.json())

// hacer que routerApi sepa que utlizando la aplicacion "app"

routerApi(app)

const connectDB = () => {

  mongoose

    .connect(MONGO\_URI)

    .then(() => {

      console.log('Connected to Database :)')

    })

    .catch((error) => {

      console.log('Could not connect to Database', error)

    })

}

//ruta normal

app.get('/', (req, res) => {

  res.send('Hola saludame, :)')

})

app.listen(port, () => {

  console.log(`Server is running at http://localhost:${port}`)

  connectDB() //conecto la base de datos pokedex

})

app.use(logErrors) //error que se muestra en el servidor (en la terminal)

app.use(boomErrorHandler)

app.use(errorHandler) // error 500

# Video 2

## Fase 1 (Autenticación por medio de json web token)

### Utilidades

Para configurar la autenticación u algún proceso es necesario tener diferentes rutas que organicen la estructura del código, por ello crearé un nuevo directorio de utilidades dentro del directorio “src”, llamado “utils”, este directorio conforme vaya avanzando será el encargado de almacenar mis utilidades, no solo la de autenticación.

Como parte de la actividad es necesario crear el tipo, modelo, servicio y router de la clase **usuario**, es de suma importancia tener dicha abstracción, ya que, la principal razón de la autenticación es tener un usuario existente en el sistema para comparar y validar credenciales.

### Tipo User versión 1:

// user.type.ts

// importar modelo de mongoose

import type { Model } from 'mongoose'

// tipo de dato

export type User = {

  id?: string

  name: string

  email: string

  address: string

  phoneNumber: string

  // en una bdd por cuestiones de auditorias se agrega:

  // cuando fue creado

  createdAt?: Date

  // cuando fue modificado

  lastModified?: Date

}

// exportar modelo

export type UserModel = Model<User>

### Modelo User versión 1:

Parte de la creación de un modelo es validar el formato que las propiedades dentro de la colección estén bien definidas para el usuario, un ejemplo sería el correo electrónico, esta propiedad es de tipo string, eso hace que el usuario pueda ingresar una cadena aleatoria, sin embargo, se sabe que todos los correos tienen una estructura definida, por lo que en dicho campo no debe escribirse cualquier cosa, una solución es utilizar **expresiones regulares** (una serie de reglas que debe cumplir una cadena), para ello creé un archivo nuevo en el directorio “utils” este se llama “contants.ts” y en dicho archivo guardare constantes, de la misma manera que hago con las variables de entorno en el archivo .env:

export const EMAIL\_REGEX =

  /(([^<>()\[\]\\.,;:\s@"]+(\.[^<>()\[\]\\.,;:\s@"]+)\*)|(".+"))@((\[[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}])|(([a-zA-Z\-0-9]+\.)+[a-zA-Z]{2,}))/

export const PHONE\_NUMBER\_REGEX =

  /^(\+\d{1,2}\s?)?\(?\d{3}\)?[\s.-]?\d{3}[\s.-]?\d{4}$/

Pasando al modelo User, la configuración será la siguiente:

// user.model.ts

import { Schema, model } from 'mongoose'

import { User, UserModel } from '../types/user.type'

// importo las constantes

import { EMAIL\_REGEX, PHONE\_NUMBER\_REGEX } from '../utils/constants'

// nombre con el que voy a referir en el servicio

export const USER\_REFERENCE = 'User'

const Users = new Schema<User, UserModel>({

  name: {

    type: String,

    required: true,

    trim: true

  },

  email: {

    type: String,

    required: true,

    unique: true,

    index: true,

    trim: true,

    // le paso el arreglo de las expresiones regulares que debe cumplir, y un mensaje en caso de que no sea valido

    match: [EMAIL\_REGEX, 'Please enter a valid email address']

  },

  password: {

    type: String,

    required: true

  },

  address: {

    type: String,

    trim: true

  },

  phoneNumber: {

    type: String,

    trim: true,

    // le paso el arreglo de las expresiones regulares que debe cumplir, y un mensaje en caso de que no sea valido

    match: [PHONE\_NUMBER\_REGEX, 'Please enter a valid phone number']

  },

  createdAt: {

    type: Date,

    default: () => Date.now()

  },

  lastModified: {

    type: Date,

    default: () => Date.now()

  }

})

export default model(USER\_REFERENCE, Users)

Nota: existe una pagina para poder obtener los regex ya establecidos para las expresiones regulares:

[i Hate Regex - The Regex Cheat Sheet](https://ihateregex.io/)

<https://ihateregex.io/>

### Servicio User versión 1:

// user.service.ts

// import el modelo

import Users from '../models/user.model'

// importo el tipo

import { User, UserModel } from '../types/user.type'

// importo la libreria de boom

import boom from '@hapi/boom'

class UserService {

  // Función crear usuario

  async create(user: User) {

    const newUser = await Users.create(user).catch((error) => {

      console.log(' Error while connecting to the DB', error)

    })

    //envio error en caso de no crear el usuario

    if (!newUser) {

      throw boom.badRequest('Could not create user')

    }

    return newUser

  }

  // Función para buscar por correo

  async findByEmail(email: string) {

    const user = await Users.findOne({ email }).catch((error) => {

      console.log('Error while connecting to the DB', error)

    })

    if (!user) {

      throw boom.notFound('User not found with email: ' + email)

    }

    return user

  }

}

export default UserService

#### NOTA IMPORTANTE:

en las importaciones, existe una diferencia en exportar con corchetes y exportar un nombre:

import nomSinCorchetes from 'ruta'

import { nomConCorchetes } from 'ruta'

cuando importas sin corchetes estas importando un nombre por default, esto es, al momento de exportar en la ruta donde se esta creando la clase u objeto, puedes importar dicho objeto en otro archivo con otro nombre, de allí el default.

**Ejemplo:**

// user.service.ts

export default UserService

y se importa:

// user.route.ts

import UserService from '../services/user.service'

o también puede importarse lo mismo, de esta manera (con otro nombre):

// user.route.ts

import OtraCosa from '../services/user.service'

en otro archivo donde se importará lo que se exporto…

import nomSinCorchetes from 'ruta'

Por otro lado, con los corchetes se importa una cosa (objeto) en específico, esto lo suelo hacer en los archivos de los ‘tipos’, donde exporto objetos por separado de esta forma:

**Ejemplo:**

// pokemon.type.ts

export type Pokemon = {

  id?: string

  name: string

  type: string

  weight: string

  height: string

}

export type PokemonModel = Model<Pokemon>

aquí se están exportando Pokemon y PokemonModel

y para recibir en tal caso únicamente el tipo ‘Pokemon’ en un archivo diferente, se hace de esta manera:

//pokemon.route.ts

import { Pokemon } from '../types/pokemon.type'

### Router User versión 1:

// user.route.ts

// importo express

import express from 'express'

// importo los tipo user

import { User } from '../types/user.type'

// importo el servicio de user

import UserService from '../services/user.service'

const router = express.Router()

const service = new UserService()

// crear usuario en la BD

router.post('/', async (req, res, next) => {

  try {

    const user: User = req.body

    const newUser = await service.create(user)

    res.status(201).json(newUser)

  } catch (error) {

    next(error)

  }

})

// Buscar por todos los usuarios (no recomandable, solo para pruebas)

router.get('/', async (req, res, next) => {

  try {

    const user = await service.findAll()

    res.status(200).json(user)

  } catch (error) {

    next(error)

  }

})

// Buscar por email

router.get('/:email', async (req, res, next) => {

  try {

    const user = await service.findByEmail(req.params.email)

    res.status(200).json(user)

  } catch (error) {

    next(error)

  }

})

export default router

Por otro lado, se también se tiene que actualizar el archivo src/router/index.ts, únicamente se agrega el router usuario:

// importo el router UserRouter

import UserRouter from './user.route'

  // subrouter user

  router.use('/users', UserRouter)

# Video 3

### Encriptación de contraseña

Con el tipo, modelo, servicio y router finalizados, ya puedo guardar datos de un usuario en la base de datos, sin embargo, un punto a recalcar es que aun faltan puntos a mejorar, como la encriptación de las contraseñas de usuario o retornar por método GET solo algunos campos (“name”, “email”, etc) y no todos los atributos (“password”, “createdAt”, etc)

**Ejemplo:**

En el siguiente ejemplo no se debe retornar los atributos de auditorías, y la contraseña la cual no se encuentra encriptada

Captura de pantalla de computadora

Descripción generada automáticamente

Para encriptar la contraseña haré uso de una dependencia:

npm i bcrypt

🡪 librería para encriptar

Esta librería la incluiré en el archivo de servicios, existen otras maneras para utilizarla y hacer el mismo proceso de encriptación en la contraseña, como interceptar la contraseña directamente en el tipo, en esta ocasión la encriptare en el método de creación de usuario.

**user.service.ts:**

// importo la libreria de encriptación

import bcrypt from 'bcrypt

  // Función crear usuario

  async create(user: User) {

    const hashedPassword = await bcrypt.hash(user.password, 10)

    // Le paso el objeto (...user) que copiará todas las propiedades del objeto user dentro del nuevo objeto que se va a crear

    // este proceso lo que hará es pasar individualmente cada propiedad del objeto

    // y por otro lado especifico como instará el campo contraseña (password: hashedPassword)

    const newUser = await Users.create({

      ...user, //importante no omitir los '...'

      password: hashedPassword

    }).catch((error) => {

      console.log('Could not save user', error)

    })

    return newUser

  }

Ahora bien, ya que tengo la parte del usuario resuelta, puedo pasar con la autenticación, parte de este proceso también provino del **video 2**, pero apenas empezaré con la implementación.

Para esta fase utilizare la herramienta llamada “Passport”, primeramente, en el proyecto es necesario instalar las dependencias, las cuales no serán de desarrollo, sino dependencias normales que también utilizare en producción.

npm i passport

*passport es una herramienta que abstrae la capa de autenticación de una manera más sencilla, de esa manera solo solo me encargaría ver donde utilizaría esa autenticación, passport tiene diferentes estrategias, cada una es una manera de autenticación, de momento utilizare la estrategia local, que es por medio de código propio (existe la manera de autenticarse por Google, facebook, GitHub, etc).*

Aparte tendré que instalar lo siguiente:

npm i passport-jwt

npm i passport-local

Ahora bien, dentro de utils crearé otro directorio llamado “auth” y dentro de auth un directorio llamado “strategies”, debe quedar de la siguiente manera:

src

└───utils

└───auth

└───strategies

Después dentro del directorio “strategies” crearé una estrategia, con nomenclatura:

**nombre.strategy.ts**

La primera **local.strategy.ts**, que se encargará de la autenticación de manera local

// local.strategy.ts

// importar Strategy de passport (es como su propia estrategia)

import { Strategy } from 'passport-local'

// importar bcrypt (libreria para encriptar)

import bcrypt from 'bcrypt'

// importar boom en caso de errores

import boom, { unauthorized } from '@hapi/boom'

// importo los servicios user

import UserService from '../../../services/user.service'

// importo el tipo user

import { User } from '../../../types/user.type'

// creo el objeto de las opciones

// usernameField = el campo que identificara al usuario (tambien se puede enviar el nombre del usuario, pero se opta mejor por el correo)

// passwordField = el campo que identificara la constraseña del mismo usuario

// Nota importante usernameField y passwordField deben estar escritos de esa forma no de otra sino no funcionará

const options = { usernameField: 'email', passwordField: 'password' }

// instancio un objeto de tipo UserService

const service = new UserService()

// creo la estrategia, le paso el objeto de opciones(options),

// tambien le mando una funcion que maneje el correo(email) la contraseña(password) y un middleware(next)

// dicha funcion intentara extraer la información del usuario

const LocalStrategy = new Strategy(options, async (email, password, next) => {

  try {

    const user: User = (await service.findByEmail(email)) as unknown as User

    // validar constraseña

    if (user) {

      // bcrypt compara la contraseña con la que esta encriptada

      const isMatch = await bcrypt.compare(password, user.password)

      delete user.password //hago que no retorne la contraseña

      // si hace match se permite el acceso

      if (isMatch) {

        next(null, user)

      } else {

        // en caso contrario se envia un unauthorized

        next(boom.unauthorized('Usuario o contraseña no validas'), false)

      }

    } else next(boom.unauthorized('Usuario o contraseña no validas'), false)

  } catch (error) {

    next(error, false)

  }

})

export default LocalStrategy

Para utilizar la estrategia, es necesario crear una nueva ruta donde se pueda efectuar la autenticación, por ello en el directorio src/route/ creé un nuevo archivo

**auth.route.ts** y dentro del mismo, como **primera versión** de prueba:

// auth.route.ts

import express from 'express'

import passport from 'passport'

import UserService from '../services/user.service'

const router = express.Router()

const service = new UserService()

router.post(

  '/login',

  // en passport uso el metodo de autenticación y le paso la estrategia 'local'

  passport.authenticate('local', { session: false }),

  // esta function callback funcionará siempre y cuando el middleware passport

  async (\_req, res, next) => {

    try {

      res.status(200).json({ message: 'Todo en orden YEAH!!' })

    } catch (error) {

      next(error)

    }

  }

)

export default router

Después actualizo el archivo src/routes/index.ts, agregando:

// importo el router AuthRouter

import AuthRouter from './auth.route'

  // subrouter auth

  router.use('/auth', AuthRouter)

Como parte de la actividad también es necesario configurar passport, para ello creo un

archivo utils/auth/index.ts y dentro:

// importo passport

import passport from 'passport'

// importo la strategia local

import LocalStrategy from './strategies/local.strategy'

passport.use(LocalStrategy)

por otro lado, también debo importar passport y el archivo de configuración ‘utils/auth/index.ts’ dentro de la aplicación (app.ts):

// importo passport para que la aplicación la utilice

import passport from 'passport'

// importo el archivo que contiene las estrategias de autenticación ./utils/auth/index.ts NO SE PONE index.ts

import './utils/auth'

app.use(passport.initialize())

// hacer que routerApi sepa que utlizando la aplicacion "app"

Haciendo la prueba:

Captura de pantalla de computadora

Descripción generada automáticamente

# Video 4

## JsonWebTokens

Al ya poder acceder a una ruta protegida, por medio autenticación, ahora sigue utilizar **json web tokens**, para ello instalo la siguiente dependencia:

**npm i jsonwebtoken**

Por consiguiente, es necesario crear ahora una estrategia para identificarme con un **JWT**, cuando el usuario se loguea, en la ruta de autenticación, tengo que retornarle al mismo usuario el JWT que tiene que guardar para identificarse en posteriores llamadas.

Para lograrlo, tengo que ir a mi ruta de autenticación y lo que haré será consultar el usuario:

**auth.route.ts** versión 2

// auth.route.ts

import express from 'express'

import passport from 'passport'

import UserService from '../services/user.service'

// importo Request de express

import type { Request } from 'express'

// importo el tipo user

import type { User } from '../types/user.type'

// importo la dependencia JWT

import jwt from 'jsonwebtoken'

// importo el archivo de configuración, para utilizar el secreto jwt

import { config } from '../config/config'

const router = express.Router()

const service = new UserService()

// aparte de tener todos los request, tambien voy a tener un usuario

type RequestType = Request & {

  user: User

}

router.post(

  '/login',

  // en passport uso el metodo de autenticación y le paso la estrategia 'local'

  passport.authenticate('local', { session: false }),

  // esta function callback funcionará siempre y cuando el middleware passport

  //el req apartir de ahora será de tipo RequestType

  async (req: RequestType, res, next) => {

    try {

      //consultar el usuario, por medio de un Request

      // de request voy a obtener user

      const { user } = req

      // console.log(user) console.log para mostrar los datos completos del usuario en la terminal

      // sub is the id of the subscribed user

      const payload = { sub: user.id }

      // la variable jwt tiene un metodo llamado sign de firmar

      // dicha función recibirá el payload que va a firmar y el secreto

      // esta función es la crea el token

      const token = jwt.sign(payload, config.jwtSecret)

      //ahora retorno el usuario, junto el token

      res.status(200).json({ user, token })

    } catch (error) {

      next(error)

    }

  }

)

export default router

#### NOTA IMPORTANTE:

passport cuando realiza la autenticación inyecta el usuario en el **request**, este proceso se aprovechó para que el mismo request (req) pudiera admitir un tipo user

// importo el tipo user

import type { User } from '../types/user.type'

// aparte de tener todos los request, tambien voy a tener un usuario

type RequestType = Request & {

  user: User

}

  async (req: RequestType, res, next) => {

Por otro lado, normalmente en el **payload** se guarda información, por ello se en la ruta de autenticación se aprovecha para almacenar en la variable payload el id del usuario que se asigna con el nombre ‘sub’.

const payload = { sub: user.id }

esto sirve para que en futuras consultas del payload se pueda validar por medio del reconocimiento del id el rol del usuario, en este caso si es administrador o usuario normal, se puede lograr usando un middleware.

un ejemplo de un payload puede ser:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente}

Imagen extraída de <https://jwt.io/>

Ahora bien, después de establecer la ruta ‘/login’ realizo la prueba de esta nueva versión de código del archivo:

Captura de pantalla de computadora

Descripción generada automáticamente

Por otro lado, si se descomenta el console.log(user), en la consola se puede apreciar como passport inyecta el usuario:

Texto

Descripción generada automáticamente

Con auth.route.ts de momento terminado, ahora es necesario crear una manera de desencriptar un webtoken, para ello creare un nuevo archivo en src/utils/auth/jwAuth.ts, en dicho archivo creare una función que sirva para verificar un JsonWebToken:

// jwAuth.ts

// importo la dependencia de JWT

import jwt from 'jsonwebtoken'

// importo el objeto config

import { config } from '../../config/config'

// crearé una función que reciba un objeto tipo string

// y valide que el token no este vacio

const verifyJwt = (token: string) => {

  // si no recibe un token, retorna null

  if (!token) return null

  try {

    // decodifico el token

    const decoded = jwt.verify(token, config.jwtSecret)

    //si no se logra la decodificación regresa null

    if (!decoded) return null

    // retorno el payload {sub: id}

    return decoded

  } catch (error) {

    console.log(error)

  }

}

// función para extraer

const extractFromJwt = (authorization: string) => {

  // regresa Bearer token

  const token = authorization.split(' ')[1]

  return verifyJwt(token)

}

// exporto las funciones, extractFromJwt no es necesario exportarla

export { verifyJwt, extractFromJwt }

el archivo jwtAuth sirve para leer los JsonWebToken, lo siguiente es crear una **estrategia** que utilice el método verifyJwt, para ello creo una nueva estrategia en la carpeta strategies y con nombre **jwt.strategy.ts** y dentro de ella:

// jwt.strategy.ts

// importo la estrategia de passport-jwt

import { Strategy, ExtractJwt } from 'passport-jwt'

// importo el objeto config para hacer uso del secreto

import { config } from '../../../config/config'

// variable de opciones

const options = {

  // funciona igual que la función del archivo jwtAuth.ts "extractFromJwt"

  jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),

  secretOrKey: config.jwtSecret

}

// creo la estrategia, donde recibe las opciones una función que a su vez recibe el payload, el next (middleware)

const JwtStrategy = new Strategy(options, (payload, next) => {

  return next(null, payload)

})

export default JwtStrategy

#### NOTA IMPORTANTE:

Se supone que **jwt.strategy.ts**, debería utilizar o hacer uso de las funciones del archivo **jwAuth.ts**, sin embargo, esto no se hizo porque el método:

jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),

prácticamente hacia la funcionalidad de la función “extractFromJwt” por lo que al final no se utilizó ninguna función.

Para utilizar la estrategia es necesario inicializarla en el src/utils/auth/index.ts y respetar el orden de las declaraciones passport.use()

// src/utils/auth/index.ts

// importo passport

import passport from 'passport'

// importo la strategia local

import LocalStrategy from './strategies/local.strategy'

// importo la strategia jwt

import JwtStrategy from './strategies/jwt.strategy'

passport.use(LocalStrategy)

passport.use(JwtStrategy)

Al tener la configuración para la autenticación con JsonWebToken, puedo realizar una prueba con una ruta, en este caso utilizare las rutas del archivo **pokemon.route.ts**, en ella lo único que debo hacer es exportar passport y escribir una línea de código en un método (para el ejemplo uno con GET), antes de la función asíncrona:

// importo passport

import passport from 'passport'

// Rutas GET consulta pokemon

router.get(

  '/',

  passport.authenticate('jwt', { session: false }),

  async (req, res, next) => {…}

)

Al realizar la prueba, se tiene:

Captura de pantalla de un celular

Descripción generada automáticamente

Esto es correcto porque no se está logueado, y para poder ingresar se tiene que utilizar el token del usuario, insertándolo en la sección Authorization🡪Berer Token

Captura de pantalla de un celular

Descripción generada automáticamente

¡¡Listo!! Prácticamente la autenticación por jsonwebtoken esta implementada para esta ruta, para aplicar este mismo proceso con las demás **rutas que si necesitan autenticación**, solo basta con poner la sentencia que permite autenticarse con JWT:

  passport.authenticate('jwt', { session: false }),

esta sentencia, debe estar antes de los métodos asíncronos de las rutas.

Por otro lado, recordando que passport inyecta el usuario al momento de autenticar ya sea por la estrategia jwt o local en el request, con ello puedo hacer que el tipo al que infiere el request sea parte de los tipos de user, esto es, que en lugar de tener el siguiente tipo únicamente en el archivo **auth.route.ts:**

// importo el tipo user

import type { User } from '../types/user.type'

// aparte de tener todos los request, tambien voy a tener un usuario

type RequestType = Request & {

  user: User

}

lo coloque en el archivo **user.type.ts**:

export type UserRequestType = Request & {

  user: User

}

De esta manera puedo hacer que otras rutas puedan utilizar este tipo y de esta forma recuperar lo que inyecte la autenticación passport con estrategia local o jwt, en este caso la estrategia retorna el payloat que contiene únicamente el id del usuario ‘sub’.

Como había mencionado esto es bueno, porque así puedo validar el id para verificar su rol (usuario admin o usuario normal), ejemplo en **pokemon.route.ts.**

// importo el tipo UserRequestType para obtener el usuario por medio del request

import { UserRequestType } from '../types/user.type'

// Rutas GET consulta pokemon

router.get(

  '/',

  passport.authenticate('jwt', { session: false }),

  async (req: UserRequestType, res, next) => {

    try {

      const { user } = req

      console.log(user) //console.log para mostrar los datos del paylot (id) del usuario en la terminal

A continuación, muestro la inyección teniendo descomentado el console.log(user):

Captura de pantalla de computadora

Descripción generada automáticamente

Texto

Descripción generada automáticamente

# Video 5

Para continuar, oficialmente pasere a implementar la función que se encargue de ocultar información específica de un usuario cuando este se crea, consulta o loguea:

Una manera de implementar dicho procedimiento es por medio de un método, que retorne el usuario con algunos atributos necesarios (no todos), para el cliente. Moongose nos permite crear una especie de función, o métodos que pueden inyectar a un modelo.

Tomando en cuenta el archivo **user.model.ts**, agrego el método (no dentro del squema)**:**

// user.model.ts

// Users tiene una propiedad que se llama metodos

// y detro de metodos crearé la función toClient

// y el toClient va a retornar un objeto con todas

// las propiedades que si se necesitan

Users.methods.toClient = function () {

  return {

    id: this.\_id as unknown as string,

    name: this.name,

    email: this.email,

    address: this.address,

    phoneNumber: this.phoneNumber

  }

} // este metodo debe de estar definido en el tipo

Como se menciona el método creado debe estar definido en el tipo user, en esta nueva implementación se modificara un poco lo que ya se tenía, **user.type.ts** versión 2

// user.type.ts

// importar modelo de mongoose

import type { Model } from 'mongoose'

// importo Request de express

import type { Request } from 'express'

// tipo de dato

// el usuario será todo lo que contenga toClient

export type User = ToClientUser & {

  //composicion

  password: string

  // en una bdd por cuestiones de auditorias se agrega:

  // cuando fue creado

  createdAt?: Date

  // cuando fue modificado

  lastModified?: Date

}

export type UserRequestType = Request & {

  user: User

}

// creé un tipo que contendra todas las propiedades que iran al cliente

export type ToClientUser = {

  id?: string

  name: string

  email: string

  address: string

  phoneNumber: string

}

// UserMethods tiene una función llamada toClient que es una función que retorna

// un objeto tipo ToClientUser

export type UserMethods = {

  toClient: () => ToClientUser

}

// exportar modelo, dentro del mismo el User, objeto vacio porque no modificare

// las propiedades del documento y por ultimo los metodos

export type UserModel = Model<User, {}, UserMethods>

Es importante recordar que es necesario especificar en el squema de usuarios que se usara **UserMethods**, por lo que regresando a **user.model.ts:**

import { User, UserMethods, UserModel } from '../types/user.type'

// En el modelo de usuarios se agrega UserMethods

const Users = new Schema<User, UserModel, UserMethods>({

Para finalizar, solo se tiene que aplicar la función **toClient()** dentro de las rutas, que retornan la información del usuario, estas están presentes en **auth.route.ts** y **user.route.ts**,

// user.route.ts

// crear usuario en la BD

router.post('/', async (req, res, next) => {

  try {

    const user: User = req.body

    const newUser = await service.create(user)

    res.status(201).json({ user: newUser.toClient() })

  } catch (error) {

    next(error)

  }

})

// Buscar por email

router.get('/:email', async (req, res, next) => {

  try {

    const user = await service.findByEmail(req.params.email)

    res.status(200).json({ user: user.toClient() })

  } catch (error) {

    next(error)

  }

})

// auth.route.ts

router.post(

  '/login',

  // en passport uso el metodo de autenticación y le paso la estrategia 'local'

  passport.authenticate('local', { session: false }),

  // esta function callback funcionará siempre y cuando el middleware passport

  //el req apartir de ahora será de tipo RequestType

  async (req: UserRequestType, res, next) => {

    try {

      //consultar el usuario, por medio de un Request

      // de request voy a obtener user

      const { user } = req

      //console.log(user) //console.log para mostrar los datos completos del usuario en la terminal

      // sub is the id of the subscribed user

      const payload = { sub: user.id }

      // la variable jwt tiene un metodo llamado sign de firmar

      // dicha función recibirá el payload que va a firmar y el secreto

      // esta función es la crea el token

      const token = jwt.sign(payload, config.jwtSecret)

      // obtengo el 'modelo' del mismo usuario por medio del backend

      // esto es importante ya que la función toClient() funciona

      // con los usuarios provenientes del modelo y no del req

      const bdUser = await service.findByEmail(user.email)

      //ahora retorno el usuario, junto el token

      res.status(200).json({ user: bdUser.toClient(), token })

    } catch (error) {

      next(error)

    }

  }

# Video 6

## Modelo para el tipo de dato para otro modelo

Para esta parte de la actividad primero será necesario crear un nuevo modelo, el cual llamare ‘**generations**’:

// generation.type.ts

import type { Model } from 'mongoose'

export type Generation = ToClientUser & {

  createdAt?: Date

  lastModified?: Date

}

export type ToClientUser = {

  id?: string

  generation: number

}

export type GenerationMethods = {

  toClient: () => ToClientUser

}

export type GenerationModel = Model<Generation, {}, GenerationMethods>

//generation.model.ts

import { Schema, model } from 'mongoose'

import {

  Generation,

  GenerationMethods,

  GenerationModel

} from '../types/generation.type '

export const GENERATION\_POKEMON\_REFERENCE = 'Generation'

const Generations = new Schema<Generation, GenerationModel, GenerationMethods>({

  generation: {

    type: Number,

    required: true,

    unique: true,

    index: true

  }

})

Generations.methods.toClient = function () {

  return {

    id: this.\_id as unknown as string,

    generation: this.generation

  }

}

export default model(GENERATION\_POKEMON\_REFERENCE, Generations)

//generation.service.ts

import Generations from '../models/generation.model'

import { Generation } from '../types/generation.type '

import boom from '@hapi/boom'

class GenerationService {

  // Metodo para crear una generación

  async create(generation: Generation) {

    const newGeneration = await Generations.create(generation).catch(

      (error) => {

        console.log('Error while connecting to the DB', error)

      }

    )

    //envio error en caso de no crear el generation

    if (!newGeneration) {

      throw boom.badRequest('Could not create generation')

    }

    return newGeneration

  }

  //Metodo para que encuentre todo

  async findAll() {

    const generations = await Generations.find().catch((error) => {

      // Ahora este error sería de mongo

      console.log('Error while connecting to the DB', error)

    })

    // Si no existen generations manda el error

    if (!generations) {

      throw boom.notFound('There are not generations')

    }

    return generations

  }

  // Metodo para buscar por id

  async findById(id: string) {

    const generation = await Generations.findById(id).catch((error) => {

      console.log('Error while connecting to the DB', error)

    })

    if (!generation) {

      throw boom.notFound('Generation not found')

    }

    return generation

  }

  // Metodo para buscar por nombre de generación

  async findByGeneration(generation: number) {

    const generationData = await Generations.findOne({ generation }).catch(

      (error) => {

        console.log('Could not retrieve generation info', error)

      }

    )

    if (!generationData) {

      throw boom.notFound('Gen not found')

    }

    return generationData

  }

}

export default GenerationService

//generation.route.ts

import express from 'express'

import { Generation } from '../types/generation.type '

import GenerationService from '../services/generation.service'

const router = express.Router()

const service = new GenerationService()

// crear generación en la BD

router.post('/', async (req, res, next) => {

  try {

    const generation: Generation = req.body

    const newGeneration = await service.create(generation)

    console.log(newGeneration)

    res.status(201).json({ generation: newGeneration.toClient() })

  } catch (error) {

    next(error)

  }

})

// Buscar todas las generaciones

router.get('/', async (req, res, next) => {

  try {

    const generations = await service.findAll()

    res.status(200).json(generations)

  } catch (error) {

    next(error)

  }

})

// Buscar por id

router.get('/id/:id', async (req, res, next) => {

  try {

    const generation = await service.findById(req.params.id)

    res.status(200).json({ generation: generation.toClient() })

  } catch (error) {

    next(error)

  }

})

// Buscar por generation

router.get('/generation/:generation', async (req, res, next) => {

  try {

    const generation = await service.findByGeneration(

      parseInt(req.params.generation)

    )

    res.status(200).json({ generation: generation.toClient() })

  } catch (error) {

    next(error)

  }

})

export default router

Archivo index.ts

// importo el router GenerationRouter

import GenerationRouter from './generation.route'

  // subrouter auth

  router.use('/generations', GenerationRouter)

Ahora usando el método post inserto 8 generaciones:

Captura de pantalla de computadora

Descripción generada automáticamente

El tipo, modelo, servicios y rutas generation (generation.type.ts, generation.model.ts, generation.service.ts, generation.route.ts) los hice con la finalidad de que el modelo pokemon (pokemon.model.ts) pueda contener generation.

Para lograr esta relación en el tipado de pokemon (pokemon.type.ts), agregare un nuevo campo:

//pokemon.type.ts

// importo el tipo Generation

import { Generation } from './generation.type '

export type Pokemon = {

  // código faltante

  generation: Generation //agregue generation de tipo Generation

}

El tipado también se debe reflejar en el modelo de pokemon (pokemon.model.ts) de esta forma:

//pokemon.model.ts

// importo la referecia de Generation

import { GENERATION\_POKEMON\_REFERENCE } from './generation.model'

const Pokemons = new Schema<Pokemon, PokemonModel>({

  //código faltante

  generation: {

    // generation tendra un id, representaria la foreign key

    type: Schema.Types.ObjectId,

    // especifico la referencia

    ref: GENERATION\_POKEMON\_REFERENCE

  }

})

# Apuntes Primer video

Para instalaciones:

install = i

-D = --save-dev

Nota: recordar la diferencia entre dependencias de desarrollo y dependencias de producción, no son lo mismo

Para instalar una dependencia de desarrollo siempre, se debe poner o -D o --save-dev

En lo que llevo aprendido, las dependencias de producción y de desarrollo se pueden encontrar en el archivo “package.json” y dentro de la misma:

El apartado “devDependencies”:{…} para dependencias de desarrollo

El apartado “dependencias”:{…} para dependencias de producción

Min 1:05:06

Tareas a investigar:

Dependencias de desarrollo y producción

Configuraciones en el archivo “tsconfig.json”

Configuraciones para inter(eslint) y prettier

Que son las promesas y como aplicarlas

Investigar los datos(“status”) 200, 404, 201

En desarrollo web que son los modelos, tipos y servicios (enfocado a express y typescript)

Investigar si la instancia pokemons en el método get puede ser un arreglo

Investigar el contenido del archivo yamel mongo

Investigar que son los middlewares

Investigar como el propósito de este middleware:

const errorHandler = (error, \_req, res, \_next) => {

  if (!error.isBoom) {

    // internal server error

    res.status(500).json({

      message: error.message, // mostrar el error

      stack: error.stack // stack que mande el error

    })

  }

}

Investigar middlewares de autenticación

Investigar el patron de diseño de Express (singleton)

Investigar la funcionalidad completa del archivo src/config/config.ts

Investigar los métodos find de mongo (findOneAndDelete, findByIdAndDelete, findOne, find, etc) y los métodos de boom (notFound, badRequest, badImplementation, etc)

Investigar lo que son migraciones

Investigar la función completa de:

    createdAt?: Date

    lastModified?: Date

Migración: cuando agrego nueva información a los modelos, tengo que generar un script, que tendrá la funcionalidad de ir a todos los modelos y va a llenar con un dato default el dato faltante en el modelo, guarda la versión de la bdd, la fecha en la que se hizo etc.

Investigar a fondo las **expresiones regulares** y como aplicarlas, y verificar que es regex

Investigar para que sirve spread operator: video 3 min 24

Investigar la función:

delete user.password //hago que no retorne la contraseña

se supone que al retornar user, esta elimina password para no mostrarla al buscar por el endpoint (usando GET)

Investigar porque usernameField y passwordField deben estar escritos de esta de esa forma, para que no provoque error en la autenticación en el archivo **local.strategy.ts**:

Forma correcta

const options = { usernameField: 'email', passwordField: 'password' }

Forma incorrecta, cualquiera de las mostradas a continuación:

const options = { usernamefield: 'email', passwordfield: 'password' }

const options = { userNameField: 'email', passwordField: 'password' }

const options = { UsernameField: 'email', PasswordField: 'password' }

…etc

Investigar a detalle como funciona el archivo **jwt.strategy.ts**, del video 4 en min 19:00

Investigar cómo funciona la siguiente línea de código:

  passport.authenticate('jwt', { session: false }),

Estudiar a detalle el funcionamiento de los métodos aplicados en el tipado de **user.type.ts**

Modelo: es la identidad en nuestra base de datos

Tipo: es una utilidad para tipar los dobjetos que estamos utilizando dentro de typescript

{

  "name": "Pikachu",

  "type": "Electric",

  "weight": "6",

  "height": "0.4",

  "generation": 1

}