ThePower business school

🇪🇸 Promesas JS😇🤖

Las promesas son una forma de manejar la asincronía en JavaScript. Una promesa es un objeto que representa la eventual finalización o el fracaso de una operación asíncrona. Las promesas proporcionan una forma más sencilla y clara de trabajar con operaciones asíncronas en comparación con el uso de callbacks.

Una promesa tiene tres posibles estados:

- Pendiente: significa que la operación asíncrona aún no ha finalizado ni ha fallado.
- Cumplida: significa que la operación asíncrona se ha completado con éxito y la promesa tiene un valor.
- Rechazada: significa que la operación asíncrona ha fallado y la promesa tiene un error.

Cuando se crea una promesa, se pasa una función a la que se le llama inmediatamente con dos argumentos: `resolve` y `reject`. `Resolve` es una función que se utiliza para indicar que la promesa se ha cumplido y `reject` se utiliza para indicar que la promesa ha fallado.

A continuación se muestra un ejemplo de cómo se crea y se utiliza una promesa:

```
const promesa = new Promise((resolve, reject) => {
  // La operación asíncrona va aquí

  if (/* operación exitosa */) {
    resolve(/* valor opcional */);
  } else {
    reject(/* error opcional */);
  }
});

promesa.then((valor) => {
  // La promesa se ha cumplido
}, (error) => {
  // La promesa ha fallado
});
```

También se puede utilizar el método `catch` para manejar el error en lugar de proporcionar un segundo manejador al método `then`:

```
promesa.then((valor) => {
  // La promesa se ha cumplido
}).catch((error) => {
  // La promesa ha fallado
});
```

Además, las promesas tienen varios métodos estáticos útiles para trabajar con ellas. Por ejemplo, el método `Promise.all` toma una matriz de promesas y espera a que todas se cumplan o una de ellas falle. El método `Promise.race` espera a que la primera promesa de una matriz se cumpla o falle. Y el método `Promise.resolve` crea una promesa cumplida con un valor específico y `Promise.reject` crea una promesa rechazada con un error específico.

Manejando una promesa

Una **Promise** (promesa en castellano) es un objeto que representa la terminación o el fallo de una operación asíncrona. Surgen en ES6 para mejorar el proceso de callbacks.

Por lo general en nuestro proyectos NO vamos a **crear** promesas, vamos a consumirlas (**then/catch**), pero veamos la base de las promesas:

```
const addItem = (item, list) => {
  const promise = new Promise((resolve, reject) => {
    if (!list) {
      reject('No existe el array');
    }

    setTimeout(function () {
      list.push(item);
      resolve(list);
    }, 2000);
  });

  return promise;
};

const list = ['Rojo', 'Azul', 'Verde'];

addItem('Amarillo', list)
  .then((list) => {
    console.log(`El listado final es: ${list.join(', ')}`);
  })
  .catch((err) => {
    throw new Error(err);
  });
```

Ahora la función **addItem** crea un objeto **Promise** que recibe como parámetros una función con las funciones **resolve** y **reject**. Llamaremos a **resolve** cuando nuestra ejecución finalice correctamente, y a **reject** para indicar que ha habido un rechazo (error) en la ejecución.

De esta manera, podemos escribir código de manera más elegante, y el *Callback Hell* anterior puede ser resuelto así:

```
const list = ['Raising Arizona','Fargo','Barton Fink'];

addItem('The big Lewoski', list)
  .then(() => addItem('O Brother, Where Art Thou?', list))
  .then(() => addItem('The Man Who Wasnt There', list))
  .then(() => addItem('The Ladykillers', list))
  .then(() => {
    console.log(list);
  });

// (4 seg. de delay) -> ['Raising Arizona','Fargo','Barton Fink', ...];
```

Esto es conocido como ***anidación promesas***.

La forma de **tratar errores en una promesa**, es por medio de la **función catch** que **recoge** lo que **enviamos** en la **función reject** dentro de la Promesa. Y esta función solo hay que invocarla una vez, no necesitamos comprobar en cada llamada si existe error o no. Lo cual reduce mucho la cantidad de código:

```
const filmography = '';
addToCoenBrothers('The big Lewoski', filmography)
  .then(...)
  .then(...)
  .then(...)
  .catch(err => console.log(err.message));

// No existe el array -> es un string - salta error
```

Repasamos

Lo más habitual es que **consumamos** promesas ya creadas, pero vamos a repasar cómo se crearía una promesa.

```
// ES5
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(function() {
    resolve("done")
  }, 1000);
});

// ES6
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(
  (result) => {
    // Manejamos el resultado
    console.log(result);
  },
  (err) => {
    // Manejamos el reject concreto
    console.error(err);
  }
);

// Error general
promise.catch(
  (err) => {
    // Manejamos el error
    console.error(err);
  }
);
```

Si nos fijamos, **then(function, function)** puede recibir 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.

Es decir, podemos controlar también el fallo de una promesa concreta, dependiendo de la necesidad usaremos el control específico o el **catch**.

Rechazo de promesas

En el ejemplo anterior hemos visto qué ocurre cuando nuestro código cumple una promesa a raíz de una petición de forma asíncrona, pero no cuando esta promesa es rechazada.

Si se diera el caso de que la URL utilizada no exista o haya algún tipo de problema al realizar la petición o cualquier otra funcionalidad de nuestro código, la ejecución rompería dando un error no-controlado. Para poder tener el control cuando una promesa es rechazada haremos uso del método **catch.**

```
fetch(URL)
  .then(function(respuesta) {
  console.log(respuesta)
  //Se ejecuta cuando la promesa se resuelve
})
.catch(function(error){
  console.log("No se puede realizar la petición")
  //Se ejecuta cuando la promesa es rechazada
})
```

Mediante el método catch tenemos una “red de seguridad” a la hora de encontrarnos con un rechazo en una promesa, por lo que nuestro código no romperá, si no que ejecutará lo que le indiquemos dentro de nuestro catch atrapando esa incidencia. Es una especie de “else” para la ejecución promesas.

Finally

Mediante el método finally haremos que la función termine tanto si se cumple la promesa como si se rechaza, creando una ruptura en el código una vez hayamos terminado un bloque de ejecución cerrado.

Vamos a ver como utilizar todo lo que hemos visto con arrow functions:

```
fetch(URL)
  .then(respuesta => console.log(respuesta))
  .finally(() => console.log("Fetch terminado"))
  .catch(error => console.error(respuesta));
```

De esta manera estaremos cerrando la ejecución de nuestro fetch con un finally en el caso de que se cumpla como si no, y recogiendo nuestro error en el caso de que la promesa no se cumpla.

`Promise.all`

El método `Promise.all` toma una matriz de promesas y espera a que todas se cumplan o una de ellas falle. Devuelve una nueva promesa que se cumple con una matriz de valores cuando todas las promesas se han cumplido o se rechaza con el primer error encontrado si una de las promesas falla.

Aquí tienes un ejemplo de cómo se utiliza `Promise.all`:

```
const promesa1 = Promise.resolve(1);
const promesa2 = Promise.resolve(2);
const promesa3 = Promise.resolve(3);

Promise.all([promesa1, promesa2, promesa3]).then((valores) => {
  console.log(valores); // [1, 2, 3]
});
```

En este caso, `Promise.all` espera a que se cumplan todas las promesas de la matriz y, cuando lo hacen, se llama al manejador `then` con una matriz de valores. Si una de las promesas falla, `Promise.all` se rechaza inmediatamente con el error de esa promesa y no espera a que se cumplan las demás promesas.

`Promise.race`

El método `Promise.race` espera a que la primera promesa de una matriz se cumpla o falle. Devuelve una nueva promesa que se cumple con el valor de la primera promesa cumplida o se rechaza con el error de la primera promesa fallida.

Aquí tienes un ejemplo de cómo se utiliza `Promise.race`:

```
const promesa1 = new Promise((resolve, reject) => {
  setTimeout(() => resolve('Ganadora'), 100);
});
const promesa2 = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error('Perdedora')), 50);
});

Promise.race([promesa1, promesa2]).then((valor) => {
  console.log(valor); // 'Ganadora'
}).catch((error) => {
  console.error(error);
});
```

En este caso, `Promise.race` espera a que se cumpla o falle cualquiera de las dos promesas. Como la promesa2 falla antes que la promesa1, `Promise.race` se rechaza con el error de la promesa2. Si la

promesa1 se hubiera cumplido antes que la promesa2, `Promise.race` se habría cumplido con el valor de la promesa1.