

# CURSO DE JAVASCRIPT

**Corporación Ecuatoriana para el Desarrollo de la  
Investigación y la Academia**

**Ing. María Fernanda Granda J., PhD**

**3-12-2025**

**“Conectando ideas,  
transformando  
sociedades”**

## **Módulo 9:** Comunicación asincrónica

- Definición de un método asincrónico.
- Solicitud de ejecución asincrónica.
- Procesamiento de respuesta de un método asincrónico

## Definición de un Método Asíncronico

- Vamos a ver dos formas:
- **1.- Uso de fetch():** fetch es una función incorporada que realiza peticiones HTTP y devuelve una promesa.

```
fetch("https://jsonplaceholder.typicode.com/users")  
  .then(response => response.json())  
  .then(data => console.log(data))  
  .catch(error => console.error("Error:", error));
```

## Ejemplo de un método asíncronico

- ❑ La función **fetch** en JavaScript se utiliza para realizar solicitudes de red, como obtener datos de una API o enviar datos a un servidor es soportado por la mayoría de los navegadores modernos.

- `fetch()` devuelve una promesa que representa la petición HTTP.
- `.then()` se usa para procesar la respuesta. Si la respuesta no es exitosa (por ejemplo, un código de estado 404), se lanza un error que es capturado por el `catch()`.
- `.then()` se usa para convertir la respuesta a JSON.
- `.then()` finalmente maneja los datos JSON.
- `.catch()` captura cualquier error que ocurra en la cadena de promesas.

```
fetch('https://jsonplaceholder.typicode.com/users')
  .then(response => {
    if (!response.ok) {
      throw new Error('La red no responde');
    }
    return response.json(); // Convertir a JSON
  })
  .then(data => {
    console.log(data); // Manejar los datos
  })
  .catch(error => {
    console.error('Hubo un problema con la operación fetch:', error); // Manejar errores
  });
```

## ¿Qué es JSON?

- ❑ JSON, o Notación de Objetos de JavaScript, es un formato ligero para el intercambio de datos.
- ❑ Utiliza pares clave-valor y estructuras de datos como arreglos para representar información.
- ❑ Se utiliza comúnmente para transmitir datos en aplicaciones web, actuando como una alternativa más simple y liviana al XML.

```
{  
  "nombre": "Juan Pérez",  
  "edad": 30,  
  "ciudad": "Madrid",  
  "intereses": ["música", "libros", "deportes"]  
}
```

# Definición de un Método Asíncrono

- 2.- Uso de async / await (sintaxis más moderna)

```
async function obtenerDatos() {  
  try {  
    const respuesta = await fetch("https://jsonplaceholder.  
    typicode.com/users");  
    const datos = await respuesta.json();  
    console.log(datos);  
  } catch (error) {  
    console.error("Error:", error);  
  }  
}
```

## Ejemplo

- ❑ Petición a una API pública usando fetch

```
fetch("https://jsonplaceholder.typicode.com/posts/1")  
  .then(res => res.json())  
  .then(data => {  
    console.log("Título:", data.title);  
  });
```

# Ejemplo

## ❑ Versión con async/wait

```
async function mostrarUsuario(id) {  
  try {  
    const res = await fetch(`https://jsonplaceholder.typicode.com/users/${id}`);  
    const user = await res.json();  
    console.log("Usuario:", user.name);  
  } catch (err) {  
    console.error("Error:", err);  
  }  
}
```

## Ejemplo

- ❑ Crear una función que haga dos peticiones y junte los resultados

```
async function cargarDatos() {  
  const post = await fetch("https://jsonplaceholder.  
typicode.com/posts/1").then(r => r.json());  
  const user = await fetch("https://jsonplaceholder.  
typicode.com/users/1").then(r => r.json());  
  console.log("Post:", post.title);  
  console.log("Autor:", user.name);  
}
```

# Ejemplo

- Ejemplo que simula que carga datos en 3 seg. en la página web

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Ejemplo de Promesa</title>
  <style>
    body { font-family: Arial, sans-serif; padding: 20px; }
    #mensaje { font-weight: bold; color: green; margin-top: 20px; }
  </style>
</head>
<body>
  <h2>Simulación de carga de datos</h2>
  <p>Haz clic en el botón para iniciar una operación asincrónica.</p>
  <button onclick="iniciarPromesa()">Iniciar</button>
  <p id="mensaje"></p>
  <script src="codigo.js">
  </script>
</body>
</html>
```

```
function iniciarPromesa() {
  document.getElementById("mensaje").textContent = "Cargando datos...";
  // Simulamos una promesa que se resuelve después de 3 segundos
  let promesa = new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("✅ Datos cargados correctamente.");
    }, 3000);
  });
  // Manejo del resultado de la promesa
  promesa.then(resultado => {
    document.getElementById("mensaje").textContent = resultado;
  });
}
```

# Ejemplo

- Ejemplo que simula que carga datos en la página web

## Simulación de carga de datos

Haz clic en el botón para iniciar una operación asincrónica.

Iniciar

**Cargando datos...**

## Simulación de carga de datos

Haz clic en el botón para iniciar una operación asincrónica.

Iniciar

✓ **Datos cargados correctamente.**

## **EXTENSIÓN DEL EJEMPLO: Gestor de Tareas con DOM + Promesas**

La aplicación ahora:

- ☐ Simula una base de datos remota usando Promesas.
- ☐ Implementa funciones como:
  - ☐ guardarTarea()
  - ☐ obtenerTareas()
  - ☐ eliminarTareaAsync()
- ☐ Incluye un “cargador” visual mientras la Promesa está en ejecución.
- ☐ Integra async/await dentro del flujo del DOM.

## **EXTENSIÓN DEL EJEMPLO: Gestor de Tareas con DOM + Promesas**

Se añaden:

- Promesas para simular operaciones asincrónicas.
- Esperas con `setTimeout()` como si fuera un servidor real.
- Manejo de errores.
- Integración con el DOM.

# ¿Qué conceptos de Promesas y asincronía se aplican?

## 1. Creación de Promesas

Simulación de servidor para obtener, guardar y eliminar tareas.

```
return new Promise((resolve, reject) => {
```

## 2. resolve() y reject()

Para indicar éxito y fracaso respectivamente.

# ¿Qué conceptos de Promesas y asincronía se aplican?

## 3. `setTimeout()` simulando latencia del servidor

Usado para generar la sensación de petición remota.

## 4. Manejo con `async/await`.

```
async function iniciarApp() {  
    mostrarCargador();  
    tareas = await obtenerTareas();  
    ocultarCargador();  
    renderizarLista();  
}
```

# ¿Qué conceptos de Promesas y asincronía se aplican?

## 5. Uso de try / catch / finally dentro de funciones asincrónicas

```
try {  
    const resp = await guardarTarea(texto);  
    mensaje.textContent = "✓ " + resp;  
    mensaje.style.color = "green";  
} catch (error) {  
    mensaje.textContent = "✗ " + error;  
    mensaje.style.color = "red";  
} finally {  
    ocultarCargador();  
    renderizarLista();  
}
```

# ¿Qué conceptos de Promesas y asincronía se aplican?

## 6. Integración DOM + Promesas

Cargador visible mientras espera.

Actualiza UI tras resolución de la promesa.

# VERSIÓN COMPLETA: DOM + Fetch API + JSON Server

Ahora conectada a un servidor real con Fetch API, usando un archivo JSON como "base de datos".

Para esta versión necesitaremos usar JSON Server, una herramienta que convierte un archivo JSON en un servidor REST real.

1. Instalación de JSON Server usando Window PowerShell como administrador

```
npm install -g json-server
```

A veces es necesario dar permisos para la instalación:

- Ejecutar en Window PowerShell como administrador:

```
Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass
```

# VERSIÓN COMPLETA: DOM + Fetch API + JSON Server

También es necesario agregar en las variables de ambiente, la siguiente entrada para la variable PATH:

C:\Users\<TU\_USUARIO>\AppData\Roaming\npm\node\_modules

# VERSIÓN COMPLETA: DOM + Fetch API + JSON Server

2. Crea un archivo llamado db.json en la carpeta del proyecto:

```
{  
  "tarefas": [  
    { "id": 1, "texto": "Comprar leche" },  
    { "id": 2, "texto": "Estudiar JavaScript" }  
  ]  
}
```

# VERSIÓN COMPLETA: DOM + Fetch API + JSON Server

3. Luego ejecuta el servidor:

```
json-server --watch db.json --port 3000
```

Si no deja hay que volver a dar permisos usando Windows PowerShell como administrador:

`Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass`

4. index.htm (Igual que antes)

5. styles.css (incluye cargador)

# VERSIÓN COMPLETA: DOM + Fetch API + JSON Server

6. codigo.js

Versión con Fetch API + JSON Server

Este archivo implementa:

- GET para cargar tareas
- POST para agregar
- DELETE para eliminar
- DOM + asincronía + cargador

# GRACIAS