Javascript S7 | Peticiones

Por fin ha llegado el momento que todos esperábamos, no es otro que aprender a utilizar **AJAX** que es un **enlace entre cliente**, por ejemplo Chrome, **y el servidor**. Esto nos ayudará a enlazar el frontend y el backend de nuestra aplicación. Las **peticiones AJAX** nos **permiten acceder y manipular datos** en el servidor **desde el frontend**.

Antes de nada estaría bien que entendamos la **diferencia** entre **frontend y backend**. Empezamos por el **frontend.**

Frontend es la parte de nuestra aplicación a la que un usuario puede acceder. Esto hace referencia a todas las tecnologías de desarrollo web o lenguje de marcado que corren en el navegador o cliente. Los principales lenguajes son HTML, CSS y Javascript, cabe destacar que a partir de aquí se abre un gran abanico de frameworks y librerías que nos ayudan a realizar nuestras interfaces. Angular, React, PostCss, SASS son algunos de ellos.

Backend dentro de nuestra aplicación es lo que se conoce como servidor, una zona donde el usuario no tiene acceso directo. Desde el backend enviaremos la información solicitada, utilizando Javascript en el backend encontramos node.js y en caso de utilizar el tipado con typescript deno.js

Javascript asíncrono

Javascript por defecton corre en un único *hilo*, es decir, las instrucciones se van ejecutando una detrás de otra y una instrucción no puede ejecutarse hasta que la anterior ha terminado.

Un código asíncrono no es más que un código que empieza *ahora* y es capaz de abrir *hilos* de ejecución en paralelo de tal forma que hay varias partes del código ejecutándose a la vez. Es posible (o no) que pasado el tiempo vuelvan a *juntarse*.

Promesas

Una `Promise` (promesa en castellano) es un objeto que representa la terminación o el fallo de una operación asíncrona. ¿Muy parecido a los callbacks no? Es una manera sencilla de implementar código asíncrono.

Lo más habitual es que *consumamos* promesas ya creadas, pero primero vamos a ver cómo se crea una promesa.

```
let promise = new Promise(function (resolve, reject) {
  // El ejecutor se ejecuta automáticamente cuando se// construye la promesaconsole.log("EXECUTED EXECUTER");

  // Pasado 1 seg estamos resolviendo la promesa con el // valor "done"
  setTimeout(() => resolve("done"), 1000);
});

// Arrow functions

let promise = new Promise((resolve, reject) => {
  // El ejecutor se ejecuta automáticamente cuando se// construye la promesaconsole.log("EXECUTED EXECUTER");

  // Pasado 1 seg estamos resolviendo la promesa con el // valor "done"
  setTimeout(() => resolve("done"), 1000);
});
```

Hemos visto cómo crear una promesa, pero ya decíamos lo más habitual es consumirlas:

```
promise.then(
  function (result) {
    // Manejamos el resultado en caso de okconsole.log("Resultado!!");
    console.log(result);
  },
  function (error) {
    // Manejamos el resultado en caso de okconsole.log("Resultado!!");
    console.log(result);
  }
);
```

Si nos fijamos, `then(function,function)` recibe dos parámetros: 2 funciones o callbacks.

- La primera función se ejecutará en caso de éxito y nos permitirá manejar la respuesta en caso de OK.
- La segunda función se ejecutará en caso de error y nos permitirá manejar el KO.

Peticiones XHR

Con la salida de ES6 apareció `fetch`. `Fetch` nos permite hacer peticiones AJAX de una manera más cómoda haciendo uso de promesas.

```
fetch('https://pokeapi.co/api/v2/pokemon/')
  .then(function (response) {
    return response.json();
  })
  .then(function (myJson) {
    console.log(myJson);
  });

// Arrow functions

fetch('https://pokeapi.co/api/v2/pokemon/')
  .then((response) => {
    return response.json();
  })
  .then((myJson) => {
    console.log(myJson);
  });
```

¿Por qué dos `.then`?

El método .json de response es asíncrono y devuelve también una promesa, así que al hacer `return response.json()` el primer `.then` devuelve una promesa y necesitamos encadenar otro `.then` para resolverla.