4- Lenguajes de script

A lo largo de los años, los lenguajes de script han evolucionado significativamente, adaptándose a las necesidades cambiantes de la industria tecnológica. En programación y desarrollo de software, los lenguajes de script juegan un papel fundamental.

Estos lenguajes son herramientas poderosas que permiten a los desarrolladores automatizar tareas, gestionar sistemas y crear aplicaciones web y de escritorio de manera eficiente.

Los lenguajes de script son comúnmente utilizados para tareas como la gestión de sistemas, el procesamiento de texto, la automatización de flujos de trabajo y el desarrollo de aplicaciones web.

A diferencia de los lenguajes de programación compilados, los lenguajes de script son **interpretados**, lo que significa que el código fuente se ejecuta directamente sin necesidad de una fase de compilación previa.

Características principales de los lenguajes de script:

1. **Interpretados**: Los lenguajes de script son interpretados en lugar de compilados. Esto significa que el código se ejecuta directamente línea por línea por un intérprete, lo que facilita la depuración y hace que el ciclo de desarrollo sea más rápido.
2. **Simplicidad y facilidad de uso**: Están diseñados para ser fáciles de aprender y utilizar, con sintaxis simplificada en comparación con lenguajes compilados como C o Java. Esto los hace accesibles tanto para desarrolladores experimentados como para principiantes.
3. **Tipado dinámico**: En muchos lenguajes de script, el tipo de las variables se determina en tiempo de ejecución, lo que proporciona flexibilidad y reduce la necesidad de declaraciones de tipos explícitas.

4. **Portabilidad**: La mayoría de los lenguajes de script son portátiles y pueden ejecutarse en diferentes sistemas operativos sin necesidad de modificaciones en el código fuente.
5. **Bibliotecas y módulos extensos**: Los lenguajes de script suelen tener un amplio conjunto de bibliotecas y módulos disponibles, lo que permite a los desarrolladores reutilizar código existente y acelerar el desarrollo de aplicaciones.
6. **Integración y automatización**: Se utilizan ampliamente para la integración de sistemas y la automatización de tareas repetitivas, como la gestión de archivos, la configuración de sistemas y la ejecución de tareas programadas.
7. **Usos comunes**: Entre los usos más comunes de los lenguajes de script se encuentran la creación de scripts de shell (por ejemplo, Bash), la manipulación de datos (Python, Perl), el desarrollo web (JavaScript, PHP), y la automatización de tareas administrativas (PowerShell).

Ejemplos de lenguajes de script populares:

- **JavaScript**: Utilizado principalmente para el desarrollo web del lado del cliente, pero también del lado del servidor con Node.js.
- **Python**: Conocido por su sintaxis clara y su amplia aplicación en ciencia de datos, desarrollo web, automatización y scripting general.
- **Ruby**: Utilizado en desarrollo web (Ruby on Rails) y scripting general.
- **Perl**: Famoso por su capacidad de procesamiento de texto y administración de sistemas.
- **PHP**: Utilizado principalmente para el desarrollo web del lado del servidor.
- **Bash**: Utilizado para la creación de scripts de shell en sistemas Unix y Linux.

Los lenguajes de script son herramientas esenciales en el arsenal de cualquier desarrollador, proporcionando una forma rápida y eficiente de automatizar tareas y desarrollar aplicaciones. Su simplicidad, flexibilidad y extensibilidad los hacen ideales para una amplia variedad de aplicaciones en el mundo de la tecnología.

4.1- JavaScript

JavaScript es uno de los lenguajes de programación más populares y fundamentales en el desarrollo web moderno. Su versatilidad y capacidad para ejecutarse tanto en el lado del cliente como en el lado del servidor lo han convertido en una herramienta indispensable para los desarrolladores.

Nacido en los años 90, JavaScript ha evolucionado desde ser un simple lenguaje de scripting para añadir interactividad a las páginas web, hasta convertirse en un pilar esencial para la creación de aplicaciones web complejas y dinámicas.

Originalmente desarrollado por Netscape como un medio para **agregar comportamientos dinámicos a los sitios web**, JavaScript es ahora un estándar mantenido por ECMA International bajo la especificación ECMAScript.

JavaScript es un lenguaje de programación interpretado, orientado a objetos y basado en prototipos, utilizado principalmente en el desarrollo web para crear páginas y aplicaciones interactivas.

Características:

1. **Interactividad del lado del cliente**: JavaScript se ejecuta en el navegador del usuario, lo que permite crear experiencias de usuario interactivas y dinámicas sin necesidad de recargar la página completa. Esto incluye la validación de formularios, la creación de efectos visuales y la actualización de contenido en tiempo real.
2. **Orientado a objetos y basado en prototipos**: A diferencia de otros lenguajes orientados a objetos que utilizan clases, JavaScript usa prototipos para la herencia. Esto permite una gran flexibilidad en la creación y manipulación de objetos.
3. **Lenguaje interpretado**: JavaScript es interpretado, lo que significa que el código se ejecuta línea por línea en lugar de ser compilado antes de la ejecución. Esto facilita la depuración y el desarrollo rápido.

4. **Versatilidad y portabilidad**: JavaScript puede ejecutarse en cualquier dispositivo que tenga un navegador web, lo que lo hace extremadamente portátil. Además, con la llegada de entornos de ejecución como Node.js, JavaScript también se utiliza ampliamente en el desarrollo del lado del servidor.
5. **Eventos y manipulación del DOM**: JavaScript es fundamental para la manipulación del Document Object Model (DOM), lo que permite a los desarrolladores cambiar dinámicamente la estructura, el contenido y el estilo de las páginas web en respuesta a eventos del usuario, como clics, movimientos del ratón y entradas de teclado.
6. **Asincronía**: JavaScript soporta operaciones asincrónicas mediante callbacks, promesas y la moderna sintaxis async/await. Esto es esencial para realizar tareas como la carga de datos desde un servidor sin bloquear la ejecución del código.
7. **Amplio ecosistema y bibliotecas**: JavaScript cuenta con una vasta cantidad de bibliotecas y frameworks que facilitan el desarrollo de aplicaciones complejas. Algunos ejemplos populares son React, Angular y Vue.js para el desarrollo front-end, y Express.js para el desarrollo back-end con Node.js.
8. **Compatibilidad con múltiples plataformas**: Gracias a su capacidad de ejecución en navegadores web, servidores (a través de Node.js) y aplicaciones móviles (con frameworks como React Native), JavaScript es un lenguaje verdaderamente multiplataforma.

A continuación, se muestra un pequeño ejemplo de cómo JavaScript puede ser utilizado para cambiar el contenido de una página web en respuesta a una interacción del usuario:

```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo de JavaScript</title>
    <script>
        function mostrarMensaje() {
            alert("¡Hola, este es un mensaje de JavaScript!");
        }
    </script>
</head>
<body>
    <button onclick="mostrarMensaje()">Haz clic aquí</button>
</body>
</html>
```

En este ejemplo, cuando el usuario hace clic en el botón, se ejecuta la función mostrarMensaje que muestra una alerta con un mensaje.

JavaScript es un lenguaje de programación esencial en el desarrollo web moderno, proporcionando las herramientas necesarias para crear aplicaciones interactivas, dinámicas y de alto rendimiento. Su versatilidad, facilidad de uso y amplio ecosistema lo convierten en una elección ideal tanto para desarrolladores novatos como experimentados.

4.2- Elementos del lenguaje JavaScript

JavaScript, como cualquier lenguaje de programación, se compone de varios elementos y conceptos fundamentales que los desarrolladores utilizan para escribir código funcional y eficiente. A continuación se describen los principales elementos del lenguaje de JavaScript:

Variables: Almacenan datos que pueden cambiar durante la ejecución del programa.

```
var nombre = "Juan";
let edad = 30;
const PI = 3.1416;
```

Las variables declaradas con **var** tienen un alcance de función o global. Si se declaran dentro de una función, sólo son accesibles dentro de esa función. Si se declaran fuera de una función, tienen un alcance global mientras que las variables declaradas con **let** tienen un alcance de bloque. Esto significa que solo son accesibles dentro del bloque de código donde se declararon

Tipos de Datos: Incluyen primitivos como `number`, `string`, `boolean`, `null`, `undefined`, y `symbol`.

```
let numero = 10;
let texto = "Hola";
let esVerdad = true;
let valorNulo = null;
let valorIndefinido;
```

Operadores aritméticos: `+`, `-`, `*`, `/`, `%`

```
let suma = 5 + 3;
let resta = 5 - 3;
let multiplicacion = 5 * 3;
let division = 5 / 3;
let modulo = 5 % 3;
```

Operadores de asignación: =, +=, -=, *=, /=, %=

```
let x = 10;
x += 5; // x ahora es 15
```

Operadores de comparación: ==, ===, !=, !==, >, <, >=, <=

```
let esIgual = (5 == '5'); // true
let esEstrictamenteIgual = (5 === '5'); // false
```

El operador **===**, conocido como "igualdad estricta" o "triple igual", es un operador de comparación en JavaScript que verifica tanto el valor como el tipo de los operandos. A diferencia del operador ==, que realiza una conversión de tipo antes de comparar, === compara los operandos sin realizar ningún tipo de conversión, lo que lo hace más estricto y predictivo.

El uso de este operador en JavaScript es esencial para realizar comparaciones de igualdad estricta que tengan en cuenta tanto el valor como el tipo de los operandos. Su uso es una buena práctica que ayuda a evitar errores y hace que el código sea más seguro y fácil de mantener.

Operadores lógicos: `&&`, `||`, `!`

```
let y = true && false; // false
let o = true || false; // true
let no = !true; // false
```

Estructuras de control condicionales: `if`, `else if`, `else`, `switch`

```
if (edad > 18) {
    console.log("Es mayor de edad");
} else {
    console.log("Es menor de edad");
}
```

```
switch (dia) {
    case 1:
        console.log("Lunes");
        break;
    case 2:
        console.log("Martes");
        break;
    default:
        console.log("Otro día");
}
```

Estructuras de Control - Bucles: `for`, `while`, `do...while`

```
for (let i = 0; i < 5; i++) {
    console.log(i);
}

let j = 0;
while (j < 5) {
    console.log(j);
    j++;
}

let k = 0;
do {
    console.log(k);
    k++;
} while (k < 5);
```

Funciones: define bloques de código reutilizables.

```
function saludar(nombre) {
    return "Hola, " + nombre;
}
let saludo = saludar("Pedro");
console.log(saludo);
```

Funciones anónimas y arrow functions: Sintaxis compacta para funciones.

javascript

```
let suma = function(a, b) {
    return a + b;
};

let sumaArrow = (a, b) => a + b;
```

Objetos: Colecciones de pares clave-valor.

```
let persona = {
    nombre: "Juan",
    edad: 30,
    saludar: function() {
        return "Hola, soy " + this.nombre;
    }
};
console.log(persona.saludar());
```

Arrays: Listas ordenadas de elementos.

```
let numeros = [1, 2, 3, 4, 5];
console.log(numeros[0]); // 1
```

Excepciones: Las excepciones son un mecanismo en JavaScript para manejar errores y condiciones excepcionales en el flujo de un programa. Permiten que el programa reaccione a situaciones inesperadas de manera controlada, evitando que falle de forma abrupta. El manejo de excepciones en JavaScript se realiza mediante las estructuras **try, catch, finally** y **throw.+**

La estructura básica para el manejo de excepciones en JavaScript es try...catch. Aquí se intenta ejecutar un bloque de código y, si se produce una excepción, se captura y maneja en el bloque catch.

```
try {
    // Código que puede lanzar una excepción
    let resultado = dividir(10, 0);
    console.log(resultado);
} catch (error) {
    // Manejo de la excepción
    console.error("Se produjo un error:", error.message);
```

```
} finally {
    // Código que se ejecuta siempre, ocurra o no una excepción
    console.log("Operación de división completada.");
}

function dividir(a, b) {
    if (b === 0) {
        throw new Error("No se puede dividir por cero.");
    }
    return a / b;
}
```

Puedes lanzar tus propias excepciones utilizando la palabra clave throw. Esto es útil para validar condiciones específicas y manejar errores personalizados.

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

try {
    validarEdad(-1);
} catch (error) {
    console.error("Error de validación:", error.message);
}
```

El bloque finally contiene código que se ejecuta siempre, independientemente de si se produjo o no una excepción. Es útil para liberar recursos o realizar tareas de limpieza.

```
try {
    // Código que puede lanzar una excepción
    let resultado = dividir(10, 2);
    console.log(resultado);
} catch (error) {
    // Manejo de la excepción
    console.error("Se produjo un error:", error.message);
} finally {
    // Código que se ejecuta siempre
    console.log("Operación de división completada.");
}
```

A continuación se muestra un ejemplo completo que ilustra el uso de try, catch, finally y throw en una función de conversión de JSON.

```
function parsearJSON(jsonString) {
    try {
        let objeto = JSON.parse(jsonString);
        console.log("Conversión exitosa:", objeto);
    } catch (error) {
        console.error("Error al convertir JSON:", error.message);
    } finally {
        console.log("Operación de conversión JSON completada.");
    }
}

let jsonValido = '{"nombre": "Juan", "edad": 30}';
let jsonInvalido = '{"nombre": "Juan", "edad": 30'; // Falta la llave de
cierre

parsearJSON(jsonValido);
parsearJSON(jsonInvalido);
```

El manejo de excepciones en JavaScript es una herramienta poderosa para crear aplicaciones robustas y fiables. Utilizando try, catch, finally y throw, puedes gestionar errores de manera efectiva, asegurar la ejecución de código de limpieza y lanzar excepciones personalizadas cuando sea necesario.

4.3- Programación orientada a objetos en Javascript

La Programación Orientada a Objetos (POO) es un paradigma de programación que se basa en el concepto de "objetos", que pueden contener datos, en forma de campos, a menudo conocidos como atributos o propiedades, y código, en forma de procedimientos, a menudo conocidos como métodos

La POO se centra en el diseño y organización del software en torno a estos objetos, facilitando la **reutilización de código**, la encapsulación y la modularidad.

Conceptos Clave de la POO

1. **Clase**: Una clase es una plantilla o modelo que define las propiedades y comportamientos (métodos) comunes de un conjunto de objetos.
2. **Objeto**: Una instancia de una clase. Un objeto es una entidad que tiene estado (atributos) y comportamiento (métodos).
3. **Encapsulación**: El principio de ocultar los detalles internos de un objeto y exponer solo lo necesario a través de una interfaz pública.
4. **Herencia**: El mecanismo por el cual una clase puede heredar propiedades y métodos de otra clase.
5. **Polimorfismo**: La capacidad de los objetos de diferentes clases relacionadas por herencia de responder a la misma interfaz de métodos de diferentes maneras.

JavaScript soporta la POO mediante el uso de prototipos y, más recientemente, con la introducción de la sintaxis de clases en ECMAScript 6 (ES6). A continuación, se explica cómo utilizar la POO en JavaScript.

Definición de Clases y Creación de Objetos

Con ES6, se pueden definir clases de manera similar a otros lenguajes de programación orientados a objetos.

```
// Definición de una clase
class Persona {
    constructor(nombre, edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    // Método de la clase
    saludar() {
        console.log(`Hola, me llamo ${this.nombre} y tengo ${this.edad} años.`);
    }
}

// Creación de un objeto (instancia de la clase)
const persona1 = new Persona("Juan", 30);
persona1.saludar(); // Salida: Hola, me llamo Juan y tengo 30 años.
```

Encapsulación

En JavaScript, la encapsulación se puede lograr utilizando convenciones de nombres (por ejemplo, subrayados) y, en ES2020, mediante el uso de campos privados.

```
class Persona {
    #nombre; // Campo privado
    #edad; // Campo privado

    constructor(nombre, edad) {
        this.#nombre = nombre;
        this.#edad = edad;
    }

    // Método público
```

```
    saludar() {
        console.log(`Hola, me llamo ${this.#nombre} y tengo ${this.#edad} años.`);
    }
}

const persona1 = new Persona("Juan", 30);
persona1.saludar(); // Salida: Hola, me llamo Juan y tengo 30 años.
// console.log(persona1.#nombre); // Error: campo privado
```

Herencia

JavaScript permite la herencia de clases mediante la palabra clave extends.

```
class Persona {
    constructor(nombre, edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    saludar() {
        console.log(`Hola, me llamo ${this.nombre} y tengo ${this.edad} años.`);
    }
}

// Clase Estudiante que hereda de Persona
class Estudiante extends Persona {
    constructor(nombre, edad, grado) {
        super(nombre, edad); // Llama al constructor de la clase padre
        this.grado = grado;
    }

    estudiar() {
        console.log(`${this.nombre} está estudiando en el grado ${this.grado}.`);
    }
}

const estudiante1 = new Estudiante("Ana", 20, "Segundo Año");
estudiante1.saludar(); // Salida: Hola, me llamo Ana y tengo 20 años.
estudiante1.estudiar(); // Salida: Ana está estudiando en el grado Segundo Año.
```

Polimorfismo

El polimorfismo se puede lograr en JavaScript mediante la sobreescritura de métodos en clases derivadas.

```
class Animal {
    hacerSonido() {
        console.log("El animal hace un sonido.");
    }
}

class Perro extends Animal {
    hacerSonido() {
        console.log("El perro ladra.");
    }
}

class Gato extends Animal {
    hacerSonido() {
        console.log("El gato maúlla.");
    }
}

const animales = [new Animal(), new Perro(), new Gato()];

animales.forEach(animal => {
    animal.hacerSonido();
    // Salida:
    // El animal hace un sonido.
    // El perro ladra.
    // El gato maúlla.
});
```

La Programación Orientada a Objetos en JavaScript permite a los desarrolladores crear aplicaciones más organizadas, modulares y reutilizables. Con la introducción de la sintaxis de clases en ES6, la POO en JavaScript se ha vuelto más accesible y fácil de usar, alineándose más con la forma en que otros lenguajes orientados a objetos implementan estos conceptos. La comprensión y el uso de la POO son fundamentales para el desarrollo de aplicaciones complejas y escalables en JavaScript.

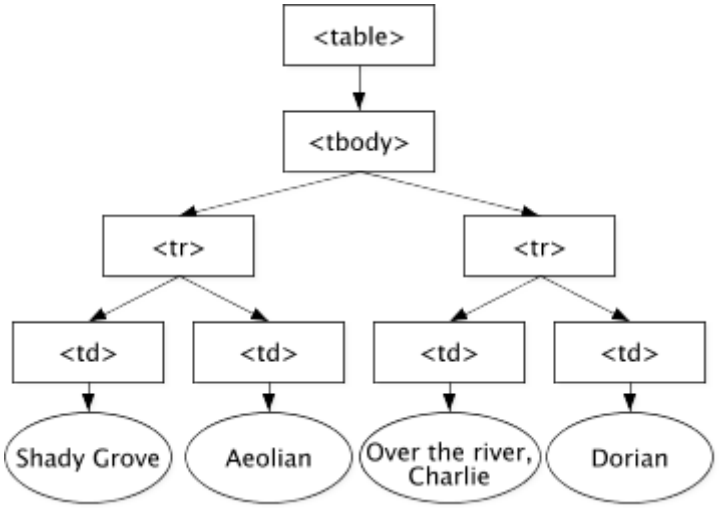
4.4- Qué es el DOM

En el desarrollo web, uno de los conceptos clave es el Document Object Model (DOM). El DOM es una interfaz de programación que permite a los desarrolladores acceder y manipular el contenido, la estructura y el estilo de los documentos HTML y XML

El Document Object Model (DOM) es una representación estructural del documento HTML o XML en forma de un árbol de nodos. Cada elemento, atributo y pieza de contenido dentro de un documento HTML se representa como un nodo en este árbol. El DOM proporciona una API que permite a los lenguajes de programación, como JavaScript, interactuar y modificar la estructura, el estilo y el contenido de los documentos web.

A través del DOM, los desarrolladores pueden **crear dinámicamente contenido** web interactivo y responsivo, mejorando la experiencia del usuario.

En este dibujo podemos ver una representación de cómo funciona el DOM, en forma de árbol, los elementos de una tabla y al final su contenido. Los programadores podemos acceder a cada nodo del árbol y podemos modificar o transformar cada nodo en función de lo que sea requerido en ese momento.



Estructura de árbol del DOM

(fuente: W3C)

Características principales del DOM:

1. **Estructura en árbol**: El DOM representa el documento como una estructura jerárquica en forma de árbol, donde cada nodo puede tener hijos. El nodo raíz es el documento mismo, y de él cuelgan todos los demás elementos y contenido.

2. **Nodos**: Los nodos son las unidades fundamentales del DOM. Existen varios tipos de nodos, como:
   - **Elementos**: Representan las etiquetas HTML (por ejemplo, `<div>`, `<p>`, `<a>`).
   - **Atributos**: Representan los atributos de los elementos (por ejemplo, `class`, `id`).
   - **Texto**: Representan el contenido textual dentro de los elementos.
   - **Comentarios**: Representan los comentarios en el código HTML.

3. **Accesibilidad y manipulación**: A través de lenguajes de programación como JavaScript, los desarrolladores pueden acceder y manipular cualquier parte del documento. Pueden agregar, modificar o eliminar nodos, cambiar atributos y estilos, y responder a eventos del usuario.

4. **Eventos**: El DOM permite la creación y gestión de eventos. Los desarrolladores pueden definir comportamientos específicos en respuesta a acciones del usuario, como clics, desplazamientos, cambios de formulario, entre otros.

5. **Compatibilidad con múltiples lenguajes**: Aunque comúnmente se asocia con JavaScript, el DOM es un estándar independiente del lenguaje y puede ser manipulado por cualquier lenguaje de programación que pueda interactuar con el navegador.

6. **Interactividad y dinamismo**: El DOM permite la creación de contenido dinámico e interactivo. Mediante el uso de JavaScript y el DOM, los desarrolladores pueden actualizar partes del documento en respuesta a la interacción del usuario sin necesidad de recargar la página completa.

Un ejemplo común de uso del DOM es la manipulación de elementos HTML mediante JavaScript. A continuación, se muestra un pequeño ejemplo de cómo se puede cambiar el contenido de un elemento mediante JavaScript:

```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo de DOM</title>
    <script>
        function cambiarContenido() {
            // Acceder al elemento con id "miDiv"
            var elemento = document.getElementById("miDiv");
            // Cambiar su contenido
            elemento.innerHTML = "¡El contenido ha cambiado!";
        }
    </script>
</head>
<body>
    <div id="miDiv">Este es el contenido original.</div>
    <button onclick="cambiarContenido()">Cambiar contenido</button>
</body>
</html>
```

4.5- Uso asíncrono de JavaScript

La asincronía es un concepto fundamental en la programación que permite que ciertas operaciones se ejecuten de manera no bloqueante. Esto significa que un programa puede iniciar una operación (como una solicitud de red, lectura de un archivo, o acceso a una base de datos) y continuar ejecutando otras tareas sin esperar a que esta operación se complete. Cuando la operación asíncrona termina, el programa es notificado y puede manejar el resultado de dicha operación.

JavaScript es un lenguaje de programación que permite la ejecución asíncrona, lo cual es crucial para manejar operaciones que toman tiempo, como las solicitudes de red, la lectura de archivos, y la interacción con bases de datos, sin bloquear la ejecución del programa. Existen varias formas de manejar la asincronía en JavaScript: callbacks, promesas y la sintaxis async/await introducida en ECMAScript 2017.

La asincronía es una característica poderosa de JavaScript que permite construir aplicaciones **eficientes y responsivas**. A través de callbacks, promesas y async/await, los desarrolladores tienen diversas herramientas para manejar tareas asíncronas de manera efectiva

Un **callback** es una función que se pasa como argumento a otra función y se ejecuta después de que la operación se complete.

```
function hacerAlgo(callback) {
    setTimeout(function() {
        console.log("Operación completada.");
        callback();
    }, 1000);
}

function despuesDeHacerAlgo() {
    console.log("Esto se ejecuta después de la operación.");
}
```

```
hacerAlgo(despuesDeHacerAlgo);
// Salida:
// Operación completada.
// Esto se ejecuta después de la operación.
```

Las **promesas** son objetos que representan la eventual finalización (o falla) de una operación asíncrona y su valor resultante. Proveen métodos then y catch para manejar la resolución o rechazo de la operación.

```
let miPromesa = new Promise((resolve, reject) => {
    let exito = true;
    setTimeout(() => {
        if (exito) {
            resolve("Operación exitosa.");
        } else {
            reject("Operación fallida.");
        }
    }, 1000);
});

miPromesa
    .then((mensaje) => {
        console.log(mensaje); // Operación exitosa.
    })
    .catch((error) => {
        console.log(error);
    });
```

La sintaxis **async/await** es una forma más moderna y clara de trabajar con código asíncrono, haciendo que se parezca más a código sincrónico. async se utiliza para declarar una función asíncrona, y await se utiliza para esperar a que una promesa se resuelva.

```
async function hacerSolicitud(url) {
    try {
        let respuesta = await fetch(url);
        if (!respuesta.ok) {
            throw new Error("Error en la solicitud: " + respuesta.statusText);
        }
        let datos = await respuesta.json();
        return datos;
```

```
    } catch (error) {
        console.error("Error:", error);
    }
}

async function procesarDatos() {
    let url = "https://jsonplaceholder.typicode.com/todos/1";
    let datos = await hacerSolicitud(url);
    if (datos) {
        console.log("Datos obtenidos:", datos);
    }
}

procesarDatos();
// Salida esperada:
// Datos obtenidos: { userId: 1, id: 1, title: "delectus aut autem", completed:
false }
```

El manejo de excepciones en código asíncrono, especialmente con `async` y `await`, también es importante. En estos casos, se pueden utilizar `try...catch` dentro de funciones `async`.

```
async function obtenerDatos(url) {
    try {
        let respuesta = await fetch(url);
        if (!respuesta.ok) {
            throw new Error("Error en la solicitud: " + respuesta.statusText);
        }
        let datos = await respuesta.json();
        console.log("Datos obtenidos:", datos);
    } catch (error) {
        console.error("Error al obtener los datos:", error.message);
    }
}

obtenerDatos("https://jsonplaceholder.typicode.com/todos/1");
obtenerDatos("https://jsonplaceholder.typicode.com/todos/0"); // URL incorrecta
```

El manejo de excepciones en código asíncrono es crucial para mantener la estabilidad y la previsibilidad de las aplicaciones modernas.

4.6- Frameworks JavaScript

Un framework es una estructura de software que proporciona una base estándar sobre la cual los desarrolladores pueden construir aplicaciones. Ofrece un conjunto de herramientas, bibliotecas y convenciones que facilitan y agilizan el desarrollo de software al proporcionar soluciones predefinidas a problemas comunes.

Los frameworks ayudan a estandarizar el desarrollo, mejorar la eficiencia y la mantenibilidad del código, y permitir a los desarrolladores enfocarse en las características específicas de su aplicación en lugar de reinventar componentes básicos.

Los frameworks de JavaScript como Angular, React, Vue.js, Ember.js y Svelte proporcionan herramientas y estructuras poderosas para desarrollar aplicaciones web modernas y eficientes. Cada uno tiene sus propias características y fortalezas, lo que permite a los desarrolladores elegir el que mejor se adapte a sus necesidades específicas y al tipo de proyecto que están desarrollando. Estos frameworks no sólo simplifican el proceso de desarrollo, sino que también ayudan a mantener el código organizado, escalable y mantenible.

Angular

- Descripción: Angular es un framework de desarrollo de aplicaciones web de código abierto, mantenido por Google. Está diseñado para construir aplicaciones web dinámicas y robustas con una arquitectura basada en componentes y una fuerte integración con TypeScript.
- Características:
    - Arquitectura basada en componentes.
    - Data binding bidireccional.
    - Inyección de dependencias.
    - Soporte integral para testing.
    - Herramientas de desarrollo y CLI.

React

- Descripción: Aunque técnicamente es una biblioteca, React es a menudo considerado un framework debido a su ecosistema robusto. Desarrollado por Facebook, React se centra en la construcción de interfaces de usuario con una arquitectura basada en componentes.
- Características:
    - Arquitectura basada en componentes.
    - Virtual DOM para un rendimiento eficiente.
    - Unidirectional data flow.
    - JSX para la escritura de componentes.
    - Ecosistema rico con herramientas como React Router y Redux.

Vue.js

- Descripción: Vue.js es un framework progresivo para construir interfaces de usuario. Se puede integrar fácilmente con otros proyectos y bibliotecas, y es conocido por su enfoque incremental y su curva de aprendizaje amigable.
- Características:
    - Data binding bidireccional.
    - Sistema de componentes reactivos.
    - Enfoque modular y combinable.
    - Herramientas oficiales como Vue Router y Vuex.
    - Documentación extensa y amigable.

Ember.js

- Descripción: Ember.js es un framework para construir aplicaciones web ambiciosas. Es conocido por su enfoque en la productividad del desarrollador y sus convenciones estrictas que ayudan a mantener el código organizado y predecible.
- Características:
    - Enrutamiento avanzado.
    - Data binding bidireccional.
    - Sistema de plantillas basado en Handlebars.
    - Herramientas integradas como Ember CLI.
    - Convenciones sobre configuración.

Svelte

- Descripción: Svelte es un framework moderno que se diferencia de otros al compilar el código en componentes altamente eficientes durante la construcción, eliminando la necesidad de un framework en tiempo de ejecución.
- Características:
    - Compilación a código altamente eficiente.
    - Sintaxis concisa y fácil de aprender.
    - Reactividad integrada sin necesidad de estado global.
    - No hay necesidad de Virtual DOM.
    - Tamaño de bundle reducido

React es una biblioteca de JavaScript para construir interfaces de usuario. Es mantenida por Facebook y una comunidad de desarrolladores individuales y compañías. React permite la creación de componentes reutilizables que pueden gestionar su propio estado.

A continuación, se muestra un ejemplo básico de una aplicación React que incluye la creación de un componente, el uso del estado y la gestión de eventos.

Paso 1: Instalación y Configuración Inicial

Para comenzar con React, asegúrate de tener Node.js y npm instalados en tu sistema. Luego, puedes crear una nueva aplicación React utilizando Create React App:

```
npx create-react-app mi-aplicacion-react
cd mi-aplicacion-react
```

Inicia el servidor de desarrollo:

```
npm start
```

Abre tu navegador y navega a **http://localhost:3000/** para ver la aplicación React en funcionamiento.

Paso 2: Creación de un Componente

Vamos a crear un componente llamado Saludo que mostrará un mensaje de bienvenida.

Edita el archivo src/App.js para incluir el nuevo componente Saludo:

```
import React from 'react';
import './App.css';

// Definición del componente Saludo
function Saludo({ nombre }) {
  return (
    <div>
      <p>Hola, {nombre}!</p>
    </div>
  );
}

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <Saludo nombre="Juan" />
      </header>
    </div>
  );
}

export default App;
```

Paso 3: Uso del Estado

Vamos a añadir un componente que gestione su propio estado utilizando el hook useState.

Edita src/App.js para incluir un nuevo componente Contador:

```
import React, { useState } from 'react';
import './App.css';

// Definición del componente Saludo
function Saludo({ nombre }) {
  return (
    <div>
```

```
      <p>Hola, {nombre}!</p>
    </div>
  );
}

// Definición del componente Contador
function Contador() {
  const [contador, setContador] = useState(0);

  return (
    <div>
      <p>Has hecho clic {contador} veces.</p>
      <button onClick={() => setContador(contador + 1)}>
        Incrementar
      </button>
    </div>
  );
}

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <Saludo nombre="Juan" />
        <Contador />
      </header>
    </div>
  );
}

export default App;
```

En este ejemplo, el componente Contador utiliza el hook useState para manejar su propio estado. Cada vez que el usuario hace clic en el botón, se incrementa el contador.

Paso 4: Gestión de Eventos

Vamos a añadir un evento para actualizar el nombre en el componente Saludo utilizando un campo de entrada (input).

Edita src/App.js para incluir la gestión de eventos:

```
import React, { useState } from 'react';
import './App.css';

// Definición del componente Saludo
function Saludo({ nombre }) {
  return (
    <div>
      <p>Hola, {nombre}!</p>
    </div>
  );
}

// Definición del componente Contador
function Contador() {
  const [contador, setContador] = useState(0);

  return (
    <div>
      <p>Has hecho clic {contador} veces.</p>
      <button onClick={() => setContador(contador + 1)}>
        Incrementar
      </button>
    </div>
  );
}

function App() {
  const [nombre, setNombre] = useState('Juan');

  return (
    <div className="App">
      <header className="App-header">
        <Saludo nombre={nombre} />
        <input
          type="text"
          value={nombre}
          onChange={(e) => setNombre(e.target.value)}
          placeholder="Escribe tu nombre"
        />
        <Contador />
      </header>
    </div>
  );
}

export default App;
```

En este ejemplo, el campo de entrada permite al usuario escribir su nombre, y el componente Saludo se actualiza en tiempo real para mostrar el nuevo nombre.

Con estos pasos, has creado una aplicación React básica que incluye la creación de componentes, el uso del estado y la gestión de eventos. Este ejemplo ilustra cómo React facilita la construcción de interfaces de usuario dinámicas y reactivas mediante el uso de componentes reutilizables y el manejo eficiente del estado. React ofrece muchas más funcionalidades avanzadas, como el enrutamiento con React Router, la gestión de estado global con Redux y la creación de aplicaciones móviles con React Native, que puedes explorar para construir aplicaciones más complejas y robustas.