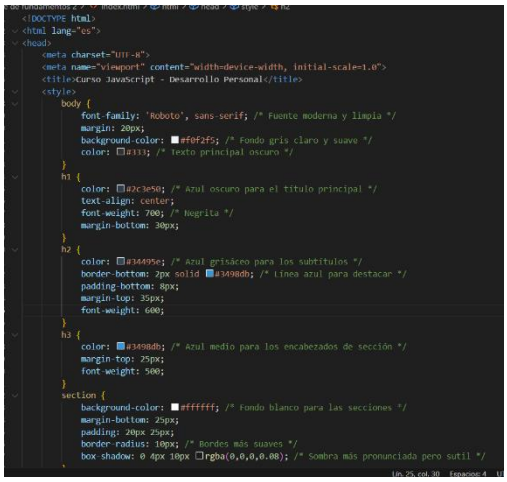
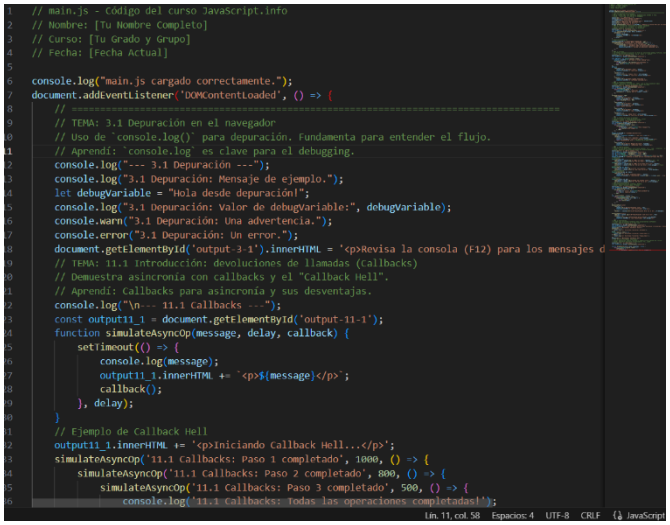
EVIDENCIAS 11

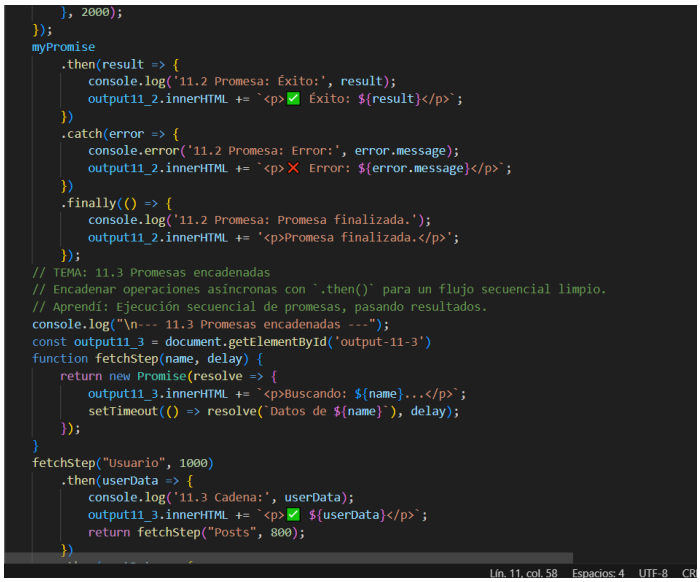


```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Curso JavaScript - Desarrollo Personal</title>
    <style>
        body {
            font-family: 'Roboto', sans-serif; /* Fuente moderna y limpia */
            margin: 20px;
            background-color: #f0f2f5; /* Fondo gris claro y suave */
            color: #333; /* Texto principal oscuro */
        }
        h1 {
            color: #2c3e50; /* Azul oscuro para el título principal */
            text-align: center;
            font-weight: 700; /* Negrita */
            margin-bottom: 30px;
        }
        h2 {
            color: #34495e; /* Azul grisáceo para los subtítulos */
            border-bottom: 2px solid #3498db; /* Línea azul para destacar */
            padding-bottom: 8px;
            margin-top: 35px;
            font-weight: 600;
        }
        h3 {
            color: #3498db; /* Azul medio para los encabezados de sección */
            margin-top: 25px;
            font-weight: 500;
        }
        section {
            background-color: #ffffff; /* Fondo blanco para las secciones */
            margin-bottom: 25px;
            padding: 20px 25px;
            border-radius: 10px; /* Bordes más suaves */
            box-shadow: 0 4px 10px rgba(0,0,0,0.08); /* Sombra más pronunciada pero sutil */
```



```
// main.js - Código del curso JavaScript.info
// Nombre: [Tu Nombre Completo]
// Curso: [Tu Grado y Grupo]
// Fecha: [Fecha Actual]

console.log("main.js cargado correctamente.");
document.addEventListener("DOMContentLoaded", () => {
    // ================================================================================
    // TEMA: 3.1 Depuración en el navegador
    // Uso de `console.log()` para depuración. Fundamenta para entender el flujo.
    // Aprendí: `console.log` es clave para el debugging.
    console.log("--- 3.1 Depuración ---");
    console.log("3.1 Depuración: Mensaje de ejemplo.");
    let debugVariable = "Hola desde depuración!";
    console.log("3.1 Depuración: Valor de debugVariable:", debugVariable);
    console.warn("3.1 Depuración: Una advertencia.");
    console.error("3.1 Depuración: Un error.");
    document.getElementById('output-3-1').innerHTML = '<p>Revisa la consola (F12) para los mensajes d
    // TEMA: 11.1 Introducción: devoluciones de llamadas (Callbacks)
    // Demuestra asincronía con callbacks y el "Callback Hell".
    // Aprendí: Callbacks para asincronía y sus desventajas.
    console.log("\n--- 11.1 Callbacks ---");
    const output11_1 = document.getElementById('output-11-1');
    function simulateAsyncOp(message, delay, callback) {
        setTimeout(() => {
            console.log(message);
            output11_1.innerHTML += `<p>${message}</p>`;
            callback();
        }, delay);
    }
    // Ejemplo de Callback Hell
    output11_1.innerHTML += '<p>Iniciando Callback Hell...</p>';
    simulateAsyncOp('11.1 Callbacks: Paso 1 completado', 1000, () => {
        simulateAsyncOp('11.1 Callbacks: Paso 2 completado', 800, () => {
            simulateAsyncOp('11.1 Callbacks: Paso 3 completado', 500, () => {
                console.log('11.1 Callbacks: Todas las operaciones completadas!');
```



```
    }, 2000);
});
myPromise
    .then(result => {
        console.log('11.2 Promesa: Éxito:', result);
        output11_2.innerHTML += `<p>✅ Éxito: ${result}</p>`;
    })
    .catch(error => {
        console.error('11.2 Promesa: Error:', error.message);
        output11_2.innerHTML += `<p>❌ Error: ${error.message}</p>`;
    })
    .finally(() => {
        console.log('11.2 Promesa: Promesa finalizada.');
        output11_2.innerHTML += '<p>Promesa finalizada.</p>';
    });
// TEMA: 11.3 Promesas encadenadas
// Encadenar operaciones asíncronas con `.then()` para un flujo secuencial limpio.
// Aprendí: Ejecución secuencial de promesas, pasando resultados.
console.log("\n--- 11.3 Promesas encadenadas ---");
const output11_3 = document.getElementById('output-11-3')
function fetchStep(name, delay) {
    return new Promise(resolve => {
        output11_3.innerHTML += `<p>Buscando: ${name}...</p>`;
        setTimeout(() => resolve(`Datos de ${name}`), delay);
    });
}
fetchStep("Usuario", 1000)
    .then(userData => {
        console.log('11.3 Cadena:', userData);
        output11_3.innerHTML += `<p>✅ ${userData}</p>`;
        return fetchStep("Posts", 800);
    })
```

```
});
// TEMA: 11.5 API de promesas
// Métodos estáticos para trabajar con múltiples promesas: `all`, `race`, `allSettled`,
// Aprendí: Gestión de colecciones de promesas para diversos escenarios.
console.log("\n--- 11.5 API de promesas ---");
const output11_5 = document.getElementById('output-11-5');
const pA = new Promise(r => setTimeout(() => r('A'), 1000));
const pB = new Promise(r => setTimeout(() => r('B'), 500));
const pFail = new Promise((_, rej) => setTimeout(() => rej(new Error('Fallo!')), 700));
output11_5.innerHTML += '<p>Revisa la consola para los resultados de Promise API.</p>';
// Promise.all
Promise.all([pA, pB])
    .then(results => console.log('11.5 API: all éxito:', results)) // ['A', 'B']
    .catch(error => console.error('11.5 API: all error:', error.message));
Promise.all([pA, pFail])
    .then(results => console.log('11.5 API: all con fallo (no se ve):', results))
    .catch(error => console.error('11.5 API: all con fallo:', error.message)); // Fallo
// Promise.race
Promise.race([pA, pB, pFail])
    .then(result => console.log('11.5 API: race (primero):', result)) // 'B'
    .catch(error => console.error('11.5 API: race (no se ve):', error.message));
// Promise.allSettled
Promise.allSettled([pA, pB, pFail])
    .then(results => {
        console.log('11.5 API: allSettled (todos):', results);
        output11_5.innerHTML += `<p>allSettled: ${results.map(r => r.status).join(', ')
    });
// Promise.any
Promise.any([pFail, pA, pB])
    .then(result => console.log('11.5 API: any (primero éxito):', result)) // 'B'
    .catch(error => console.error('11.5 API: any (no se ve):', error.message));
```

```
ocument.addEventListener('DOMContentLoaded', () => {
    .catch(err => {
        output11_6.innerHTML += `<p>❌ Promisificado: ${err.message}</p>`;
    });
// TEMA: 11.7 Microtareas
// El Event Loop: microtareas (promesas) antes que macrotareas (setTimeout)
// Aprendí: Orden de ejecución en el Event Loop.
console.log("\n--- 11.7 Microtareas ---");
const output11_7 = document.getElementById('output-11-7');
output11_7.innerHTML += '<p>Ver la consola para el orden: Síncrónico -> Mic
console.log('11.7 Microtareas: Síncrónico: Inicio');
setTimeout(() => {
    console.log('11.7 Microtareas: Macrotarea: setTimeout');
}, 0);
Promise.resolve().then(() => {
    console.log('11.7 Microtareas: Microtarea: Promise.then 1');
}).then(() => {
    console.log('11.7 Microtareas: Microtarea: Promise.then 2');
});
console.log('11.7 Microtareas: Síncrónico: Fin');
// TEMA: 11.8 Asíncrono/espera (async/await)
// Sintaxis moderna para Promesas, haciendo el código asíncrono legible com
// Aprendí: Simplificar asincronía con async/await y `try...catch`.
console.log("\n--- 11.8 Async/Await ---");
const output11_8 = document.getElementById('output-11-8');

function delayResolve(value, delay, shouldFail = false) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (shouldFail) reject(new Error(`Fallo en ${value}`));
```