TEMA 1 INTRODUCCIÓN A NODE.JS

Parte V – Introducción a los Servicios REST

5.3 EJEMPLO API REST

La Base de Datos y los métodos básicos

- Aprovecharemos el código ya realizado en el punto 4. Pero en el index.js principal en vez de hacer las llamadas de prueba a cada cosa, realizaremos un Servidor básico.

```
const http = require('http');

const modelo = require('./modelo');

let atenderPeticion = (request, response) => {
// Código por implementar
};

http.createServer(atenderPeticion).listen(8080);
```

Peticiones GET

- Vamos a realizar dos tipos de consulta:
  - Listado de todos los libros, con la URI /libros. Llamaremos a “listarLibros”
  - Obtener un solo libro a partir de un id. Con la URI /libros/id. Llamaremos a “buscarLibroPorId”.
- Para modularizar el server usaremos unos métodos auxiliares que llamen a los métodos y devuelvan en JSON. TODO DEBE DEVOLVER UN JSON AL CLIENTE.

```
let listarLibros = () => {
    return JSON.stringify(modelo.libros.listarLibros());
}

let atenderPeticion = (request, response) => {
    response.writeHead(200, {"Content-Type": "text/plain"});
    if (request.method === 'GET') {
            if (request.url === '/libros') {
                    response.write(listarLibros());
            }
    }
    response.end();
};
```

EJERCICIO 1

- **Añade a continuación la otra petición GET para obtener los datos de un libro, dado su *id* en la propia URI (mirar el documento 2.3 si no te acuerdas). Recuerda exportar el método para obtener un libro dado su *id*, para poderlo utilizar desde fuera del modelo.**

Añadir información complementaria

- Normalmente a parte de devolver los datos, también necesitamos si ha ido todo bien o no. Si ha habido errores. Por lo tanto se suele encapsular un objeto de datos dentro de otro con información extra. Por ejemplo:

```
{
    error: false,
    mensajeError: "",
    libro:
    {
        id: 1,
        titulo: "El juego de Ender",
        autor: "Orson Scott Card",
        precio: 9.9
    }
}
```

EJERCICIO 2

- **Añade esta "envoltura" a los datos devueltos para el libro del ejercicio anterior.**

Peticiones POST

- Es una petición de inserción de datos.
- Suelen ser URI de dos modos:
  - Query-string: parametro=valor&parametro=valor...
  - Otro formato como JSON
- Se puede pasar en JSON, que el Servidor Node lo procese, extraerlo del cuerpo convertirlo a objeto javaScript y añadirlo.

Peticion POST II

- El método recibe un objeto libro como parámetro (extraído de la petición y convertido desde JSON). Como respuesta, envía una cadena JSON con los atributos error y mensajeError similares a los del caso anterior, para indicar si todo ha ido bien o no.

```
let nuevoLibro = (libro) => {
    let resultado = {};
    if (libro.id && modelo.libros.nuevoLibro(libro.id,
    libro.titulo, libro.autor, libro.precio)) {
            resultado.error = false;
            resultado.mensajeError = "";
    } else {
            resultado.error = true;
            resultado.mensajeError = "Error al añadir el nuevo libro";
    }
    return JSON.stringify(resultado);
}
```

Petición POST III

- Este método es llamado desde el caso correspondiente de nuestro *callback* para atender peticiones:

```
let atenderPeticion = (request, response) => {

response.writeHead(200, {"Content-Type": "text/plain"});

if (request.method === 'GET') {
...
} else if (request.method === 'POST' && request.url === '/libro') {
      responseCerrado = true;
      let body = [];
      request.on('data', (chunk) => {
                  body.push(chunk);}).on('end', () => {
                              body = Buffer.concat(body).toString();
                              response.write(nuevoLibro(JSON.parse(body)));
                  });
}
```

Peticion POST III

- Glosario de palabras:
- ‘data’: Esta etiqueta es un evento ya definido en REQUEST (tiene un emit interno), donde indica que el evento se lanza si llegan trozos de datos.
- ‘end’: En este caso indica que el evento se lanza si es el final de datos.
- Chunk: Son fragmentos de datos.
- En este caso hemos visto que hemos metido un método “.on” dentro de otro, porque uno va detrás del otro, pero hubiera funcionado igualmente separándolo en dos.

Petición POST IV

- Notar que en este caso estamos haciendo un procesamiento asíncrono (procesamos el cuerpo de la petición hasta terminar de leerlo, y entonces llamamos al método auxiliar para insertar el libro).
- Por lo tanto, la instrucción response.end() del final nos puede dar problemas (se ejecutará antes de que finalice esta llamada asíncrona, cerrando el flujo de datos cliente-servidor).
- Debemos corregir esto de algún modo (se deja como ejercicio complementario).

EJERCICIO III

- **Nos faltan por implementar los otros dos comandos típicos de toda API REST:**
- **El comando PUT para hacer modificaciones. En este caso, pasaríamos los datos de un libro y el *id* del libro a modificar. Para simplificar las cosas, vamos a suponer que el *id* no se puede modificar, con lo que no hace falta pasar dos *id* diferentes (viejo y nuevo). Los datos del libro se pasan de la misma forma que en el comando POST.**
- **El comando DELETE para borrados. Su funcionamiento es similar al comando GET, y podemos utilizar una URI dinámica para indicar el *id* del libro a borrar en ella.**
- **Añade estos dos comandos al servidor, con la URI */libro.* Se emplea la misma URI que para el comando POST, pero como variamos el comando, podemos permitir que el servidor lo trate como una petición diferente.**