


[Teoría Módulo 9] Llamadas Asíncronas

Introducción

Hasta ahora todas las operaciones que hemos visto obtienen el resultado de forma inmediata al finalizar la ejecución de la instrucción. Podemos entenderlas como tareas **síncronas**, que esperan a obtener el resultado antes de continuar con la siguiente instrucción de forma secuencial.

¿Qué ocurre con operaciones que no son inmediatas? Por ejemplo, recuperar datos de un servidor no es algo inmediato: tiene un tiempo de espera, tiempo de respuesta del servidor, tiempo de descarga de datos, hasta que éstos llegan como respuesta a la operación en nuestra aplicación. Lo deseable, en este caso, es que nuestra aplicación **no se quede congelada** durante ese periodo de tiempo, dando la sensación que está "bloqueada", si no que lance la petición de datos y espere en un segundo plano a que estos lleguen de alguna forma como resultado, **sin bloquear la interfaz y permitiendo la ejecución de otras tareas**. Estamos ante un ejemplo de **llamada asíncrona**. Otro caso sería, por ejemplo, una tarea que espera 10 segundos para mostrar un aviso al usuario.

Por tanto, los conceptos **síncrono** vs **asíncrono** se refiere a cuando tendrá lugar la respuesta:


- **Síncrono**: la operación espera a obtener la respuesta antes de seguir la ejecución.
- **Asíncrono**: la respuesta a la operación no es inmediata, y por tanto no espera de forma bloqueante a que esta respuesta llegue. 

Cuando trabajamos con asincronía, tenemos **mecanismos para tratar la respuesta cuando llegue**:

- Callbacks
- Promesas

Nota: tenéis un artículo de recomendada lectura sobre Javascript y asincronía en [Javascript Asíncrono: La guía definitiva — Lemoncode formacion](#)

Callbacks

Los *callbacks* son la pieza clave para que Javascript pueda funcionar de forma asíncrona. **Un callback es una función que se pasa como argumento de otra función**, y que será invocada en su debido momento para completar algún tipo de acción. 

El siguiente es un ejemplo sencillo utilizando *callback* y la función *setTimeout* de javascript:

```
setTimeout(function() {  
  alert("Hola después de 1 segundo");  
}, 1000);
```

El método *setTimeout* de Javascript ejecutará el *callback* que recibe como primer parámetro pasados los milisegundos que se indican en su segundo parámetro. Tenéis más información sobre el método *setTimeout* en https://www.w3schools.com/jsref/met_win_settimeout.asp

El ejemplo anterior abrirá una ventana *alert* después de esperar durante 1 segundo (1000 milisegundos).

Como vemos, nuestra función *callback* es la siguiente:

```
function() {  
  alert("Hola después de 1 segundo");  
}
```

y la pasamos como argumento al método *setTimeout*. También podemos asignarla a una variable con nombre, en lugar de pasarla como función anónima:

```
const myCallback = function() {  
  alert("Hola después de 1 segundo");  
}  
  
setTimeout(myCallback, 1000);
```

Y podemos utilizar *arrow functions* igualmente:

```
const myCallback = () => alert("Hola después de 1 segundo");  
  
setTimeout(myCallback, 1000);
```

```
setTimeout(() => alert("Hola después de 1 segundo"), 1000);
```

Callback Hell



Los callbacks pueden lanzar nuevas llamadas asíncronas, anidándose tantas como se desee:

```
setTimeout(function() {  
  console.log("Etapa 1 completada");  
  setTimeout(function() {  
    console.log("Etapa 2 completada");  
    setTimeout(function() {  
      console.log("Etapa 3 completada");  
      setTimeout(function() {  
        console.log("Etapa 4 completada");  
        setTimeout(function() {  
          console.log("Etapa 5 completada");  
        }, 5000);  
      }, 4000);  
    }, 3000);  
  }, 5000);  
}, 3000);
```

```
    }, 2000);  
  }, 1000);
```

Ésto es uno de los inconvenientes clásicos de los *callbacks*, además de la indentación, resta legibilidad, dificulta el mantenimiento, y añade complejidad a nuestro código. Al *Callback Hell* también se le conoce como **Pyramid of Doom** o **Hadouken**.

Promesas



Una Promesa (Promise) es un objeto que representa el resultado de una operación asíncrona. Este resultado podría estar disponible ahora o en el futuro.

Una Promesa en Javascript entenderá dos posibles resultados:

- Éxito (resolved)
- Fracaso (rejected)

Podemos crear una promesa sencilla, que siempre se resuelva correctamente, y que devuelva sólo un texto, de la siguiente manera:

```
const myPromise = new Promise((resolve, reject) => {  
  resolve("Hola, esto es la respuesta correcta de la promesa :");  
});
```

Como vemos, al crear la promesa tenemos dos *callbacks* a los que invocar según el tipo de resultado que queramos resolver en ella: **resolve** y **reject**, para resultado correcto y fallido respectivamente. Cada *callback* lo invocaremos pasándole el resultado a devolver (ya sea una cadena de texto como el ejemplo anterior, un número, un array, un objeto, etc. o el propio error en caso de que ocurra un fallo).

Con nuestra promesa creada, podremos recuperar el resultado (cuando éste se produzca) con **.then()** de la siguiente forma:

```
myPromise.then(result => {  
  alert(result);  
});
```



Igualmente, podremos saber si se ha producido un error lanzado con **reject** con **.catch()**:

```
const myPromise = new Promise((resolve, reject) => {  
  reject("Se ha producido un error! :(");  
});  
  
myPromise.then(result => {  
  alert(result);  
}).catch(error => {
```

```
    alert(error);  
  });
```

Llamadas asíncronas

fetch API

La API *fetch* nos proporciona un canal para obtener recursos (como datos) a través de la red.

Se basa en peticiones (**request**) que realiza nuestra aplicación y las respuestas (**response**) que recibe del servidor remoto:

- El método *fetch()* realiza una petición (*request*) al servidor de los datos que necesita.
- El propio objeto devuelve una *Promesa* con el objeto *response* de la petición (tanto si tiene éxito como si no). El propio objeto de respuesta tiene información sobre su *status* (correcto / error).
- Cuando la promesa resuelva la respuesta, ésta proporciona métodos y herramientas para manejar su contenido.

Vamos a ver un ejemplo para obtener los datos de la siguiente URL:

<https://api.github.com/orgs/lemoncode/members>

Si la abrimos con un navegador, veremos un array de objetos con la información de los miembros de *Lemoncode* en *Github*. Vamos a recuperar esa lista de miembros y manejarlos con javascript utilizando la API *fetch*:

```
fetch("https://api.github.com/orgs/lemoncode/members")  
  .then(response => {  
    if(response.ok) {  
      return response.json();  
    } else {  
      console.log(response.statusText);  
    }  
  });
```

El ejemplo anterior realiza una petición (*request*) a la URL con los datos. Cuando estos son devueltos por el servidor, se ejecuta el *callback* que pasamos como argumento de *.then()*. Este *callback* se encarga de comprobar que la respuesta ha sido correcta (*response.ok === true*) y devuelve la misma en formato *json* con los datos.

Vamos a mostrar estos datos por consola:

```
fetch("https://api.github.com/orgs/lemoncode/members")  
  .then(response => {  
    if(response.ok) {  
      return response.json();  
    } else {  
      console.log(response.statusText);  
    }  
  });
```

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

Como vemos, tenemos dos `.then` consecutivos: el primero convierte la respuesta en datos *json*, y el segundo **utiliza la respuesta que devuelve del anterior** para mostrar estos datos por consola. Efectivamente, un `.then()` recibe los datos (o la promesa de datos) que devuelve el anterior.

Para capturar errores de red, podemos utilizar `.catch()` como ya hemos visto:

```
fetch("https://api.githubTTTTTTTTT.com/orgs/lemoncode/members")  
  .then(response => {  
    if(response.ok) {  
      return response.json();  
    } else {  
      console.log(response.statusText);  
    }  
  })  
  .then(data => console.log(data))  
  .catch(error => console.log("Se ha producido un error"));
```

Como vemos, al escribir el dominio de la petición se nos quedó pulsada la letra "T" y, por tanto, *fetch* devolverá un error al intentar hacer una petición a una URL inexistente, que capturaremos con *catch*.

Axios

Axios: [GitHub - axios/axios: Promise based HTTP client for the browser and node.js](#) es una de las librerías más famosa para trabajar con peticiones de recursos/datos a un servidor.

Axios nos ayuda a gestionar tanto las peticiones como las respuestas de las mismas, sobre todo en escenarios complejos, con herramientas avanzadas, aunque nos quedaremos a un nivel básico en esta sesión.

El siguiente ejemplo muestra cómo utilizar *axios* para obtener los datos de los ejemplos anteriores:

```
import axios from "axios";  
  
axios.get("https://api.github.com/orgs/lemoncode/members")  
  .then(response => response.data)  
  .then(data => console.log(data))  
  .catch(error => console.log("Se ha producido un error"));
```

Como vemos, *axios* nos ofrece un método *get* para realizar la petición de obtener datos, pero no es el único método del que disponemos para comunicarnos con el servidor: podríamos enviar datos, o pedir que se eliminen.

Verbos HTTP

Hasta ahora hemos visto como **obtener** datos de un servidor. En una aplicación además de obtener datos necesitaremos enviar datos al servidor (como dar de alta un usuario nuevo), modificar datos (como actualizar un perfil de usuario), o eliminar datos (como eliminar un usuario).

Para esto utilizamos lo que conocemos como *verbos HTTP* y se corresponden con:

- GET: obtener
- POST: añadir
- PUT: modificar
- DELETE: eliminar

En este módulo vamos a quedarnos sólo con traer datos del servidor (GET) como ya hemos visto, y veremos el resto en módulos futuros.