**CONFIGURAR SERVIDOR, MÓDULOS Y COMANDOS**

Creamos la carpeta server y la abrimos con visual studio code, podemos navegar por las carpetas con el comando cd.

>>cd server

Ahora vamos a crear el archivo package.json

>>npm init --yes

{

  "name": "server",

  "version": "1.0.0",

  "description": "",

  "main": "index.js",

  "scripts": {

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

  },

  "keywords": [],

  "author": "",

  "license": "ISC"

}

En la parte de scripts podemos crear los comandos que vamos a utilizar.

Ahora vamos a instalar las dependencias que vamos a necesitar.

-Express es un modulo que nos evita crear la conexión con servidor manualmente y esta muy probado en este entorno

-Morgan nos permite ver en consola del servidor lo que las aplicaciones nos van pidiendo

-promise-mysql para poder conectar de node.js a mysql, este modulo recibe promesas de java script así el código se ve más ordenado

-cors nos permite comunicar dos servidores ya que angular levanta un servidor en el puerto 4200 y este se va a tener que comunicar con el otro servidor que es el de la rest api

>>npm i express morgan promise-mysql cors

Podemos ver que aparece una carpeta llamada node\_modules, se crea package-lock.json y se actualiza el package.json con la información de las dependencias instaladas.

"dependencies": {

    "cors": "^2.8.5",

    "express": "^4.17.1",

    "morgan": "^1.10.0",

    "promise-mysql": "^5.0.3"

  }

Ahora vamos a crear la carpeta src para poner todo el código de nuestro servidor, dentro de src vamos a crear un archivo llamado index.ts y en este archivo vamos a poner un log de consola.

console.log('WORKS')

para poder correr este código debemos compilar de ts a js para eso vamos a instalar el typescript en nuestro equipo, para instalarlo debemos abiri la terminal como administrador.

>>npm install -g typescript

En caso de ya tenerlo solo vamos a ejecutar el comando tsc, si no tenemos el archivo de configuración de ts nos muestra un mensaje en consolo por lo de debemos crear este archivo, lo hacemos con el siguiente comando.

>>tsc –init

Esto nos crea el archivo tsconfig.json, dentro de este archivo vamos a cambiar unas opciones; dentro de target vamos a cambiar la versión para que lo compile a es6.

"target": "es6",

en outDir le quitamos el comentario y le ponemos la ruta donde queremos que haga la compilación.

"outDir": "./build",

Para poder comprobarlo vamos a guardar el archivo de configuración y en la terminal escribimos el comando tsc, ahora podemos ver que se creó la carpeta build con el código de js, para poder correr el código podemos utilizar el comando node.

>>node build/index.js

Podemos ver que se ejecuta correctamente el código de js pero al hacer cambios en el ts no se reflejan en el js, para poder automatizar este proceso vamos a crear un comando.

"scripts": {

        "build": "tsc -w"

    },

Para poder ejecutar este script utilizamos el siguiente comando.

>>npm run build

Este comando vigila los cambios del ts y los compila a ts, pero aun debemos ejecutar el comando node build/index.js para poder correr el archivo de js, para esto podemos instalar otro modulo llamado nodemon, para instalarlo lo podemos hacer solo en modo de desarrollo para que al compilar nuestro proyecto para producción no incluya este modulo

>>npm i nodemon -D

Esto me crea las dependencias de desarrollo estas no se van al modo de despliegue lo que hace que nuestro proyecto no se cargue de módulos y dependencias que no necesita.

"devDependencies": {

        "nodemon": "^2.0.12"

    }

Ahora vamos a crear otro comando para ejecutar el nodemon.

"scripts": {

        "build": "tsc -w",

        "dev": "nodemon build/index.js"

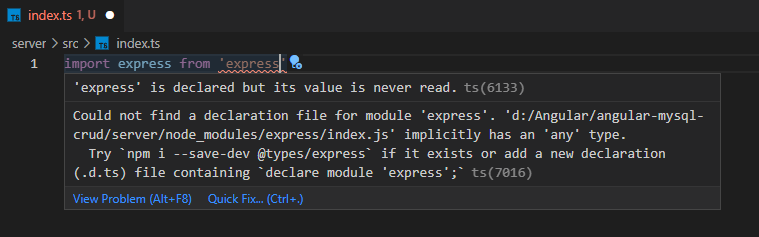
    },

Si ejecutamos este comando podemos ver que escucha los cambios del js y vuelve a iniciar el código.

>>npm run dev

**ESCRIBIR CÓDIGO DEL SERVIDOR**

Primero vamos a importar el express en nuestro index.ts, aquí se nos muestra un error diciendo que no se puede utilizar el autocompletado ya que no se entiende este módulo, para ayudar a ts vamos a instalar los types.



>>npm i @types/express -D

Ya se quita el error y podemos empezar a utilizar express, vamos a crear una clase llamada server y esta va a iniciar el servidor.

class Server {

}

new Server();

Y vamos a ejecutar un new Server para que me devuelva un objeto.

Creamos el constructor para que se ejecute cada vez que se crea una nueva clase server ( new Server() ), el constructor va a ejecutar express, este me va a devolver un objeto que lo vamos a almacenar en una propiedad.

constructor(){

        express();

    }

La propiedad la vamos a llamar app y es de tipo Aplication esto esta dentro de express para ello vamos a importarlo asi.

import express, { Application } from 'express'

La propiedad nos queda asi.

public app: Application;

el constructor nos queda asi.

constructor(){

        this.app = express();

    }

Con esto ya tengo la inicialización de express.

Ahora vamos a crear un método que se encarga de configurar la propiedad app y otro que se encarga de dar las rutas al app.

config(): void {

    }

    routes(): void {

    }

Por último, vamos a crear un método que se encarga de iniciar el servidor que será el típico listener.

start(): void {

    }

Vamos a crear el métodos config.

config(): void {

        // si nos dan un puerto en un servicio en la nube se lo toma sino es asi toma el puerto 3000

        this.app.set('port', process.env.PORT || 3000 );

    }

Ahora el método start.

start(): void {

        this.app.listen(this.app.get('port'), () =>{

            console.log('Server port '+ this.app.get('port'));

        });

    }

Ahora ejecutamos los métodos en el constructor.

constructor(){

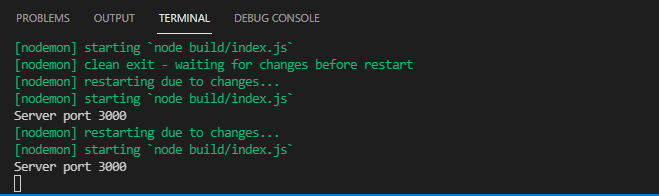
        this.app = express();

        this.config();

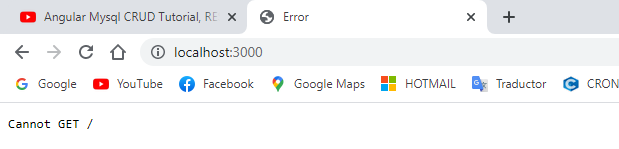
        this.routes();

    }

En la terminal que esta ejecutando nodemon podemos ver el mensaje en consola que el servidor esta corriendo por el puerto 3000.



Pero si vamos a un navegador y buscamos el localhost con el puerto 3000 nos da un error.



Lo que pasa es que aun no configuramos las rutas y express por defecto si no hay rutas nos devuelve un error.

Ahora vamos a definir las rutas en el método routes, pero estas las tendre en un archivo por separado, vamos a crear el archivo de rutas dentro de src/routes/indexRoutes.ts, src/routes/gamesRoutes.ts.

Estos archivos los debemos importar.

//Routes

import indexRoutes from './routes/indexRoutes';

import gamesRoutes from './routes/gamesRoutes';

estos archivos dan error porque están vacíos, vamos a empezar a definir el indexRoutes.

import { Router } from 'express';

class IndexRoutes {

    public router: Router = Router();

    constructor(){

        this.config();

    }

    config(): void {

        this.router.get('/', (req, res) => res.send('Hello IndexRoutes'));

    }

}

const indexRoutes = new IndexRoutes();

export default indexRoutes.router;

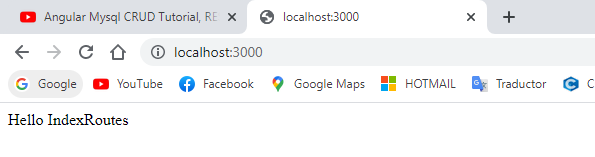
en el método routes del index lo llamamos asi.

routes(): void {

        this.app.use(indexRoutes);

    }

Si vamos al navegador podemos ver que esta ruta esta funcionando.



Ahora vamos a crear el archivo de rutas de games.

import { Router } from 'express';

class GamesRoutes {

    public router: Router = Router();

    constructor(){

        this.config();

    }

    config(): void {

        this.router.get('/', (req, res) => res.send('Hello GamesRoutes'));

    }

}

const gamesRoutes = new GamesRoutes();

export default gamesRoutes.router;

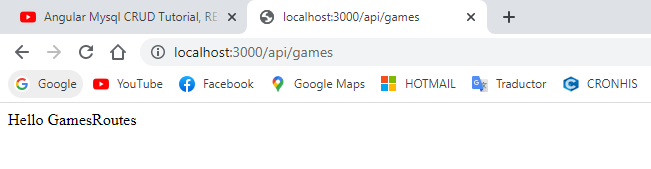
podemos ver que el archivo es muy parecido al del index perohay un cambio al llamar estas rutas, ya que para llamar a los juegos solo se podrá si se espesifica la ruta /api/games.

routes(): void {

        this.app.use(indexRoutes);

        this.app.use('/api/games', gamesRoutes);

    }



Ahora vamos a utilizar las dependencias de Morgan y cors para ello primero debemos importarlas.

import morgan from 'morgan';

import cors from 'cors';

pero tenemos el mismo error ya que no reconoce el tipo de dato para ello vamos a instalarlos.

>>npm i @types/morgan @types/cors -D

Ahora vamos al método config y vamos a utilizar Morgan, este método pide como mínimo un argumento en este caso vamos a enviar el dev.

config(): void {

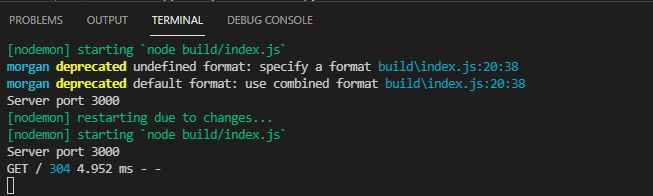
        // si nos dan un puerto en un servicio en la nube se lo toma sino es asi toma el puerto 3000

        this.app.set('port', process.env.PORT || 3000 );

        this.app.use(morgan('dev'));

    }

Si vamos al navegador y recargamos la pagina podemos ver que en la consola que se esta ejecutando el js nos muestra que se ha hecho una petición get.



**UNIR BASE DE DATOS A NUESTRO SERVIDOR**

Vamos a crear nuestra base de datos pero antes debemos verificar si el comando mysql se reconoce en nuestro equipo, en caso de no ser asi hacemos lo siguiente.

1. Abrimos panel de control, sistema y seguridad, sistema, configuración avanzada del sistema.
2. Se abre una ventana donde debemos ir a opciones avanzadas, variables de entorno.
3. Ubicamos la que dice path en variables del sistema y la abrimos.
4. Agregamos la ruta %systemDrive%\xampp\mysql\bin\
5. c:\xampp\mysql\bin\mysql.exe

debemos reiniciar visual studio code y ahora podemos utilizar el comando

>>mysql -u root -p

Una vez ingresamos agregamos el siguiente codigo para crear la base de datos y una tabla

CREATE DATABASE ng\_games\_db;

USE ng\_games\_db;

CREATE TABLE games(

    id INT(11) NOT NULL AUTO\_INCREMENT PRIMARY KEY,

    title VARCHAR(180),

    description VARCHAR(255),

    image VARCHAR(200),

    created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

DESCRIBE games;

Ya tenemos nuestra base de datos creada ahora debemos conectarla, para eso en el src vamos a crear el archivo database.ts.

Ahora en este archivo vamos a importar el módulo que instalamos.

import mysql from 'promise-mysql';

y debemos ejecutar el comando createPool() que es una forma de crear nuestra conexión en modo de producción.

mysql.createPool();

para los parámetros de conexión vamos a crear otro archivo llamada keys.ts

export default {

    database: {

        host: 'localhost',

        user: 'root',

        password: 'admin',

        database: 'ng\_games\_db'

    }

}

En el archivo de databese.ts le pasamos las keys a la conexión

const pool = mysql.createPool(keys.database);

para esto debemos importarlas

import keys from './keys';

ahora le decimos a la constante pool que me cree la conexión, como esto devuelve una promesa utilizamos el then.

pool.getConnection()

    .then(connection => {

      pool.releaseConnection(connection);

        console.log('DB is Connected');

    });

En caso de que no nos detecta el método getConnection ejecutamos el siguiente código

>>npm i [promise-mysql@3.3.1](mailto:promise-mysql@3.3.1)

Ahora solo debemos exportar la constante pool.

export default pool;

Con esto ya tenemos la conexión para poder utilizarla vamos a hacerlo en las rutas pero para poder darle mas orden vamos a crear una carpeta llamada controllers en el src

Ahí creamos un controlador para index y para games y aquí extraemos el código de la respuesta que estaba en las rutas.

El indexController tendrá el siguiente código

import { Request, Response } from 'express';

class IndexController {

   public index ( req: Request, res: Response ) {

       res.send('The API is /api/games')

   }

}

export const indexController = new IndexController();

para utilizarlo en la rutas vamos a importarlo y lo llamamos

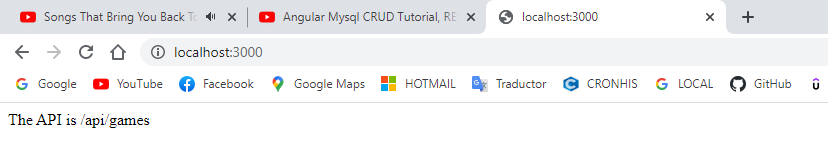
import { indexController } from '../controllers/indexController';

en el método config

config(): void {

        this.router.get('/', indexController.index );

    }



Hacemos lo mismo para la ruta game, en esta parte debemos hacer un llamado a la conexión de la base de datos, podemos ver en consola que nos dice que la conexión esta establecida y en el navegador se pasa la respuesta que le enviamos; El controlador de game queda asi.

import { Request, Response } from 'express';

import pool from '../database';

class GamesController {

   public index ( req: Request, res: Response ) {

       pool.query('DESCRIBE games');

       res.json('games');

   }

}

const gamesController = new GamesController();

export default gamesController;

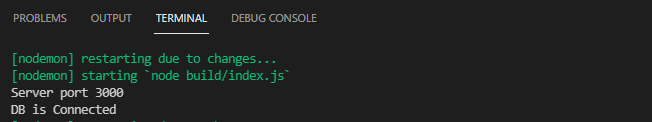
en las rutas del game vamos a importar este controlador y lo utilizamos en el config

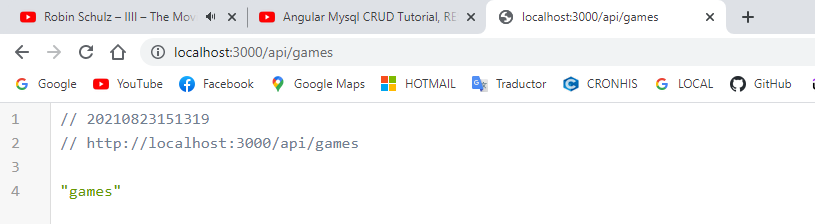
import  gamesController  from '../controllers/gamesController'

config(): void {

        this.router.get('/', gamesController.index );

    }





**CREAR LAS RUTAS DEL SERVIDOR QUE VA A UTILIZAR ANGULAR**

Vamos a crear la ruta para el método post, este método se utiliza para crear un nuevo juego, en este caso es el mismo que el get solo que cambia la petición, lo vamos a comprobar con postman.

config(): void {

        this.router.get('/', gamesController.index );

        this.router.post('/', gamesController.create );

    }

El método créate lo vamos a definir en el controlador de game

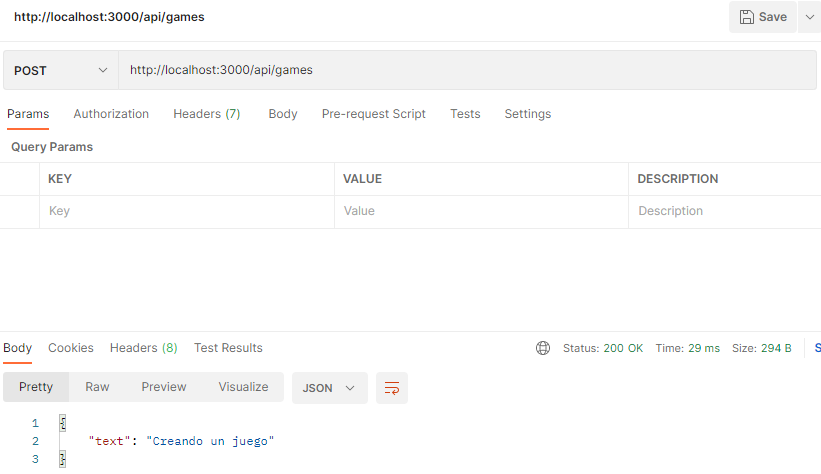
public create (req: Request, res: Response) {

       res.json({

           text:'Creando un juego'

       });

   }



Podemos utilizar postman para verificar que por el método post se ejecuta este método créate.

Ahora vamos a utilizar el método delete que llamara al método delete del controlador del game, este método requiere que se pase por la ruta el id del juego que se desea eliminar.

this.router.delete('/:id', gamesController.delete );

en el controlador creamos este método.

public delete (req: Request, res: Response) {

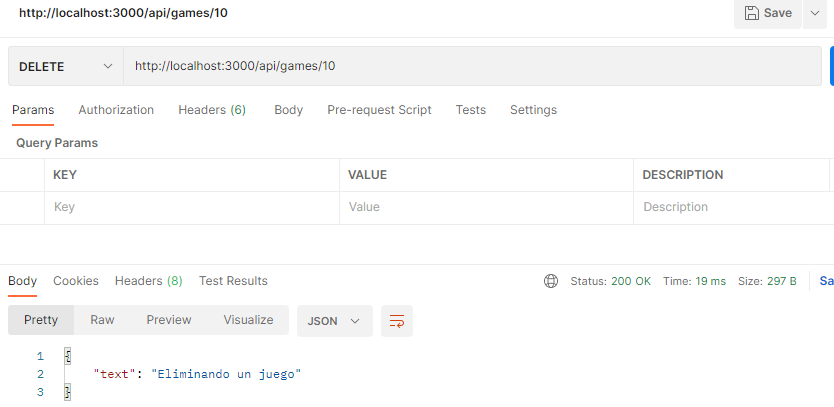
    res.json({

        text:'Eliminando un juego'

    });

   }

Para ejecutarlo con postman debemos enviar el parámetro del id por ahora lo haremos manualmente.



El método de actualizar es parecido al delete solo que este utiliza el método put y hace una llamada al método update del controlador game.

this.router.put('/:id', gamesController.update );

public update (req: Request, res: Response) {

    res.json({

        text:'Actualizando un juego'

    });

   }

Vamos a cambiar el método index del controlador y le llamaremos list ya que este método se encarga de listar todos los elementos de esta tabla.

this.router.get('/', gamesController.list );

public list ( req: Request, res: Response ) {

       res.json({

           text:'Listando juegos'

        });

   }

También se da el caso de que queremos listar solo un juego para ello lo hacemos también con el metodo get pero le pasamos un parámetro.

this.router.get('/:id', gamesController.getOne );

public getOne (req: Request, res: Response) {

    res.json({

        text:'Listando juego con id: ' + req.params.id

    });

   }

El método de configuración del gameRoutes.ts queda así

config(): void {

        this.router.get('/', gamesController.list );

        this.router.get('/:id', gamesController.getOne );

        this.router.post('/', gamesController.create );

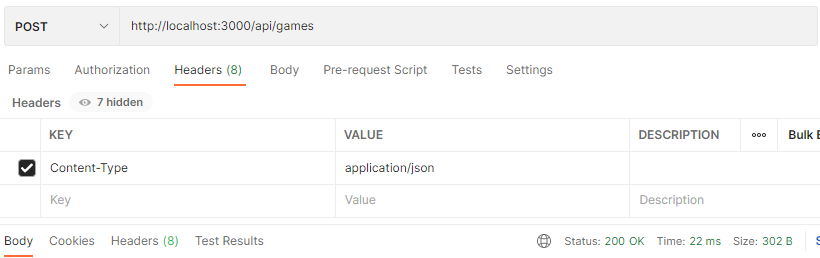
        this.router.put('/:id', gamesController.update );

        this.router.delete('/:id', gamesController.delete );

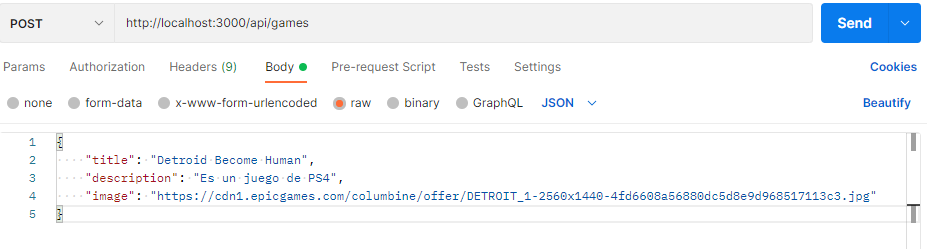
    }

Como podemos observar ya funcionan las rutas y ejecutan los mensajes de consola que están definidos en cada método, ahora necesitamos hacer que estos métodos funcionen con la base de datos.

Primero vamos a crear un nuevo juego para esto utilizamos el postman con la ruta sin parámetros y el metodo post, pero antes debemos configurar el body; en los headers le decimos que es de tipo aplication json.



Ahora en el body levamos a enviar un objeto json, debemos tener en cuenta que el id y la fecha son datos que se generan automáticamente por eso estos no los enviamos atreves del body.



El metodo del controlador quedaría asi.

public async create (req: Request, res: Response): Promise<void> {

       //console.log(req.body);

       await pool.query('INSERT INTO games SET ?',[req.body]);

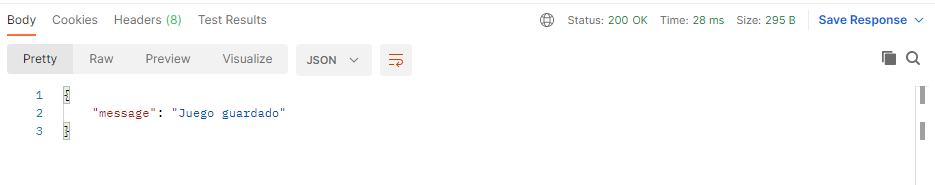
       res.json({

           message:'Juego guardado'

       });

   }

Al enviar el método a través del postman no debe dar errores y se retorna en consola la respuesta.



Podemos ir a la base de datos y verificar que el juego esta guardado.

Podemos cambiar el método list para que me muestre todos los registros almacenados en la base de datos.

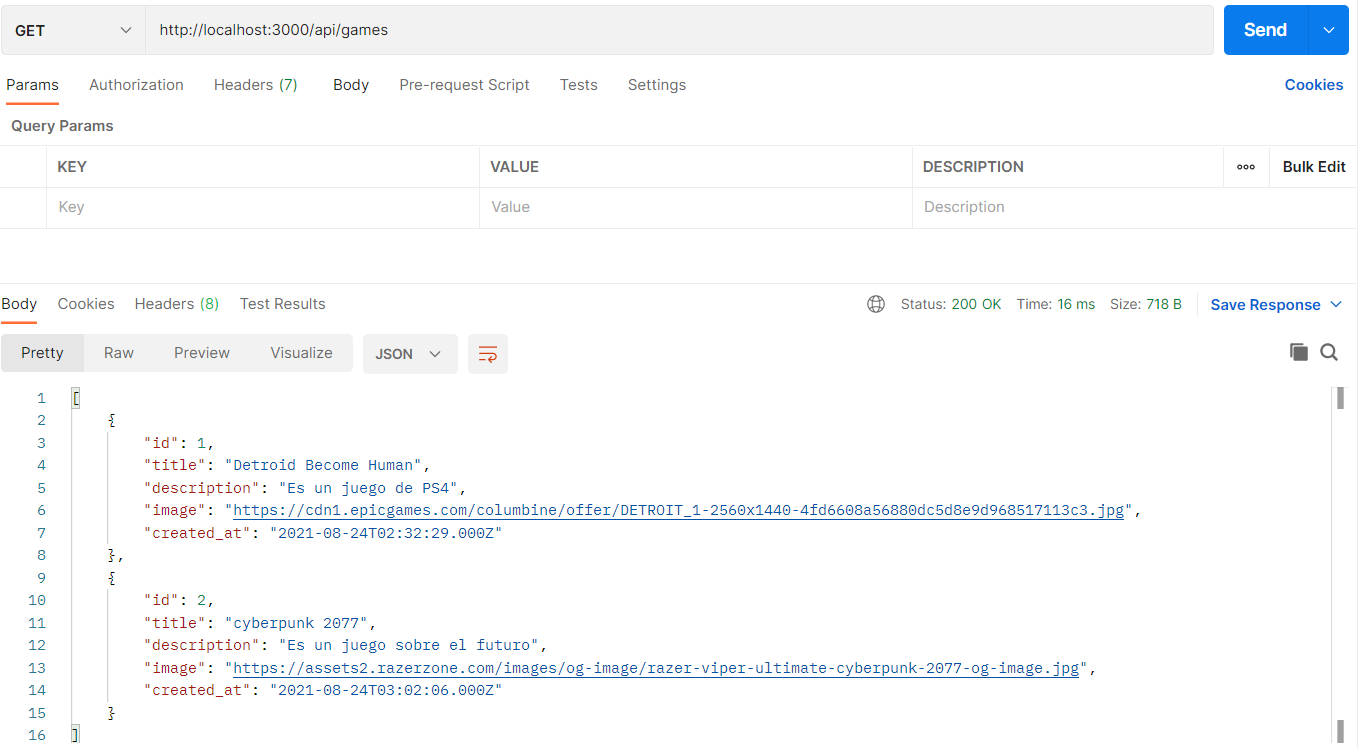
public async list ( req: Request, res: Response ) {

       const games = await pool.query('SELECT \* FROM games');

       res.json(games);

   }

Esto me retorno un arreglo con todos los juegos almacenados en la tabla games.



Ahora vamos a crear el método para buscar un registro en la tabla games, en la url debemos pasar el id del juego que deseamos buscar, lo primero que hacemos es almacenar el id que viene del request en una constante pero vamos a utilizar la desestructuración de js por que solo necesitamos este valor, reste id viene de req.params.

public async getOne (req: Request, res: Response): Promise<any> {

   const { id } = req.params;

   const games = await pool.query('SELECT \* FROM games WHERE id = ?',[id]);

       if( games.length > 0){

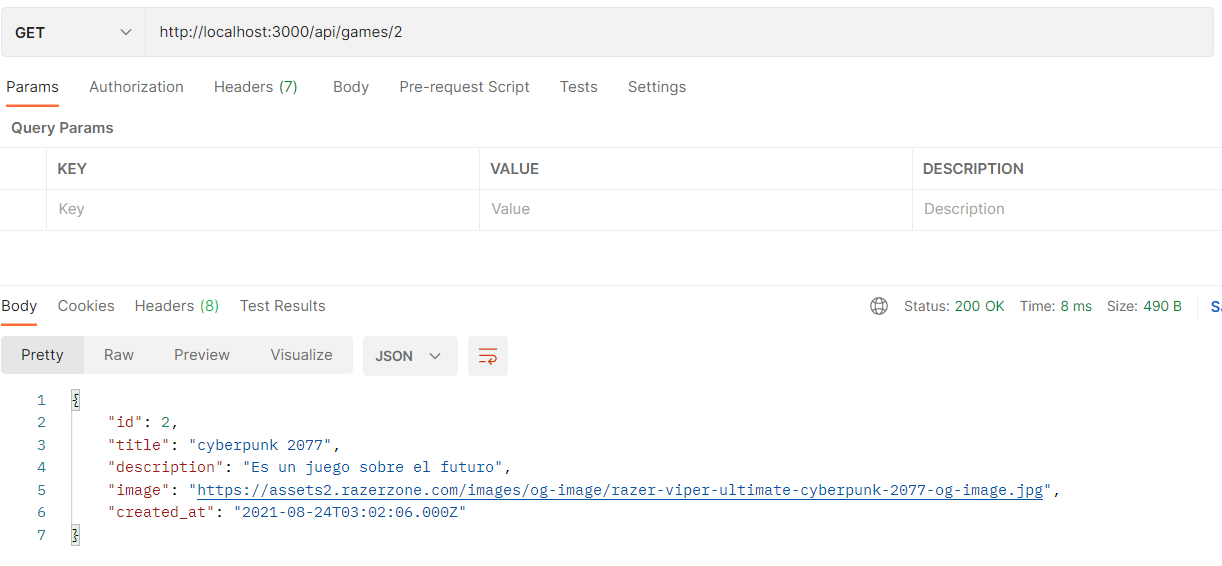
           return res.json(games[0]);

       }

       res.status(404).json({message:'El juego no existe'});

   }

Después debemos validar que la constante games tenga por lo menos un elemento en el arreglo, si es así retornamos una res de tipo json y será la primera posición del arreglo así nos retorna solo un objeto y no un array



Ahora vamos a crear el método delete.

public async delete (req: Request, res: Response): Promise<void> {

       const { id } = req.params;

       await pool.query('DELETE FROM games WHERE id = ?', [id]);

       res.json({message:'juego borrado'});

   }

Y por último el método update este método recibe tanto el id del juego como el body con los campos que se va a actualizar.

public async update (req: Request, res: Response): Promise<void> {

       const { id } = req.params;

       await pool.query('UPDATE games SET ? WHERE id = ?', [req.body, id]);

       res.json({message:'El juego fue actualizado'});

   }