Docente: Matías Santos

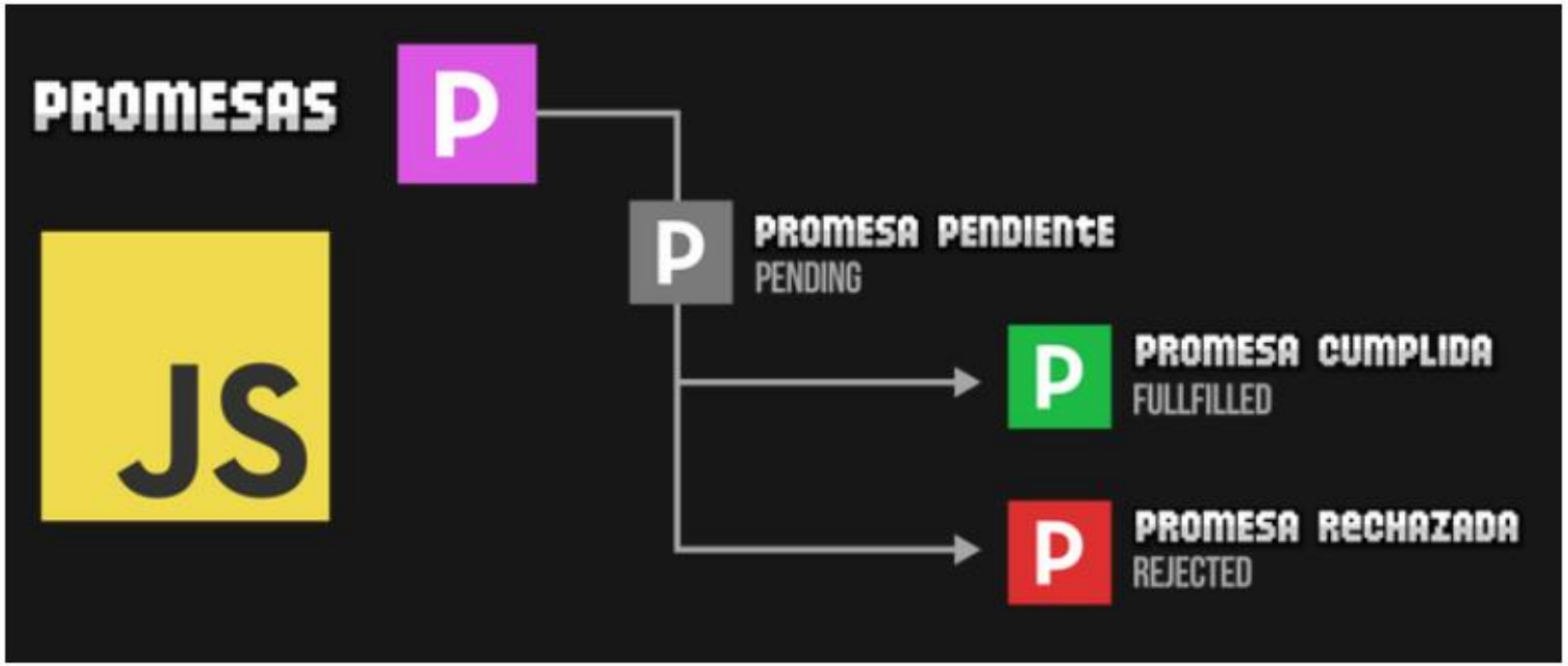
Introducción a JavaScript: Async / Await y fetchAPI

Para consumir APIs desde Javascript habitualmente se suele utilizar ***fetchAPI***.

fetchAPI es un sistema moderno basado en promesas para realizar peticiones HTTP asíncronas de una forma más legible y cómoda.

Pero, antes de ver cómo funciona fetchAPI, veamos que son las *Promesas* y que es *async/await* en Javascript.

Una promesa representa un valor que podrá estar disponible ahora, en un futuro o nunca. Las promesas en Javascript se representan a través de objetos, y cada promesa estará en un estado concreto: pendiente, aceptada o rechazada.



Si la promesa se cumple su estado será **Fullfilled**.
Si la promesa es rechazada su estado será **Rejected**.
Y si la promesa está pendiente su estado será **Pending**.

Para definir una promesa vamos a usar lo que se conoce como funciones flecha en JS, la promesa recibe dos métodos que llamaremos “resolve” y “reject”.

El *resolve* es lo que se ejecutará cuando la promesa sea resuelta, y *reject* se ejecutará cuando exista algún error en la promesa o la misma sea rechazada.

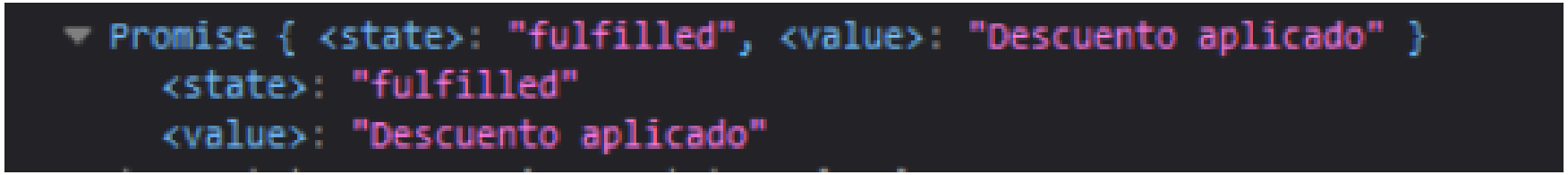
```
const aplicarDescuento = new Promise( (resolve, reject) => {
    const descuento = true;

    if(descuento){
        resolve("Descuento aplicado");
    }else{
        reject("No se pudo aplicar descuento");
    }
})
```

En el ejemplo anterior, definimos una promesa, y lo que hace es controlar si se aplica un descuento al usuario o no.

Cuando ejecutemos las promesas se pasarán los dos valores automáticamente, uno será el *resolve* y el otro será el *reject*, por lo tanto, ambos métodos tendrán los valores que les definimos entre paréntesis.

Ejecutemos la promesa antes definida: aplicarDescuento

Si mostramos en la consola el resultado de la promesa, utilizando *console.log(aplicarDescuento)*, veremos lo siguiente:



Cambiemos el valor de la variable booleana descuento a falso, ejecutamos nuevamente *console.log(aplicarDescuento):*

```
▼ Promise { <state>: "rejected", <reason>: "No se pudo aplicar descuento" }
    <state>: "rejected"
    <reason>: "No se pudo aplicar descuento"
```

Imaginemos ahora que definimos la promesa, pero sin cuerpo, o sea, sin código dentro de la función promesa y ejecutamos nuevamente *console.log(aplicarDescuento):*

```
▼ Promise { <state>: "pending" }
    <state>: "pending"
```

Lo que estamos viendo en estas tres pruebas son los estados que las promesas pueden tener, y los que nosotros deberemos manejar o controlar.

Para trabajar con las promesas, ellas tienen asociadas los siguientes métodos:

.then(function resolve): ejecuta la función del *resolve* cuando la promesa se cumple

.catch(function reject): ejecuta la función del *reject* cuando la promesa se rechaza

.finally(function end): ejecuta la función *end* en ambos casos

Entonces para trabajar con nuestras promesas teniendo en cuenta sus estados y los métodos que nos proporciona, podemos definirlas y utilizarlas de la siguiente manera:

```
const aplicarDescuento = new Promise( (resolve, reject) => {
    const descuento = false;

    if(descuento){
        resolve("Descuento aplicado");
    }else{
        reject("No se pudo aplicar descuento");
    }
})


aplicarDescuento
    .then( resultado => {
        console.log(resultado);
    })
    .catch( rechazo => {
        console.log(rechazo);
    })
    .finally( () => {
        console.log("Fin")
    })
```

Lo que hemos visto es la forma de manejar operaciones asíncronas en Javascript, cuyas operaciones pueden tener tres resultados, al utilizarlas podremos tener en cuenta sus estados y actuar en consecuencia.

Otro ejemplo podría ser para obtener un listado de clientes desde un servidor, en caso de que el listado se pueda obtener tomaremos una acción, pero si por algún motivo el servidor no está disponible, y la promesa no puede completarse tomaremos otra acción.

Otro ejemplo podría ser el caso de login de un usuario, si el usuario puede autenticarse diremos que la promesa ha sido resuelta y tomaremos una acción en concreto, pero en caso de que el usuario falle en sus credenciales, el método de autenticación fallará por lo cual podremos tomar otro tipo de acción.

Async / Await

Las palabras ***async/await*** son una forma que se introduce en JS para la gestión de las promesas.

Lo que hace ***await***, es detener la ejecución del programa y no continuar, se espera a que se resuelva la promesa, y hasta que no lo haga, no continua.

Por ejemplo, definamos una función que nos retorna una promesa, simulemos una espera de tres segundos con la función de JS setTimeout().

```
function descargarClientes(){
    return new Promise( resolve => {
        console.log("Descargando clientes... espere...");

        //Simulamos la descarga con cierto tiempo de espera
        setTimeout(()=>{
            console.log("Pasaron 3 segundos...");
            resolve("Clientes descargados");
        }, 3000);
    })
}
```

Ahora que definimos la función que nos retorna una promesa, nos queda utilizar async/await, y para utilizar async/await tenemos que definir una nueva función e indicarle que dicha función debe ser asíncrona, para ello al inicio de la definición de la función agregamos la palabra "**async**".

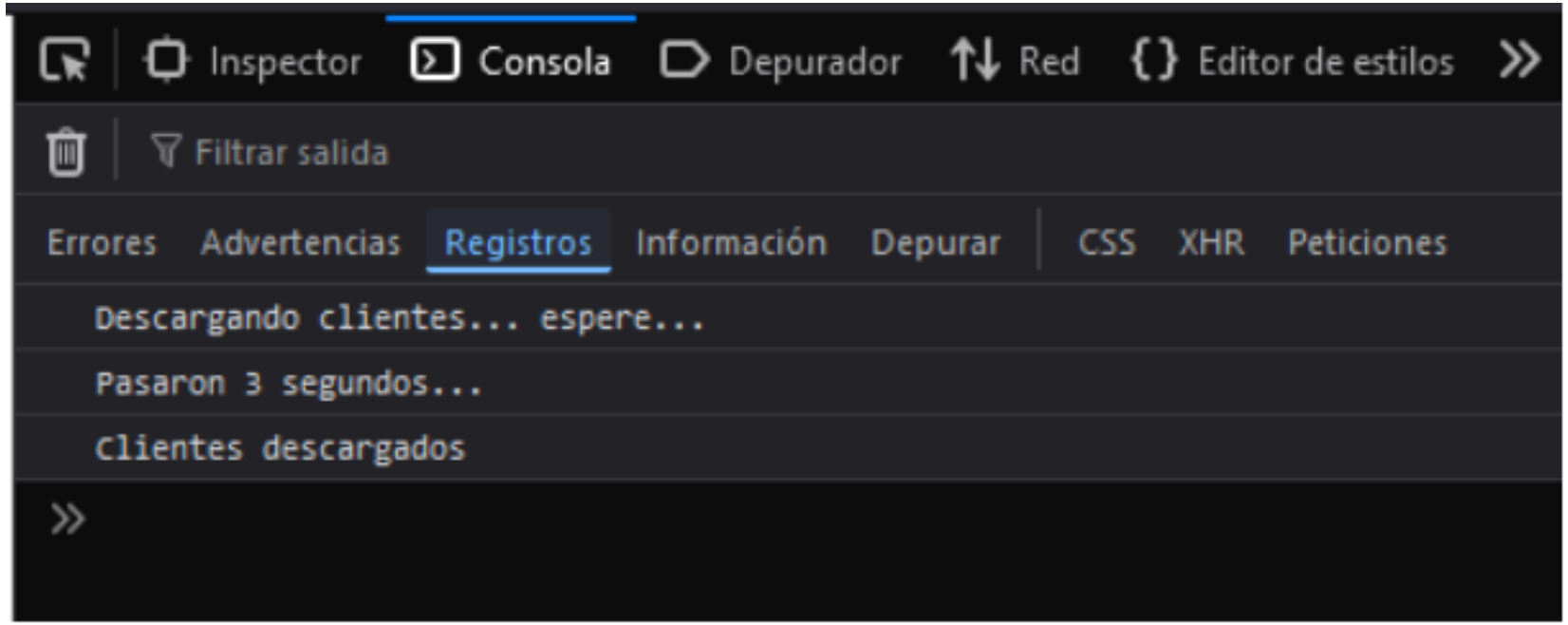
Cuando invoquemos a la función que nos retorna una promesa le tenemos que indicar que espere a su resolución, la cual podrá tener alguno de los estados que vimos anteriormente, esto lo hacemos utilizando la palabra "**await**" antes de llamar a dicha función.

```
async function app(){
    try{
        const resultado = await descargarClientes();
        console.log(resultado);
    }catch(error){
        console.log(error);
    }
}
```

El código completo quedaría implementado de la siguiente manera:

```
function descargarClientes(){
    return new Promise( resolve => {
        console.log("Descargando clientes... espere...");

        //Simulamos la descarga con cierto tiempo de espera
        setTimeout(()=>{
            console.log("Pasaron 3 segundos...");
            resolve("Clientes descargados");
        }, 3000);
    })
}


async function app(){
    try{
        const resultado = await descargarClientes();
        console.log(resultado);
    }catch(error){
        console.log(error);
    }
}

//Invocamos la función app()
app();
```

Para ver el resultado de lo que acabamos de implementar, abrimos nuestro navegador y observamos en la consola.
Veremos el mensaje:
“Descargando clientes… espere…” y luego de tres segundos veremos el resultado de nuestra promesa, o sea:



Si realizamos la siguiente modificación, con el fin de ver la espera en la resolución de la promesa:

```
async function app(){
    try{
        const resultado = await descargarClientes();

        console.log("Código en espera del resultado de la promesa...");

        console.log(resultado);
    }catch(error){
        console.log(error);
    }
}
```

Lo que acabamos de ver y comprobar es la utilidad de ***async/await***, que en sí es esperar a la ejecución y resultado de la promesa para continuar con la ejecución de nuestro código.

Esta es la ventaja que nos proporciona ***async/await***, de esto trata ***async/await***.

Si en vez de mostrar en consola el mensaje que hemos definido anteriormente, llamamos a una función que muestre el listado de los clientes descargados, dicha función no se ejecutará hasta que nuestro listado se haya descargado.

De esta manera podremos tener código en el que nos interesara el resultado de una llamada a una API para ejecutarse y tomar alguna acción cuando obtengamos la respuesta.

fetchAPI

La introducción anterior es necesaria porque ***fetchAPI*** es un sistema moderno basado en promesas para realizar peticiones HTTP asíncronas de una forma más legible y cómoda, tal cual lo mencionamos anteriormente.

Básicamente se trata de utilizar la palabra reservada ***fetch()*** y pasarle por parámetro la URL de la API para realizar una petición HTTP.

```
fetch("/descargar")
    .then(function(response){
        //Código
    })
    .catch(function(error){
        console.log(error);
    })
```

fetch() retornara una promesa que será aceptada cuando reciba una respuesta y será rechazada ante algún fallo de red o si por alguna razón no se puede completar la petición.

Al método ***.then()*** le pasamos el parámetro "*response*" que es el objeto de respuesta de la petición realizada, dentro de dicha función implementaremos la lógica que queremos hacer a partir de la respuesta obtenida.

A la función *fetch()* además de pasarle la URL de nuestra API, también podemos pasarle, opcionalmente, un objeto "*options*" con determinadas opciones para dicha petición.

Por ejemplo, podemos crear un objeto *options* con datos como:

method; método HTTP de la petición (GET, POST, PUT, DELETE, etc)

body; cuerpo de la petición, objetos en formato JSON

headers; cabeceras HTTP

Lo primero y más habitual suele ser indicar el método HTTP a realizar en la petición, por defecto es GET, pero lo podemos cambiar a lo que precisemos.

Si quisiéramos podemos definir objetos para pasarle en el BODY de la petición, así como modificar las cabeceras en el campo HEADERS.

Un ejemplo de fetchAPI con todo lo mencionado anteriormente podría ser:

```
const options ={
    method: "POST",
    headers: {
        "Content-Type": "application/json"
    },
    body: JSON.stringify(datos)
}

fetch("/descargar", options)
    .then(response => response.json())
    .then(data => {
        //Procesamos la respuesta obtenida
        //en formato JSON
    })
    .catch(function(error){
        console.log(error);
    })
```

En el ejemplo anterior tenemos un primer ***.then()*** con un objeto "***response***" el cual lo convertimos a ***JSON*** y derivamos a un segundo ***.then()*** para su procesamiento, esta es la respuesta que obtenemos de la API a la cual realizamos la petición.

En el segundo ***.then()*** utilizamos una función flecha de JS para procesar los datos que nos pasa el primer ***.then()*** en formato JSON, y en caso de obtener algún error se ejecutaría el ***catch()***, que solo mostrara en la consola del navegador el error recibido.

Una buena práctica en principio es definir en el segundo ***.then()*** un ***console.log(data)*** para observar en la consola de nuestro navegador como vienen formateados los datos, para luego analizar la mejora manera de procesarlos.

Esta es una breve introducción al funcionamiento de *fetchAPI* de Javascript para realizar peticiones HTTP, si se desea profundizar más, a continuación, se encuentran unos links interesantes sobre el tema:

https://developer.mozilla.org/es/docs/Web/API/Fetch_API/Using_Fetch

https://es.javascript.info/fetch

https://www.freecodecamp.org/espanol/news/javascript-fetch-api-para-principiantes/

https://lenguajejs.com/javascript/peticiones-http/fetch/

FIN.