Universidad Camilo José Cela

Facultad de Tecnología y Ciencia
Grado en Ingeniería Informática

Desarrollo de Front-END — Tema 8: AJAX

Desarrollo de Front-END

Código: 76042

Autor: César Andrés Sánchez
Versión: v2.0.2
Año: 2025-2026

Contenidos

Capítulo 8

Desarrollo de Front-END — Tema 8: AJAX

8.1 Introducción

La aparición de AJAX (*Asynchronous JavaScript and XML*) transformó la interacción entre el navegador y el servidor. Hasta entonces, cada acción relevante en una página solía exigir una recarga completa, con la consiguiente pérdida de continuidad para el usuario. Con AJAX se consolidó un modelo distinto: enviar y recibir datos de manera asíncrona y actualizar sólo las partes necesarias de la interfaz. Aunque el término conserve la referencia a *XML*, en la práctica actual predomina el uso de formatos ligeros como JSON y de interfaces más expresivas, como `fetch()`, que favorecen un estilo de programación claro y predecible. El resultado es una experiencia más fluida, próxima a la de una aplicación de escritorio, sin abandonar la naturaleza abierta de la web.

El objetivo de esta introducción es fijar un marco común: qué entendemos hoy por AJAX, en qué se diferencia de otras aproximaciones posteriores y qué conocimientos previos conviene dominar para utilizarlo con criterio en contextos reales de desarrollo.

8.1.1 ¿Qué es AJAX hoy?

En la actualidad, el término AJAX alude menos a una pieza concreta de tecnología y más a un patrón de interacción: el navegador solicita datos al servidor y actualiza la interfaz sin recargar el documento. Este patrón busca tres metas complementarias: mejorar la experiencia de uso, evitar transferencias innecesarias y mantener una interfaz responsiva incluso cuando el intercambio de información es intenso.

Origen y evolución histórica

Los primeros pasos se apoyaron en el objeto `XMLHttpRequest`, que permitió emitir peticiones HTTP desde el navegador sin bloquear la página. Con el tiempo, el uso de XML cedió terreno a JSON por su sencillez de representación y su encaje natural con JavaScript. En paralelo, la estandarización de `fetch()` aportó una forma más directa de expresar las solicitudes y de tratar sus respuestas, lo que facilitó una escritura más legible y una gestión ordenada de los posibles fallos. Sobre este sustrato han prosperado utilidades y bibliotecas que añaden capacidades prácticas, como la gestión de la caché de datos, los reintentos o el tratamiento coherente de estados en la interfaz. En conjunto, puede decirse

que AJAX ha pasado de ser una técnica concreta a convertirse en un enfoque general para la comunicación cliente–servidor en la web.

AJAX frente a SPA y actualizaciones en tiempo real

Conviene distinguir AJAX de otros conceptos cercanos pero distintos. Una *Single Page Application* (SPA) organiza la aplicación en torno a una única página que gestiona su propio ciclo de vida y navegación interna. Aunque suele apoyarse en AJAX para obtener datos, la SPA define una arquitectura completa de interfaz, en la que el enrutado, el estado y la composición de vistas permanecen en el lado del cliente.

Por otra parte, las llamadas *actualizaciones en tiempo real* agrupan soluciones en las que el servidor puede enviar información de manera continua al navegador, como los *Server-Sent Events* o los *WebSockets*. La diferencia clave es el sentido de la iniciativa: en AJAX, el cliente solicita y el servidor responde; en los mecanismos de tiempo real, el servidor puede emitir novedades sin una petición previa específica del cliente. Elegir entre uno u otro enfoque depende del problema: para formularios, búsquedas puntuales o cargas parciales, el patrón AJAX suele resultar suficiente; para chats, paneles que reflejan cambios inmediatos o escenarios colaborativos, los canales persistentes son preferibles.

Casos de uso comunes de AJAX

Entre los usos habituales destacan:

1. **Autocompletado de búsqueda:** el sistema propone resultados a medida que el usuario escribe, sin interrumpir la interacción.

2. **Validación de formularios:** se comprueban datos relevantes antes del envío definitivo, lo que ahorra pasos innecesarios y reduce errores.

3. **Carga diferida y paginación:** se incorporan fragmentos de contenido bajo demanda, con impacto positivo en tiempos de respuesta y consumo de red.

4. **Paneles y reportes actualizados:** se refrescan indicadores de forma periódica sin recargar la página.

5. **Integración con frameworks modernos:** se combinan las peticiones con patrones de gestión de estado para mantener coherencia entre datos y vistas.

En todos los casos, la idea rectora es la misma: realizar transferencias ajustadas a la necesidad, comunicar con claridad el estado de la operación y preservar la continuidad de la interfaz.

8.1.2 Requisitos previos

Antes de abordar implementaciones, es conveniente repasar tres áreas que sirven de fundamento: el modelo de documento del navegador, la asincronía en JavaScript y el funcionamiento elemental de HTTP.

HTML y el modelo DOM

La actualización parcial de la interfaz exige conocer cómo se representa una página en el navegador. El *Document Object Model* permite seleccionar, crear y modificar elementos para reflejar los datos que llegan del servidor. Entender ese modelo es esencial para insertar resultados, mostrar mensajes de estado o retirar contenidos obsoletos con naturalidad.

JavaScript moderno: promesas y asincronía

La comunicación asíncrona se apoya en promesas y en la sintaxis `async/await`, que ofrecen una forma lineal y legible de expresar operaciones que no concluyen de inmediato. Dominar estos recursos ayuda a escribir código predecible, a diferenciar los errores derivados de la red de los debidos a la lógica de la aplicación y a mantener el control del flujo sin bloquear la interfaz.

Fundamentos de HTTP

Toda solicitud AJAX se traduce en un intercambio HTTP con un método, unas cabeceras y un posible cuerpo. Conocer el significado de los métodos más habituales, la semántica de los códigos de estado y el papel de cabeceras como `Content-Type`, `Accept` o `Authorization` permite diagnosticar problemas con rapidez y dialogar con el servidor de acuerdo con expectativas claras.

Intercambio de datos: JSON, XML y `FormData`

JSON es hoy el formato predominante por su concisión y legibilidad. Sin embargo, el conocimiento de XML sigue siendo útil en entornos con sistemas heredados. La interfaz `FormData` facilita el envío de formularios, incluidos archivos, sin construir manualmente la carga útil, y resulta especialmente práctica en procesos de subida y validación progresiva.

Herramientas complementarias

Para trabajar con solvencia conviene familiarizarse con las herramientas del propio navegador (especialmente las pestañas de *Red* y *Consola*), con clientes externos que permiten explorar una API de forma aislada (`curl`, Postman o Insomnia) y con un servidor local de pruebas. Este conjunto permite observar el intercambio real, reproducir escenarios de error y afianzar el criterio técnico antes de integrar la lógica en la interfaz.

En suma, entender AJAX como un patrón de comunicación y de actualización gradual de la interfaz aporta un marco sólido para el desarrollo web actual. Sobre estas bases se asientan los temas que siguen: el uso de las APIs modernas, la relación con el DOM, los aspectos de seguridad y las decisiones de diseño que afectan a rendimiento y mantenibilidad.

8.2 Fundamentos Web

Antes de abordar la implementación de AJAX conviene repasar el marco en el que opera: por un lado, el protocolo HTTP, que articula el intercambio de información en la web; por otro, las políticas de seguridad del navegador, que condicionan qué datos pueden solicitarse

y bajo qué reglas. Comprender ambos planos permite anticipar el comportamiento de las aplicaciones y diagnosticar con criterio incidencias frecuentes durante el desarrollo.

Esta sección presenta una síntesis de los aspectos esenciales de HTTP y de las restricciones impuestas por el navegador, con el propósito de ofrecer una base sólida para el uso posterior de AJAX.

8.2.1 HTTP en 20 minutos

HTTP (*HyperText Transfer Protocol*) define el diálogo entre un cliente (el navegador u otro programa) y un servidor. Toda petición se expresa mediante un método, puede incluir cabeceras y, en su caso, un cuerpo con datos; toda respuesta devuelve un código de estado y, habitualmente, un contenido.

Métodos HTTP: la gramática del intercambio

Los métodos indican la intención de la solicitud:

- **GET:** recupera información sin alterar el estado del servidor; adecuado para consultas y listados.
- **POST:** envía datos para crear un recurso; típico en formularios o subidas de archivos.
- **PUT:** reemplaza por completo un recurso existente.
- **PATCH:** actualiza parcialmente un recurso.
- **DELETE:** solicita la eliminación de un recurso identificado.

La respuesta del servidor incluye un código que sintetiza el resultado. Algunos de uso común son: `200 OK` (operación correcta), `201 Created` (recurso creado), `204 No Content` (sin cuerpo de respuesta), `400 Bad Request` (petición mal formada), `401 Unauthorized` (requiere autenticación), `403 Forbidden` (falta de permisos), `404 Not Found` (recurso inexistente) y `500 Internal Server Error` (error en el servidor). Conocer su significado orienta la interpretación de las respuestas y agiliza la detección de problemas.

Cabeceras HTTP: el contexto de la comunicación

Las cabeceras acompañan a petición y respuesta y aportan información sobre formato, idioma, autenticación o almacenamiento en caché. Entre las más relevantes para AJAX destacan:

- `Accept`: tipos de contenido que el cliente está dispuesto a recibir (por ejemplo, `application/json`).
- `Content-Type`: tipo de datos enviados en el cuerpo de la solicitud (por ejemplo, `application/json` o `multipart/form-data`).
- `Authorization`: credenciales o tokens de acceso cuando la API lo exige.

Ejemplo ilustrativo con `fetch`:

```
fetch("https://api.ejemplo.com/usuarios", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "Authorization": "Bearer 123abc456def"
  },
  body: JSON.stringify({ nombre: "César", edad: 30 })
})
.then(r => r.json())
.then(data => console.log(data));
```

Caching: aprovechar respuestas previas

Guardar temporalmente respuestas evita repetir descargas y mejora la percepción de rapidez. Para ello se emplean, entre otras, estas cabeceras:

- `Cache-Control`: pauta temporal y condiciones del almacenamiento (*p. ej.*, `max-age=3600`).
- `ETag`: identificador de la versión de un recurso; permite validar si ha cambiado.
- `Last-Modified`: fecha de última modificación; posibilita consultas condicionales.

Si el cliente envía `If-None-Match` con el `ETag` almacenado y el recurso no ha variado, el servidor responde `304 Not Modified` y se evita transferir de nuevo el contenido.

Un intercambio completo, a modo de ejemplo

```
GET /api/usuarios HTTP/1.1
Host: api.ejemplo.com
Accept: application/json
Authorization: Bearer 123abc456def

HTTP/1.1 200 OK
Content-Type: application/json
Cache-Control: max-age=600
ETag: "v2.1"

[{"id":1,"nombre":"Ana"},{"id":2,"nombre":"Luis"}]
```

Este esquema resume el ciclo habitual: el cliente formula la petición, el servidor responde y el navegador interpreta los datos para actualizar la interfaz.

8.2.2 Políticas del navegador

Los navegadores aplican medidas destinadas a proteger a la persona usuaria y a preservar la integridad del sitio. Dichas medidas influyen directamente en las peticiones AJAX, en especial cuando intervienen varios orígenes.

Same-Origin Policy (SOP)

La *política de mismo origen* restringe el acceso de un script a recursos que no comparten protocolo, dominio y puerto con la página que lo ejecuta. Esta limitación evita que un sitio lea información sensible de otro sin permiso. Cuando la aplicación necesita comunicarse con una API en un dominio distinto, entra en juego el mecanismo CORS.

CORS: autorización entre orígenes

CORS (*Cross-Origin Resource Sharing*) permite que un servidor autorice, de forma explícita, el acceso desde determinados orígenes. Para ello, la respuesta incluye cabeceras como:

```
Access-Control-Allow-Origin: https://miapp.com
Access-Control-Allow-Methods: GET, POST, PUT
Access-Control-Allow-Headers: Content-Type, Authorization
```

Si la solicitud no es "simple" (por ejemplo, utiliza `PUT` o cabeceras personalizadas), el navegador realiza antes una verificación previa (`OPTIONS`) para confirmar que la operación está permitida. La ausencia o discordancia de estas cabeceras suele estar detrás de los errores conocidos como *CORS errors*.

Cookies, credenciales y sesiones

El mantenimiento de la sesión se apoya, con frecuencia, en cookies. El atributo `SameSite` regula si se envían en peticiones procedentes de otros orígenes: `Strict` (solo mismo origen), `Lax` (ciertas navegaciones) y `None` (requiere `Secure`). Cuando la aplicación necesita incluir credenciales en `fetch`, debe indicarlo de forma expresa:

```
fetch("https://api.miapp.com/datos", { credentials: "include" });
```

Para que el navegador las tenga en cuenta en un contexto entre orígenes, el servidor debe responder, además, con `Access-Control-Allow-Credentials: true` y no puede utilizar el comodín `*` en `Access-Control-Allow-Origin`.

En arquitecturas actuales son frecuentes los esquemas basados en tokens (por ejemplo, JWT) enviados en la cabecera `Authorization`. Este enfoque reduce la dependencia de cookies, pero exige proteger el lado cliente frente a inyecciones de código.

En suma, el trabajo con AJAX se beneficia de una comprensión clara del protocolo HTTP y de las normas de seguridad del navegador. Estas nociones permiten planificar las peticiones con sentido, interpretar las respuestas sin ambigüedades y resolver con agilidad incidencias relacionadas con permisos, autenticación y caché. Sobre esta base se construirán, en las siguientes secciones, las herramientas y pautas de implementación necesarias para un uso eficaz y responsable de AJAX.

8.3 Herramientas y entorno

Para trabajar con solvencia en AJAX no basta con conocer los conceptos: es necesario disponer de un entorno de apoyo que permita observar, ensayar y depurar el intercambio con el servidor. Esta sección presenta un conjunto de herramientas que facilitan esa labor,

desde las utilidades integradas en el navegador hasta clientes externos de API y servidores locales para pruebas. El objetivo es contar con un marco práctico para analizar solicitudes y respuestas HTTP, revisar cabeceras, simular fallos y generar datos ficticios (*mock data*) sin depender de servicios de terceros.

8.3.1 Las DevTools del navegador

Los navegadores actuales (Chrome, Firefox, Edge o Safari) incorporan un panel de desarrollo (*DevTools*) orientado a inspeccionar el documento, depurar scripts, medir el rendimiento y, en particular, estudiar el tráfico de red. Su uso habitual permite entender lo que ocurre durante una interacción AJAX y localizar con rapidez el origen de un problema.

Pestaña *Network*

La pestaña **Network** muestra, en tiempo real, todas las solicitudes HTTP que realiza la página: recursos estáticos y peticiones asíncronas. Para cada entrada se registran, entre otros, el método empleado, la URL, el código de estado, el tiempo de respuesta, las cabeceras de petición y de respuesta, el cuerpo enviado (*payload*) y el contenido devuelto. Por ejemplo, al ejecutar:

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then(resp => resp.json())
  .then(data => console.log(data));
```

en *Network* se apreciará la petición `/posts/1`, su código `200`, el tipo de contenido (`application/json`) y el cuerpo de la respuesta. Esta visión directa del intercambio ayuda a identificar desajustes de formato, errores de autenticación o discrepancias en cabeceras.

Sugerencia. La opción `Preserve log` evita perder el histórico tras una recarga, algo útil cuando las peticiones se producen durante el arranque de la página.

Pestaña *Sources*

La pestaña **Sources** permite navegar por los archivos JavaScript y establecer puntos de interrupción para seguir la ejecución paso a paso. Es especialmente valiosa cuando una petición falla y conviene conocer el estado del programa en el momento del error. En una función como:

```
async function cargarUsuarios() {
  const res = await fetch("/api/usuarios");
  const data = await res.json();
  mostrarLista(data);
}
```

puede colocarse un punto de ruptura en la línea del `fetch()` para inspeccionar la respuesta antes de su uso, revisar variables locales y analizar excepciones.

Pestaña *Application*

La pestaña **Application** ofrece acceso a los datos persistidos en el navegador: cookies, `localStorage`, `sessionStorage`, `IndexedDB` y el *cache* de red. Es de ayuda al trabajar con autenticación y sesiones, pues permite comprobar el estado de un token, limpiar almacenamientos y reproducir escenarios desde un contexto limpio. Si aparece un `401 Unauthorized`, conviene verificar en *Application* → *Cookies* la vigencia del token y la correspondencia de dominio.

8.3.2 Clientes de API: curl, Postman e Insomnia

Las *DevTools* resultan idóneas dentro del navegador, pero es recomendable complementarlas con herramientas externas para diseñar y automatizar peticiones. `curl` ofrece una interfaz de línea de comandos; Postman e Insomnia, una interfaz gráfica centrada en colecciones reutilizables.

Uso básico de `curl`

`curl` permite enviar solicitudes y observar las respuestas desde la terminal, sin las restricciones del navegador (por ejemplo, CORS):

GET sencillo

```
curl -X GET https://jsonplaceholder.typicode.com/posts/1
```

POST con JSON

```
curl -X POST https://api.miapp.com/usuarios \
     -H "Content-Type: application/json" \
     -d '{"nombre":"César","edad":30}'
```

Autenticación con token

```
curl -H "Authorization: Bearer abc123" https://api.miapp.com/datos
```

Estos ensayos permiten aislar si un problema es propio del servidor o se origina en el entorno del navegador.

Postman e Insomnia

Postman e Insomnia facilitan construir, guardar y compartir colecciones de peticiones. Permiten definir cabeceras, parámetros, cuerpos de solicitud, variables de entorno y flujos de autenticación (por ejemplo, OAuth2 o JWT). Su uso típico consiste en: crear la petición, ajustarla con las cabeceras necesarias, ejecutarla y guardar el resultado como punto de partida para futuras pruebas. Este hábito contribuye a documentar el comportamiento de la API y a reducir incertidumbres antes de integrar la lógica en el frontend.

8.3.3 Servidor de pruebas local

Disponer de un servidor local simplifica la práctica con AJAX: evita depender de servicios externos, permite controlar las respuestas y reduce la fricción provocada por políticas entre orígenes.

Servidor sencillo con Node.js (Express)

```
const express = require('express');
const app = express();
app.use(express.json());

app.get('/usuarios', (req, res) => {
  res.json([{id: 1, nombre: "Ana"}, {id: 2, nombre: "Luis"}]);
});

app.post('/usuarios', (req, res) => {
  res.status(201).json({mensaje: "Usuario creado", data: req.body});
});

app.listen(3000, () => console.log('http://localhost:3000'));
```

Con esta base pueden ensayarse peticiones desde una página HTML o desde Postman, observando el resultado en el navegador y en la consola del servidor.

Servidor en Python con Flask

```
from flask import Flask, jsonify, request
app = Flask(__name__)

@app.route('/usuarios', methods=['GET'])
def get_usuarios():
    return jsonify([{"id":1, "nombre":"Ana"}, {"id":2, "nombre":"Luis"}])

@app.route('/usuarios', methods=['POST'])
def create_usuario():
    data = request.get_json()
    return jsonify({"mensaje":"Usuario creado", "data":data}), 201

if __name__ == '__main__':
    app.run(port=5000, debug=True)
```

Este entorno permite comprobar, de forma controlada, la codificación de cabeceras, los formatos de datos y la gestión de errores.

Datos simulados (*mock data*)

Cuando el backend no está disponible o los datos reales no pueden exponerse, es útil recurrir a servicios de simulación. Algunas opciones habituales son `jsonplaceholder.typicode.com`, `mockapi.io` o el uso de `msw` (Mock Service Worker) para interceptar solicitudes en el propio navegador. Con ello se pueden reproducir respuestas previsibles, introducir retrasos y ensayar errores sin afectar a sistemas en producción.

8.3.4 Buenas prácticas del entorno de trabajo

- Mantener un entorno local aislado y reproducible; si es posible, con contenedores o entornos virtuales.

- Trabajar con las *DevTools* abiertas y limpiar periódicamente la caché para evitar resultados engañosos.
- Documentar las peticiones de prueba y sus variantes en colecciones de Postman o Insomnia.
- Utilizar HTTPS en el entorno local cuando se ensayen cookies seguras o políticas CORS realistas.
- Registrar peticiones y respuestas en el servidor de pruebas para analizar tiempos, tamaños y códigos de estado.

En conjunto, estas herramientas constituyen un apoyo decisivo para aprender y aplicar AJAX con criterio. Permiten observar el intercambio real, reproducir escenarios representativos y avanzar desde la idea general hacia una práctica informada y verificable.

8.4 AJAX clásico con `XMLHttpRequest`

Antes de la incorporación generalizada de `fetch()`, las aplicaciones web utilizaban el objeto `XMLHttpRequest` (XHR) para comunicarse de forma asíncrona con el servidor. Aunque su sintaxis resulte hoy más extensa, conocer su funcionamiento aporta contexto histórico, ayuda a mantener sistemas heredados y permite comprender el origen de muchos patrones todavía vigentes.

8.4.1 Primeros pasos

`XMLHttpRequest` fue la base del paradigma que hoy denominamos AJAX. Su introducción posibilitó enviar y recibir datos sin recargar la página completa, con un modelo centrado en eventos y en un ciclo de vida bien definido. A continuación se presenta el flujo mínimo de uso y el significado de sus etapas internas.

Creación y estructura básica

El uso elemental de XHR sigue cuatro pasos: crear la instancia, abrir la conexión con método y URL, registrar un manejador para los cambios de estado y, por último, enviar la petición.

Ejemplo básico

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://jsonplaceholder.typicode.com/posts/1");
xhr.onreadystatechange = function() {
  if (xhr.readyState === 4 && xhr.status === 200) {
    var data = JSON.parse(xhr.responseText);
    console.log(data);
  }
};
xhr.send();
```

En este esquema, `open()` define la operación, `onreadystatechange` permite observar la evolución y `send()` inicia la comunicación. La respuesta se procesa cuando la operación concluye y el servidor ha indicado éxito.

El ciclo de vida: `readyState`

Durante la ejecución, XHR avanza por varios estados (`readyState`) y emite el evento `readystatechange` en cada transición. Conocer estos estados facilita la depuración y aclara el momento adecuado para leer la respuesta.

Valor	Constante	Descripción
0	`UNSENT`	Objeto creado; aún no se ha llamado a `open()`.
1	`OPENED`	Conexión abierta; pendiente de `send()`.
2	`HEADERS_RECEIVED`	Cabeceras de respuesta recibidas.
3	`LOADING`	Recepción progresiva del cuerpo de la respuesta.
4	`DONE`	Operación concluida (con éxito o con error).

Un registro sencillo de los cambios de estado ayuda a observar el flujo de ejecución:

```
xhr.onreadystatechange = function() {
  console.log("Estado:", xhr.readyState);
};
```

Métodos GET y POST

Solicitud GET Recupera información sin modificar el estado del servidor.

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://api.ejemplo.com/usuarios?rol=admin", true);
xhr.onload = function() {
  if (xhr.status === 200) {
    console.log(JSON.parse(xhr.responseText));
  }
};
xhr.send();
```

Solicitud POST Envía datos para crear recursos o procesar formularios.

```
var xhr = new XMLHttpRequest();
xhr.open("POST", "https://api.ejemplo.com/usuarios", true);
xhr.setRequestHeader("Content-Type", "application/json;charset=UTF-8");
xhr.onload = function() {
  if (xhr.status === 201) {
    console.log("Usuario creado:", xhr.responseText);
  }
};
xhr.send(JSON.stringify({ nombre: "César", edad: 30 }));
```

En envíos con cuerpo es preciso indicar un `Content-Type` coherente con el formato utilizado.

Tiempos de espera y cancelación

Para evitar esperas indefinidas, XHR permite fijar un límite temporal (`timeout`). Si se supera, se activa `ontimeout`. Además, la operación puede cancelarse con `abort()` cuando deja de tener sentido continuar (por ejemplo, si el usuario cambia de vista).

Ejemplo

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://api.ejemplo.com/lenta", true);
xhr.timeout = 5000; // milisegundos
xhr.ontimeout = function() {
  console.error("La solicitud ha superado el tiempo de espera");
};
xhr.send();

// Cancelación opcional en otro punto:
// xhr.abort();
```

Seguimiento de progreso en descarga y subida

El modelo basado en eventos permite informar del avance de una transferencia. Durante la descarga se utiliza `onprogress` en el propio objeto; durante la subida, los eventos análogos están disponibles en `xhr.upload`. Esta capacidad resulta útil para barras de progreso y mensajes de estado.

Progreso de descarga

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://api.ejemplo.com/archivo.zip", true);
xhr.responseType = "blob";

xhr.onprogress = function(e) {
  if (e.lengthComputable) {
    var pct = (e.loaded / e.total) * 100;
    console.log("Descargando:", pct.toFixed(2) + "%");
  }
};

xhr.onload = function() {
  if (xhr.status === 200) {
    console.log("Descarga completa");
  }
};

xhr.send();
```

Progreso de subida

```
var xhr = new XMLHttpRequest();
xhr.open("POST", "/upload", true);

xhr.upload.onprogress = function(e) {
  if (e.lengthComputable) {
    var pct = (e.loaded / e.total) * 100;
    console.log("Subiendo:", pct.toFixed(1) + "%");
  }
};

xhr.onload = function() {
  if (xhr.status === 200) console.log("Archivo enviado");
};

var formData = new FormData();
formData.append("archivo", archivoSeleccionado);
xhr.send(formData);
```

Gestión de errores y reintentos elementales

XHR ofrece eventos para distinguir fallos de red, tiempos de espera o cancelaciones. Conviene tratar explícitamente los códigos de estado HTTP y, cuando proceda, aplicar reintentos acotados.

Detección de errores

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "https://api.inexistente.com/datos", true);

xhr.onerror = function() {
  console.error("Fallo de red o servidor inaccesible");
};

xhr.onload = function() {
  if (xhr.status >= 400) {
    console.warn("Error HTTP:", xhr.status, xhr.statusText);
  } else {
    console.log("Respuesta:", xhr.responseText);
  }
};

xhr.send();
```

Reintento con límite

```
function solicitarConReintento(url, intentos = 3) {
  var xhr = new XMLHttpRequest();
  xhr.open("GET", url, true);
```

```
  xhr.onload = function() {
    if (xhr.status === 200) {
      console.log("Éxito:", xhr.responseText);
    } else if (intentos > 0) {
      solicitarConReintento(url, intentos - 1);
    }
  };

  xhr.onerror = function() {
    if (intentos > 0) {
      solicitarConReintento(url, intentos - 1);
    } else {
      console.error("No se pudo completar la solicitud");
    }
  };

  xhr.send();
}
```

Este enfoque resulta adecuado ante fallos temporales, siempre que la operación sea segura de repetir.

Limitaciones del modelo clásico

Aunque XHR continúa disponible y operativo, presenta limitaciones que explican la adopción de `fetch()`:

- Ausencia de soporte nativo para promesas, lo que complica el manejo ordenado de la asincronía.
- Propensión a estructuras de callbacks difíciles de mantener.
- Sintaxis más extensa y menos expresiva para flujos encadenados.
- Tratamiento de errores menos uniforme en entornos antiguos.

Aun así, comprender XHR permite mantener aplicaciones existentes y entender la evolución hacia interfaces modernas basadas en promesas y `async/await`.

En suma, `XMLHttpRequest` inauguró la actualización parcial de la interfaz en la web y estableció el patrón de comunicación que hoy sigue vigente. Conocer su ciclo de vida, sus eventos y sus límites prepara el terreno para la transición a `fetch()`, que recupera estas ideas con una sintaxis más clara y una integración más natural con la asincronía de JavaScript.

8.5 La API moderna: `fetch()`

Con la estandarización de ECMAScript 2015 (ES6), la API `fetch()` se consolidó como la vía preferente para realizar solicitudes HTTP desde el navegador. Frente a `XMLHttpRequest`,

propone una interfaz más clara y coherente con el modelo de promesas, lo que facilita expresar la asincronía de forma legible e integra con naturalidad funcionalidades actuales como `AbortController`, *Service Workers* y *Streams*.

La idea central es sencilla: una solicitud devuelve una promesa que, al resolverse, ofrece un objeto de respuesta. A partir de ahí, el código puede transformar el cuerpo al formato deseado, comprobar el estado devuelto por el servidor y, en su caso, tratar los errores de forma explícita. Este enfoque favorece un estilo de programación directo y predecible.

8.5.1 Uso esencial

Sintaxis básica y flujo de trabajo

El uso elemental de `fetch()` admite una forma concisa:

```
fetch(url, opciones)
  .then(respuesta => /* procesar */)
  .catch(error => /* tratar fallo de red */);
```

El primer argumento es la URL; el segundo, opcional, permite indicar método, cabeceras y cuerpo. Por defecto, `fetch()` realiza una petición `GET`.

Ejemplo básico

```
fetch("https://jsonplaceholder.typicode.com/posts/1")
  .then(r => r.json())
  .then(d => console.log("Datos recibidos:", d))
  .catch(e => console.error("Error de red:", e));
```

El método `response.json()` transforma el cuerpo en un objeto JavaScript. Existen, además, otras funciones de lectura:

- `response.text()`: para contenido textual.
- `response.blob()`: para datos binarios (imágenes, PDF).
- `response.formData()`: para formularios.
- `response.arrayBuffer()`: para tratamiento de bajo nivel.

Descarga de una imagen

```
fetch("/imagenes/logo.png")
  .then(r => r.blob())
  .then(b => {
    const url = URL.createObjectURL(b);
    document.getElementById("logo").src = url;
  });
```

Promesas y `async/await`

Las promesas permiten encadenar operaciones, pero cuando la lógica crece resulta más claro recurrir a `async/await`, que ofrece una lectura lineal.

Ejemplo con `async/await`

```
async function cargarDatos() {
  try {
    const r = await fetch("https://api.ejemplo.com/usuarios");
    const datos = await r.json();
    console.log(datos);
  } catch (e) {
    console.error("Error de red:", e);
  }
}
```

Diferenciar error de red y error HTTP

`fetch()` sólo rechaza la promesa cuando existe un fallo de red o la petición no puede completarse. Si el servidor responde con un error HTTP (por ejemplo, `404` o `500`), la promesa se resuelve y corresponde al código verificarlo mediante `response.ok` y `response.status`.

Patrón recomendado

```
fetch("https://api.ejemplo.com/recurso")
  .then(r => {
    if (!r.ok) throw new Error("HTTP " + r.status);
    return r.json();
  })
  .then(datos => /* usar datos */)
  .catch(e => console.error("Fallo:", e.message));
```

Envío de datos: POST, PUT y DELETE

El segundo parámetro permite especificar método, cabeceras y cuerpo. Para JSON, conviene indicar el tipo de contenido y serializar el objeto:

```
fetch("https://api.ejemplo.com/usuarios", {
  method: "POST",
  headers: { "Content-Type": "application/json" },
  body: JSON.stringify({ nombre: "César", edad: 31 })
})
  .then(r => r.json())
  .then(d => console.log("Respuesta:", d));
```

8.5.2 Temas avanzados

`AbortController` y cancelación

En interfaces dinámicas resulta habitual cancelar solicitudes obsoletas, por ejemplo al cambiar un filtro. `AbortController` permite interrumpir la petición de forma controlada.

Ejemplo

```
const controller = new AbortController();

fetch("https://api.ejemplo.com/datos", { signal: controller.signal })
  .then(r => r.json())
  .then(d => console.log(d))
  .catch(e => {
    if (e.name === "AbortError") {
      console.log("Solicitud cancelada");
    } else {
      console.error("Error:", e);
    }
  });

// Cancelar más tarde:
controller.abort();
```

Formularios y archivos con `FormData`

`FormData` facilita el envío de formularios —incluidos archivos— sin construir manualmente el cuerpo ni fijar cabeceras complejas.

Ejemplo

```
const form = document.querySelector("#mi-formulario");
const data = new FormData(form);

fetch("/subir", { method: "POST", body: data })
  .then(r => r.json())
  .then(d => console.log("Subida:", d))
  .catch(e => console.error("Error:", e));
```

Procesamiento por *streams* (visión general)

`fetch()` permite leer respuestas de gran tamaño como flujo, lo que evita cargar todo el contenido en memoria y posibilita un tratamiento progresivo.

Ejemplo simplificado

```
const r = await fetch("/grande.json");
const reader = r.body.getReader();

let recibidos = 0;
while (true) {
  const { done, value } = await reader.read();
  if (done) break;
  recibidos += value.length;
  console.log("Bytes:", recibidos);
}
```

Credenciales y cookies

Por defecto, las solicitudes entre orígenes no incluyen credenciales. Para enviarlas es preciso indicarlo de modo explícito y coordinarlo con la configuración del servidor:

```
fetch("https://api.segura.com/perfil", { credentials: "include" });
```

El servidor debe permitirlo con `Access-Control-Allow-Credentials: true` y declarar un origen concreto en `Access-Control-Allow-Origin`.

Autenticación con tokens

Es habitual emplear tokens (por ejemplo, JWT) en la cabecera `Authorization` para mantener la sesión sin recurrir a cookies:

```
const token = localStorage.getItem("jwt");
fetch("https://api.ejemplo.com/datos", {
  headers: { "Authorization": `Bearer ${token}` }
})
  .then(r => r.json())
  .then(d => console.log(d));
```

En conjunto, `fetch()` ofrece una base moderna y clara para la comunicación asíncrona en la web. Su integración con `async/await`, la posibilidad de cancelar solicitudes y el soporte de flujos y formularios lo convierten en un recurso adecuado para aplicaciones actuales, tanto en páginas tradicionales como en arquitecturas de una sola página. La sección siguiente se centrará en cómo trasladar estos datos a la interfaz de manera progresiva y coherente con la experiencia de uso.

8.6 Manipulación del DOM y UX

Una vez dominada la realización de solicitudes asíncronas con `fetch()` o `XMLHttpRequest`, el paso siguiente consiste en integrar los resultados de forma clara y coherente en la interfaz. La experiencia de uso (UX) exige que la aplicación comunique en todo momento qué está ocurriendo: que muestre progreso mientras espera, que confirme el éxito al concluir y que explique, con precisión, los errores cuando se producen. Esta sección aborda la actualización parcial del documento, la gestión explícita de estados y los criterios de accesibilidad que garantizan una interacción inclusiva.

8.6.1 Actualizaciones parciales y estados de carga/éxito/error

El objetivo de AJAX es actualizar porciones acotadas de la página sin recargar el documento completo. El flujo recomendado consta de tres pasos: anunciar que se está cargando, realizar la solicitud y, por último, sustituir el fragmento del DOM con la información recibida, tratando de forma explícita tanto el éxito como el error.

Actualización parcial del contenido

```
async function cargarUsuarios() {
  const zona = document.querySelector("#usuarios");
  zona.innerHTML = "<p>Cargando usuarios. . . </p>";

  try {
    const r = await fetch("/api/usuarios");
    if (!r.ok) throw new Error("HTTP " + r.status);

    const datos = await r.json();
    zona.innerHTML = ""; // elimina el estado previo

    for (const u of datos) {
      const li = document.createElement("li");
      li.textContent = `${u.nombre} (${u.email})`;
      zona.appendChild(li);
    }
  } catch (e) {
    zona.innerHTML =
     `<p class="error">No se pudieron cargar los usuarios: ${e.message}</p>`;
  }
}
```

Este patrón hace visibles los estados intermedios y evita dejar a la persona usuaria sin referencia durante la espera.

Estados visibles y coherentes

Es conveniente representar los estados con clases o atributos semánticos, para separar la lógica de la presentación:

- `.loading`: operación en curso.
- `.success`: operación concluida correctamente.
- `.error`: operación fallida.

```
const btn = document.querySelector("#actualizar");

btn.addEventListener("click", async () => {
  btn.classList.add("loading");
  btn.disabled = true;

  try {
    const r = await fetch("/api/actualizar");
    if (!r.ok) throw new Error();
    btn.classList.replace("loading", "success");
    btn.textContent = "Actualizado";
  } catch {
```

```
    btn.classList.replace("loading", "error");
    btn.textContent = "Error al actualizar";
  } finally {
    setTimeout(() => {
      btn.classList.remove("loading", "success", "error");
      btn.textContent = "Actualizar";
      btn.disabled = false;
    }, 2000);
  }
});
```

Este enfoque reduce interacciones duplicadas, comunica el estado y devuelve la interfaz a la normalidad sin ambigüedades.

8.6.2 Indicadores de progreso, bloqueo temporal y *Optimistic UI*

Indicadores de progreso

En operaciones superiores al segundo, conviene mostrar un indicio de actividad. La forma exacta (barra, icono o texto) es secundaria respecto a la claridad del mensaje.

```
<div id="spinner" class="oculto"></div>

<style>
#spinner {
  border: 4px solid #eee;
  border-top: 4px solid #333;
  border-radius: 50%;
  width: 40px; height: 40px;
  animation: girar 1s linear infinite;
}
@keyframes girar { from {transform:rotate(0)} to {transform:rotate(360deg)} }
.oculto { display: none; }
</style>

<script>
const sp = document.getElementById("spinner");
async function cargarDatos() {
  sp.classList.remove("oculto");
  const r = await fetch("/api/datos");
  const d = await r.json();
  sp.classList.add("oculto");
  console.log(d);
}
</script>
```

Bloqueo temporal de acciones

Para evitar envíos repetidos mientras una petición está en curso, puede deshabilitarse el control implicado y restablecerlo al finalizar:

```
const enviar = document.querySelector("#enviar");

async function enviarFormulario() {
  enviar.disabled = true;
  enviar.textContent = "Enviando. . . ";
  try {
    const r = await fetch("/api/form", { method: "POST" });
    enviar.textContent = r.ok ? "Enviado" : "Error";
  } finally {
    setTimeout(() => {
      enviar.disabled = false;
      enviar.textContent = "Enviar";
    }, 2000);
  }
}
```

Optimistic UI

La interfaz optimista actualiza la vista de inmediato y corrige después si el servidor informa de un fallo. Este patrón mejora la percepción de fluidez, siempre que exista un mecanismo claro de reversión.

```
const botonLike = document.querySelector("#like");
let liked = false;

botonLike.addEventListener("click", async () => {
  liked = !liked;
  botonLike.textContent = liked ? "Quitar me gusta" : "Me gusta";
  try {
    const r = await fetch("/api/like", { method: "POST" });
    if (!r.ok) throw new Error();
  } catch {
    liked = !liked; // revierte el estado
    botonLike.textContent = liked ? "Quitar me gusta" : "Me gusta";
  }
});
```

8.6.3 Accesibilidad: roles ARIA, foco y mensajes

Las actualizaciones dinámicas deben ser perceptibles también para quien navega con teclado o lector de pantalla. La accesibilidad no es un añadido, sino un requisito de calidad.

Regiones vivas y mensajes

Las regiones marcadas con `aria-live` informan de los cambios sin necesidad de recargar la página:

```
<div id="avisos" aria-live="polite"></div>

<script>
async function cargarMensaje() {
  const r = await fetch("/api/mensaje");
  const msg = await r.text();
  document.getElementById("avisos").textContent = msg;
}
</script>
```

El modo `polite` evita interrupciones bruscas; `assertive` debe usarse con moderación.

Gestión del foco

Tras una actualización sustancial, conviene trasladar el foco al elemento pertinente para mantener la continuidad de la interacción por teclado.

```
const resultado = document.querySelector("#resultado");
resultado.setAttribute("tabindex", "-1");
resultado.focus();
```

Confirmación accesible de operaciones

Los cambios relevantes deben anunciarse también mediante texto interpretable por tecnologías de apoyo:

```
<div id="estado" role="status" aria-live="polite" class="oculto-visualmente"></div>

<style>
.oculto-visualmente {
 position: absolute; left: -9999px; width: 1px; height: 1px; overflow: hidden;
}
</style>

<script>
async function guardarCambios() {
  const r = await fetch("/api/guardar", { method: "POST" });
  document.getElementById("estado").textContent =
   r.ok ? "Cambios guardados correctamente" : "No se pudieron guardar los cambios";
}
</script>
```

Recomendaciones generales

- Indicar `aria-busy="true"` en el contenedor mientras se actualiza.
- No depender exclusivamente del color para distinguir estados; añadir texto claro.
- Mantener un orden del DOM coherente tras insertar elementos.
- Proporcionar mensajes de error específicos y accionables.

En conjunto, la manipulación del DOM y el cuidado de la UX convierten a las respuestas asíncronas en interacciones comprensibles. Un tratamiento explícito de los estados, indicadores de progreso proporcionados y criterios de accesibilidad bien aplicados contribuyen a una experiencia continuada, predecible y adecuada para todas las personas usuarias.

8.7 Seguridad aplicada a AJAX

La seguridad constituye un requisito transversal en el desarrollo web. En aplicaciones con intercambio asíncrono de datos, la combinación de llamadas desde el navegador, manipulación del DOM y almacenamiento local introduce riesgos específicos que deben tratarse de forma explícita. Esta sección resume tres ámbitos clave —**CORS**, **CSRF** y **XSS**— y las cabeceras y pautas que contribuyen a mitigarlos.

8.7.1 CORS: control de orígenes cruzados

El mismo origen y sus límites

La *Same-Origin Policy* (SOP) restringe que scripts cargados desde un origen accedan a recursos de otro distinto (protocolo, dominio y puerto). Esta política protege frente a accesos no autorizados, pero también limita escenarios legítimos de arquitectura distribuida (por ejemplo, una interfaz en `localhost:3000` que consume una API en `localhost:5000`).

El papel de CORS

CORS (*Cross-Origin Resource Sharing*) define un mecanismo para autorizar, de forma controlada, solicitudes entre orígenes. La autorización se expresa mediante cabeceras en la respuesta del servidor:

- `Access-Control-Allow-Origin`: origen o lista de orígenes permitidos.
- `Access-Control-Allow-Methods`: métodos aceptados (p. ej., `GET`, `POST`, `PUT`).
- `Access-Control-Allow-Headers`: cabeceras personalizadas que el cliente puede enviar.
- `Access-Control-Allow-Credentials`: habilita el envío de credenciales entre orígenes.

Ejemplo en Express

```
app.use((req, res, next) => {
  res.setHeader("Access-Control-Allow-Origin", "https://miapp.com");
  res.setHeader("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE");
 res.setHeader("Access-Control-Allow-Headers", "Content-Type, Authorization");
  res.setHeader("Access-Control-Allow-Credentials", "true");
  next();
});
```

La solicitud previa (*preflight*)

Cuando la operación no es "simple" (por ejemplo, `PUT` o cabeceras personalizadas), el navegador realiza antes una solicitud `OPTIONS` para confirmar que la acción está permitida:

```
OPTIONS /api/usuarios
Origin: https://frontend.com
Access-Control-Request-Method: PUT
Access-Control-Request-Headers: Authorization
```

Si el servidor responde autorizando método, cabeceras y origen, el navegador ejecuta la solicitud principal; en caso contrario, la bloquea en el cliente.

Buenas prácticas CORS

- Evitar `Access-Control-Allow-Origin: *` cuando haya credenciales.
- Limitar orígenes a una lista blanca verificable.
- Responder sólo con métodos y cabeceras estrictamente necesarios.
- No confiar ciegamente en `Origin` recibido; validar también en lógica de negocio.

8.7.2 CSRF: falsificación de peticiones entre sitios

Concepto

El *Cross-Site Request Forgery* (CSRF) aprovecha que el navegador adjunta automáticamente credenciales (como cookies) al realizar una petición a un sitio en el que la persona usuaria ya está autenticada. Un tercero puede inducir, sin conocimiento del usuario, acciones no deseadas contra dicho sitio.

Ejemplo clásico

```
<!-- Página maliciosa -->
<form action="https://banco.com/transferir" method="POST">
  <input type="hidden" name="cuenta" value="999999">
  <input type="hidden" name="monto" value="5000">
  <button type="submit">Ver video</button>
</form>
```

Si la sesión está activa en `banco.com`, el navegador enviará la cookie y podría ejecutarse la transferencia.

Medidas de mitigación

Tokens CSRF El servidor emite un token aleatorio asociado a la sesión que debe acompañar cada petición de cambio de estado. En AJAX puede enviarse en una cabecera específica:

```
fetch("/api/actualizar", {
  method: "POST",
  headers: {
    "X-CSRF-Token": tokenCSRF,
    "Content-Type": "application/json"
  },
  body: JSON.stringify({ valor: 42 })
});
```

Cabeceras no simples El uso de cabeceras personalizadas obliga a *preflight*, lo que dificulta automatismos de sitios externos.

Atributo `SameSite` en cookies Controla cuándo se envían en contextos entre orígenes:

- `Strict`: sólo mismo origen.
- `Lax`: permite ciertos escenarios de navegación, no `POST` programáticos.
- `None`: entre orígenes; requiere `Secure`.

Ejemplo:

```
Set-Cookie: sessionid=abc123; SameSite=Lax; Secure; HttpOnly
```

Buenas prácticas CSRF

- Token único por sesión (y, si procede, por formulario).
- Cookies con `SameSite=Lax/Strict`, `Secure` y `HttpOnly`.
- Validación de `Origin`/`Referer` en operaciones sensibles.

8.7.3 XSS: inyección de código y sanitización

Definición

El *Cross-Site Scripting* (XSS) permite inyectar y ejecutar scripts en el contexto de la página. En flujos AJAX, el riesgo aparece al insertar en el DOM datos no confiables sin un filtrado previo.

Ejemplo vulnerable

```
fetch("/api/comentarios")
  .then(r => r.json())
  .then(lista => {
    const cont = document.getElementById("comentarios");
    lista.forEach(c => {
      cont.innerHTML += `<p>${c.texto}</p>`;
    });
  });
```

Si `c.texto` contiene `<script>...`, se ejecutará al renderizar.

Medidas de protección

Sanitización Depurar o codificar caracteres peligrosos antes de insertar en el DOM. Puede emplearse una librería consolidada (p. ej., DOMPurify):

```
const limpio = DOMPurify.sanitize(c.texto);
cont.innerHTML += `<p>${limpio}</p>`;
```

Inserción segura Priorizar `textContent` o `createTextNode()` frente a `innerHTML` cuando el contenido sea textual.

Validación en servidor El filtrado no es exclusivo del cliente; el backend debe validar y normalizar datos de entrada y salida.

Cabeceras de refuerzo

- **Content-Security-Policy (CSP)**: restringe orígenes y tipos de recursos. Ejemplo mínimo:

  ```
  Content-Security-Policy: default-src 'self'; script-src 'self'
  ```

- **X-Frame-Options**: evita *clickjacking* (`DENY` o `SAMEORIGIN`).
- **X-Content-Type-Options**: impide la detección heurística de tipos (`nosniff`).

Pautas adicionales

- No insertar HTML procedente de usuarios sin sanitización previa.
- Evitar construcciones peligrosas (como `eval()` o `Function`).
- Activar CSP en entornos de producción y revisar periódicamente su cobertura.
- Emplear herramientas de análisis (OWASP ZAP, Burp Suite) de forma regular.

En conjunto, la seguridad aplicada a AJAX depende de medidas coordinadas en cliente y servidor. CORS acota qué orígenes pueden interactuar con la API; los controles contra CSRF protegen operaciones que dependen de credenciales persistentes; y la prevención de XSS garantiza que los datos mostrados no introduzcan código ejecutable. Un tratamiento sistemático de estos aspectos reduce de forma significativa la superficie de ataque y mejora la fiabilidad global de la aplicación.

8.8 Rendimiento y escalabilidad

El rendimiento y la capacidad de crecer con la demanda son requisitos imprescindibles en aplicaciones que realizan un uso intensivo de AJAX. Cuando varias solicitudes se ejecutan en paralelo o se disparan con alta frecuencia, pueden saturar tanto al navegador como al servidor. Conviene, por tanto, aplicar técnicas que reduzcan la latencia, controlen el volumen de peticiones y aprovechen al máximo las capacidades de transporte y caché. Esta sección repasa patrones prácticos —*debounce*, *throttle*, paginación e *infinite scroll*—, estrategias de almacenamiento, y las ventajas que ofrecen HTTP/2 y HTTP/3, junto con una introducción a los *Service Workers*.

8.8.1 Control de frecuencia: *debounce* y *throttle*

Las interacciones con campos de búsqueda o con el desplazamiento de la página pueden generar decenas de eventos por segundo. Enviar una solicitud por cada evento degrada la experiencia y sobrecarga el backend.

Debounce: agrupar interacciones consecutivas

Debounce difiere la ejecución hasta que se detiene la actividad durante un intervalo dado. Es apropiado para búsquedas en vivo o validaciones inmediatas.

Ejemplo

```
function debounce(fn, delay) {
  let timer;
  return function(...args) {
    clearTimeout(timer);
    timer = setTimeout(() => fn.apply(this, args), delay);
  };
}

const buscar = debounce(async (texto) => {
  const r = await fetch(`/api/buscar?q=${encodeURIComponent(texto)}`);
  const datos = await r.json();
  mostrarResultados(datos);
}, 400);

document.querySelector("#busqueda").addEventListener("input", e => {
  buscar(e.target.value);
});
```

Throttle: limitar la cadencia

Throttle garantiza una ejecución como máximo cada cierto tiempo, aunque se produzcan múltiples invocaciones. Es útil en `scroll` o `resize`.

Ejemplo

```
function throttle(fn, intervalo) {
  let enCurso = false;
  return function(...args) {
    if (!enCurso) {
      fn.apply(this, args);
      enCurso = true;
      setTimeout(() => enCurso = false, intervalo);
    }
  };
}

window.addEventListener("scroll", throttle(() => {
  if (window.innerHeight + window.scrollY >= document.body.offsetHeight) {
    cargarMas();
  }
}, 1000));
```

8.8.2 Optimización de carga: paginación e *infinite scroll*

Paginación clásica

Dividir los resultados en páginas reduce la carga inicial y mejora la percepción de rapidez. El servidor debe acompañar la respuesta con metadatos (total de elementos, página actual, páginas restantes) para facilitar la navegación.

Ejemplo

```
async function cargarPagina(n) {
  const r = await fetch(`/api/usuarios?page=${n}`);
  const datos = await r.json();
  renderizarUsuarios(datos);
}
```

Infinite scroll

El desplazamiento infinito automatiza la carga de páginas al aproximarse al final de la lista. Debe aplicarse con cautela para evitar solicitudes redundantes y problemas de accesibilidad.

Ejemplo básico

```
let pagina = 1, cargando = false;

async function cargarMas() {
  if (cargando) return;
  cargando = true;
  const r = await fetch(`/api/usuarios?page=${++pagina}`);
  const datos = await r.json();
```

```
  renderizarUsuarios(datos);
  cargando = false;
}

window.addEventListener("scroll", () => {
 if (window.innerHeight + window.scrollY >= document.body.offsetHeight - 200) {
    cargarMas();
  }
});
```

Recomendaciones

- Mostrar un estado de "cargando..." al final de la lista.
- Evitar duplicados mediante identificadores o cursores.
- Ofrecer un botón "Cargar más" como alternativa y mejora de accesibilidad.
- Preferir `IntersectionObserver` a eventos de `scroll` cuando sea posible.

Batching: agrupar solicitudes

Agrupar varias consultas en una sola reduce latencia y sobrecarga de cabeceras.

Ejemplo

```
fetch("/api/productos?ids=1,2,3")
  .then(r => r.json())
  .then(datos => mostrar(datos));
```

Caching: reutilización inteligente

La caché evita descargas repetidas y mejora la resiliencia ante cortes de red. Puede gestionarse a distintos niveles:

- **Caché del navegador**: mediante `Cache-Control`, `ETag` y validaciones condicionales.
- **Caché en cliente**: estructuras en memoria o almacenamiento local (`localStorage`, `sessionStorage`) para datos poco volátiles.
- **Caché de red**: a través de *Service Workers* o proxies.

Caché simple en memoria

```
const cache = {};
async function obtener(url) {
  if (cache[url]) return cache[url];
  const r = await fetch(url);
  const data = await r.json();
  cache[url] = data;
  return data;
}
```

8.8.3 Protocolos modernos: HTTP/2 y HTTP/3

HTTP/2: multiplexación y compresión

HTTP/2 permite enviar múltiples solicitudes simultáneas sobre una sola conexión, comprime cabeceras y ofrece prioridad de recursos. Estas mejoras reducen esperas asociadas a la apertura de conexiones y benefician, en particular, a aplicaciones con muchas peticiones breves.

HTTP/3: transporte sobre QUIC

HTTP/3, basado en QUIC, mejora la latencia en redes móviles o inestables: evita bloqueos por pérdida de paquetes, reanuda conexiones con menor coste y resulta eficiente para flujos cortos y frecuentes. Su adopción, ya extendida en navegadores modernos, ofrece ventajas tangibles sin cambios en el código del cliente.

8.8.4 *Service Workers* y *Cache Storage*: una introducción

Concepto

Los *Service Workers* actúan como un intermediario entre la aplicación y la red. Pueden interceptar peticiones, decidir su origen (red o caché) y aplicar estrategias de actualización como *stale-while-revalidate*. Requieren HTTPS y una política de versionado cuidadosa.

Registro

```
if ("serviceWorker" in navigator) {
  navigator.serviceWorker.register("/sw.js")
    .then(() => console.log("Service Worker registrado"))
    .catch(err => console.error("Error:", err));
}
```

Intercepción y caché

```
// sw.js
self.addEventListener("fetch", event => {
  event.respondWith(
    caches.match(event.request)
      .then(r => r || fetch(event.request))
  );
});
```

Ventajas

- Respuestas más rápidas al servir desde caché.
- Funcionamiento básico sin conexión para recursos previamente almacenados.
- Posibilidad de sincronización en segundo plano y notificaciones.

Precauciones

- Gestionar el versionado para evitar datos obsoletos.
- Desactivar temporalmente el *Service Worker* al depurar problemas de red.
- Evaluar qué datos conviene cachear y durante cuánto tiempo.

En conjunto, el rendimiento en AJAX depende de decisiones coordinadas en cliente y servidor: controlar la cadencia de peticiones (*debounce*/*throttle*), reducir cargas mediante paginación y *batching*, aprovechar la caché y apoyarse en las mejoras del transporte (HTTP/2 y HTTP/3). La incorporación de *Service Workers* completa esta estrategia al permitir respuestas inmediatas y resiliencia frente a la red, sin sacrificar claridad ni mantenibilidad.

8.9 Diseño de APIs para AJAX

El éxito de una aplicación que consume datos de forma asíncrona no depende sólo del código del cliente: la calidad del diseño de la API es determinante. Una API bien construida debería ser intuitiva, consistente, segura y capaz de evolucionar sin quebrar la compatibilidad con integraciones existentes. En esta sección se revisan los principios de diseño en REST, su comparación con GraphQL y un conjunto de prácticas sobre filtrado, paginación, idempotencia y versionado.

8.9.1 Principios de diseño REST

REST (*Representational State Transfer*) es un estilo arquitectónico ampliamente adoptado para exponer recursos a través de HTTP. Su valor reside en el empleo de convenciones claras: recursos identificables, métodos bien definidos y respuestas con semántica homogénea.

Recursos y rutas

En REST, los datos y sus relaciones se modelan como **recursos** accesibles mediante URLs estables.

Ejemplo de rutas RESTful

```
/api/usuarios
/api/usuarios/123
/api/pedidos/45/items
```

Pautas recomendadas

- Utilizar sustantivos en plural (`/usuarios`, `/productos`).
- Evitar acciones en la ruta (no `/crearUsuario`); las acciones se expresan con el método HTTP.
- Reflejar relaciones con subrutas cuando sea pertinente (`/pedidos/45/items`).

Verbos HTTP y operaciones CRUD

Los métodos de HTTP expresan la intención de la operación siguiendo el patrón CRUD:

Verbo	Acción	Ejemplo
`GET`	Consultar	`/api/usuarios`
`POST`	Crear	`/api/usuarios`
`PUT`	Reemplazar completamente	`/api/usuarios/123`
`PATCH`	Modificar parcialmente	`/api/usuarios/123`
`DELETE`	Eliminar	`/api/usuarios/123`

Códigos de estado HTTP

Los códigos de estado comunican el resultado de la operación sin obligar al cliente a analizar el cuerpo de la respuesta.

Usos frecuentes

- `200 OK`: operación correcta.
- `201 Created`: recurso creado.
- `204 No Content`: operación correcta sin cuerpo de respuesta.
- `400 Bad Request`: solicitud mal formada.
- `401 Unauthorized` / `403 Forbidden`: autenticación o permisos insuficientes.
- `404 Not Found`: recurso inexistente.
- `500 Internal Server Error`: error del servidor.

Respuesta ilustrativa

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "id": 123,
  "nombre": "César Sánchez",
  "email": "cesar@alquiber.es"
}
```

Representaciones y formatos

Aunque pueden emplearse varios formatos (JSON, XML, CSV), **JSON** es el predominante por su sencillez y su encaje con JavaScript.

Negociación de contenido

```
Accept: application/json
Content-Type: application/json
```

El cliente expresa su preferencia con `Accept` y el servidor responde con el `Content-Type` correspondiente.

8.9.2 GraphQL vs REST: cuándo elegir cada uno

Qué aporta GraphQL

GraphQL propone un único punto de acceso (`/graphql`) al que el cliente envía consultas declarativas indicando exactamente qué campos necesita.

Ejemplo de consulta

```
{
  usuario(id: 123) {
    nombre
    email
    pedidos { id total }
  }
}
```

Respuesta ejemplo

```
{
  "usuario": {
    "nombre": "César Sánchez",
    "email": "cesar@alquiber.es",
    "pedidos": [{ "id": 45, "total": 120.50 }]
  }
}
```

Comparación operativa

Ventajas de REST

- Modelo simple y ampliamente soportado.
- Integra bien mecanismos HTTP (caché, proxies, CORS).
- Adecuado para APIs públicas y contratos estables.

Ventajas de GraphQL

- Respuestas ajustadas a la necesidad (evita sobrecarga).
- Agrega datos de múltiples recursos en una sola consulta.
- Esquema tipado con capacidad de introspección.

Criterios de elección

- REST resulta preferible en superficies funcionales acotadas o cuando se valora la simplicidad y el soporte nativo de HTTP.
- GraphQL encaja mejor en interfaces ricas, con vistas muy variables o relaciones complejas, donde reducir viajes de red aporta valor tangible.

8.9.3 Filtros, ordenación, paginación e idempotencia

Filtros y parámetros de búsqueda

La API debe permitir seleccionar subconjuntos de datos de forma expresiva y combinable.

Ejemplos

```
/api/productos?categoria=vehiculos&precio_max=10000
/api/usuarios?edad[gte]=30&edad[lte]=50
```

Ordenación y selección de campos

Ofrecer control sobre el orden y los campos devueltos reduce carga y latencia.

Ejemplos

```
/api/pedidos?sort=-fecha
/api/usuarios?fields=nombre,email
```

El prefijo `-` indica orden descendente; `fields` acota la representación al mínimo útil.

Paginación y límites

La paginación controla el volumen por respuesta y mejora la experiencia.

Página y tamaño

```
/api/usuarios?page=3&limit=20
```

Paginación por cursor

```
/api/usuarios?after=MjAyNS0xMS0wNFQxMzo0NToyMw==
```

El cursor es preferible para flujos continuos o con *infinite scroll*.

Idempotencia y seguridad de métodos

La **idempotencia** garantiza que repetir una operación produce el mismo estado final. `GET`, `PUT` y `DELETE` deben ser idempotentes; `POST` no lo es.

Ejemplos

- `DELETE /usuarios/123` repetido mantiene el mismo resultado (el recurso no existe).
- `POST /usuarios` repetido crea recursos distintos.

El respeto a la idempotencia facilita reintentos automáticos ante fallos de red.

8.9.4 Versionado de APIs y compatibilidad hacia atrás

La evolución de una API debe planificarse para no interrumpir a los clientes existentes. El versionado explicita cambios incompatibles y permite transiciones ordenadas.

Estrategias de versionado

En la URL

```
/api/v1/usuarios
/api/v2/usuarios
```

Directa y clara; sencilla de adoptar.

Mediante cabeceras

```
Accept: application/vnd.miapp.v2+json
```

Separa la versión del espacio de rutas y favorece la negociación de contenido.

Por subdominio

```
https://v3.api.miapp.com/usuarios
```

Apropiado para operar versiones en paralelo a gran escala.

Compatibilidad hacia atrás

Para minimizar rupturas:

- Evitar la eliminación abrupta de campos; anunciar deprecaciones con antelación.
- Incluir una cabecera de aviso temporal:

  ```
  Deprecation: version=1.0; sunset="2026-01-01"
  ```

- Mantener documentación clara de cambios y ventanas de soporte.
- Preferir adiciones no disruptivas (nuevos campos opcionales frente a cambios de significado).

Buenas prácticas generales

- Consistencia en nombres, formatos, códigos de estado y estructura de errores.
- Documentación con OpenAPI/Swagger y ejemplos ejecutables.
- Validación de entrada y salida (por ejemplo, con JSON Schema).
- Inclusión de metadatos útiles (paginación, totales, enlaces de navegación).
- Uso de HTTPS, autenticación robusta (tokens o claves de API) y control de permisos.

En síntesis, el diseño de APIs orientadas a AJAX requiere equilibrio entre simplicidad del contrato, flexibilidad para consultas reales y capacidad de evolución. Un buen diseño —en REST o en GraphQL— debe ser predecible y autoexplicativo, optimizar el intercambio de datos y ofrecer un camino claro para crecer sin comprometer la compatibilidad ni la seguridad.

8.10 Librerías y frameworks

Aunque la API nativa `fetch()` cubre la mayoría de necesidades actuales, el ecosistema web ofrece librerías y marcos que simplifican la gestión de solicitudes, uniforman el tratamiento de errores y facilitan su integración en arquitecturas complejas. Su evolución abarca desde utilidades históricas basadas en `XMLHttpRequest` hasta soluciones orientadas a la gestión declarativa del estado remoto en interfaces reactivas.

8.10.1 Legado: `jQuery.ajax()` y el origen de AJAX moderno

Antes de la estandarización de `fetch()`, `jQuery` desempeñó un papel central al proporcionar una capa de compatibilidad entre navegadores y una sintaxis concisa para operaciones AJAX.

Contexto y motivación

`jQuery.ajax()` abstraía diferencias de implementación y ofrecía métodos abreviados como `$.get()`, `$.post()` o `$.getJSON()`, lo que permitió popularizar el uso de peticiones asíncronas en la web de su época.

Ejemplo

```
$.ajax({
  url: "/api/usuarios",
  method: "GET",
  success: function(data) { console.log("Usuarios:", data); },
  error: function(xhr) { console.error("Error:", xhr.statusText); }
});
```

Lectura actual Si bien su aportación histórica es indudable, hoy se considera una dependencia innecesaria en proyectos nuevos: no integra promesas nativas ni `async/await`, y añade peso a la aplicación. Mantener su conocimiento resulta útil para intervenir en sistemas heredados y planificar migraciones graduales hacia `fetch()` o alternativas modernas.

8.10.2 Axios: una interfaz uniforme y configurable

Axios es una librería basada en promesas, disponible en navegador y Node.js, orientada a ofrecer una experiencia homogénea: conversión automática de JSON, interceptores de solicitud y respuesta, instancias configurables y soporte de cancelación.

Rasgos distintivos

- Manejo coherente de errores: los códigos HTTP no exitosos se tratan como excepciones.
- Interceptores: permiten añadir autenticación, trazas o renovación de tokens de forma transversal.
- Instancias: posibilitan una configuración común (URL base, cabeceras, *timeout*) reutilizable.

Ejemplos

```
axios.get("/api/usuarios")
  .then(r => console.log(r.data))
  .catch(e => console.error(e));

const api = axios.create({
  baseURL: "https://api.miapp.com",
  timeout: 5000,
  headers: { "Authorization": `Bearer ${token}` }
});

api.interceptors.response.use(
  r => r,
  e => {
    if (e.response?.status === 401) console.warn("Sesión expirada");
    return Promise.reject(e);
  }
);
```

Cancelación con `AbortController`

```
const controller = new AbortController();
axios.get("/api/datos", { signal: controller.signal })
  .catch(err => {
    if (err.name === "CanceledError") console.log("Solicitud cancelada");
  });
controller.abort();
```

8.10.3 Integración en frameworks modernos

Las bibliotecas para vistas (React, Vue) y los marcos de aplicación (Angular) requieren situar las peticiones en el ciclo de vida de componentes y coordinar su resultado con el estado de la interfaz.

React

El patrón habitual consiste en realizar la solicitud dentro de `useEffect` y almacenar el resultado en estado local:

```
import { useEffect, useState } from "react";
import axios from "axios";

function Usuarios() {
  const [usuarios, setUsuarios] = useState([]);
  useEffect(() => {
    axios.get("/api/usuarios")
      .then(r => setUsuarios(r.data))
      .catch(console.error);
```

```
  }, []);
  return <ul>{usuarios.map(u => <li key={u.id}>{u.nombre}</li>)}</ul>;
}
```

Vue

Con la Options API, la carga suele ubicarse en `mounted`; con la Composition API, en `onMounted` y `ref`:

```
export default {
  data() { return { usuarios: [] }; },
  async mounted() {
    const r = await axios.get("/api/usuarios");
    this.usuarios = r.data;
  }
};
```

Angular

Angular incorpora un cliente HTTP propio, basado en `RxJS`, que opera sobre *observables*:

```
this.http.get<Usuario[]>("/api/usuarios").subscribe({
  next: data => this.usuarios = data,
  error: err => console.error(err)
});
```

La arquitectura favorece interceptores para autenticar, registrar o transformar respuestas de manera consistente.

8.10.4 Gestión avanzada del estado remoto en cliente

A medida que aumenta la complejidad, surgen necesidades de caché, revalidación, invalidación y sincronización entre pestañas. Para ello destacan **SWR** y **React Query (TanStack Query)**.

SWR: *stale-while-revalidate*

SWR ofrece datos "al instante" desde caché y, en segundo plano, obtiene una versión actualizada. El resultado es una experiencia ágil con mínimos cambios de código.

Ejemplo

```
import useSWR from "swr";
import axios from "axios";
const fetcher = url => axios.get(url).then(r => r.data);

function Perfil() {
  const { data, error, isLoading } = useSWR("/api/usuario", fetcher);
  if (isLoading) return <p>Cargando. . . </p>;
  if (error) return <p>Error</p>;
  return <div>Bienvenido, {data.nombre}</div>;
}
```

Ventajas

- Revalidación en segundo plano y sincronización entre pestañas.
- API simple, centrada en el dato como "fuente de verdad" remota.

React Query (TanStack Query)

TanStack Query amplía el control del ciclo de vida de las consultas: caché configurable, estrategias de *refetch*, reintentos, *devtools*, soporte SSR y patrones de invalidación por claves.

Ejemplo

```
import { useQuery } from "@tanstack/react-query";
import axios from "axios";

function ListaUsuarios() {
  const { data, error, isFetching } = useQuery({
    queryKey: ["usuarios"],
    queryFn: () => axios.get("/api/usuarios").then(r => r.data)
  });
  if (isFetching) return <p>Cargando. . . </p>;
  if (error) return <p>Error</p>;
  return <ul>{data.map(u => <li key={u.id}>{u.nombre}</li>)}</ul>;
}
```

Cuándo elegir

- **SWR**: proyectos ligeros o medianos con necesidades de revalidación sencillas.
- **TanStack Query**: aplicaciones extensas, múltiples endpoints, políticas de caché/invalidación complejas y flujos *offline*.

8.10.5 Criterios de elección y mantenimiento

- **Simplicidad vs. capacidades**: priorizar `fetch()` cuando sea suficiente; incorporar Axios para uniformar errores y configuración; escalar a SWR o TanStack Query si la gestión del estado remoto lo exige.
- **Consistencia**: centralizar cabeceras, autenticación y manejo de errores (instancias Axios o interceptores/servicios en el marco elegido).
- **Observabilidad**: añadir trazas e identificadores de petición en interceptores para depurar y correlacionar eventos.
- **Evolución**: planificar migraciones desde jQuery en sistemas heredados, con pruebas que aseguren equivalencia funcional.

En conjunto, estas herramientas complementan la API nativa y proporcionan niveles crecientes de abstracción. El criterio de selección debería partir de la necesidad real: desde llamadas puntuales con `fetch()` hasta la orquestación completa del estado remoto con librerías de caché y revalidación, siempre con atención a la claridad, la mantenibilidad y la escalabilidad del código.

8.11 Patrones de actualización en tiempo real

Hasta ahora, el intercambio de datos se ha descrito con un patrón **petición–respuesta**: el cliente inicia la solicitud y el servidor devuelve un resultado. Sin embargo, hay escenarios —paneles de monitorización, mensajería, seguimiento logístico o cotizaciones bursátiles— que requieren sincronización continua sin intervención del usuario. Esta sección presenta los principales enfoques: desde *polling* y revalidación progresiva hasta canales persistentes con **Server-Sent Events (SSE)** y **WebSockets**.

8.11.1 *Polling* y estrategias de revalidación

Concepto de *polling*

El *polling* consiste en realizar solicitudes periódicas para comprobar si existen cambios. Es simple, compatible con cualquier servidor HTTP y suficiente cuando la frecuencia de actualización es moderada.

Ejemplo básico

```
setInterval(async () => {
  const r = await fetch("/api/estado");
  const datos = await r.json();
  actualizarVista(datos);
}, 5000);
```

Su sencillez tiene coste: si los datos cambian poco, se generan peticiones innecesarias y carga adicional en servidor y red.

Polling exponencial

Para equilibrar inmediatez y consumo, puede ampliarse gradualmente el intervalo cuando no se detectan cambios y reiniciarlo cuando aparezcan novedades.

Ejemplo

```
let intervalo = 2000;

async function consultar() {
  try {
    const r = await fetch("/api/novedades");
    const d = await r.json();
    if (d.hayCambios) { actualizarVista(d); intervalo = 2000; }
    else { intervalo = Math.min(intervalo * 2, 60000); }
```

```
  } catch {
    intervalo = Math.min(intervalo * 2, 60000);
  } finally {
    setTimeout(consultar, intervalo);
  }
}
consultar();
```

Ventajas

- Reduce la frecuencia en periodos de inactividad.
- Se adapta automáticamente al ritmo real de cambios.
- No requiere protocolos específicos.

Limitaciones

- Detección con cierto retraso.
- Posibles ráfagas si muchos clientes reanudan a la vez.

Patrón *stale-while-revalidate* (SWR)

Este enfoque muestra de inmediato datos en caché —aunque estén "antiguos"— y, a la vez, solicita una versión actualizada en segundo plano. La interfaz permanece ágil y se sincroniza cuando llega la respuesta.

Esquema

1. Lectura instantánea desde caché local.
2. Solicitud asíncrona al servidor.
3. Sustitución de la vista y actualización de la caché al recibir datos nuevos.

Ejemplo

```
let cache = null;

async function obtenerDatos() {
  if (cache) mostrar(cache);
  const r = await fetch("/api/datos");
  const nuevos = await r.json();
  cache = nuevos;
  mostrar(nuevos);
}
setInterval(obtenerDatos, 10000);
```

8.11.2 Server-Sent Events (SSE)

Descripción

SSE es un estándar que habilita una conexión HTTP *unidireccional* desde servidor hacia cliente. El navegador abre un canal con `EventSource` y recibe eventos mientras la conexión permanezca activa.

Cliente

```
const source = new EventSource("/api/stream");
source.onmessage = (e) => actualizarVista(JSON.parse(e.data));
source.onerror = (e) => console.error("Error SSE:", e);
```

Servidor (Node.js)

```
app.get("/api/stream", (req, res) => {
  res.setHeader("Content-Type", "text/event-stream");
  res.setHeader("Cache-Control", "no-cache");
  res.setHeader("Connection", "keep-alive");
  const t = setInterval(() => {
    res.write(`data: ${JSON.stringify({ hora: new Date() })}\n\n`);
  }, 2000);
  req.on("close", () => clearInterval(t));
});
```

Características

- Reintento automático gestionado por el navegador.
- Implementación sencilla sobre HTTP estándar.
- Apropiado para notificaciones, *logs* y paneles con flujo continuo.

Limitaciones

- Sólo servidor → cliente.
- Orientado a texto; no idóneo para grandes binarios.

8.11.3 WebSockets

Descripción

WebSockets habilita un canal *bidireccional* persistente. Tras un *upgrade* desde HTTP, cliente y servidor pueden intercambiar mensajes en cualquier momento con baja latencia.

Cliente

```
const socket = new WebSocket("wss://miapp.com/socket");
socket.onopen = () => console.log("Conectado");
socket.onmessage = (e) => procesar(e.data);
socket.send("Hola");
```

Servidor (Node.js + ws)

```
import { WebSocketServer } from "ws";
const wss = new WebSocketServer({ port: 8080 });
wss.on("connection", ws => {
  ws.send("Bienvenido");
  ws.on("message", msg => ws.send(`Echo: ${msg}`));
});
```

Características

- Comunicación interactiva y de baja latencia.
- Válido para chats, edición colaborativa, juegos o telemetría.

Limitaciones

- Requiere soporte específico en la infraestructura.
- Mayor complejidad operativa y de observabilidad.

8.11.4 Comparativa resumida: SSE vs WebSockets

Criterio	**SSE**	**WebSockets**
Dirección	Unidireccional (S→ C)	Bidireccional (C↔S)
Protocolo	HTTP/1.1 (texto)	WS (tras *upgrade*)
Uso típico	Notificaciones, *feeds*, *logs*	Chat, colaboración, juegos, IoT
Reintentos	Automáticos (nativo)	Personalizados
Complejidad	Baja	Media/alta

8.11.5 Criterios de elección

- ***Polling* / SWR**: cambios poco frecuentes, simplicidad operativa o restricciones de infraestructura.
- **SSE**: flujo constante de eventos del servidor con menor complejidad que un canal bidireccional.
- **WebSockets**: interacción en tiempo real en ambos sentidos o necesidad de baja latencia sostenida.

Ejemplos orientativos

- Seguimiento de flotas con notificaciones periódicas → SSE.
- Mensajería interna con escritura/lectura simultánea → WebSockets.
- Informes que se actualizan cada minuto → *polling* con revalidación.

En síntesis, las actualizaciones en tiempo real amplían el modelo AJAX clásico. *Polling* y SWR priorizan compatibilidad y simplicidad; SSE añade un flujo descendente eficiente; y WebSockets proporciona intercambio bidireccional con latencias bajas. La elección depende de la urgencia de actualización, el volumen de mensajes, la complejidad operativa y los requisitos de bidireccionalidad.

8.12 Pruebas, depuración y observabilidad

El intercambio asíncrono de datos añade complejidad al ciclo de desarrollo: cada solicitud puede fallar por causas distintas (red, autenticación, latencias, formatos inesperados). Para asegurar aplicaciones fiables y mantenibles, conviene abordar de forma sistemática tres frentes: **pruebas**, **depuración** y **observabilidad**. Esta sección resume herramientas y prácticas para analizar el comportamiento de las peticiones, desde las *DevTools* y los archivos HAR, hasta pruebas automatizadas con entornos simulados y registro estructurado con identificadores de correlación.

8.12.1 Trazas en DevTools, HAR y análisis *waterfall*

DevTools y la pestaña *Network*

La vista *Network* permite inspeccionar, en tiempo real, las solicitudes que genera la aplicación: método, URL, estado, tiempos, tamaño, cabeceras y cuerpo. Es el primer recurso para aislar problemas de comunicación.

Elementos relevantes

- **Method**: método HTTP efectivo (`GET`, `POST`, etc.).
- **Status**: código devuelto por el servidor.
- **Time**: duración total de la transacción.
- **Size**: peso combinado de cabeceras y cuerpo.
- **Preview/Response**: inspección formateada de la respuesta.

Diagnóstico inicial

- `(blocked by CORS)`: falta de autorización del origen en el backend.
- `(failed)`: fallo de red, DNS o dominio inaccesible.
- `401/403`: credenciales ausentes o permisos insuficientes.
- `500`: error interno del servidor o dependencia.

Archivos HAR: captura y análisis

El formato HAR (*HTTP Archive*) registra la sesión de red para su análisis posterior o para compartir con otros equipos.

Qué contiene

- Cronología completa y tiempos por fase.
- Cabeceras íntegras de solicitud y respuesta.
- Detalles de conexión: DNS, TCP, TLS.
- Tiempos de espera, bloqueo y transferencia.

Procedimiento sugerido

1. Activar `Preserve log` en *Network*.
2. Reproducir el escenario problemático.
3. Exportar: *Save all as HAR*.
4. Analizar en herramientas específicas (*HAR Analyzer*, *Charles*, *Fiddler*).

Lectura del gráfico *waterfall*

El desglose por fases ayuda a localizar el cuello de botella:

- **Queueing/Stalled**: espera por disponibilidad de conexión.
- **DNS Lookup**: resolución del dominio.
- **Initial Connection**: establecimiento TCP y negociación TLS.
- **Request Sent**: envío del cuerpo al servidor.
- **Waiting (TTFB)**: tiempo hasta el primer byte (cómputo en servidor).
- **Content Download**: transferencia de la respuesta.

Interpretación práctica

- TTFB elevado: saturación en backend o base de datos.
- Retrasos en DNS/TLS: problemas de red o configuración de certificados.
- Descarga prolongada: respuestas voluminosas o compresión ineficiente.

8.12.2 Pruebas de red con Jest/Vitest y `msw` (Mock Service Worker)

Motivación

Las pruebas de integración sobre la capa de red validan el comportamiento del cliente sin depender de un backend real, lo que acelera los ciclos de CI y reduce la flakiness. El objetivo es reproducir condiciones de éxito y de error con control total sobre tiempos y contenidos.

Simulación con `msw`

MSW intercepta `fetch()` y Axios a nivel de red, devolviendo respuestas programadas. A diferencia de los *mocks* unitarios, mantiene el flujo real de la aplicación y resulta más representativo.

Instalación

```
npm install msw --save-dev
```

Handlers de ejemplo

```
// src/mocks/handlers.js
import { rest } from "msw";
export const handlers = [
  rest.get("/api/usuarios", (_req, res, ctx) =>
    res(ctx.status(200), ctx.json([
      { id: 1, nombre: "César" }, { id: 2, nombre: "Ana" }
    ]))
  ),
];
```

Servidor de prueba

```
// src/mocks/server.js
import { setupServer } from "msw/node";
import { handlers } from "./handlers";
export const server = setupServer(...handlers);
```

Integración en Jest/Vitest

```
import { server } from "../mocks/server";
import axios from "axios";

beforeAll(() => server.listen());
afterEach(() => server.resetHandlers());
afterAll(() => server.close());

test("obtiene usuarios", async () => {
  const r = await axios.get("/api/usuarios");
  expect(r.data).toHaveLength(2);
});
```

Cobertura de errores

```
server.use(
  rest.get("/api/usuarios", (_req, res, ctx) =>
    res(ctx.status(500), ctx.json({ error: "Fallo interno" }))
  )
);
```

Además de los casos nominales, conviene incluir: respuestas `4xx/5xx`, latencias artificiales, `timeout`, reintentos y cancelación.

8.12.3 Logs estructurados y correlación de peticiones

Registro estructurado

Los logs en formato estructurado (p. ej., JSON) facilitan indexación y consulta en plataformas de observabilidad. Esto permite derivar métricas, paneles y alertas a partir de eventos homogéneos.

Ejemplo

```
{
  "timestamp": "2025-11-04T10:15:23Z",
  "level": "info",
  "service": "frontend",
  "event": "fetch_success",
  "url": "/api/usuarios",
  "status": 200,
  "duration_ms": 152,
  "request_id": "3a7b91e2"
}
```

Identificadores de correlación

Un `request-id` propagado extremo a extremo permite seguir una misma operación a través de frontend, pasarelas y microservicios.

Esquema

1. El cliente genera un UUID y lo envía en `X-Request-ID`.

   ```
   fetch("/api/datos", { headers: { "X-Request-ID": uuidv4() } });
   ```

2. Los servicios intermedios registran y reenvían el identificador.

3. Las herramientas de observabilidad correlacionan eventos por ese campo.

Buenas prácticas

- Niveles de severidad consistentes: `debug`, `info`, `warn`, `error`.
- Evitar datos sensibles (PII, secretos, tokens) en los logs.
- Registrar duraciones, tamaños y códigos de estado.
- Rotación, retención y compresión según política de cumplimiento.
- Sincronización horaria (NTP) para coherencia temporal.

Monitoreo y visualización

La centralización de eventos y métricas facilita la detección temprana de incidencias:

- **Elastic Stack (ELK)**: indexación y paneles analíticos.
- **Grafana + Loki**: agregación ligera de logs y visualización.
- **Datadog / New Relic**: observabilidad unificada (logs, métricas, trazas).

En síntesis, la calidad de una aplicación que depende de AJAX se apoya en diagnósticos reproducibles y en señales de observabilidad fiables. La combinación de trazas detalladas, pruebas con *mocks* de red y registro estructurado con *request-id* permite anticipar errores, localizar cuellos de botella y mantener una operación predecible en producción.

8.13 Buenas prácticas y anti-patrones

El uso de AJAX no consiste únicamente en establecer la comunicación entre cliente y servidor; implica también ofrecer una experiencia coherente, segura y mantenible. En la práctica, muchos problemas no son estrictamente técnicos, sino de diseño: manejo deficiente de estados, peticiones redundantes, validaciones incompletas o condiciones de carrera. Esta sección recoge **buenas prácticas** recomendadas y **anti-patrones** frecuentes que conviene evitar.

8.13.1 Estados de carga, error y vacío coherentes

Modelado explícito de estados

Cada solicitud asíncrona transita por estados distinguibles: *inicial* (sin datos), *cargando*, *éxito*, *vacío* (sin resultados) y *error*. Es preferible representarlos de forma explícita en el código para evitar ambigüedades.

Ejemplo en React

```
const [estado, setEstado] = useState("idle");
const [datos, setDatos] = useState([]);

useEffect(() => {
  setEstado("loading");
  fetch("/api/usuarios")
    .then(r => r.json())
    .then(d => {
      setDatos(d);
      setEstado(d.length ? "success" : "empty");
    })
    .catch(() => setEstado("error"));
}, []);
```

Representación visual sugerida

- `loading`: indicador de progreso o texto informativo.
- `success`: renderizado del resultado.
- `empty`: mensaje claro de "sin resultados".
- `error`: explicación breve y opción de reintento.

Anti-patrón Codificar estados con valores implícitos (por ejemplo, `null` para "cargando" o `false` para "error") dificulta el mantenimiento y genera inconsistencias.

8.13.2 Reintentos con *backoff* y *jitter*

Motivación

Los fallos de red intermitentes son habituales. Reintentar puede mejorar la resiliencia, pero hacerlo de forma simultánea desde muchos clientes origina *retry storms* y sobrecarga el servidor.

Retroceso exponencial con aleatoriedad

El *exponential backoff* aumenta el intervalo entre intentos fallidos y, combinado con *jitter* (componente aleatorio), evita sincronizaciones indeseadas.

Ejemplo

```
async function fetchConReintento(url, intentos = 5) {
  for (let i = 0; i < intentos; i++) {
    try {
      const res = await fetch(url);
      if (!res.ok) throw new Error(res.status);
      return await res.json();
    } catch {
      const backoff = Math.min(1000 * 2 ** i, 16000);
      const jitter = Math.random() * 500;
      await new Promise(r => setTimeout(r, backoff + jitter));
    }
  }
  throw new Error("No se pudo completar la solicitud");
}
```

Alcance Aplicar reintentos sólo a operaciones seguras e idempotentes (`GET`, `PUT`, `DELETE`). Reintentar `POST` puede producir duplicados.

Anti-patrón Reintento inmediato e ilimitado: amplifica fallos y degrada el rendimiento global.

8.13.3 Evitar condiciones de carrera y respuestas obsoletas

Respuestas fuera de orden

Solicitudes consecutivas pueden resolverse en orden distinto al de envío. Si se actualiza la vista con respuestas antiguas, se muestra información obsoleta.

Control por contexto activo

```
let ultimaBusqueda = "";

async function buscar(q) {
  ultimaBusqueda = q;
  const r = await fetch(`/api/buscar?q=${encodeURIComponent(q)}`);
  const data = await r.json();
  if (q === ultimaBusqueda) mostrarResultados(data);
}
```

Cancelación con `AbortController`

```
let controller;

async function buscar(q) {
  controller?.abort();
  controller = new AbortController();
  const r = await fetch(`/api/buscar?q=${encodeURIComponent(q)}`,
                        { signal: controller.signal });
  const data = await r.json();
  mostrarResultados(data);
}
```

Anti-patrón Disparar múltiples peticiones sin cancelación ni verificación del contexto, saturando el backend y degradando la coherencia visual.

8.13.4 Manejo de formularios: validación, accesibilidad e idempotencia

Validación en cliente y servidor

La validación en el cliente mejora la experiencia, pero no sustituye a la validación en el servidor.

Ejemplo

```
const form = document.querySelector("#registro");
form.addEventListener("submit", async (e) => {
  e.preventDefault();
  const datos = Object.fromEntries(new FormData(form));
  if (!datos.email.includes("@")) { mostrarError("Email inválido"); return; }
```

```
  try {
    const r = await fetch("/api/registro", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify(datos)
    });
    r.ok ? mostrarExito("Registro completado")
         : mostrarError("Error en el servidor");
  } catch {
    mostrarError("No hay conexión");
  }
});
```

Estados y accesibilidad

Comunicar el estado del envío y asegurar la accesibilidad:

- Botón deshabilitado durante el envío (`disabled` y `aria-busy`).
- Mensajes anunciados por lectores de pantalla (`role="status"`, `aria-live`).
- Conservación del foco tras errores y confirmaciones.

Prevención de duplicados: idempotencia de cliente a servidor

Evitar envíos repetidos con identificadores de idempotencia:

```
async function enviarFormulario(datos) {
  const key = crypto.randomUUID();
  await fetch("/api/pago", {
    method: "POST",
    headers: { "Content-Type": "application/json",
               "Idempotency-Key": key },
    body: JSON.stringify(datos)
  });
}
```

El servidor debe registrar temporalmente las claves recibidas y rechazar repeticiones.

Anti-patrones en formularios

- Suprimir errores de red sin avisar al usuario.
- Mensajes genéricos sin guía de acción.
- Reemplazar el formulario sin respetar el foco.
- Omitir validaciones mínimas de campos requeridos.

8.13.5 Principios generales

- Modelar estados explícitos y representarlos en la interfaz.
- Implementar reintentos con *backoff* y *jitter*; limitar su alcance a operaciones idempotentes.
- Cancelar o ignorar respuestas obsoletas para evitar carreras.
- Validar siempre en cliente y servidor; no confiar en el lado cliente.
- Prevenir duplicidades con claves de idempotencia y bloqueo temporal de UI.
- Redactar mensajes de error claros y accesibles, con opciones de recuperación.

En síntesis, las buenas prácticas en AJAX pivotan sobre la **coherencia de estados**, la **resiliencia ante fallos** y la **claridad para el usuario**. Aplicarlas reduce incidencias difíciles de diagnosticar, facilita el mantenimiento y mejora la fiabilidad global del sistema.

8.14 Casos prácticos guiados

Esta sección reúne ejemplos completos que integran los conceptos del temario. Cada caso aborda una situación habitual del desarrollo con AJAX y muestra, paso a paso, cómo aplicar buenas prácticas de asincronía, control del DOM y diseño de la interacción para lograr una experiencia de uso predecible y eficiente.

8.14.1 Autocompletado de búsqueda

El autocompletado ofrece sugerencias conforme el usuario escribe, sin recargar la página. El objetivo es responder con agilidad sin sobrecargar ni cliente ni servidor.

Objetivos

- Evitar solicitudes redundantes mediante **debounce**.
- Cancelar peticiones obsoletas cuando hay nueva entrada.
- Resaltar coincidencias y mantener la accesibilidad del listado.

Estructura HTML

```
<input id="busqueda" type="text" placeholder="Buscar producto..."
       role="combobox" aria-autocomplete="list" aria-expanded="false"
       aria-controls="sugerencias" aria-activedescendant="" />
<ul id="sugerencias" role="listbox" aria-label="Sugerencias"></ul>
```

Implementación paso a paso

1. Debounce y cancelación

```
let controller;
const input = document.querySelector("#busqueda");
const lista = document.querySelector("#sugerencias");

function debounce(fn, delay) {
  let timer;
  return function(...args) {
    clearTimeout(timer);
    timer = setTimeout(() => fn.apply(this, args), delay);
  };
}

const buscar = debounce(async (q) => {
  controller?.abort();
  controller = new AbortController();

  const texto = q.trim();
  if (!texto) {
    lista.innerHTML = "";
    input.setAttribute("aria-expanded", "false");
    return;
  }

  const res = await fetch(`/api/buscar?q=${encodeURIComponent(texto)}`, {
    signal: controller.signal
  });
  if (!res.ok) return; // manejo mínimo; en producción, informar al usuario
  const data = await res.json();
  mostrarSugerencias(texto, data);
}, 300);

input.addEventListener("input", (e) => buscar(e.target.value));
```

2. Resaltado de coincidencias y accesibilidad

```
function mostrarSugerencias(q, resultados) {
 const regex = new RegExp(`(${q.replace(/[.*+?^${}()|[\]\\]/g, '\\$&')})`, "ig");
  lista.innerHTML = resultados.map((r, i) => {
    const id = `sug-${i}`;
    const nombre = r.nombre.replace(regex, "<strong>$1</strong>");
   return `<li id="${id}" role="option" aria-selected="false">${nombre}</li>`;
  }).join("");

  input.setAttribute("aria-expanded", String(resultados.length > 0));
}
```

Buenas prácticas aplicadas

- Cancelación de solicitudes previas (`AbortController`).
- Control de frecuencia con *debounce*.
- Atributos ARIA para una interacción con lector de pantalla predecible.

8.14.2 Formulario con subida de archivos

La subida asíncrona evita refrescos completos, permite validar antes de enviar y ofrece retroalimentación de progreso.

Objetivos

- Enviar archivos mediante `FormData`.
- Mostrar progreso de subida (`progress`).
- Validar tamaño y tipo antes del envío.

Estructura HTML

```
<form id="upload-form" aria-describedby="estado">
  <input type="file" id="archivo" name="archivo" accept=".png,.jpg" />
 <progress id="barra" max="100" value="0" aria-label="Progreso de subida"></progress>
  <button type="submit">Subir</button>
</form>
<div id="estado" role="status" aria-live="polite"></div>
```

Código JavaScript

```
const form = document.querySelector("#upload-form");
const barra = document.querySelector("#barra");
const estado = document.querySelector("#estado");

form.addEventListener("submit", (e) => {
  e.preventDefault();

  const archivo = form.archivo.files[0];
  if (!archivo) { estado.textContent = "Seleccione un archivo."; return; }
  if (archivo.size > 5 * 1024 * 1024) {
    estado.textContent = "Archivo demasiado grande (máx. 5MB).";
    return;
  }

  const formData = new FormData();
  formData.append("archivo", archivo);

  const xhr = new XMLHttpRequest();
  xhr.open("POST", "/api/upload");
```

```
  xhr.upload.onprogress = (e) => {
    if (e.lengthComputable) {
      const pct = (e.loaded / e.total) * 100;
      barra.value = pct;
      barra.setAttribute("aria-valuenow", String(Math.round(pct)));
    }
  };

  xhr.onload = () => {
    estado.textContent = (xhr.status === 200)
      ? "Subida completada."
      : "Error del servidor.";
  };

  xhr.onerror = () => { estado.textContent = "Error de conexión."; };
  xhr.send(formData);
});
```

Puntos clave

- `FormData` gestiona la codificación `multipart/form-data` de forma nativa.
- `progress` comunica el avance y mejora la percepción de rapidez.
- Validación previa evita tráfico innecesario y errores previsibles.

Anti-patrón Convertir el archivo a Base64 para enviarlo en JSON: añade sobrecarga de CPU y memoria, sin aportar beneficios en este caso.

8.14.3 *Infinite scroll* y paginación

El desplazamiento infinito carga contenido adicional cuando el usuario se aproxima al final de la lista. Debe implementarse con cuidado para evitar duplicados y peticiones simultáneas.

Objetivos

- Detectar el final de la lista con `IntersectionObserver`.
- Usar paginación por cursor para eficiencia y orden estable.
- Evitar condiciones de carrera y elementos repetidos.

Estructura HTML

```
<ul id="lista" aria-live="polite" aria-busy="false"></ul>
<div id="sentinela" aria-hidden="true"></div>
```

Código JavaScript

```
const lista = document.querySelector("#lista");
const sentinela = document.querySelector("#sentinela");

let cursor = null;
let cargando = false;

async function cargarMas() {
  if (cargando) return;
  cargando = true;
  lista.setAttribute("aria-busy", "true");

  const url = cursor ? `/api/items?after=${cursor}` : "/api/items";
  const res = await fetch(url);
 if (!res.ok) { cargando = false; lista.setAttribute("aria-busy", "false"); return; }

  const data = await res.json();

  data.items.forEach(item => {
    const li = document.createElement("li");
    li.textContent = item.nombre;
    li.setAttribute("data-id", item.id);
    lista.appendChild(li);
  });

  cursor = data.nextCursor;
  lista.setAttribute("aria-busy", "false");
  cargando = false;
}

const observador = new IntersectionObserver((entries) => {
  if (entries[0].isIntersecting) cargarMas();
}, { rootMargin: "200px" });

observador.observe(sentinela);
```

Buenas prácticas

- Bandera `cargando` para evitar paralelismo no deseado.
- Paginación por *cursor* para estabilidad entre consultas.
- Alternativa accesible: botón “Cargar más” cuando `IntersectionObserver` no esté disponible.

Anti-patrón Escuchar eventos `scroll` de forma continua sin *throttle*, provocando cálculos y repintados excesivos.

8.14.4 Dashboard con datos vivos

Los paneles de métricas requieren actualización frecuente o continua. La elección del mecanismo depende de la latencia tolerable y la direccionalidad de la comunicación.

Estrategias

- **Polling**: sencillo y universal; adecuado para métricas de baja frecuencia.
- **SSE**: canal descendente eficiente; idóneo para *feeds* y registros en vivo.
- **WebSockets**: bidireccional en tiempo real; recomendado para colaboración y control interactivo.

Ejemplo con Server-Sent Events (SSE)

Servidor (Node.js)

```
app.get("/api/metricas", (req, res) => {
  res.setHeader("Content-Type", "text/event-stream");
  res.setHeader("Cache-Control", "no-cache");
  res.setHeader("Connection", "keep-alive");

  const t = setInterval(() => {
    const payload = JSON.stringify({
      cpu: Math.random() * 100,
      memoria: Math.random() * 100
    });
    res.write(`data: ${payload}\n\n`);
  }, 2000);

  req.on("close", () => clearInterval(t));
});
```

Cliente

```
const sse = new EventSource("/api/metricas");
sse.onmessage = (e) => {
  const d = JSON.parse(e.data);
  document.querySelector("#cpu").textContent = d.cpu.toFixed(1) + "%";
  document.querySelector("#mem").textContent = d.memoria.toFixed(1) + "%";
};
sse.onerror = () => { /* reconexión automática gestionada por EventSource */ };
```

Buenas prácticas en dashboards

- Actualizar sólo los elementos que cambian; evitar repintados globales.
- Pausar flujos en pestañas inactivas (`document.visibilityState`).
- Añadir tolerancia a fallos: reconexión con retroceso exponencial cuando proceda.

Anti-patrón Recrear el gráfico completo en cada evento: preferible actualizar *datasets* o valores internos de la librería empleada.

En conjunto, estos casos ilustran cómo trasladar los principios de AJAX a situaciones reales: estados explícitos, cancelación de solicitudes obsoletas, validación progresiva, accesibilidad y comunicación eficiente. La aplicación de estas pautas produce interfaces más rápidas, robustas y mantenibles.