Programación asíncrona en JavaScript

En JavaScript, la programación asíncrona es una parte fundamental del lenguaje. Es el mecanismo que nos permite realizar operaciones en segundo plano, sin bloquear la ejecución del hilo principal. Esto es especialmente importante en el navegador, donde el hilo principal es responsable de actualizar la interfaz de usuario y responder a las acciones del usuario.

En general, la programación asíncrona es un tema complejo que requiere mucha práctica para dominar, pero en mi opinión, requiere un cambio en tu forma de pensar. Necesitarás empezar a pensar en cómo desglosar tu código en pequeños fragmentos que puedan ejecutarse en segundo plano, y cómo combinarlos para lograr el resultado deseado. Te encontrarás con la programación asíncrona regularmente mientras codificas en JavaScript. La mayoría de las operaciones que involucran interacciones con recursos externos, como enviar y recibir datos de un servidor o una base de datos y leer el contenido de un archivo, requerirán su uso.

Comencemos explorando cómo la programación asíncrona difiere de la programación convencional y cómo necesitamos adoptar una mentalidad diferente.

La mentalidad asíncrona

El primer paso para dominar la programación asíncrona es cambiar tu mentalidad. Necesitas comenzar a pensar en tu código de una manera no lineal; pensarás más en "qué debería suceder después" en lugar de "qué debería suceder primero".

Después de haber estudiando la actividad 4 vimos que una función es solo un fragmento de código que se puede ejecutar en cualquier momento. En esta parte, conectaremos ese fragmento de código con eventos previos y futuros.

Hay muchas maneras de realizar operaciones asíncronas en JavaScript. En esta nota, nos centraremos en las más comunes, que son las siguientes:

- Callbacks: Un callback es una función que se pasa como argumento a otra función, y se ejecuta cuando ocurre un cierto evento. Esta es la forma más básica de realizar operaciones asíncronas en JavaScript, y es la base de todos los demás mecanismos.
- Promesas: ES6 introdujo el concepto de promesas, que puedes usar para manejar operaciones asíncronas de una manera avanzada porque utilizan una máquina de estados con varios estados (pendiente, cumplido y rechazado) para realizar un seguimiento de las operaciones. Las promesas tienen muchas ventajas sobre los callbacks en términos de legibilidad, reutilización y simplicidad general. Esta es la forma más común de realizar operaciones asíncronas en JavaScript moderno hoy en día.
- Async/await: Async/await actúa como un envoltorio (wrapper) sobre las promesas para hacer que el código sea más legible (azúcar sintáctico). Actualmente, es la forma más popular de manejar operaciones asíncronas.

Los callbacks

Los callbacks explotan la capacidad de JavaScript para pasar funciones. Hay dos partes esenciales en esta técnica:

- Una función que se pasa como argumento a otra función

- La función pasada se ejecuta cuando ocurre un cierto evento

Vamos a crear un ejemplo básico para ilustrar este concepto. En los siguientes fragmentos de código, mostraremos cómo se define el callback como argumento y cómo se pasa una función como argumento cuando ocurre la ejecución:

1. En este ejemplo, definiremos una función (doSomething) que espera una función como argumento:

```
const doSomething = (cb) => {
  console.log('Doing something...');
  cb();
};
```

2. En este punto, tenemos una función llamada doSomething que recibe una función como argumento y la ejecuta como el último paso, lo que ilustra la idea de que los callbacks son solo un patrón donde esperamos que la siguiente función que se va a ejecutar sea en realidad llamada como el paso final (llámame cuando termines - callback). Veamos cómo podemos usar esta función:

```
const nextStep = () => {
  console.log('Callback called');
};
doSomething(nextStep);
```

3. Una vez que se ejecuta la función, el resultado esperado será el siguiente:

Doing something...

Callback called

Ahora, tenemos una función llamada nextStep que se pasa como argumento a doSomething. Cuando doSomething se ejecuta, imprimirá Doing something..., y luego ejecutará la función que se pasó como argumento, que imprimirá Callback called como último paso.

Es importante tener en cuenta que la función que se pasa como argumento no se ejecuta inmediatamente, ya que solo queremos ejecutarla cuando la operación haya finalizado. Por otro lado, la ejecución inmediata requerirá el uso de paréntesis (doSomething(nextStep())) y producirá un resultado diferente y un error:

```
doSomething(nextStep())
// Callback called
// Doing something...
// Error: cb is not a function
```

También podemos pasar una función anónima como argumento. Esta es la forma más común de usar callbacks, ya que no necesitamos definir las funciones previamente. En la mayoría de los casos, no reutilizamos esa función más adelante:

```
doSomething(() => {
  console.log('Callback called');
});
```

También es posible pasar una función que reciba argumentos:

```
const calculateNameLength = (name, cb) => {
  const length = name.length;
  cb(length);
};
calculateNameLength('Kapu', (length) => {
  console.log(The name length is ${length}); // The name length is 4
});
```

Como puedes ver, la técnica de callback es muy simple, pero aún no hemos visto ninguna operación asíncrona. Al final del día, asumimos que un callback es literalmente un enfoque de “llámame cuando hayas terminado”.

Temporizadores e intervalos

Hay dos funciones que se usan comúnmente para retrasar la ejecución de una función, setTimeout y setInterval. Ambas funciones reciben un callback como argumento y lo ejecutan después de una cierta cantidad de tiempo. Ahora, definamos y usemos estas funciones en ejemplos.

La función setTimeout se emplea para diferir la ejecución de una función por una cantidad de tiempo especificada.

Veamos cómo funciona setTimeout con un ejemplo simple:

```
console.log('Before setTimeout');
const secondInMilliseconds = 1000;
setTimeout(() => {
  console.log('A second has passed');
}, secondInMilliseconds);
console.log('after setTimeout');
```

Si ejecutamos este código, veremos la siguiente salida:

Before setTimeout

after setTimeout

A second has passed

Como puedes ver, el callback se ejecuta después del resto del código, aunque se definió antes. Esto se debe a que el callback se ejecuta de manera asíncrona, lo que significa que se ejecuta en segundo plano, y el resto del código se ejecuta en el hilo principal.

La función setTimeout recibe dos argumentos. El primero es el callback, y el segundo es la cantidad de tiempo que el callback debe retrasarse. La cantidad de tiempo se expresa en milisegundos, por lo que en este caso, estamos retrasando la ejecución del callback en 1,000 milisegundos, que es 1 segundo.

La función setInterval se usa para ejecutar una función repetidamente, con un retraso de tiempo fijo entre cada ejecución.

Veamos cómo funciona setInterval con un ejemplo simple:

```
const secondInMilliseconds = 1000;

let totalExecutions = 0

console.log('Before setInterval');

setInterval(() => {
    totalExecutions++;
    console.log(A second has passed, this is the ${totalExecutions} execution);
}, secondInMilliseconds);

console.log('After setInterval');
```

Si ejecutamos este código, veremos la siguiente salida:

Before setInterval

After setInterval

A second has passed, this is the 1 execution

...

A second has passed, this is the 50 execution

Como puedes ver, el callback se ejecuta cada segundo, y se ejecuta en segundo plano, por lo que el resto del código se ejecuta en el hilo principal.

La función setInterval recibe dos argumentos. El primero es el callback, y el segundo es la cantidad de tiempo que el callback debe retrasarse. La cantidad de tiempo se expresa en milisegundos, por lo que en este caso, estamos retrasando la ejecución del callback en 1,000 milisegundos, que es 1 segundo.

Errores con callbacks

Hemos visto cómo usar callbacks para gestionar operaciones asíncronas, pero no vimos cómo manejar errores. En esta parte, veremos cómo manejar errores en los callbacks.

La forma más común de manejar errores en los callbacks es usar el **patrón de error primero(error first pattern)**. Este patrón consiste en pasar un error como el primer argumento del callback, y el resultado como el segundo argumento. Veamos cómo funciona esto con un ejemplo simple:

```
const doSomething = (cb) => {
  const error = new Error('Something went wrong');
  cb(error, null);
};
doSomething((error, result) => {
  if (error) {
    console.log('There was an error');
    return;
  }
  console.log('Everything went well');
});
```

La salida de este código será la siguiente:

```
There was an error
```

En este ejemplo, tenemos una función llamada doSomething que recibe un callback como argumento. Este callback recibe dos argumentos. El primero es un error, y el segundo es el resultado. En este caso, estamos pasando un error como primer argumento, y null como segundo argumento porque ocurrió un error. Cuando se ejecuta el callback, verificamos si el primer argumento es un error, y si lo es, imprimimos There was an error. De lo contrario, imprimimos Everything went well.

Veamos cómo funciona esto cuando todo va bien:

```
const doSomething = (cb) => {
  const result = 'It worked!';
  cb(null, result);
};
doSomething((error, result) => {
  if (error) {
```

```
    console.log('There was an error');
    return;
  }
  console.log(result);
  console.log('Everything went well');
});
```

La salida de este código será la siguiente:

```
 It worked!
Everything went well
```

En este caso, estamos pasando null como primer argumento ya que no hay ningún error, y el resultado como segundo argumento. Cuando se ejecuta el callback, verificamos si el primer argumento es un error, y si lo es, imprimimos There was an error. De lo contrario, imprimimos el resultado, y Everything went well.

El infierno de los callbacks

Anteriormente, vimos cómo usar callbacks para gestionar operaciones asíncronas y cómo manejar errores con el patrón de error primero.

El problema con los callbacks es que no son muy fáciles de leer, y cuando tenemos muchos callbacks anidados, el código se vuelve muy difícil de leer. Esto se llama infierno de los callbacks, y es un problema muy común al usar callbacks.

En el siguiente ejemplo de código, observa cómo las funciones están estructuradas en una pirámide inclinada con callbacks anidados, lo que hace que el código sea difícil de comprender:

```
readFile("docs.md", (err, mdContent) => {
  convertMarkdownToHTML(mdContent, (err, htmlContent) => {
    addCssStyles(htmlContent, (err, docs) => {
      saveFile(docs, "docs.html",(err, result) => {
        ftp.sync((err, result) => {
          // ...
        })
      })
    })
  })
})
```

Como puedes ver, el código es muy difícil de leer, y es muy fácil cometer errores. Por eso necesitamos una mejor manera de gestionar las operaciones asíncronas. Hay algunas maneras de prevenir el infierno de los callbacks, como usar funciones con nombre en lugar de funciones anónimas, pero una de las formas más comunes de evitar el infierno de los callbacks es usar promesas (promises).

Promesas

Una promesa funciona como una máquina de estados, simbolizando el eventual éxito o fracaso de una operación asíncrona, junto con su valor resultante. Puede existir en cualquiera de los tres estados: pendiente, cumplido o rechazado.

Cuando se crea una promesa, está en estado pendiente. Cuando se cumple una promesa, está en estado cumplido. Cuando se rechaza una promesa, está en estado rechazado.

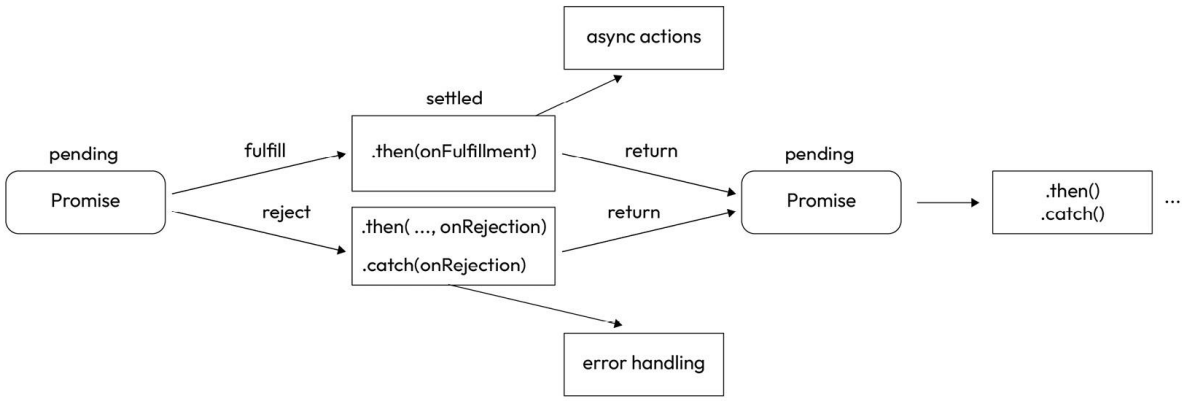
Importante: En el contexto proporcionado, una máquina de estado es un modelo conceptual que describe un sistema basado en una serie de estados predefinidos y transiciones entre estos estados. Cada estado representa una situación particular del sistema, y las transiciones son activadas por eventos específicos o condiciones.

En relación con las promesas en JavaScript, estas se comportan como una máquina de estado porque pueden existir en uno de tres estados posibles:

1. **Pendiente (Pending):** Cuando una promesa es creada, comienza en este estado. Significa que la operación asíncrona que la promesa representa aún no ha concluido, y por lo tanto, el resultado de dicha operación no está disponible.
2. **Cumplido (Fulfilled):** Este estado indica que la operación asíncrona ha finalizado exitosamente. En este punto, la promesa tiene un valor resultante que puede ser accedido a través de los métodos .then() o await.
3. **Rechazado (Rejected):** Este estado refleja que la operación asíncrona ha fallado. La promesa en este caso contendrá un error o motivo de rechazo, que puede ser manejado utilizando los métodos .catch() o dentro de un bloque try...catch si se usa await.

Una vez que una promesa transiciona de "pendiente" a "cumplido" o "rechazado", no puede regresar al estado pendiente ni cambiar al otro estado terminal (cumplido o rechazado). Es por esto que se dice que una promesa funciona como una máquina de estados, ya que modela la evolución de un proceso asíncrono a través de un conjunto limitado y predefinido de estados.

El siguiente diagrama muestra los diferentes estados de una promesa y las conexiones entre ellos:



https://developer.mozilla.org/en-US/docs/MDN/Writing_guidelines/Attrib_copyright_license

Después de que una promesa se cumple o se rechaza, se vuelve inmutable. Para gestionar el cumplimiento, se emplea el método then, mientras que el método catch se utiliza para abordar el rechazo de la promesas.

Crear promesas

Puedes crear una promesa utilizando el constructor Promise, que recibe un callback como argumento. Este callback recibe dos argumentos, resolve y reject. La función resolve se utiliza para resolver la promesa, y la función reject se utiliza para rechazar la promesa. Veamos cómo funciona esto con un ejemplo simple:

```
const setTimeoutPromise = (time) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve();
    }, time);
  });
};
console.log('Before setTimeoutPromise');
setTimeoutPromise(1000).then(() => console.log('one second later'))
console.log('After setTimeoutPromise');
```

La salida de este código será la siguiente:

Before setTimeoutPromise

After setTimeoutPromise

one second later

En este ejemplo, tenemos una función llamada setTimeoutPromise que recibe un tiempo como argumento. Esta función devuelve una promesa que se resolverá después del tiempo especificado. Cuando se resuelve la promesa, imprimimos one second later en la consola.

Infierno de callbacks con promesas

Las promesas son una excelente manera de lidiar con las limitaciones que introducen los callbacks cuando necesitamos realizar múltiples operaciones asíncronas que deben ejecutarse en orden consecutivo.

Las promesas manejarán los errores con mayor facilidad, por lo que la legibilidad del código debería ser más clara y fácil de mantener a largo plazo.

En la parte anterior, vimos que el infierno de callbacks es algo muy real en JavaScript. A estas alturas, deberías estar más familiarizado con la pirámide inclinada y los callbacks anidados. Aquí está el fragmento que usamos para explicar cómo el infierno de callbacks se puede lograr fácilmente en una parte anterior:

```
readFile("docs.md", (err, mdContent) => {
   convertMarkdownToHTML(mdContent, (err, htmlContent) => {
      addCssStyles(htmlContent, (err, docs) => {
         saveFile(docs, "docs.html",(err, result) => {
ftp.sync((err, result) => {
               // ...
            })
         })
      })
   })
})
```

Ahora veamos cómo podemos resolver este problema utilizando promesas:

```
readFile("docs.md")
  .then(convertMarkdownToHTML)
// atajo para .then(mdContent => convertMarkdownToHTML(mdContent))
  .then(addCssStyles)
  .then(docs => saveFile(docs, "docs.html"))
  .then(ftp.sync)
```

```
  .then(result => {
    // ... otras cosas
  })
  .catch(error => console.log(error));
```

Como puedes ver, el código es mucho más fácil de leer, y es mucho más fácil de hacer cambios. Esta es una de las principales ventajas de usar promesas. Ahora los errores se manejan en el último método catch, por lo que no necesitamos manejar los errores en cada método then, lo que hace que el código sea mucho más limpio.

Promesas en paralelo

Otra ventaja de usar promesas es que podemos ejecutar múltiples promesas en paralelo. Básicamente, proporcionamos un array de promesas y elegimos una estrategia para manejar los resultados (Promise.race() o Promise.all()). Esta es una excelente manera de reducir el tiempo de ejecución, ya que estamos utilizando las habilidades de Node.js para gestionar operaciones de I/O de manera asíncrona.

En los siguientes ejemplos, usaremos esta función para generar una promesa de tiempo de espera aleatorio como ejemplo de una operación asíncrona:

```
const randomTimeOutPromise = () => {
  return new Promise((resolve, reject) => {
    const time = Math.floor(Math.random() * 100);
    setTimeout(() => {
      console.log(Promise resolved after ${time}ms);
      resolve(time);
    }, time);
  });
};
```

Esta función devolverá una promesa que se resolverá después de un tiempo aleatorio entre 0 y 100 milisegundos. Ahora que tenemos una función asíncrona, podemos emplear varias estrategias para agrupar múltiples solicitudes juntas según nuestras necesidades específicas. En este caso, nuestro objetivo es iniciar varias solicitudes en paralelo y esperar su resolución.

Promise.all(): El método all produce una única promesa que se resuelve una vez que todas las promesas se resuelven o si alguna de las promesas es rechazada:

```
 Promise.all([
  randomTimeOutPromise(),
```

```
  randomTimeOutPromise(),
  randomTimeOutPromise(),
  randomTimeOutPromise(),
  randomTimeOutPromise(),
]).then((results) => {
  console.log("results:", results);
});
```

La salida de este código será algo así cuando todas las promesas se resuelvan con éxito:

Promise resolved after 0ms

Promise resolved after 26ms

Promise resolved after 31ms

Promise resolved after 37ms

Promise resolved after 62ms

results: [37, 31, 26, 62, 0]

Como puedes ver, el método then se llamará cuando todas las promesas se resuelvan, y recibirá un array con los resultados de cada promesa en el orden en que se agregan en el array de promesas, no por el orden en que se resuelven.

Importante: En el ejemplo anterior, todas las promesas se resolvieron con éxito, ya que se basan en una operación de temporizador. Pero cuando confiamos en promesas para acceder a recursos externos como archivos en un sistema o obtener datos de Internet, debemos tener en cuenta que estos recursos pueden no estar siempre disponibles. Por ejemplo, si Internet está caído, entonces una o varias promesas pueden fallar y esto hará que nuestra aplicación se bloquee. Obviamente, esta situación de bloqueo se puede evitar si manejamos los errores usando una declaración catch, pero incluso en ese caso, es muy importante recordar que cuando usamos este enfoque de paralelismo, debemos tener en cuenta que si una sola promesa genera un error, las promesas resueltas serán ignoradas de la misma manera que si estuviéramos usando una sola promesa.

Un enfoque alternativo a Promise.all() es agregar todas las solicitudes pero resolver la promesa tan pronto como se complete la primera. De esta manera, no es necesario esperar a que se cumplan todas las solicitudes.

Promise.race(): El método race devuelve una única promesa que se cumple o rechaza tan pronto como una de las promesas se cumple o rechaza. Esto puede llevar a resultados inesperados si no se gestiona con cuidado, ya que las promesas no se detendrán incluso si una de las promesas ya fue rechazada o cumplida:

```
Promise.race([
  randomTimeOutPromise(),
```

```
  randomTimeOutPromise(),
  randomTimeOutPromise(),
  randomTimeOutPromise(),
  randomTimeOutPromise(),
]).then((result) => {
  console.log("result:", result);
});
```

La salida de este código será algo así:

```
 Promise resolved after 30ms
results: 30
Promise resolved after 33ms
Promise resolved after 60ms
Promise resolved after 79ms
Promise resolved after 83ms
```

Como puedes ver, el método then se llamará cuando se resuelva la primera promesa, y recibirá el resultado de la primera promesa que se resuelva. Las otras promesas seguirán ejecutándose, pero el método then no se llamará nuevamente.

Manejo de errores

En los ejemplos anteriores, vimos cómo manejar errores usando el método catch, pero hay otra forma de manejar errores: usando la función reject. Veamos cómo funciona esto con este ejemplo:

```
const generatePromise = shouldFail => {
  return new Promise((resolve, reject) => {
    if (shouldFail) {
      return reject(new Error("Rejected!"));
    }
    resolve("Success!");
  });
};
generatePromise(true).catch(error => console.log("Error message:", error));
// Error message: Error: Rejected!
```

// ...

Es importante indicar que la función reject no detendrá la ejecución del código, por lo que necesitamos devolver la función después de llamar a la función reject. Esto significa que, cuando llamas a reject, la ejecución del código dentro de la función anónima proporcionada a la promesa no se detiene automáticamente. En otras palabras, si no haces nada más, el código que sigue a la llamada a reject continuará ejecutándose, lo cual podría llevar a resultados inesperados.

El enfoque final es cuando necesitamos realizar una acción una vez que una promesa se ha concluido, independientemente de si fue exitosa o rechazada. Es importante recordar que los rechazos de promesas no manejados pueden llevar a errores en tiempo de ejecución que harán que tu aplicación se bloquee.

Promise.finally(): A veces, no nos importa si la promesa se resuelve o se rechaza; solo queremos saber cuándo se ha resuelto o rechazado la promesa. Para este caso, podemos usar el método finally:

```
generatePromise(true)
  .then(result => console.log("Result:", result))
  .catch(error => console.log("Error message:", error))
  .finally(() => console.log("Promise settled"));
```

Encadenar promesas

También podemos encadenar promesas; podemos devolver una promesa en el método then, y esta promesa se resolverá antes de llamar al siguiente método then. El método catch se llamará si alguna de las promesas en la cadena es rechazada. Veamos un ejemplo:

```
generatePromise()
  .then(generatePromise)
  .then(result => {
    return generatePromise(true);
  })
  .then(() => console.log("This will not be called"))
  .catch(error => console.log("Error message:", error));
```

Cuando se llama a la tercera función generatePromise, devolverá una promesa que será rechazada, por lo que se llamará al método catch y luego no se ejecutará el último método then.

Hemos estado utilizando promesas durante un tiempo, y la sintaxis puede ser bastante verbosa, requiriendo palabras clave como then y catch constantemente. Una sintaxis más avanzada y estéticamente agradable implica el uso de **async** y **await**.

Usar async y await para manejar código asíncrono

ES2017 introdujo una nueva forma de manejar código asíncrono, las palabras clave async y await. Estas palabras clave son azúcar sintáctico para las promesas; no son una nueva forma de manejar código asíncrono, pero hacen que el código sea mucho más fácil de leer y escribir.

Esencialmente, la palabra clave async se emplea para definir una función asíncrona, mientras que la palabra clave await se usa para pausar y esperar la resolución de una promesa dentro de esa función. Incluso si usas la palabra async, no hace que tu código sea asíncrono, eso solo ocurrirá cuando realmente tengas código asíncrono en él (una promesa). Para hacerlo más simple, podemos decir que para usar await, necesitamos definir el bloque de código usando async.

async

Cuando una función se define con la palabra clave async, siempre devolverá una promesa que se puede manejar como cualquier promesa regular. Veamos un ejemplo:

```
const asyncFun = async (generateError) => {
  if (generateError) {
    throw new Error("Error generated");
  }
  return 1;
};
asyncFun().then((result) => console.log(result));
asyncFun(true).catch((error) => console.log(error));
```

Como esto es azúcar sintáctico para promesas, podemos construir una función similar usando promesas:

```
const asyncFun =  (generateError) => new Promise((resolve, reject) => {
  if (generateError) {
    reject(new Error("Error generated"));
  }
resolve(1);
});
asyncFun().then((result) => console.log(result));
asyncFun(true).catch((error) => console.log(error));
```

Ahora, familiaricémonos con await; podremos combinar sin problemas ambas palabras clave y eliminar la necesidad de usar then o catch.

await

Veamos cómo podemos usar la palabra clave await para esperar promesas:

```
// Definición de la función asíncrona usando async/await
const asyncFun = async (generateError) => {
  if (generateError) {
    throw new Error("Error generated"); // Lanzamos un error si generateError es true
  }
  return 1; // Si no hay error, retornamos 1
};

// Uso de la función asíncrona con await en otra función asíncrona
const testAsyncFunction = async () => {
  try {
    const result = await asyncFun(false); // Esperamos a que asyncFun se resuelva con generateError = false
    console.log(result); // Imprimimos el resultado si la promesa se resuelve exitosamente
  } catch (error) {
    console.error("Caught an error:", error); // Capturamos y mostramos cualquier error
  }

  try {
    const result = await asyncFun(true); // Esperamos a que asyncFun se resuelva con generateError = true
    console.log(result); // Este código no se ejecutará porque asyncFun lanzará un error
  } catch (error) {
    console.error("Caught an error:", error); // Capturamos y mostramos el error generado
  }
};

// Llamamos a la función de prueba para ver los resultados
```

```
testAsyncFunction();
```

El uso de await en este contexto elimina la necesidad de encadenar .then() y .catch(). En su lugar, se puede manejar la lógica de manera más lineal y legible, utilizando try...catch para manejar errores asíncronos.

Como puedes ver, el código es mucho más fácil de leer y escribir utilizando async y await. La palabra clave await solo se puede usar dentro de una función async. Necesitamos usar el bloque try/catch para manejar los errores.

Ahora, exploremos cómo podemos combinar las expresiones de función inmediatamente invocadas (Immediately Invoked Function Expressions IIFEs) con async para emplear este azúcar sintáctico incluso en versiones anteriores de Node.js.

IIFEs

En algunos casos, queremos usar la palabra clave await fuera de una función async, por ejemplo, cuando estamos utilizando la palabra clave await en el nivel superior de un módulo. En este caso, podemos usar una IIFE para envolver la palabra clave await dentro de una función async. Una IIFE es una función que se ejecuta inmediatamente después de ser creada. Es un patrón de diseño que se utiliza para evitar contaminar el ámbito global con variables y funciones. En el siguiente ejemplo, podemos observar la sintaxis fundamental:

```
 (function () {
  // ... algo de código aquí
})();
```

La idea es crear una función anónima y ejecutarla inmediatamente después de ser creada. Para lograr esto, necesitamos envolver la función entre paréntesis, y luego agregar otro par de paréntesis para ejecutar la función: (...)().

Podemos usar async y await dentro de una IIFE fácilmente:

```
(async () => {
   try {
      const asyncFun = async (generateError) => {
         if (generateError) {
            throw new Error("Error generated"); // Lanzamos un error si generateError es true
         }
         return 1; // Si no hay error, retornamos 1
```

```
    };

    // Usamos await dentro de la IIFE para esperar la resolución de la promesa
    const result1 = await asyncFun(false); // No generará un error
    console.log(result1); // Imprimimos el resultado: 1

    const result2 = await asyncFun(true); // Generará un error
    console.log(result2); // Este código no se ejecutará debido al error

  } catch (error) {
    console.error("Caught an error:", error); // Capturamos y mostramos el error
  }
})();
```

La IIFE se ejecuta inmediatamente después de ser definida, lo que permite que el código asíncrono se ejecute en cuanto se carga el script.

Ejercicios

1. Callbacks

- Ejercicio: Implementa una función fetchData que simule la obtención de datos desde un servidor. Esta función debería tomar un callback que se ejecutará con los datos obtenidos.

```
function fetchData(callback) {
  setTimeout(() => {
    const data = { id: 1, name: "John Doe" };
    callback(data);
  }, 2000);
}

// Uso de fetchData
fetchData((data) => {
  console.log("Data received:", data);
```

```
});
```

2. Temporizador y intervalos

Crea un temporizador que cuente de 1 a 5, imprimiendo cada número con un segundo de diferencia. Usa setInterval y clearInterval.

```
let count = 1;
const intervalId = setInterval(() => {
    console.log(count);
    if (count === 5) {
        clearInterval(intervalId);
    }
    count++;
}, 1000);
```

3. Errores en callbacks

Implementa una función processData que simule el procesamiento de datos. Esta función debe tomar un callback que maneje un error o los datos procesados. Simula un error cuando se pasa un valor específico.

```
function processData(data, callback) {
    setTimeout(() => {
        if (data === null) {
            return callback(new Error("No data provided"), null);
        }
        callback(null, data.toUpperCase());
    }, 1000);
}

// Uso de processData
processData("hello", (error, result) => {
    if (error) {
        console.error("Error:", error.message);
```

```
    } else {
        console.log("Processed data:", result);
    }
});
```

4. Infierno del callbacks

Simula una serie de operaciones asíncronas que dependen unas de otras, usando callbacks anidados (callback hell). Implementa tres funciones asíncronas (step1, step2, step3) y encadénalas.

```
function step1(callback) {
    setTimeout(() => {
        console.log("Step 1 complete");
        callback();
    }, 1000);
}

function step2(callback) {
    setTimeout(() => {
        console.log("Step 2 complete");
        callback();
    }, 1000);
}

function step3(callback) {
    setTimeout(() => {
        console.log("Step 3 complete");
        callback();
    }, 1000);
}

// Infierno callbacks
step1(() => {
```

```
    step2(() => {
        step3(() => {
            console.log("All steps completed");
        });
    });
});
```

5. Promesas

Transforma la función processData de callbacks a promesas. Luego úsala para procesar datos con .then() y .catch().

```
function processData(data) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (data === null) {
                reject(new Error("No data provided"));
            } else {
                resolve(data.toUpperCase());
            }
        }, 1000);
    });
}

// Uso de promesas
processData("hello")
    .then(result => {
        console.log("Processed data:", result);
    })
    .catch(error => {
        console.error("Error:", error.message);
    });
```

6. Promesas paralelas

Utiliza Promise.all para ejecutar en paralelo tres promesas que simulen la obtención de datos desde diferentes fuentes. Maneja los resultados o los errores.

```
javascript

function fetchData(source) {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve(${source} data);
        }, Math.random() * 2000);
    });
}


// Promesas en paralelo
Promise.all([
    fetchData("Source 1"),
    fetchData("Source 2"),
    fetchData("Source 3")
])
.then(results => {
    console.log("All data received:", results);
})
.catch(error => {
    console.error("Error in fetching data:", error);
});
```

7. Encadenando promesas

Crea una cadena de promesas donde cada promesa dependa del resultado de la anterior. Implementa tres funciones (getData, processData, saveData) que se encadenen usando .then().

```javascript
function getData() {
  return new Promise((resolve) => {
    setTimeout(() => resolve("raw data"), 1000);
  });
}

function processData(data) {
  return new Promise((resolve) => {
    setTimeout(() => resolve(${data.toUpperCase()} processed), 1000);
  });
}

function saveData(data) {
  return new Promise((resolve) => {
    setTimeout(() => resolve(${data} saved), 1000);
  });
}

// Cadena de promesas
getData()
  .then(data => processData(data))
  .then(processedData => saveData(processedData))
  .then(finalResult => {
    console.log(finalResult);
  })
  .catch(error => {
    console.error("Error:", error.message);
  });
```

8. Usando async y await

Reescribe la cadena de promesas del ejercicio anterior usando async y await. Maneja los errores con try-catch.

```
async function handleDataFlow() {
    try {
        const rawData = await getData();
        const processedData = await processData(rawData);
        const finalResult = await saveData(processedData);
        console.log(finalResult);
    } catch (error) {
        console.error("Error:", error.message);
    }
}
// Llamada a la función asíncrona
handleDataFlow();
```

9. Combinando callbacks y promises

Crea una función que acepte tanto un callback como una promesa. Si el callback es proporcionado, úsalo, de lo contrario, devuelve una promesa.

```
function fetchDataWithFallback(callback) {
    const promise = new Promise((resolve) => {
        setTimeout(() => resolve("Data fetched"), 1000);
    });

    if (callback) {
        promise.then(result => callback(null, result)).catch(err => callback(err));
    } else {
        return promise;
```

```
    }
}
// Uso con callback
fetchDataWithFallback((err, data) => {
    if (err) {
        console.error("Error:", err.message);
    } else {
        console.log("Callback:", data);
    }
});


// Uso con promesa
fetchDataWithFallback()
    .then(data => console.log("Promise:", data))
    .catch(err => console.error("Error:", err.message));
```

10. Convierte una función que usa callbacks (por ejemplo, readFile de Node.js) a una que usa promesas. Luego, escribe un código que lea un archivo usando la versión con promesas y maneje errores si el archivo no existe.

11. Implementa una función que intente resolver una promesa varias veces en caso de fallo, antes de rechazarla definitivamente. El número de intentos debe ser configurable.

12. Crea una función asíncrona que simule una operación de larga duración (5 segundos) y retorne un valor. Luego, usa async/await para esperar su resultado e imprime el tiempo total transcurrido.

13. Crea una cadena de promesas donde una de las funciones lance un error intencionalmente. Asegúrate de manejar el error correctamente en la cadena y continuar con la ejecución después de manejar el error.

14. Crea una función que realice múltiples operaciones asíncronas usando una combinación de Promise.all y async/await. Una de las operaciones debe fallar, y debes manejar ese error sin interrumpir las demás operaciones.

15. Implementa una función que resuelva una promesa si se completa dentro de un tiempo límite especificado. Si el tiempo límite se excede, la promesa debe ser rechazada con un error de "timeout".

16. Implementa una función que tome un array de funciones asíncronas y las ejecute en secuencia, pasando el resultado de una como argumento a la siguiente. Usa reduce para implementar la solución.

17. Escribe una función asíncrona que realice tres operaciones asíncronas. Asegúrate de manejar los errores de cada operación individualmente usando bloques try-catch específicos para cada una.

18. Crea un array de promesas que se resuelvan en diferentes momentos. Usa for...of con await para esperar que cada promesa se resuelva antes de continuar con la siguiente.

19. Crea tres promesas que se resuelvan o rechacen en diferentes momentos. Usa Promise.race para retornar la primera que se complete y maneja tanto el éxito como el fallo.

20. Escribe una función que devuelva una promesa, que a su vez retorne otra promesa dentro de su bloque .then(). Luego, maneja correctamente el resultado de la promesa anidada.

21. Implementa una función de polling que verifique el estado de una operación asíncrona cada segundo hasta que se complete. Usa una promesa que se resuelva cuando la operación finaliza exitosamente.

22. Crea una lista de promesas donde al menos una de ellas se rechace. Usa Promise.allSettled para manejar los resultados de todas las promesas y reportar cuáles se resolvieron y cuáles se rechazaron.

23. Simula la carga de múltiples recursos (como imágenes) secuencialmente, asegurando que cada recurso se cargue completamente antes de intentar cargar el siguiente.

24. Crea una función que simule una llamada a una API con diferentes tiempos de respuesta. Usa async/await para esperar la respuesta de la API y manejar los diferentes tiempos de espera.

25. Simula una condición de carrera donde dos operaciones asíncronas intentan modificar el mismo recurso simultáneamente. Observa el comportamiento y propón una solución para evitar la condición de carrera.