UTN FACULTAD REGIONAL CÓRDOBA
INGENIERÍA EN SISTEMAS DE INFORMACIÓN

Desarrollo de Software
Notas de Clase

Apunte 10 - Acceso a recursos externos - Programación Asincrónica

Introducción

Como vimos en el apunte anterior, Nodejs maneja un único hilo donde se ejecuta el ciclo de eventos que permite recibir peticiones y ejecutar acciones para resolver dichas peticiones. Cuando alguna de estas peticiones requiere acceder a un recurso como podría ser un archivo en el sistema de archivos, acceder a una base de datos, o consumir datos de una aplicación externa, Nodejs asume que esa acción puede llevar más tiempo del esperado o bien que pueden surgir demoras en dicho proceso por lo que envía la ejecución del proceso a un hilo secundario administrado por él y se encargará de avisarnos cuando dicho proceso haya terminado.

El presente apunte se enfoca en los mecanismos del lenguaje Javascript para administrar estas llamadas y respuestas en los casos en que no se pueden llevar a cabo las tareas en el hilo principal y necesitan delegarse en un hilo secundario para que tareas que pueden demorar no bloqueen el hilo principal.

En este hilo principal, JavaScript ejecuta el código secuencialmente en orden descendente. Sin embargo, hay algunos casos en los que el código se ejecuta (o debe ejecutarse) después de que ocurra otra cosa es decir que se va a romper esta secuencia esperada en el orden de ejecución del código. Esto se llama **programación asíncrona**, llamamos programación asincrónica al conjunto de herramientas necesarias para lograr escribir código que no va a seguir el orden o secuencia natural.

Resulta que JavaScript es un lenguaje de programación asíncrono. Lo que quiere decir esto es que al ejecutar código JavaScript el hilo de ejecución continuará a pesar de encontrarse en situaciones en las que no obtenga un resultado inmediatamente. Por ejemplo, cuando hacemos el pedido de información a un servidor, la respuesta posiblemente demore un poco. Sin embargo, el hilo de ejecución de JavaScript continuará con las demás tareas que hay en el código.

Los callbacks aseguran que una función no se va a ejecutar antes de que se complete una tarea, sino que se ejecutará justo después de que la tarea requerida se haya completado. Nos ayuda a desarrollar código JavaScript asíncrono y nos mantiene a salvo de problemas y errores.

Un ejemplo practico de esto seria una aplicación web que necesita llenar una tabla de datos, asi que el código hará un pedido al servidor de los datos que necesita llenar. Pero el hilo de ejecución no se detiene asi que ejecutará el código que pinta la tabla en el navegador. Esto se convierte en un problema ya que los datos del servidor llegan después de que la tabla se haya pintado en pantalla, una tabla sin datos evidentemente.

Trabajar con código asíncrono puede tener muchas ventajas pero en casos como este presenta un gran problema. Pues bien, para solucionar esto algunas funciones de JavaScript tienen como parámetro una función (ya vimos que algunas de las formas de definir funciones en Javascript permiten que estas sean

tratadas como datos) y esta función se conoce como callback, a continuación presentamos en concepto de callbacks y sus diferentes aristas.

Callbacks en Javascript

> [!IMPORTANT] Un callback es simplemente una función que se pasa como parámetro a otra función. Un callback en JavaScript es una función que se pasa como argumento a otra función, y que se invoca o "llama de vuelta" en algún punto dentro de la función que la recibe. Los callbacks son una forma de asegurar que cierto código no se ejecute hasta que otro código haya terminado de ejecutarse, es decir, se utilizan para manejar la asincronía en JavaScript.

Por ejemplo, si tienes una función que carga datos de un servidor (una operación que puede tomar tiempo), puedes pasar una función callback que se ejecutará una vez que los datos se hayan cargado completamente. De esta manera, puedes asegurarte de que no intentas trabajar con los datos hasta que estén completamente cargados.

En una visión un poco más profunda: en Js, **las funciones son objetos**. Por ello, las funciones admiten otras como argumentos y pueden ser devueltas por otras funciones.

Las funciones que hacen esto se denominan funciones de orden superior (high-order). Cualquier función que se pase como argumento se denomina función **Callback**.

En JS es posible construir funciones que usen callbacks de la siguiente manera.

```
function imprimir(callback){
    callback();
}

imprimir(function() {
    console.log('Texto impreso!');
});
```

En el ejemplo anterior la función **callback** se pasa en forma anónima a la función imprimir() que es la encargada de invocarla. Otra forma de verlo sería:

```
function imprimir(callback){
    callback();
}

const imprimirConsola = function() {
    console.log('Texto impreso!');
};

imprimir(imprimirConsola);
```

Ahora bien, este mecanismo por si solo no produce asincronía puesto que el orden de ejecución más allá de la forma de codificarlo es secuencial, es más, este mecanismo podría ser más asociado a polimorfismo

funcional que a asincronía, puesto que la función `imprimir`, recibe la función que realiza la impresión por lo que si además de `imprimirConsola` tuviéramos otra función `imprimirArchivo` podríamos usar la función `imprimir` enviado el modo en que queremos imprimir según corresponda.

¿Qué hace falta entonces para que sí haya asincronía? Hace falta un proceso que provoque una espera o demora en el hilo principal, la más elemental y simple función de Javascript para demorar esto es la función `setTimeout` que vemos a continuación.

Función setTimeout()

Una función muy conocida en Js que tiene un callback es **setTimeout()**. Esta función ejecuta el callback después de esperar cierto tiempo el cual también le pasamos como parámetro.

La forma general de la función `setTimetout` es: `setTimeout(funcionCallback, milisegundos)` por lo que la función va a hacer una pausa por tantos milisegundos como indique la variable `milisegundos` y luego de ello va a ejecutar la `funcionCallback` llevando a cabo la tarea que allí se indique.

```
// setTimeout(funcionCallback, tiempoMs);
setTimeout(function(){
    console.log('Hola');
}, 2000);
```

Esta función ejecuta el callback solo después de que hayan pasado 2000 milisegundos. Con ayuda de esta función podemos crear un código que nos permita visualizar la asincronía de JavaScript.

```
// Usando función anónima:
// Imprimo A
console.log('A');

// Luego ejecuto setTimeout con un callback que imprime B pero luego de 2 segundos
setTimeout(function () {
    console.log('B');
}, 2000)

// Al final imprimo C
console.log('C');

/* Resultado del código anterior
 A
 C
 B  => Después de 2 segundos
*/
```

> [!NOTE] Sería natural esperar que las letras se impriman en orden, de hecho si no conociéramos de asincronía ni lo que venimos conversando de la función `setTimeout` esperaríamos que el resultado sean las letras A, B Y C en ese orden.

Sin embargo, la asincronía de JS nos permite visualizar en consola la letra C mucho antes de la B, el cual demora 2 segundos. Como se puede ver la ejecución no se detiene dos segundos, esta detención genera el hilo secundario pero devuelve el control del hilo principal para que pueda continuar, este continua imprimiendo C y luego de dos segundos, cuando termina la tarea secundario y se ejecuta el callback, aparece C.

Un ejemplo simple: Aquí tienes un ejemplo simple de cómo se puede usar un callback en JavaScript:

function saludo(nombre) { console.log('Hola ' + nombre); }

function procesarEntradaUsuario(callback) { let nombre = prompt('Por favor ingresa tu nombre.'); callback(nombre); }

procesarEntradaUsuario(saludo);

En este ejemplo, procesarEntradaUsuario es una función que toma un callback como argumento. Dentro de procesarEntradaUsuario, se solicita al usuario que ingrese su nombre y luego se llama al callback, pasando el nombre ingresado como argumento.

La función saludo es la que se pasa como callback. Recibe un nombre como argumento y lo imprime en la consola.

Cuando se llama a procesarEntradaUsuario(saludo), se solicita al usuario que ingrese su nombre y luego se imprime un saludo personalizado en la consola.

Para tener un cierto control en el código asíncrono de JavaScript existe un concepto muy interesante: **las promesas**.

Promesas

Una promesa es el objeto que representa la respuesta de una tarea asíncrona, es decir la respuesta a una tarea que de antemano no es posible saber cuándo terminará y se obtendrá su resultado. Por esta razón se deja preparado dentro de la promesa el código que se ejecutará cuando el resultado llegue o incluso cuando el resultado es un error. Un ejemplo típico es la obtención de respuesta a un pedido a una aplicación externa, en general estas comunicaciones a aplicaciones externas las hacemos del mismo modo que cuando navegamos una página web usando el protocolo HTTP y por ello decimos que es un pedido HTTP, que luego analizaremos en detalle.

Las promesas como mecanismo propio de JS llegan en la versión 6 del estándar ECMAScript, hasta entonces era necesario utilizar librerías (como JQuery) o frameworks externos para poder incorporarlas.

Toda promesa puede tener 4 estados:

- Pendiente: Es su estado inicial, no se ha cumplido ni rechazado.
- Cumplida: La promesa se ha resuelto satisfactoriamente.
- Rechazada: La promesa se ha completado con un error.
- Arreglada: La promesa ya no está pendiente. O bien se ha cumplido, o bien se ha rechazado.

Es posible crear una promesa de la siguiente manera.

```
let  x = 11;
const p = new Promise((resolve, reject) =>{
    if(x == 10){
        resolve('La variable es igual a 10');
    }else{
        reject('La variable no es igual a 10');
    }
});
```

La función flecha del código anterior recibe dos parámetros, el primero **resolve** es una función que recibe como parámetro el objeto que queremos que devuelva cuando el código tuvo el resultado que esperamos. Mientras que **reject** es una función que toma como parámetro el objeto que devolverá si existe un error en el código asincrónico.

En resumen, usando una promesa es posible recibir el resultado que se necesita de una espera y ejecutar código luego de que el resultado llegue. Una forma sencilla de probar el uso de una promesa es mediante la función **setTimeout** mencionada anteriormente.

```
let mensaje = new Promise((resolve, reject)=>{
    setTimeout(function () {
        resolve('Este es el mensaje');
    }, 2000);
});
```

De esta forma se crea un objeto promesa con un mensaje como resultado favorable que se devolverá luego de 2 segundos. Ahora para controlar la promesa se utilizan los métodos: **then** y **catch** que vienen junto con las promesas.

```
mensaje.then(m =>{
    console.log(m);
}).catch(function () {
    console.log('error');
});
```

Entonces para capturar el resultado favorable de la promesa se usa then(). En este caso se trata del mensaje que se muestra usando un console.log(). Por otra parte el catch captura el resultado fallido o reject de la promesa. En este caso no implementamos ningún código específico, por lo que sencillamente se ejecuta la función anónima que muestra la palabra *error* por consola.

A trabajar !!=>

A continuación proponemos realizar un programa para procesar un archivo de texto separado por comas CSV como los utilizados en primer año para contener datos estructurados en forma de tabla en texto plano.

En el archivo `tbbt.csv` tenemos los datos de todas las temporadas con sus episodios de la serie The Big Ban Theory, el CSV cuya fuente es el sitio de interés IMDB, tiene el siguiente formato:


```
season,episode_num,title,original_air_date,imdb_rating,total_votes,desc
1,1,Unaired Pilot,1 May 2006,6.7,2048,"The first Pilot of what will become ""The
1,2,Pilot,24 Sep. 2007,8.2,6135,"A pair of socially awkward theoretical physicis
1,3,The Big Bran Hypothesis,1 Oct. 2007,8.3,4924,Penny is furious with Leonard a
1,4,The Fuzzy Boots Corollary,8 Oct. 2007,7.7,4392,"Leonard gets upset when he c
1,5,The Ficus Effect,15 Oct. 2007,8.1,4434,Sheldon's mother is called to
1,6,The Hamburger Postulate,22 Oct. 2007,7.9,4227,"Leslie seduces Leonard, but a
1,7,The Middle Earth Paradigm,29 Oct. 2007,8.4,4402,"The guys are invited to Per
1,8,The Dumpling Paradox,5 Nov. 2007,8.1,4143,"When Howard hooks up with Penny's
1,9,The Grasshopper Experiment,12 Nov. 2007,8.2,4172,"When Raj's parents set him
1,10,The Cooper-Hofstadter Polarization,17 Mar. 2008,8.0,4024,"Leonard and Shelc
1,11,The Loobenfeld Decay,24 Mar. 2008,8.0,4119,"Leonard lies to Penny so that H
1,12,The Pancake Batter Anomaly,31 Mar. 2008,8.2,4094,"When Sheldon gets sick, L
1,13,The Jerusalem Duality,14 Apr. 2008,7.9,3902,"Sheldon decides to give up his
1,14,The Bat Jar Conjecture,21 Apr. 2008,8.3,4015,"Sheldon becomes so intent on
1,15,The Nerdvana Annihilation,28 Apr. 2008,8.1,3950,"Penny gets mad at the guys
1,16,The Pork Chop Indeterminacy,5 May 2008,8.2,4153,"Leonard, Howard and Raj fi
1,17,The Peanut Reaction,12 May 2008,8.3,3914,"When Penny learns that Leonard ha
1,18,The Tangerine Factor,19 May 2008,8.5,4101,"After a bad breakup, Penny final
2,1,The Bad Fish Paradigm,22 Sep. 2008,8.2,3990,"Leonard becomes concerned when
2,2,The Codpiece Topology,29 Sep. 2008,8.1,3680,Sheldon is annoyed when Leonard
```

Y lo que vamos a necesitar ahora es comenzar con un proyecto para llevar a cabo el proceso del archivo para transformarlo en un array de objetos en memoria para luego comenzar a trabajar con los objetos. Allí vamos:

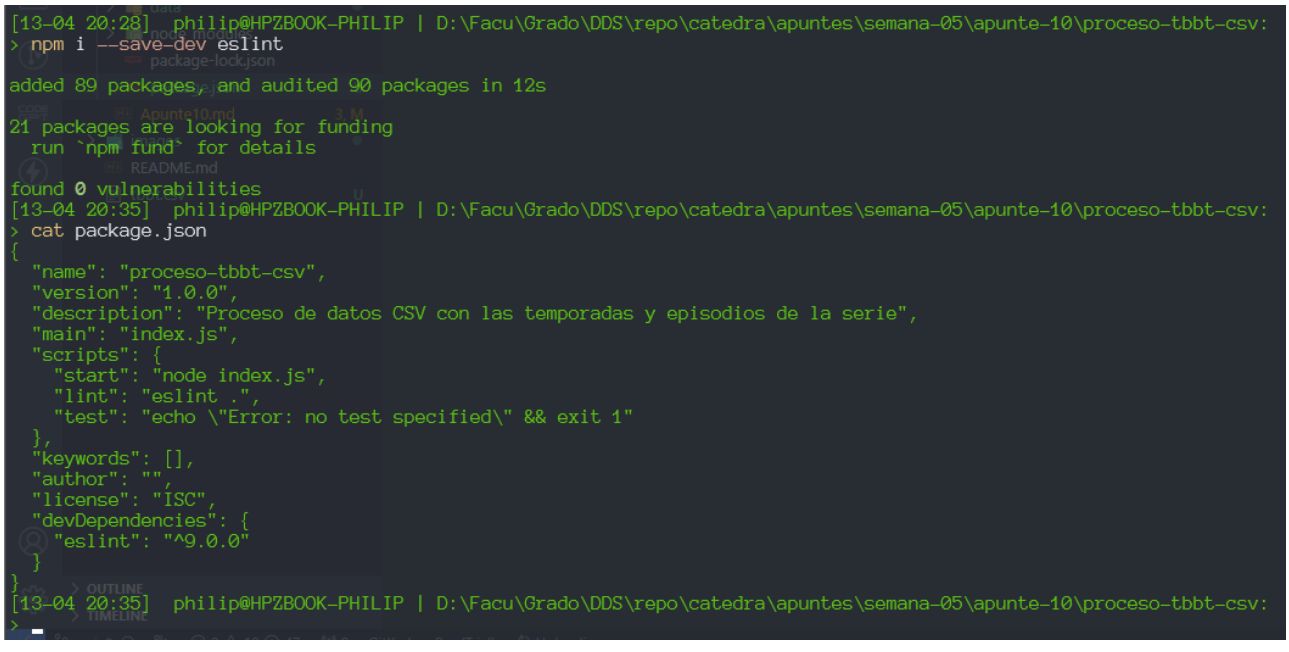
1. Vamos a crear el proyecto con `npm` como vimos en el apunte anterior, por cierto conversando con el profe Martín me hizo acordar que `npm init` tiene una opción bastante interesante para cuando queremos crear un proyecto con todas las respuestas por defecto. Para hacerlo usamos `npm init -y`, en este caso la opción -y responde con el valor por defecto a todas las preguntas de npm init creando el proyecto y dejando el archivo `package.json` para editarlo luego a medida que sea necesario.

   > [!TIP] Recordar que esta opción no va a solicitar ningún dato de forma interactiva al usuario y por lo tanto va a usar el nombre del directorio local como nombre del proyecto y va a dejar todas las demás opciones con su valor por defecto.

Una vez creado el proyecto, vamos a agregar un directorio para los archivos de datos dentro dle proyecto y allí vamos a guardar el archivo `tbbt.csv`.

Luego vamos a instalar eslint para verificar nuestro código Javascript y vamos a editar nuestro `package.json` para reflejar todo lo que hemos aprendido hasta el momento.

Nuestra estructura de proyecto debería haber quedado aproximadamente así:



2. Ahora vamos a comenzar a codificar un primer módulo que tenga por responsabilidad llevar a cabo la lectura del archivo CSV y su transformación a un vector de objetos.

Para llevar a cabo estas tareas vamos a utilizar dos módulos propios de Javascript, en primer lugar el módulo `fs` (File System) y en segundo lugar el módulo csv-parser.

- `fs`: es un módulo integrado en Node.js que proporciona una API para interactuar con el sistema de archivos del sistema operativo. Permite leer, escribir, modificar, mover y eliminar archivos y directorios. Entre su contenido fs incluye `createREadStream` que se utiliza para crear un flujo de lectura desde un archivo (es similar a `open` de Python) y devuelve una promesa.

  ```
  createReadStream(path [, options]);
  ```

  Toma como parámetros el path del archivo a leer y devuelve un objeto ReadStream que es un EventEmitter que emite eventos cuando se reciben datos del archivo. El parámetro opcional options es un objeto que permite configurar el flujo en detalle que por ahora no vamos a necesitar.

  En definitiva `createReadStream(...)` será el proveedor de nuestro flujo de lectura, del canal de entrada de datos a nuestro programa.

- `csv-parser`: es una biblioteca de Node.js que se utiliza para analizar archivos CSV y convertirlos en objetos Javascript. Proporciona una forma sencilla de leer y procesar archivos CSV línea por línea, emitiendo eventos para cada fila procesada.

  En nuestro caso vamos a combinar el flujo de entrada que obtendremos de `createReadStream` y el flujo de escritura que obtendremos del csv-parser a través del método `pipe(...)` para realizar la transformación de cada línea del archivo csv y agregar el resultado devenido en un objeto en el array resultante.

```
1  import { createReadStream } from 'fs';
2  import csv from 'csv-parser';
3
4  // Función para leer archivo CSV y devolver una promesa con el resultado
5  function readCSV(file) {
6      return new Promise((resolve, reject) => {
7          const results = [];
8          createReadStream(file)
9              .pipe(csv())
10             .on('data', (data) => results.push(data))
11             .on('end', () => resolve(results))
12             .on('error', (error) => reject(error));
13     });
14 }
```

En este caso se puede observar la creación de la promesa con la respuesta de `resolve` cuando pipe devuelve `'end'` y la respuesta de `reject` en el caso en que pipe devuelva `'error'`.

3. Ahora bien en una primera versión de uso del proceso anterior podríamos simplemente mostrar el contenido del array por consola:

```
(function main() {
    readCSV('./data/tbbt.csv')
        .then(data => {
            data.forEach(element => {
                console.log(element);
            });
        })
        .catch(error => console.error('Error al leer archivo CSV:', error));
})();
```

y aquí podemos observar el uso de `then` y de `catch` para procesar la respuesta exitosa de la promesa y capturar el caso en que se pueda haber provocado un error.

4. Pero ahora recién estamos a mitad de camino porque si bien un array nos permite trabajar los datos, todo empieza a cobrar sentido cuando sobre esos datos requerimos operaciones por ejemplo mostrar solo una temporada, determinar el nombre de un episodio específico o calcular el rating promedio de una temporada particular.

```
(function main() {
    readCSV('./data/tbbt.csv')
        .then(data => {
            console.log("Temporada 1");
            data
                .filter(element => element.season === '1')
                .forEach(element => {
                    console.log(element.title, " => ", element.imdb_rating);
                });

            console.log("Episodio 22 de la temporada 3",
                data.find(element => element.season === '3' && element.episode_num === '22').title
            );

            const season3 = data.filter(element => element.season === '3');
            const ratingAvg = season3.reduce((suma, element) => suma += parseFloat(element.imdb_rating), 0) / season3.length;
            console.log("Rating IMDB promedio de la temporada 3: ", ratingAvg);

        })
        .catch(error => console.error('Error al leer archivo CSV:', error));
})();
```

Todo esto se puede resolver utilizando los métodos de array que vimos en el apunte 7, en el fragmento de código anterior podemos ver un ejemplo... y funciona 😛.

Sin embargo, al usar un array, debemos ser conscientes que todas las operaciones son lineales y estas acciones no se llevarían a cabo de manera óptima, para lograr mejor eficiencia tendríamos que organizar los datos en un Mapa donde cada episodio puede ser identificado de forma unívoca, e incluso quizás las temporadas completas también.

Para esto es necesario una colección que organice los objetos en forma que puedan ser accedidos de manera eficiente, en javascript la colección que cumple con este requisito es `Map`. Analicemos pues el funcionamiento de los maps en general.

Mapas en Javascript - La colección Map

Los Maps son estructuras de datos en JavaScript que almacenan una colección de pares clave-valor, donde cada clave puede ser de cualquier tipo de dato y los valores pueden ser de cualquier tipo, incluso otros objetos. Los Maps proporcionan una forma eficiente de mapear claves a valores y permiten un rápido acceso y manipulación de los datos.

Por dentro estas colecciones implementan normalmente una estructura de datos conocida como tabla de dispersión o tabla de hash que está especialmente diseñada para permitir un acceso eficiente a partir de la clave de un objeto y quizás no tanto para los recorridos.

Existen varias formas de crear un Map en javascript:

```
// Crear un Map vacío
const miMapa = new Map();

// y allí agregar los mese del año usando como clave el número que les corresponde
miMapa.set(1, "Enero");
miMapa.set(2, "Febrero");
miMapa.set(3, "Marzo");
// ...

// O generar el mapa a partir de un array de arrays
const arrayMeses = [
    [1, "Enero"],
    [2, "Febrero"],
    [3, "Marzo"],
    [4, "Abril"],
    [5, "Mayo"],
    [6, "Junio"],
    [7, "Julio"],
    [8, "Agosto"],
    [9, "Septiembre"],
    [10, "Octubre"],
    [11, "Noviembre"],
    [12, "Diciembre"]
];
const meses = new Map(arrayMeses);

// O incluso desde un array de objetos de pares ordenados clave-valor
const arrayObjetosMes[
    { numero: 1, nombre: "Enero" },
    { numero: 2, nombre: "Febrero" },
    { numero: 3, nombre: "Marzo" },
    { numero: 4, nombre: "Abril" },
    { numero: 5, nombre: "Mayo" },
    { numero: 6, nombre: "Junio" },
    { numero: 7, nombre: "Julio" },
    { numero: 8, nombre: "Agosto" },
    { numero: 9, nombre: "Septiembre" },
    { numero: 10, nombre: "Octubre" },
    { numero: 11, nombre: "Noviembre" },
    { numero: 12, nombre: "Diciembre" }
];
const mapaMeses = new Map(arrayObjetosMes);
```

La verdad que las alternativas son muchas, pero incluso la más interesante es la alternativa funcional donde el constructor de Map recibe la funcion capaz de generar el par ordenado, pensemos en base a los datos del objeto que generamos en el ejemplo de proceso del CSV de TBBT, el siguiente fragmento crearía un Map donde la clave es un objeto que contiene la temporada y el episodio; el valor es el objeto completo:

```
const mapTbbt = new Map( element => [season: element.season + element.episode_num, element]);
```

O aún mejor, aunque un poco más riguroso a la hora de programarlo, sería construir un mapa de mapas en base al array que venimos trabajando donde el primero tenga como clave la temporada y cada temporada tenga asociado un mapa que contenga como clave el número de episodio:

```
// Recordar el uso del operador ternario como condición
const mapTbbt = new Map()
data.forEach(element => {
    mapTbbt.has(element.season) ?
        mapTbbt.get(element.season).set(element.episode_num, element)
        : mapTbbt.set(element.season, new Map([[element.episode_num, element]]));
});

/* El bloque anterior también se podría escribir así
const mapTbbt = new Map()
data.forEach(element => {
    if (mapTbbt.has(element.season))
        mapTbbt.get(element.season).set(element.episode_num, element);
    else
        mapTbbt.set(element.season, new Map([[element.episode_num, element]]));
});

*/
```

Una vez que construimos el `Map` tendremos acceso directo a cada par clave-valor a través de su clave, y este acceso se realiza de forma constante es decir sin la necesidad de realizar ningún recorrido por más grande que sea la cantidad de datos contenidos.

Los principales métodos de acceso son:

- `has(clave)`: Devuelve true si el Map contiene la clave especificada, de lo contrario, devuelve false.

  ```
  // Verificar si hay un mes con clave 1
  mapaMeses.has(1); // devolverá true

  mapaMeses.has(13); // devolverá false
  ```

- `get(clave)`: Devuelve el valor asociado a la clave especificada. Devolverá `undefined` si la clave no existe.

```
// Buscar un mes con clave 1
mapaMeses.get(1); // devolverá "Enero"

mapaMeses.get(13); // devolverá undefined
```

- `set(clave, valor)`: Como vimos, añade un nuevo par clave-valor al Map.

```
// Buscar un mes con clave 1
mapaMeses.set(1, "Ene"); // En el caso que la clave ya existiera se
reemplaza el valor asociado

mapaMeses.get(13, "Mes lunar"); // devolverá undefined
```

- `delete(clave)`: Como vimos, añade un nuevo par clave-valor al Map.

```
// Buscar un mes con clave 1
mapaMeses.delete(13); // Elimina el "Mes lunar" que agregamos antes
```

En los casos de los mapas de episodios de TBBT que construimos, tendremos la opción de acceder con la clave compuesta por la sumas de cadenas de temporada y episodio, o el esquema más específico donde accedemos primero con la clave de temporada y sobre el resultado que es un mapa accedemos con la clave de episodio, este último ejemplo quedaría así:

```
// Una vez organizados los datos de esta forma tendríamos acceso inmediato e
inequívoco a cualquier episodio de la serie conociendo la temporada y el número de
episodio
console.log(mapTbbt.get('3').get('22'));
```

Recorridos en mapas

Finalmente es importante mencionar que no todo es color de rosas, los recorridos en mapas no sólo no son lo esperado pues la estructura de datos no fue optimizada para eso sino que como la estructura se organiza

para mejorar el acceso en caso de acceder en forma de recorrido no tenemos garantía de ver los elementos en el mismo orden que los insertamos.

Tenemos dos métodos de `Map` que nos permiten recorridos, uno para recorrer las claves y otro para recorrer los values, a saber:

- `keys()`: Métodos que devuelve un iterador de las claves.

```
// Recorriendo a partir de las claves
const keys = mapaMeses.keys();

for(const clave of keys) {
    console.log(mapaMeses.get(clave));
}
```

- `values()`: Métodos que devuelve un iterador de los valores almacenados como colección.

```
// Recorriendo a partir de las claves
const keys = mapaMeses.keys();

for(const clave of keys) {
    console.log(mapaMeses.get(clave));
}
```

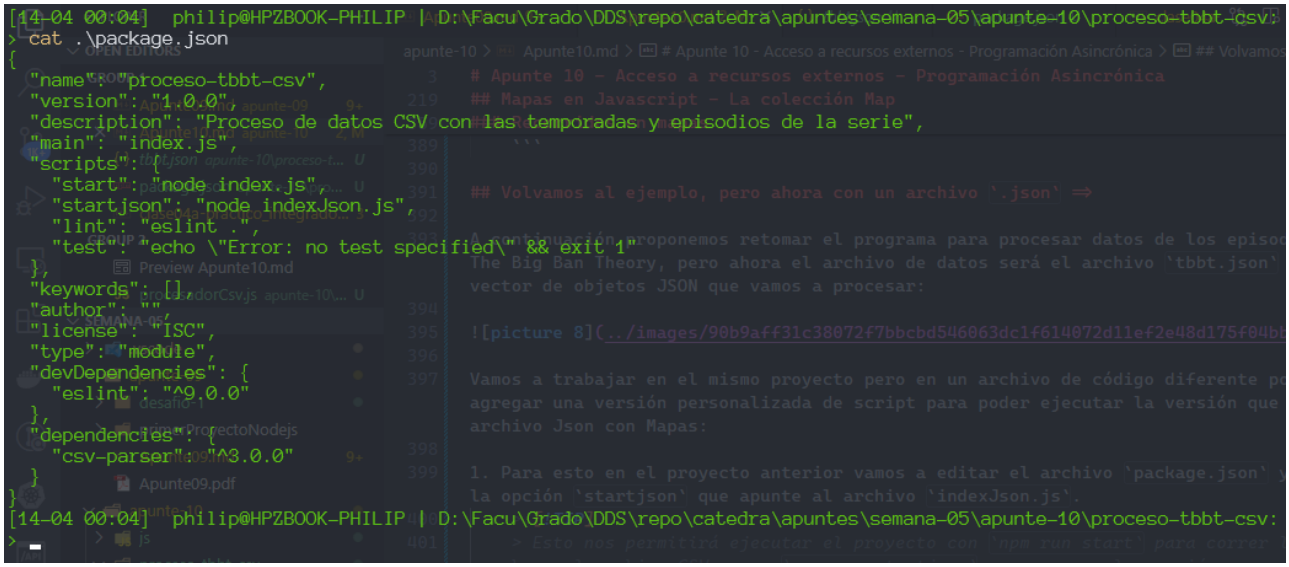
Volvamos al ejemplo, pero ahora con un archivo `.json` =>

A continuación proponemos retomar el programa para procesar datos de los episodios de la serie The Big Ban Theory, pero ahora el archivo de datos será el archivo `tbbt.json` que contiene un vector de objetos JSON que vamos a procesar:

Vamos a trabajar en el mismo proyecto pero en un archivo de código diferente por lo que vamos a agregar una versión personalizada de script para poder ejecutar la versión que procesa el archivo Json con Mapas:

1. Para esto en el proyecto anterior vamos a editar el archivo `package.json` y vamos a agregar la opción `startjson` que apunte al archivo `indexJson.js`.

> [!TIP] Esto nos permitirá ejecutar el proyecto con `npm run start` para correr la versión en base al archivo CSV o con `npm run startjson` para correr la versión que procesa el archivo json.



Nuestro archivo `package.json` quedará aproximadamente así donde podemos observar las dependencias y configuraciones anteriores y la alternativa de ejecución que agregamos `startjson`.

2. Ahora vamos a comenzar a codificar un módulo que tenga por responsabilidad llevar a cabo la lectura del archivo JSON y su carga a un vector de objetos en memoria.

   Para llevara cabo estas tareas vamos a utilizar el módulo propio de Javascript que ya utilizamos, el módulo `fs` (File System), pero en este caso vamos a aprovechar readFile porque vamos a leer el vector de objetos JSON completo y no línea por línea.

```
1   "use strict";
2
3   import { readFile } from 'fs/promises';
4
5   // Función para leer archivo con el vector de objetos JSON y
6   //  devolver una promesa con el resultado
7   function readJSON(file) {
8       return readFile(file, 'utf8')
9           .then(data => JSON.parse(data));
10  }
```

   En este caso se puede observar que readFile ya es una promesa y por lo tanto solo resta reaccionar en el caso de éxito con el código que vemos en then para procesar los objetos.

3. Un detalle interesante a revisar es que si intentamos utilizar el vector de objetos obtenido ahora desde el archivo JSON, obtendremos el mismo resultado que antes y por lo tanto lo podemos consumir exactamente de la misma manera:

```
(function main() {
    readJSON('./data/tbbt.json')
        .then(data => {
            console.log("Temporada 1");
            data
                .filter(element => element.season === '1')
                .forEach(element => {
                    console.log(element.title, " => ", element.imdb_rating);
                });

            console.log("Episodio 22 de la temporada 3",
                data.find(element => element.season === '3' && element.episode_num === '22').title
            );

            const season3 = data.filter(element => element.season === '3');
            const ratingAvg = season3.reduce((suma, element) =>
                suma += parseFloat(element.imdb_rating), 0) / season3.length;
            console.log("Rating IMDB promedio de la temporada 3: ", ratingAvg);

```

4. Pero, más allá de haber probado ahora con otro esquema de consumo de datos desde un archivo, muy utilizado por cierto en Javascript, vamos ahora a trabajar este conjunto desde el punto de vista de un map, para resolver los mismos requisitos.

```
            // En esta primera alternativa vamos a utilizar como clave del map una cadena compuesta por
            //  la cadena de la temporada concatenada con la cadena del número de episodio
            const mapTbbt = new Map(data.map( element =>
                [element.season + element.episode_num, element]
            ));

            // Vemos un gran beneficio a la hora de acceder al episodio 22 de la temporada 3
            //  es un acceso directo
            console.log("Episodio 22 de la temporada 3");
            // console.log(mapTbbt.has('322'));
            console.log(mapTbbt.get('322'));

            // Quizás no vemos tanto beneficio al acceder para buscar toda la temporada 1
            //  y algo parecido va a pasar para calcular el rating promedio de la temporada 3
            console.log("Temporada 1");
            Array.from(mapTbbt.values())
                .filter(element => element.season === '1')
                .forEach(element => console.log(element.title, " => ", element.imdb_rating));

```

5. Si finalmente intentamos la alternativa de construir un Mapa de Mapas, logramos ambos beneficios ya sin necesidad de iterar en ningún caso, cómo sería:

```
        // Ahora construyamos un mapa de mapas, nos va a costar un poco más construirlo pero veremos los beneficios
        //  este bloque ya lo analizamos cuando vimos alternativas de construcción de mapas
        // Esencialmente lo que hace es para cada elemento,
        //   si la temporada ya existe en el mapa de temporadas agrega el episodio al mapa asociado a la temporada
        //   si la temporada no existe agrega una nueva entrada al mapa de temporadas con un mapa nuevo como valor conteniendo
        //     este único episodio
        const mapTbbt = new Map()
        data.forEach(element => {
            mapTbbt.has(element.season) ?
                mapTbbt.get(element.season).set(element.episode_num, element)
                : mapTbbt.set(element.season, new Map([[element.episode_num, element]]));
        });

        // Vemos un gran beneficio a la hora de acceder al episodio 22 de la temporada 3
        //  es un acceso directo
        console.log("Episodio 22 de la temporada 3");
        // El primer get obtiene el mapa de la temporada 3 y a ese nuevo mapa le pido el episodio 22
        console.log(mapTbbt.get('3').get('22'));

        // Pero ahora si mostrar la temporada 1 se vuelve simple
        console.log("Temporada 1");
        // Al mapa de temporadas le pido la 1 y sobre esa temporada muestro todos los episodios
        Array.from(mapTbbt.get('1').values())
            .forEach(element => console.log(element.title, " => ", element.imdb_rating));

        // E incluso en el caso del rating promedio de la temporada 3 seguimos muy simple
        //  puesto que vuelvo a trabajar siempre sobre una sola temporada
        console.log("Rating promedio de la temporada 3")
        const ratingAvg = Array.from(mapTbbt.get('3').values())
                        .reduce((suma, element) => suma += parseFloat(element.imdb_rating), 0) / mapTbbt.get('3').size;
        console.log("Rating IMDB promedio de la temporada 3: ", ratingAvg);

```

El método Fetch

Una promesa es similar a un tipo de dato y es por este motivo que muchas funciones de JavaScript y/o de librerías externas cuyo resultado es asíncrono, o sea que demorará un tiempo en llegar están implementados en promesas. Uno de estos métodos es fetch.

La función **fetch** permite hacer una petición a un API y es justamente un callback. Por lo que tenemos que recibirlo usando then y catch de la siguiente forma.

```
const pokemones = fetch("https://pokeapi.co/api/v2/pokemon/1");

pokemones.then(res => res.json())
    .then(data => {
        console.log(data.name);
    }).catch(error => console.log(error))
```

Esta es la forma que comunmente se utiliza a fetch pero hay que acomodarlo de tal manera que sea un poco más entendible.

```
let pokemones = fetch("https://pokeapi.co/api/v2/pokemon/1");

pokemones
    .then(res => {
        return res.json()
    })
    .then(data => {
        console.log(data.name);
    }).catch(error => console.log(error))
```

Como puedes ver en el código anterior se pueden encadenar métodos then, pero se pueden hacer solo cuando el return del then anterior es una promesa. En este caso si nosotros vemos la implementación de json() veremos que este método devuelve una promesa. Por lo que el segundo then es la captura del resolve del json(). Si intentamos descomponer más al código anterior tendríamos algo así:

```
// primera promesa
let pokemones = fetch("https://pokeapi.co/api/v2/pokemon/1");

// segunda promesa
let respuesta = pokemones.then(res => {return res.json()});

respuesta.then(data => {
    console.log(data.name);
})
```

Entonces si nosotros quisieramos ejecutar el fetch de manera ordenada y síncrona para ver los cinco primeros pokemones podemos hacer lo siguiente:

```
function obtener_pokemon(id){
    let url = "https://pokeapi.co/api/v2/pokemon/" + id;
    return fetch(url).then(res => {return res.json()});
}

obtener_pokemon(1).then(data => {
    console.log(data.name);
    return obtener_pokemon(2);
}).then(data =>{
    console.log(data.name);
    return obtener_pokemon(3);
}).then(data =>{
    console.log(data.name);
    return obtener_pokemon(4);
}).then(data =>{
    console.log(data.name);
    return obtener_pokemon(5);
}).then(data =>{
    console.log(data.name);
})
```

Async Await

Una de las características más recientes de Javascript: **async / await**. Usamos el async para definir una función donde se encontrará el await que nos permitirá esperar una promesa de tal forma que podamos volver nuestro código sincrónico.

```
function obtener_pokemon(id){
    let url = "https://pokeapi.co/api/v2/pokemon/" + id;
    return fetch(url).then(res => {return res.json()});
}

async function nombrar_pokemones() {
    let pokemon1 = await obtener_pokemon(1);
    console.log(pokemon1.name);
}
```

La palabra clave await se usa en JavaScript para esperar a que una promesa se resuelva o rechace. Se utiliza dentro de una función async, y hace que la ejecución de la función se pause hasta que la promesa se resuelva y devuelva su resultado.

En el código de ejemplo, await se usa para esperar a que la función obtener_pokemon termine de ejecutarse. obtener_pokemon devuelve una promesa que se resuelve con los datos de un Pokémon específico obtenidos de la API de Pokémon.

Cuando se llama a obtener_pokemon(1), se realiza una solicitud a la API de Pokémon para obtener los datos del Pokémon con ID 1. Esta solicitud puede tardar un tiempo en completarse, por lo que obtener_pokemon devuelve una promesa.

Al usar await antes de obtener_pokemon(1), nos aseguramos de que JavaScript no continúe ejecutando el resto del código en nombrar_pokemones hasta que se haya completado la solicitud a la API de Pokémon y se haya obtenido la respuesta. Una vez que la promesa se resuelve, pokemon1 se establece con el valor de la respuesta de la API, y luego se imprime el nombre del Pokémon en la consola.

Usando el **async await** podemos lograr esto, que la variable espere por su resultado antes de ejecutar el console.log(). Y como podemos hacer que una variable espere podemos lograr incluso esto.

```
async function nombrar_pokemones() {
    for (let i = 1; i < 6; i++) {
        let pokemon = await obtener_pokemon(i);
        console.log(pokemon.name);
    }
}
```

Como se puede observar, el **async await** nos permite esperar el **resolve de una promesa** de tal manera que podamos obtener un resultado síncrono.

O incluso podemos hacerlo usando forEach de la siguiente manera.

```
function obtener_pokemones(){
    let url = "https://pokeapi.co/api/v2/pokemon/";
    return fetch(url).then(res => {return res.json()});
}

async function nombrar_pokemones() {
    let pokemones = await obtener_pokemones();
    pokemones.results.forEach(pokemon => {
        console.log(pokemon.name)
    })
}
```

En este caso lo que pedimos al api es una lista con los 20 primeros pokemones pero estos están como una lista de objetos dentro del objeto result en la respuesta json que obtendremos. Usamos el forEach para recorrer cada elemento del objeto results.

Lo explicamos mas detallado:

- `obtener_pokemones`: Esta función realiza una solicitud HTTP GET a la API de Pokémon utilizando la función fetch. La URL de la API es "https://pokeapi.co/api/v2/pokemon/", que devuelve una lista de Pokémon. La función fetch devuelve una promesa que se resuelve con la respuesta de la API. Luego, se utiliza el método then para transformar la respuesta en formato JSON cuando la promesa se resuelve.
- `nombrar_pokemones`: Esta es una función async, lo que significa que puede contener la palabra clave await para esperar a que se resuelva una promesa. En esta función, se llama a obtener_pokemones y se espera a que se resuelva la promesa que devuelve. Una vez que la promesa se resuelve, el resultado (la lista de Pokémon en formato JSON) se asigna a la variable pokemones. Luego, se utiliza el método

forEach para iterar sobre la lista de Pokémon (que se encuentra en pokemones.results) y se imprime el nombre de cada Pokémon en la consola.

Sin embargo, muchas veces lo que buscaremos será llevar una lista de objetos directamente a un arreglo para eso existe el método **map**. El cual lo podemos utilizar de la siguiente forma.

```
async function nombrar_pokemones() {
    let pokemones = await obtener_pokemones();
    let arregloPokemones = pokemones.results.map(pokemon => pokemon.name)
    console.log(arregloPokemones)
}
```

Esta es una función async, lo que significa que puede contener la palabra clave await para esperar a que se resuelva una promesa.

let pokemones = await obtener_pokemones();: Aquí se llama a la función obtener_pokemones y se espera a que se resuelva la promesa que devuelve. Una vez que la promesa se resuelve, el resultado (la lista de Pokémon en formato JSON) se asigna a la variable pokemones.

let arregloPokemones = pokemones.results.map(pokemon => pokemon.name): Aquí se utiliza el método map para crear un nuevo arreglo que contiene los nombres de todos los Pokémon. map toma una función como argumento, que se aplica a cada elemento del arreglo pokemones.results. En este caso, la función toma un pokemon como argumento y devuelve pokemon.name, es decir, el nombre del Pokémon.

console.log(arregloPokemones): Finalmente, se imprime el arreglo de nombres de Pokémon en la consola.

Bibliografía

- Mozilla Developer Network - https://developer.mozilla.org/
- Free Code Camp en Español - https://www.freecodecamp.org/espanol