Mecanismos de comunicación asíncrona y almacenamiento web

25 ENERO

DWEC

Creado por:

Ignacio Bize Font
Daniel Fernández Benavente
Arturo Fernández-Gallardo García
Alex Fernández Haro
Sandra Fernandez Sanchez
Sergio Malpartida Romero
Fausto Obama Ngomo Afang

JS

Índice

1. Introducción a los Mecanismos de comunicación asíncrona

Se define la comunicación asíncrona como el intercambio de información (en nuestro caso a través de internet) que sucede en dos tiempos distintos, esto es, con una brecha de tiempo entre el envío de la información hasta la recepción de la misma. Por otra parte, la comunicación síncrona sería aquella comunicación en que emisor y receptor intercambian información al mismo tiempo.

Uno de los ejemplos que permite visualizar claramente este tipo de comunicaciones quedaría representado por la comunicación epistolar y telefónica, siendo la primera de ellas un ejemplo de comunicación asíncrona y la segunda un ejemplo de comunicación síncrona.
En el mundo de las telecomunicaciones, la comunicación asíncrona queda representada en herramientas como los correos electrónicos, entornos de trabajo compartidos o aplicaciones de mensajería y redes sociales.

1.1. Características

Las características que presenta la comunicación asíncrona pasan por:

- Flexibilidad temporal, esto es, que no está subordinada a tiempos precisos: no se esperan respuestas simultáneas ni inmediatas.
- La información no desaparece, queda conservada, de la misma manera que permanecen archivados los mensajes en aplicaciones como WhatsApp.
- Al no tener respuestas inmediatas, la velocidad de la comunicación es menor que en el caso de la comunicación síncrona debido a no tratarse de una comunicación en tiempo real.
- Los datos que pueden ser compartidos pueden ser más complejos ya que pueden revisarse y comprender distintos tipos de archivos.
- Los servicios en la nube permiten que, mediante la comunicación asíncrona, dos o más personas puedan trabajar sobre un mismo proyecto o documento.
- Así, en este tipo de comunicación, el intercambio de información puede darse entre un par de usuarios o entre un conjunto, ya sea información escrita o audiovisual.

Así, en este tipo de comunicación, el intercambio de información puede darse entre un par de usuarios o entre un conjunto, ya sea información escrita o audiovisual.

1.2. Ventajas

Dadas estas características, las principales ventajas que presenta este tipo de comunicación residen en la nueva estructura posible a la hora de trabajar con información, como puede ser establecer equipos de trabajo que no se encuentren físicamente cerca ya que no hay obstáculos en este tipo de comunicación. De igual manera, este tipo de estructuras permite trabajar de manera menos estresante e invasiva. Al tratarse de información con permanencia, la reflexión y modificación de los mensajes permite la elaboración de los mismos de una manera más trabajada. Por último, no es necesario la conexión simultánea de las partes que van a mantener la comunicación.

1.3. Inconvenientes

Algunas de las desventajas de este tipo de comunicación se basan en la falta de sincronía de las partes involucradas en la interacción, ya que, de este modo, pueden presentarse retardos en la retroalimentación o *feedback* de los mensajes, no hay certeza de saber si nuestro interlocutor recibe los mensajes o de si se encuentra conectada.

2. Áreas de aplicación

2.1. Comunicación en la nube

La comunicación asíncrona en la nube se refiere a la capacidad de compartir y acceder a información en tiempo real a través de internet. Esto se logra mediante la utilización de servicios de nube, como Google Drive o Dropbox, que permiten a los usuarios almacenar y compartir archivos en línea. También se pueden utilizar plataformas de comunicación asíncrona, como Slack o Microsoft Teams, para colaborar en tiempo real en proyectos o discutir temas de trabajo. Estas herramientas permiten a los usuarios acceder a la información en cualquier momento y lugar, lo que aumenta la eficiencia y la productividad en el trabajo colaborativo.

Además, la comunicación asíncrona en la nube también permite a los usuarios trabajar en equipos distribuidos geográficamente, ya que no requiere que todos los miembros del equipo estén en línea al mismo tiempo. Esto es especialmente útil para equipos de trabajo que tienen miembros en diferentes zonas horarias o para equipos de trabajo remotos.

Otra ventaja de la comunicación asíncrona en la nube es la posibilidad de mantener un registro de la comunicación, lo que permite a los usuarios acceder a la información antigua y revisar cuando sea necesario. Esto es especialmente útil para equipos de trabajo que deben seguir un proceso estricto y cumplir con regulaciones y normas.

La comunicación asíncrona en la nube es una herramienta poderosa para el trabajo colaborativo, ya que permite a los usuarios compartir y acceder a información en tiempo real, colaborar en proyectos y discutir temas de trabajo, independientemente de su ubicación geográfica, y permite mantener un registro de la comunicación.

Ejemplo:

Un ejemplo práctico de comunicación asíncrona en la nube podría ser un equipo de trabajo distribuido geográficamente que está trabajando en un proyecto. El equipo utiliza una plataforma de comunicación asíncrona en la nube, como *Slack*, para colaborar en tiempo real. Los miembros del equipo pueden compartir archivos, comentar en discusiones y mantenerse al día con los progresos del proyecto en cualquier momento y desde cualquier lugar.

Por ejemplo, el líder del proyecto puede crear un canal en *Slack* para discutir el progreso de una tarea específica. Los miembros del equipo pueden unirse al canal y compartir informes de progreso, preguntas y comentarios. Esto permite a los miembros del equipo mantenerse al día con el progreso del proyecto sin tener que programar una reunión en persona o por teléfono.

En otro caso, un miembro del equipo puede compartir un documento en Google Drive para revisión y comentarios, otro miembro del equipo puede agregar comentarios y sugerencias en el documento y notificar al resto del equipo mediante una mención en un canal de *Slack*, de esta manera pueden trabajar en el mismo documento al mismo tiempo sin necesidad de estar en línea al mismo tiempo.

Estos ejemplos ilustran cómo un equipo de trabajo puede utilizar la comunicación asíncrona en la nube para colaborar en tiempo real en un proyecto, independientemente de su ubicación geográfica, aumentando la eficiencia y la productividad en el trabajo colaborativo.

Un ejemplo de código para utilizar la comunicación asíncrona en la nube utilizando JavaScript podría ser el uso de la API de Firebase para enviar un mensaje a una base de datos en tiempo real. A continuación, se muestra un ejemplo de código utilizando el SDK de Firebase para JavaScript:

```
// Inicializamos Firebase
var firebaseConfig = {
    apiKey: "llave de la api",
    authDomain: "dominio del autor",
    databaseURL: "url de la base de datos",
    projectId: "id del proyecto",
    storageBucket: "your-project.appspot.com",
    messagingSenderId: "00000000000",
    appId: "id de la app"
};      You, 1 second ago • Uncommitted changes

firebase.initializeApp(firebaseConfig);

// Obtenemos una referencia a la base de datos
var database = firebase.database();

// Enviamos un mensaje a la base de datos
database.ref("mensajes").push({
    texto: "Hola equipo, ¿cómo están?",
    autor: "Juan"
});
```

En este ejemplo se utiliza el método "*push()*" de la API de *Firebase* para agregar un nuevo elemento a la rama "mensajes" de la base de datos. El nuevo elemento contiene el texto del mensaje y el autor. Cualquier dispositivo o aplicación conectada a la base de datos en tiempo real recibirá automáticamente el nuevo mensaje, lo que permite una comunicación en tiempo real entre los dispositivos.

Es importante mencionar que para poder ejecutar este código es necesario tener configurado un proyecto en *Firebase*, y tener las credenciales correctas en el *firebaseConfig*, y también es importante tener cargado el SDK de *Firebase* en tu proyecto.

2.2. Comunicación asíncrona en GitHub

La comunicación asíncrona en *Git/GitHub* se refiere a la forma en que varios desarrolladores colaboran en un proyecto de código utilizando el sistema de control de versiones *Git* y el repositorio en línea GitHub.

Con *Git*, los desarrolladores pueden descargar una copia del repositorio (conocido como un "clon"), realizar cambios en su propia copia local, y luego enviar dichos cambios de vuelta al repositorio (mediante un "*commit*" y un "*push*").

En GitHub, se pueden crear "*issues*" o problemas para discutir problemas o funcionalidades específicas, y los desarrolladores pueden crear "*pull requests*" para proponer cambios específicos al código. Otros desarrolladores pueden revisar y discutir estos cambios antes de aceptarlos o rechazarlos.

En resumen, *Git* y *GitHub* permiten a los desarrolladores trabajar en un proyecto de forma simultánea y asíncrona, permitiendo la colaboración a distancia y la discusión de cambios antes de aceptarlos.

Además, GitHub tiene un sistema de "*branches*" (ramas) que permite a los desarrolladores trabajar en versiones diferentes del código al mismo tiempo. Por ejemplo, un desarrollador puede crear una rama para una nueva funcionalidad y otro desarrollador puede trabajar en una rama para arreglar un error, mientras que la rama principal del proyecto sigue siendo estable.

También se pueden crear "*pull requests*" para fusionar ramas diferentes, lo que permite a los desarrolladores revisar y discutir los cambios antes de aceptarlos en la rama principal del proyecto.

En general, la comunicación asíncrona en *Git/GitHub* permite a los desarrolladores colaborar de forma eficiente y organizada en un proyecto de código, independientemente de su ubicación geográfica o el tiempo que puedan dedicar al proyecto.

En GitHub existe una característica llamada "*code review*" que permite a los desarrolladores revisar y comentar el código de otros antes de que sea fusionado en una rama principal, esto ayuda a mejorar la calidad del código y a identificar posibles problemas antes de que se conviertan en errores en el proyecto.

Otra característica útil es la capacidad de crear "etiquetas" o "*labels*" en los "*issues*" y "*pull requests*" para organizar y priorizar el trabajo. Esto permite a los desarrolladores saber qué tareas son más urgentes y cuáles necesitan ser abordadas primero.

Git y GitHub son herramientas valiosas para la comunicación asíncrona en el desarrollo de software, ya que permiten una colaboración eficiente y organizada, una revisión de código y un seguimiento de los problemas y funcionalidades del proyecto.

Ejemplo:

Un ejemplo práctico de esto sería el cómo se puede utilizar la librería de GitHub "octokit" para interactuar con un repositorio de GitHub de forma asíncrona:

```
//Utilizamos la libreria octokit para obtener la lista de commits en un repositorio especifico
const Octokit = require('@octokit/rest');
const octokit = new Octokit();

async function getRepoCommits(owner, repo) {
    try {
        //La palabra reservada await es para que espere a que termine la promesa devuelta antes de continuar
        // con el codigo
        const commits = await octokit.repos.listCommits({
            owner,
            repo
        });
        //Mostramos la lista con los commits obtenidos
        console.log(commits);
    } catch (err) {
        console.error(err);
    }
}
//Llamamos a la función poniendo como primer paramero el propietario del repositorio y como segundo el el nombre del repositorio
getRepoCommits('octocat', 'Hello-World');
```

2.3. Comunicación asíncrona en correos electrónicos

La comunicación asíncrona en el correo electrónico se refiere a la capacidad de enviar y recibir mensajes de correo electrónico en cualquier momento, sin necesidad de estar en línea al mismo tiempo. Esto significa que el remitente y el destinatario no tienen que estar conectados al mismo tiempo para comunicarse. Este es uno de los principales beneficios del correo electrónico, ya que permite a las personas comunicarse a su propio ritmo y en su propio tiempo. Sin embargo, también puede dificultar la resolución de problemas urgentes o la toma de decisiones importantes.

La comunicación asíncrona en el correo electrónico tiene varios beneficios. En primer lugar, permite a las personas comunicarse en su propio tiempo, lo que es especialmente útil para aquellos que tienen horarios ocupados o que se encuentran en diferentes zonas horarias. También permite a los usuarios leer y responder a los mensajes cuando sea más conveniente para ellos, lo que puede mejorar la productividad.

Además, el correo electrónico asíncrono permite una mayor flexibilidad en la comunicación. Los mensajes de correo electrónico pueden ser enviados y recibidos desde cualquier lugar con acceso a internet, lo que significa que las personas pueden comunicarse desde la oficina, desde casa o mientras viajan. Esto es especialmente útil para las empresas que tienen empleados que trabajan desde casa o en diferentes ubicaciones geográficas.

Sin embargo, la comunicación asíncrona también tiene algunos desafíos. Puede ser difícil resolver problemas urgentes o tomar decisiones importantes si las respuestas a los mensajes de correo electrónico no son recibidas de manera oportuna. También puede haber malentendidos debido a la falta de tono o contexto en los mensajes de texto escritos. Por último, también existe el riesgo de que los correos electrónicos importantes se pierdan o no sean vistos entre una gran cantidad de correos no importantes.

Algunos ejemplos de cómo se puede utilizar la comunicación asíncrona en el correo electrónico son:

- Comunicación entre equipos: Los miembros de un equipo pueden enviar y recibir correos electrónicos para discutir proyectos, compartir informes y documentos, y actualizarse mutuamente sobre el progreso, sin necesidad de estar en línea al mismo tiempo.

- Comunicación entre empresas: Las empresas pueden utilizar el correo electrónico para comunicarse con clientes, proveedores y otras empresas, lo que permite una mayor flexibilidad y eficiencia en las transacciones comerciales.
- Comunicación con el servicio al cliente: Las empresas pueden proporcionar servicio al cliente a través del correo electrónico, lo que permite a los clientes enviar preguntas y recibir respuestas a su propio ritmo.

En cualquiera de estos casos, es importante tener en cuenta que la comunicación asíncrona en el correo electrónico no debe ser la única forma de comunicación, ya que puede ser difícil resolver problemas urgentes o tomar decisiones importantes de manera efectiva. Es importante también utilizar otras formas de comunicación como videollamadas o chat en línea, para asegurar una comunicación efectiva y oportuna.

2.4. Comunicación asíncrona en WhatsApp

La comunicación asíncrona en WhatsApp se refiere a la capacidad de enviar y recibir mensajes de texto, imágenes, videos y audio en cualquier momento, sin necesidad de estar conectado al mismo tiempo. Al igual que el correo electrónico, esto significa que el remitente y el destinatario no tienen que estar en línea al mismo tiempo para comunicarse.
Algunos beneficios de la comunicación asíncrona en WhatsApp son:
Facilidad de uso: WhatsApp es una aplicación fácil de usar, que se encuentra disponible en diferentes sistemas operativos y dispositivos móviles, lo que hace que sea fácil de utilizar para la mayoría de las personas.

- Comunicación en grupo: WhatsApp permite crear grupos de chat para comunicarse con varias personas al mismo tiempo, lo que es especialmente útil para coordinar eventos, proyectos o grupos de amigos.
- Compartir multimedia: WhatsApp permite enviar y recibir fotos, vídeos, audio y documentos, lo que puede ser especialmente útil para compartir información o materiales relacionados con un proyecto o un evento.

Sin embargo, también existen desventajas en la comunicación asíncrona en WhatsApp, como la posibilidad de malinterpretar los mensajes debido a la falta de tono y contexto en los mensajes de texto escritos, y el riesgo de recibir un gran volumen de mensajes no deseados o notificaciones irrelevantes.

En resumen, la comunicación asíncrona en WhatsApp puede ser una herramienta útil para comunicarse con otras personas, ya sea en un contexto personal o profesional. Sin embargo, es importante utilizar esta herramienta de manera responsable y tener en cuenta las posibles desventajas.

WhatsApp también es una herramienta valiosa para las empresas, ya que les permite comunicarse con sus clientes de manera rápida y eficiente, lo que puede mejorar la satisfacción del cliente y aumentar la lealtad. También puede ser utilizado para el seguimiento de pedidos, la gestión de inventarios y la atención al cliente.

3. Comunicación asíncrona en JavaScript

JavaScript es un lenguaje de programación síncrono por naturaleza. Esto significa que las instrucciones se ejecutan en orden, una después de otra, y el programa se detiene hasta que se complete cada instrucción antes de continuar con la siguiente. No obstante, también dispone de características asíncronas como la función *setTimeout*, encargada de simular cierta asincronía en el programa; y de mecanismos para su gestión.

En resumen, JavaScript es un lenguaje de programación síncrono por naturaleza, pero cuenta con características asíncronas que permiten al programador manejar tareas que pueden tardar en completarse y ejecutar eventos en el futuro.

3.1. Concurrencia y *EventLoop*

3.1.1. Concurrencia

JavaScript es principalmente utilizado en el lado del cliente en navegadores web, y la concurrencia en este entorno se refiere a cómo se manejan varias tareas al mismo tiempo en el navegador.

Para comenzar, detallaremos aquello en lo que consiste la concurrencia.

La concurrencia se refiere al manejo simultáneo de varias tareas, es decir, varios procesos o hilos que se ejecutan al mismo tiempo. Esto permite que un programa pueda realizar varias tareas al mismo tiempo, mejorando la eficiencia y el rendimiento del sistema.

La concurrencia es especialmente importante en sistemas con varios núcleos o procesadores, ya que permite aprovechar al máximo la capacidad del sistema para realizar varias tareas al mismo tiempo.

JavaScript es un lenguaje interpretado y, como tal, su código es ejecutado directamente por el intérprete del lenguaje en tiempo de ejecución. Sin embargo, JavaScript cuenta con características de programación asíncrona que lo diferencian de otros lenguajes interpretados.

A nivel de procesos, JavaScript se ejecuta en un único hilo de ejecución, lo que significa que sólo puede realizar una tarea a la vez. Esto significa que si una tarea bloqueante, como una operación de entrada/salida, es ejecutada, el hilo de ejecución se bloqueará hasta que la tarea sea completada.

En cambio, la programación asíncrona permite a JavaScript ejecutar varias tareas al mismo tiempo, lo que permite una mejor eficiencia del código. Esto se logra mediante el uso de colas de eventos y *callbacks*, que permiten que tareas no bloqueantes sean colocadas en una cola y ejecutadas en orden cuando el hilo de ejecución está disponible.

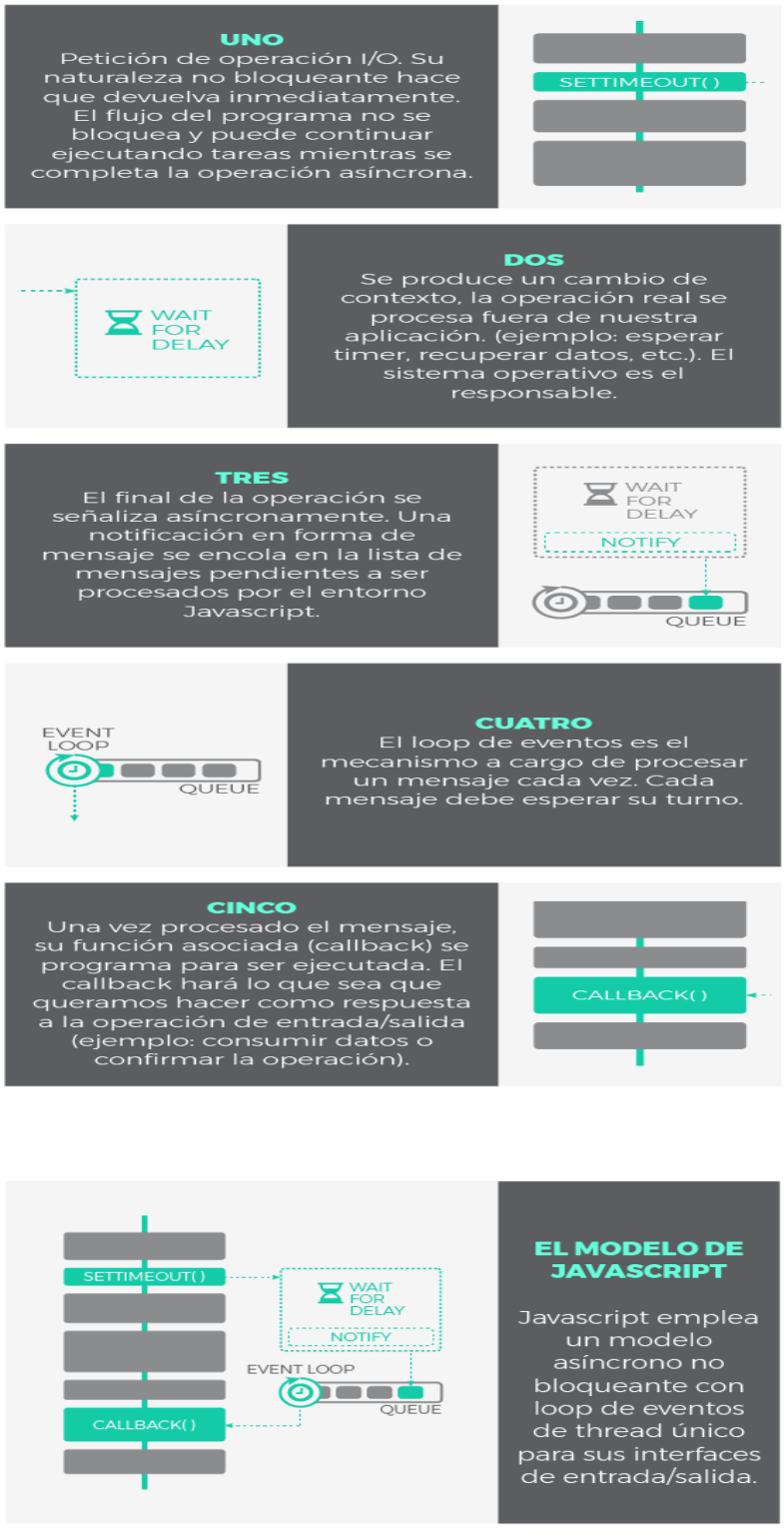
Por ejemplo, si una tarea asíncrona es ejecutada, como una petición HTTP, el hilo de ejecución no se bloqueará mientras espera la respuesta, sino que seguirá ejecutando el código siguiente. Cuando la respuesta llega, un *callback* es colocado en la cola de eventos y es ejecutado en el siguiente ciclo de eventos.

3.1.2. *Event Loop*

Es el mecanismo que permite a JavaScript ejecutar tareas asíncronas. Maneja las colas de eventos y callbacks.

Cuando se ejecuta una tarea asíncrona, como un setTimeout o una petición HTTP, el event loop coloca un callback en la cola de eventos y lo ejecuta en el siguiente ciclo de eventos, cuando el hilo de ejecución está disponible.

La asincronía sigue el siguiente esquema:



Un ejemplo de ejecución asíncrona a nivel interno sería el siguiente:

El funcionamiento de las funciones asíncronas, *async/await* a nivel de CPU se basa en la utilización de un mecanismo llamado "*task queue*" (cola de tareas) y "*event loop*" (bucle de eventos).

Cuando se ejecuta una función asíncrona con el operador "*await*", la ejecución del código dentro de la función se detiene hasta que la promesa especificada se resuelve o rechaza. Mientras tanto, el hilo de ejecución continúa ejecutando el código fuera de la función asíncrona.

Cuando la promesa se resuelve, se crea una tarea o *callback* para manejar la respuesta y esta tarea es colocada en la cola de tareas. El *event loop*, que es un bucle en segundo plano que se ejecuta constantemente, detecta que hay una tarea pendiente en la cola de tareas y la ejecuta. Una vez que se ejecuta el *callback*, el valor de la promesa se asigna a una variable y la ejecución del código dentro de la función asíncrona continúa.

A nivel de CPU, esto significa que el hilo de ejecución no se bloquea mientras se espera a que la promesa se resuelva, sino que continúa ejecutando otras tareas mientras tanto. Esto permite una mejor eficiencia del código y una mayor escalabilidad en aplicaciones con un alto nivel de concurrencia.

Es importante mencionar que las funciones *async/await* no crean nuevos hilos de ejecución, sino que trabajan con el hilo principal y utilizan el mecanismo de *event loop* para programar tareas y manejar su ejecución.

3.2. Herramientas para la comunicación asíncrona

A continuación, se desarrollarán los distintos mecanismos disponibles en JavaScript, para gestionar la asincronía en este lenguaje.

3.2.1. Funciones Callback

Son funciones que se pasan como argumento a otra función y se ejecuta cuando la función principal ha terminado. Son uno de los mecanismos de comunicación asíncrona más antiguos en JavaScript y se utilizan para manejar tareas asíncronas.

Por ejemplo, supongamos que quieres descargar un archivo de internet. En lugar de bloquear el hilo principal de ejecución mientras se descarga el archivo, puedes usar un callback para manejar la descarga. La función de descarga tomaría el callback como argumento y lo ejecutaría una vez que la descarga se haya completado.

Ejemplo:

```
function downloadFile(url, callback) {
  // logica para descargar el archivo
  ...
  callback();
}

downloadFile("example.com/file.txt", function(){
  console.log("El archivo se descargó correctamente");
});
```

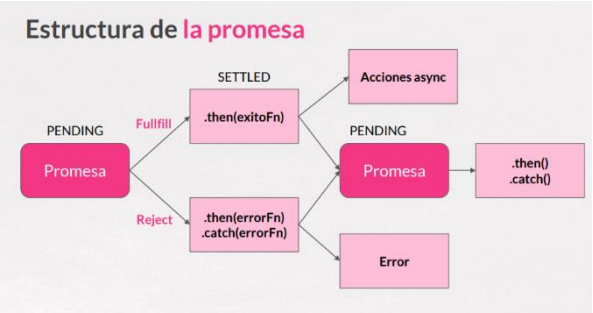
En este ejemplo, la función downloadFile toma dos argumentos: la URL del archivo y el callback. Dentro de la función de downloadFile, se lleva a cabo la lógica para descargar el archivo. Una vez que la descarga se ha completado, se ejecuta el callback pasado como argumento.

Sin embargo, el uso excesivo de los callbacks puede generar un problema conocido como "Callback Hell", donde el código se vuelve ilegible y difícil de mantener debido a la acumulación de callbacks anidados.

Para solucionar esto, se introdujo el uso de promesas y async/await. Las promesas permiten manejar tareas asíncronas de una manera más clara y legible, ya que utilizan una sintaxis similar a la de los callbacks, pero con una estructura de control más sencilla. El uso de async/await permite escribir código asíncrono de manera síncrona, lo que lo hace más fácil de leer y comprender.

3.2.2. Promesas

Una promesa es un objeto que representa el resultado de una operación asíncrona. Tiene dos estados: pendiente y cumplida.
Una vez que una se cumple, se ejecuta una función "then" que maneja el resultado.



Se inicializa con una función constructor que toma dos argumentos: una función resolve y una función reject.
La función resolve se ejecuta cuando la promesa se cumple y la función reject se ejecuta cuando la promesa falla.

Ejemplo:

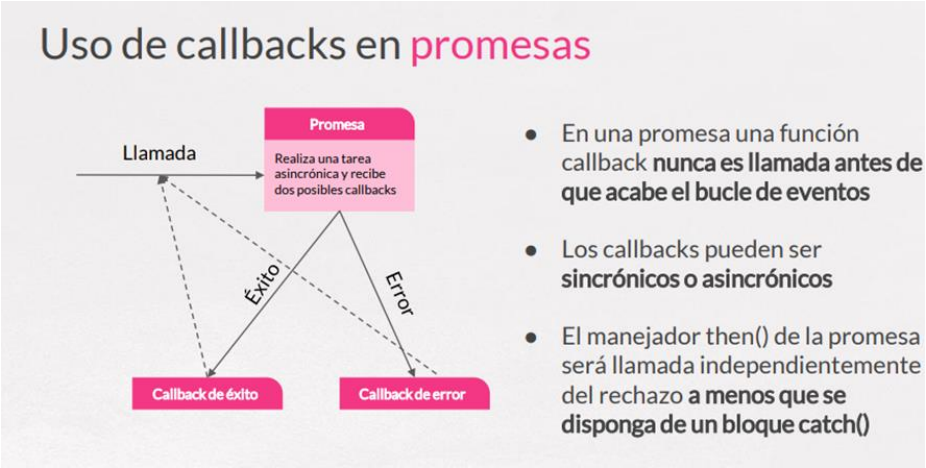
Por ejemplo, si quieres descargar un archivo de internet con una promesa, podrías escribir algo así:

```
function compruebaNombre(name) {
    return new Promise(function (resolve, reject) {
        if (name === "pablo") {
            resolve("Bien, te llamas Pablo");
        } else {
            reject("Un momento, tu no eres Pablo");
        }
    });
}

compruebaNombre('Pablo')
    .then(response => console.log(response))
    .catch(error => console.log(error));
```

En este ejemplo, la función *compruebaNombre* devuelve una nueva promesa, dentro de la cual se lleva a cabo la lógica de la misma. Si se introduce el nombre “pablo” se ejecuta el mensaje de éxito. En caso contrario, se llama a la función "*reject*" pasando un mensaje de error.

Luego, se utilizan los métodos de promesa *then* y *catch* para manejar el resultado de la promesa, en caso de éxito o error respectivamente.

Las promesas son una manera más clara y legible de manejar tareas asíncronas en comparación con los *callbacks*, ya que permiten manejar tanto el éxito como el error en una sola estructura de control, evitando la acumulación de *callbacks* anidados.




```
//callbacks en promesas

const miFuncion = (val) => {
    return new Promise((resolve, reject) => {
        if (val) {
            resolve("el valor es true!");
        } else {
            reject("el valor es false!");
        }
    });
}
```

```
//igual que arriba pero con manejadores no hay que introducir directamente
//las funciones
const funcExito = (res) => {
    console.log(res)
};
const funcError = (res) => {
    console.error(res);
};

miFuncion(true).then(funcExito, funcError);
//Logueará el mensaje de exito por console.log
miFuncion(true).then(funcExito).catch(funcError);
//Logueará el mensaje de exito por console.log

miFuncion(false).then(funcExito, funcError);
//Logueará el mensaje de error por console.error
miFuncion(false).then(funcExito).catch(funcError);
```

3.2.3. Funciones asíncronas *Async – Await*

Async/await es una sintaxis introducida en JavaScript para manejar tareas asíncronas de manera más fácil de leer y comprender. El uso de *async/await* permite escribir código asíncrono como si fuera síncrono, lo que lo hace más fácil de entender y mantener.

La palabra clave "*async*" se utiliza para declarar una función asíncrona y la palabra clave "await" se utiliza para esperar a que una promesa se cumpla antes de continuar con la ejecución del código.

En general, *Async/await* es una manera más fácil de escribir código asíncrono de manera síncrona, lo que lo hace más fácil de leer y comprender. Pero es importante tener en cuenta que Async/await solo puede ser utilizado dentro de funciones marcadas como asíncronas.

Además, es importante mencionar que el uso de *Async/await* no es compatible con navegadores antiguos, y es importante validar esta compatibilidad en caso de ser necesario.

Otro detalle importante a mencionar es que al utilizar *Async/await*, el código se ejecuta de manera secuencial, lo que significa que se espera a que una promesa se cumpla antes de continuar con la ejecución del siguiente código. Esto puede ser beneficioso en algunos casos, como cuando se requiere que una tarea se complete antes de continuar con otra, pero en otros casos, puede afectar el rendimiento de la aplicación ya que se estaría bloqueando la ejecución hasta que se complete la tarea.

Async/await es una forma clara y legible de manejar tareas asíncronas en JavaScript, pero es importante tener en cuenta las limitaciones y consideraciones específicas de su uso, así como manejar correctamente los errores.

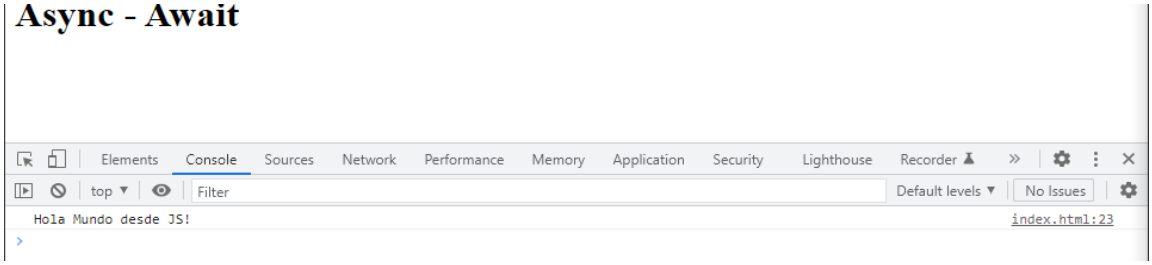
Es importante elegir la forma adecuada de manejar la asincronía en función del contexto y las necesidades específicas del proyecto. En algunos casos, los *callbacks* pueden ser suficientes, mientras que, en otros casos, las promesas o *async/await* pueden ser más adecuadas. Además, es importante tener en cuenta que el uso de *async/await* solo es compatible con versiones más recientes de JavaScript y es importante validar esta compatibilidad en caso de ser necesario.

Ejemplo:

Su implementación se llevaría a cabo de la siguiente manera:

```
'use strict';

let resultado = new Promise((resolve,reject)=>{ // Esta es una promesa
    setTimeout(() => { // Simulamos una demora de 3 segundos a la hora de devolver el saludo
        resolve('Hola Mundo desde JS!'); // Se devuelve una cadena pasados 3 segundos
    }, 3000);

});

async function mostrarSaludo(){ // Esta funcion asíncrona muestra el texto que contiene la variable
    let respuesta = await resultado; // Guardamos la cadena en una variable
    console.log(respuesta); // Imprimimos la variable
}

mostrarSaludo(); // Mostramos el saludo por consola

```

El resultado sería el siguiente, pasados tres segundos:



Se devuelve la información una vez se encuentre disponible.

4. Comunicación asíncrona con servidores

El acceso a datos de o información brindada por un servidor es uno de los aspectos más representativos de JavaScript como lenguaje estructural utilizado en el lado cliente. Este acceso o comunicación posee cierta naturaleza asíncrona, lo cual presenta ciertos inconvenientes en los lenguajes estructurales y síncronos como este caso.
Por este motivo, se implementaron ciertas herramientas encargadas de mitigar el contratiempo que conlleva el desfase en este tipo de comunicaciones, las cuales se detallarán en los apartados posteriores.

4.1. Peticiones AJAX

Ajax, acrónimo de *Asynchronous Javascript* and XML, es una técnica para crear aplicaciones web interactivas (RIA), que se ejecutan en el lado del cliente. Es asíncrona porque se ejecuta en paralelo y de esta forma es posible realizar cambios sobre las páginas sin necesidad de recargarlas.
Está formado por un conjunto de tecnologías independientes que se unen, tales como:

- XHTML y CSS, encargados de la presentación del contenido.
- DOM, para la manipulación dinámica de la presentación.
- XML, JSON y otros, para manipular la información.
- XMLHTTPREQUEST, para el intercambio asíncrono de la información
- JS, para unir las demás tecnologías.

4.1.1. XMLHttpRequest

XMLHttpRequest es un objeto imprescindible para utilizar AJAX, ya que nos permite manipular los datos devueltos por el servidor en procesos de segundo plano,esto es,el proceso no se detiene mientras espera respuesta. Los datos pueden tener formato simple o XML.

La manera de instanciarlo es de la siguiente manera:

```
httpRequest = new XMLHttpRequest();
```

Entre sus atributos podemos encontrar:

- **readyState**: Devuelve el estado del objeto, siendo 0 sin inicializar, 1 abierto, 2 cabeceras recibidas, 3 cargando y 4 completado.
- **responseBody**: Devuelve la respuesta como un array de bytes.
- **responseText**: Devuelve la respuesta como una cadena.
- **responseXML**: Devuelve la respuesta como XML.
- **status**: Devuelve el estado como un número, siendo 200 respuesta correcta, 404 no encontrado y el 500 error interno del servidor.
- **statusText**: Devuelve el estado como una cadena.

Y entre sus métodos se destacan:

- **abort()**: Cancela la petición en curso.
- **getAllResponseHeaders()**: Devuelve todas las cabeceras HTTP de la respuesta del servidor.
- **GetResponseHeader(nombreHeader)**: Devuelve el valor de la cabecera HTTP especificada.
- **open(método, URL,[asíncrono[nombreUser[clave]]])**: Especifica el método, la URL y otros atributos opcionales de una petición.
- **send([datos])**: Envía la petición a la URL especificada en open. Ha de hacerse posteriormente a un open, nunca antes.
- **setTimeout(tiempo)**: Indica la duración máxima de la petición.
- **SetRequestHeader(etiqueta,valor)**: Añade un par etiqueta/valor a la cabecera HTTP a enviar. Indicará el formato de las cabeceras enviadas.

Los eventos más importantes de este objeto son:

- **onreadystatechange**: Evento que se dispara con cada cambio de estado.
- **onabort**: Evento que se dispara al abortar la operación.
- **onload**: Evento que se dispara al completar la carga.
- **onloadstart**: Evento que se dispara al iniciar la carga.
- **onprogress**: Evento que se dispara periódicamente con información de estado
- **error**: Si se produce un error procesando la solicitud
- **timeout**: Cuando se alcanza el tiempo especificado, se lanza el evento
- **loadend**: Se dispara cuando la solicitud ha sido completada con o sin éxito

Ejemplo:

Para entender la asincronía en primer lugar tenemos que entender cómo funciona una llamada síncrona:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <a href="javascript:comprobarAjax()">Comprobar Ajax</a>
    <div id="respuesta"></div>
    <script src="index.js"></script>
</body>
</html>
```

```
function comprobarAjax(){
    var miObj=null;
    if(window.XMLHttpRequest){
        //mozilla firefox
        miObj=new XMLHttpRequest();
    }else if (window.ActiveXObject){
        //IExplorer
        miObj=newActiveXObject("Microsoft.XMLHTTP");
    }else{
        alert("el navegador no admite el uso de ajax");
        return;
    }

    miObj.open("GET","http://www.inforbooks.com/cataleg.php", false);
    miObj.send(null);

    //gestion de una llamada sincrona que bloquea la ejecucion hasta que se recibe la respuesta
    if(miObj.readyState==4){
        //utiliza la respuesta para asignar datos a un elemento en pantalla
        document.getElementById("respuesta").innerHTML=miObj.responseText;
    }else{
        document.getElementById("respuesta").innerHTML="no hay respuesta";
    }
}
```

Al ejecutar este código aparece una pantalla con un enlace llamado comprobar *AJAX*, y si se hace click se ejecuta la función *comprobarAjax()*.
La llamada síncrona tiene algunas ventajas como por ejemplo un mejor rendimiento de la navegación ya que la cantidad de datos enviados y recibidos es menor, pero a cambio la página queda bloqueada hasta que termina de ejecutarse el script y por lo tanto el usuario no puede hacer uso de ella. Gracias a la comunicación asíncrona el usuario puede seguir usando la página sin esperar la respuesta del servidor, ya que el script no queda bloqueado y la página sigue siendo operativa.

Para convertir este código en una llamada asíncrona tendríamos que hacer lo siguiente:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>
    <a href="javascript:comprobarAjaxAsy()">Comprobar Ajax asincrono</a>
    <div id="respuesta"></div>
    <a href="http://www.inforbooks.com/cataleg.php/">acceso sin ajax</a>
    <script src="index.js"></script>
</body>
</html>
```

```
function comprobarAjaxAsy(){
    var miObj=null;
    if(window.XMLHttpRequest){
        //mozilla firefox
        miObj=new XMLHttpRequest();
    }else if (window.ActiveXObject){
        //IExplorer
        miObj=newActiveXObject("Microsoft.XMLHTTP");
    }else{
        alert("el navegador no admite el uso de ajax");
        return;
    }

    miObj.open("GET","http://www.inforbooks.com/cataleg.php", true);
    //gestion de una llamada asincrona que no bloquea la ejecucion hasta que se recibe la respuesta
    miObj.onreadystatechange=function(){

        if(miObj.readyState==4){
            //utiliza la respuesta para asignar datos a un elemento en pantalla
            document.getElementById("respuesta").innerHTML=miObj.responseText;
        }
    }
    miObj.send(null);
}
```

Por lo tanto, la diferencia entre ambas es que el parámetro del método open debe ser true y es necesario el uso del evento *onreadystatechange* para definir una función que se ejecuta al producirse el evento. Este evento se produce cuando se modifica el estado del objeto *XMLHttpRequest*.

4.1.2. Fetch

Fetch es una interfaz para hacer solicitudes AJAX mediante la cual podemos acceder y manipular partes del canal HTTP, tales como peticiones y respuestas. También provee un método global *fetch()* que proporciona una forma fácil y lógica de obtener recursos de forma asíncrona por la red.

Este tipo de funcionalidad se conseguía previamente haciendo uso de *XMLHttpRequest*. Fetch proporciona una alternativa mejor que puede ser empleada fácilmente por otras tecnologías. También aporta un único lugar lógico en el que definir otros conceptos relacionados con HTTP como CORS y extensiones para HTTP.

La especificación fetch difiere de *JQuery.ajax()* en dos formas principales:

- El objeto *Promise* devuelto desde *fetch()* no será rechazado con un estado de error HTTP incluso si la respuesta es un error HTTP 404 o 500. En cambio, este se resolverá normalmente (con un estado ok configurado a false), y este solo será rechazado ante un fallo de red o si algo impidió completar la solicitud.
- Por defecto, fetch no enviará ni recibirá cookies del servidor, resultando en peticiones no autenticadas si el sitio permite mantener una sesión de usuario (para mandar cookies, *credentials* de la opción init deberá ser configurada). Desde el 25 de agosto de 2017. La especificación cambió la política por defecto de las credenciales a *same-origin*.

Implementación:

```
1  fetch('http://example.com/movies.json')
2      .then(response => response.json())
3      .then(data => console.log(data));
4
```

Aquí estamos recuperando un archivo JSON a través de red e imprimiendo la información en la consola.

El uso de *fetch()* más simple toma un argumento (la ruta del recurso que quieres obtener) y devuelve un objeto *Promise* conteniendo la respuesta, un objeto Response.
Esto es, por supuesto, una respuesta HTTP no el archivo JSON. Para extraer el contenido en el cuerpo del JSON desde la respuesta, usamos el método *json()*.

Parámetros de la función FETCH:

- URI: El primer parámetro de la función *fetch*. Corresponde a la ubicación del recurso al que deseamos acceder.
- Ajustes de la petición: El segundo parámetro es un objeto de JavaScript en el que se definirían los criterios para la petición, tales como: las cabeceras, el cuerpo, el método, las credenciales, entre otros.

```
{
    method: 'POST', // por defecto GET. Más opciones: POST, PUT, DELETE, etc.
    mode: 'cors', // no-cors,  same-origin, por defecto: cors
    cache: 'no-cache', // por defecto default.  Más opciones: no-cache, reload, force-cache, only-if-cached
    credentials: 'same-origin', // include, *same-origin, omit
    headers: {
    'Content-Type': 'application/json'
    // 'Content-Type': 'application/x-www-form-urlencoded',
    },
    redirect: 'follow', // manual, *follow, error
    referrerPolicy: 'no-referrer', // no-referrer, *no-referrer-when-downgrade, origin,
                                   // origin-when-cross-origin, same-origin, strict-origin, strict-origin-when-cross-origin, unsafe-url
    body: JSON.stringify(data) // Debe coincidir con el tipo de dato en  "Content-Type"
}
```

4.1.3. Axios

Axios es una librería JavaScript que nos permite usar *AJAX* de una manera muy cómoda y potente. Se trata de una capaz de ejecutarse tanto en el navegador como en *NodeJS*, que facilita todo tipo de operaciones como cliente HTTP.

Axios permite realizar solicitudes contra un servidor, completamente configurables, y recibir la respuesta de una manera sencilla de procesar.

La librería está basada en promesas, por lo que al usarla se generará un código asíncrono bastante ordenado.
Es capaz de usar *Async - Await* para mejorar la sintaxis, con la desventaja de tener que transpilar el código para hacerla compatible con ciertos clientes web.

Esta librería resulta útil al trabajar en *frameworks* de JavaScript (como Angular o Vue.js).
Muchas personas usan *jQuery* para poder disponer de los métodos relacionados con Ajax, pero lo cierto es que usar *jQuery* solamente por este motivo es bastante poco aconsejable, pues es una librería bastante pesada, en comparación con *Axios*.

Para aprovechar las características nativas del navegador, usar FETCH puede llegar a ser una alternativa mejor. Sin embargo, esta API no está disponible en todos los navegadores, obligando a instalar algunos *polyfill* para su uso. Esto no resulta perjudicial en tareas sencillas; sin embargo, para tareas más complejas, como el consumo de *APIs REST*, es posible que se presenten inconvenientes. Circunstancias eludibles con la implementación de *Axios*.

Ventajas al usar *Axios*:

- Ofrece una API unificada para las solicitudes Ajax
- Está altamente pensado para facilitar el consumo de servicios web, API REST que devuelvan datos JSON.
- Es muy sencillo de usar y puede ser un complemento ideal para sitios web convencionales, donde no se esté usando *jQuery* y aplicaciones *frontend* modernas basadas en librerías como *React* o *Polymer*, que no disponen en su "core" de mecanismos para hacer de cliente HTTP.
- Tiene muy poco peso (13Kb minimizado y todavía menos si contamos que el servidor lo envía comprimido), en unas pocas Kb nos ahorrará mucho trabajo de codificación en las aplicaciones.
- Aparte de *NodeJS*, *Axios* tiene compatibilidad con todos los navegadores en versiones actuales. Pero ojo, Internet Explorer se soporta solamente a partir de la versión 11. Si deseas que tu Ajax funcione en versiones más antiguas de Explorer, entonces te interesa más *jQuery* o *Fetch* con sus correspondientes *polyfill*.

Implementación

Para trabajar con *Axios* usamos el API de promesas. Esto quiere decir que, cuando se reciba la respuesta del servidor se ejecutará una función *callback* configurada en el "*then*" y cuando se produzca un error (por ejemplo, una respuesta con status 404 de recurso no encontrado) se ejecutará la función *callback* definida por el "catch".

Estructura

- Peticiones GET

```
axios.get('RECURSO' | 'URL', {
    responseType: 'text' // tambien puede ser 'json' segun el tipo de dato esperado
  })
    .then(function(res) {
        console.log(res); // Cuando esté disponible la respuesta, se muestra por consola.
    })
    .catch(function(err) {
      console.log(err); // Si se da algún error, se mostrará por consola
    })

```

- Peticiones POST

```
axios.post('URL', {
    responseType: 'json'
  })
    .then(function(res) {
       console.log(res);
    })
    .catch(function(err) {
      console.log(err);
    })

```

Axios es una buena alternativa a tener en cuenta sobre todo en aplicaciones o componentes donde se requiera de acceso a recursos de un API REST.

Agrega mucho valor gracias a una gran cantidad de opciones de configuración de las solicitudes, simplificarán la comprensión del API con la que se esté conectando. Recibe las respuestas de una manera fácil de procesar. Es una buena alternativa al API de *Fetch + polyfills*, por ser más sencilla de utilizar.

4.2. AJAX con JQuery

JQuery posee varios métodos para trabajar con Ajax. Sin embargo, todos están basados en el método *$.ajax*, por lo tanto, su comprensión es obligatoria. A continuación, se abordará dicho método y luego se indicará un breve resumen sobre los demás métodos.

Generalmente, es preferible utilizar el método *$.ajax* en lugar de los otros, ya que ofrece más características y su configuración es muy comprensible.

El método *$.ajax* es configurado a través de un objeto, el cual contiene todas las instrucciones que necesita JQuery para completar la petición. Dicho método es particularmente útil debido a que ofrece la posibilidad de especificar acciones en caso que la petición haya fallado o no. Además, al estar configurado a través de un objeto, es posible definir sus propiedades de forma separada, haciendo que sea más fácil la reutilización del código.

Implementación

```
$.ajax({
    url : 'post.php', // la URL para la peticion
    data : { id : 123 }, // la informacion a enviar  (tambien es posible utilizar una cadena de datos)
    type : 'GET',  // especifica si sera una peticion POST o GET
    dataType : 'json', // el tipo de informacion que se espera de respuesta

    // codigo a ejecutar si la peticion es satisfactoria;  la respuesta es pasada como argumento a la funcion

    success : function(res) {
       console.log(res);
    },
    // codigo a ejecutar si la peticion falla. Son pasados como argumentos a la funcion
    // el objeto de la peticion en crudo y codigo de estatus de la peticion
    error : function(xhr, status) { // recibe el elemento de la peticion y el estado
        console.log('Mensaje de error');
    },
    // codigo a ejecutar sin importar si la peticion falla o no
    complete : function(xhr, status) { // recibe el elemento de la peticion y el estado
        alert('Petición realizada'); // Por ejemplo
    }
});
```

5. WEB Storage y Cookies

5.1. WEB Storage

Web Storage es un API de JS proporcionada por los navegadores que facilita a los sitios web almacenar datos de forma parecida a las *cookies* pero a mayor escala y sin información en las cabeceras HTTP.

La data del Web Storage no se transmite automáticamente al servidor en cada HTTP *request* y un servidor web no puede escribir directamente al Web Storage, aunque ambas cosas pueden lograrse con scripts el lado del cliente. Hay dos tipos mayores de Almacenamiento Web que son controlados de manera independiente: *Local Storage* y *Session Storage*.

Capacidad de almacenamiento:

Opera 10.20 → 5MB
Safari 8 → 5MB
Firefox 34 → 10MB
Google Chrome → 10MB
Internet Explorer → 10MB

- **sessionStorage**: Utiliza un área de almacenamiento de datos diferente por cada origen que esté disponible mientras no se haya cerrado el navegador y la sesión se mantenga aún iniciada por lo que los datos sobreviven a una recarga de página. En ningún caso se transfiere la data al servidor.

- **localStorage**: Tiene el mismo mecanismo, pero se mantienen los datos cuando el navegador se cierra y demás su límite de almacenamiento es menor. Solo se limpia mediante JS o limpiando el caché del navegador.

Un ejemplo sencillo de ello sería el siguiente:

```
localStorage.colorSetting = '#a4509b';
localStorage['colorSetting'] = '#a4509b';
localStorage.setItem('colorSetting', '#a4509b');
```

En el podemos observar tres formas para el cambio de color del objeto *localStorage*. Algunos navegadores ofrecen la posibilidad de deshabilitarlo. Para saber si el navegador lo soporta y detecta.

Esto sería una función que detecta si *localStorage* está disponible y soportada tendríamos que hacer uso de la función *strageAvailable* que aparece en el siguiente ejemplo:

```
function storageAvailable(type) {
    let storage;
    try {
        storage = window[type];
        const x = '__storage_test__';
        storage.setItem(x, x);
        storage.removeItem(x);
        return true;
    }
    catch (e) {
        return e instanceof DOMException && (
            // todo menos firefox
            e.code === 22 ||
            // Firefox
            e.code === 1014 ||

            // todo menos firefox
            e.name === 'QuotaExceededError' ||
            // Firefox
            e.name === 'NS_ERROR_DOM_QUOTA_REACHED') &&

            (storage && storage.length !== 0);
    }
}
if (storageAvailable('localStorage')) {
    // Yippee! funciona :v
  }
  else {
    // no funciona, kekw
  }
storageAvailable('sessionStorage');
```

Para comprobar si la memoria tiene valores hacemos lo siguiente:

```
if (!localStorage.getItem('bgcolor')) {
    populateStorage();
  } else {
    setStyles();
  }
```

En este caso, si ***localStorage*** nos devuelve que *bgcolor* no existe *bgcolor* existe *usamospopulateStorage()* para añadir los valores personalizados actuales a la memoria y en caso contrario *usamossetStyles()* para actualizar el estilo de la página con los valores almacenados.

Se puede acceder a ambos a través de las propiedades *Window.sessionStorage* y *Window.localStorage* del objeto *Window*, cuya invocación crea una instancia del objeto Storage diferente en cada uno, aunque ambos poseen las mismas funciones: crear, recuperar y eliminar datos.

5.1.1. Funcionalidades

Obtener datos de la memoria:

```
function setStyles() {
    const currentColor = localStorage.getItem('bgcolor');
    const currentFont = localStorage.getItem('font');
    const currentImage = localStorage.getItem('image');

    document.getElementById('bgcolor').value = currentColor;
    document.getElementById('font').value = currentFont;
    document.getElementById('image').value = currentImage;

    htmlElem.style.backgroundColor = `#${currentColor}`;
    pElem.style.fontFamily = currentFont;
    imgElem.setAttribute('src', currentImage);
}
```

En primer lugar recuperamos tres datos del almacenamiento *local*(*bgcolor*, *font* e *image*= y les asignamos los valores de las tres propiedades que haya e ese momento en el formulario y actualizamos el método *.style* logrando así que todos esos valores sobrevivan a la recarga de página.

Guardar datos en la memoria:

Storage.setItem() sirve tanto para crear nuevos data *item* y actualizar los existentes. Para ello serán necesarias la clave del *item* que queremos modificar y el valor que queremos actualizar. Todo ello lo hacemos dentro de la función *populateStorage()* y finalmente aplicamos *setStyes()* para actualizar los estilos de página, como se puede observar a continuación:

```
function populateStorage() {
    localStorage.setItem('bgcolor', document.getElementById('bgcolor').value);
    localStorage.setItem('font', document.getElementById('font').value);
    localStorage.setItem('image', document.getElementById('image').value);

    setStyles();
}
```

Responder a cambios en la memoria con *storageEvent*:

StorageEvent se dispara cada vez que se hace un cambio en el objeto *storage*, pero no trabaja en la misma página en la que se estén realizando los cambios, sino que su objetivo es que las páginas que están haciendo uso de la memoria puedan aplicar sus cambios. no funciona en la misma web que hace los cambios, sino que es una forma para que otras páginas en el domino que usen el *storage* se sincronicen con los cambios realizados. Páginas en otros dominios no pueden acceder a los mismos *storage object*. Un ejemplo de uso será el siguiente:

```
window.addEventListener('storage', (e) => {
    document.querySelector('.my-key').textContent = e.key;
    document.querySelector('.my-old').textContent = e.oldValue;
    document.querySelector('.my-new').textContent = e.newValue;
    document.querySelector('.my-url').textContent = e.url;
    document.querySelector('.my-storage').textContent = JSON.stringify(e.storageArea);
  });
```

En este caso añadimos un evento que se dispara cuando el objeto Storage asociado al origen actual cambia y mostramos el contenido de la clave del *item* modificado, el valor antiguo, el nuevo valor, la url del documento que cambió la memoria y el objeto de almacenamiento convertido previamente a *string* mediante *stringify*.

Borrar registros:

- **Storage.removeItem()**: Recibe como argumento la clave del dato que queremos eliminar y lo elimina del objeto de almacenamiento.

- **Storage.clear()**: No recibe argumentos y vacía todo el objeto.

Importar almacenamiento web asíncrono:

En primer lugar, es necesario ejecutar el comando para habilitar esta opción mediante:

```
npm install async-web-storage
```

A continuación, podemos ver un ejemplo de uso:

```
import { asyncLocalStorage, asyncSessionStorage } from 'async-web-storage';


async function saveLocally() {

    await asyncLocalStorage.setItem('foo', 'bar');

    const foo = await asyncLocalStorage.getItem('foo');// bar


    const person = {

        name: 'mcha',

    };

    await asyncLocalStorage.setItem('user_1', person);

    const user = await asyncLocalStorage.getItem('user_1');

}
```

En este ejemplo se almacenan los valores en el almacenamiento local especificando todas las claves y sus valores. Estos datos se almacenan en forma de String como se muestra a continuación, junto al a fecha de creación:

Key	Value
foo	"{\"foo\":\"bar\",\"createdAt\":1600516661351}"
user_1	"{\"name\":\"mcha\",\"createdAt\":1600516591899}"

Si quisiéramos tener acceso a los objetos almacenados sin procesar podríamos hacer lo siguiente:

```
const rawStoredUser1 = await asyncLocalStorage.getItem('user_1', { raw: true });

console.log(rawStoredUser1);// {name: "mcha", createdAt: 1600516591899}
```

De nuevo, hacemos uso del método *getItem()* y especificamos como segundo parámetro "*raw:true*", que en inglés significa no procesado, para indicarle queremos que nos devuelva los datos que estén sin procesar hasta el momento.

5.2. ASYNC COOKIE API

Para gestionar la información asíncrona de las cookies de HTTP en primer lugar es necesario habilitar la API mediante el siguiente comando:

```
chrome --enable-blink-features=CookieStore
```

Leer una cookie:

Para leer una cookie primero hacemos uso de *cookieStore* y le pedimos la cookie mediante un id, lo que nos devolverá la información en forma de promesa.

Tenemos dos opciones: o bien usar *cookieStore.get()* si lo que queremos es el primer resultado de la consulta o bien *getAll()* que nos devuelve todo:

```
try {
    const cookie = await cookieStore.get('session_id');
    if (cookie) {
        console.log(`Found ${cookie.name} cookie: ${cookie.value}`);
    } else {
        console.log('Cookie not found');
    }
} catch (error) {
    console.error(`Cookie store error: ${error}`);
}
```

Modificar/escribir una cookie:

```
try {
    await cookieStore.set('opted_out', '1');
} catch (error) {
    console.error(`Failed to set cookie: ${error}`);
}
```

Borrar una cookie

```
try {
    await cookieStore.delete('session_id');
} catch (error) {
    console.error(`Failed to delete cookie: ${error}`);
}
```

Monitorizar cookies:

```
cookieStore.addEventListener('change', event => {

    console.log(`${event.changed.length} changed cookies`);

    for (const cookie in event.changed)
        console.log(`Cookie ${cookie.name} changed to ${cookie.value}`);

    for (const cookie in event.deleted)
        console.log(`Cookie ${cookie.name} deleted`);

});
```

Borrar cookies:

```
try {
    await cookieStore.delete('session_id');
} catch (error) {
    console.error(`Failed to delete cookie: ${error}`);
}
```

Monitorizar cookies:

```
cookieStore.addEventListener('change', event => {

    console.log(`${event.changed.length} changed cookies`);

    for (const cookie in event.changed)
        console.log(`Cookie ${cookie.name} changed to ${cookie.value}`);

    for (const cookie in event.deleted)
        console.log(`Cookie ${cookie.name} deleted`);

});
```

6. Bibliografía

- **¿Qué se entiende por asincronía?**

Lifeder. (2 de febrero de 2021). Comunicación asíncrona o asincrónica. Recuperado de: https://www.lifeder.com/comu

Ayala, M. (2021, febrero 1). Comunicación asíncrona: características, tipos, ventajas, ejemplos. Lifeder. https://www.lifeder.com/comunicacion-asincronica/

(s/f). Openwebinars.net. Recuperado el 18 de enero de 2023, de https://openwebinars.net/academia/aprende/javascript-avanzado/10774/

- **Áreas de aplicación: Comunicación asíncrona en la nube y GitHub**

Cloudflare Workers documentation. (s/f). Cloudflare.com. Recuperado el 22 de enero de 2023, de https://developers.cloudflare.com/workers/

(S/f). Google.com. Recuperado el 22 de enero de 2023, de https://cloud.google.com/solutions/asynchronous-messaging

(S/f-b). Microsoft.com. Recuperado el 22 de enero de 2023, de https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/asynchronous-messaging

Book. (s/f). Git-scm.com. Recuperado el 22 de enero de 2023, de https://git-scm.com/book/es/v2/Fundamentos-de-Git-Trabajando-con-ramas

Book. (s/f). Git-scm.com. Recuperado el 22 de enero de 2023, de https://git-scm.com/book/es/v2/Fundamentos-de-Git-Trabajando-con-ramas

Book. (s/f). Git-scm.com. Recuperado el 22 de enero de 2023, de https://git-scm.com/book/es/v2/Fundamentos-de-Git-Trabajando-con-ramas

- **Áreas de aplicación: Comunicación asíncrona en WhatsApp, Correos y JavaScript**

Academy, C. I., & Castillo, A. A. (2017). Curso de Programación Web: JavaScript, Ajax y jQuery. 2a Edición (2.). Createspace Independent Publishing Platform.

(S/f). Github.io. Recuperado el 18 de enero de 2023, de http://asanzdiego.github.io/curso-javascript-avanzado-2015/slides/export/javascript-avanzado-reveal-slides.pdf

Quintero, I. L. (2023). Node.js - Javascript En El Lado Del Servidor - Manual Practico Avanzado. Alfaomega Grupo Editor.

(S/f-b). Profile.es. Recuperado el 18 de enero de 2023, de https://profile.es/blog/consumir-apt/i-javascript

de Zúñiga, F. G. (2019, abril 5). Axios Javascript: analizamos las características de este ligero cliente HTTP. Blog de arsys.es; Arsys Internet. https://www.arsys.es/blog/axios

Librería Axios: cliente HTTP para Javascript. (s/f). Desarrolloweb.com. Recuperado el 18 de enero de 2023, de https://desarrolloweb.com/articulos/axios-ajax-cliente-http-javascript.html

Axios. (s/f). Axios-http.com. Recuperado el 18 de enero de 2023, de https://axios-http.com/es/

Uso de Fetch. (s/f). Mozilla.org. Recuperado el 18 de enero de 2023, de

- **Concurrencia en JavaScript**

Calzado, J. (2018, enero 31). Javascript asíncrono: La guía definitiva —. Lemoncode Formacion. https://lemoncode.net/lemoncode-blog/2018/1/29/javascript-asincrono

- **El almacenamiento Web**

Usando la API de almacenamiento web. (s/f). Mozilla.org. Recuperado el 24 de enero de 2023, de https://developer.mozilla.org/es/docs/Web/API/Web_Storage_API/Using_the_Web_Storage_API