Asincronismo

En javascript el proceso de resolver eventos es síncrono. Lo que significa que para que una tarea se resuelva, se espera a que se resuelva la tarea anterior, y así sucesivamente hasta no tener tareas. El asincronismo, por el contrario, sería el flujo donde no se espera que una tarea se resuelva para poder continuar con la siguiente. Esto en javascript se realiza con los callbacks, promesas o funciones `async/await`.

Javascript es un lenguaje de programación de un solo hilo bloqueante, por lo cual no puede realizar más de una tarea a la vez; por esto es importante el asincronismo en javascript, ya que nos permite realizar múltiples tareas al mismo tiempo sin bloquear el único hilo de ejecución y así, no bloquear el programa hasta resolver tareas que no sabemos si tienen solución, o que son procesos pesados de ejecutar.

Para poder entender cómo funciona el asincronismo en javascript, debemos entender el **javascript runtime,** que es todo el proceso de cómo funciona javascript y sus conceptos:

Conceptos básicos

Memory heap

El memory heap es el espacio en memoría donde se almacenan todas las variables y funciones que tiene nuestro programa.

Call stack - Task queue

Es la lista de tareas donde se van apilando los eventos deben ir resolviendo en nuestro programa, y funciona bajo el concepto de ***LIFO*** (Last-in, First-out), donde la última tarea en entrar al call stack, deberá ser la primera en salir o resolverse.

Callback Queue

Es la lista de tareas asíncronas que deberán realizarse una vez el call stack de tareas sincronas está vacío.

Event loop

El event loop o bucle de eventos, es un patrón de diseño que le permite a javascript esperar y llamar eventos. Este se encarga de asegurarse que el **call stack** está vacío, para así empezar a gestionar las tareas del **callback queue**.

¿Cómo funciona el javascript runtime?

Tenemos una lista de usuarios en JSON, que contiene la información del nombre y edad de un usuario. Necesitamos transformar esta lista de usuarios en elementos de HTML donde se muestre el nombre y edad de cada usuario ordenada de manera ascendente por edad.

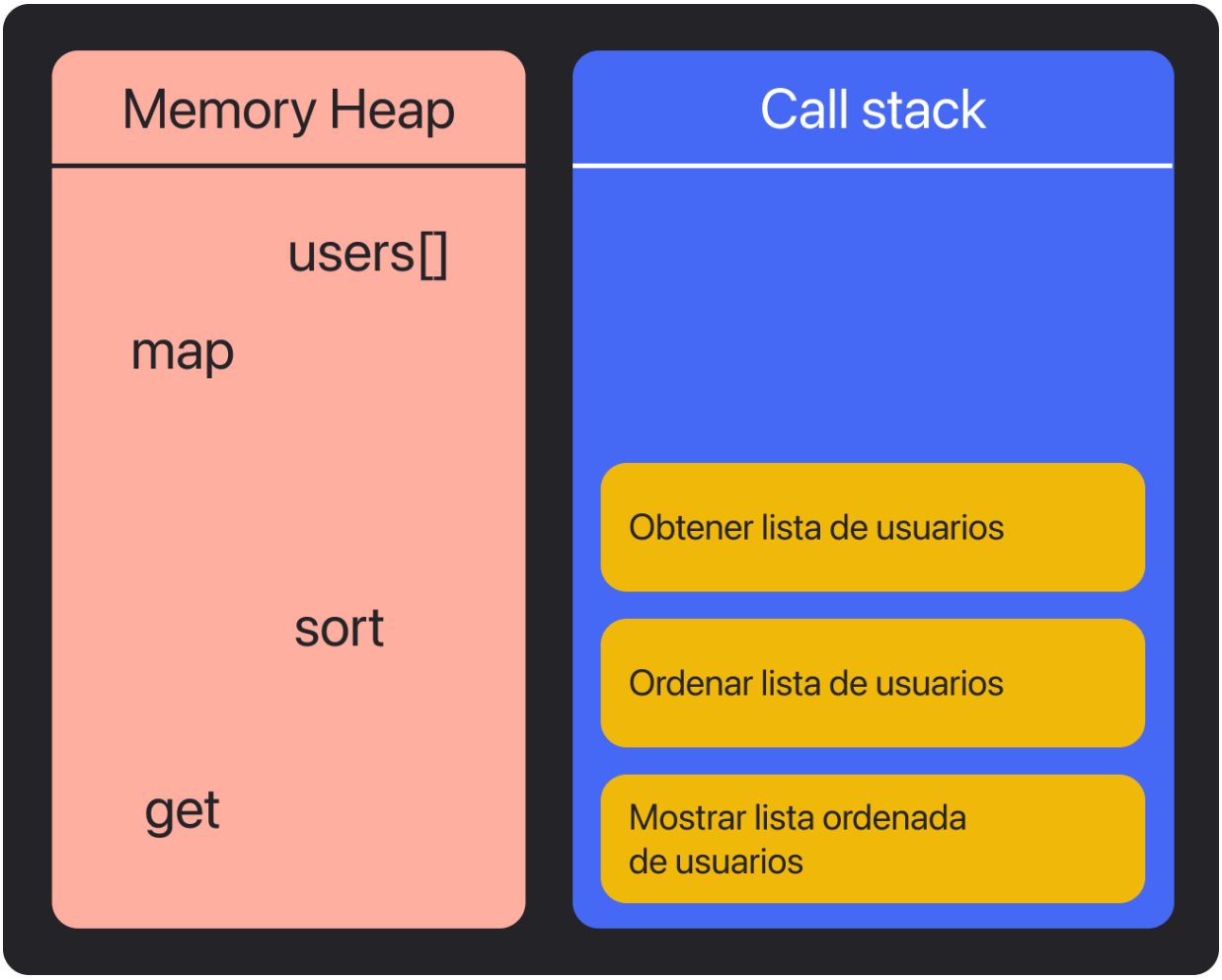
Para realizar esta tarea, debemos realizar las siguientes tareas:

1. Obtener la lista de usuarios.
2. Ordenar la lista de usuarios de manera ascendente por edad.
3. Mostrar en consola la lista de usuarios ordenada.

Para realizar estas tareas, tenemos en el **memory heap** almacenada toda la información relacionada a las funciones de javascript necesarias para obtener, recorrer y mostrar una lista de elementos.

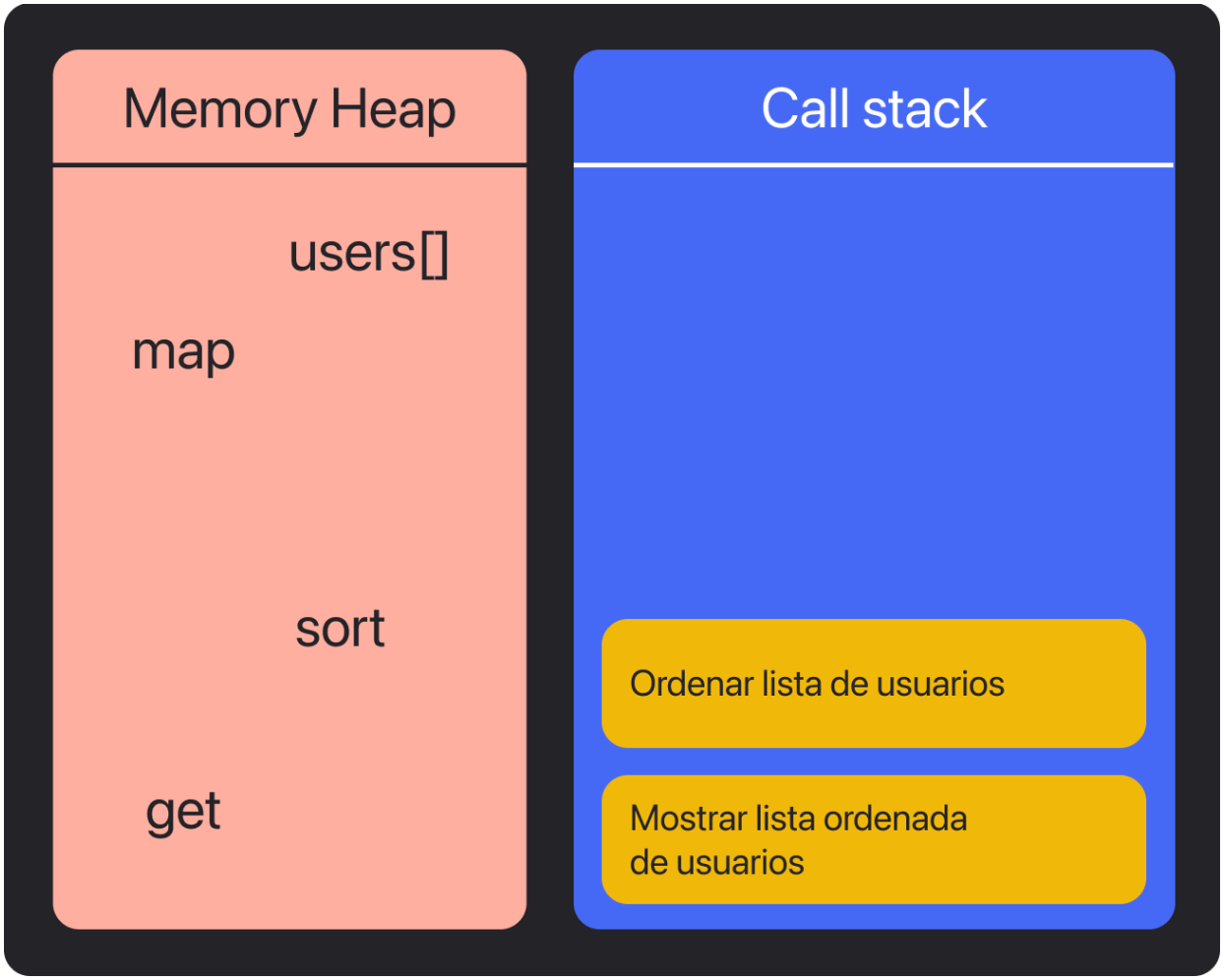
Cómo nuestra tarea principal es mostrar una lista de usuarios ordenados por edad de manera ascendente, esta será la tarea que se apilara primero en nuestro **call stack**.
Para poder realizar esta tarea, necesitamos realizar otras tareas primero como ordenar la lista de usuarios de manera ascendente, pero para poder realizar esta tarea, necesitamos obtener la lista de tarea.

Así se vería nuestro javascript runtime hasta el momento:

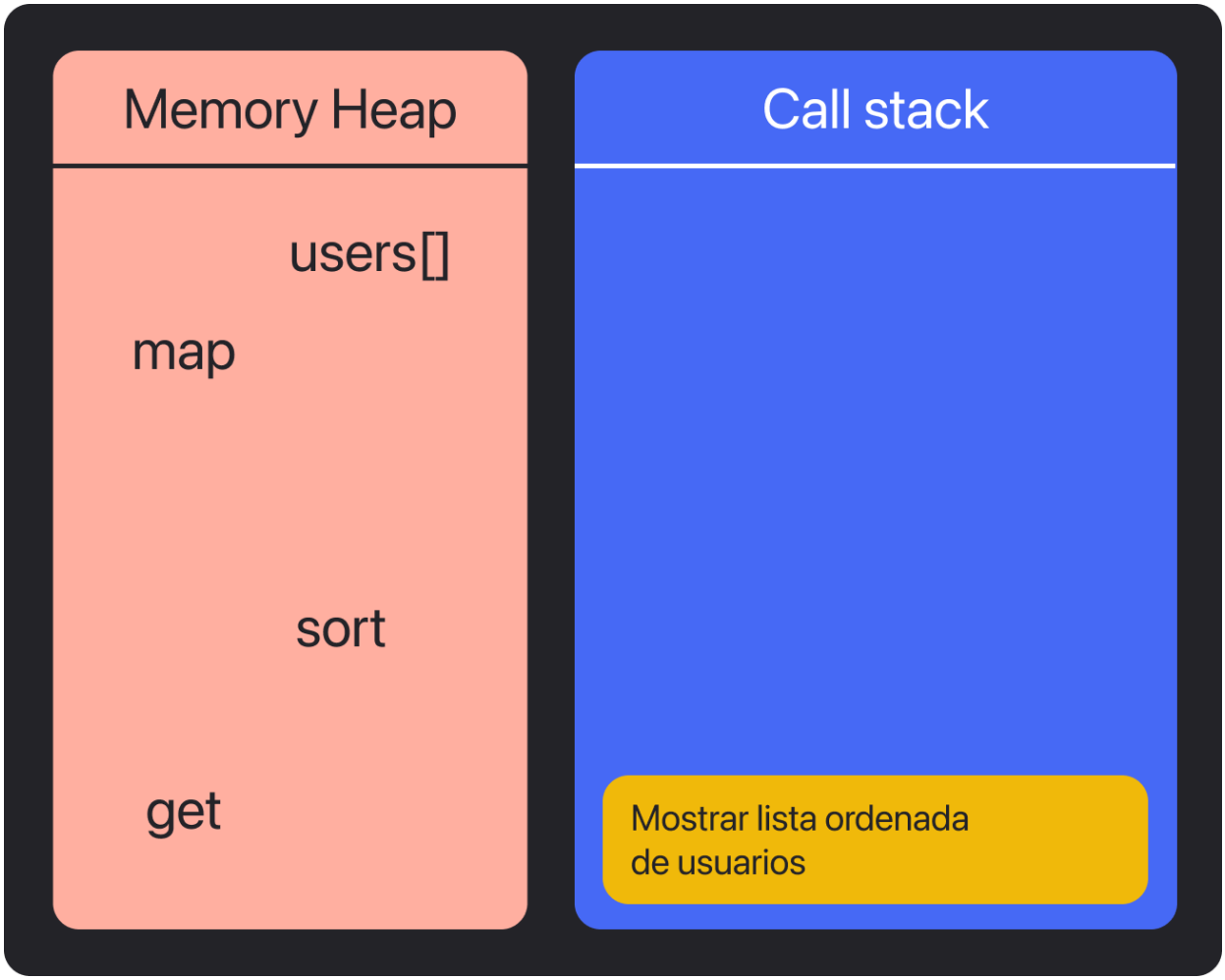


Siguiendo el concepto de **LIFO**, la primera tarea en ejecutarse sería la obtención de la lista de usuarios. Después la siguiente tarea a realizar sería la de ordenar la lista de usuarios por edad de manera ascendente, y para finalizar mostrar la lista de elementos en la consola.
Siendo gráficamente representado de la siguiente manera:

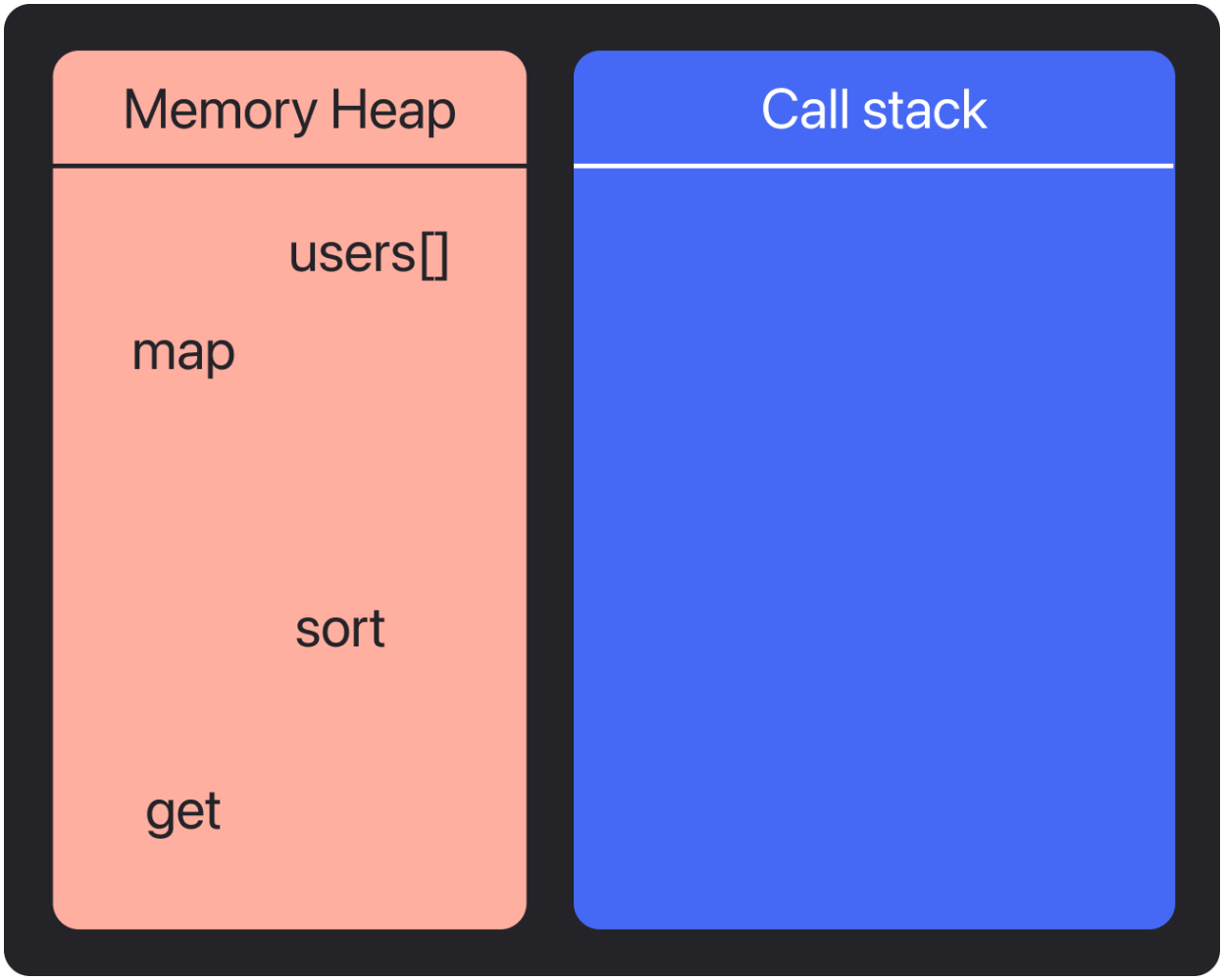
Lista de usuarios obtenida



Lista de usuarios ordenada

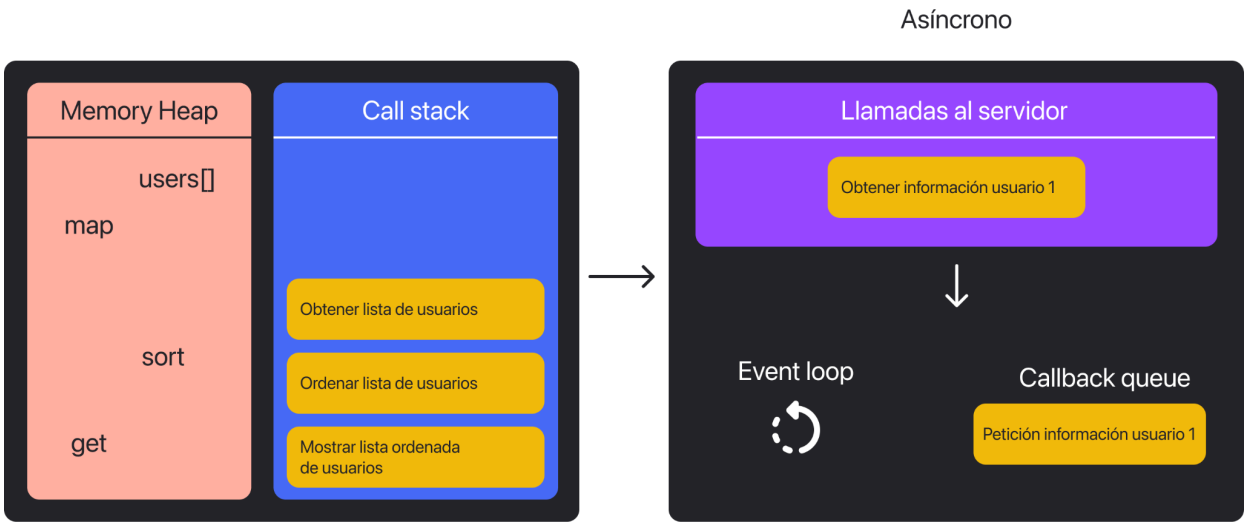


Call stack vacio



Ahora, necesitamos obtener la información de cada usuario y para eso necesitamos realizar una petición al servidor para obtenerla.

Para esto, nuestros eventos del call stack realizará una tarea asíncrona, que será la petición al servidor. Esta tarea será agregada al **callback queue**, será procesada y cuando esté lista será resuelta por el **event loop**.



Callbacks

Los **callbacks** son funciones que se pasan como argumentos a otras funciones y que se ejecutan después de que una operación asíncrona haya sido completada o en respuesta a un evento. En JavaScript, los **callbacks** son una forma común de trabajar con operaciones asíncronas.

Cuando una función realiza una operación asíncrona, como una solicitud de red o una lectura de archivo, en lugar de esperar a que la operación se complete, se proporciona un **callback**. Este **callback** es una función que se ejecutará más tarde, una vez que la operación asíncrona haya terminado y los datos estén disponibles o el evento haya ocurrido.

JavaScript

```
function requestData(callback) {
  // Simular una solicitud de red que toma tiempo
  setTimeout(function() {
    // Una vez completada la solicitud, se llama al callback
    callback({ data: 'Datos de la solicitud' });
  }, 1000);
}
```

```
// Llamada a la función requestData con un callback
requestData(function(response) {
  console.log('Datos recibidos:', response.data);
});
```

En este ejemplo, requestData es una función que realiza una simulación de una solicitud de red que toma un tiempo (simulada mediante setTimeout). La función requestData toma un argumento **callback**, que es una función que se ejecutará una vez que la solicitud de red haya sido completada. Cuando se completa la solicitud de red, se llama al callback con los datos obtenidos como argumento.

Los callbacks son una forma importante de manejar operaciones asíncronas en JavaScript y son utilizados ampliamente en bibliotecas y marcos, así como en el desarrollo de aplicaciones web y de servidor. Sin embargo, el uso excesivo de **callbacks** anidados puede llevar a lo que se conoce como "***callback hell***" (infierno de callbacks), que puede dificultar la legibilidad y mantenibilidad del código. Para evitar este problema, se pueden utilizar otras técnicas como Promesas o async/await.

XMLHTTPRequest

XMLHttpRequest (XHR) es una interfaz proporcionada por los navegadores web que permite realizar solicitudes HTTP desde JavaScript, de manera asíncrona, sin necesidad de recargar toda la página. Es una tecnología fundamental para la comunicación entre el cliente y el servidor en aplicaciones web modernas.

Con XMLHttpRequest, se pueden realizar diferentes tipos de solicitudes HTTP, como GET, POST, PUT y DELETE. Además, se puede enviar y recibir datos en varios formatos, como texto plano, XML, JSON, entre otros.

El XMLHttpRequest, tiene 5 estados:

1. Inicializado.
2. Cargando
3. Cargado
4. Procesando (En caso de procesar alguna descarga)
5. Completado

Por ejemplo:

```javascript
function request(url, callback) {
  var xhttp = new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4) {
      if (this.status == 200) {
        // La solicitud se completó correctamente
        callback(null, xhttp.responseText);
      } else {
        // La solicitud falló
        callback(new Error('Error en la solicitud: ' + xhttp.status));
      }
    }
  };
  xhttp.open("GET", url, true);
  xhttp.send();
}

// Ejemplo de uso de la función request
request("https://ejemplo.com/ruta", function(error, response) {
  if (error) {
    console.error(error);
  } else {
```

```
    console.log("Respuesta recibida:", response);
  }
});
```

En este ejemplo hicimos lo siguiente:

1. Creamos una función llamada **request** que recibe dos parámetros: **url** y **callback**.
2. Creamos una instancia de XMLHttpRequest y lo guardamos en la variable **xhttp**, esta es utilizada para realizar la petición HTTP.
3. Manejamos el estado del evento con la función onreadystatechange, para recibir una notificación cuando la petición cambia de estado. Adicionalmente, validamos que cuando el estado de la petición es 4 (estado completado), y la respuesta de la petición sea 200 (respuesta http ok), ejecutaremos el callback y le pasamos el error como null y pasamos la respuesta.
4. En caso de que la petición sea completada, pero la petición no sea resuelta con 200, ejecutamos el callback, pero pasándole como parámetro un error.
5. Abrimos la conexión con el servidor con el método **open** y especificando el tipo de solicitud (“GET”), la url a donde hacer la petición y pasandole true al final, para que sea asíncrona.
6. Enviamos la solicitud al servidor con el método send.
7. Ejecutamos la petición request, donde la pasamos la url, y un callback que obtiene un error y la respuesta.

XMLHttpRequest es el primer acercamiento al asincronismo, y aunque funciona, tiene algunas desventajas o limitaciones como:

- Sintaxis compleja y propensa a errores
- Manejo manual de los errores
- Callback hell
- Problemas de CORS

Callback hell

El `callback hell` es el código JavaScript se vuelve difícil de leer y mantener, debido a múltiples niveles de anidamiento de funciones de devolución de llamada (callbacks). Esto suele ocurrir cuando se trabaja con operaciones asíncronas que dependen unas de otras, como las solicitudes de red o las operaciones de E/S.

Cuando se anidan múltiples callbacks uno dentro del otro, el código puede volverse rápidamente complicado y difícil de seguir. Esto puede hacer que el código sea propenso a errores, difícil de depurar y complicado de mantener en el futuro.

Aquí hay un ejemplo de cómo podría verse el callback hell:

```JavaScript
const API = 'https://www.example.com';
fetchData(`${API}/products`, function(error1, products){
   if(error1) return throw Error(error1)
      fetchData(`${api}/products/${products[0].id}`, function(error2, product){
      if(error1) return throw Error(error2)
         fetchData(`${api}/categories/${product.category}`, function(error3,
category){
         if(error1) return throw Error(error3)
            console.log(category)
      })
   })
})
```

En este ejemplo:

- Se realizan múltiples solicitudes al servidor utilizando la función fetchData, y cada solicitud depende del resultado de la anterior.

- Para cada solicitud, se realiza un procesamiento de los datos recibidos utilizando funciones de devolución de llamada anónimas, lo que lleva a un anidamiento excesivo de callbacks.
- El código se vuelve difícil de leer y seguir a medida que se anidan más y más callbacks, lo que lleva al callback hell.

Promesas

Las promesas en javascript son objetos que representan un resultado eventual de una operación asíncrona. Estas se pueden resolver en 3 momentos: Ahora mismo, en x tiempo, o nunca. Las promesas son utilizadas para manejar tareas asíncronas de manera más limpia y fácil de leer, evitando los callbacks anidados o callback hell.

Las promesas tienen 3 estados:

1. **Pending (Pendiente):** Cuando la promesa está en proceso.
2. **Fulfilled (Cumplida):** Cuando la promesa se completó de manera exitosa.
3. **Rejected (Rechazada):** Cuando la promesa falló.

Las promesas, permiten encadenar operaciones asíncronas de una manera más estructurada, a través de los métodos `then`, `catch`**,** `finally`; donde el método `then`, se utiliza para obtener la información de la petición anterior y poder transformarla o manipularla, el `catch` para manejar los errores de la petición en caso de que se presenten, y el `finally` para poder saber cuando la petición finalizó.

```JavaScript
const API = 'https://www.example.com';
const fetchData = (url) =>
 new Promise((resolve, reject) => {
   fetch(url)
     .then((response) => {
```

```
      if (!response.ok) {
        reject(
          new Error("Error en la petición Fetch: " + response.statusText)
        );
      }
      resolve(response.json());
    })
    .catch((error) => reject(error));
});

fetchData(`${API}/products`)
 .then((products) => {
   return fetchData(`${API}/products/${products[0].id}`);
 })
 .then((product) => {
   return fetchData(`${API}/categories/${product.category}`);
 })
 .then((category) => {
   console.log(category);
 })
 .catch((error) => {
   console.error("Se produjo un error:", error);
 })
.finally(() => {
   console.log("Petición finalizada")
});
```

En este código, hemos envuelto las llamadas Fetch dentro de una nueva promesa para asegurarnos de que estemos utilizando promesas de manera explícita. La función `fetchData` ahora devuelve una nueva promesa que se resuelve con los datos obtenidos de la URL

proporcionada o se rechaza si ocurre algún error. Luego, encadenamos estas promesas para manejar las llamadas de manera secuencial y limpiar el código.

Podemos observar, como el uso de las promesas nos permite leer de una mejor manera y escribir peticiones sin validar los errores en cada paso, como lo deberíamos hacer con los callback.

Funciones asíncronas

Las funciones asíncronas que utilizan las palabras clave `async` y `await`, fueron introducidas en ES8 y proporcionan una sintaxis más limpia y legible para trabajar con operaciones asìncronas en Javascript.

- Async: La palabra clave async se utiliza para declarar una función que devuelve una promesa, y a su vez permite utilizar la palabra clave await dentro del scope de la función.

- Await: La palabra clave await se utiliza dentro de la función async para esperar que una promesa se resuelva antes de continuar con la ejecución del código. Esto significa que se puede esperar a que la operación asíncrona se termine, sin bloquear el hilo principal de ejecución.

```JavaScript
const API = 'https://www.example.com';
const fetchData = (url) =>
 new Promise((resolve, reject) => {
   fetch(url)
     .then((response) => {
       if (!response.ok) {
         reject(
           new Error("Error en la petición Fetch: " + response.statusText)
```

```
          );
        }
        resolve(response.json());
      })
      .catch((error) => reject(error));
});

const getData = async() => {
   const products = await fetchData(`${API}/products`);
   const product = await fetchData(`${API}/products/${products[0].id}`);
   const category = await fetchData(`${API}/categories/${product.category}`);
   console.log(category);
};

getData();
```

En este código:

- La función `fetchData` es ahora una función async, lo que significa que devuelve una promesa. Dentro de esta función, utilizamos `await` para esperar a que la promesa devuelta por fetch se resuelva y luego a que la promesa devuelta por `response.json()` se resuelva.
- `getData` es otra función async que se encarga de hacer las llamadas secuenciales utilizando `await` para esperar que cada llamada termine antes de continuar con la siguiente.
- Finalmente, llamamos a `getData` para iniciar el proceso de solicitar los datos secuencialmente.

Con esta forma de manejar las peticiones asíncronas, podemos almacenar las respuestas en variables. Sin embargo, no podemos manejar cuando las peticiones no finalizan con éxito.

Try catch

El try/catch es una estructura de control que se utiliza para manejar las excepciones o errores que pueden ocurrir en un bloque de código.

- Try: Este bloque contiene el código que queremos ejecutar y que podría lanzar una excepción. Si ocurre un error dentro de este bloque, la ejecución del código se detiene y pasa al bloque del `catch`.

- Catch: Este bloque se ejecuta si y sólo si, ocurre una excepción dentro del bloque `try`. Se utiliza para manejar y responder a la excepción. Este bloque recibe como parámetro, el error lanzado en el bloque `try`.

- Finally: Es el bloque de código no obligatorio que se ejecuta cuando la ejecución del bloque `try` termina de manera exitosa, o cuando el bloque `catch` finaliza después de ser ejecutado.

JavaScript

```
const API = "https://www.example.com";
const fetchData = (url) =>
 new Promise((resolve, reject) => {
   fetch(url)
     .then((response) => {
       if (!response.ok) {
         reject(
           new Error("Error en la petición Fetch: " + response.statusText)
         );
       }
       resolve(response.json());
     })
     .catch((error) => reject(error));
```

```
 });

const getData = async () => {
 try {
   const products = await fetchData(`${API}/products`);
   const product = await fetchData(`${API}/products/${products[0].id}`);
   const category = await fetchData(`${API}/categories/${product.category}`);
   console.log(category);
 } catch (error) {
   console.error("Se produjo un error:", error);
 } finally {
   console.log("Petición finalizada");
 }
};

getData();
```

En este caso, utilizando el mismo ejemplo del `async/await`, aquí podemos obtener y manejar el error, en caso de que se presente en cualquiera de las peticiones realizadas.

Recursos adicionales

- Pila de ejecución - Call stack | La cocina del código
- ¿Qué es el event loop en javascript? - Eduardo Fierro