

- 8.34. Busca en la documentación de JavaScript qué API permite leer códigos de barras y escribe un pequeño aplicativo con el que el usuario pueda subir la imagen de un código de barras y el programa muestre la información que la API proporciona.
- 8.35. Escribe un programa que, por medio de una API incluida en JavaScript, sea capaz de acceder a los datos leídos por algún sensor de un dispositivo móvil: giroscopio, acelerómetro, etc. Ejecuta tu aplicación desde un dispositivo móvil para comprobar su funcionamiento.

Enlaces web de interés

-  **Mozilla Developers** - https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Global_Objects/Error
(website para desarrolladores)
-  **Microsoft Docs** - <https://docs.microsoft.com/es-es/microsoft-edge/developer/>
(documentación para desarrolladores de Edge)
-  **JavaScript Info** - <https://es.javascript.info/>
(aprendizaje de JavaScript a través de tutoriales)
-  **Keep Coding** - <https://keepcoding.io/>
(escuela de programadores)
-  **ECMAScript 2022** - <https://262.ecma-international.org/13.0/>
(última actualización de la especificación del lenguaje)
-  **W3C** - <https://www.w3.org/standards/webdesign/script>
(World Wide Web Consortium)
-  **IBM Docs** - <https://www.ibm.com/docs/en/baw/22.x?topic=apis-javascript>
(documentación sobre el IBM Business Automation Workflow)



Programación asíncrona

Objetivos

- Entender la diferencia entre sincronía y asíncronía.
- Identificar los problemas inherentes a la naturaleza asíncrona de JavaScript.
- Descubrir las desventajas de usar los *callback hell*.
- Analizar las posibilidades que ofrecen las promesas.
- Estudiar las ventajas que ofrecen las funciones **async**.
- Conocer las diferencias entre la asíncronía de la ejecución y la asíncronía de las peticiones a un servidor.
- Profundizar en la tecnología AJAX.
- Hallar las diferentes formas de realizar peticiones a un servidor.
- Aprender a manejar las respuestas ofrecidas por un servidor a peticiones AJAX.
- Combinar la asíncronía de la ejecución con la asíncronía de las comunicaciones HTTP.

Contenidos

- 9.1. Sincronía y asíncronía
- 9.2. Orquestar la asíncronía
- 9.3. AJAX

Introducción

Terminar una obra dedicada al desarrollo web en entorno cliente con una unidad dedicada a la programación asíncrona a algunos lectores les puede resultar algo poco intuitivo, puesto que JavaScript es por naturaleza un lenguaje de programación asíncrono. Sin embargo, es muy importante entender que a ciertos niveles esta asíncronía genera ciertos problemas cuya complejidad no puede analizarse si no se tiene una extensa lista de conceptos asimilados.

En esta unidad se abordan esos problemas, cómo pueden solventarse y qué estrategias son las más eficaces para alcanzar un determinado objetivo.

Además, resuelta la primera cuestión, la programación asíncrona con JavaScript abre otra puerta que esconde un auténtico tesoro: el consumo de datos remotos de forma transparente para el usuario. Este procesamiento en segundo plano ha provocado una auténtica revolución en la experiencia del usuario y ha llevado a las aplicaciones web a dar un salto de calidad nunca antes visto en la historia de internet.

Ponte cómodo y enfoca todos tus sentidos en las próximas páginas porque aquí residen los conceptos que te convertirán en un programador JavaScript avanzado.

9.1. Sincronía y asíncronía

Si se piensa en la mayoría de los lenguajes de programación y se observa su flujo de ejecución se ve que es completamente síncrono. El programa ejecuta la línea 1, termina, ejecuta la línea 2, termina, ejecuta la línea 3, termina; y así sucesivamente hasta que se ejecutan todas las líneas. Si el usuario solicitó un dato en la línea 1 que debe usar en la línea 21, debe esperar a que se ejecuten las otras 20 líneas intermedias antes de poder usarlo, porque la sincronía provoca que hasta que no termine una tarea no puede empezar la siguiente. Se trata de un modelo síncrono monohilo.

Algunos lenguajes de programación intentan aportar cierta asíncronía por medio de la creación de hilos. Cada tarea se lanza de forma independiente y sus caminos solo se tocan cuando tienen que acceder a un recurso compartido.

Tomando JavaScript como referencia, en una línea de código se solicita modificar el tipo y el tamaño de fuente de todos los elementos textuales del DOM, en la siguiente se consulta la temperatura de una localidad y en la siguiente se solicita la tabla de mareas para mañana. Es perfectamente posible que el usuario vea en pantalla primero la tabla de mareas, luego el cambio de estilo del texto y, por último, la temperatura. Esto ocurre porque el usuario no tiene que esperar a que acabe la primera tarea para que se ejecute la segunda; JavaScript deja trabajando cada tarea y gestiona los datos conforme van llegando. Pero, ¿qué ocurre si la primera tarea es la primera en terminar? El resultado será toda la página con un nuevo estilo de letra, excepto la temperatura y la tabla de mareas, cuyos datos al llegar más tarde no han sido formateados con los nuevos estilos aplicados por la primera tarea.

Este problema, aunque parezca una novedad, no lo es, porque así funcionan precisamente los *callback* que se han estudiado en unidades anteriores.

Si se hace esto:

```
tablaMareas.incluidaEnDOM( () => { actualizarFuente(); } );
```

se consigue modificar el tipo y el tamaño de fuente tan pronto como esté disponible en el DOM.

Pero, cuando intervienen muchas acciones distintas, esto no es tan sencillo.

9.2. Orquestar la asíncronía

Tener muchas acciones realizando tareas por separado en ocasiones genera ciertos problemas que no pueden resolverse de una manera sencilla usando un *callback* como los vistos hasta ahora. Por eso se necesitan otras estrategias que ayuden a poner cierto orden en la ejecución de las acciones, sobre todo porque en ciertos momentos hay que asegurarse de que una tarea ha terminado para poder abordar de manera segura otras, que a su vez tienen sus restricciones de orden. Es preciso orquestar la asíncronía.

9.2.1. La solución infernal

La primera aproximación para resolver estos problemas son los llamados *callback hell*. Se los llamó así porque se trata de utilizar un *callback* anidado dentro de otro varias veces y, a pesar de que puede ayudar a resolver un problema de cierta complejidad, resulta en un código complejo de leer, interpretar y sobre todo mantener.

Por ejemplo, este código define una caja que contiene tres encabezados a los que se han aplicado ciertos estilos:

```
<div class="contenedor">
  <h2 class="palabra">El</h2>
  <h2 class="palabra">Callback</h2>
  <h2 class="palabra">Hell</h2>
</div>
```

Se pretende que tras la carga de la página se cree una animación donde vayan apareciendo las tres palabras para construir la frase «El Callback Hell». Inicialmente, la página estará en blanco y tras 500 milisegundos de espera aparecerá la primera palabra; luego, tras 300 milisegundos aparecerá la segunda palabra, y lo mismo sucederá con la tercera palabra tras 100 milisegundos, hasta que se muestre la frase completa:

El Callback Hell

Figura 9.1. Aspecto en el navegador tras finalizar la animación.

Para conseguir este efecto, se podría tener un programa JavaScript como el siguiente:

```
let palabras=document.querySelectorAll(".palabra");
let animarPalabras=(animar)=>{
    setTimeout(()=>{
        animar(palabras[0]);
        setTimeout(()=>{
            animar(palabras[1]);
            setTimeout(()=>{
                animar(palabras[2]);
            }, 500)
        }, 300)
    }, 100)
}
let animar=(palabra)=>{
    palabra.classList.add("animar");
}
animarPalabras(animar);
```

El final de la definición de **animarPalabras** es lo que se denomina la «pirámide del infierno». Esta pieza de código ya es por sí misma poco legible, a pesar de ser un ejemplo muy sencillo por tratarse de una misma tarea repetida tres veces. Pero si fueran muchas más tareas, cada tarea fuera algo más compleja e incluyeran control de errores, esta solución sería eso, un auténtico infierno.

Se necesita alguna otra solución que ofrezca una mejor organización.

9.2.2. Promesas

Las promesas forman parte del estándar desde la ES2015, momento desde el que se utilizan de forma intensiva, puesto que resuelven el problema de los *callback hell*.

Una promesa permite realizar de forma asíncrona una tarea, de manera que puede detectarse si esa tarea finalizó con éxito o no. Así se puede organizar mejor el resto de las tareas. Programáticamente son objetos **Promise** que pueden encontrarse en alguna de estas cuatro situaciones:

- **rejected**: la tarea no ha finalizado con éxito.
- **resolved o fulfilled**: la tarea ha finalizado con éxito.
- **pending**: la tarea está en proceso de finalización.
- **settled**: la tarea ha finalizado (con o sin éxito).

Para crear una promesa se debe crear un objeto de tipo **Promise**, asociarle un *callback* y, dentro de él, indicarle cómo actuar en caso de éxito y en caso de fracaso. Por ejemplo:

```
let promesa = new Promise((exito,fracaso)=>{
    let cadena1 = "la casa por el tejado";
    let cadena2 = "casa";
    if (cadena1.includes(cadena2))
        exito("La cadena2 está dentro de la cadena1");
    else
        fracaso(Error("La cadena2 no está dentro de la cadena1"));
});
```

Este programa ha creado una promesa que indica si una cadena de caracteres se encuentra dentro de la otra. Como se había avanzado, se le asocia un *callback* con el nombre de dos funciones: **éxito** y **fracaso**. Cada una de ellas se utiliza para escenificar la situación correspondiente. Si se lanza este código no se verá nada; sin embargo, la mera creación del objeto ya lo está lanzando en segundo plano. ¿Cómo se captura el resultado? Con el método **then**. Si se ejecuta esto:

```
promesa.then((resultado)=>{
    console.log(resultado);
});
```

y se vuelve a cargar el programa, se obtiene lo que se muestra en la Figura 9.2.

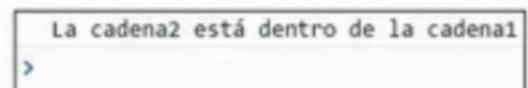


Figura 9.2. Resultado en consola de la ejecución.

Porque según la definición de la promesa, el escenario es de éxito. Si se modifica **cadena2="casssssa"**; se puede simular el escenario de fracaso (Figura 9.3).

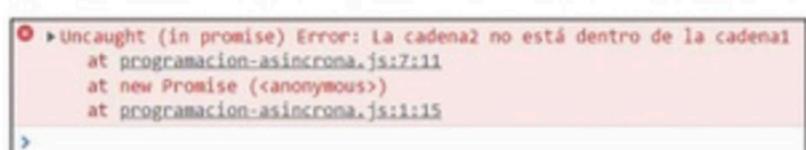


Figura 9.3. Error no capturado tras no encontrarse la cadena2 en la cadena1.

Según la Figura 9.3, se ha lanzado el error que se había previsto, pero aparece **Uncaught Error** (error no capturado). Efectivamente, por eso puede completarse el método **then** para que capture los errores:

```
promesa.then((resultado)=>{
    console.log(resultado);
}).catch(function(resultado){
    console.log("Ahora el error está controlado");
});
```

Haciendo esto, el gestor de errores ya está funcionando para poder actuar en consecuencia dándole el tratamiento que interese en cada momento (Figura 9.4).



Figura 9.4. Error capturado tras no encontrarse la cadena2 en la cadena1.

Aquí no termina la utilidad de las promesas, puesto que pueden encadenarse para construir soluciones sencillas a problemas complejos de asincronía. Pero eso se verá un poco más adelante.

Una última cuestión interesante de las promesas es la combinación de los métodos que ofrecen para conseguir sincronizar tareas asíncronas:

■ **Promise.resolve** crea una promesa exitosa:

```
let promesaExitosa = Promise.resolve("Finalizada con éxito");
promesaExitosa.then((resultado)=>{
    console.log(resultado);
});
```

■ **Promise.reject** crea una promesa fracasada:

```
let promesaFracasada = Promise.reject("Finalizada sin éxito");
promesaFracasada.then((resultado)=>{
    console.log(resultado);
}).catch((error)=>{
    console.log(`Fallo de tipo: ${error.name}`);
});
```

■ **Promise.all** devuelve una promesa cumplida, si todas las promesas que gestiona han finalizado con éxito.

```
let pro1 = new Promise((exito)=>{
    exito("Promesa 1: hecho.");
});
let pro2 = new Promise((exito)=>{
    exito("Promesa 2: hecho.");
});
let pro3 = new Promise((exito)=>{
    exito("Promesa 3: hecho.");
});
let pro4 = new Promise((exito)=>{
    setTimeout(()=>{exito("Promesa 4: hecho.")},5000);
});
let proTodas = Promise.all([pro1,pro2,pro3,pro4]);
proTodas.then((resultados)=>{
    for (let resultado of resultados) {
        console.log(resultado);
    }
    console.log("Sincronizadas");
});
```

En este último ejemplo crean cuatro promesas. La última de ellas no finalizará con éxito hasta que hayan transcurrido cinco segundos. Finalmente se utiliza **Promise.all** para que recoja el resultado de la ejecución de todas ellas. Lo tremadamente importante de este ejemplo es que, por el propio funcionamiento de **Promise.all**, hasta que no terminen todas las promesas con éxito, no devolverá su propia promesa cumplida. Es decir, es un fantástico sistema de sincronización.

Para poder ver las evidencias, no hay más que ejecutar el ejemplo anterior. Lo esperable sería que justo tras comenzar la ejecución se vieran los mensajes de éxito de las tres primeras promesas, y tras cinco segundos el resultado de la cuarta. Pero no, al estar definido **Promise.all** como gestor, no aparecerá ningún mensaje hasta que finalice la última promesa. Cuando pasan los cinco segundos y termina la cuarta promesa, entonces es cuando se muestran al mismo tiempo todos los mensajes de éxito. Se han sincronizado.

Actividad resuelta 9.1

Promesas encadenadas

Crea una promesa encadenada que muestre tres mensajes por consola cuando se resuelve de manera exitosa.

Solución

```
let promesa = new Promise(function (exito, fracaso) {
    exito("Promesa resuelta con éxito");
});
promesa
.then(function mensaje1() {
    console.log("Mensaje de éxito 1.");
})
.then(function mensaje2() {
    console.log("Mensaje de éxito 2.");
})
.then(function mensaje3() {
    console.log("Mensaje de éxito 3.");
});
```

9.2.3. Funciones Async

Las funciones **async** son un tipo especial de función que se comporta como un objeto de tipo **AsyncFunction**. Al usarlas devuelven automáticamente una promesa. Lo interesante de estas funciones, porque ya se conocen las ventajas de las promesas, es que pueden combinarse con el uso del operador **await**. Este operador se aplica a una promesa, y si se cumple, hace que la siguiente línea se ejecute; si no, se mantendrá esperando.

Se retoma el ejemplo anterior para ver la diferencia:

```
let pro1 = new Promise((exito)=>{
    exito("Promesa 1: hecho.");
});
let pro2 = new Promise((exito)=>{
    exito("Promesa 2: hecho.");
});
let pro3 = new Promise((exito)=>{
    exito("Promesa 3: hecho.");
});
let pro4 = new Promise((exito)=>{
    setTimeout(()=>{exito("Promesa 4: hecho.")},5000);
});
async function resultado() {
    let resultado1 = await pro1;
    console.log(resultado1);
    let resultado2 = await pro2;
    console.log(resultado2);
    let resultado3 = await pro3;
    console.log(resultado3);
```

```

let resultado4 = await pro4;
console.log(resultado4);
}
resultado();

```

Las promesas siguen definidas igual que antes, pero ahora su gestión se ha incluido dentro de una función **async** y se ha aplicado el operador **await** a cada promesa. Si se observa el resultado que se obtiene tras la ejecución, se ve que ya no se espera a que terminen todas para mostrarse los mensajes, sino que se ejecuta una promesa y cuando finaliza con éxito se ejecuta la siguiente, y cuando finaliza esta con éxito se ejecuta la siguiente, y así con todas. Lo que ha ocurrido es que se han sincronizado las promesas por separado.

Es interesante matizar que, en el ejemplo anterior, las cuatro promesas se han lanzado de forma asíncrona, pero es la recogida del resultado el que se ha sincronizado.

Si lo que se quiere hacer es que una promesa ni siquiera se cree antes de obtener el resultado de la anterior, hay que modificar el código y usar el operador **await** sobre la propia creación de los objetos:

```

async function resultado() {
    let pro1 = await new Promise((exito)=>{
        exito("Promesa 1: hecho.");
    });
    console.log(pro1);
    let pro2 = await new Promise((exito)=>{
        exito("Promesa 2: hecho.");
    });
    console.log(pro2);
    let pro3 = await new Promise((exito)=>{
        exito("Promesa 3: hecho.");
    });
    console.log(pro3);
    let pro4 = await new Promise((exito)=>{
        setTimeout(()=>{exito("Promesa 4: hecho.")},5000);
    });
    console.log(pro4);
}
resultado();

```

Una última cuestión interesante sobre **await** es que no necesita operar con objetos de tipo **Promise**, como se ha visto aquí, sino que puede aplicarse a cualquier expresión. El operador convertirá la expresión en una promesa y seguirá realizando su trabajo. Eso sí, hay que usarlo siempre dentro de una función **async**:

```

function saludar(usuario) {
    return `Buenas tardes, ${usuario}.`;
}
async function saludoSincronizado(nombre){
    let resultado = await saludar(nombre);
    console.log(resultado);
}
saludoSincronizado("Martín");

```

En este ejemplo se ha aplicado el operador **await** a una función que no es una promesa. Tras haberlo aplicado sobre la función **saludar**, el operador le ha indicado al intérprete que no saque en pantalla el **resultado** hasta que no termine de ejecutarse la función **saludar**. Un procesamiento asíncrono se ha convertido en uno síncrono.

Actividad resuelta 9.2

Asincronía máxima

Escribe un ejemplo en el que se evidencie de forma simple la utilidad de las funciones **async** evitando que un programa se congele mientras espera por el procesamiento de una rutina.

Solución

```

let accionAsincrona = () => {
    return new Promise(exito => {
        setTimeout(() => exito("Promesa cumplida con éxito"), 5000)
    });
}
let muestraMensaje = async () => {
    console.log(await accionAsincrona());
};
console.log("Antes de ejecutar accionAsincrona");
muestraMensaje();
console.log("Después de ejecutar accionAsincrona");

```

El mensaje «Promesa cumplida con éxito» es el último en aparecer cuando ya se han ejecutado todos los demás mensajes. La asincronía de la función ha realizado su trabajo correctamente.

Ahora que ya se dominan los conceptos de la asincronía y se tienen recursos suficientes para sincronizar la ejecución de los elementos que interesen, ha llegado el momento de aplicarlos para descubrir la última joya de este aprendizaje.

9.3. AJAX

AJAX son las siglas de *Asynchronous JavaScript and XML*. A pesar de que el acrónimo incluye a XML, en la actualidad ya no se utiliza XML como lenguaje de intercambio de datos, sino que es JSON el que está jugando este papel. ¿Pero a qué se refiere lo de «intercambio de datos»? A la comunicación asíncrona con el servidor.

Como ejemplo, una aplicación web ofrece información a turistas sobre una lista de ciudades que el usuario puede seleccionar. Si no se utilizara AJAX, cada cambio en los controles de la web, como, por ejemplo, ver qué tiempo hará, cuáles son los vuelos disponibles o los horarios de los autobuses del aeropuerto al centro de la ciudad, generaría una petición al servidor, este devolvería una respuesta y la página se recargaría para construir el nuevo HTML formateado con los nuevos datos solicitados por el usuario. Con AJAX, esta comunicación es transparente para el usuario, es decir, la página no se recargará,

sino que cada dato que solicite el usuario generará una petición asíncrona al servidor (**http request**), este responderá (**http response**) y el dato será presentado al usuario tan pronto como llegue de vuelta. Todo ello sin necesidad de que la página se recargue, por lo que la experiencia de usuario mejora considerablemente.

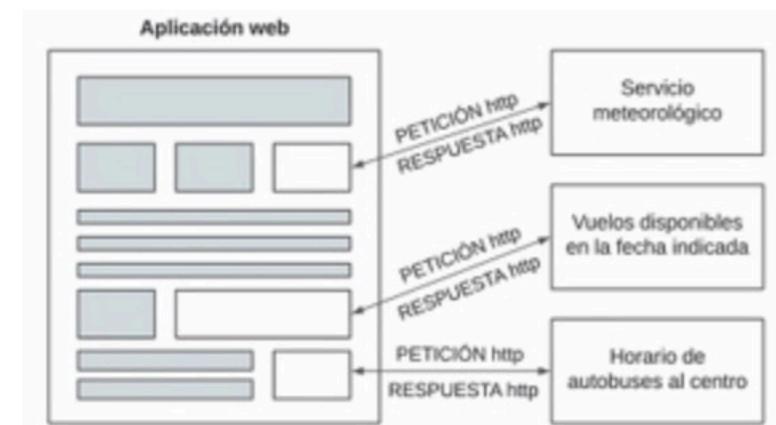


Figura 9.5. Esquema de comunicación asíncrona entre el cliente y el servidor.

Esto supuso en su día una tremenda revolución en el mundo del desarrollo de aplicaciones web. Tanto es así que se la considera como la tecnología desencadenante de una nueva etapa en la forma de entender internet, la llamada Web 2.0. Sin esta tecnología no se entienden funcionalidades ya completamente normales, como el scroll infinito, las reacciones con emoticonos en las redes sociales (y sus estadísticas) o los sistemas de comentarios y respuestas en tiempo real integrados en todo tipo de servicios web. Todas ellas, funcionalidades que eran imposibles de abordar sin AJAX.

En la Tabla 9.1 se recopilan algunas de las ventajas y desventajas de esta tecnología antes de comenzar a programarla.

Tabla 9.1. Ventajas y desventajas de AJAX

Ventajas	Desventajas
<ul style="list-style-type: none"> ■ Simplicidad para gestionar la comunicación por medio de API de terceros. ■ Eficiencia en el intercambio de datos. ■ Favorece la creación de más servicios independientes de la plataforma. ■ Mejora la compatibilidad de servicios al ser una tecnología que no depende del lenguaje de programación del lado servidor. ■ Mejor rendimiento de las aplicaciones al ejecutar su lógica en segundo plano. ■ Gestión inteligente de formularios y validación más robusta. ■ Resolución de problemas complejos de forma sencilla. 	<ul style="list-style-type: none"> ■ Cierta complejidad a nivel de programación, solo recomendada para programadores con bastante experiencia. ■ El intercambio de datos no es más seguro que en escenarios tradicionales. ■ Peor posicionamiento en buscadores de los contenidos que incorporan de forma asíncrona. ■ Incrementa la carga de trabajo de los servidores. ■ Se pierden referencias en el historial de navegación.

9.3.1. Peticiones

Empezar a trabajar con AJAX para realizar peticiones HTTP a otros servicios de interés supone tener claros algunos conceptos.

API para todo

La primera herramienta que puede utilizarse para realizar peticiones son las **API**. De ellas se habló con anterioridad, pero ahora cobran un sentido mucho más especial. Servicios externos de terceros ofrecen una interfaz que incluye una serie de operaciones y una forma concreta de comunicarse con ellos. Las API, a cambio, ofrecen el resultado de las peticiones en formato JSON. Estas peticiones a las API son tan fáciles de implementar que a veces generan problemas de rendimiento en los servidores. Un servicio que se popularice puede recibir millones de peticiones en períodos cortísimos de tiempo, lo cual puede tumbar el servidor.

Nota técnica

Cuando se habla de API, hay que fijarse muy bien en el contexto que nos encontramos, porque podríamos estar simplemente ante una interfaz de programación que proporciona objetos, métodos, propiedades y otros componentes; o podríamos estar ante un servicio externo que proporciona operaciones que podemos solicitar y consumir de forma remota.

Ejemplos de las primeras podrían ser API DOM, API Geolocation, API Notification o API Web Storage; y ejemplos de las segundas podrían ser cualesquiera de los servicios de terceros a los que nos conectamos, como el registro o inicio de sesión de Google, Facebook y Twitter, el servicio de información meteorológica, los datos proporcionados por el Instituto Nacional de Estadística, etcétera.



CORS

Es una estrategia de protección, a la que recurren los servidores que proporcionan una API, resolver peticiones que procedan únicamente de un puñado de dominios que tienen validados. Se trata de la política **CORS** (Cross-Origin Resource Sharing), que actualmente ya forma parte del protocolo HTTP. Esta política consiste en incluir en las cabeceras de los paquetes HTTP datos sobre CORS permitiendo la comunicación entre el cliente y el servidor, pero solo de aquellos dominios que cumplen con las normas establecidas; así, el servidor se descarga de un importante procesamiento de peticiones no confiables.

API Fetch

Al inicio de la implantación de AJAX como vehículo para el consumo de datos vía API en JavaScript, se empezó a utilizar ampliamente el objeto **XMLHttpRequest**. Tal fue su éxito que acabó incluido en el estándar HTML5. Sin embargo, y a pesar de que no han pasado

tantos años, aquella época representa la prehistoria de las aplicaciones web. Hoy en día existe otro modelo mucho más sencillo, eficiente y manejable que se llama **Fetch**.

Actividad propuesta 9.1

XMLHttpRequest

Investiga por tu cuenta cómo se realiza una petición AJAX usando el objeto **XMLHttpRequest** y escribe un pequeño programa que ejemplifique su uso.

La API Fetch permite trabajar con promesas, por lo que es mucho más potente que su predecesora.

9.3.2. Peticiones con Fetch

La API Fetch pone a disposición del programador un método llamado **fetch** al que hay que indicarle la URL destino de la petición. Su valor devuelto es una promesa que termina con éxito si se reciben resultados del destino sin error. Por ejemplo:

```
fetch("https://servicios.ine.es/wstempus/js/ES/OPERACIONES_DISPONIBLES")
  .then(respuesta=>{
    console.log(`Código respuesta: ${respuesta.status} = ${respuesta.statusText}`);
  })
  .catch(error=>{
    console.log(error);
});
```

En el ejemplo anterior se realiza una petición HTTP a la URL que se incluye, que forma parte de la API del Instituto Nacional de Estadística. Esta solicitud pide la lista de operaciones que se pueden solicitar al servicio. En este caso **then** maneja un resultado que se llama **respuesta** y que almacena un objeto de tipo **Response**, y que además representa el paquete HTTP que contiene la respuesta. La salida permite comprobar cómo la comunicación se ha producido sin errores (Figura 9.6).

Código respuesta: 200 = OK

Figura 9.6. Estado devuelto por el objeto de tipo Response.

Además, si se añade **console.info(respuesta)**; se puede ver el aspecto del objeto de respuesta (Figura 9.7).

Tal y como se había estudiado en las promesas, el bloque **catch** captura el error (si se produjera) y lo gestiona según interese. Por ejemplo:

```
fetch("https://fakeurl")
  .then(respuesta=>{
    console.log(respuesta.statusText);
  })
  .catch(error=>{
    console.log(error);
});
```

```
.catch(error=>{
  console.log(error);
});
```

```
Response {type: 'cors', url: 'https://servicios.ine.es/wstempus/js/ES/OPERACIONES_DISPONIBLES', redirected: false, status: 200, ok: true, ...}
  ↴
  body: (...)
```

```
  ↴
  headers: Headers {}
  ok: true
  redirected: false
  status: 200
  statusText: "OK"
  type: "cors"
  url: "https://servicios.ine.es/wstempus/js/ES/OPERACIONES_DISPONIBLES"
  ↴ [[Prototype]]: Response
```

Figura 9.7. Contenido del objeto Response.

Se ha indicado una URL que no existe para comprobar cómo procedería el programa (Figura 9.8).



Figura 9.8. Error producido tras no poder localizar la URL indicada.

9.3.3. Respuestas

Como se ve, se puede acceder a mucha información gracias al objeto de respuesta (tipo **Response**), pero todavía no se conoce con exactitud cuál es el significado de cada una de esas propiedades y métodos miembros que incluye.

Tabla 9.2. Propiedades y métodos del objeto Response

Miembro	Significado
headers	Objeto que contiene las cabeceras HTTP de la respuesta.
body	Contiene los datos de la respuesta.
status	Código de la respuesta a la petición HTTP (estandarizado en el protocolo HTTP).
statusText	Texto que interpreta el código de estado.
ok	Si vale true quiere decir que la petición se produjo sin errores.
redirected	Si vale true indica que la respuesta es fruto de una redirección.

Miembro	Significado
url	Dirección de la que parte la respuesta.
type	Contiene el tipo de respuesta recibida.
redirect()	Redirecciona la respuesta a otra dirección.
clone()	Clona la respuesta en otro objeto que interese.
error()	Devuelve un objeto respuesta asociado a un error de red.
text()	Toma un flujo de objeto respuesta y lo lee hasta completarlo. Devuelve una promesa que resuelta con éxito obtiene los datos de la respuesta en modo texto, siempre decodificada con UTF-8.
json()	Misma funcionalidad que <code>text()</code> , pero devuelve la respuesta en formato JSON.
blob()	Misma funcionalidad que <code>text()</code> , pero devuelve la respuesta como un objeto binario.

Para saber más

Con el siguiente enlace o con el código QR se accede a un listado de códigos de estado del protocolo http:

<https://www.iana.org/assignments/http-status-codes>



Como se ha visto repasando las propiedades y los métodos del objeto de respuesta, puede obtenerse esta en varios formatos, aunque los más comunes son el formato texto y el formato JSON.

Respuestas en formato texto

A continuación se presenta un ejemplo que ilustra su funcionamiento. En la url:

<https://lopegonzalez.es/servicios/vehiculos.php>

hay una rutina que acepta peticiones HTTP de tipo **GET**. Si se pasa el parámetro **matrícula** especificando la matrícula de un vehículo, ofrecerá información en formato texto del vehículo en cuestión (si está registrado en la aplicación).

Para ello hay que usar el método `text()` de la API Fetch sobre el objeto de la respuesta y se obtiene una pieza de texto plano como resultado de la solicitud:

```
fetch("https://lopegonzalez.es/servicios/vehiculos.php?matricula=1702TGG")
  .then(respuesta=>respuesta.text())
  .then(textoDevuelto=>{console.log(textoDevuelto)})
  .catch(error=>{
    console.log(error);
});
```

El primer `then` del código devuelve el resultado de invocar al método `text()` y crea otra promesa que si se resuelve con éxito muestra en consola el texto plano resultado de la solicitud (Figura 9.9).

```
Citroën C4, 110CV, Rojo
>
```

Figura 9.9. Datos de la respuesta en formato de texto plano.

Respuestas en formato JSON

En este caso, el método `json()` del objeto de la respuesta genera una promesa que de resolverse con éxito entrega los datos como un objeto JSON que puede manipularse fácilmente desde JavaScript.

```
fetch("https://lopegonzalez.es/servicios/vehiculos.php?matricula=1702TGG")
  .then(respuesta=>{
    if(respuesta.ok){
      return respuesta.json();
    }
  })
  .then(datos=>{
    console.log(datos);
  })
  .catch(error=>{
    console.log(error);
});
```

El resultado devuelto por esta solicitud puede verse en la consola, en formato JSON (Figura 9.10).

```
▼ {Marca: "Citroën", Modelo: "C4", Cilindrada: "110CV", Color: "Rojo"}
  Cilindrada: "110CV"
  Color: "Rojo"
  Marca: "Citroën"
  Modelo: "C4"
  ▶ [[Prototype]]: Object
>
```

Figura 9.10. Datos de la respuesta en formato JSON.

Actividad propuesta 9.2

Formateo de respuestas

Utilizando el código del ejemplo anterior, intégralo en una aplicación web que consiga presentar los datos en formato HTML y aplique *a posteriori* estilos CSS, de manera que la marca y el modelo sean un elemento **h2**, el color sea una caja coloreada y la cilindrada aparezca en negrita dentro de un **span**.

9.3.4. Peticiones personalizadas

Hasta el momento, y con el objetivo de ser lo más pedagógicos posible, se había realizado la petición más simple posible, indicando solo la URL. Sin embargo, cuando se realiza una petición se está indicando implícitamente un objeto **Request**. Este objeto tiene algunas propiedades que permiten personalizar mucho las peticiones (modificando el paquete HTTP que se envía), por lo que es fundamental conocerlas. En la Tabla 9.3 se recogen algunas de las más interesantes.

Tabla 9.3. Propiedades del objeto Request

Propiedad	Utilidad
url	Dirección destino a la que se realiza la petición.
method	Acepta cualquier método HTTP (GET, POST, PUT, DELETE...), aunque los más frecuentes son los dos primeros; si no se especifica, el valor predeterminado es GET .
headers	Cabeceras de la petición. Tratamiento de la directiva CORS. Existen estas opciones: <ul style="list-style-type: none"> ■ no-cors: restringe las cabeceras que se pueden usar y solo admite los métodos GET, POST y PUT. ■ cors: valor predeterminado que permite todas las respuestas. ■ same-origin: fuerza a que la respuesta sea solo del mismo dominio del que parte la petición. ■ navigate: la petición no es para ser usada con AJAX, sino directamente como dirección del navegador.
mode	Los navegadores normalmente cachean las respuestas para ganar en eficiencia. Pero con esta propiedad puede alterarse su funcionamiento. Las opciones permitidas son: <ul style="list-style-type: none"> ■ default: comportamiento predeterminado. Si la respuesta estaba cacheada la recupera. ■ no-cache: se comprueba si la respuesta es la misma, y si lo es se recoge de la caché; si no, se construye la nueva respuesta. ■ force-cache: se fuerza a usar la caché. Si no está presente se pide al servidor y se cachea para la siguiente ocasión. ■ only-if-cached: solo se usan los datos almacenados en la caché. ■ reload: fuerza a construir una respuesta nueva, pero aun así se cachea. ■ no-store: no cachea nunca la respuesta.
cache	Permite indicar cómo se debe actuar ante las redirecciones: follow (se siguen), error (se devuelve un error), manual (el usuario debe intervenir).
credentials	Ofrece la posibilidad de indicar si se admiten <i>cookies</i> .
integrity	Un algoritmo criptográfico genera un <i>hash</i> para comprobar la integridad de la petición.

Todas estas propiedades del objeto **Request** pueden indicarse especificando su presencia y su valor utilizando el segundo parámetro de **fetch**. A continuación se presenta un ejemplo:

```
fetch("https://lopegonzalez.es/servicios/request.php",{
  method:"POST",
  mode:"same-origin",
  cache:"reload",
  redirect:"follow",
  credentials:"include"
});
```

De la propiedad **headers** no se ha indicado nada en la petición anterior porque merece un repaso algo más profundo.

Como se sabe, por tratarse de una comunicación HTTP, tanto las peticiones como las respuestas incluyen en sus paquetes una serie de cabeceras. Estas cabeceras son objetos que se pueden crear y personalizar hasta cumplir con las necesidades.

Crear un objeto de este tipo no tiene más ciencia que usar su constructor e indicar los valores de las propiedades, tal y como se hace con cualquier otro objeto:

```
let miCabecera = new Headers({
  "Content-Type": "multipart/form-data",
  "WWW-Authenticate": "Basic",
  ...
});
```

La lista de cabeceras HTTP que se pueden especificar es extensa, y se pueden revisar accediendo a esta dirección: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers>.

Pero no hay por qué establecer todas estas cabeceras usando el constructor, sino que se pueden ir modificando dinámicamente por medio de los métodos que ofrece la Tabla 9.4.

Tabla 9.4. Métodos del objeto Headers

Método	Utilidad
append	Añade un nuevo valor a una cabecera existente en un objeto Headers o añade la cabecera si no existe.
delete	Elimina una cabecera de un objeto Headers .
entries	Devuelve un iterador que permite recorrer los pares clave-valor contenidos en el objeto.
get	Devuelve el valor de una cabecera que se especifique.
has	Devuelve true si el objeto Headers tiene una cabecera en concreto.
keys	Devuelve un iterador que permite recorrer las claves de las cabeceras del objeto.
set	Modifica el valor de una cabecera existente en un objeto Headers o añade la cabecera si no existe.
values	Devuelve un iterador que permite recorrer los valores de las cabeceras del objeto.

Lo que se ha hecho, de momento, en relación con las peticiones es establecer quién va a atenderla y formatear el objeto **Request** para personalizarlas. Pero sigue faltando cierta flexibilidad a la hora de enviar datos en la petición.

Por ahora, se asume el comportamiento predeterminado, el método **GET**, para enviar datos en la propia URL de la petición:

<https://lopegonzalez.es/servicios/vehiculos.php?matricula=1702TGG>

Se envía el dato **matricula** con el valor **1702TGG**. De querer enviar más datos, solo habría que encadenar con el carácter **&** más pares clave-valor, por ejemplo de esta manera:

vehiculos.php?matricula=1702TGG&color=azul&cilindrada=110CV

Pero **GET** no es el único método que existe para enviar datos. En general, puede usarse cualquier comando HTTP en la petición. De hecho, **GET** suele utilizarse para pedir datos, **POST** para enviar datos a un servidor, **PUT** para modificar datos o **DELETE** para borrarlos.

Importante



Es necesario conocer cómo funciona el servicio al que se accede para poder realizar las peticiones con las operaciones, los datos y el formato correctos.

Envío de datos de un formulario

Enviar con AJAX un par de valores no es de una complejidad abrumadora, pero sí podría plantear algunas dificultades cuando se trata de enviar un conjunto de datos más extenso como los que se obtienen desde un formulario.

Al enviar datos de un formulario sin utilizar AJAX, generalmente se utiliza el método **POST**, puesto que los datos no son visibles en la URL del navegador. Esto no quiere decir que los datos sean confidenciales, ya que por medio de las cabeceras HTTP podrían verse (usando las herramientas para desarrolladores que incorporan los navegadores, por ejemplo).

Cuando se utiliza AJAX, no solo debe indicarse que el método utilizado es **POST**, sino que se debe indicar en las cabeceras de la petición que se trata de datos procedentes de un formulario e incluir los datos en la propiedad **body**. Aunque los datos se envíen por el método **POST**, internamente es posible indicarlos en la forma clave-valor, tal y como se hace con **GET**. Por ejemplo:

```
fetch("https://lopegonzalez.es/servicios/nuevovehiculo.php",{
  method:"POST",
  headers:{'Content-Type':'application/x-www-form-urlencoded'},
  body:"matricula=1402LGM&marca=Peugeot&modelo=206"
});
```

Además, existe otra forma de enviar los datos, y es por medio del tipo de contenido **application/multipart/form-data**. Lo que diferencia este método del anterior, desde el punto de vista práctico, es que se utiliza preferentemente para el envío de datos de gran tamaño. Para formatear los datos que viajan en esta petición está el objeto **FormData** en el que se pueden ir añadiendo los pares clave-valor del formulario. Por ejemplo:

```
let formulario = new FormData();
formulario.append("matricula","1402LGM");
formulario.append("marca","Peugeot");
formulario.append("modelo","206");
formulario.set("modelo","306");
formulario.delete("modelo");
fetch("https://lopegonzalez.es/servicios/nuevovehiculo.php",{
  method:"POST",
  headers:{'Content-Type':'application/multipart/form-data'},
  body:formulario
});
```

El objeto **formulario** de tipo **FormData**, como se ve en el ejemplo, tiene algunos métodos muy útiles que permiten ir añadiendo pares clave-valor [**append()**], modificar el valor de una clave [**set()**] o eliminar pares [**delete()**]. También están disponibles **get()**, para obtener el valor de una clave, y **has()**, para comprobar si existe una clave.

Además, hacer esto manualmente sería poco eficiente. El objeto se puede construir con todo un formulario de una sola vez, y el constructor se encargará de gestionar los pares clave-valor internamente. Se podría hacer de este modo:

```
let datosFormulario = document.querySelector("form");
datosFormulario.addEventListener("submit",(evento)=>{
  evento.preventDefault();
  let formulario = new FormData(datosFormulario);
  fetch(datosFormulario.getAttribute("action"),{
    method:"POST",
    headers:{'Content-Type':'application/multipart/form-data'},
    body:formulario
  });
});
```

En este ejemplo se recoge todo el formulario usando **querySelector** con la etiqueta **form**. Luego se capture el clic del usuario para enviar el formulario, se cancela su comportamiento predeterminado y se construye la petición AJAX recogiendo del propio formulario la URL de destino. Finalmente, con el objeto **FormData** se construyen los datos que se proporcionarán al bloque **body** de la petición.

Actividad propuesta 9.3

Formulario de alta

Escribe un programa que envíe a un hipotético servicio remoto los datos de registro de un usuario. Los datos que debes solicitar y llenar en tu página HTML son: nombre y apellidos, edad, correo electrónico y teléfono. Los datos son todos obligatorios y deben ser validados antes de ser enviados. Una vez enviados, en el texto del botón enviar debe aparecer el mensaje «¡ENVIADO!».

Envío de datos en formato JSON

Aunque lo habitual para trabajar con la mayoría de los servicios ya se ha repasado [peticiones con envío de datos simples (**GET**) y datos más elaborados (**POST**)], existen ciertos servicios que prefieren recibir los datos de las peticiones en formato JSON.

Programáticamente, esto es algo que ya no supone ninguna dificultad, porque se ha abordado muchas veces: se construye un objeto y se convierte en una cadena de texto JSON usando el conocido método `stringify()`. El resto de la tarea en la petición es especificar el tipo de contenido que viaja en la petición `application/json`:

```
let vehiculo = {
    marca:"Peugeot",
    modelo:"206",
    color:"negro"
}
fetch("https://lopegonzalez.es/servicios/nuevovehiculo.php",{
    method:"POST",
    headers:{'Content-Type':'application/json; charset=UTF-8'},
    body:JSON.stringify(vehiculo)
});
```

9.3.5. Funciones asíncronas con Fetch

En este apartado se cierra el círculo, puesto que se relaciona la asincronía vista con respecto a los tiempos de ejecución de las funciones, con la asincronía vista respecto a cómo se realizan peticiones remotas de forma transparente para el usuario.

Se concluye el estudio de la API Fetch sin darnos cuenta de que al trabajar con promesas podemos aplicarle nuestro operador `await` y nuestras funciones `async`. Los únicos cambios que deben tenerse en cuenta son sustituir la lógica `then` por sentencias `await`, y también que si antes se usaba `catch` para capturar errores, ahora debe utilizarse la estructura `try...catch`. Es preciso reescribir el último ejemplo para obtener la foto completa:

```
let vehiculo = {
    marca:"Peugeot",
    modelo:"206",
    color:"negro"
}
async function solicitarAltaVehiculo(){
    try {
        let respuesta = await fetch("https://lopegonzalez.es/servicios/nuevovehiculo.php",{
            method:"POST",
            headers:{'Content-Type':'application/json; charset=UTF-8'},
            body:JSON.stringify(vehiculo)
        });
        // tratar la respuesta
    } catch(error) {
        console.log(error);
    }
}
solicitarAltaVehiculo();
```

De esta manera tan elegante se sincronizan comportamientos asíncronos en los que intervienen peticiones remotas desencadenadas por acciones del usuario.

