Programación asíncrona en Javascript

Promesas

https://platzi.com/blog/que-es-y-como-funcionan-las-promesas-en-javascript/

Introducción

Manejar flujos de datos asíncronos es complejo, quién no ha terminado con código como este:

```
checkWeather('buenos aires', (error, weather) => {
      if (error) throw error;

      if (weather === 'well') {
            return checkFlights('buenos aires', (err, flights) => {
                  if (err) throw err;

                  buyTicket(flights[0], (e, ticket) => {
                        if (e) throw e;
                        console.log('ticket n° %d', ticket.number);
                  });
            });
      }

      console.log('el clima es malo');
});
```

Esto se conoce como el callback hell, un código complejo y difícil de mantener, pero existe una solución, Promesas. El código anterior con promesas sería algo similar a esto:

```
checkWatcher('buenos aires')
      .then(weather => {
            if (weather === 'well') {
                  return checkFlights('buenos aires');
            }
            throw new Error('el clima es malo');
      })
      .then(flights => buyTicket(flights[0]))
      .then(ticket => {
            console.log('ticket n° %d', ticket.number);
      })
      .catch(error => console.error(error));
```

Menos líneas y mucha menos indentación en el código, pero entendamos como funciona una promesa. Pongamos otro ejemplo. Supongamos que vamos a comprar comida a un restaurante de comida rápida, cuando terminamos de pagar por nuestra comida nos dan un ticket con un número, cuando llamen a ese número podemos entonces ir a buscar nuestra comida.

Ese ticket que nos dieron es nuestra promesa, ese ticket nos indica que eventualmente vamos a tener nuestra comida, pero que todavía no la tenemos. Cuando llaman a ese número para que vayamos a buscar la comida entonces quiere decir que la promesa se completó. Pero resulta que una promesa se puede completar correctamente o puede ocurrir un error, ¿Qué error puede ocurrir en nuestro caso? Por ejemplo puede pasar que el restaurante no tenga más comida, entonces cuando nos llamen con nuestro número pueden pasar dos cosas.

1. Nuestro pedido se resuelve y obtenemos la comida.
2. Nuestro pedido es rechazado y obtenemos una razón del por qué.

Pongamos esto en código:

```
const ticket = getFood();

ticket
      .then(food => eatFood(food))
      .catch(error => getRefund(error));
```

Cuando tratamos de obtener la comida (getFood) obtuvimos una promesa (ticket), si esta se resuelve correctamente entonces recibimos nuestra comida (food) y nos la comemos (eatFood). Si nuestro pedido es rechazado entonces obtenemos la razón (error) y pedimos que nos devuelvan el dinero (getRefund).

Promesas

Crear una promesa

Las promesas se crean usando un constructor llamado Promise y pasándole una función que recibe dos parámetros, resolve y reject, que nos permiten indicarle a esta que se resolvió o se rechazó.

```
const promise = new Promise((resolve, reject) => {
      const number = Math.floor(Math.random() * 10);

      setTimeout(
            () => number > 5
                  ? resolve(number)
                  : reject(new Error('Menor a 5')),
            1000
      );
});

promise
      .then(number => console.log(number))
```

```
      .catch(error => console.error(error));
```

Lo que acabamos de hacer es crear una nueva promesa que se va a completar luego de 1 segundo, si el número aleatorio que generamos es mayor a 5 entonces se resuelve, si es menor a 5 entonces es rechazada y obtenemos un error.

Estados de las promesas

Esto nos lleva a hablar del estado de una promesa, básicamente existen 3 posibles estados.

- Pendiente
- Resuelta
- Rechazada

Una promesa originalmente esta Pendiente. Cuando llamamos a resolve entonces la promesa pasa a estar Resuelta, si llamamos a reject pasa a estar Rechazada, usualmente cuando es rechazada obtenemos un error que nos va a indicar la razón del rechazo. Cuando una promesa se resuelve entonces se ejecuta la función que pasamos al método .then, si la promesa es rechazada entonces se ejecuta la función que pasamos a .catch, de esta forma podemos controlar el flujo de datos.

También es posible pasar una segunda función a .then la cual se ejecutaría en caso de un error en vez de ejecutar el .catch

Recibiendo parámetros

Antes creamos una promesa, esa promesa se completa luego de 1 segundo y se resuelve si el número generado es mayor a 5. ¿Qué pasa si queremos hacerlo dinámico? La solución es muy simple, creamos una función que recibe los parámetros necesarios y devuelve la instancia de Promise.

```
function randomDelayed(max = 10, expected = 5, delay =  1000) {
      return new Promise((resolve, reject) => {
            const number = Math.floor(
                  Math.random() * max)
            );

            setTimeout(
                  () => number > expected
                        ? resolve(number)
                        : reject(new Error('número menor al esperado'));
                  delay
            );
      });
}

randomDelayed(100, 75, 2500)
    .then(number => console.log(number))
    .catch(error => console.error(error));
```

Cuando ejecutamos randomDelayed(100, 75, 2500) creamos una promesa que luego de 2.5 segundos se va a resolver siempre que el número generado (entre 0 y 100) sea mayor a 75. Lo mismo que habíamos hecho antes, pero esta vez personalizable.

Pasando de callback a promesas

¿Qué ocurre si una función que queremos utiliza callbacks? ¿Cómo podríamos usarla con promesas? Muy simple, podemos crear una versión con promesas de esa función haciendo lo que hicimos arriba. Por ejemplo leer una archivo usando el módulo fs de Node.js.

```
import fs from 'fs';

function readFile(path) {
      return new Promise((resolve, reject) => {
            fs.readFile(path, 'utf8', (error, data) => {
                  if (error) return reject(error);
                  return resolve(data);
            });
      });
}

readFile('./archivo.txt')
      .then(data => console.log(data))
      .catch(error => console.error(error));
```

De esta forma creamos una función que lee un archivo del disco como utf8 y si no ocurre ningún error entonces se resuelve, si hay un error es rechazada.

Encadenando promesas

En el primer ejemplo de promesas vimos algo muy interesante, usamos muchos .then y llamamos a varias funciones que devuelven promesas. Este patrón se llama promise chaining o encadenamiento de promesas.

Básicamente nos evita anidar código, en vez de eso una promesa puede devolver otra promesa y llamar al siguiente .then de la cadena. Veamos un ejemplo, supongamos que archivo.txt devuelve un string con el path de otro archivo, y queremos leer este segundo archivo, con callbacks quedaría algo así:

```
fs.readFile('./archivo.txt', 'utf8', (error, path) => {
      if (error) throw error;
      fs.readFile(path, 'utf8', (err, data) => {
            console.log(data);
      });
});
```

Como vemos dentro de nuestro primer callback tenemos que validar el primer error, luego llamar a otra función que obtiene los datos de verdad, y si tenemos que ir anidando muchas funciones que usen callback podemos llegar a tener muchos niveles de indentación. Con promesas esto quedaría así:

```
readFile('./archivo.txt')
      .then(readFile)
      .then(data => console.log(data))
      .catch(error => console.error(error));
```

Qué hacemos acá, primero leemos ./archivo.txt, si ocurre un error esta promesa se rechaza y lo mostramos en el console.error, si todo va bien se ejecuta el primer .then, este ejecuta un nuevo readFile, como .then recibe el path al nuevo archivo y readFile solo recibe un argumento (el path) entonces podemos pasar directamente readFile y la promesa se encarga de ejecutarlo.

Este segundo readFile devuelve una nueva promesa, otra vez si hay un error se ejecuta el .catch, pero si podemos leer el archivo sin problema entonces se ejecuta el segundo .then, el cual recibe el contenido del segundo archivo y lo muestra en consola.

Como vemos, podemos simplemente encadenar tantos .then como queramos y seguir ejecutando funciones que devuelvan promesas. ¿Lo mejor? No solo hay que devolver promesas, ya que si la función que pasamos a .then hace un return entonces el valor devuelto pasa al siguiente .then de la cadena, sin importar que sea una promesa, un objeto, un string, un número o cualquier otro tipo de datos. Por ejemplo:

```
import { resolve } from 'path';

readFile('./archivo.txt.')
      .then(resolve)
      .then(readFile)
      .then(data => console.log(data))
      .catch(error => console.error(error));
```

La función resolve que exporta el módulo path nos permite armar la ruta absoluta a un archivo y devuelve un string, gracias a los .then podemos hacer una función que recibe el fileName del primero archivo y luego devuelve esta ruta absoluta la cual llega como parámetro al segundo .then, el cual usa esa ruta para leer el segundo archivo y devuelve una promesa que se resuelve con el contenido de este. Y como siempre si en algún momento hay ocurre un error se ejecuta el .catch.

Promesas en paralelo

Hasta ahora solo vimos como ejecutar una función asíncrona a la vez (en serie), sin embargo es muy común que necesitemos realizar múltiples al tiempo, por ejemplo para obtener varios datos de un API. Para eso la clase Promise tiene un método estático llamado Promise.all el cual recibe un único parámetro, una lista de promesas las cuales se ejecutan simultáneamente, si alguna de estas es rechazadas entonces toda la lista lo es, pero si todas se resuelven entonces podemos obtener una lista de todas las respuestas.

```
import { resolve } from 'path';

Promise.all([readFile('./archivo1.txt'), readFile('./archivo2.txt')])
      .then(data => data.map(resolve))
      .then(data => Promise.all(data.map(readFile)))
      .then(finalData => console.log(finalData))
      .catch(error => console.error(error));
```

Lo que hacemos en el ejemplo de arriba es leer 2 archivos al tiempo, eso nos devuelve una lista (data) de contenidos, los cuales contienen la ruta para otro archivo, los convertimos entonces a una nueva lista de rutas absolutas (resolve) y usamos esas rutas para crear una nueva lista de promesas a partir de readFile. Si en algún momento ocurrió un error lo mostramos como tal en consola, si todo se resuelve bien entonces escribimos en consola la lista de contenidos de archivos.

Carrera de promesas

Antes hablamos de ejecutar varias promesas en paralelo y obtener una respuesta cuando todas se completen, existe otro método que nos permite correr varias al tiempo, pero solo obtener el resultado de la primer promesa. Gracias a esto es posible mandar múltiples peticiones HTTP a un API y luego recibir una sola respuesta, la primera. Este método se llama Promise.race.

```
import { resolve } from 'path';

Promise.race([readFile('./archivo1.txt'), readFile('./archivo2.txt'])
      .then(resolve)
      .then(readFile)
      .then(data => console.log(data)
      .catch(error => consol.error(error));
```

Como vemos en el ejemplo otra vez leemos 2 archivos, pero esta vez solo obtenemos el contenido de 1, el que primero se termine de leer. O si alguno se completó con un error entonces entramos al catch y mostramos el error en consola.

Promesas resueltas inmediatamente

Algunas veces la forma de manejar el flujo de datos de las promesas encadenando then nos facilita trabajar con nuestro código, para eso podemos crear una promesa que inicie resuelta directamente usando un método estático de Promise.

```
Promise.resolve()
      .then(() => {
            // acá podemos hacer lo que queramos
      });
```

Otra opción es pasarle un parámetro a resolve para que nuestro primer then reciba ese valor.

```
import { resolve } from 'path';

Promise.resolve('./archivo1.txt')
      .then(resolve)
      .then(readFile)
      .then(data => console.log(data))
      .catch(error => console.error(error));
```

Como vemos iniciamos la cadena con un string y desde ahí obtenemos el path absoluto, leemos el archivos y lo mostramos en consola.

Promesas rechazadas inmediatamente

De la misma forma que creamos promesas resuelta inmediatamente podemos crear promesas rechazadas. Solo que esta vez usamos Promise.reject.

```
Promise.reject(new Error('Nuestro error'))
      .then(() => {
            // esta función jamás se ejecuta
      })
      .catch(error => console.error(error));
```

¿Para qué nos sirve esto? Si tenemos una función síncrona que queremos resolver mediante promesas podríamos si da un error devolver Promise.reject con el error y Promise.resolve con la respuesta correcta. De esta forma en vez de crear una instancia de la clase Promise simplemente devolvemos la promesa ya resuelta o ya rechazada.

async/await

La declaración de función async define una función asíncrona, la cual devuelve un objeto AsyncFunction.

Es posible definir también funciones asíncronas a través de una expresión de función async.

Sintaxis

```
async function name([param[, param[, ... param]]]) {
   statements
}
```

Parámetros

- name: El nombre de la función.
- Param: El nombre de un argumento que se debe pasar a la función.
- Statements: Las declaraciones que conforman el cuerpo de la función.

Valor de retorno

Un objeto AsyncFunction, que representa una función asíncrona que ejecuta el código contenido dentro de la función.

Descripción

Cuando se llama a una función async, esta devuelve un elemento Promise. Cuando la función async devuelve un valor, Promise se resolverá con el valor devuelto. Si la función async genera una excepción o algún valor, Promise se rechazará con el valor generado.

Una función async puede contener una expresión await, la cual pausa la ejecución de la función asíncrona y espera la resolución de la Promise pasada y, a continuación, readuna la ejecución de la función async y devuelve el valor resuelto.

La finalidad de las funciones async/await es simplificar el comportamiento del uso síncrono de promesas y realizar algún comportamiento específico en un grupo de Promises. Del mismo modo que las Promises son semejantes a las devoluciones de llamadas estructuradas, async/await se asemejan a una combinación de generadores y promesas.

Ejemplos

Ejemplo sencillo

```
function resolveAfter2Seconds(x) {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve(x);
    }, 2000);
  });
}

async function add1(x) {
  const a = await resolveAfter2Seconds(20);
  const b = await resolveAfter2Seconds(30);
```

```
  return x + a + b;
}

add1(10).then(v => {
  console.log(v);  // prints 60 after 4 seconds.
});

async function add2(x) {
  const p_a = resolveAfter2Seconds(20);
  const p_b = resolveAfter2Seconds(30);
  return x + await p_a + await p_b;
}

add2(10).then(v => {
  console.log(v);  // prints 60 after 2 seconds.
});
```

No se deben confundir await y Promise.all

En add1, la ejecución se suspende durante dos segundos correspondientes al primer operador await, y luego durante otros dos segundos correspondientes al segundo await. El segundo temporizador no se crea hasta que el primero no se haya disparado ya. En add2, ambos temporizadores se crean y, acto seguido, ambos reciben await. Esto provoca la resolución en dos segundos y no cuatro, ya que los temporizadores se ejecutaron de manera simultánea. Sin embargo, ambas llamadas await aún pueden ejecutarse en series, no en paralelo: esto no constituye ninguna aplicación automática de Promise.all. Si se desea aplicar await a dos o más promesas en paralelo, es preciso utilizar Promise.all.

Cadena de promesas

Reescritura de una cadena de promesas con una función async

Una API que devuelva una Promise tendrá como resultado una cadena de promesas, y dividirá la función en muchas partes. Estudie este código:

```
function getProcessedData(url) {
  return downloadData(url) // returns a promise
    .catch(e => {
      return downloadFallbackData(url)  // returns a promise
    })
    .then(v => {
      return processDataInWorker(v); // returns a promise
    });
}
Es posible reescribirlo utilizando un solo operador async de esta manera:

async function getProcessedData(url) {
  let v;
  try {
    v = await downloadData(url);
  } catch(e) {
    v = await downloadFallbackData(url);
```

```
  }
  return processDataInWorker(v);
}
```

Observe que, en el ejemplo anterior, no hay ninguna instrucción await dentro de la instrucción return, porque el valor de retorno de una async function queda implícitamente dentro de un Promise.resolve.