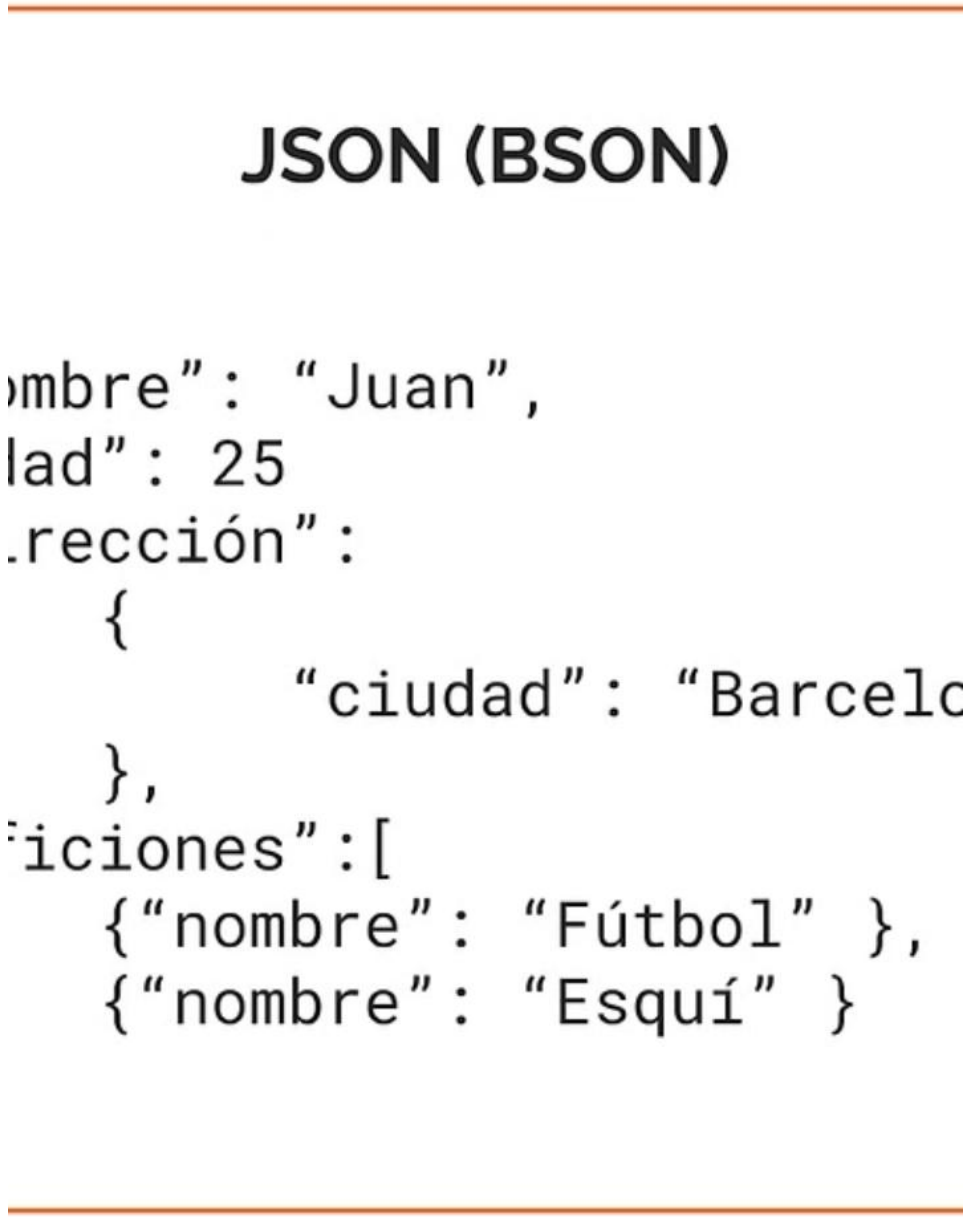
Persistencia local y Conexiones remotas

JSON

¿Qué es JSON

- JSON (JavaScript Object Notation) es un formato de intercambio de datos ligero y fácil de leer y escribir, que se ha convertido en un estándar ampliamente adoptado en el desarrollo web y móvil.
- Permite representar de manera sencilla y eficiente estructuras de datos complejas.
- Llevar los tipos de javascript/typescript a un documento de texto.

https://www.json.org/json-es.html



Persistencia Local

Almacenamiento local: Local Storage

Persistencia de datos

Local Storage permite almacenar datos en el navegador del usuario, lo que permite que la información se mantenga incluso después de cerrar la página.

Acceso rápido

Los datos almacenados en Local Storage están disponibles de manera instantánea, lo que los hace ideales para aplicaciones web que requieren acceso veloz a información.

Seguridad limitada

Si bien Local Storage es más seguro que las cookies, aún tiene ciertas limitaciones de seguridad, por lo que es importante manejar con cuidado la información almacenada.

Versatilidad

Local Storage puede almacenar una amplia variedad de datos, desde preferencias de usuario hasta estados de aplicaciones complejas.

API LocalStorage

- **localStorage.setItem(key, value):** Esta función se utiliza para guardar un par clave-valor en el almacenamiento local. Tanto la clave como el valor deben ser cadenas de texto (string). Si ya existe una clave, su valor será sobrescrito.
  - Ejemplo: localStorage.setItem('nombre', 'Juan’)
- **localStorage.getItem(key):** Se utiliza para recuperar el valor asociado a una clave específica. La función devuelve una cadena de texto. Si la clave no existe, devuelve null.
  - Ejemplo: const nombre = localStorage.getItem('nombre’)
- **localStorage.removeItem(key):** Elimina un par clave-valor del almacenamiento local.
  - Ejemplo: localStorage.removeItem('nombre’)
- **localStorage.clear():** Elimina todos los pares clave-valor del almacenamiento local para el dominio actual.
  - Ejemplo: localStorage.clear()

API LocalStorage – Tomar en cuenta

- **Solo almacena cadenas de texto**: Si necesitas guardar objetos o arrays, primero debes convertirlos a formato JSON usando JSON.stringify() antes de guardarlos. Al recuperarlos, debes usar JSON.parse() para convertirlos de nuevo a su formato original.

```
const usuario = { id: 1, nombre: 'Ana' };
localStorage.setItem('usuario', JSON.stringify(usuario));

const usuarioGuardado =
JSON.parse(localStorage.getItem('usuario'));
console.log(usuarioGuardado.nombre); // 'Ana'
```

- **Almacenamiento por dominio**: El almacenamiento local es específico para el dominio que lo creó. Un sitio web no puede acceder a los datos de localStorage de otro sitio.
- **Seguridad**: localStorage no es un lugar seguro para almacenar información sensible como contraseñas, ya que los datos no están encriptados y pueden ser accedidos fácilmente a través de JavaScript.

Almacenamiento de sesión: Session Storage

- Session Storage es una API de JavaScript que permite almacenar datos de forma temporal en el navegador del usuario. A diferencia de Local Storage, los datos almacenados en Session Storage se eliminan cuando el usuario cierra la pestaña o el navegador.
    1. Session Storage es ideal para almacenar información sensible o temporal que no necesita persistir más allá de la sesión actual.
    2. Los datos almacenados en Session Storage no se envían al servidor con cada solicitud HTTP, lo que mejora el rendimiento de la aplicación.
    3. Session Storage ofrece una mayor privacidad y seguridad que Local Storage, ya que los datos no se comparten entre diferentes pestañas o ventanas del navegador.

Caso de uso: Manejo de sesiones

1

Inicio de Sesión

Los usuarios inician sesión en la aplicación web mediante un proceso de autenticación seguro, que verifica su identidad y les concede acceso a funcionalidades específicas.

2

Almacenamiento de Sesión

La información de la sesión del usuario, como preferencias y estado, se guarda de forma temporal en el servidor o en el dispositivo del usuario mediante cookies o almacenamiento persistente (local o session storage).

3

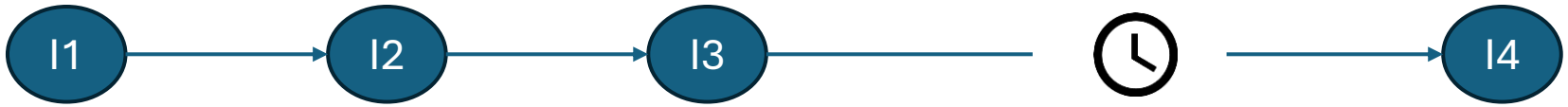
Cierre de Sesión

Cuando el usuario decide finalizar su sesión, se destruye la información de la sesión en el servidor y se elimina cualquier dato almacenado en el dispositivo del usuario.

Promesas y Async / Await

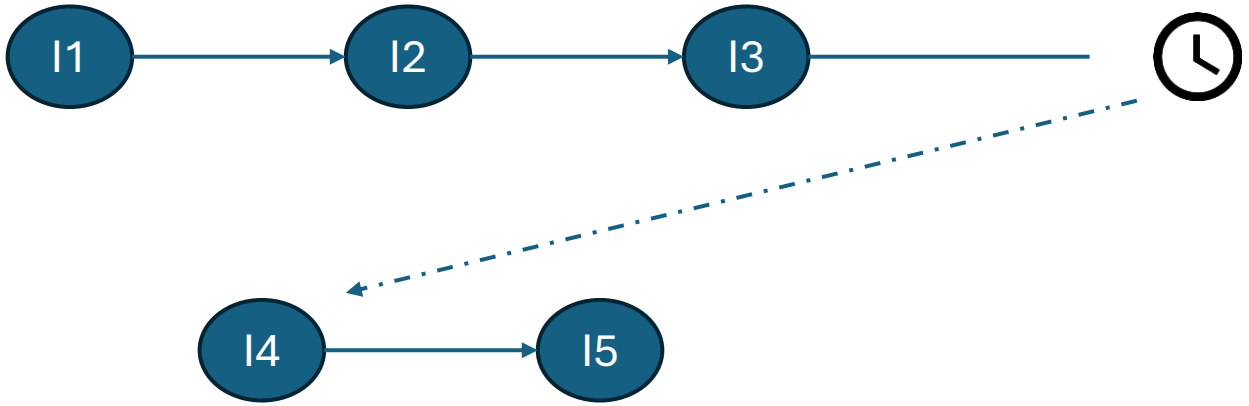
Programación Síncrona

- **Síncrona:** Tareas se ejecutan en orden secuencial. Existe "bloqueo" que sucede cuando una instrucción tarda demasiado (programa congelado).



Programación Asíncrona

- **Asíncrona:** Un programa puede iniciar una tarea que puede tardar un tiempo en completarse y, en lugar de esperar a que termine, continúa ejecutando otras tareas. No hay “bloqueo”.
- Cuando la tarea larga finaliza, se notifica al programa (mediante funciones callbacks) para que continue con si ejecución



Promesas

- Una promesa (promise) es una funcionalidad de javascript que nos permite trabajar con peticiones asíncronas.

- Cuando una función nos retorne un objeto de tipo Promise significa que podemos setear funciones de tipo callback en caso que se realizó correctamente la llamada (then) o si hubo un error (catch).

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Fetch

- La función fetch recibe como argumentos de entrada:
  - url
  - parámetros:
    - method
    - headers
    - body
- Devuelve un Promise.

```
fetch('http://example.com/movies.json’)
.then(response => response.json())
.catch(error => console.error(data));x
```

Async / Await

- **async:** Se utiliza para declarar una función como asíncrona. Una función declarada con async siempre devuelve una Promesa, incluso si el valor de retorno no es una.
  - Si la función retorna un valor, JavaScript lo envuelve automáticamente en una Promesa resuelta.
- **await:** Solo se puede usar dentro de una función async. El operador await "pausa" la ejecución de la función hasta que una Promesa se resuelva o sea rechazada. Una vez que la Promesa se resuelve, await devuelve su valor de resolución.
  - Si la Promesa es rechazada, await lanza una excepción, la cual puede ser capturada con un bloque try...catch.

Promesas vs Async / Await

```
const obtenerDatos = () => {
    fetch('https://api.ejemplo.com/datos')
        .then(response => {
            if (!response.ok) {
                throw new Error('Error al obtener los datos');
            }
            return response.json();
        })
        .then(data => {
            console.log(data);
        })
        .catch(error => {
            console.error('Ha ocurrido un error:', error);
        });
}
```

Promesas vs Async / Await

```
const obtenerDatosAsync = async () =>  {
    try {
        const respuesta = await fetch('https://api.ejemplo.com/datos');

        if (!respuesta.ok) {
            throw new Error('Error al obtener los datos');
        }

        const datos = await respuesta.json();
        console.log(datos);
    } catch (error) {
        console.error('Ha ocurrido un error:', error);
    }
}
```

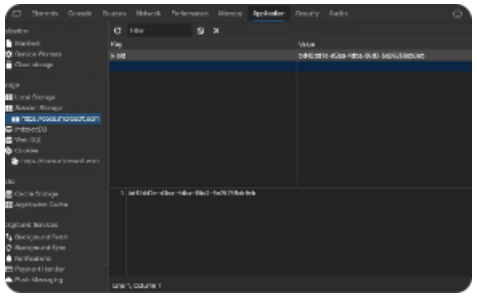
Casos de uso y ejemplos prácticos

JSON en aplicaciones web

El formato JSON se utiliza ampliamente en aplicaciones web para el intercambio de datos, desde API hasta integración de servicios en la nube. Permite una transmisión de datos eficiente y compatible con múltiples lenguajes de programación.

Local Storage para aplicaciones web

El almacenamiento local (Local Storage) permite guardar datos del usuario directamente en el navegador, mejorando la experiencia al evitar cargas constantes desde el servidor. Es útil para recordar preferencias, carritos de compra y otros datos no sensibles.



Session Storage para para aplicaciones web

El almacenamiento de sesión (Session Storage) se utiliza para almacenar datos temporales, como tokens de autenticación o contexto de la aplicación, que se borran cuando el usuario cierra la pestaña o ventana del navegador.

Fetch API para solicitudes HTTP

La Fetch API permite realizar solicitudes HTTP de manera sencilla y elegante, reemplazando a las antiguas llamadas AJAX. Es ampliamente utilizada en aplicaciones web modernas para obtener datos de APIs y servicios en línea.

React - useEffect

- Nos permite definir código que se ejecutará como efecto secundario.
- Un efecto secundario es una acción que ocurre **fuera del flujo** normal de renderizado de React, como la obtención de datos de una API, la manipulación directa del DOM, o la configuración de suscripciones o temporizadores.

1. Definición de la función de efecto

```
import React, { useEffect } from 'react';

function MiComponente() {
  useEffect(() => {
    // Código del efecto secundario aquí
    console.log('¡Este efecto se ejecutó!');
  }); // Se ejecuta después de cada renderizado

  return <h1>Hola, mundo</h1>;
}
```

2. Listado de dependencias

```
useEffect(() => {
  // Se ejecuta solo una vez al montar el componente
  console.log('Componente montado...');
  // fetch('https://api.ejemplo.com/datos')
}, []);
```

El listado de dependencias vacío indica que se ejecutará **solo una vez** después del primer renderizado del componente. Esto es útil para tareas que solo necesitan ser realizadas una vez, como la obtención inicial de datos.

El listado de dependencias con variables indica que se ejecutará cada vez que cualquiera de las variables en el array cambie de valor. Esto es útil para sincronizar el estado del componente con fuentes externas.

```
import React, { useEffect, useState } from 'react';

function MiComponenteConDatos() {
  const [usuarioId, setUsuarioId] = useState(1);

  useEffect(() => {
    // Se ejecuta cuando usuarioId cambia
    console.log(`Obteniendo datos para el usuario
${usuarioId}...`);
    //
fetch(`https://api.ejemplo.com/usuarios/${usuarioId}`)
  }, [usuarioId]); // El efecto se vuelve a ejecutar
si usuarioId cambia

  return <p>Mostrando datos del usuario
{usuarioId}</p>;
}
```

UNIVERSIDAD DE LIMA

3. Función de limpieza

- El useEffect puede devolver una función de limpieza (cleanup function). Esta función es opcional y se ejecuta cuando el componente se desmonta o **antes de que el efecto se vuelva a ejecutar** debido a un cambio en las dependencias.
- Es crucial para limpiar recursos, como temporizadores (clearInterval) o suscripciones, para evitar fugas de memoria.

```
useEffect(() => {
  const temporizador = setInterval(() => {
    console.log('Ejecutando cada 1 segundo');
  }, 1000);

  // La función de limpieza
  return () => {
    clearInterval(temporizador);
    console.log('Temporizador limpiado');
  };
}, []);
```