La mirada puesta en las promesas

> *"Son las funciones asincrónicas las que ofrecen la promesa de Promesas, por así decirlo."*
>
> David Herron — Software Engineer and author (Node.js Web Development)

Photo by Glenn Carstens-Peters on Unsplash

Antes de comenzar, veamos un *hack* que debes tener en cuenta para convertirte en un/a full stack developer y hacer que tu trabajo sea accesible tanto para futuros/as compañeros/as como usuarios/as. Ten en cuenta que cuando programas, debes hacer comentarios en el código:

- **Comenta antes de programar.** Antes de definir tu primera variable, esperar la resolución de una promesa o desarrollar tu primera función, empieza a comentar tu código. Escribir lo que debes hacer te ayudará a comprender mejor el problema.

- **Trata de que tu comentario sea útil.** La mayoría de las veces nuestro código será leído por otro/a programador/a y es importante sumar comentarios que aportan valor a la compresión del código. No es necesario explicar cosas obvias. Aquí, un ejemplo:

```
<!-- utilizó un span para mostrar el precio -->
U$D <span id="price"></span>
<script>
//injecto el precio en el elemento id = price
document.getElementById('price').innerHTML = 250;
</script>
```

- **No dejes los comentarios para el final.** En general, a los desarrolladores no nos gusta comentar, por lo que lo dejamos para el final. Esperar hasta el final significa que estarás obligado/a a comentar mucho código de una vez. Se te hará más pesado y no saldrá como esperas, porque las ideas pierden frescura y precisión a medida que pasa el tiempo. Te dejamos una regla fundamental:

  `Comenta > Escribe código > Comenta > Escribe código...`

🤓 Recuerda estos pasos: pienso, comento luego desarrollo.

Más promesas

En el encuentro anterior te mostramos cómo trabajar con promesas: podemos tener una serie de instrucciones que finalizarán con un resultado exitoso o no y, a cada uno de los posibles resultados, podemos asignar un callback para ejecutar la o las acciones que creamos conveniente.

¿Eso es todo? No, aún hay más.

Nos podemos encontrar con escenarios donde debemos ejecutar más de una promesa y, de acuerdo a lo que debamos realizar, podemos necesitar:

1. tener **una acción para cuando finalicen todas las promesas**
2. que la **resolución de una promesa se ejecute al finalizar la anterior**,
3. o simplemente **capturar la primera promesa que se resuelva sin importar el resto de las promesas.**

Grafiquemos estos 3 escenarios posibles.

[1] **Todas las promesas**

"Tener una acción para cuando finalicen todas las promesas."

Tenemos un ciclo de N cantidad de repeticiones y por cada vuelta lanzamos una promesa. Puede ser para buscar información, acceder a una imagen o ejecutar una consulta a una base de datos. Solamente nos sirve que todas las promesas se cumplan de manera satisfactoria, para esto contamos con `promise.all`.

```
const p1 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(1);
  }, 200);
});
const p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(2);
  }, 100);
});
const p3 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve(3);
  }, 300);
});

Promise.all([p1, p2, p3]).then((resp) => {
  console.log(resp); // Logs [1,2,3]
}, (err) => {
  console.log(err); // Not executed
});
```

[2] Promesas en cadena

"Que la resolución de una promesa se ejecute al finalizar la anterior."

Mediante las promesas en cadena poder encadenar un resultado tras otro, es decir, ejecutar una promesa y pasar a otra promesa.

De esta manera podemos ejecutar una serie de instrucciones y, cuando concluyan, ejecutar otra serie:

```
new Promise((resolver, rechazar) => {
    console.log('Inicial');

    resolver();
})
.then(() => {
    throw new Error('Algo falló');

    console.log('Haz esto');
})
.catch(() => {
    console.log('Haz aquello');
})
.then(() => {
     console.log('Haz esto sin que importe lo que sucedió
antes');
});
```

[3] Carrera de promesas

"Capturar la primera promesa que se resuelva sin importar el resto de las promesas."

Race ejecuta un array de promesas tal como lo hace .all pero el then se ejecutará en el momento en el que la primera promesa se resuelva, sin esperar el resultado del resto de las promesas:

```
var p1 = new Promise( (resolve, reject) => {
    setTimeout(resolve, 500, "uno");
});
var p2 = new Promise( (resolve, reject) => {
    setTimeout(resolve, 100, "dos");
});

Promise.race([p1, p2]).then( value => {
  console.log(value); // "dos"
  // Ambas se resuelven, pero la p2 antes.
});
```

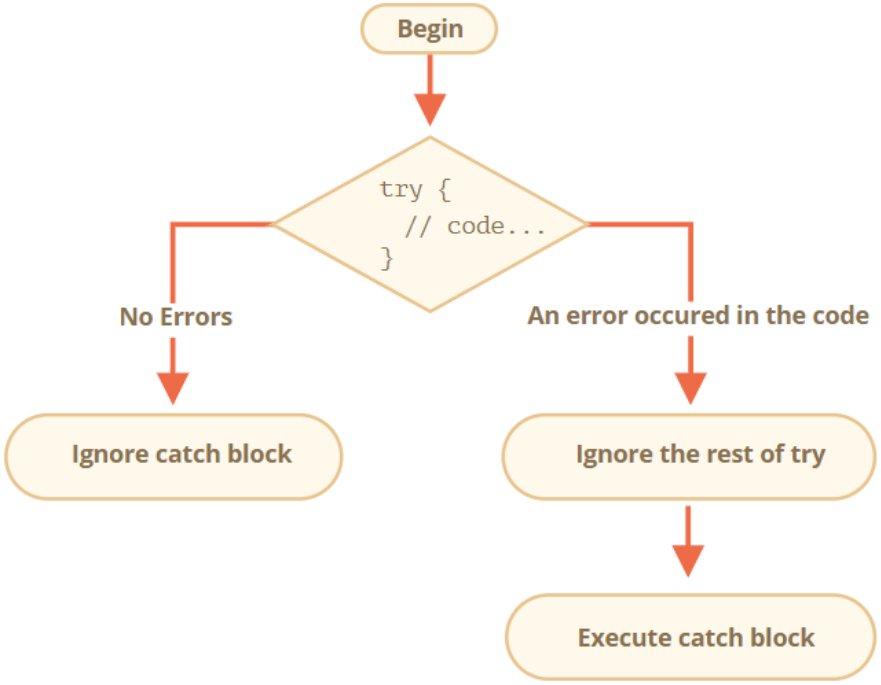
Manejo de fallos

Hay muchos factores propios y ajenos a nuestro código que pueden hacer que un programa falle. Al trabajar con código asincrónico, cualquiera de los procesos que se ejecutan paralelamente pueden fallar. Es por esto que es importante afectar el programa que se ingresa de estos errores ocurridos e indicar cómo continuar en estos casos.

Según la forma que elijamos para escribir funciones asincrónicas, tendremos distintas formas de hacer esto:

- Al utilizar `callbacks`, una convención es utilizar dos parámetros donde el primero es el indicador de la acción falló, mientras que el segundo contiene el resultado al que se llega por el proceso si es exitoso.
    - El problema es que esto nos obliga siempre a chequear si se produjo o no un error.

Las promesas pueden ser resueltas o rechazadas. Así como las promesas resueltas devuelven un resultado, también lo hacen cuando son rechazadas. Esto se conoce como la **"razón" del Fallo**. Cuando una excepción produce el rechazo de la promesa, el valor de esa excepción es utilizado como la razón.




```
try {
    //hacer algo
}catch{
    //ocurrió algún error
    //hacer algo
}
```

¡Prepárate para el próximo encuentro!

Profundiza

📄 Todo sobre promesas

📹 Asincronía en JavaScript - Promise.all

Challenge

📝 Crea 3 promesas diferentes que tomen al azar un número del 1 al 100. Cuando las 3 promesas tienen el número al azar mayor a 50 muestra un mensaje de éxito, de lo contrario muestra un mensaje que indique que al menos 1 promesa no superó el número 50.