

No todo es callbacks, también hay ¡Promesas!

"No se trata de hacer asíncrono lo síncrono. Se trata de pensar asíncronamente."

Ulises Gascón Gonzalez – Software Engineer & Consultant | Co-founder
@os_weekends | Instructor @fictiziaescuela | xGoogler xIBMer



Photo by [Brooke Cagle](#) on [Unsplash](#)

Si hay algo que nos destaca a los/as developers es que encontramos nuevas formas de hacer las cosas. No nos conformamos, somos dinámicos/as y constantemente buscamos maneras para optimizar nuestro trabajo. Lo que hemos aprendido con JavaScript hasta el momento es un claro ejemplo de los cambios que fueron sucediendo en nuestra manera de evolucionar una aplicación.

En la bitácora anterior abordamos un tema fundamental: los procesos asincrónicos para optimizar los tiempos de una aplicación. ¡Aprendimos cómo utilizar [callback](#)! A medida que avancemos, te encontrarás con que si tus respuestas tienen muchas dependencias, escribirlas se puede volver engorroso. Observa lo que pasa en este ejemplo:



```

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');
});

```

A los códigos que se ven así, se los suele denominar “[callback hell](#)”. La buena noticia es que TODO tiene solución (y en eso somos expertos/as).

En un momento, JavaScript sólo contaba con callback para los procesos asincrónicos. En esta bitácora vas a encontrar otra forma que fue recientemente incorporada para poder trabajar con este tipo de códigos complejos de una forma más rápida y sencilla.

¡Hola, promesas! (adiós ‘callback hell’)

Manejar flujos de datos asíncronos es complejo, pero hay una manera de hacerlo mejor: para simplificar el anidado de callbacks y trabajar con acciones asincrónicas más ordenadas, con menos líneas, en JavaScript se introdujeron las [promesas](#).

Una [promesa](#) será una acción que podrá resolverse en un momento dado en el futuro. Nuestra promesa en algún momento generará un resultado, ya sea positivo o negativo, y nosotros seremos capaces de capturar esa respuesta para aplicar la acción que creamos conveniente.

Al mismo código anterior lo podemos pasar de callback a promesa de la siguiente manera:



```

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));

```

Es diferente, ¿no? Es mucho más armónico a la vista y también lo es para trabajarlo. Para que nuestro código pueda verse así es que aprenderemos a usar las promesas.

Este concepto está muy asociado a la asincronía. Esto se debe a que JavaScript ejecuta instrucciones y no espera su resultado, por lo que cuando necesitamos consumir algo externo no sabemos cuanto puede llegar a demorar. De esta manera no importa cuánto tiempo tarde la instrucción en finalizar, va a haber algo que esté esperando la finalización de la promesa para ejecutarse.

Las promesas pueden pasar por tres estados:

1. **pendiente** mientras se está ejecutando y todavía no hay un resultado
2. **resuelta** cuando la promesa se ejecutó exitosamente
3. **rechazada** si la promesa falló y finaliza sin el resultado deseado

Antes de entrar en las líneas de código, llevemos el trabajo de promesas a un ejemplo de la vida cotidiana.

Esta noche quiero cenar Risotto Italiano, pero como no tengo los ingredientes necesarios debo ir al supermercado a conseguirlos.

```

cenar_risotto = creo una promesa {
  //ejecuté mis instrucciones
  //ir al supermercado
  //comprar ingredientes
  //pagar en la caja
  //volver a casa
  //cocinar

```



```

    if(tengo todos los ingredientes? && pude preparar el risotto){
        promesa exitosa
    }else{
        promesa rechazada
    }
}

```

```

cenar_risotto.exitosa( callback () { estoy cenando risotto } )
cenar_risotto.rechazada( callback () { debo preparar otra cena } )

```

En el momento que la promesa se encuentra ejecutando instrucciones (ir al supermercado, buscar los ingredientes, etc.), la promesa se encuentra en pendiente. Luego hacemos la pregunta para ver si logré el objetivo o no, y allí es donde finalizo mi promesa de manera exitosa o la rechazo.

¿Cómo trabajar con promesas?

Mediante la clase `Promise` podemos construir un objeto con los métodos necesarios para ejecutar la resolución de las promesas. El constructor de la clase recibe un parámetro de tipo función, la cual será la encargada de contener las instrucciones para la lógica de tu promesa y definirá si es aprobada o rechazada:

```

let mi_promesa = new Promise((resolve, reject) => {
    const number = Math.floor(Math.random() * 5);
    if(number > 0){
        resolve((number % 2)? "es impar" : "es par");
    }else{
        reject("es cero")
    }
});

```



Dentro de la función que envió al constructor, se ejecuta tu serie de sentencias. En este caso: tomamos un número al azar entre 0 y 5:

- si el número tomado al azar es mayor a 0 finalizamos la promesa de manera exitosa, calculando si el número es par o impar.
- en cambio, si el número seleccionado al azar es cero, finalizamos la promesa rechazándola, enviando el mensaje de que el número es cero.

Como no sabemos cuánto tiempo puede demorar toda esta serie de instrucciones, cuando capturamos la resolución de las promesas tenemos los métodos `then` y `catch`.

- **Obtener el resultado de una promesa de forma exitosa.** Con el método `then` se puede registrar una función callback que será ejecutada cuando la promesa se resuelva y devuelva un resultado. Al utilizar `then` podemos indicar dos funciones, la primera se ejecutará en el caso en el que la promesa sea resuelta exitosamente, y la segunda cuando falle.
- **Capturar promesas rechazadas.** Con el método `.catch` podemos capturar las promesas rechazadas y definir nuestro bloque de código para trabajar con el error.

```
mi_promesa
  .then(number => console.log(number))
  .catch(error => console.error(error));
```

Observa aquí el código completo funcionando:

[Ejemplo: Promesas](#)

Las promesas llegaron para facilitarnos la vida, ¡y sobre todo la forma en que escribimos código! ¿No lo crees? Ahora ya sabes lo que puedes hacer con ellas. Adelante, animate y ¡comienza a trabajar!



Live Coding

¿Te quedaste con ganas de más? ¿Prefieres escuchar más que leer para aprender? ¿Quieres ver al director de la carrera - Daniel Segovia - codear lo que te contamos en la bitácora? Aquí te dejamos un video:

[Javascript Promesas](#)


¡Prepárate para el próximo encuentro!

Profundiza

 [Teoría de las promesas](#)

 [Manteniendo las Promesas con JavaScript](#)

Challenge

 Las promesas llegaron para resolver muchas tareas, como por ejemplo el *callback hell* mencionado en esta bitácora.

En este challenge te proponemos que encuentres al menos UN problema común que se resuelva a través de las promesas de JavaScript.

¡En el próximo encuentro compartiremos los resultados y empezaremos a ver cómo resolverlos!

Potencia tu Talento - CV

CV (Hoja de vida)

Ya sé lo que estás pensando, ¿todavía se sigue utilizando el CV? Ya trabajamos sobre el perfil de LinkedIn, ¿no alcanza con bajar el pdf? Aunque su función se ha transformado el CV sigue vigente, y suele ser muy solicitado en la mayoría de los procesos de aplicación y selección de personal.



Si estás buscando trabajo es importante que tengas un CV listo para compartir, pero ten en cuenta que no debe tener más de una hoja y tiene que dar claridad sobre lo que TU puedes aportar a la organización, es decir, dale tu toque personal.

Te compartimos algunos artículos que creamos especialmente para que nuestra comunidad tenga los mejores CV que se hayan visto.



Feedback sobre tu CV:

¿Te animas a pedirle feedback sobre tu CV a alguna persona de tu grupo? Les proponemos que intercambien sus CVs y se den feedback tomando en cuenta los tips que mencionamos.

Pueden usar algunas de estas preguntas para guiar el feedback: ¿Están claras cuáles son las fortalezas de cada uno/a? ¿Y sus objetivos profesionales? ¿Quisiéramos saber algo más de esta persona? ¿Si queremos ponernos en contacto, es fácil hacerlo? ¿Tiene links a su perfil de LinkedIn o portfolio para más información?



[Trabajo en tecnología: crea el CV ideal](#)



[Tres errores comunes en el diseño del CV](#)



[La foto perfecta para tu CV](#)



[IG Live con Reclufit sobre diseño del CV](#)

