educación IT
CODE YOUR FUTURE

Bootcamp Java Developer

Fase 2 - Java Web Developer

Módulo 19

educación IT
CODE YOUR FUTURE

Promesas

Promesas

Como mencionamos, una promesa es una tarea asincrónica que tiene tres estados posibles: pendiente, resuelta o rechazada.

Las promesas se crean con el constructor **`new Promise`**. Veámoslo en detalle.

```
1  const miPromesa = new Promise(function(resolver, rechazar)
2  {
3      // Código de mi promesa
4      // Este código se ejecuta cuando se ejecuta el new
5  });
6
```

API Promise

1. La promesa empieza con el estado de pendiente. Al ejecutarse **`new Promise`**, JavaScript lee en ese mismo momento la función pasada al constructor y ejecuta línea por línea cada instrucción.
2. El ejecutor recibe dos argumentos. El **primer argumento** es una función que, al llamarse, **cambia el estado de la promesa a 'resuelta'**.
3. El **segundo argumento** es una función que **cambia el estado de la promesa a 'rechazada'.** Es importante aclarar que el rechazo de una promesa no significa un error de código, sino que no se puede continuar con la operación.

Ejemplo didáctico sobre la sintaxis para las promesas:

```
const miPromesa = new Promise(function(resolver, rechazar)
{
    const n1 = Number(prompt('Ingrese el primer número'));
    const n2 = Number(prompt('Ingrese el segundo número'));

    if (n2 == 0) {
        rechazar('División por cero');
        return;
    }

    resolver(n1 / n2);
    return;
});
```

Las promesas **deben ser usadas para operaciones del *call-stack*** y, en el ejemplo anterior, no hay ninguna de ellas. Sin embargo, nos sirve a modo didáctico.

Es una división. La división falla si el segundo número es 0 ¿Cómo falla? Ejecutando la función **`rechazar`** (2do argumento). Se puede pasar una razón del rechazo, que será el valor devuelto al programa principal (lo veremos más adelante).

Si todo está bien, llamamos a la función **`resolver`** pasándole como argumento el resultado de la división.

En este ejemplo tenemos un caso muy común:

```
1   const get = url => new Promise(function(resolver, rechazar)
2   {
3       const xhr = new XMLHttpRequest();
4       xhr.responseType = 'json';
5       xhr.open('GET', url);
6       xhr.send();
7       xhr.addEventListener('load', () => resolver(xhr.response));
8   });
9
10
11  get('/posts');
12
```

Sobre el ejemplo anterior:

Si omitimos la línea 11, la promesa no se ejecuta. Esta es una forma de retrasar la ejecución de las promesas (**`thunk promise`**).

Esta promesa representa un **`XMLHttpRequest`** común. Ahora bien, ¿dónde tenemos la llamada a rechazar? En este caso no se utiliza. Puede haber promesas que no tengan el estado de rechazada.

Hasta ahora, vimos cómo ejecutar una promesa. Sin embargo, esa es la mitad del trabajo. La otra mitad consiste en **registrar *callbacks*** para ser ejecutados según la promesa cambie de estado.

El objeto devuelto por **`Promise`** tiene tres métodos:

- **`.then(function)`** ejecuta la función dada si la promesa se resuelve.
- **`.catch(function)`** ejecuta la función dada si la promesa se rechaza.
- **`.finally(function)`** ejecuta la función dada independientemente de su estado.

Ejemplo:

```
miPromesa
    .then((division) => console.log('La división es ', division))
    .catch((error) => console.log(error))
```

```
get('/posts')
    .then(function(respuesta)
    {
        console.log(respuesta);
    });
```

Función asincrónica

Es una función común declarada con la palabra reservada async, que permite el uso del operador `await` dentro de ella.

Al ser declarada como asincrónica, la función retorna una promesa que resuelve en su valor de retorno (en lugar de retornar dicho valor directamente). Por esta razón, es una forma mucho más sencilla de crear promesas.

Dentro de una función asincrónica, podemos usar **`await`**. El operador **`await`** espera a que una promesa se resuelva y retorna el valor pasado a la función resolver. De esta forma, es mucho más sencillo el manejo de promesas. Se ha pasado de necesitar ***callbacks*** a usar un simple operador.

Ejemplo:

```
const obtenerPosts = async () =>
{
    const res = await fetch('/posts');
    const data = await res.json();
    return data;
}

const filtrarPorUsuario = async (usuario, datos) => datos.filter((item) => item.username == usuario);

const obtenerCantidad = async (datos) => datos.length;

async function Proceso()
{
    let posts = await obtenerPosts();
    posts = await filtrarPorUsuario('aperez', posts);

    const cantidad = await obtenerCantidad(posts);

    console.log('Cantidad de posts de aperez: ', cantidad);

    return cantidad;
}
```

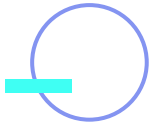
El ejemplo anterior es el mismo ejemplo del manual anterior, reescrito con **`async functions`**. A primera vista, hay un cambio: no se usó **`new Promise`**.

JavaScript incluye la palabra reservada **`async`**, al decorar una función con dicha palabra automáticamente se transforma en una tarea asincrónica, devolviendo una promesa que, al resolver, retorna dicho valor.

`() => number;`	`async () => Promise<number>`
`() => Post;`	`async () => Promise<Post>`
`(id) => User;`	`async (id) => Promise<User>`

Dentro de una **función asincrónica** podemos utilizar el operador **`await.`** Es un operador unitario -se opera sobre un solo dato- y realiza las siguientes acciones:

- **Ejecuta** la promesa dada.
- **Pausa la ejecución** de la función hasta que la promesa se haya resuelto (por eso solo se puede ejecutar en funciones asincrónicas).
- Al resolverse la promesa dada, **`await` retorna el valor** pasado como argumento a la función resolver (**`resolver(12) => 12 = await...`**).
- En caso de que la promesa se rechace, **`await` termina la ejecución** de la función y **ejecuta el `.catch`** asociado al retorno de a función.



¡Sigamos trabajando!