Programación asíncrona, comunicación con el servidor y almacenamiento

Contenido

Programación asíncrona, comunicación con el servidor y almacenamiento 1
- Programación asíncrona 2
  - Callback Hell 3
- Promesas (Promise) 3
  - Sincronización entre promesas 4
- Async y await 4
- Ejemplo programación asíncrona 5
  - Ejemplo división con callback 5
  - Ejemplo división con promesa 6
  - Ejemplo división con async/await 7
- Manejo de errores 7
- Comunicación con el servidor 10
  - Fetch 11
  - Obtener información 12
  - Enviar información 13
  - Objetos Reques y Response 14
- WebSocket 15
- Almacenamiento en el navegador 16
  - LocalStorage y SessionStorage 17
  - Cookies 18
- Anexo APIS de prueba 20
- Bibliografía 20

Programación asíncrona

JavaScript es un lenguaje de ejecución **mono hilo**, lo que significa que solo puede ejecutar una tarea a la vez en su hilo principal. Sin embargo, para manejar tareas que requieren tiempo, como eventos o solicitudes a un servidor o temporizadores, JavaScript utiliza un modelo asincrónico conocido como "**loop de eventos**" (***event loop***). Estas funciones más demandantes son asociadas a mensajes y estos se encolan, cuando un mensaje se procesa no se interrumpe hasta su conclusión. Debes recordar que en JavaScript nunca se interrumpe a una tarea en ejecución como si sucede en otros lenguajes.

NOTA: Podemos tener multihilo real con **Web Workers**, en este caso creamos otros hilos de ejecución independientes del hilo principal permitiendo la comunicación entre los distintos hilos a través de mensajes con el método "***postMessage***".

Por ejemplo, un ***callback*** que se ejecuta después de concluir el código principal.

```
console.log("Inicio");

setTimeout(() => {
    console.log("Esto se ejecuta después de 2 segundos");
}, 2000);

console.log("Fin");
```

Que genera la siguiente salida.

```
Inicio
Fin
Esto se ejecuta después de 2 segundos
```

En el siguiente ejemplo vamos a comprobar como JavaScript es mono hilo y que no se interrumpe el código en ejecución.

```
console.log('Inicio');

function esperaActiva(segundos) {
    console.log(`Inicio de espera activa: ${new
Date().toLocaleTimeString()}`);
    const start = Date.now();
    while (Date.now() - start < segundos * 1000) {
        // Espera activa
    }
    console.log(`Fin de espera activa: ${new
Date().toLocaleTimeString()}`);
}

console.log('Programamos un mensaje para dentro de 2 segundos')
setTimeout(()=>{
    console.log('Mensaje temporizado a los 2 segundos');
```

```
}, 2000);

console.log('Iniciamos un proceso largo');
esperaActiva(5);

console.log('Fin');
```

El código genera la siguiente salida.

```
Programamos un mensaje para dentro de 2 segundos
Iniciamos un proceso largo - 5 segundos
Inicio de espera activa: 19:22:05
Fin de espera activa: 19:22:10
Fin
Mensaje temporizado a los 2 segundos
```

Fíjate como el temporizador ha tenido que esperar hasta que la función de espera activa a terminado. Si la espera se prolonga demasiado el navegador nos muestra un mensaje para detener el script o para continuar su ejecución.

TRUCO: "setTimeout" con 0 segundos.

El uso de "***setTimeout***" con un tiempo de 0 milisegundos (***setTimeout(callback, 0)***) no ejecuta el código de inmediato, sino que coloca la función en la cola de tareas del "***event loop***" para ejecutarse después de que el hilo principal termine de procesar las operaciones actuales. Esto es especialmente útil para tareas no críticas o para ceder el control del hilo a otras operaciones.

Callback Hell

El **callback hell** (infierno de callbacks) se refiere a un problema común en programación asíncrona, especialmente en JavaScript, cuando se anidan demasiadas funciones callback, creando un código que es difícil de leer, mantener y depurar.

Para mitigar el problema del anidamiento de callbacks se han introducido en el lenguaje nuevas herramientas, en primer lugar "***promises***" y después "***await***" y "***async***".

Promesas (Promise)

Las promesas se introducen en el estándar ES2015 y permiten realizar de manera asíncrona una tarea gestionan su estado. Los posibles estados para una promesa son:

- **Pendiente (pending)**: La operación aún no se ha completado.
- **Resuelta (fulfilled)**: La operación se completó con éxito.
- **Rechazada (rejected)**: La operación falló.

La sintaxis de una promesa pasa por instanciar un objeto de tipo "**Promise**", el cual recibe dos manejadores de **callback**. El primero para cuando el código es exitoso (**resolve**) y el segundo, opcional, para cuando se produce un fallo (**reject**).

```
// Definimos
const promesa = new Promise((resuelve, rechazo) => {
    const exito = true; // Código de la tarea (DEBERÍA SER ASINCRONO)
    // Gestión del resultado con callbacks
    if (exito) {
        resuelve("La operación fue exitosa");
    } else {
        rechazo("Hubo un error");
    }
});

// Para usarla
promesa.then(
    (resultado) => {
        console.log(resultado); // "La operación fue exitosa"
    })
    .catch((error) => {
        console.error(error); // "Hubo un error"
    })
    .finally(() => {
        console.log("Siempre se ejecuta");
    });
```

NOTA: La gestión de errores y el código final son opcionales, en este caso se han desarrollado para mostrar la sintaxis completa.

Sincronización entre promesas

Disponemos de métodos adicionales para sincronizar tareas.

- **Promise.resolve('Mensaje de éxito')**: crea una promesa exitosa.
- **Promise.reject('Mensaje de fracaso')**: crea una promesa fallida.
- **Promise.all([promesa1, promesa2, promesa3,...])**: ejecuta todas las promesas en paralelo y cuando todas terminan exitosamente devuelve un array con todos los callback de éxito ".then(array_resuelve)". En caso de error para con el primer callback de rechazo ".catch(rechazo)". Este método es importante porque permite sincronizar múltiples tareas asíncronas.

Async y await

Las palabras reservadas "***async***" y "***await***" son una forma más moderna y legible de manejar promesas, introducida en ES2017. Permite escribir código asincrónico como si fuera sincrónico, facilitando la lectura y el mantenimiento.

El punto de partida es la declaración de una función con "***async***", lo cual convierte automáticamente a la función en una promesa. Este tipo de funciones se conocen como "***AsyncFunction***". Lo interesante de este tipo de funciones es que pueden combinarse con **"*await*"**. El "***await***" antes de la llamada a una función "*async*" o una promesa hace que el código espere hasta que el código asíncrono se resuelva o se rechace, con lo que conseguimos un código secuencial legible.

RECUERDA: una función que internamente emplee "***await***" debe definirse como "***async***".

Por ejemplo.

```
async function obtenerUsuario() {
    try {
        const usuario = await obtenerDatos(); // Espera la resolución de
la promesa
        console.log("Usuario obtenido:", usuario);
    } catch (error) {
        console.error("Error:", error);
    } finally {
        console.log("Finalizando operación");
    }
}

obtenerUsuario();
```

El código equivalente con promesas sería.

```
obtenerDatos()
    .then((datos) => console.log("Datos:", datos))
    .catch((error) => console.error("Error:", error));
```

Ejemplo programación asíncrona

Vamos a probar las tres maneras comentadas previamente disponibles para definir código asíncrono. En todos los ejemplos vamos a implementar la misma lógica en la que dividiremos 2 números, en el caso de encontrarse un divisor igual a 0 se produce un error.

Soy consciente que el código se puede refactorizar, pero se ha optado por una versión extendida por claridad.

NOTA: es buena práctica de codificación añadir el sufijo "***Async***" a las funciones con código asíncrono.

Ejemplo división con callback

```
/* Con callbacks */
function dividirCallbackAsync(dividendo, divisor, callback) {
    let resultado;
    if (divisor !== 0) {
```

```
        resultado = dividendo / divisor;
    }

    // Metemos la respuesta asincrona
    setTimeout(() => {
        if (resultado !== undefined) {
            callback(resultado, null);
        } else {
            callback(null, new Error('No se puede dividir por cero'));
        }
    }, 0);
}

dividirCallbackAsync(10, 5, (resuelve, rechaza) => {
    if (resuelve) {
        console.log('Resultado: ', resuelve);
    } else {
        console.log('Error: ', rechaza.message);
    }
});
```

Ejemplo división con promesa

```
/* Con promises */
function dividirPromiseAsync(dividendo, divisor) {
    return new Promise((resuelve, rechaza) => {
        let resultado;
        if (divisor !== 0) {
            resultado = dividendo / divisor;
        }

        // Metemos la respuesta asíncrona
        setTimeout(() => {
            if (resultado !== undefined) {
                resuelve(resultado);
            } else {
                rechaza(new Error('No se puede dividir por cero'));
            }
        }, 0);
    });
}

dividirPromiseAsync(10, 5)
    .then(resultado => console.log('Resultado: ', resultado))
    .catch(error => console.log('Error: ', error.message));
```

Ejemplo división con async/await

```
/* Con async/await */
async function dividirAsync(dividendo, divisor) {
    return new Promise((resuelve, rechaza) => {
        let resultado;
        if (divisor !== 0) {
            resultado = dividendo / divisor;
        }

        // Metemos la respuesta asíncrona
        setTimeout(() => {
            if (resultado !== undefined) {
                resuelve(resultado);
            } else {
                rechaza(new Error('No se puede dividir por cero'));
            }
        }, 0);
    });
}

/* Puedo convertir una función sin código asíncrono en asíncrona */
async function dividir(dividendo, divisor) {
    if (divisor === 0) {
        return new Error('No se puede dividir por cero');
    }
    return dividendo / divisor;
}

async function ejecutarDivisionAsync() {
    try {
        const resultado = await dividirAsync(10, 5);
        //const resultado = await dividir(10, 5);
        console.log('Resultado: ', resultado)
    } catch (error) {
        console.log('Error: ', error.message);
    }

}

ejecutarDivisionAsync();
```

Manejo de errores

El lenguaje JavaScript dispone de la clase "***Error***" para gestionar las excepciones de manera similar a otros lenguajes. Las excepciones nos permiten gestionar errores inicialmente no esperados y no debemos emplearlos para codificar lógica condicional.

Try-Catch-Finally

La estructura "***try-catch***" permite ejecutar un bloque de código y capturar errores si ocurren. El bloque opcional "***finally***" se ejecuta siempre, independientemente de si hubo errores o no.

```
try {
    // Código que puede lanzar un error
} catch (error) {
    // Código para manejar un error
} finally {
    // Código final (opcional)
}
```

Si quisiéramos gestionar distintos tipos de error debemos emplear "***instanceof***" o la propiedad "error.name" en el catch ya que este es único.

En el caso de las **promesas** el código es un poco distinto pero similar, **"*.catch()*"** para gestionar el error y **"*.finally()*"** para el código final.

Objeto Error

Los objetos "**Error**" se lanzan cuando ocurren errores en tiempo de ejecución. También puedes utilizar el objeto "**Error**" como objeto base para excepciones definidas por el usuario.

Propiedades

- **message**: Descripción del error.
- **name**: Tipo de error (TypeError, SyntaxError, etc.).
- **stack**: Pila de llamadas, CUIDADO, no está estandarizada.

Por ejemplo.

```
try {
    let resultado = 10 / 0; // Operación válida, pero ilógica
    console.log(resultado);

    JSON.parse("{"); // Error de sintaxis
} catch (error) {
    console.log(`Se capturó un error ${error.name} : ${error.message}`);
    console.log('Pila de llamadas: ', error.stack);
} finally {
    console.log("Este bloque siempre se ejecuta.");
}
```

Que genera la siguiente salida.

```
Infinity                                                                    01-propiedades.js:3
Se capturó un error SyntaxError : Expected property name or '}' in JSON at position  01-propiedades.js:7
1 (line 1 column 2)
Pila de llamadas:  SyntaxError: Expected property name or '}' in JSON at position 1  01-propiedades.js:8
(line 1 column 2)
    at JSON.parse (<anonymous>)
    at http://127.0.0.1:5500/08-errores/01-propiedades.js:5:10
Este bloque siempre se ejecuta.                                             01-propiedades.js:10
```

Errores personalizados

Podemos lanzar errores personalizados con el operador "***throw***". Esto permite adaptarse a necesidades específicas de nuestra aplicación. En este caso instanciamos el nuevo error con "***new***" pasándole el mensaje que describe el problema y después lo lanzamos con "***throw***".

Por ejemplo.

```
function validarEdad(edad) {
    if (edad < 0) {
        throw new Error("La edad no puede ser negativa.");
    }
    return "Edad válida";
}

try {
    console.log(validarEdad(-5));
} catch (error) {
    console.error("Error:", error.message);
}
```

Que genera la siguiente salida.

```
▶ Error: La edad no puede ser negativa.
>
```

Definir nuevos tipos de error

Podemos extender la clase "**Error**" para definir nuestras propias clases de errores. Es importante inicializar la propiedad "**name**" e invocar al constructor de la clase padre.

Por ejemplo.

```
class PersonalizadoError extends Error {
    constructor(mensaje) {
        super(mensaje);
        this.name = this.constructor.name;
    }
}
```

```
try {
    throw new PersonalizadoError("Esto es un error personalizado");
} catch (error) {
    console.log(error.name); // PersonalizadoError
    console.log(error.message); // Esto es un error personalizado
    console.log(error.stack); // Muestra el stack trace
}
```

Que genera la siguiente salida.

```
PersonalizadoError
Esto es un error personalizado
PersonalizadoError: Esto es un error personalizado
    at 03-extender.js:11:11
```

Tipos de error

Además del constructor genérico Error, hay otros siete constructores de errores en el núcleo de JavaScript.

- **RangeError**: Valor fuera del rango permitido.
  - "123".repeat(-1)
- **ReferenceError**: Acceso a una variable no definida.
  - console.log(noDefinida)
- **SyntaxError**: Problema de sintaxis en el código.
  - eval("var x = ;")
- **TypeError**: Operación sobre un tipo de dato incorrecto.
  - null.toString()
- **URIError**: Error en manejo de URIs.
  - decodeURIComponent("%")

Los siguientes tipos no son comunes.

- EvalError: Crea un error que ocurre respecto a la función global "eval()".
- InternalError: Sólo soportado por Firefox, similar a RangeError.

Comunicación con el servidor

La comunicación entre cliente y servidor en las aplicaciones web ha evolucionado significativamente desde los inicios de la web. Originalmente, cualquier interacción con el servidor, como enviar un formulario o cargar datos, requería recargar toda la página, lo que resultaba en una experiencia de usuario lenta y poco dinámica.

El surgimiento de AJAX

En los años 2000, se introdujo una técnica revolucionaria llamada **AJAX** (***Asynchronous JavaScript and XML***). Esta técnica permitía enviar y recibir datos del servidor sin recargar

toda la página, utilizando el objeto **XMLHttpRequest** (XHR). Con AJAX, se lograron experiencias más interactivas, como la carga parcial de contenido.

En este periodo, tecnologías como **OWA** (Outlook Web Access) de Microsoft fueron pioneras en la implementación de aplicaciones web dinámicas, mostrando el potencial de estas técnicas.

Sin embargo, trabajar con ***XMLHttpRequest*** tenía sus complicaciones: sintaxis complejas, dependencias entre callbacks, sin soporte nativo para JSON…

Evolución hacia fetch

En el 2015 se introdujo la API "***fetch***" que modernizo la comunicación cliente-servidor. El código de las peticiones con fetch se simplifico muchísimo dejándolo en unas pocas líneas, se basa en promesas y añade soporte para el formato JSON.

Además, contamos con "***WebSocket***" para comunicaciones full-duples persistentes entre un cliente y el servidor.

Fetch

La API "***fetch***" es la herramienta moderna para realizar solicitudes HTTP en JavaScript. Fue introducida en ES2015 (ECMAScript 6) para reemplazar al antiguo **XMLHttpRequest**. Con "***fetch***" podemos obtener información del servidor (***GET***), enviar información (***POST***) y otras operaciones menos comunes como actualizar un recurso (***PUT***), actualizar parcialmente un recurso (***PATCH***) y eliminar un recurso (***DELETE***).

Sintaxis básica

La API "***fetch***" utiliza promesas, lo que permite manejar las solicitudes de forma asíncrona sin necesidad de callbacks complejos. Devuelve una promesa que se resuelve con un objeto "***Response***", representando la respuesta del servidor. Sin embargo, incluso si la solicitud falla (por ejemplo, un error 404), la promesa no se rechaza automáticamente; se considera exitosa a menos que haya un problema de red o se produzca un error en el código. Por ello, es importante verificar explícitamente la propiedad "***response.ok***".

```
fetch(url, [opciones])
    .then((response) => {
        // Gestionamos los errores reportados por el servidor
        // Es decir, hay comunicación correcta y el servidor
        // devuelve un mensaje de error
        if (!response.ok) {
            throw new Error(`Error HTTP: ${response.status}`);
        }
        // Manejar la respuesta del servidor
    })
    .catch((error) => {
        // Manejar errores de red
```

```
});
```

Obtener información

NOTA: el método por defecto de una petición es ***GET***, por lo que no lo indicamos.

Por ejemplo. **Obtener información** con la sintaxis de promesa.

```
/* Petición GET con then y catch */
fetch("https://jsonplaceholder.typicode.com/posts/1")
    .then((response) => {
        if (!response.ok) {
            // Este nuevo error no esta asociado al catch
            throw new Error(`Error HTTP: ${response.status}`);
        }
        //console.log(response);
        return response.json(); // Convertimos la respuesta en JSON
    })
    .then((data) => {
        console.log("Datos recibidos:", data);
    })
    .catch((error) => {
        console.error("Error al realizar la solicitud:", error.message);
    });
```

Y el mismo código con una sintaxis más genérica en el que sincronizamos las promesas.

```
/* Petición GET con async/await y try-catch */
async function obtenerDatos() {
    try {
        // Sin await response es una promesa
        const response = await fetch(
"https://jsonplaceholder.typicode.com/posts/1");
        if (!response.ok) {
            throw new Error(`Error HTTP: ${response.status}`);
        }
        // Sin await data es una promesa
        const data = await response.json();
        console.log("Datos recibidos:", data);
    } catch (error) {
        console.error("Error al realizar la solicitud:", error.message);
    }
}

obtenerDatos();
```

Enviar información

El método ***POST*** se utiliza para enviar datos al servidor. En este caso, usamos el método "***JSON.stringify***" para convertir los datos en una cadena ***JSON***. En este caso empleamos el segundo parámetro del método "***fetch***" para modificar la petición.

```
/* Petición POST */
const peticion = fetch("https://jsonplaceholder.typicode.com/posts", {
    method: "POST",
    headers: {
        "Content-Type": "application/json",
    },
    body: JSON.stringify({
        title: "Nuevo post",
        body: "Contenido del post",
        userId: 1,
    }),
});

/* OJO. Este servicio devuelve lo que se le envía */
peticion.then((response) => response.json())
    .then((data) => {
        console.log("Datos enviados correctamente:", data);
    })
    .catch((error) => {
        console.error("Error al enviar los datos:", error.message);
    });
```

Envío de formulario (FormData)

Al trabajar con formularios "***FormData***" nos facilita la creación del cuerpo del mensaje. Habitualmente vamos a instanciar el objeto a partir de un formulario, "***new FormData(document.forms[0])***". Otra manera de trabajar con "***FormData***" es como array de pares de clave valor.

RECUERDA: con "***FormData***" el "***Content-Type***" del cuerpo se codifica con "***multipart/form-data***". Al emplear "FormData" se asigna automáticamente el "Content-Type".

NOTA: el tipo de codificación empleado en la petición no tiene porque ser el mismo tipo de codificación empleado en por la respuesta. Depende de la configuración del servidor.

```
/* Petición POST con FormData */
const formData = new FormData();
formData.append("username", "Juan");
formData.append("age", 30);

const peticionFormulario =
fetch("https://jsonplaceholder.typicode.com/posts", {
```

```
    method: "POST",
    body: formData,
});

/* Respuesta, depende del código del servidor */
peticionFormulario.then((response) => response.json())
    .then((data) => {
        console.log("Datos enviados correctamente:", data);
    })
    .catch((error) => {
        console.error("Error al enviar los datos:", error.message);
    });
```

Objetos Reques y Response

Request

Representa una solicitud HTTP, encapsulando información como la URL, el método y los datos enviados.

- "**url**": Dirección del recurso solicitado.
- "**method**": Método HTTP (GET, POST, etc.).
- "**headers**": Encabezados asociados a la solicitud.
- "**body**": Contenido enviado en la solicitud (en métodos como POST o PUT).

Response

Representa la respuesta del servidor, incluyendo el estado, los datos y los encabezados recibidos.

- "**status**": Código de estado HTTP (200, 404, etc.).
- "**statusText**": Mensaje asociado al estado (OK, Not Found).
- "**ok**": Indica si la solicitud fue exitosa (true para códigos 200–299).
- "**headers**": Encabezados incluidos en la respuesta.
- "**body**": Datos devueltos por el servidor, que pueden procesarse (JSON, texto, etc.).

`Headers`

El objeto "***Headers***" es una clase que representa una estructura de clave-valor utilizada para gestionar los encabezados HTTP en solicitudes y respuestas. Permite agregar, leer, modificar y eliminar encabezados de forma sencilla. Se emplea tanto en las peticiones como en las respuestas.

Valores habituales.

- "**Content-Type**": Especifica el tipo de contenido (application/json, text/html, etc.).
- "**Authorization**": Usado para autenticación (Bearer <token>, Basic <token>, Apikey <token>, etc...).

- “**Accept**”: Define los formatos que el cliente puede manejar (application/json).
- “**User-Agent**”: Identifica el cliente que realiza la solicitud (por ejemplo, navegador o aplicación).
- “**Cache-Control**”: Controla la caché (no-cache, max-age=3600).
- “**Cookie**”: Transfiere cookies del cliente al servidor.
- “**Referer**”: Indica la página que originó la solicitud.
- “**Origin**”: Identifica el dominio de origen para solicitudes de recursos.

WebSocket

“WebSocket” es una API que permite establecer una comunicación bidireccional en tiempo real entre el cliente y el servidor mediante un único canal persistente.

NOTA: es mucho más eficiente que las técnicas tradicionales de “***polling***” consistente es preguntar periódicamente al servidor para simular una conexión bidireccional.

Para poder usar ***WebSocket*** necesitamos un servidor que admita este tipo de conexiones. Las URL comienzan por “***ws://***” o por “***wss://***” si la conexión es segura.

Eventos

- “**open**”: Se dispara cuando la conexión se abre.
- “**message**”: Captura los mensajes recibidos del servidor.
- “**error**”: Se dispara cuando ocurre un error en la conexión.
- “**close**”: Se dispara cuando la conexión se cierra.

Métodos

- “**send(data)**” para enviar datos al servidor.
- “**close()**” para cerrar la conexión de forma manual.

Para crear una nueva conexión instanciamos un objeto ***WebSocket*** pasándole como parámetro la URL del servicio.

Por ejemplo. Conexión de un cliente a un servicio ***WebSocket*** de pruebas.

```
// Crear una conexión WebSocket
const socket = new WebSocket("wss://echo.websocket.org");

// Evento: Conexión abierta
socket.addEventListener("open", () => {
    console.log("Conexión establecida con el servidor.");

    // Enviar un mensaje al servidor
    socket.send("¡Hola, servidor!");
});

// Evento: Mensaje recibido
socket.addEventListener("message", (event) => {
```

```
    console.log("Mensaje recibido del servidor:", event.data);
});

// Evento: Error
socket.addEventListener("error", (event) => {
    console.error("Error en la conexión:", event);
});

// Evento: Conexión cerrada
socket.addEventListener("close", () => {
    console.log("Conexión cerrada.");
});

// Cerramos la conexión
setTimeout(() => {
    socket.close();
}, 5000);
```

Que genera la siguiente salida.

```
Conexión establecida con el servidor.
Mensaje recibido del servidor: Request served by 7811941c69e658
Mensaje recibido del servidor: ¡Hola, servidor!
Conexión cerrada.
```

La codificación del lado servidor es análoga al cliente, pero ya depende del lenguaje elegido. Desarrollar la parte servidor va más allá del contenido de este manual.

Almacenamiento en el navegador

El almacenamiento en el navegador permite guardar datos directamente en el cliente, mejorando la experiencia del usuario al evitar solicitudes constantes al servidor. Las opciones principales son "***localStorage***", "***sessionStorage***" y "***Cookies***".

- **localStorage**: Almacenamiento persistente que permanece disponible incluso después de cerrar el navegador.
- **sessionStorage**: Almacenamiento temporal que se borra cuando se cierra la pestaña o ventana del navegador.
- **Cookies**: Datos pequeños enviados con cada solicitud HTTP, útiles para autenticación o personalización del usuario.

Limitaciones generales.

- ***localStorage*** y ***sessionStorage***: Generalmente tienen un límite de entre 5MB y 10MB por dominio (varía según el navegador).
- ***Cookies***: Limitadas a 4KB por cookie y se envían automáticamente con cada solicitud al servidor, lo que puede afectar el rendimiento.

LocalStorage y SessionStorage

El uso de ambas APIs es análogo por lo que vamos desarrollarlas de manera conjunta. Ambas APIs almacenan pares de clave-valor.

En el caso de "***localStorage***" los datos no tienen fecha de expiración y permanecen disponibles incluso después de cerrar y reabrir el navegador.

En contra, en el caso de "***sessionStorage***" los datos sólo están disponibles durante la sesión del navegador, es decir, se eliminan automáticamente al cerrar la pestaña o la ventana del navegador.

Métodos principales (compartidos)

- **setItem(key, value)**: Almacena un valor bajo una clave.
- **getItem(key)**: Recupera el valor asociado a una clave.
- **removeItem(key)**: Elimina una clave y su valor asociado.
- **clear()**: Elimina todos los datos almacenados.

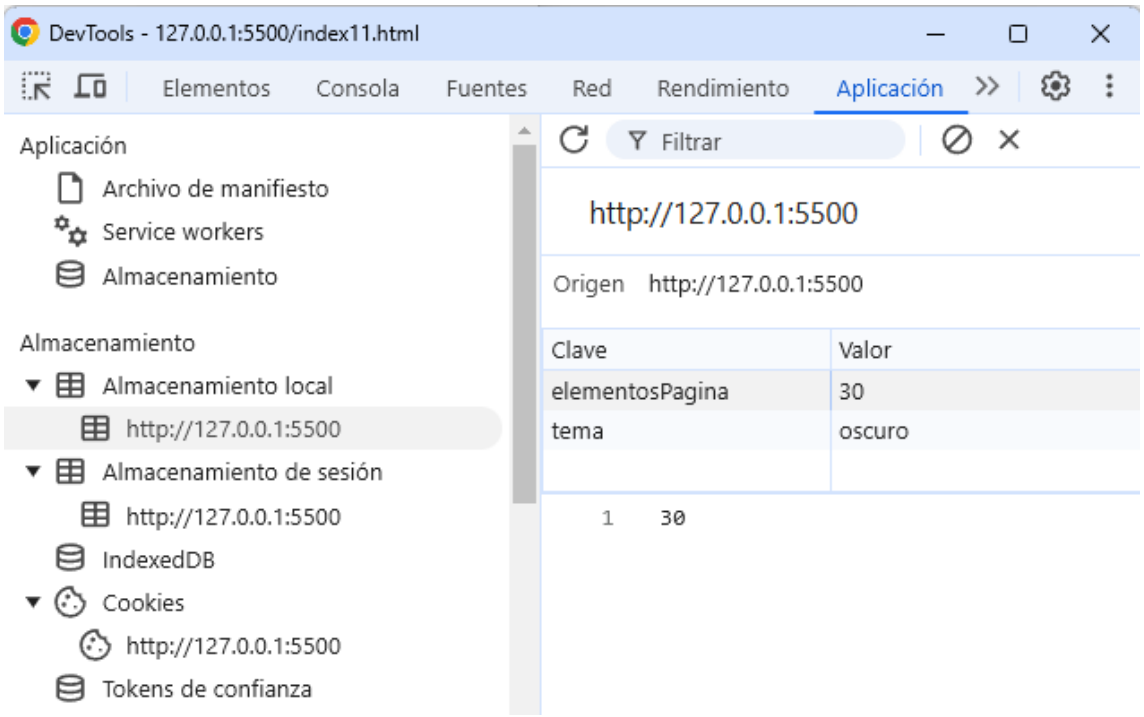
En ambos casos sólo se admiten datos en formato cadena lo que si se desea almacenar un objeto deberá serializarse con **JSON.stringfy**.

Por ejemplo.

```
/* Datos persistentes */
// Guardar datos
localStorage.setItem("tema", "oscuro");
localStorage.setItem("elementosPagina", "30");

// Recuperar datos
console.log(localStorage.getItem("tema")); // "oscuro"

// Eliminar un dato
localStorage.removeItem("elementosPagina");

// Limpiar todo el almacenamiento
localStorage.clear();

/* Datos temporales de sesión */
// Guardar datos temporales
sessionStorage.setItem("carrito", "la wii station, el supermario world,
aero fighter II");

// Recuperar datos
console.log(sessionStorage.getItem("carrito"));
```

Cuando emplear uno u otro dependerá del caso de uso. El almacenamiento persistente es útil para valores de configuración o preferencias del usuario. El almacenamiento de sesión es útil para carros de la compra o para controlar la navegación entre páginas.

En las herramientas de desarrollador podemos ver el registro de los datos para el almacenamiento local, el almacenamiento de sesión y las cookies.



Cookies

Las cookies son pequeños fragmentos de datos que el navegador almacena y envía automáticamente al servidor con cada solicitud HTTP. **Son útiles para autenticación, personalización y seguimiento del usuario**.

NOTA: Actualmente deberíamos emplearlo únicamente para autenticación...

Cuando hay varias cookies almacenadas en el mismo dominio, se concatenan en una única cadena separada por punto y coma (**;**) y se envían juntas en el encabezado Cookie de la solicitud HTTP.

Formato cadena con múltiples cookies

"clave1=valor1; clave2=valor2; clave3=valor3"

RECUERDA: las cookies son cadenas que se envían en cada petición automáticamente, su uso ideal es almacenar las credenciales de seguridad de la sesión. También se usan para almacenar preferencias de usuario y para rastrear al usuario (seguimiento web) por lo que te recomiendo que hagas el mínimo uso de ellas.

Por ejemplo.

```
/* Definimos 3 cookies nombre=valor */
document.cookie = "usuario=Juan";
```

```
document.cookie = "tema=oscuro";
document.cookie = "idioma=es";

/* Esta es la cadena con la que trabajamos */
console.log("Agrupadas: ", document.cookie);
```

Que genera la siguiente salida.

```
Agrupadas:  usuario=Juan; tema=oscuro; idioma=es                02-cookies.js:5
```

Propiedades

- **name**: Nombre de la cookie.
- **value**: Valor asociado a la cookie.
- **expires**: Fecha de expiración de la cookie. Sólo formato GMT estándar.
- **max-age**: Tiempo de expiración de la cookie en segundos.
- **path**: Rutas donde la cookie es válida. Es un filtro, si la URL incluye el path la cookie se incluye en el header de la petición.
- **domain**: Dominios y subdominios en los que la cookie es válida. Similar a path pero con nombres de dominio.
- **Secure**: Solo se envía en conexiones HTTPS.
- **SameSite**: Controla el envío en solicitudes de terceros (Strict, Lax).

IMPORTANTE: Cuando definamos una cookie debemos indicar al menos los parámetros opcionales de tiempo de vida (fecha o segundos) y el path. Si no se indica el tiempo de vida la cookie se borra al cerrar la pestaña o el navegador (dura la sesión). Si no se indica el path se establece el de la URL en donde se creó.

NOTA: para eliminar una cookie debemos establecer "***expires***" o "***max-age***" a una fecha anterior o poner una duración de 0 segundos. Además, tenemos que indicar el "***path***" o el "***domain***" ya que la cookie está asociada a estos valores específicos. Asignar cadena vacía a una cookie no la elimina.

Por ejemplo.

```
// Crear una cookie con una duración de 7 días (7 días * 24 horas * 60
minutos * 60 segundos)
document.cookie = `usuario=Juan; max-age=${7 * 24 * 60 * 60}; path=/`;
document.cookie = `tema=oscuro; max-age=${7 * 24 * 60 * 60}; path=/`;

// Mostrar las cookies almacenadas
console.log("Cookies actuales:", document.cookie);
```

```
// Realizar una solicitud de ejemplo
const peticion = fetch("https://jsonplaceholder.typicode.com/posts/1", {
    method: "GET",
    headers: {
        "Content-Type": "application/json",
    },
    credentials: "include", // Enviar cookies al servidor (NO ESTAMOS EN
EL MISMO ORIGEN)
});

/* Respuesta del servidor */
peticion.then((response) => {
    console.log("Respuesta recibida:", response.status);
}).catch((error) => {
    console.error("Error en la solicitud:", error.message);
});
```

Anexo APIS de prueba

JSONPlaceHolder – API falsa permite simular operaciones CRUD

https://jsonplaceholder.typicode.com/

PokeAPI - Pokemons

https://pokeapi.co/

The Rick and Morty API

https://rickandmortyapi.com/

Open Weather Map -> Requiere registro – autenticación

https://openweathermap.org/api

Rest Countries – permite consultas de búsqueda y filtrado.

https://restcountries.com/

The Movie Database -> Requiere registro

https://www.themoviedb.org/

https://api.themoviedb.org/

Bibliografía

Documentación MDN – Promise

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Promise

Documentación MDN – Usando promesas

https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Using_promises

Documentación MDN – async

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

Documentación MDN – await

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Operators/await

Documentación MDN -Usando fetch

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch

Documentación MDN – "event loop"

https://developer.mozilla.org/es/docs/Web/JavaScript/Event_loop

Documentación MDN - Error

https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Error

Documentación MDN – AJAX

https://developer.mozilla.org/es/docs/Learn/JavaScript/Client-side_web_APIs/Fetching_data

Documentación MDN – fetch

https://developer.mozilla.org/es/docs/Web/API/Window/fetch

Documentación MDN – Header

https://developer.mozilla.org/es/docs/Web/API/Headers

Documentación MDN – localStorage

https://developer.mozilla.org/es/docs/Web/API/Window/localStorage

Documentación MDN – sessionStorage

https://developer.mozilla.org/es/docs/Web/API/Window/sessionStorage

Documentación MDN – Cookie

https://developer.mozilla.org/es/docs/Web/HTTP/Cookies

Desarrollo Web en entono cliente – Asincronía y Promesas.

https://xxjcaxx.github.io/libro_dwec/promesas.html

Desarrollo Web en entono cliente – AJAX

https://xxjcaxx.github.io/libro_dwec/ajax.html

JavaScriptInfo – Almacenando datos en el navegador

https://es.javascript.info/data-storage

Cursos de programación web Manz.dev

https://lenguajejs.com/