ASINCRONISMO EN JAVASCRIPT

TEMA 07 – PROCESAMIENTO ASÍNCRONO CON XML Y JSON

7.1. Introducción

En el ámbito del desarrollo web, especialmente cuando trabajamos en el entorno de JavaScript, es fundamental comprender a fondo cómo se gestionan los datos en distintos formatos y de qué manera podemos intercambiarlos de forma asíncrona entre el cliente y el servidor. Dos formatos muy utilizados para la transferencia de datos son XML y JSON.

Aunque JSON ha ido ganando más popularidad, especialmente debido a su integración más sencilla con JavaScript, es conveniente conocer las ventajas y aplicaciones de ambos.

A continuación, abordaremos tres puntos esenciales que ayudarán a consolidar el conocimiento sobre el uso de XML y JSON en procesos asíncronos:

1. Diferencias entre XML y JSON.
2. Cómo trabajar con JSON usando la API `fetch`.
3. Introducción básica al manejo de XML con `DOMParser`.

Con ello, pretendemos que tengas una visión clara de estos dos formatos, sepas utilizarlos en aplicaciones web reales y apliques buenas prácticas a la hora de desarrollar tus proyectos.

7.2. Diferencias entre XML y JSON

Estructura y sintaxis

- **XML (Extensible Markup Language)**
    - Está basado en etiquetas (similar al HTML, pero sin etiquetas predefinidas).
    - Cada bloque de datos se encierra en etiquetas de apertura y cierre.
    - Requiere una sintaxis estricta y un cierre correcto de cada etiqueta.

- Puede incluir atributos en las etiquetas.
- Es más “verbose” a la hora de representar información, ya que cada elemento requiere sus etiquetas correspondientes.

```
<persona>
  <nombre>Juan</nombre>
  <edad>25</edad>
</persona>
```

- **JSON (JavaScript Object Notation)**
  - Está basado en la notación de objetos de JavaScript.
  - Utiliza llaves `{}` para definir objetos y corchetes `[]` para definir listas o arreglos.
  - Los datos se representan en pares `clave: valor`, separados por comas.
  - Requiere un uso correcto de comillas y comas, especialmente en las claves y cadenas de texto.
  - Es menos “verbose”, lo que facilita su lectura y escritura en la mayoría de los casos.

```
{
  "nombre": "Juan",
  "edad": 25
}
```

Legibilidad y popularidad

- **XML** era extremadamente popular hace años en el intercambio de datos entre sistemas. Su principal fortaleza residía en la flexibilidad y la posibilidad de definir esquemas para validar la estructura (XSD). Sin embargo, la complejidad en la escritura y lectura lo ha hecho menos práctico en entornos de desarrollo web más modernos.
- **JSON** se ha convertido en el estándar de facto para la transferencia de datos en aplicaciones web (por ejemplo, en APIs REST). Es mucho más ligero y está directamente integrado en JavaScript, ya que la sintaxis coincide con la definición de objetos en este lenguaje. Asimismo, su popularidad se debe a la facilidad de serializar y deserializar datos (convertir de objeto a cadena y viceversa).

Soporte en JavaScript

- **XML** puede requerir bibliotecas o APIs concretas para su manipulación, como *`DOMParser`* y *`XMLSerializer`*, o la propia interfaz *`XMLHttpRequest`* de antaño si nos remontamos a las primeras implementaciones del AJAX clásico.
- **JSON** se maneja de forma nativa en JavaScript a través de los métodos *`JSON.stringify()`* (para convertir un objeto en una cadena JSON) y *`JSON.parse()`* (para convertir una cadena JSON en un objeto JavaScript).

Uso actual

- **XML** se suele utilizar en casos muy específicos (por ejemplo, la manipulación de ficheros de configuración, ciertas arquitecturas empresariales más antiguas, o la lectura de ficheros SVG, que internamente están basados en XML, aunque no siempre se gestione de la misma manera que un archivo XML puro).
- **JSON** es la opción preferida para casi cualquier tipo de comunicación con APIs modernas, así como para almacenar configuraciones, ya que ofrece mayor ligereza y una integración fluida con JavaScript.

En conclusión, la elección entre XML y JSON dependerá del contexto, de las necesidades del proyecto y de los requerimientos de compatibilidad o de validación. Sin embargo, hoy en día, JSON es la alternativa más habitual y recomendada para la mayoría de desarrollos web en JavaScript.

7.3. Cómo trabajar con JSON

Una de las formas más comunes de realizar peticiones asíncronas para obtener datos en JSON es a través de la API *`fetch`*. Esta API nos permite, de manera nativa en el navegador, realizar solicitudes HTTP y procesar las respuestas de forma relativamente sencilla y moderna, sustituyendo el antiguo uso de *`XMLHttpRequest`*.

Estructura básica de una petición con `fetch`

```
fetch('https://api.ejemplo.com/datos')
  .then(response => {
```

```
    // Comprobamos si la respuesta es correcta
    if (!response.ok) {
                throw new Error('Error en la respuesta: ' +
response.status);
    }
    // parseamos la respuesta a JSON
    return response.json();
  })
  .then(data => {
    // Aquí manejamos el objeto JS resultante
    console.log(data);
  })
  .catch(error => {
    // Manejamos cualquier error durante la petición
    console.error('Ha ocurrido un error:', error);
  });
```

Primero, llamamos a *`fetch`* con la URL o endpoint que queramos.

- Después, la promesa resultante contiene un objeto Response que necesitamos verificar con *`response.ok`*.
- A continuación, utilizamos el método *`response.json()`* para convertir la respuesta a un objeto JavaScript (antes era una cadena).
- Una vez que tenemos el objeto, podemos trabajar con él directamente, por ejemplo, para mostrar datos en una página o para procesarlos de la forma que necesitemos.

Fetch con async/await

```
async function obtenerDatos() {
  try {
    const response = await fetch('https://api.ejemplo.com/datos');
    if (!response.ok) {
      throw new Error('Error en la respuesta: ' +
response.status);
    }
```

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

- *async* se declara en la función que va a utilizar *await*.
- *await* “pausa” la ejecución hasta que la promesa se resuelva, evitando la necesidad de múltiples *.then*.

Buenas prácticas al trabajar con JSON

- Validar siempre que las respuestas del servidor sean correctas, tanto por código de estado (*status*) como por el contenido retornado.
- Estructurar el JSON de manera clara y limpia para facilitar la lectura y el mantenimiento del código.
- Manejar adecuadamente los errores (por ejemplo, si el JSON no está bien formado o si la petición no devuelve lo esperado).
- Usar HTTPS en la medida de lo posible para proteger la transferencia de datos.

Con estos puntos, se pretende mostrar que la combinación de JSON y *fetch* facilita sobremanera la comunicación con servidores y la gestión de datos asíncronos, posicionando a JSON como la opción más cómoda en proyectos JavaScript modernos.

7.4. Introducción al manejo de XML con DOMParser

Aunque JSON sea el formato más común, sigue habiendo situaciones en las que se trabaja con XML. Para ello, JavaScript proporciona herramientas nativas que permiten parsear el XML y transformarlo en un objeto manipulable.

¿Qué es DOMParser?

DOMParser es un objeto disponible en la mayoría de los navegadores modernos, que convierte cadenas de texto que contienen marcado XML (o HTML) en un documento DOM que se puede manipular con las mismas funciones que se utilizan para manipular el DOM del documento HTML de la página.

Uso básico de DOMParser

Imaginemos que recibimos un *string* con contenido XML. Para parsearlo, podemos hacer lo siguiente:

```
const xmlString = `
  <catalog>
    <book id="1">
      <title>Aprendiendo JavaScript</title>
      <author>Juan Pérez</author>
    </book>
    <book id="2">
      <title>Profundizando en XML</title>
      <author>María García</author>
    </book>
  </catalog>
`;

// Creamos una instancia de DOMParser
const parser = new DOMParser();
// Parseamos la cadena con el formato "application/xml" o
"text/xml"
const xmlDoc = parser.parseFromString(xmlString, "text/xml");

// Ahora xmlDoc es un documento DOM con nodos accesibles
const catalog = xmlDoc.getElementsByTagName('catalog')[0];
const books = catalog.getElementsByTagName('book');

for (let i = 0; i < books.length; i++) {
```

```
  const title = books[i].getElementsByTagName('title')
     [0].textContent;
  const author = books[i].getElementsByTagName('author')
     [0].textContent;
  console.log(`Libro ${i + 1}: ${title}, de ${author}`);
}
```

Se crea una instancia de DOMParser.

- Se utiliza *parseFromString*, indicando el tipo de contenido.
- El resultado es un documento DOM con métodos como *getElementsByTagName()*, *querySelectorAll()*, etc.
- De esta forma, podemos recorrer el árbol XML y obtener la información de cada elemento.

Conversión inversa con XMLSerializer

Si necesitamos convertir el DOM en una cadena de texto XML (por ejemplo, tras haber hecho modificaciones sobre el DOM resultante), podemos usar *XMLSerializer* de la siguiente manera:

```
const serializer = new XMLSerializer();
const nuevoXMLString = serializer.serializeToString(xmlDoc);
console.log(nuevoXMLString);
```

Este proceso nos permite, por ejemplo, actualizar nodos o atributos en un documento XML y luego generar de nuevo la cadena resultante para enviarla a un servidor o guardarla en un fichero.

- **Consideraciones al trabajar con XML**
- Es importante asegurarse de que el XML sea válido (etiquetas bien anidadas, atributos correctamente definidos, etc.). De lo contrario, *DOMParser* puede generar errores o ignorar partes del documento.
- A diferencia de JSON, que se integra más naturalmente con objetos JavaScript, el trabajo con XML implica la manipulación del DOM, lo que puede resultar menos intuitivo para desarrolladores no familiarizados con la estructura en forma de árbol.

7.5. Ejemplos

Ejemplo 1: Estudiante y asignaturas que cursa.

El formato JSON podría ser:

```
{
  "nombre": "Carla",
  "edad": 22,
  "matriculada": true,
  "asignaturas": ["Programación", "Bases de Datos", "Entornos de Desarrollo"]
}
```

- **Clave-valor (nombre, edad, matriculada)**: Cada propiedad se expresa como un par *`"clave": valor`*.
- *`"nombre"`* y *`"edad"`* son propiedades del objeto, con valores “Carla” (tipo cadena) y 22 (tipo numérico), respectivamente.
- *`"matriculada"`* es una propiedad booleana (true/false).
- **Array de asignaturas**: La clave *`"asignaturas"`* contiene un array delimitado por corchetes *`[ ... ]`* y compuesto por valores de tipo *`string`*.

Para procesar este JSON en JavaScript, usaríamos, por ejemplo, el método *`JSON.parse()`* si se encuentra en forma de cadena, o podríamos recibirlo ya convertido a objeto gracias a la API *`fetch`*:

```
const jsonString = `
{
  "nombre": "Carla",
  "edad": 22,
  "matriculada": true,
  "asignaturas": ["Programación", "Bases de Datos", "Entornos de Desarrollo"]
}
```

```
`;

// Parseamos la cadena para convertirla en un objeto JavaScript

const estudiante = JSON.parse(jsonString);

console.log(estudiante.nombre); // "Carla"

console.log(estudiante.asignaturas[0]); // "Programación"
```

Una vez parseado, `estudiante` se convierte en un objeto normal de JavaScript: *`estudiante.nombre`* accedería al valor *`"Carla"`* y *`estudiante.asignaturas`* daría acceso al array de asignaturas.

La correspondecia en XML sería:

```
<estudiante>
  <nombre>Carla</nombre>
  <edad>22</edad>
  <matriculada>true</matriculada>
  <asignaturas>
    <asignatura>Programación</asignatura>
    <asignatura>Bases de Datos</asignatura>
    <asignatura>Entornos de Desarrollo</asignatura>
  </asignaturas>
</estudiante>
```

Parseo en JavaScript con DOMParser

```
const xmlString = `
<estudiante>
  <nombre>Carla</nombre>
  <edad>22</edad>
```

```
  <matriculada>true</matriculada>
  <asignaturas>
    <asignatura>Programación</asignatura>
    <asignatura>Bases de Datos</asignatura>
    <asignatura>Entornos de Desarrollo</asignatura>
  </asignaturas>
</estudiante>
`;
// Creación de un DOMParser
const parser = new DOMParser();
// Parseamos el string como XML
const xmlDoc = parser.parseFromString(xmlString, "text/xml");
// Acceso al contenido
const nombre = xmlDoc.getElementsByTagName("nombre")[0].textContent;
const edad = xmlDoc.getElementsByTagName("edad")[0].textContent;
const matriculada = xmlDoc.getElementsByTagName("matriculada")[0].textContent;
// Para el array de asignaturas:
const asignaturas = xmlDoc.getElementsByTagName("asignatura");
console.log("Nombre:", nombre);           // "Carla"
console.log("Edad:", edad);               // "22"
console.log("Matriculada:", matriculada); // "true"
for (let i = 0; i < asignaturas.length; i++) {
  console.log(`Asignatura ${i + 1}:`, asignaturas[i].textContent);
}
```

En este ejemplo, cada nodo *<asignatura>* se maneja de forma individual. No hay un array nativo, sino una colección de nodos que podemos recorrer con un bucle.

Ejemplo 2: Estructuras anidadas con objetos y arrays

En este segundo ejemplo, representaremos la información de un curso con varios módulos y alumnos inscritos. Aquí se apreciará la anidación de objetos y arrays dentro de un mismo JSON:

```json
{
  "curso": "Desarrollo de Aplicaciones Web",
  "codigo": "DAW2024",
  "modulos": [
    {
      "nombre": "Lenguaje de Marcas",
      "numHoras": 120,
      "aprobados": [
        { "alumno": "Lucía", "nota": 8.5 },
        { "alumno": "Pedro", "nota": 7.0 }
      ]
    },
    {
      "nombre": "Entornos de Desarrollo",
      "numHoras": 100,
      "aprobados": [
        { "alumno": "Ana", "nota": 9.0 },
        { "alumno": "Carlos", "nota": 6.8 }
      ]
    }
  ]
}
```

Objeto principal: Contiene las claves *"curso"*, *"codigo"* y *"modulos"*.

- **Array de módulos**: *"modulos"* es un array en el que cada elemento es un objeto con sus propias propiedades (nombre, numHoras...) y un array anidado llamado "aprobados".
- **Array "aprobados"**: Cada elemento de este array es un objeto con dos propiedades: *"alumno"* y *"nota"*.

Manipulación en JavaScript

Una vez que tenemos este JSON (por ejemplo, al recibirlo de un servidor), podríamos trabajar con él como sigue:

```
const jsonString2 = `
{
  "curso": "Desarrollo de Aplicaciones Web",
  "codigo": "DAW2024",
  "modulos": [
    {
      "nombre": "Lenguaje de Marcas",
      "numHoras": 120,
      "aprobados": [
        { "alumno": "Lucía", "nota": 8.5 },
        { "alumno": "Pedro", "nota": 7.0 }
      ]
    },
    {
      "nombre": "Entornos de Desarrollo",
      "numHoras": 100,
      "aprobados": [
        { "alumno": "Ana", "nota": 9.0 },
        { "alumno": "Carlos", "nota": 6.8 }
      ]
```

```
    }
  ]
}
`;
const datosCurso = JSON.parse(jsonString2);
console.log("Nombre del curso:", datosCurso.curso);
console.log("Código:", datosCurso.codigo);

datosCurso.modulos.forEach((modulo, index) => {
  console.log(`\nMódulo ${index + 1}:`, modulo.nombre);
  console.log(`Horas: ${modulo.numHoras}`);

  modulo.aprobados.forEach(aprobado => {
    console.log(`- Alumno: ${aprobado.alumno}, Nota: ${aprobado.nota}`);
  });
});
```

datosCurso.modulos es un array de objetos.

- Cada objeto del array *modulos* tiene su propio array aprobados.
- Navegamos cada nivel con JavaScript de manera sencilla, pues JSON se traduce “uno a uno” a la estructura interna de objetos y arrays de JavaScript.

Correspondencia en XML

Si quisiéramos describir la misma información en **XML**, podríamos diseñar la siguiente estructura:

```
<curso>
  <nombre>Desarrollo de Aplicaciones Web</nombre>
  <codigo>DAW2024</codigo>
  <modulos>
    <modulo>
```

```
      <nombre>Lenguaje de Marcas</nombre>
      <numHoras>120</numHoras>
      <aprobados>
        <aprobado>
          <alumno>Lucía</alumno>
          <nota>8.5</nota>
        </aprobado>
        <aprobado>
          <alumno>Pedro</alumno>
          <nota>7.0</nota>
        </aprobado>
      </aprobados>
    </modulo>
    <modulo>
      <nombre>Entornos de Desarrollo</nombre>
      <numHoras>100</numHoras>
      <aprobados>
        <aprobado>
          <alumno>Ana</alumno>
          <nota>9.0</nota>
        </aprobado>
        <aprobado>
          <alumno>Carlos</alumno>
          <nota>6.8</nota>
        </aprobado>
      </aprobados>
    </modulo>
  </modulos>
</curso>
```

- **Elemento** `<curso>`: Sirve de contenedor principal.

- **Estructuras anidadas**:
  - `<modulos>` se compone de varios elementos `<modulo>`.
  - Cada `<modulo>` dispone de sus elementos `<nombre>`, `<numHoras>` y `<aprobados>`.
  - `<aprobados>` contiene múltiples elementos `<aprobado>`, uno por cada alumno con su correspondiente `<alumno>` y `<nota>`.
- No hay arrays nativos, pero la repetición de elementos `<modulo>` y `<aprobado>` funciona como una forma de lista o conjunto de elementos.

Parseo en JavaScript

Usamos la misma idea con **DOMParser**:

```
const xmlString2 = `
<curso>
  <nombre>Desarrollo de Aplicaciones Web</nombre>
  <codigo>DAW2024</codigo>
  <modulos>
    <modulo>
      <nombre>Lenguaje de Marcas</nombre>
      <numHoras>120</numHoras>
      <aprobados>
        <aprobado>
          <alumno>Lucía</alumno>
          <nota>8.5</nota>
        </aprobado>
        <aprobado>
          <alumno>Pedro</alumno>
          <nota>7.0</nota>
        </aprobado>
      </aprobados>
```

```
    </modulo>
    <modulo>
      <nombre>Entornos de Desarrollo</nombre>
      <numHoras>100</numHoras>
      <aprobados>
        <aprobado>
          <alumno>Ana</alumno>
          <nota>9.0</nota>
        </aprobado>
        <aprobado>
          <alumno>Carlos</alumno>
          <nota>6.8</nota>
        </aprobado>
      </aprobados>
    </modulo>
  </modulos>
</curso>
`;
const parser = new DOMParser();
const doc = parser.parseFromString(xmlString2, "text/xml");

const nombreCurso = doc.getElementsByTagName("nombre")[0].textContent;
const codigoCurso = doc.getElementsByTagName("codigo")[0].textContent;

console.log("Curso:", nombreCurso);
console.log("Código:", codigoCurso);

const modulos = doc.getElementsByTagName("modulo");
```

```
for (let i = 0; i < modulos.length; i++) {
  const modulo = modulos[i];
  const nombreModulo = modulo.getElementsByTagName("nombre")[0].textContent;
  const numHoras = modulo.getElementsByTagName("numHoras")[0].textContent;

  console.log(`\nMódulo ${i + 1}: ${nombreModulo}`);
  console.log(`Horas: ${numHoras}`);

  const aprobados = modulo.getElementsByTagName("aprobado");
  for (let j = 0; j < aprobados.length; j++) {
    const aprobado = aprobados[j];
    const alumno = aprobado.getElementsByTagName("alumno")[0].textContent;
    const nota = aprobado.getElementsByTagName("nota")[0].textContent;
    console.log(`- Alumno: ${alumno}, Nota: ${nota}`);
  }
}
```

Ejemplo 2: Petición JSON y parseo XML local

En este ejemplo, primero realizamos una llamada para obtener datos en formato JSON (simulando una lista de películas), y después parseamos una cadena XML local (simulando catálogos distintos), todo en una misma función.

```
async function obtenerDatosYParsearXML() {
  try {
    // 1. Obtenemos datos en JSON desde un endpoint ficticio
    const respuestaJSON = await fetch('https://api.ejemplo.com/peliculas');
    if (!respuestaJSON.ok) {
      throw new Error('Error al obtener las películas: ' + respuestaJSON.status);
```

```
    }
    const peliculas = await respuestaJSON.json();
    console.log('Películas en JSON:', peliculas);
    // 2. Parseamos una cadena XML local
    const xmlString = `
      <catalogo>
        <entrada>
          <nombre>Entrada General</nombre>
          <precio>10</precio>
        </entrada>
        <entrada>
          <nombre>Entrada VIP</nombre>
          <precio>25</precio>
        </entrada>
      </catalogo>
    `;
    const parser = new DOMParser();
    const xmlDoc = parser.parseFromString(xmlString, 'application/xml');
    const entradas = xmlDoc.getElementsByTagName('entrada');
   // Recorremos las entradas y mostramos la información
    for (let i = 0; i < entradas.length; i++) {
      const nombre = entradas[i].getElementsByTagName('nombre')[0].textContent;
      const precio = entradas[i].getElementsByTagName('precio')[0].textContent;
      console.log(`Entrada ${i + 1}: ${nombre}, precio: ${precio}`);
    }
  } catch (error) {
    console.error('Error en el proceso combinado:', error);
```

```
  }

}

// Llamada a la función

obtenerDatosYParsearXML();
```

Ejemplo 3: construcción de JSON y XML dinámicamente

Vamos a crear un objeto JSON y un documento XML en tiempo de ejecución. Después, simulamos un envío (por consola) de ambos formatos, como si se fuesen a mandar al servidor.

```
function generarYMostrarDatos() {

  // 1. Generar un objeto JSON dinámicamente

  const usuario = {

    nombre: 'Ana',

    edad: 30,

    tecnologias: ['JavaScript', 'Node.js', 'React']

  };

  // Convertimos el objeto en una cadena JSON

  const usuarioJSON = JSON.stringify(usuario);

  console.log('Objeto JSON generado:\n', usuarioJSON);

  // 2. Generar un documento XML dinámicamente

  const docXML = document.implementation.createDocument('', '', null);

  const raiz = docXML.createElement('usuario');

  // Creamos nodos para nombre y edad

  const nombreNodo = docXML.createElement('nombre');

  nombreNodo.textContent = 'Ana';

  const edadNodo = docXML.createElement('edad');

  edadNodo.textContent = '30';

  // Creamos un nodo "tecnologias" con varios hijos
```

```
const tecnologiasNodo = docXML.createElement('tecnologias');
const tecnologia1 = docXML.createElement('tecnologia');
tecnologia1.textContent = 'JavaScript';
const tecnologia2 = docXML.createElement('tecnologia');
tecnologia2.textContent = 'Node.js';
const tecnologia3 = docXML.createElement('tecnologia');
tecnologia3.textContent = 'React';
// Añadimos las tecnologías al nodo "tecnologias"
tecnologiasNodo.appendChild(tecnologia1);
tecnologiasNodo.appendChild(tecnologia2);
tecnologiasNodo.appendChild(tecnologia3);
// Unimos nombre, edad y tecnologias a la raíz
raiz.appendChild(nombreNodo);
raiz.appendChild(edadNodo);
raiz.appendChild(tecnologiasNodo);
// Añadimos la raíz al documento
docXML.appendChild(raiz);
// Serializamos
const serializer = new XMLSerializer();
const usuarioXML = serializer.serializeToString(docXML);
console.log('Documento XML generado:\n', usuarioXML);
// Envío de datos JSON al servidor,
fetch('https://api.ejemplo.com/subir-datos', {
method: 'POST',
   headers: {
     'Content-Type': 'application/json' // En caso de mandar JSON
   },
   body: usuarioJSON
 })
```

```
    .then(res => res.json())
    .then(resultado => console.log('Respuesta del servidor:', resultado))
    .catch(error => console.error('Error al enviar datos JSON:', error));
  // Envío de datos XML al servidor,
   fetch('https://api.ejemplo.com/subir-xml', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/xml' // En caso de mandar XML
    },
    body: usuarioXML
   })
    .then(res => res.text())
     .then(resultado => console.log('Respuesta del servidor al XML:',
resultado))
    .catch(error => console.error('Error al enviar datos XML:', error));
}
// Ejecutamos la función
generarYMostrarDatos();
```

Estos ejemplos muestran cómo, en JavaScript**,** puedes:

- **Recibir o enviar datos en formato JSON** usando *`fetch`*.
- **Parsear y manipular XML** usando el objeto *`DOMParser`* y, si lo necesitas, volver a serializar con *`XMLSerializer`*.
- Combinar ambos formatos en un mismo script, ya sea porque tu aplicación consume datos JSON y mantiene configuraciones en XML, o por otros motivos de compatibilidad y requerimientos de un proyecto.

Recuerda que:

- **JSON** se integra de manera más natural con JavaScript y es el estándar predominante para APIs modernas.
- **XML** sigue siendo útil en contextos específicos (sistemas legacy, configuraciones con esquemas formales, ficheros SVG, etc.).

- Con `fetch`, `async`/`await` y `DOMParser`, dispones de herramientas nativas y potentes para cubrir la mayoría de escenarios de comunicación y procesamiento de datos asíncronos.

Estos ejemplos constituyen la base para manejar de forma profesional e independiente las peticiones asíncronas y la gestión de datos en cualquier aplicación web real desarrollada en JavaScript.