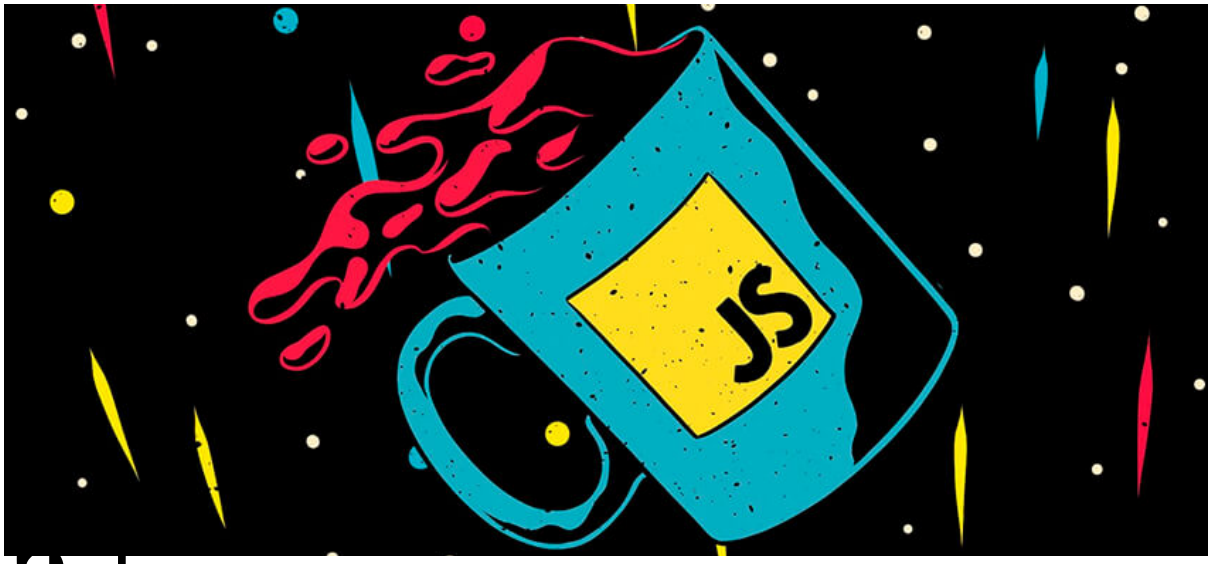

Javascript S6 | Manejando la asincronía

Después de esta lección podrás:

1. Manejo de la asincronía en Js.
2. Diferenciar entre síncrono y asíncrono.
3. Trabajar con Callbacks.
4. Entender y realizar Promesas.

La evolución de Javascript nos permite manejar llamadas asíncroncas y manejarla a tavés de algunas funciones. Normalmente se suele utilizar en operaciones de entrada o salida de datos como escritura de un JSON o la lectura de disco pero donde realmente se ve su uso son con las peticones AJAX.

Vamos a ver cómo se han gestionado hasta 2015 y con la aparición de ES6 el estándar actual, eso no quita que os encontréis situaciones o código que usen la forma más tradicional. Veamos cuales son.

Callbacks

Es la primera y la forma más común de controlar la asincronía en JavaScript hasta 2015, y como siempre la mejor forma de comprender algo es con un ejemplo.

```
// Pasamos 3 parámetros - String - Array - Callback
function addToCoenBrothers (film, filmography, callback) {
  // NO existe el Array
  if (!filmography) {
    // FUNCIÓN de callback crea un ojbeto que recibe un string, null
    callback(new Error('No existe el array'), null);
  } else {
    // SI existe Array - Pusheamos nuestro String
    filmography.push(film)
    // Callback recibe null, Array
    callback(null, filmography);
  }
}
```

Vamos a ver qué ha pasado en el siguiente ejemplo anterior. Nuestra **función que recibe como parámetros** un **dato de entrada**: `film`, un **array con datos** `filmography` y una **función de callback**: `callback`.

Con estos tres parámetros hacemos lo siguiente, es muy sencillo, a `filmography` **se le añade** `film` que **viene por parámetro** y cuando **termine**, **llama a la función de** `callback` que recibe por parámetro, en ese caso la llama con la `filmography` modificada.

Hemos añadido un pequeño bloque para comprobar si la ***filmography*** existe y si no lanzar un error que pasaremos al callback. El control y manejo de errores lo veremos más adelante de momento tenéis que confiar en nosotros.

Ahora vamos a ejecutar nuestra función para ver cómo tratar el ***callback*** y veamos qué está pasando:

```
let filmography = ['Raising Arizona','Fargo','Barton Fink'];

addToCoenBrothers('The Big Lebowski', filmography, function (err) {
  if (err){
    return console.log(err.message)
    console.log(filmography)
  }
});

// ["Raising Arizona", "Fargo", "Barton Fink", "The Big Lebowski"]
```

¿Qué ha pasado? Cuando se ha termiando de ejecutar `addToCoenBrothers` se ejecuta el ***callback*** y nuestro array ***filmography*** tiene un nuevo dato.

Frente a eso me diréis que podemos añadir el dato al array y después hacer un ***console.log***, pero qué hubiese sucedido si queremos añadir un dato a un array que aún no tenemos? en esto consiste la asíncrona, como son las peticiones vía AJAX.

Como hasta la próxima sesión no veremos las peticiones vamos a simularlo con la función `setTimeout` para añadir un retardo de 1 segundo:

```
// Pasamos 3 parámetros - String - Array - Callback
function addToCoenBrothers (film, filmography, callback) {
  // NO existe el Array
  if (!filmography) {
    // FUNCIÓN de callback crea un ojbeto que recibe un string, null
    callback(new Error('No existe el array'), null);
  }
    // SI existe Array - Pusheamos nuestro String
    setTimeout(function() {
      // Callback recibe null, Array
      filmography.push(film)
      callback(null, filmography)
    }, 1000);
}

let filmography = ['Raising Arizona','Fargo','Barton Fink'];

addToCoenBrothers('The Big Lebowski', filmography, function (err) {
  if (err){
    return console.log(err.message)
    console.log(filmography)
  }
```

```
});

// (1 seg)-> ["Raising Arizona", "Fargo", "Barton Fink", "The Big Lebowski"]
```

Si no tuviéramos una función de ***callback***, y la función `addToCoenBrothers` fuera:

```
function addToCoenBrothers (film, filmography) {
  setTimeout(function() {
    filmography.push(film)
  }, 1000)
}
```

y ejecutáramos la función, nos devolvería lo siguiente:

```
let filmography = ['Raising Arizona','Fargo','Barton Fink'];

addToCoenBrothers('The Big Lewoski', filmography);

console.log(filmography);

// ['Raising Arizona','Fargo','Barton Fink']
```

> Cuando imprimimos el array aún no se ha añadido el nuevo item, por lo tanto el comportamiento que sucede no es el buscado. De esta forma los callbacks nos ayudan a que esto no suceda. En resumen nos ayuda a manejar la asincronía.

Pero si tenemos varias funciones así... puede ocurrir lo siguiente:

```
let filmography = ['Raising Arizona','Fargo','Barton Fink'];

addToCoenBrothers('The Big Lewoski', filmography, function (err) {
```

```
  if (err) ...
  addToCoenBrothers('O Brother, Where Art Thou?', filmography, function (err) {
    if (err) ...
    addToCoenBrothers('The Man Who Wasnt There', filmography, function (err) {
      if (err) ...
      addToCoenBrothers('The Ladykillers', filmography, function () {
        // Y podemos seguir con su extensa filmografía...
      })
    })
  })
});
```

A esto se le conoce como **Callback Hell** o **Pyramid of Doom.**

Promesas

Una Promise (promesa en castellano) es un objeto que representa la terminación o el fallo de una operación asíncrona. ¿Muy parecido a los callbacks no?

Veamos el mismo ejemplo que antes pero utilizando Promesas nativas de ES2015:

```
// Función por parametro tiene un String - Array
function addToCoenBrothers (film, filmography) {
  // Creamos una promesa
  const promise = new Promise(function (resolve, reject) {
    setTimeout(function() {
      filmography.push(film)
      resolve(filmography)
    }, 1000);

    if (!filmography) {
      reject(new Error('No existe filmography'))
    }
  })

  return promise
}

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

addToCoenBrothers('The big Lewoski', filmography).then(function () {
  console.log(filmography)
})
```

Ahora la función `addToCoenBrothers` crea un objeto `Promise` que recibe como parámetros una función con las funciones `resolve` y `reject`. Llamaremos a `resolve` cuando nuestra ejecución finalice correctamente.

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

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

addToCoenBrothers('The big Lewoski', filmography)
  .then(function() { return addToArray('O Brother, Where Art Thou?', filmography) })
  .then(function() { return addToArray('The Man Who Wasnt There', filmography) })
  .then(function() { return addToArray('The Ladykillers', filmography) })
  .then(function () {
    console.log(filmography)
  })

// (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
```

Async/Await

Hay una sintaxis especial para trabajar con las promesas de una manera más cómoda, llamada ***async / wait***. Es sorprendentemente fácil de entender y usar. La sintaxis para una función que utilice ***async/await*** es la siguiente:

```
async function myFuncion () {
  try {
    var result = await functionAsincrona()
  } catch (err) {
    ...
  }
}
```

La función irá precedida por la palabra reservada `async` y dentro de ella tendremos un bloque `try-catch`. Dentro del `try` llamararemos a la función asíncrona con la palabra reservada `await` delante, con esto hacemos que la función *espere* a que se ejecute y el resultado de la misma está disponible en este caso en la variable `result`.

Si ocurre algún error durante la ejecución, se ejecutará el bloque `catch` donde trataremos el error. Combinando async/await con una función basada en Promesas, podemos hacer lo siguiente con el ejemplo que estábamos viendo:

```
// Función por parametro tiene un String - Array
function addToCoenBrothers (film, filmography) {
  // Creamos una promesa
  const promise = new Promise(function (resolve, reject) {
    setTimeout(function() {
      filmography.push(film)
      resolve(filmography)
    }, 1000);

    if (!filmography) {
      reject(new Error('No existe filmography'))
    }
  })
```

```
  return promise
}

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

async function processFilm (film, filmography) {
  try {
    const result = await addToArray(film, filmography);
    console.log(result)
  } catch (err) {
    return console.log(err.message);
  }
}

processFilm('The big Lewoski', filmography)
// ['Raising Arizona','Fargo','Barton Fink', 'The big Lewoski']
processFilm('O Brother, Where Art Thou?', filmography)
// ['Raising Arizona','Fargo','Barton Fink', 'The big Lewoski'...]
processFilm('The Ladykillers', filmography)
// ['Raising Arizona','Fargo','Barton Fink', 'The big Lewoski'...]
```

De esta manera estamos escribiendo código de manera secuencial pero JavaScript está por debajo ejecutando código asincrónico.

Promise all

Por último veremos como sería resolver varias promesas. Encadenaremos varias promesas y haremos que se resuelvan todas a la vez, cuando estén resultas nos devolverá nuestra "agrupadora de promesas" resuelta. Veamos un ejemplo:

```
let theBigLewoski = Promise.resolve('The big Lewoski');
var theLadyKillers = 'The Lady Killers';
var trueGrit = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'True Grit');
});

Promise.all([theBigLewoski, theLadyKillers, trueGrit]).then(films => {
  console.log(films);
  // ['The big Lewoski', 'The Lady Killers', 'True Grit']
});
```

Promise.all espera a que todo se cumpla (o bien al primer rechazo). Rechaza si uno de los elementos ha sido rechazado y ***Promise.all*** falla rápido: Si tienes cuatro promesas que se resuelven después de un timeout y una de ellas falla inmediatamente, entonces ***Promise.all*** se rechaza inmediatamente.

Veamos un ejemplo de ***Promise.all*** si alguna de las promsesas es rechazada:

```
var theBigLewoski = new Promise((resolve, reject) => {
  setTimeout(resolve, 1000, 'The big Lewoski');
});
var theLadyKillers = new Promise((resolve, reject) => {
  setTimeout(resolve, 2000, 'The Lady Killers');
});
var intolerableCruelty = new Promise((resolve, reject) => {
  setTimeout(resolve, 3000, "Intolerable Cruelty");
});
var trueGrit = new Promise((resolve, reject) => {
  setTimeout(resolve, 4000, 'True Grit');
});
var joJoRabbit = new Promise((resolve, reject) => {
  reject('jo jo rabbit - error Taika Waititi');
});

Promise.all([theBigLewoski, theLadyKillers, intolerableCruelty,...])
  .then(films => {
    console.log(films);
  }, reason => {
    console.log(reason)
});

// From console:
// 'jo jo rabbit - error Taika Waititi'

// Evenly, it's possible to use .catch
Promise.all([theBigLewoski, theLadyKillers, intolerableCruelty,...]).then(films => {
  console.log(films);
}).catch(reason => {
  console.log(reason)
});

// From console:
// 'jo jo rabbit - error Taika Waititi'
```