API Rest profesional con MySQL [1]

Preparando el servidor

- Instalar el nodeJS
- luego ejecutar en la carpeta creada para el proyecto ***npm*** *init -y*
- Instalar express con ***npm install express***
- Crear un archivo donde pondremos nuestro código llamado index.js
- Instalar nodemon con ***npm install nodemon*** *-D (“-D” para que lo instale en el apartado Dev de package.json)*
    - Luego agregar la siguiente línea en el script

```
Unset
  "scripts": {
    "dev": "nodemon index.js"}
```

- A partir de acá para ejecutar el proyecto debemos hacerlo con *npm run dev* (llamará al comando que se encuentre en el script “dev”)

Generando los Endpoints

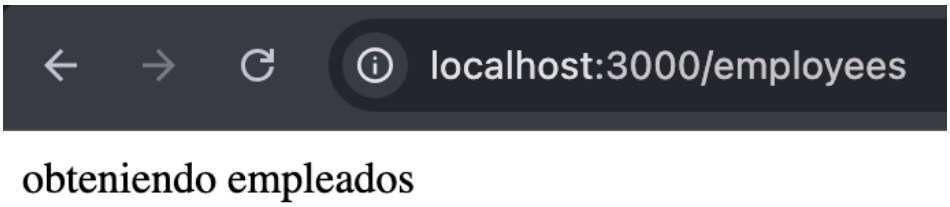
- Generamos los 4 verbos con una respuesta básica en cada uno de ellos en *index.js*:

```
JavaScript
import express from "express"
const app = express()
```

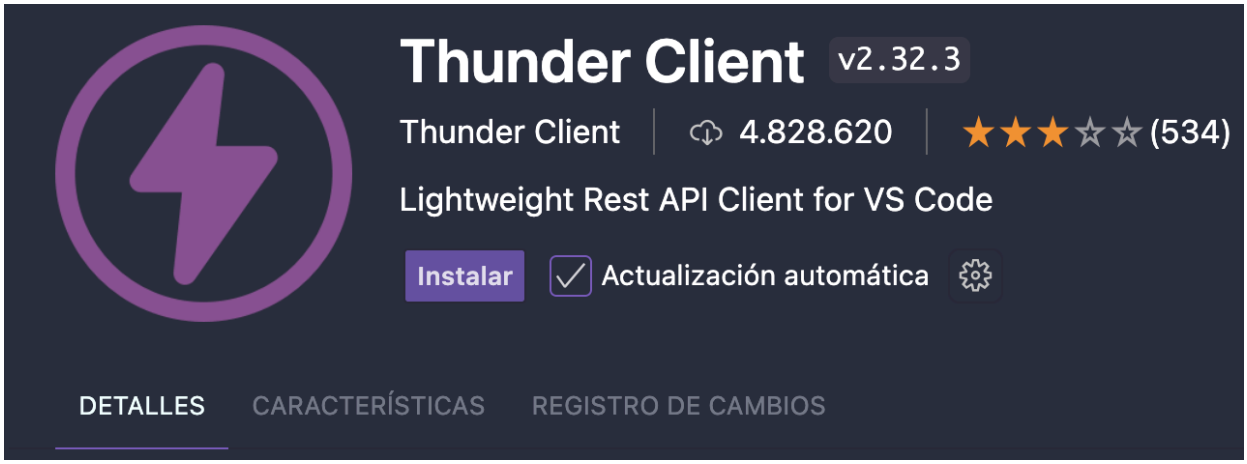
[1] belaunde@gmail.com - video explicativo: https://youtu.be/3dSkc-DIM74

```
app.listen(3000)

app.get('/employees', (req, res) =>res.send("obteniendo empleados"))
app.post('/employees', (req, res) =>res.send("creando empleados"))
app.put('/employees', (req, res) =>res.send(" modificando empleados"))
app.delete('/employees', (req, res) =>res.send("borrando empleados"))

console.log("corriendo servidor en 3000")
```

Probamos…



- Para realizar las pruebas a las peticiones POST, PUT y DELETE podemos utilizar la extensión de VS Code llamada Thunder Client, muy utilizada para pruebas de funcionalidad de API Rest.



Creando la conexión con la base de datos

- Creamos la carpeta llamada db donde pondremos el archivo database.sql:

```
Unset
CREATE DATABASE companydb;

USE companydb;

CREATE TABLE employee (
  id INT(11) NOT NULL AUTO_INCREMENT,
  name VARCHAR(45) DEFAULT NULL,
  salary INT(11) DEFAULT NULL,
  PRIMARY KEY(id)
);

INSERT INTO employee values
  (1, 'Ryan Ray', 20000),
  (2, 'Joe McMillan', 40000),
  (3, 'John Carter', 50000);

SELECT * FROM employee;
```

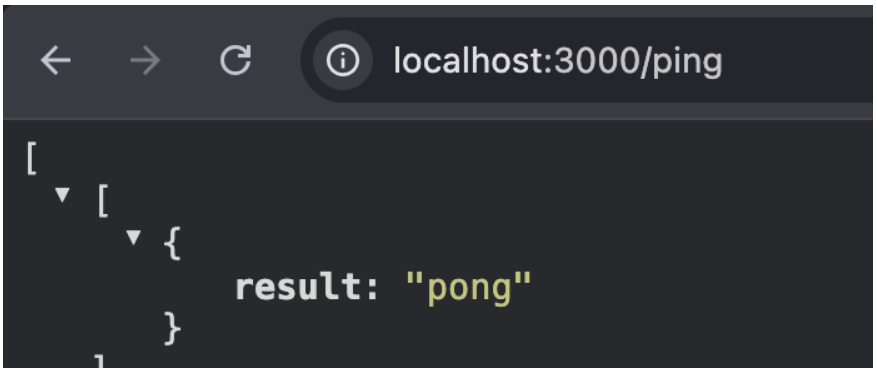
- Instalamos el módulo de MySQL en la versión 2 con *npm i mysql2*
- Luego creamos un archivo para las conexiones a la base de datos llamado db.js donde hacemos uso del módulo mysql2/promise y establecemos la conexión pero usamos la función “pool” en reemplazo de “conection” porque me facilita usar varias conexiones en vez de una y adicionamos “promise” porque es lo más actualizado ya que utilizaremos una función *async* para hacer las querys.

db.js

```
JavaScript
import {createPool} from "mysql2/promise";
export const pool = createPool({
  host: 'localhost',
  user: 'root',
  password: 'root',
  port: '3306',
  database: 'companydb',
});
```

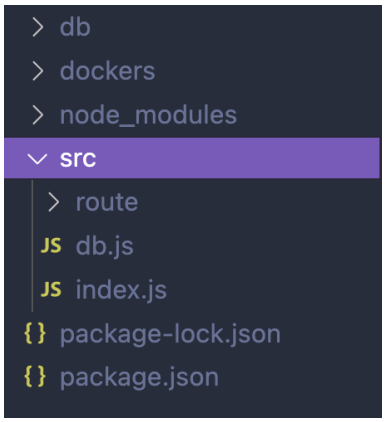
- Ahora para probar si todo está bien agregamos un verbo GET al archivo *index.js* con el siguiente código. De esta forma cuando se acceda a la ruta /ping responderá con la palabra “pong”.

```JavaScript
app.get('/ping', async(req, res) => {
const result = await pool.query('SELECT "pong" AS result')
res.json(result)
})
```



Creando las rutas para mejor organización y gestión del proyecto

- Creamos una carpeta llamada *src* donde colocaremos todo el código del proyecto y dentro creamos una carpeta llamada *route*



  - Cambiamos de nuevo la línea en el script en *package.json* por la nueva ruta de index.js

```Unset
  "scripts": {
    "dev": "nodemon src/index.js"}
```

- Dentro de la carpeta route creamos 2 archivos, employees.routes.js y index.routes.js donde pondremos las rutas que ya generamos pero con el módulo “route”.
- Luego cargamos las rutas en ambos archivos de la siguiente manera:

employees.routes.js

```javascript
import { Router } from "express";
const router = Router()

router.get('/employees', (req, res) =>res.send("obteniendo empleados"))

router.post('/employees', (req, res) =>res.send("creando empleados"))

router.put('/employees', (req, res) =>res.send(" modificando empleados"))

router.delete('/employees', (req, res) =>res.send("borrando empleados"))

export default router
```

index.routes.js

```javascript
import { Router } from "express";
import {pool} from "../db.js"

const router = Router()

router.get('/ping', async(req, res) => {
    const result = await pool.query('SELECT "pong" AS result')
    res.json(result)
    })

export default router
```

- Luego de esto debemos actualizar el archivo index.js porque hemos sacado los verbos para colocarlos en las rutas. A cambio en el index debemos hacer referencia a estos archivos e importarlos.

index.js

```javascript
import express from "express"
import employeesRoutes from "./route/employees.routes.js";
import indexRoutes from "./route/index.routes.js"
```

```
const app = express()

app.listen(3000)

app.use(employeesRoutes)
app.use(indexRoutes)

console.log("corriendo servidor en 3000")
```

- Por último controlar que todo esté funcionando correctamente luego de estos cambios.

Agregamos controladores para las funciones que acceden a datos

- Creamos una carpeta llamada controllers y dentro un archivos employees.controller.js y index.controller.js donde pondremos las funciones que se utilizaban en las rutas (routes) y creamos objetos para importarlos en el archivo index.routes.js y employees.routes.js. De esta forma el código quedará mucho más legible.

employees.controller.js

```
JavaScript
export const getEmployees = (req, res) =>res.send("obteniendo empleados")

export const createEmployee = (req, res) =>res.send("creando empleados")

export const updateEmployee = (req, res) =>res.send(" modificando empleados")

export const deleteEmployee = (req, res) =>res.send("borrando empleados")
```

index.controller.js

```
JavaScript
import {pool} from "../db.js"

export const ping = async(req, res) => {
    const result = await pool.query('SELECT "pong" AS result')
    res.json(result)
    }
```

- Y de esta forma el archivo employees.routes.js queda de la siguiente manera con los objetos importados del controlador (controller).

employees.routes.js

```javascript
import { Router } from "express";
import {getEmployees, createEmployee, updateEmployee, deleteEmployee} from
"../controllers/employees.controller.js";

const router = Router()

router.get('/employees', getEmployees)
router.post('/employees', createEmployee)
router.put('/employees', updateEmployee)
router.delete('/employees', deleteEmployee)

export default router
```

Generamos las acciones para la petición POST de empleados

- Antes que nada debemos agregar una línea en el archivo index.js para que toda información que llegue por “request” sea interpretada en formato json. Para esto hacemos uso de la función express.json().

index.js

```javascript
app.use(express.json()) //debe colocarse antes que cualquier linea de app.use

app.use(employeesRoutes)
app.use(indexRoutes)
```

- Luego se debe modificar la función createEmployee agregando la consulta SQL para insertar los datos que llegan por el body de la petición POST.. Al tratarse de una consulta a una base de datos debe ser asincrónica, para ello utilizamos async y await.

employees.controller.js

```javascript
export const createEmployee = async (req, res) =>{
    const {name, salary} = req.body
```

```
    const [rows] = await pool.query('INSERT INTO employee (name, salary) VALUE
(?, ?)',[name, salary])
    res.send(rows)
}
```

- Una vez hecha la inserción se responde con “toda” la información que devuelve la base de datos incluyendo el ID del registro ingresado. Por ejemplo esto devolvería por medio del sistema postman (o extensión Thunder Client de VS Code) una petición POST:

```
{
    "fieldCount": 0,
    "affectedRows": 1,
    "insertId": 4,
    "info": "",
    "serverStatus": 2,
    "warningStatus": 0,
    "changedRows": 0
}
```

Generamos las acciones para la petición GET de empleados

- De la misma manera que lo hicimos con post, vamos a modificar la función getEmployees para hacer una consulta SQL que me devuelva todas las filas que hay en la tabla “employee”.

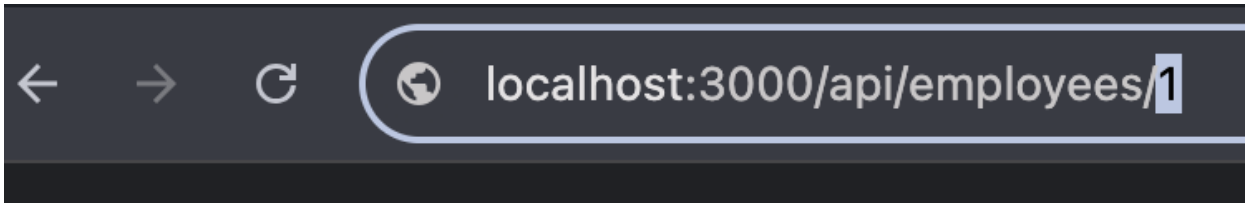
employees.controller.js

JavaScript

```
export const getEmployees = async (req, res) =>{
    const [rows] = await pool.query('SELECT * FROM employee')
    res.send(rows)
}
```

Generamos las acciones para la petición GET de un solo empleado (ID)

- Creamos una nueva función que será utilizada cuando realicen una petición a la ruta “/employees” pero sumándole un parámetro que será el ID que buscaremos para devolver como respuesta.



- También tenemos que sumar una nueva ruta donde indicaremos que permitiremos que nos ingrese un valor en la url '/employees/:id' . Este se sumará al archivo employees.routes.js

employees.routes.js

```
JavaScript
router.get('/employees/:id', getEmployee)
```

- Luego tendremos que sumar toda la lógica para que haga la búsqueda en la tabla por medio del ID recepcionado con el parámetro “req.params.id”. En caso que esa búsqueda resulte nula (vacía) responderemos que un error 404 y mensaje de empleado no encontrado. Caso contrario, la API responderá con los datos del registro devuelto por la consulta SQL.

employees.controller.js

```
JavaScript
export const getEmployee = async (req, res) => {
    const id = req.params.id
    const [rows] = await pool.query('SELECT * FROM employee WHERE id = ?',
[id])

    if (rows.length <= 0) return res.status(404).json({ mensaje: "id no
encontrado" })

    res.json(rows)
}
```

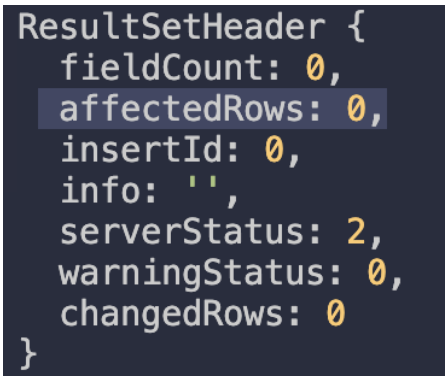
Generamos las acciones para la petición DELETE de empleados

- Primero vamos a modificar la ruta de la petición “delete” para que permita parámetros porque la eliminación la haremos por medio del id del empleado en la tabla “employee”.

employees.routes.js

```javascript
router.delete('/employees/:id', deleteEmployee)
```

- Luego modificamos la función para que se pueda hacer una consulta SQL de eliminación. Esta consulta devuelve un arreglo con determinada información que solo nos interesará el valor “affectedRows”, en el caso que sea 0 no habrá eliminado nada y asumimos que el ID no fue encontrado.



- En caso que se haya eliminado un registro devolverá 1 o más, por eso utilizaremos del arreglo “result” el objeto affectedRows, “result.affectedRows” para confirmar con el IF. En ese caso responde con el código http “204” que representa que todo salió bien y que no recibirá info como respuesta.

employees.controller.js

```javascript
export const deleteEmployee = async (req, res) => {
    const id = req.params.id
    const [result] = await pool.query('DELETE FROM employee WHERE id =?', [id])

    if (result.affectedRows <= 0) return res.status(404).json({
        mensaje: "empleado no encontrado"
```

```
    })

    res.sendStatus(204) //codigo todo OK pero sin devolver info
}
```

Generamos las acciones para la petición PATCH de un empleado (ID)

- Primero debemos modificar el verbo PUT por el PATCH para que se permita actualizar todos los datos o solo algunos de la base por medio de una petición patch. Para ello modificamos el archivo employees.routes.js.

employees.routes.js

```
JavaScript
router.patch('/employees/:id', getEmployee)
```

- Debemos modificar la función updateEmployee para que se pueda actualizar solo los datos recibidos por el body siempre y cuando coincida con el id recibido por parámetro. Para que el script de SQL sea optimizado y actualice solo los datos recibidos y no todos (evitamos el riesgo de no recibir alguno de ellos y que la tabla se actualice con datos NULL), utilizamos la función SQL llamada IFNULL que permite **confirmar el mismo valor** en caso de recibir un null o de lo contrario modificarlo. Con esto solo actualizaremos todos o algunos de los datos.
  Por otro lado, una vez actualizado, volvemos a realizar una llamada select solo para responder con los datos de la tabla ya actualizados.

employees.controller.js

```
JavaScript
export const updateEmployee = async (req, res) => {
    const { id } = req.params // es lo mismo que "const id = req.params.id"
    const { name, salary } = req.body // recupero del body solo estos 2 datos

    const [result] = await pool.query('UPDATE employee SET name = IFNULL(?,
name), salary =IFNULL(?, salary) WHERE id =?',
        [name, salary, id])

    if (result.affectedRows <= 0) return res.status(404).json({
        mensaje: "empleado no encontrado"
    })
```

```
    const [rows] = await pool.query('SELECT * FROM employee WHERE id=?', [id])
    res.json(rows[0])
}
```

Manejo de errores dentro del API rest

- Para hacer un primer acercamiento al manejo de errores haremos uso de la declaración de Try….Catch. Su definición dice lo siguiente: *“La declaración try...catch señala un bloque de instrucciones a intentar usar (**try**), y especifica una respuesta si se produce una excepción o error (**catch**)”.* Por eso podemos colocar cada una de las funciones del controlador para ajustarla a esta estructura. En caso de error responder con un estado 404 y un mensaje. Por ejemplo la función getEmployees quedaría de la siguiente manera:

employees.controller.js

```javascript
export const getEmployees = async (req, res) => {
    try {
        const [rows] = await pool.query('SELECT * FROM employee')
        res.json(rows)
    } catch (error) { res.status(404).json({ mensaje: "error inesperado" }) }
}
```

Responder a una solicitud/página no encontrada (Not found)

- Si ninguna de las rutas llegara a corresponder a la petición realizada podremos una línea más con app.use para que devuelva un mensaje con el estado 404. Por lo tanto agregar las siguientes líneas por debajo de los primeros 2 middleware.

index.js

```javascript
app.use((req, res, nex)=>{
    res.status(404).json({
        mensaje:"endpoint no encontrado"
```

```
    })
})
```

- Este código asegura que si un usuario intenta acceder a un endpoint no definido (por ejemplo, /no-existe), en lugar de devolver un error genérico o vacío, la aplicación enviará una respuesta clara y controlada indicando que el endpoint no existe

Manejo de variables de entorno

- Antes que nada debemos instalar el módulo para el manejo de variables de entorno llamada dotenv. con ***npm install dotenv*** Luego estaremos en condiciones de trabajar con este tipo de variables muy importantes para usarlas en producción con los datos propios del entorno de producción con ser nombre de servidor, puerto, etc.
- Creamos un archivo .env en la carpeta raiz donde colocaremos todas las variables.

.env

```
Unset
PORT=3000
DB_HOST=localhost
DB_USER=root
DB_PASSWORD=root
DB_DATABASE=companydb
DB_PORT=3306
```

- Para que esta información quede aislada y podemos hacer uso del Operador lógico (evalúa dos valores y devuelve el primero que sea verdadero. Si el primer valor es falso, devuelve el segundo), en un archivo aparte llamado config.js. De esta forma tomamos los valores cargados en las variables de entorno y en caso de no tener valor se asigna uno por defecto, como lo siguiente:

config.js

```
JavaScript
import { config } from "dotenv";
config();

export const PORT = process.env.PORT || 3000;
export const DB_HOST = process.env.DB_HOST || "localhost";
export const DB_USER = process.env.DB_USER || "root";
export const DB_PASSWORD = process.env.DB_PASSWORD || "root";
```

```
export const DB_DATABASE = process.env.DB_DATABASE || "companydb";
export const DB_PORT = process.env.DB_PORT || 3306;
```

- En el archivo db.js modificamos lo siguiente, incorporando la importación de las constantes expuestas de config.js.

config.js

```
JavaScript
import { createPool } from "mysql2/promise";
import {DB_DATABASE, DB_HOST ,DB_PASSWORD ,DB_PORT ,DB_USER} from
"./config.js";

export const pool = createPool({
  host: DB_HOST,
  user: DB_USER,
  password: DB_PASSWORD,
  port: DB_PORT,
  database: DB_DATABASE,
});
```

Separando el código para ser más simple de leer

- Para que el código quede más simple vamos a dividir el contenido del archivo index.js en dos. Dejaremos en este solo las llamadas la ejecución del servidor en un puerto determinado y la escritura del mensaje con este dato en el servidor.

index.js

```
JavaScript
import {PORT} from "./config.js";
import app from "./app.js";

app.listen(PORT)

console.log("corriendo servidor en ", PORT)
```

- Luego pondremos el uso de las rutas en un archivos llamado app.js donde estará el resto del código sacado del archivo index.js. por último exportamos la constante app para sea importada desde index.

app.js

```javascript
import employeesRoutes from "./route/employees.routes.js";
import indexRoutes from "./route/index.routes.js"

const app = express()
app.use(express.json()) //interpreta toda info que llega por request en formato
json

app.use('/api', employeesRoutes) //le agrego el prefijo "/api" para todos los
verbos de employees
app.use(indexRoutes)

app.use((req, res, nex)=>{      //middleware para manejar el error
    res.status(404).json({
        mensaje:"endpoint no encontrado"
    })
})

export default app;
```

Próximos pasos…

- Deployar el API en un servicio gratuito como lo es Railway en https://railway.app
- Analizar la posibilidad de estudiar React y aplicarle un front end a las respuestas de la API rest
    - https://www.youtube.com/watch?v=rLoWMU4L_qE&t=0s
    -