Decálogo de errores comunes

1. *Thens* consecutivos vs. *Thens* paralelos

Cuando se está utilizando una *Promise* en cadena no se puede repetir la promesa delante de cada manejador *.then()* o *.catch(),* ya que, de ser así, el valor resultante no se encadenará con el siguiente. En este último caso, el método *.then()* crearía una *Promise* completamente nueva e independiente, es decir, se ejecutararian las promesas en paralelo.

En el siguiente ejemplo, sería necesario eliminar todos los *ten,* excepto el primero, para que el *calback result* funcionase como valor acumulado. Pero, en caso contrario, si queremos que cada *.then()* reciba el mismo argumento y sean llamadas independientes a una promesa, el código resultante tendría que ser algo así:

```
const ten = new Promise((resolve, reject) => {
    resolve(10);
});

ten
.then((result) => {
   // returns 20
    return result + 10;
})
ten
.then((result) => {
   // returns 100
    return result * 10;
})
ten
.then((result) => {
   // returns 0
    return result - 10;
})
ten
.then((result) => {
   // logs 10 in the console.
    console.log(result);
});
```

2. Olvidarse del *callback* en *.then()*

El método *.then()* toma dos funciones *callback* como argumentos, donde la primera se utiliza para manejar el caso resuelto y la segunda para el caso rechazado. Si en lugar de usar la función *callback*, utilizamos cualquier otro valor en su lugar, con toda probabilidad el resultado no será el esperado. Por ejemplo, en el siguiente código esperamos que la salida sea "Hola Mundo" pero el primer *.then()* no utiliza la función *callback* sino que asigna un *string*, por lo que directamente, el resultado se anula y sólo se pasa por consola el valor del segundo *callback*, es decir, "Hola":

```
const hello = Promise.resolve("Hello");
hello.then('World').then(result => console.log(result));
```

3. Manejar errores con promesas

El modo más conveniente de manejar errores en promesas es con el método *.catch()* acompañado siempre de uno o múltiples *.then().* Es un error muy común olvidarse de incluir el manejador *.catch()* o prencindir de usar el bloque *try-catch* con *async/await* para gestionar errores en *Promises,* algo bastante crucial ya que, los casos de prueba podrían fallar o bloquearse la aplicación durante la fase de producción. Hay que incluir siempre la gestión de errores en una aplicación para que ésta sea más resistente.

```
const oddEven = (num) => {
  return new Promise((resolve, reject) => {
    if (num % 2 === 0) {
      resolve("Even");
    } else {
      reject(new Error("Odd"));
    }
  });
};

oddEven(11).then((result) => {
    console.log(result);
}).catch((err) => {
    console.log(err.message);
});
```

4. *Promise Hell*

Las promesas se utilizan normalmente para evitar los *callbacks* anidados o *callback hell*. Sin embargo, un mal uso de éstas podría darse cuando, en lugar de utilizar *.then()* para devolver un resultado y manejar éste en el siguiente *.then()*, anidamos *Promises* hasta el punto en que el código podría volverse ilegible. Un ejemplo de una *Promise Hell* es:

```
userLogin('user').then(function(user){
    getArticle(user).then(function(articles){
        showArticle(articles).then(function(){
            //Your code goes here...
        });
    });
});
```

que podría desanidarse de este modo:

```
userLogin('user')
  .then(getArticle)
  .then(showArticle)
  .then(function(){
       //Your code goes here...
});
```

5. Usar el bloque *try/catch* dentro de la definición de una promesa

El bloque *try/catch* se utiliza para la gestión de errores pero no es necesario usarlo en el objeto *Promise*, ya que, si hay algún error, el objeto será capturado automáticamente en su ámbito sin necesidad del bloque *try/catch*. La propia *Promise* asegura que todas las excepciones lanzadas durante la ejecución son adquiridas y convertidas en una *Promise* rechazada. En el siguiente ejemplo, el bloque *try/catch* no debería estar dentro del objeto *Promise*:

```
new Promise((resolve, reject) => {
  try {
    const data = doThis();
    // do something
    resolve();
  } catch (e) {
    reject(e);
  }
})
  .then(data => console.log(data))
  .catch(error => console.log(error));
```

sino que el objeto *Promise* debería expresarse así:

```
new Promise((resolve, reject) => {
  const data = doThis();
  // do something
  resolve()
})
  .then(data => console.log(data))
  .catch(error => console.log(error));
```

6. Uso de una función asíncrona dentro de un bloque *Promise*

Se pueden dar efectos secundarios no deseados si utilizamos una función *async* dentro de un bloque *Promise*.

Por ejemplo, en el código que se muestra a continuación, se quiere realizar alguna tarea asíncrona dentro del bloque *Promise*, así que se añade la palabra clave *async*. El problema es que, si el código lanza un error, éste no se puede manejar tanto si hay un método *.catch()* como si se espera a la promesa dentro de un bloque *async/await*:

Error que no se puede manejar con un método *.catch()*

```
new Promise(async () => {
  throw new Error('message');
}).catch(e => console.log(e.message));
```

Error que no se puede manejar con un bloque *async/await*:

```
(async () => {
  try {
    await new Promise(async () => {
      throw new Error('message');
    });
  } catch (e) {
    console.log(e.message);
  }
})();
```

Para solucionar esto, es necesario separar la lógica asíncrona fuera del bloque *Promise* para hacer que el bloque sea síncrono. Y, si aún así es necesaria una función asíncrona, habrá que manejarla manualmente mediante un bloque *try/catch*.

Ejemplo de función asíncrona dentro de un bloque *Promise*:

```
new Promise(async (resolve, reject) => {
  try {
    throw new Error('message');
  } catch (error) {
    reject(error);
  }
}).catch(e => console.log(e.message));
```

y usando *async/await*:

```
(async () => {
  try {
    await new Promise(async (resolve, reject) => {
      try {
        throw new Error('message');
      } catch (error) {
        reject(error);
      }
    });
  } catch (e) {
    console.log(e.message);
  }
})();
```

7. No usar código *async/await*

En algunas ocasiones podría resultar más eficaz, ordenado y fácil de leer el uso de *async/await* en lugar de encadenar los métodos *.then()* y *.catch()* para manejar *Promises*. A continuación, podemos entender mejor esta diferencia con dos ejemplos comparativos que expresan lo mismo:

Promise sin *async/await*:

```
const promise = new Promise((resolve, reject) => {
  // Do something asynchronous
  if (somethingGoesWrong) {
    reject(new Error('Something went wrong!'));
  } else {
    resolve('Success!');
  }
});promise.then((result) => {
  // Handle success
}).catch((error) => {
  // Handle error
});
```

Promise con *async/await*:

```
async function doSomethingAsync() {
  try {
    const result = await promise;
    // Handle success
  } catch (error) {
    // Handle error
  }
}
```

8. Hacer un uso excesivo de promesas

Una cadena de promesas hacen que tu código sea más lento, por lo que es aconsejable no utilizarlas en exceso.

A veces encontramos código con una larga cadena de métodos *.then()*. Aunque el código pueda parecer más elegante, si utilizamos la herramienta de rastreo de algunos navegadores, nos encontramos con que hay varios bloques etiquetados como promesas y que cada uno de ellos tarda unos milisegundos en ejecutarse. Por lo tanto, cuantos más bloques de promesas haya, más tiempo tardarán en ejecutarse.

En estos fragmentos de código vemos cómo se podría simplificar un exceso de métodos *.then():*

Código con métodos *.then()* innecesarios:

```
new Promise((resolve) => {
  // Some code which returns userData
  const user = {
    name: 'John Doe',
    age: 50,
  };
  resolve(user);
}).then(userObj => {
  const { age } = userObj;
  return age;
}).then(age => {
```

```
  if (age > 25) {
    return true;
  }
  throw new Error('Age is less than 25');
}).then(() => {
  console.log('Age is greater than 25');
}).catch(e => {
  console.log(e.message);
});
```

Código con un uso de métodos *.then()* reducido:

```
new Promise((resolve, reject) => {
  // Some code which returns userData
  const user = {
    name: 'John Doe',
    age: 50,
  };
  if (user.age > 25) {
    resolve();
  } else {
    reject('Age is less than 25');
  }
}).then(() => {
  console.log('Age is greater than 25');
}).catch(e => {
  console.log(e.message);
});
```

9. Uso erróneo de *Promise.race()*

Promise.race() devuelve la promesa que primero se resuelva de un *array* de *Promises*, ya sea cumpliéndose o rechazándose. Esto no quiere decir que *Promise.race()* haga nuestro código más rápido o que el código salga inmediatamente despues de ser ejecutado, sino que veremos el código de la promesa, resuelta o rechazada, una vez que todas las promesas se hayan resuelto.

En el siguiente ejemplo, independientemente de qué función se devuelva o rechace, el tiempo que tardará aproximadamente en verse el mensaje por consola es de 3 segundos:

```
const { promisify } = require('util');
const sleep = promisify(setTimeout);

async function f1() {
  await sleep(1000);
}

async function f2() {
  await sleep(2000);
}

async function f3() {
  await sleep(3000);
```

```
}

(async () => {
  console.time('race');
  await Promise.race([f1(), f2(), f3()]);
})();

process.on('exit', () => {
  console.timeEnd('race'); // Approx 3 seconds
});
```

10. *Looping* con *Promises*

Un error muy común al utilizar promesas es manejarlas dentro de un bucle. Las promesas realizan operaciones asíncronas que pueden tardar un tiempo indeterminado en resolverse, por lo que dentro de un bucle, el tiempo de espera podría aumentar considerablemente.

Si utilizamos una promesa dentro de un *loop* del modo siguiente, podemos obtener un resultado no deseado. Por ejemplo, que el orden de las promesas no sea el mismo que en la cadena.

```
const users = ['saviomartin', 'victoria-lo', 'max-programming',
'atapas'];

const loopFetches = () => {
    for (let i = 0; i < users.length; i++) {
        console.log(`*** Fetching details of ${users[i]} ***`);
        const response = fetchData(users[i]);
        response.then(response => {
            response.json().then(user => {
                console.log(`${user.name} is ${user.bio} has $
{user.public_repos} public repos and ${user.followers} followers`);
            });
        });
    }
}

loopFetches();
```

```
*** Fetching details of saviomartin ***
*** Fetching details of victoria-lo ***
*** Fetching details of max-programming ***
*** Fetching details of atapas ***
Tapas Adhikary is Writer . YouTuber . Creator . Mentor has 83 public
repos and 348 followers
Savio Martin is Highly passionate frontend web developer - Love to
dream! 🚀 has 28 public repos and 881 followers
Victoria Lo is A software/web developer and technical blogger who's
passionate in learning about new technologies!

 has 40 public repos and 296 followers
Max Programming is Student, Programmer, Youtuber.
Learning Web Development
 has 117 public repos and 132 followers
```

Podríamos controlar el orden con el que obtenemos cada dato utilizando *async/await*, pero la ejecución de la promesa es secuencial y, ésta debería ser una actividad asíncrona. Este ejemplo devolvería por consola los datos ordenados pero de un modo secuencial:

```
const loopFetchesAsync = async () => {
    for (let i = 0; i < users.length; i++) {
        console.log(`=== Fetching details of ${users[i]} ===`);
        const response = await fetchData(users[i]);
        const user = await response.json();
        console.log(`${user.name} is ${user.bio} has ${user.public_repos}
public repos and ${user.followers} followers`);
    }
}
```

Solución: utilizar las APIs *Promise.all()* o *Promise.allSettled()* ya que toman un *array* de promesas, las ejecutan en paralelo y devuelven el resultado en el mismo orden de las entradas. Este ejemplo sería el modo más adecuado de tratar un bucle de *Promises*:

```
const loopAll = async () => {
    const responses = await Promise.all(users.map(user =>
fetchData(user)));
    const data = await Promise.all(responses.map(response =>
response.json()));
    console.log(data);
    data.map(user => {
        console.log(`*** Fetching details of ${user.name} ***`);
        console.log(`${user.name} is ${user.bio} has ${user.public_repos}
public repos and ${user.followers} followers`)
    });
}

loopAll();
```

Referencias bibliográficas:

GreenRootsBlog. (2023). *6 Common mistakes in using JavaScript Promises.* https://blog.greenroots.info/common-mistakes-in-using-javascript-promises

Medium. (2023). 5 *Common Mistakes when Using Promises.* https://blog.bitsrc.io/5-common-mistakes-in-using-promises-bfcc4d62657f

DEV Community. (2016-2023). *3 most common mistakes when using Promises in JavaScript.* https://dev.to/mpodlasin/3-most-common-mistakes-when-using-promises-in-javascript-oab

Sink In – Tech Learnings. (2023). *Common Javascript Promise mistakes every beginner should know and avoid*. https://www.gosink.in/common-javascript-promise-mistakes-beginners/

Medium. (2023). *Why you should be very careful about the usage of Promises in JavaScript*. https://medium.com/@ftiebe/why-you-should-be-very-careful-about-the-usage-of-promises-in-

javascript-c682fc09350b

Medium. (2023). *Common Mistakes Developers Make with Promises in Node.js*. https://medium.com/@aliraza.1121/common-mistakes-developers-make-with-promises-in-node-js-dd7706fc4e05