Evidencias Promises, asyncawait 11

```
index.html > html > head > style > .console-output
<html lang="es">
<head>
    <style>
        .console-output {
            font-family: 'Courier New', Courier, monospace;
            white-space: pre-wrap;
            margin-top: 10px;
        }
    </style>
</head>
<body>
    <h1>calidad del Curso</h1>
    <p style="text-align: center; font-style: italic;">&lt;F12&gt; para ver la mayoría de los

    <div class="container">
        <div class="result-output">
            <!-- Los resultados de console.log no se ven directamente aquí, pero es para
            Ver la consola (F12) para los mensajes de depuración.
        </div>
    </div>

    <!-- TEMAS DE PROMESAS, ASYNC/AWAIT -->
    <div class="section-title">11. Promesas, async/await</div>

    <div class="section-title">11.1 Introducción: devoluciones de llamadas (Callbacks)</di
    <div class="content-block">
        <p>Exploración de la programación asíncrona tradicional con callbacks y sus desafi
        <div class="result-output" id="output-11-1"></div>
        <p><em>Ver la consola para la secuencia de eventos asíncronos.</em></p>
    </div>

    <div class="section-title">11.2 Promesa</div>
    <div class="content-block">
        <p>Demostración de cómo se crea y se consume una Promesa, mostrando sus estados (p
        <div class="result-output" id="output-11-2"></div>
        <p><em>Ver la consola para los estados de la promesa.</em></p>
    </div>
```

```
    <div class="section-title">11.2 Promesa</div>
    <div class="content-block">
        <p>Demostración de cómo se crea y se consume una Promesa, mostrando sus estados (pending
        <div class="result-output" id="output-11-2"></div>
        <p><em>Ver la consola para los estados de la promesa.</em></p>
    </div>

    <div class="section-title">11.3 Promesas encadenadas</div>
    <div class="content-block">
        <p>Ejemplo de cómo encadenar múltiples operaciones asíncronas usando `.then()` para mant
        <div class="result-output" id="output-11-3"></div>
        <p><em>Ver la consola para la secuencia del encadenamiento.</em></p>
    </div>

    <div class="section-title">11.4 Manejo de errores con promesas</div>
    <div class="content-block">
        <p>Cómo capturar y manejar errores en promesas usando `.catch()` y cómo los errores se p
        <div class="result-output" id="output-11-4"></div>
        <p><em>Ver la consola para la demostración de manejo de errores.</em></p>
    </div>

    <div class="section-title">11.5 API de promesas</div>
    <div class="content-block">
        <p>Uso de métodos estáticos como `Promise.all()`, `Promise.race()`, `Promise.allSettled(
        <div class="result-output" id="output-11-5"></div>
        <p><em>Ver la consola para los resultados de la API de promesas.</em></p>
    </div>

    <div class="section-title">11.6 Promisificación</div>
    <div class="content-block">
        <p>Convertir funciones basadas en callbacks en funciones que devuelven promesas.</p>
        <div class="result-output" id="output-11-6"></div>
        <p><em>Ver la consola para el resultado de la promisificación.</em></p>
```

```
// ================================================================================
// TEMA: 3.1 Depuración en el navegador — [Fecha Actual]
// Este bloque demuestra cómo usar `console.log()` para mostrar mensajes en la consola
// del navegador. Es fundamental para depurar y entender el flujo de ejecución.
// Aquí se mostrarán mensajes que puedes ver presionando F12 en tu navegador
// y yendo a la pestaña "Console".
// Aprendí: La importancia de console.log para el debugging básico.
// ================================================================================
console.log("3.1 Depuración: Mensaje de ejemplo en la consola.");
let debugVariable = "Hola desde la depuración!";
console.log("3.1 Depuración: Valor de debugVariable:", debugVariable);
console.warn("3.1 Depuración: Esto es una advertencia.");
console.error("3.1 Depuración: Esto es un error.");


// ================================================================================
// TEMA: 11.1 Introducción: devoluciones de llamadas (Callbacks) — [Fecha Actual]
// Este bloque explora la programación asíncrona utilizando callbacks.
// Demuestra cómo una función puede recibir otra función como argumento y ejecutarla
// cuando una operación asíncrona ha terminado.
// Se ilustra el concepto de "Callback Hell" (infierno de callbacks) para mostrar
// las dificultades de anidar muchos callbacks.
// Aprendí: Qué son los callbacks, cómo se usan para asincronía y el problema del Callback Hell.
// ================================================================================
console.log("\n--- 11.1 Callbacks ---");

function loadScript(src, callback) {
    let script = document.createElement('script');
    script.src = src;
    script.onload = () => callback(null, src); // Éxito: sin error, con la fuente
    script.onerror = () => callback(new Error(`Error al cargar el script ${src}`)); // Error
    document.head.append(script);
}
```

```
// ================================================================================
// TEMA: 11.5 API de promesas — [Fecha Actual]
// Este bloque explora los métodos estáticos de la clase Promise que permiten trabajar
// con múltiples promesas simultáneamente: `Promise.all()`, `Promise.race()`,
// `Promise.allSettled()` y `Promise.any()`.
// Aprendí: Cómo gestionar colecciones de promesas para diferentes escenarios (todas resuelven
// ================================================================================
console.log("\n--- 11.5 API de promesas ---");

const p1 = new Promise(resolve => setTimeout(() => resolve(1), 1000));
const p2 = new Promise(resolve => setTimeout(() => resolve(2), 2000));
const p3 = new Promise(resolve => setTimeout(() => resolve(3), 500));
const pError = new Promise((resolve, reject) => setTimeout(() => reject(new Error("Fallo pErro

// Promise.all(): Espera a que todas las promesas se cumplan. Si una falla, todas fallan.
Promise.all([p1, p2, p3])
    .then(results => {
        console.log('11.5 API: Promise.all éxito:', results); // [1, 2, 3]
        document.getElementById('output-11-5').innerHTML += `<p>Promise.all éxito: ${results}<
    })
    .catch(error => {
        console.error('11.5 API: Promise.all error:', error.message);
        document.getElementById('output-11-5').innerHTML += `<p>Promise.all error: ${error.mes
    });

Promise.all([p1, pError, p3]) // Una de ellas falla, entonces Promise.all falla
    .then(results => {
        console.log('11.5 API: Promise.all con error (no debería verse):', results);
    })
    .catch(error => {
        console.error('11.5 API: Promise.all con error capturado:', error.message); // Fallo p
        document.getElementById('output-11-5').innerHTML += `<p>Promise.all con error capturad
    });
```

```
    .catch(error => {
        console.error('11.6 Promisificación: Error al cargar (promesa, esperado):', error.message);
        document.getElementById('output-11-6').innerHTML += `<p>❌ Error promesa (esperado): ${erro
    });


// ================================================================================
// TEMA: 11.7 Microtareas — [Fecha Actual]
// Este bloque explica el concepto de microtareas en el bucle de eventos de JavaScript,
// que son tareas de alta prioridad (como los `.then()` de las promesas) que se ejecutan
// antes de las macrotareas (como `setTimeout` o eventos DOM).
// Ayuda a entender el orden de ejecución de código asíncrono.
// Aprendí: El orden de ejecución entre microtareas y macrotareas en el event loop.
// ================================================================================
console.log("\n--- 11.7 Microtareas ---");

console.log('11.7 Microtareas: Inicio del script');
document.getElementById('output-11-7').innerHTML += '<p>Inicio del script</p>';

setTimeout(() => {
    console.log('11.7 Microtareas: setTimeout (Macrotarea)');
    document.getElementById('output-11-7').innerHTML += '<p>setTimeout (Macrotarea)</p>';
}, 0); // Se ejecuta en la próxima macrotarea

Promise.resolve()
    .then(() => {
        console.log('11.7 Microtareas: Promise.then (Microtarea 1)');
        document.getElementById('output-11-7').innerHTML += '<p>Promise.then (Microtarea 1)</p>';
    })
    .then(() => {
        console.log('11.7 Microtareas: Promise.then (Microtarea 2)');
        document.getElementById('output-11-7').innerHTML += '<p>Promise.then (Microtarea 2)</p>';
    });

console.log('11.7 Microtareas: Fin del script (síncrónico)');
```

```
// y el manejo de errores con try...catch.
// ================================================================================
console.log("\n--- 11.8 Async/Await ---");

function simulateFetch(data, delay, shouldFail = false) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (shouldFail) {
                reject(new Error(`Fallo en la carga de ${data}`));
            } else {
                resolve(data);
            }
        }, delay);
    });
}

async function fetchDataSequence() {
    try {
        console.log('11.8 Async/Await: Iniciando secuencia de datos...');
        document.getElementById('output-11-8').innerHTML += '<p>Iniciando secuencia de datos.

        const user = await simulateFetch("Usuario Objeto", 2000);
        console.log('11.8 Async/Await: Usuario obtenido:', user);
        document.getElementById('output-11-8').innerHTML += `<p>✅ Usuario obtenido: ${user}

        const posts = await simulateFetch("Posts Array", 1500);
        console.log('11.8 Async/Await: Posts obtenidos:', posts);
        document.getElementById('output-11-8').innerHTML += `<p>✅ Posts obtenidos: ${posts}

        const comments = await simulateFetch("Comentarios Objeto", 1000);
        console.log('11.8 Async/Await: Comentarios obtenidos:', comments);
        document.getElementById('output-11-8').innerHTML += `<p>✅ Comentarios obtenidos: ${c

        console.log('11.8 Async/Await: Secuencia de datos completa.');
```

Evidencias Promises, asyncawait 11

calidad del Curso

<F12> para ver la mayoría de los resultados.

```
Ver la consola (F12) para los mensajes de depuración.
```

11. Promesas, async/await

11.1 Introducción: devoluciones de llamadas (Callbacks)

Exploración de la programación asíncrona tradicional con callbacks y sus desafíos, como el "Callback Hell".

Ver la consola para la secuencia de eventos asíncronos.

11.2 Promesa

Demostración de cómo se crea y se consume una Promesa, mostrando sus estados (pending, fulfilled, rejected).