Marina López Monterde
Desarrollo Backend
28/04/2024

Servidor Backend

En el siguiente enlace de GitHub se encuentran los diferentes archivos de esta práctica: https://github.com/marinalmont/Servidor-basico

Como se puede observar en el código, el primer paso es importar los módulos *http* y *fs* porque de esta manera podremos crear el servidor HTTP y leer archivos del sistema de archivos. A continuación, definimos una función asincrónica *fetchPokemonData* para que nos pueda devolver una promesa y poder leer el archivo JSON y nos devuelva su contenido también en este formato. En el caso que la promesa no se resolviera, nos saltaría un error.

```
1   const http = require('http');
2   const fs = require('fs');
3
4   const fetchPokemonData = async () => {
5       return new Promise((resolve, reject) => {
6           fs.readFile('./pokedex.json', 'utf-8', (err, data) => {
7               if (err) {
8                   reject(err);
9               } else {
10                  resolve(JSON.parse(data));
11              }
12          });
13      });
14  };
```

Después, tenemos otra función asincrónica llamada *handleRequest*, que en este caso se encarga de manejar las solicitudes HTTP al servidor. Primero de todo, obtenemos los datos del archivo JSON y la función *pokemonData* nos permite devolver el contenido del archivo JSON como un objeto JavaScript. A continuación, decodificamos la URL, la cual queda almacenada en la variable *pathName*, y eliminamos el primer carácter de la URL (‘/’) a través de *substring(1)*. De esta forma, obtenemos el identificador del Pokémon de la URL.

```
16  const handleRequest = async (req, res) => {
17      const pokemonData = await fetchPokemonData();
18      let pathName = decodeURI(req.url);
19      pathName = pathName.substring(1);
20      console.log('Identificador de Pokémon:', pathName);
```

El siguiente paso es declarar una variable que nos permita almacenar el Pokémon que se busque. Así que, siguiendo las pautas de la actividad, que nos pide mostrar toda la información del Pokémon al llamar al servidor seguido del nombre de este en cualquier idioma o de su id, usamos

un *if… else… If* nos ayudará a encontrar el Pokémon en cuestión usando su id, ya que *¡isNaN(pathName)* verifica si *pathName* se trata de un número. *Else*, por otra parte, nos sirve para cuando el identificador no es un número, asumir que se trata de un nombre de un Pokémon y seguidamente con *Object.values(p.name).some()* buscar entre todos los nombres del archivo incluyendo todas las lenguas e ignorando las mayúsculas y minúsculas. Si se encuentra el Pokémon, se guardará en la variable *pokemon*, si no permanecerá como *undefined*.

```
22      let pokemon;
23
24      if (!isNaN(pathName)) {
25          pokemon = pokemonData.find(p => p.id.toString() === pathName);
26      }
27      else {
28          pokemon = pokemonData.find(p =>
29              Object.values(p.name).some(n => n.toLowerCase() === pathName.toLowerCase())
30          );
31      }
```

Para ir acabando, encontramos la respuesta del servidor. Si el Pokémon se encuentra en el archivo, se enviará una respuesta 200 con los datos correspondientes del Pokémon. Por otro lado, si el Pokémon no se encuentra en el archivo, se enviará un error 404 con la frase “El Pokémon no está registrado en la Pokédex”.

```
33      if (pokemon) {
34          const response = {
35              'Tipo': pokemon.type,
36              'HP': pokemon.base.HP,
37              'Ataque': pokemon.base.Attack,
38              'Defensa': pokemon.base.Defense,
39              'Ataque Especial': pokemon.base['Sp. Attack'],
40              'Defensa Especial': pokemon.base['Sp. Defense'],
41              'Velocidad': pokemon.base.Speed
42          };
43          res.writeHead(200, {'Content-type': 'application/json'});
44          res.end(JSON.stringify(response, null));
45      } else {
46          res.writeHead(404, {'Content-Type': 'text/plain'});
47          res.end('El Pokémon no está registrado en la Pokédex.');
48      }
49  };
```

Por último, creamos un servidor HTTP en el puerto 3000 desde el cual podamos manejar las solicitudes a través de la función *handleRequest*.

```
51  const server = http.createServer(handleRequest);
52
53  server.listen(3000, () => {
54      console.log('Escuchando en el puerto 3000');
55  });
```

Dicho esto, durante el proceso de programación me he encontrado con algunas dificultades. La primera y la más destacable es que en un principio mi código no detectaba los caracteres en japonés y chino. En un principio pensé que quizás el problema estaría en el formato *utf-8*, no sabía si los incluía, pero después de buscar en Internet, me he dado cuenta que el problema no radicaba ahí. Hay una función en JavaScript que está pensada para lidiar con este tipo de cuestiones, que es *decodeURIComponent()*. Algunos caracteres especiales en una URL, como pueden ser los caracteres japoneses o los emojis, se codifican con una convención llamada *percent-encoding*, el cual intercala el símbolo *%* con números. Esta función se encarga justamente de devolver estos caracteres a su estado original.

Para finalizar, también me he encontrado con algunos problemas menores con la terminal y la RapidAPI Client, ya que por algún motivo *clear* no era suficiente para volver a ejecutar el archivo JS, tenía que cerrar y volver a abrir una nueva terminal, si no me la detectaba como si todavía estuviera en uso.

Como conclusión, me parece un ejercicio práctico bueno para familiarizarse con Node.js y la creación de servidores HTTP y APIs. Aunque como ya se comentó en clase, su uso tiene algunas limitaciones, como el hecho de que solo se puede consultar desde la propia red local porque no se tiene acceso desde Internet y, por lo tanto, su acceso está restringido a otros desarrolladores.