

UNIDAD 10

AJAX

OBJETIVOS

- Asimilar el funcionamiento de los mecanismos de comunicación cliente/servidor utilizando AJAX
- Reconocer las ventajas y desventajas del uso de AJAX
- Aplicar los métodos del API Fetch para realizar peticiones AJAX y procesar sus resultados
- Personalizar las peticiones AJAX modificando las propiedades de los paquetes http
- Identificar los formatos habituales de envío y recepción de datos mediante AJAX
- Procesar los resultados de las peticiones AJAX dependiendo del tipo de datos deseado
- Enviar datos a los servicios de destino de las peticiones usando el formato apropiado

CONTENIDOS

- 10.1 COMUNICACIÓN CLIENTE/SERVIDOR**
- 10.2 INTRODUCCIÓN A AJAX**
 - 10.2.1 ¿QUÉ ES AJAX?
 - 10.2.2 VENTAJAS DE AJAX
 - 10.2.3 DESVENTAJAS DE AJAX
- 10.3 PETICIONES AJAX**
 - 10.3.1 USO DE UNA API PARA ACCEDER A UN SERVICIO WEB
 - 10.3.2 CORS
 - 10.3.3 APIS EN JAVASCRIPT PARA EL USO DE AJAX
- 10.4 REALIZAR PETICIONES MEDIANTE FETCH**
- 10.5 MANIPULAR LA RESPUESTA. OBJETO RESPONSE**
 - 10.5.1 PROPIEDADES Y MÉTODOS DEL OBJETO DE RESPUESTA
 - 10.5.2 DATOS DE LA RESPUESTA
 - 10.5.3 PROCESAMIENTO DE LAS RESPUESTAS
- 10.6 PERSONALIZAR LA PETICIÓN. OBJETO REQUEST**
 - 10.6.1 PROPIEDADES Y MÉTODOS DE REQUEST
 - 10.6.2 ESTABLECER LA CABECERA. OBJETO HEADERS
- 10.7 ENVIAR DATOS CON LA PETICIÓN**
 - 10.7.1 ENVÍO DE DATOS USANDO GET
 - 10.7.2 ENVÍO DE DATOS DE FORMULARIO
 - 10.7.3 ENVÍO DE PARÁMETROS EN FORMATO JSON
- 10.8 USO DE AWAIT/ASYNC CON FETCH**
- 10.9 PRÁCTICAS SOLUCIONADAS**
- 10.10 PRÁCTICAS RECOMENDADAS**
- 10.11 RESUMEN DE LA UNIDAD**
- 10.12 TEST DE REPASO**

10.1 COMUNICACIÓN CLIENTE/SERVIDOR

En la primera unidad (1.1 "Creación de aplicaciones Web", en la página 2) expusimos el funcionamiento general de las aplicaciones web. Hemos también diferenciado lo que se ejecuta en el lado del cliente (**front-end**) y lo que se ejecuta en el lado del servidor (**back-end**).

La explicación dada en esa unidad obviaba una cuestión que se produce cada vez más habitualmente: el hecho de que el propio código front-end pueda realizar peticiones en segundo plano a un servidor.

Supongamos que hemos creado una aplicación que, entre las diversas cosas que ofrece, indica en un cuadro la temperatura que hace en la zona que vive el usuario. Podríamos acudir a un servicio de terceros que, enviando la localidad o las coordenadas GPS, nos devuelva la temperatura en un formato que nos permita recogerlo desde JavaScript y así actualizar el panel.

El resto del contenido no depende de esa petición. Lo ideal es que se realice en segundo plano, sin que el resto de tareas tengan que esperar a que esta termine. Si el servicio de temperaturas cae, lo único que percibirá el usuario es que no se muestra la temperatura, pero el resto de la página funcionará con normalidad.

En definitiva, una aplicación web, desde este prisma, se puede ver como un conjunto de componentes independientes, algunos de los cuales son clientes que realizan peticiones a servidores de Internet. Este paradigma es la realidad más habitual de las aplicaciones web actuales.

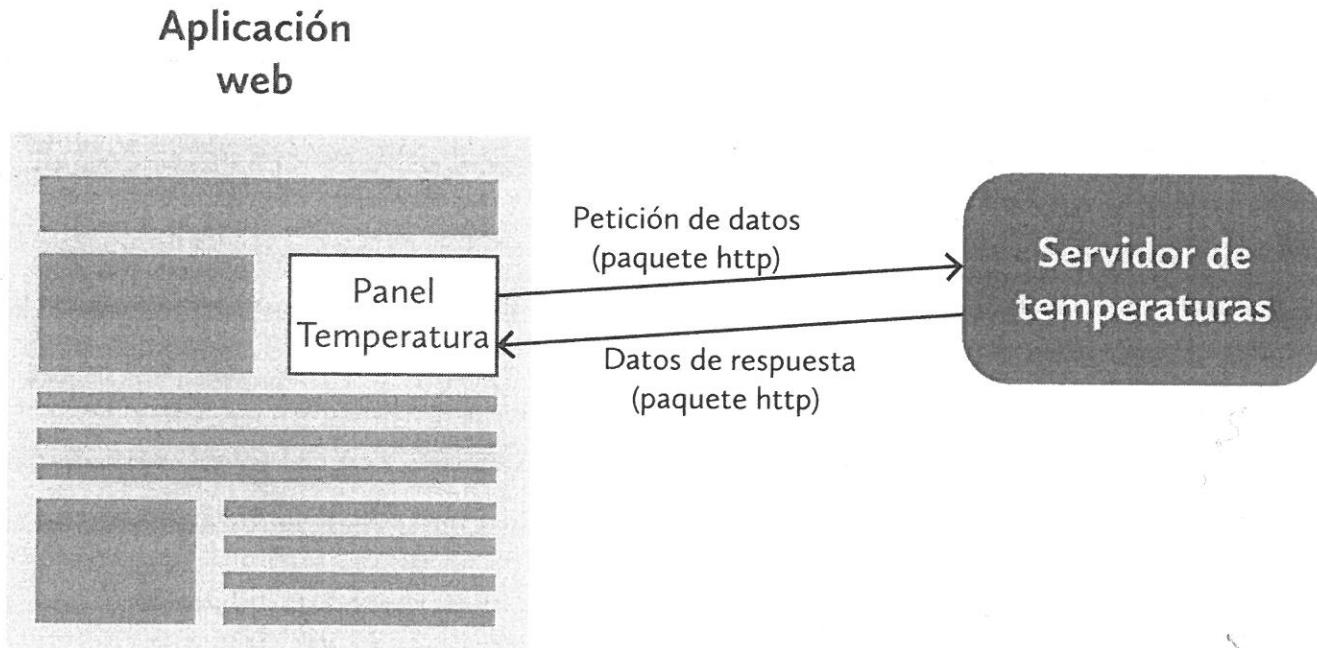


Figura 10.1: Funcionamiento básico de las peticiones AJAX

10.2 INTRODUCCIÓN A AJAX

10.2.1 ¿QUÉ ES AJAX?

Asynchronous JavaScript and XML es el acrónimo que se esconde bajo el término AJAX. Lo más importante de esas siglas es la idea de comunicación asíncrona, que soluciona la idea expresada en el punto anterior, ya que la comunicación asíncrona permite que la petición y la respuesta se ejecute sin interferir en el resto de instrucciones. El hecho de que haga referencia a XML es porque cuando se ideó esta tecnología, XML era el formato documental predominante y los datos se enviaban en este formato. Actualmente el formato predominante es JSON por lo que algunas personas hablan de AJAJ en lugar de AJAX.

El nacimiento oficial de esta tecnología se considera que ocurrió cuando en la versión 5.0 de Microsoft Internet Explorer se dio compatibilidad a un nuevo objeto llamado **XMLHttpRequest**. Inicialmente solo se podía utilizar desde componentes **ActiveX** (una tecnología propietaria de Microsoft para enriquecer las aplicaciones web). Más adelante pasó a ser un objeto integrado en todos los navegadores y ahora ya es parte de la especificación oficial de JavaScript y el DOM.

Realmente el término AJAX fue propuesto por **Jesse James Garret** en un artículo que publicó sobre cómo crear nuevas aplicaciones web y que se basaba en el funcionamiento de **Google Pages**. La tecnología ya existía, pero fue este artículo el que consiguió que se adoptara el nombre oficial.

AJAX ha supuesto una revolución enorme en la creación de aplicaciones web, ya que permite que la aplicación mute y se modifiquen sus elementos en base a las acciones del usuario, sin que este sufra tiempos de espera, ya que la carga de componentes cuyos datos proceden de bases de datos o de otros servicios de Internet se produce en segundo plano. Tanta ha sido la influencia de AJAX que, por parte de muchas personas, se la considera la principal responsable de la creación de la llamada **web 2.0** hace unos años. Lo que es indudable es que ha tenido, y tiene, una enorme influencia en numerosas técnicas de creación de aplicaciones web.

Quizá el ejemplo que mejor explica AJAX son las páginas que usan el llamado **scroll infinito**, que determinó las experiencias de usuario de numerosas aplicaciones web orientadas a redes sociales como **twitter**, **Facebook**, **Instagram** y, sobre todo por ser la precursora de esta técnica, **Pinterest**.

En estas aplicaciones el usuario ve la información ordenada de más reciente a más antigua. Lógicamente no se cargan todos los datos a la vez (en algunos casos sería miles y miles de entradas), sino los primeros, los más recientes. Cuando la aplicación detecta (mediante el evento **scroll**) que el usuario se acerca al final de los datos actuales, se cargan en segundo plano las siguientes entradas. El usuario percibe que antes estaba al final de la barra de desplazamiento y ahora la barra ha crecido y puede seguir descendiendo para ver cada vez más datos.

Un scroll infinito bien hecho hace que el usuario tenga la percepción de que jamás llegará al final, de ahí el nombre de esta técnica.

Hay muchos más ejemplos, todos se basan en que se carga información que procede de otro servidor y que esa carga se hace en segundo plano. En muchas páginas en lugar de un solo panel de aviso tipo "**Cargando...**" avisando de que hay muchos datos por cargar. Hay un aviso de car-

gando por cada componente que se esté cargando. Aunque no todos estén cargados, podremos trabajar perfectamente con los que sí.

Las peticiones se hacen enviando paquetes **http** y la respuesta es también un paquete http cuyo cuerpo contiene datos. Hace unos años los datos estaban siempre en formato **XML**, originalmente solo se daba cabida a este tipo de documento. En la actualidad **JSON** es el formato habitual para los datos, dada su excelente compatibilidad con **JavaScript**, pero, en realidad, se pueden enviar los datos en muchos otros formatos.

10.2.2 VENTAJAS DE AJAX

AJAX aporta importantes ventajas al campo de la creación de aplicaciones web. Entre ellas:

- **Carga de contenido remoto en segundo plano**, permitiendo que la aplicación funcione sin ralentizarse por realizar dicha operación de carga.
- Aporta una gran facilidad para aplicar **paradigmas de interfaces de usuarios novedosas**, eficientes y muy estéticas como páginas con **scroll infinito**, aplicaciones web de página única (**SPA, Single Page Applications**), páginas de integración de múltiples servicios, páginas con información de noticias en tiempo real, navegabilidad independiente de los botones de adelante y atrás, paginación dinámica, etc.
- **Facilidad y versatilidad** para comunicar con APIs de terceros para integrar sus servicios en nuestras aplicaciones.
- **Resolución de problemas reales** de forma sencilla y lógica.
- **Validación de formularios eficaz** al facilitar la detección de errores de forma temprana (a medida que el formulario se va llenando) y de llenado automático de controles dependientes. Por ejemplo, en muchos formularios tras elegir el país en el que vivimos, en segundo plano se rellena una segunda lista con las provincias de ese país concreto. Esa lista de provincias se rellena tras precisar el país, lo que implica una consulta a otro servicio web en segundo plano.
- **Mejor uso del ancho de banda del usuario**. No se satura, ya que se hacen peticiones pequeñas de datos cada vez. Esto es discutible porque no siempre, por razones del propio protocolo http que no es veloz con peticiones pequeñas de datos, es más eficiente.
- En la programación **back-end**, saber que un servicio puede ser destino de peticiones AJAX, hace que los servicios se programen de forma más eficiente e independiente, animando a la creación de Interfaces de Programación de Aplicaciones (**APIs**) que son la base de la programación basada en servicios. Este tipo de paradigma (conocido como **SaaS, Software as a Service**) ha impulsado un mayor desarrollo y eficiencia de las aplicaciones web.
- Facilita la **compatibilidad con servidores back-end de todo tipo de tecnologías**. No importa si el servidor se ha programado en **PHP, ASP.Net, JSP** o cualquier otra tecnología. Solo importa que acepte peticiones **http** y que envíe los datos en formatos estándares como son **XML o JSON**.

10.2.3 DESVENTAJAS DE AJAX

Es importante saber que AJAX tiene sus desventajas. Las principales son:

- Dificultad, en bastantes casos, para que los buscadores indexen todos los contenidos que integramos en nuestras aplicaciones procedentes de peticiones AJAX. Un caso emblemático es el de la paginación dinámica. Si el usuario recibe contenidos de gran tamaño, lo normal es paginarlos para no provocar una gran espera en la llegada de los datos. Si el avance a la segunda página no abandona la actual y recarga los contenidos mediante AJAX, la URL de la aplicación es la misma, y sin embargo, lo que muestra es distinto.
- Es más fácil que un buscador indexe bien el contenido si el avance a la segunda página es mediante un enlace normal. Aunque, muchos buscadores actuales saben lo que ocurre en segundo plano y son capaces de realizar cierta indexación de los contenidos dinámicos. Pero, esto último solo funciona ante peticiones http predecibles en segundo plano.
- Aunque hemos dicho que los conceptos en el uso de AJAX no son difíciles, lo cierto es que la implementación tiene sus dificultades porque requiere de un manejo avanzado de JavaScript en aspectos como funciones callback, promesas, programación asíncrona, programación basada en eventos, etc. En definitiva, manejar el lenguaje JavaScript de forma avanzada. Para nosotros, a estas alturas, esto ya no debería ser un problema.
- Las peticiones AJAX siguen siendo visibles para los usuarios, por lo que son sensibles a ataques malintencionados. No se deben pasar datos críticos como contraseñas en las peticiones. El hecho de que los datos lleguen en segundo plano no significa que sean más seguros, pero nos pueden dar esa sensación dando lugar a problemas de seguridad al programar.
- La barra de navegación (botones de adelante y atrás) no permiten ir al estado anterior en la página. Las peticiones AJAX no se reflejan en la navegación. Lo mismo ocurre con los marcadores, cuando el usuario enlaza en los marcadores la página, solo guarda la raíz de la URL, no el estado actual. Se pierde, pues, una funcionalidad clásica de las aplicaciones web.
- Puede incrementar el tráfico hacia los servidores de Internet. El hecho de que podamos realizar peticiones http a servicios de Internet, puede facilitar la creación de aplicaciones que hagan peticiones http constantemente. Ante este problema muchos servidores utilizan protección anti **CORS**, de la que hablaremos más adelante.

10.3 PETICIONES AJAX

10.3.1 USO DE UNA API PARA ACCEDER A UN SERVICIO WEB

API es el acrónimo de *Application Programming Interface*, interfaz de programación de aplicaciones. Se trata del conjunto de métodos y otros elementos, que un servicio pone a disposición de las aplicaciones para comunicarse con él.

En el caso de AJAX, es muy habitual que servicios públicos de Internet ofrezcan datos para ser usados en las aplicaciones web, pero exijan una forma concreta de comunicarse. Esas especificaciones son la API del servicio. Lo que la API ofrece, a cambio de usarse correctamente, es un conjunto de datos que, hoy en día, suele estar en formato JSON.

Es posible que el servicio esté disponible a cambio de una suscripción, en cuyo caso se nos pedirá autenticarnos antes de resolver la petición de datos. Esta autenticación suele implicar el envío de claves o tokens al servidor.

10.3.2 CORS

CORS es el acrónimo de *Cross-Origin Resource Sharing*, Intercambio de Recursos de Origen cruzado. Las peticiones AJAX son fáciles de automatizar, lo que permitiría realizar cientos o miles de peticiones con unas pocas líneas de código, lo que podrá provocar el colapso del servidor. Para que los servicios de Internet se protejan y solo resuelvan peticiones procedentes de dominios validados, se ideó una norma que ahora es parte del protocolo http. Los datos sobre CORS se colocan en la cabecera de los paquetes http y permiten una vía de comunicación entre el navegador y el servidor web (aunque la petición se haga mediante AJAX) que asegura que se cumplen las normas establecidas.

Una política CORS habitual es que solo se resuelvan peticiones procedentes del dominio en el que está el servicio. Pero hay políticas más sofisticadas. Software de servidor web como Apache o nginx tienen capacidad para modificar esas políticas.

La cabecera http que protege a los servidores de vulnerabilidades causadas por CORS es **Access-Control-Allow-Origin**. Si a esta directiva se le asigna el valor * (asterisco), entonces admite cualquier origen para las peticiones. Se pueden especificar, en su lugar, dominios concretos y así solo se admiten peticiones de estos dominios (serán dominios confiables). Hay otras cabeceras que permiten matizar aún más estas políticas.

10.3.3 APIS EN JAVASCRIPT PARA EL USO DE AJAX

El manejo de AJAX clásico consiste en trabajar con el objeto **XMLHttpRequest**. De hecho, la mayoría de aplicaciones web siguen realizando peticiones mediante ese objeto. En otros casos, no utilizan directamente el objeto sino que utilizan frameworks como **jQuery** que permiten utilizar funciones más sencillas que abstraen y facilitan la forma clásica de realizar estas peticiones.

Hace unos años los dos organismos de estándares de la web (**WHATWG** y **W3C**) incluyeron el uso del objeto XMLHttpRequest en el estándar HTML 5.

Pero, lo cierto es que la API clásica ha sido superada por otra mucho más poderosa y que es compatible con el uso de promesas, lo que la permite aprovechar las nuevas capacidades de JavaScript a la par que adapta el uso de AJAX a los servicios actuales e implementa un modelo de trabajo más sencillo y mantenible. Esta API se llama Fetch.

Fetch es un estándar vivo, forma parte de la norma actual, que se mejora continuamente. Actualmente, todos los navegadores actuales ya han implantado la API Fetch (salvo el obsoleto Internet Explorer).

Cuando decimos que Fetch es una API, lo que queremos decir es que Fetch añade al lenguaje JavaScript una serie de funciones, objetos y elementos de todo tipo que hay que conocer para realizar correctamente la labor de conectar con servicios de las redes. El funcionamiento de esta API será el objeto de estudio el resto de esta Unidad.

10.4 REALIZAR PETICIONES MEDIANTE FETCH

La interfaz de Fetch proporciona un método global que se llama precisamente **fetch**. Este método requiere, al menos, indicar la URL del destino de la petición. El resultado de fetch es una promesa, que se considera resuelta cuando se reciben resultados sin error del destino.

```
fetch(direccionServicio)
    .then(función que recibe el objeto respuesta si la petición
          finaliza bien)
    .catch(función que recibe el error producido durante la petición);
```

Ejemplo:

```
fetch("https://jorgesanchez.net/servicios/nifaleatorio.php")
    .then(response=>{
        console.log(response.status);
    })
    .catch(error=>{
        console.log("Error: " + error);
    });
};
```

Este código realiza una petición **http** de tipo GET a la dirección indicada, la cual proporciona un servicio que devuelve números NIF calculados de forma aleatoria y que se pueden usar para hacer pruebas en bases de datos y otras aplicaciones.

Como **fetch** retorna una promesa, el método **then** permite procesar el resultado. El resultado de la promesa es un objeto de respuesta que se suele conocer como **response** (aunque el nombre le podamos cambiar a voluntad) que representa el paquete (o paquetes) http que proporcionan los datos requeridos. Más adelante podremos profundizar en las posibilidades del objeto de respuesta. En este caso, salvo que haya un problema con el servicio, la propiedad **status** del objeto response devolverá 200, código http que significa **Ok** y ese será el mensaje que veremos en la consola.

El método **catch** recoge el error ocurrido si la petición no concluye bien. Observemos este código:

```
fetch("http://noexiste.com")
    .then(response=>{
        console.log(response.status);
    })
};
```

```
.catch(error=>{
  console.log("Error: "+error);
});
```

Este código retornará un código de error porque la petición no se puede resolver. Concretamente el texto que aparece por consola será:

Error: TypeError: Failed to fetch

10.5 MANIPULAR LA RESPUESTA. OBJETO RESPONSE

Como hemos visto en el apartado anterior, cuando una petición http realizada con `fetch` finaliza correctamente, el método `then` recibe un objeto conocido como **objeto de respuesta** o **response**. Vamos a ver en este apartado las capacidades de este objeto para poder utilizar de la mejor forma posible los datos de respuesta de la petición realizada.

10.5.1 PROPIEDADES Y MÉTODOS DEL OBJETO DE RESPUESTA

Se enumeran en la siguiente tabla las propiedades y métodos de este objeto:

PROPIEDAD O MÉTODO	USO
headers	Obtiene un objeto que contiene las cabeceras http del paquete de respuesta.
body	Cuerpo de la respuesta http. Contiene los datos en sí de la respuesta.
status	Devuelve el código de respuesta de la petición http. Un valor de 200 indica respuesta correcta. Los códigos se corresponden a los estándares del protocolo http ¹ .
statusText	Devuelve un texto descriptivo del código de respuesta de la petición. Por ejemplo, para el código con valor 200 , devuelve Ok .
ok	Con valor true , indica que la petición se resolvió sin problemas. Más concretamente, el valor true significa que la respuesta contiene un código de respuesta http que está entre los valores del 200 al 299 .
redirected	Con valor true, indica que la respuesta es el resultado de una redirección.
url	Devuelve la URL de la que procede la respuesta.

1 La lista de códigos http de respuesta está disponible en la URL:

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

PROPIEDAD O MÉTODO	USO
type	<p>Indica el tipo de respuesta de la petición. Posibles valores:</p> <ul style="list-style-type: none"> ■ basic: Respuesta coherente con la petición original. Muestra todas las cabeceras salvo las relacionadas con cookies. ■ cors: Indica que la respuesta procede de un origen CORS admitido. Hay cabeceras que no se pueden ver. ■ error: Hay error de red. No se describe el error ni disponemos de cabeceras de la respuesta http. ■ opaque: Respuesta para una petición marcada como no-cors; es decir, sin usar CORS. Es una respuesta muy restringida que no permite leer la mayoría de cabeceras.
redirect(URL)	Método que redirige la respuesta a otra URL.
clone()	Clona la respuesta en otro objeto.
error()	Clona la respuesta generando un error de red.
text()	Método que sirve para obtener un flujo de texto de la respuesta. Devuelve una promesa que, cumplida, resuelve obteniendo los datos de la respuesta en formato texto.
json()	Método similar al anterior, pero que trata de convertir la respuesta en un objeto de tipo JSON. Para ello crea una nueva promesa donde, si finaliza de forma correcta, la resolución es dicho objeto JSON.
blob()	Genera una promesa que es resuelta como un objeto binario en el caso de que finalice correctamente.

10.5.2 DATOS DE LA RESPUESTA

Cuando se realiza una petición a un servicio, el resultado deseado siempre es una serie de datos. El formato de esos datos puede ser de muchos tipos. Hoy en día, lo habitual es que se utilice JSON para datos que contienen información etiquetada con metadatos, que son los habituales. Pero se pueden devolver en otros muchos formatos: XML, texto puro, datos binarios, HTML, etc.

10.5.2.1 TIPOS MIME

Los paquetes http disponen de una cabecera llamada **Content-Type** que permite indicar el tipo **MIME** de los datos que van en el paquete. Los tipos MIME (*Multipurpose Internet Mail Extensions, Extensiones de Propósito Múltiple para Correos de Internet*) son una convención aceptada por todos los servidores que indica el tipo de datos.

Su formato habitual se basa en indicar un tipo, seguido de un subtipo separados por una barra de dividir. Ejemplos de tipos MIME habituales usados en peticiones AJAX son:

- **text/plain**. Texto puro.

- **text/html.** Datos en formato HTML.
- **application/json.** Datos en formato JSON.
- **application/javascript.** Datos en formato JavaScript. Históricamente este tipo se usaba también para JSON.
- **application/xml.** Datos en formato XML.

Hay muchos más tipos MIME, la lista completa de tipos MIME es extensísima. La lista oficial la mantiene la **IANA** en la dirección:

<https://www.iana.org/assignments/media-types/media-types.xhtml>

10.5.3 PROCESAMIENTO DE LAS RESPUESTAS

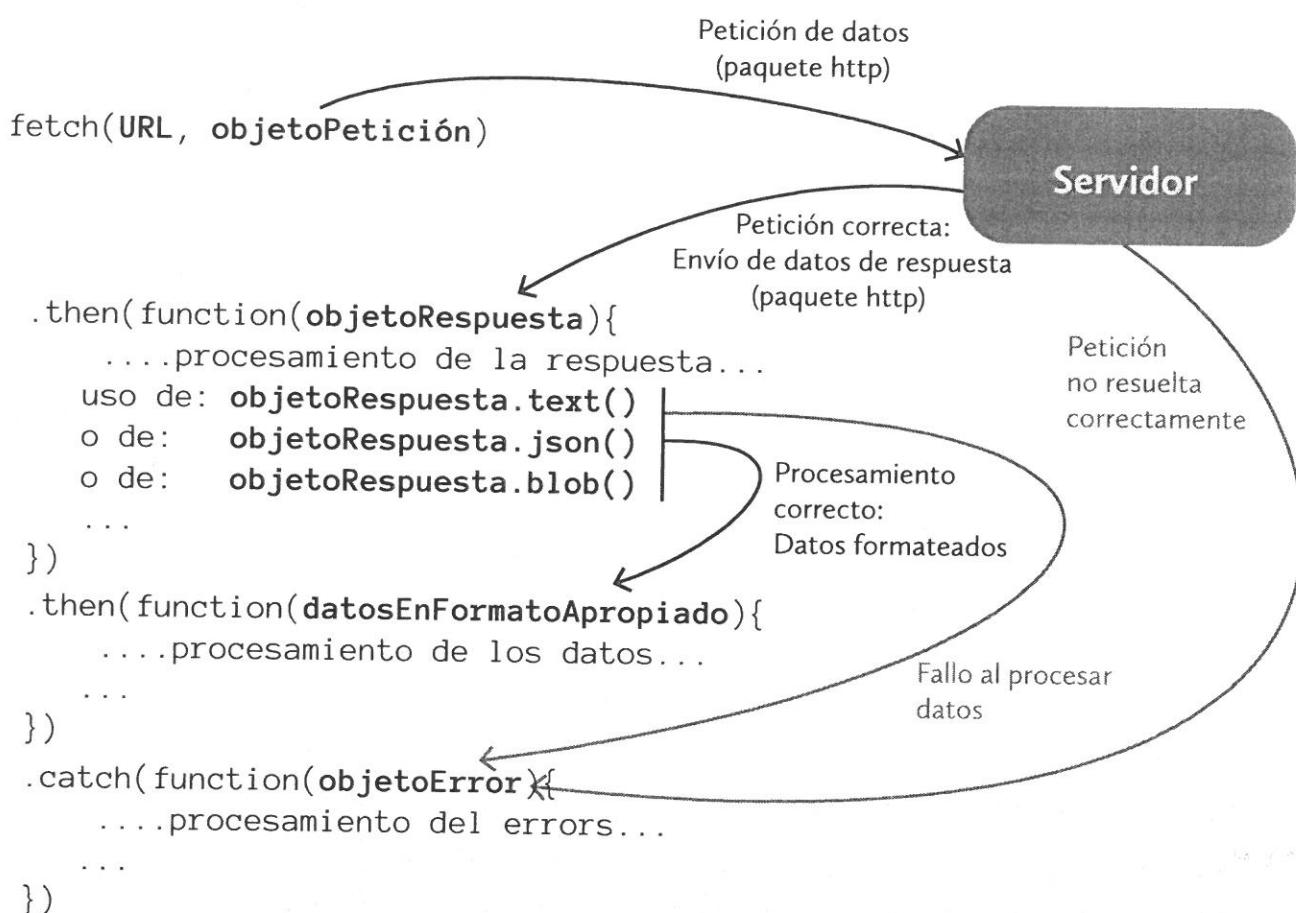


Figura 10.2: Proceso habitual de peticiones AJAX mediante el API Fetch

El proceso de la respuesta con la API Fetch sigue, normalmente, el siguiente proceso:

- [1] Invocar a la función `fetch` indicando la URL a la que realizamos la petición y un objeto **request** (objeto de petición), si es el caso, en el que matizaremos aspectos avanzados de la petición como el método http a utilizar (**GET**, **POST**, **PUT**, **DELETE**, etc.). Del objeto de petición hablaremos más adelante.

- [2] Invocamos de forma encadenada al método **then** el cual usa como parámetro una función callback que recibe como parámetro el objeto de respuesta.
- [3] En la función callback del método **then** procesamos la respuesta obteniendo los datos en el formato deseado a través de los métodos que hacen esa labor: **text**, **json** o **blob**.
- [4] Invocamos de forma encadenada a otro método **then** cuya función callback recibe el resultado de la conversión de datos realizada por los métodos **text**, **json** o **blob** (si los datos son correctos). En el cuerpo de esta función es donde realmente procesaremos los datos.
- [5] Si las promesas anteriores fallan (falla la petición o el procesamiento de los datos) un método **catch**, a través de una función callback, permite procesar el error.

10.5.3.1 MANIPULAR RESPUESTAS EN FORMATO TEXTO

El tipo de datos más sencillo es el texto. Para entender cómo funcionan, en la dirección <https://jorgesanchez.net/servicios/poesia.php> se permite obtener el texto de un poema aleatorio. Cada vez que se realiza un petición de tipo GET a esa dirección, se obtiene un poema aleatorio (puesto que es aleatorio, incluso se puede volver a obtener el mismo).

Como veremos, el problema del texto plano es que no podemos delimitar los datos para clarificarlos. En el caso del poema, se entrega el autor, el título del poema y después los versos. Pero, como el texto no permite distinguir unos datos de otros, solo podremos saber el autor si nos fijamos en la respuesta y vemos que el autor aparece en la primera línea devuelta, el título en la siguiente y después, y hasta el final, aparece el texto del poema.

La API Fetch proporciona un método llamado **text** en el objeto de respuesta. El resultado de este método es el cuerpo de la respuesta pero ya formateada como texto plano. Por supuesto, solo en el caso de que el origen de la petición devuelva los datos como texto, podremos formatearlos a ese formato. De otro modo, no se cumplirá la promesa.

Por lo tanto, podemos crear una aplicación web que muestre un poema aleatorio obtenido de la URL comentada anteriormente, de esta forma:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Poesía del día</title>
</head>
<body>
<header>
    <h1>La poesía del día</h1>
</header>
<main>
    <pre>

    </pre>
</main>
```

```

<div id="error"></div>
<script>
    let pre=document.querySelector("pre");
    let capaError=document.getElementById("error");
    fetch("https://jorgesanchez.net/servicios/poesia.php")
        .then(respuesta=>respuesta.text())
        .then(texto=>{
            pre.innerHTML=texto;
        })
        .catch(error=>{capaError.innerText=error});
</script>
</body>
</html>

```

Lo interesante de esta aplicación es el código:

```
.then(respuesta=>respuesta.text());
```

Este código usa una función flecha que devuelve el resultado de la función **text**, la cual crea otra promesa. Si la promesa se cumple de forma exitosa, el texto ya interpretado en forma de texto se pasa a la segunda función **then**. Esa respuesta se usa para llenar un elemento de tipo **pre** ya que, en este caso, es el ideal para respetar los saltos de línea originales del texto, fundamentales en el caso de la poesía.

10.5.3.2 MANIPULAR RESPUESTAS EN FORMATO JSON

El método **JSON**, del objeto de respuesta, genera una promesa que, en caso de resolverse positivamente, resuelve entregando los datos ya como objeto **JSON**. El formato **JSON** es el más utilizado actualmente para entregar datos a peticiones **AJAX** debido a su facilidad de manipulación desde **JavaScript** y a su versatilidad.

Como ejemplo, la dirección <https://jorgesanchez.net/servicios/nifaleatorio.php> permite obtener en formato **JSON** un NIF aleatorio (puede obtener una serie de ellos utilizando parámetros, como veremos más adelante). Este servicio está creado para hacer pruebas pero, evidentemente, no se pueden usar esos números para otra finalidad. Este podría ser un ejemplo de la respuesta del servicio:

```
[
  {
    "tipo": "dni",
    "numero": "74147019X"
  }
]
```

Como vemos, lo que devuelve es un array de objetos (aunque solo devuelve un elemento del array) en el cada elemento es un objeto con dos propiedades: tipo y número. Para crear una aplicación que requiere un NIF aleatorio y le muestre, el código sería:

```
<!DOCTYPE html>
<html lang="es">
```

```

<head>
    <meta charset="UTF-8">
    <title>NIF aleatorio</title>
</head>
<body>
<p id="nif"></p>
<script>
var capaNIF=document.getElementById("nif");
fetch("https://jorgesanchez.net/servicios/nifaleatorio.php")
    .then(response=>{
        if(response.ok){
            return response.json();
        }
        else{
            throw new Error("Los datos no llegaron bien");
        }
    })
    .then(listaNIFs=>{
        capaNIF.textContent=listaNIFs[0]["numero"];
    })
    .catch(error=>{
        capaNIF.textContent="Error: "+error;
    });
</script>
</body>
</html>

```

Como podemos observar, el segundo método `then` recibe los datos ya en formato directamente manipulable desde JavaScript tras haber ejecutado el método `response.json()`. Al parámetro de este segundo `then` le hemos llamado `listaNIFs`, ya sabiendo que efectivamente lo que se recibe es un array de objetos. Simplemente mostramos entonces la propiedad `numero` del primer elemento de dicho array.

10.5.3.3 MANIPULAR RESPUESTAS EN FORMATO BINARIO

Los datos de tipo **Blob** (*Binary Large Objects*) se corresponden a datos inmutables procedentes de ficheros. Se utiliza cuando la respuesta a una petición son datos procedentes de ficheros. El caso más típico es recoger una imagen de un servidor que es capaz de enviarla como respuesta a una petición http.

Es el caso de **Unsplash**, servicio muy popular de Internet para imágenes de alta calidad que permite obtener una imagen aleatoria de la dirección usando la siguiente URL:

<https://source.unsplash.com/random>

Normalmente este tipo de peticiones se rechazan cuando son de tipo CORS. Pero en este caso, el servicio **unsplash** sí las acepta.

La respuesta se debe procesar con el método **blob()** del objeto de respuesta, el cual genera un objeto de tipo **Blob**. Estos objetos disponen de varios métodos. El más interesante es **type** que obtiene el tipo MIME del objeto, por ejemplo: **imagen/jpeg**. Por otro lado el objeto predefinido y disponible desde JavaScript, **URL**, dispone de un método llamado **createObjectURL** al que se le pasa un objeto Blob y devuelve un texto que representa la URL virtual con la que podemos acceder a ese recurso como si fuera una imagen que se carga desde Internet. De esa forma, se permite acceder a la imagen desde el propio código HTML.

Veamos como poner en práctica todo lo comentado:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Imagen del día</title>
</head>
<body>
<main>
    <p>Esperando datos...</p>
</main>

<script>
var main=document.querySelector("main"); //Capa main
fetch("https://source.unsplash.com/random")
    .then(response=>{
        if(response.ok){
            return response.blob();
        }
        else{
            throw new Error("Los datos no llegaron bien");
        }
    })
    .then(blob=>{
        console.log(blob.type);
        let imagen=document.createElement("img");//nuevo elemento img
        //Asignamos al atributo src del elemento creado, la URL
        //virtual de la imagen descargada
        imagen.setAttribute("src",URL.createObjectURL(blob));
        main.innerHTML=""; //borramos mensaje de "Esperando"
        main.appendChild(imagen); //colocamos la imagen en la capa main
    })
    .catch(error=>{
        main.textContent=error;
    });
</script>
</body>
```

```
</html>
```

El primer método **then** comprueba que el código de respuesta es *Ok* y usa el método **blob** para generar una promesa que obtenga el objeto **Blob** devuelto por **Unsplash**.

El segundo método **then** recoge ese objeto, una vez que se cumple la promesa, y por consola escribirá el tipo MIME del objeto (debería ser *imagen/jpeg*). Además, crea un nuevo elemento de tipo **img** y modifica el atributo **src** de ese elemento para que recoja la URL que representa al objeto Blob obtenido. Para ello, se usa **URL.createObjectURL(blob)**. Antes de colocar la imagen se borra el contenido de la capa main (**main.innerHTML=""**) para que desaparezca el mensaje con el texto *Esperando datos....* Inmediatamente después añadimos la imagen a la capa main.

10.6 PERSONALIZAR LA PETICIÓN. OBJETO REQUEST

Cuando se realizan peticiones mediante el método **fetch**, además de la URL, podemos pasar como parámetro un objeto de tipo **Request**. Este objeto permite modificar el paquete http que se envía con la petición para que recoja aspectos que nos interesan y que la petición sea lo más ajustada posible a nuestros intereses.

10.6.1 PROPIEDADES Y MÉTODOS DE REQUEST

Las propiedades de las que disponemos en los objetos **request** son:

PROPIEDAD	USO
url	Propiedad obligatoria que contiene la URL destino de la petición.
method	Método http que se usará en la petición. Es el comando http de la petición. Normalmente se usa GET o POST , pero es válido cualquier comando http: PUT , DELETE , HEADER , DELETE , etc. Por defecto el método que se usa es GET .
headers	Permite indicar un objeto de tipo Headers que permite modificar las cabeceras http de la petición.
mode	Indica si se usa CORS en la petición. Posibilidades: <ul style="list-style-type: none"> ■ same-origin. Solo se acepta respuesta del mismo dominio en el que se ha realizado la petición. Si la petición se dirige a otro dominio, se devuelve un error. ■ cors. Se permiten respuestas de dominios cruzados. Es el valor por defecto. ■ no-cors. Solo se admiten métodos GET, POST o PUT y limita las cabeceras http que se pueden usar. ■ navigate. Indica que la petición se usa como URL del navegador y no para ser usada como parte de una petición AJAX.

PROPIEDAD	USO
cache	<p>Indica el modo de caché de la respuesta. Los navegadores cachean las respuestas a peticiones previas para acelerar la navegación. Esta propiedad permite calibrar cómo deseamos el cacheado. Posibilidades:</p> <ul style="list-style-type: none"> ■ default. Valor habitual. El navegador comprueba si la respuesta a la petición está en caché. Si es así y el caché es reciente, no realiza realmente la petición y recoge los datos del caché. ■ no-store. Obliga siempre a traer la respuesta del servidor. No cachea ninguna respuesta. ■ reload. Obliga a traer la respuesta del servidor y no de la caché, pero la respuesta se guarda en caché por si se desea usar en las siguientes peticiones. ■ no-cache. Si la respuesta a la petición ya se había cacheado, pide al servidor una comprobación de si la respuesta es la misma, si es así se toma de la caché. ■ force-cache. Fuerza a usar la caché. Si hay respuesta a esa petición en la caché, se usa. Si no, se pide al servidor y se cachea para la siguiente vez. ■ only-if-cached. Solo se admiten los datos si proceden de la caché. En otro caso, si la respuesta no está cacheada, devuelve un código 504 de petición.
redirect	<p>Indica cómo se deben de procesar las respuestas en el caso de que procedan de redirecciones. Posibilidades:</p> <ul style="list-style-type: none"> ■ follow. Se siguen las redirecciones sin problemas. ■ error. Si hay redirecciones, se retorna un error. ■ manual. Se manejan de manera manual por el usuario (debería hacer un clic sobre la respuesta antes de la redirección). <p>En peticiones AJAX, solo las dos primeras nos interesan.</p>
credentials	<p>Permite especificar si se admiten cookies en la petición. Posibilidades:</p> <ul style="list-style-type: none"> ■ omit. Nunca se envían ni reciben cookies. ■ same-origin. Es el valor por defecto, se admiten si el origen y el destino de la petición están en el mismo dominio. ■ include. Siempre se admiten cookies.
integrity	<p>Almacena una cadena de integridad creada con un algoritmo hash de criptografía (se admite sha256, sha384 y sha512), para validar la integridad de la petición.</p>

Ejemplo de petición usando objeto de tipo **request**:

```
fetch("http://direccionalvalida.com/", {
  method: "POST",
  mode: "cors",
  cache: "no-cache"
})...
```

Además los objetos **request** disponen de los mismos métodos, prácticamente, que los objetos **response**: **json()**, **blob()**, **text()**, **clone()**, etc. Sin embargo, no suele ser necesario su uso porque lo que hacen es procesar el cuerpo de la petición y ese proceso se requiere de las respuestas, no de las peticiones.

10.6.2 ESTABLECER LA CABECERA. OBJETO HEADERS

La propiedad **headers** de la petición (objeto **request**) permite establecer todas las cabeceras http que nos proporciona el protocolo. Tanto el objeto de respuesta como el de petición poseen una cabecera de tipo **headers**.

Para crear objetos de tipo **Headers** disponemos de varios constructores. Por ejemplo, podemos indicar un objeto que permite indicar propiedad http y valor de la misma:

```
var cabecera=new Headers({
  "Content-Type": "application/json",
  "Accept-Charset": "utf-8"
});
```

Además, los objetos de cabecera disponen de estos métodos:

PROPIEDAD	USO
append	Añade una entrada al objeto de cabecera. Es decir, añade una nueva cabecera http. Para ello recibe el nombre de la cabecera y su valor. Ejemplo: <code>objeto.append('Content-Type', 'application/json');</code> Si el nombre de cabecera http no es correcto, se provoca una excepción de tipo TypeError .
delete	Retira una cabecera del objeto de cabeceras. Ejemplo: <code>objeto.delete('Content-Type');</code>
get	Devuelve el valor de una cabecera en el objeto. Por ejemplo: <code>objeto.get('Content-Type');</code> Podría retornar, por ejemplo: ' application/json '. Si la cabecera de la que se pide el valor no está en el objeto, devuelve null .

PROPIEDAD	USO
set	Modifica una cabecera existente y la otorga un nuevo valor: <pre>objeto.set('Content-Type', 'text/xml');</pre> <p>Si la cabecera no existe, se añade (como hace append). Si el nombre de cabecera http no es válido, se provoca una excepción de tipo TypeError.</p>
has	Método que comprueba si el objeto de cabeceras contiene una entrada concreta. De ser así, devuelve true . <pre>objeto.has('Content-Type');</pre> <p>Si el objeto tiene esa cabecera, devuelve true.</p>
entries	Devuelve un objeto iterable que permite recorrer todas las cabeceras del objeto: <pre>for(let [clave,valor] of objetoHeader){ console.log(clave+", "+valor); }</pre>

Hay que tener en cuenta que también las respuestas poseen este mismo objeto y que, lógicamente, sus propiedades y métodos son los mismos.

10.7 ENVIAR DATOS CON LA PETICIÓN

En muchas ocasiones, los servidores de Internet que otorgan servicios, requieren o permiten enviar datos para determinar de una forma más ajustada los datos que debe devolver. De forma clásica, esos datos son pares nombre/valor, listas parámetros con valores concretos. Pero actualmente se admite enviar datos incluso en formato de texto JSON.

Los documentos HTML siempre han permitido el envío de este tipo de datos a servidores externos a través de los formularios. De manera clásica, los datos se envían usando los comandos http GET o POST. El método GET envía los datos en la URL y POST lo hace en el cuerpo del mensaje http. Esta diferencia puede hacer pensar que POST es más seguro al ocultar más los datos que envía, pero realmente no lo es. Los datos viajan de forma plana y solo si los ciframos estarán realmente protegidos.

Actualmente, los comandos http se asocian a verbos que requieren un tipo concreto de acción. Así **GET** se asocia a una petición de datos y **POST** a un envío de datos a un servidor. Otros comandos http son: **PUT** (modificación de datos), **DELETE** (borrado de datos), **PATCH** (modificación parcial de datos) o **HEAD** (petición de cabecera).

Evidentemente hay que conocer la API del servicio final para saber qué método usar, cómo enviar los datos y cómo debemos recibirlas, dependiendo de lo que el servicio es capaz de aceptar.

10.7.1 ENVÍO DE DATOS USANDO GET

En el caso de las peticiones **GET**, se envían los parámetros en la URL en la parte conocida como **cadena de consulta**.

Por ejemplo, en el caso del servicio que vimos en apartados anteriores y que permite mostrar una lista de números de identificación fiscal españoles (NIF) aleatoria. Si la petición se hace sin enviar parámetros devuelve un solo NIF. Pero, si indicamos un parámetro llamado **n** con un valor positivo, por ejemplo **10**, se nos devolverán diez NIFs aleatorios. Además, un segundo parámetro llamado **tipo** admite los valores: **dni**, **nie** o **todo**. Si indicamos el valor **nie** en el parámetro **tipo** se nos devolverán diez números de identificación de extranjeros (NIE) producidos de forma aleatoria:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>NIEs aleatorios</title>
</head>
<body>
<main>
    <ul>

        </ul>
</main>
<script>
    let lista=document.querySelector("ul");
    fetch("https://jorgesanchez.net/servicios/nifaleatorio.php" +
        "?n=10&tipo=nie",{
        method:"get",
        mode:"cors"
    })
    .then((resultado)=>resultado.json())
    .then(datos=>{
        for(let nie of datos){
            let li=document.createElement("li");
            li.textContent=nie.numero;
            lista.appendChild(li);
        }
    })
    .catch((error)=>{
        document.querySelector("main").textContent=
            "Error: "+error;
    });
</script>
</body>
</html>
```

En caso de que la petición se resuelva correctamente, se reciben los 10 números en formato JSON. Tras procesarlos, se muestran en forma de lista.

10.7.2 ENVÍO DE DATOS DE FORMULARIO

10.7.2.1 CODIFICACIÓN ESTILO URL

En el caso de POST, los datos no se envían en la URL sino en el cuerpo del propio paquete http. En el caso clásico, los datos POST proceden de datos de formulario e incluso aunque no procedan, debemos marcar el paquete para indicar que los datos se envían al estilo de los formularios. Esa forma usa un formato similar al del método GET. En las peticiones AJAX se debe asociar esos valores a la propiedad **body**. Por ejemplo:

```
fetch("https://jorgesanchez.net/servicios/validarNIF.php", {
  method: "post",
  headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
  body: "nif=12345678A&tipo=dni"
})
```

En el código anterior, lo interesante es la cabecera:

```
'Content-Type': 'application/x-www-form-urlencoded'
```

Esta cabecera es la que permite indicar que los datos usan la forma: **parámetro=valor** y que se encadenan los valores mediante el carácter ampersand: &. De esa forma, se reconocerán los parámetros **nif** y **tipo** de forma apropiada en el servicio final. Suponiendo, eso sí, que este servicio efectivamente esté esperando los datos en ese formato.

La forma en la que se envían los datos es la que aparece en el código anterior:

```
body: "nif=12345678A&tipo=dni"
```

10.7.2.2 USO DE OBJETOS FORMDATA

Hay otra forma de enviar datos de formulario que es el que se asocia al tipo MIME: **application/multipart/form-data**. Es un estilo de envío de datos por pares pensado para transmitir datos de gran tamaño o de tipo binario. Si queremos enviar, por ejemplo, un archivo, este es el formato apropiado. No obstante, se puede utilizar para enviar datos más sencillos.

JavaScript proporciona un objeto llamado **FormData** que facilita la preparación de los datos en este formato. La creación de un objeto de este tipo se hace de esta forma:

```
var form=new FormData();
```

Añadir un nuevo par parámetro/clave se hace con el método **append**:

```
form.append("nif","12345678A");
form.append("tipo","dni");
```

Si quisieramos borrar un parámetro:

```
form.delete("tipo");
```

Si quisieramos modificar:

```
form.set("nif","98765432B");
```

Saber si hemos grabado un valor lo hace el método `has`, que devuelve `true` o `false`.

```
console.log(form.has("nif")); //Escribe true
```

Obtener el valor ya grabado de un parámetro se hace con `get` (si el parámetro no existe, devuelve `null`)

```
console.log(form.get("nif")); //Escribe 98765432B
```

Finalmente, podemos recorrer todos los parámetros del objeto, por ejemplo, con el bucle `for..of`:

```
<script>
let form=new FormData();
form.append("nombre","Jorge");
form.append("edad","47");
form.append("profesion","profesor");
for(let [dato,valor] of form){
    console.log(`$ {dato}=$ {valor}`);
}
</script>
```

En definitiva, es un objeto con una interfaz muy parecida a otros objetos que representan conjuntos clave/valor como el ya comentado objeto **Headers**.

El servicio <https://jorgesanchez.net/servicios/validarNIF.php> permite enviar, vía POST, un NIF y entonces devuelve, en formato JSON, un objeto con una única propiedad llamada **error**. A su vez, error tiene dos propiedades: **codigo** que devuelve el código de error (vale cero si no hay error) y **mensaje** que contiene el mensaje de error. Por lo tanto, el código para validar el NIF (incorrecto) **123456789A** usando ese servicio es el siguiente:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Validar >NIF</title>
</head>
<body>
<main>
    <p></p>
</main>
<script>
    let p=document.querySelector("p");
    let form=new FormData();
    form.append("nif","12345678A");
    fetch("https://jorgesanchez.net/servicios/validarNIF.php",{
        method:"post",
        body:form
    })
    .then((resultado)=>resultado.json())

```

```

        .then(datos=>{
            p.textContent=datos.error.mensaje;
        })
        .catch((error)=>{
            document.querySelector("main").textContent=
                "Error: "+error;
        });
    </script>
</body>
</html>

```

Los objetos FormData se pueden construir a partir de los controles de un formulario. De esta forma se pueden recoger de golpe todos los controles del formulario. Basta con usar esta sintaxis:

```
new FormData(elementoForm);
```

El siguiente código crea una aplicación web en la que podremos escribir un NIF en un control de texto y, al pulsar un botón (*Comprobar*) se comprueba si es válido o no:

```

<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>Validación de NIF</title>
    <style>
        body{
            text-align: center;
        }
    </style>
</head>
<body>
    <form action="https://jorgesanchez.net/servicios/validarNIF.php"
          method="POST">
        <label for="nif">Escriba el NIF que desea validar</label><br>
        <input type="text" id="nif" name="nif"><br>
        <button>Comprobar</button>
    </form>
    <div id="resultado">
    </div>
    <script>
        var form=document.querySelector("form");
        var capaRes=document.getElementById("resultado");
        var tNIF=document.getElementById("nif");
        form.addEventListener("submit", (ev)=>{
            ev.preventDefault();
            let data=new FormData(form);
            fetch(form.getAttribute("action"), {

```

```

        method: form.getAttribute("method"),
        body: data
    })
    .then(respuesta=>respuesta.json())
    .then(datos=>{
        capaRes.textContent=datos.error.mensaje;
    })
    .catch(err=>{
        capaRes.textContent="Error: "+err;
    })
);
tNIF.addEventListener("focus", ev=>{
    capaRes.textContent="";
    tNIF.selectionStart=0;
    tNIF.selectionEnd=tNIF.value.length;
});
</script>
</body>
</html>

```

En este código, el método fetch usa los propios atributos del formulario para determinar el destino de la petición y el método de paso. Eso permite que sea la etiqueta form la que controle los detalles del destino.

El evento submit se ha capturado obligando a que no se envíen directamente los datos al destino (mediante el método de evento **preventDefault**) y haciendo la petición fetch que recoge los datos JSON del destino y les muestra en una capa (**capaRes**).

Para que el manejo del usuario de esta página sea más cómoda y permita validar muchos más números, cuando el usuario se disponga a escribir un nuevo NIF, se borra el mensaje anterior y se selecciona el texto del cuadro (métodos **setSelectionStart** y **setSelectionEnd**) para facilitar su borrado.

10.7.3 ENVÍO DE PARÁMETROS EN FORMATO JSON

No siempre el servicio destino de nuestra petición usa los datos en el formato clásico de los formularios. Hay servicios que aceptan los datos en formato JSON. Para estos servicios, lo que tenemos que hacer es preparar los datos a enviar en un objeto y después convertirlo a formato de texto JSON mediante el método **JSON.stringify**.

Para probar este tipo de envío podemos usar un servicio disponible en la URL:

<https://jsonplaceholder.typicode.com/>

Esta dirección permite, precisamente hacer pruebas de peticiones simulando un sistema de mensajes (**posts**) creados por usuarios. Podemos hacer peticiones para obtener datos, añadir datos, modificar, etc. En la URL anterior vienen las instrucciones de uso.

Por ejemplo, mediante una petición de tipo GET podemos obtener la lista de todos los usuarios, utilizando <https://jsonplaceholder.typicode.com/posts>

Si añadimos el número de post: <https://jsonplaceholder.typicode.com/posts/5> nos devuelve el post número 5 (hay 100 para probar).

Mediante peticiones de tipo POST podemos añadir nuevos datos. En realidad no se añaden (es un servicio par practicar, no es real), pero parece como si sí lo hicieran. Por ejemplo, este código hace como si el usuario número 5 añadiera un nuevo post con el título "*Mi mensaje*" y el texto "*Este es un mensaje de prueba*".

```
let data={  
    title:"Mi mensaje",  
    body:"Mensaje de prueba",  
    userId:5  
}  
  
fetch('https://jsonplaceholder.typicode.com/posts', {  
    method: 'POST',  
    body: JSON.stringify(data),  
    headers: {  
        "Content-type": "application/json; charset=UTF-8"  
    }  
})  
.then(resp => resp.json());  
.then(json => {console.log(json)});  
.catch(err => {console.log(err)});
```

El resultado de la petición es:

```
{  
    id: 101,  
    title: 'foo',  
    body: 'bar',  
    userId: 1  
}
```

Realmente no hay un post con el número 101, se nos muestran esos datos simulando la que daría un servidor real.

Lo interesante es que este servidor ha recibido los datos en formato JSON mediante la función `stringify`. Para saber si hay que enviarlos de esta forma o en la forma de los formularios (`FormData`) necesitamos conocer el funcionamiento del servidor destino de nuestras peticiones.

10.8 USO DE AWAIT/ASYNC CON FETCH

Puesto que la API Fetch es compatible con promesas, es posible manejarla mediante la notación `await/async` del estándar ES2017. Esta notación, para muchos desarrolladores, es más legible.

ble y facilita mucho el mantenimiento del código. El ejemplo visto en el apartado anterior para enviar datos de un supuesto POST al servicio de pruebas tripicode, sería, en esta notación, de esta forma:

```
let data={  
    title:"Mi mensaje",  
    body:"Mensaje de prueba",  
    userId:5  
}  
  
async function peticion(){  
    try{  
        const resp=await  
            fetch('https://jsonplaceholder.typicode.com/posts', {  
                method: 'POST',  
                body: JSON.stringify(data),  
                headers: {  
                    "Content-type": "application/json; charset=UTF-8"  
                }  
            });  
        const json=await resp.json();  
        console.log(json)  
    }  
    catch(err){  
        console.log(err);  
    }  
}  
peticion();
```

Si comparamos este código con el resto de conexiones realizadas en esta unidad, lo que antes suponía usar el método **then**, ahora son sentencias **await**. La captura de errores no requiere del método **catch**, si no de la estructura **try..catch**.

Puesto que esta notación está ya muy implantada, es cuestión de gustos cuál de las dos notaciones utilizar para nuestras peticiones

10.9 PRÁCTICAS SOLUCIONADAS

Práctica 10.1: Validación de NIF

- Crea un formulario que permita escribir un nombre, apellidos, email y un NIF, con una estética similar a la de la siguiente imagen:

The screenshot shows a user registration form titled "Registrar usuario". It contains four input fields: "Nombre", "Apellidos", "Email", and "NIF", each with a corresponding text input box. Below the input fields is a large, semi-transparent button labeled "Enviar".

Figura 10.3: Panel inicial de registro del usuario

- Inicialmente los cuadros de entrada de texto poseerán un borde rojo (para indicar que el contenido aún no es válido).
- El nombre y apellidos se consideran válidos si al menos hay dos letras.
- El email si hay un símbolo de arroba (@) y a su izquierda y derecha, al menos una letra.
- Se hacen las validaciones a medida que vamos escribiendo, con cada tecla pulsada.
- La validación del NIF utiliza el servicio <https://jorgesanchez.net/servicios/validarNIF.php> el cual recibe usando el comando http POST, un parámetro llamado **nif** (en minúsculas) con el NIF a validar. Responde en forma JSON con un objeto llamado **error**, el cual indica en su propiedad **codigo** (será cero si no hay error), el código del error y en la propiedad **mensaje** el mensaje de error.
- El botón de **Comprobar** permanece desactivado hasta que todos los cuadros de texto sean válidos. Cuando lo sean, el botón se activa.
- Al hacer clic en el botón **Comprobar**, simplemente aparecerá un mensaje indicando que todo es correcto.
- Al hacer clic en el mensaje, comenzamos de nuevo.

SOLUCIÓN: PRÁCTICA 10.1

El código de la solución es extenso. Vamos a explicarlo archivo a archivo.

En primer lugar, el código HTML (sería el archivo **index.html**) simplemente tiene los controles de formulario, carga el CSS (en el directorio **css**, el archivo **estilos.css**) y carga el JavaScript (en el directorio **js**, el archivo **accion.js**). Además, contiene los elementos de formulario y una capa para mostrar los errores. Para mostrar el mensaje final de datos correctos, tenemos una capa llamada **telón** que, a través de CSS, tapará toda la página con color negro al 50% de opacidad. Encima de esta capa se muestra la capa **mensaje**. Estas dos capas inicialmente están ocultas. El código de **index.html** es:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width,
initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Registrar usuario</title>
    <link rel="stylesheet" href="css/estilos.css">
</head>
<body>
    <main>
        <h1>Registrar usuario</h1>
        <form action="#" autocomplete="off">
            <label for="nombre">Nombre</label><br>
            <input type="text" id="nombre" name="nombre"><br>
            <label for="apellidos">Apellidos</label><br>
            <input type="text" id="apellidos" name="apellidos"><br>
            <label for="email">Email</label><br>
            <input type="text" id="email" name="email"><br>
            <label for="NIF">NIF</label><br>
            <input type="text" id="NIF" name="NIF"><br>
            <button id="boton" disabled>Enviar</button>
        </form>
    </main>
    <div id="error">

    </div>
    <div id="telon"></div>
    <div id="mensaje">
        <p>Los datos se han enviado correctamente</p>
    </div>
    <script src="js/accion.js"></script>
</body>
</html>
```

El archivo `estilos.css` contiene este código:

```
main{
    width:100%;
    max-width:400px;
    margin:auto;
}
h1{
    text-align: center;
    font-family: sans-serif;
}
label{
    text-align: left;
    font-family: sans-serif;
    color:gray;
}
input{
    width:97%;
    font-size:1.2em;
    padding:.2em;
    margin-bottom: .5em;
    border-radius:5px;
    border:1px solid red;
    outline:none;
}
button{
    width:100%;
    font-size:1.5em;
    border:none;
    background-color: gray;
    padding:.4em;
    color:white;
    border-radius: 10px;
}
button:disabled{
    background-color: lightgray;
    color:white;
}

/* Capa que pone borde verde a los controles
cuando su contenido es válido */
.verde{
    border-color: green;
}
```

```
/* Capa que muestra los errores */
#error{
    color:red;
    font-family: sans-serif;
    margin-top:3em;
}

/* Capa inicialmente oculta
Cuando se muestra desvanece el formulario*/
#telon{
    position: fixed;
    left:0;
    top:0;
    width:100%;
    height:100%;
    z-index:100;
    background-color: black;
    opacity: .5;
    display:none;
}

/* Capa por encima de la anterior
inicialmente oculta con el mensaje
de envío de datos correctos*/
#mensaje{
    position: fixed;
    left:50%;
    margin-left:-150px;
    margin-top:-100px;
    top:50%;
    width:300px;
    height:200px;
    z-index:200;
    background-color:white;
    display:none;
}

#mensaje p{
    font-weight: bold;
    font-size:2em;
    text-align: center;
    font-family: sans-serif;
}
```

Podemos observar que, además del CSS de maquetación de los elementos del formulario, disponemos de una clase llamada **verde** para dar borde verde a los controles cuando son válidos. Las capas de mensaje y telón se posicionan por encima del formulario dejando la apariencia inicial tapada por el fondo negro de la capa de telón.

El código JavaScript está dividido en funciones. La petición AJAX la gestiona la función `validarNIF`. El resto de validaciones tienen también funciones dedicadas. Muchas de ellas podrían ser validadas directamente por HTML, ya que la propiedad `pattern` admite una expresión regular y la propiedad `type` con valor "email" ya valida el email. Pero JavaScript admite una interactividad extra porque el usuario puede ver en caliente, a medida que escribe, si lo que hace es válido o no.

El código del archivo `accion.js` es este:

```
/***
 * Valida que al menos haya dos caracteres
 * alfábéticos en el elemento
 * (usa caracteres de lenguas españolas)
 * @param {Element} elemento
 */
function validarNombre(elemento){
    //La expresión Unicode: /^[p{Letter}]{2}/u
    //es mejor para validar, pero aun no es
    //compatible con todos los navegadores
    if(/^[[A-Zá-zñÑçÇáéíóúúÁÉÍÓÚÜÜÀÈÌÒÙàèìòù]{2}]/
        .test(elemento.value)){
        elemento.classList.add("verde");
    }
    else{
        elemento.classList.remove("verde");
    }
    comprobarTodoValido();
}

/***
 * Valida que el Elemento contiene como
 * valor un Email válido
 * Solo mira que haya alguna letra, una símbolo @
 * y otra letra
 * @param {Element} elemento
 */
function validarEmail(elemento){
    if(/.*[A-Zá-z].*@[A-Zá-z].*/.test(elemento.value)){
        elemento.classList.add("verde");
    }
    else{
        elemento.classList.remove("verde");
    }
    comprobarTodoValido();
}
```

```
/**  
 * Acude al servicio de internet de validación  
 * de NIFs para validar la corrección del mismo  
 * @param {Element} elemento  
 */  
function validarNIF(elemento){  
    //Preparación del parámetro  
    let formData=new FormData();  
    formData.append("nif",elemento.value);  
  
    fetch(`https://jorgesanchez.net/servicios/validarNIF.php`,{  
        method:"POST",  
        body:formData  
    })  
    .then(resp=>{  
        return resp.json();  
    })  
    .then(datos=>{  
        if(datos.error.codigo==0){  
            elemento.classList.add("verde");  
        }  
        else{  
            elemento.classList.remove("verde");  
        }  
        comprobarTodoValido();  
    })  
    .catch(error=>{  
        document.getElementById("error").textContent=error;  
        comprobarTodoValido();  
    })  
}  
  
/**  
 * Comprueba si todo el formulario es válido  
 * para activar el botón  
 */  
function comprobarTodoValido(){  
    if(  
        nombre.classList.contains("verde") &&  
        apellidos.classList.contains("verde") &&  
        email.classList.contains("verde") &&  
        NIF.classList.contains("verde")  
    ){
```

```
//todo es correcto
boton.disabled=false;
}
else{
    boton.disabled=true;
}
}

window.addEventListener("load", (ev)=>{
    var nombre=document.getElementById("nombre");
    var apellidos=document.getElementById("apellidos");
    var email=document.getElementById("email");
    var NIF=document.getElementById("NIF");
    var boton=document.getElementById("boton");
    var telon=document.getElementById("talon");
    var mensaje=document.getElementById("mensaje");

    nombre.addEventListener("keyup", (ev)=>{
        validarNombre(nombre);
    });
    apellidos.addEventListener("keyup", (ev)=>{
        validarNombre(apellidos);
    });
    email.addEventListener("keyup", (ev)=>{
        validarEmail(email);
    });
    NIF.addEventListener("keyup", (ev)=>{
        validarNIF(NIF);
    });

    boton.addEventListener("click", (ev)=>{
        ev.preventDefault();
        //Damos por hecho que si el botón está
        //activo, los datos son válidos
        telon.style.display="block";
        mensaje.style.display="block";
    });
    //Con el clic en el telon o en el mensaje,
    //empezamos de nuevo
    telon.addEventListener("click", (ev)=>{
        location="index.html";
    });
    mensaje.addEventListener(("click", (ev)=>{
        location="index.html";
    });
});
```

Práctica 10.2: Mostrar un usuario aleatorio

- La página <https://randomuser.me/> permite obtener datos aleatorios de personas pensando en que los desarrolladores y otros profesionales puedan utilizarlos en sus pruebas y test.
- Las instrucciones de la API de este servicio gratuito están en la URL: <https://randomuser.me/documentation>
- En todo caso la idea es hacer peticiones vía GET a la URL: <https://randomuser.me/api/>
- Se pueden pasar parámetros para indicar cuántos usuarios aleatorios deseamos, el sexo, política de contraseñas, páginas, formato de respuesta, etc.
- En la página de documentación viene un ejemplo de la estructura JSON de las respuestas. Como resumen indicamos que es un objeto formado por dos propiedades: **results** e **info**. La primera es un array donde cada elemento lo forma un objeto con los datos del usuario aleatorio. La propiedad **info** contiene otros detalles entre los que destaca una semilla que permite repetir una petición con los mismos datos y datos de paginación (cuando deseamos dividir en páginas los resultados se debe usar la misma semilla).
- La aplicación mostrará la foto, nombre, apellido, email, dirección y estado al que pertenece el usuario. Cada vez que actualicemos la página, se pedirá otro usuario. Ejemplo de resultado:



lærke sorensen
 Email: lærke.sorensen@example.com
 201 møllevænget
 nykøbing sj. (MIDTJYLLAND)

Figura 10.4: Ejemplo de usuaria aleatoria

SOLUCIÓN: PRÁCTICA 10.2

La API **Randomuser** lo hace casi todo. Lo único que debemos programar es cómo recoger los datos que deseamos y cómo mostrarlos. El código HTML podría ser este:

```

<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <title>Usuario aleatorio</title>
  <link rel="stylesheet" href="css/estilos.css">

```

```
</head>
<body>
<main></main>
<script src="js/accion.js"></script>
</body>
</html>
```

Simplemente, cargamos el JavaScript y el CSS y dejamos un elemento **main** para que sea allí donde dejamos los datos.

Hoja de estilos (css/estilos.css):

```
*{
    font-family: sans-serif;
}
main{
    text-align: center;
}
p{
    margin:0;
}
.error{
    background-color:red;
    color:white;
    font-weight: bold;
}
```

Finalmente, el código JavaScript que únicamente invoca al método **fetch** para realizar la petición, convierte la respuesta a JSON y luego localiza y coloca los datos deseados:

```
var main=document.querySelector("main");

const url = 'https://randomuser.me/api/';
fetch(url)
.then((resp) => resp.json())
.then(data=>{
    let datos=data.results[0];
    let foto=datos.picture.large;
    let mail=datos.email;
    let nombre=datos.name.first;
    let apellido=datos.name.last;
    let calle=datos.location.street;
    let ciudad=datos.location.city;
    let estado=datos.location.state;

    main.innerHTML=
        `<figure>`+
        `<img src='${foto}' alt='foto'>`+
```

```

`</figure>` +
`<p>${nombre} ${apellido}</p>` +
`<p>Email: ${mail}</p>` +
`<p>${calle}</p>` +
`${ciudad} (${estado.toUpperCase()})</p>`;
})
.catch(error=>{
  main.innerHTML=`<p class='error'>${error}</p>`;
});

```

Práctica 10.3: Mostrar 10 usuarios que cambian

- Usaremos el mismo servicio de usuarios aleatorios para esta práctica.
- Si pasamos un parámetro llamado **results** con valor **10**, conseguiremos obtener datos de 10 usuarios aleatorios.
- Mostraremos esos datos en dos filas. Distinguiremos el fondo de los pares e impares y, además, en cada usuario colocaremos un botón con el texto "**Cambiar**".
- Cuando hagamos clic en el botón **Cambiar**, el usuario en el que está situado el botón, se cambiará.
- Mostraremos el mensaje **Cargando...** al principio, mientras llegan los datos de los 10 usuarios (cuando lleguen, se quita el mensaje). También cuando pulsemos **Cambiar** en el usuario a modificar, escribiremos **Esperado usuario nuevo...** en la celda de ese usuario (habremos, antes, quitado los datos del usuario) hasta que lleguen los nuevos datos.

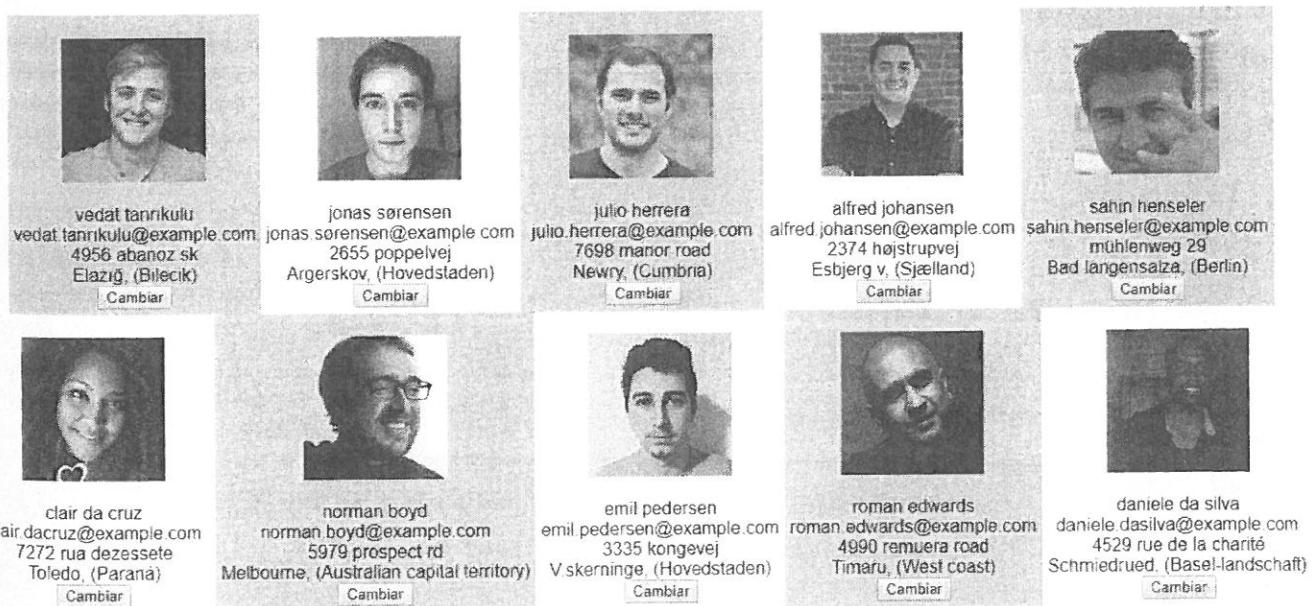


Figura 10.5: Ejemplo de página con los 10 usuarios aleatorios

SOLUCIÓN: PRÁCTICA 10.3

Nuevamente, el código HTML que necesitamos es muy simple:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <title>10 Usuarios aleatorios</title>
    <link rel="stylesheet" href="css/estilos.css">
</head>
<body>
<main></main>
<script src="js/accion.js"></script>
</body>
</html>
```

El CSS prepara la maquetación de los paneles en los que irán los usuarios. Al arrimar el ratón, sombreadmos en azul el panel de usuario sobre el que está el ratón,

```
*{
    font-family: sans-serif;
}
main{
    text-align: center;
}
main div{
    display:inline-block;
    padding:.3em;
}
main div:nth-of-type(odd){
    background-color: lightgray;
}
main div:hover{
    background-color:rgb(88, 230, 230);
}
p{
    margin:0;
}
.error{
    background-color:red;
    color:white;
    font-weight: bold;
}
```

El código JavaScript utiliza varias funciones con la finalidad de conseguir un código más modular y mantenible. Así hay:

- **primeraMayus.** Función que simplemente recibe un texto y le devuelve con la primera letra mayúscula y el resto minúsculas.
- **rellenarUsuario.** Encargada de colocar los datos en la capa correspondiente con el código HTML apropiado.
- **generarEventoBotón.** Cada capa de usuario necesita que el botón pueda modificar el contenido para leer otro usuario aleatorio. Habrá una petición http para ello, en la que solo requeriremos un usuario. La petición se asocia al evento **click** del nuevo botón. Aunque este código es mejorable (no tiene sentido crear de nuevo el botón de cambio de usuario), le hemos dejado así para recordar la recursividad: tras programar el botón, hacer la petición y grabar los datos de esa petición (con ayuda de la función **rellenarUsuario**), volvemos a invocar a esta función para que genere el evento de nuevo.

La zona principal del código se basa en hacer la petición de los 10 primeros usuarios y colocarlos en las capas correspondientes. Código:

```
/**
 * Retorn un texto con la primera letra en mayúscula
 * @param {string} texto
 */
function primeraMayus(texto){
  return texto[0].toUpperCase() +
    texto.slice(1).toLowerCase();
}

/**
 * Función que formnatea los datos del usuario aleatorio
 * en un elemento div
 * @param {Element} capaUsuario Elemento div en el
 * que se graba el usuario
 * @param {JSON} usuario El usuario que
 * contiene los datos obtenidos de randomuser.me
 */
function rellenaUsuario(capaUsuario,usuario){
  let foto=usuario.picture.large;
  let mail=usuario.email;
  let nombre=usuario.name.first;
  let apellido=usuario.name.last;
  let calle=usuario.location.street;
  let ciudad=usuario.location.city;
  let estado=usuario.location.state;

  capaUsuario.innerHTML=
    `<figure> +
      <img src='${foto}' alt='foto'> +
    </figure> +
    <div> +
      <strong>${nombre} ${apellido}</strong> -
      <p>${calle}, ${ciudad}, ${estado}</p> -
      <p>${mail}</p> -
    </div> +
  
```

```
`<p>${nombre} ${apellido}</p>` +
`<p>${mail}</p>` +
`<p>${calle}</p>` +
`${primeraMayus(ciudad)}, (${primeraMayus(estado)})</p>` +
`<p><button>Cambiar</button></p>`;
}

/** 
 * Captura el clic de botón en una capa de usuario
 * @param {Element} capaUsuario Elemento div que contiene
 * los datos a cambiar del usuario
 */
function generarEventoBoton(capaUsuario){
    //Asignamos evento al botón
    let boton=capaUsuario.querySelector("button");
    boton.addEventListener("click",(ev)=>{
        //Mensaje de espera que desaparece cuando se cargan los datos
        capaUsuario.textContent="Esperando usuario nuevo...";
        //Hacemos petición de 1 usuario aleatorio nuevo
        fetch('https://randomuser.me/api')
            .then(resp=>resp.json())
            .then(data=>{
                let usuario=data.results[0];
                //Modificamos los datos de la capa actual
                rellenaUsuario(capaUsuario,usuario);
                //programamos de nuevo el click del botón que cambia el usuario
                generarEventoBoton(capaUsuario);
            })
            .catch(error=>{
                capaUsuario.innerHTML=`<p class='error'>${error}</p>` +
                    `<p><button>Intentar de nuevo</button></p>`;
            });
    });
}

/** 
 * Zona principal del código
 */
var main=document.querySelector("main");
main.textContent="Esperando usuarios...";
//Petición de los 10 usuarios
fetch('https://randomuser.me/api/?results=10')
    .then((resp) => resp.json())
    .then(data=>{
        let datos=data.results;
```

```

main.textContent="";
for(let usuario of datos){
    //Generamos el HTML de cada capa con datos de usuario
    let divUsuario=document.createElement("div");
    rellenaUsuario(divUsuario,usuario);
    //Programamos el click del botón que cambia el usuario
    generarEventoBoton(divUsuario);
    //Añadimos la nueva capa a main
    main.appendChild(divUsuario);
}
})
.catch(error=>{
    main.innerHTML=`<p class='error'>${error}</p>`;
});

```

Práctica 10.4: Mapa meteorológico de la Aemet

- La agencia meteorológica española AEMET dispone de una API para datos abiertos en la que proporciona numerosas posibilidades de obtener información meteorológica.
- Para poder obtener los datos de esta agencia, necesitamos darnos de alta como desarrolladores en la dirección:
<https://opendata.aemet.es/centrodedescargas/altaUsuario>
- Las instrucciones de funcionamiento de la API de AEMET están en:
<https://opendata.aemet.es/dist/index.html?>
- En el alta necesitamos indicar un email que tendremos que confirmar para que se nos conceda una API Key, una clave única que se otorga a cada desarrollador. Esa API la deberemos enviar cada vez que hagamos peticiones a la AEMET.
- Nuestra aplicación, inicialmente, muestra un cuadro de texto en el que introducir nuestra API y un botón de cargar mapa

Imagen del día de la AEMET

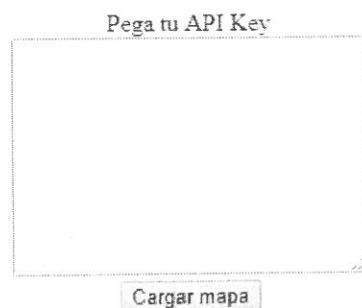
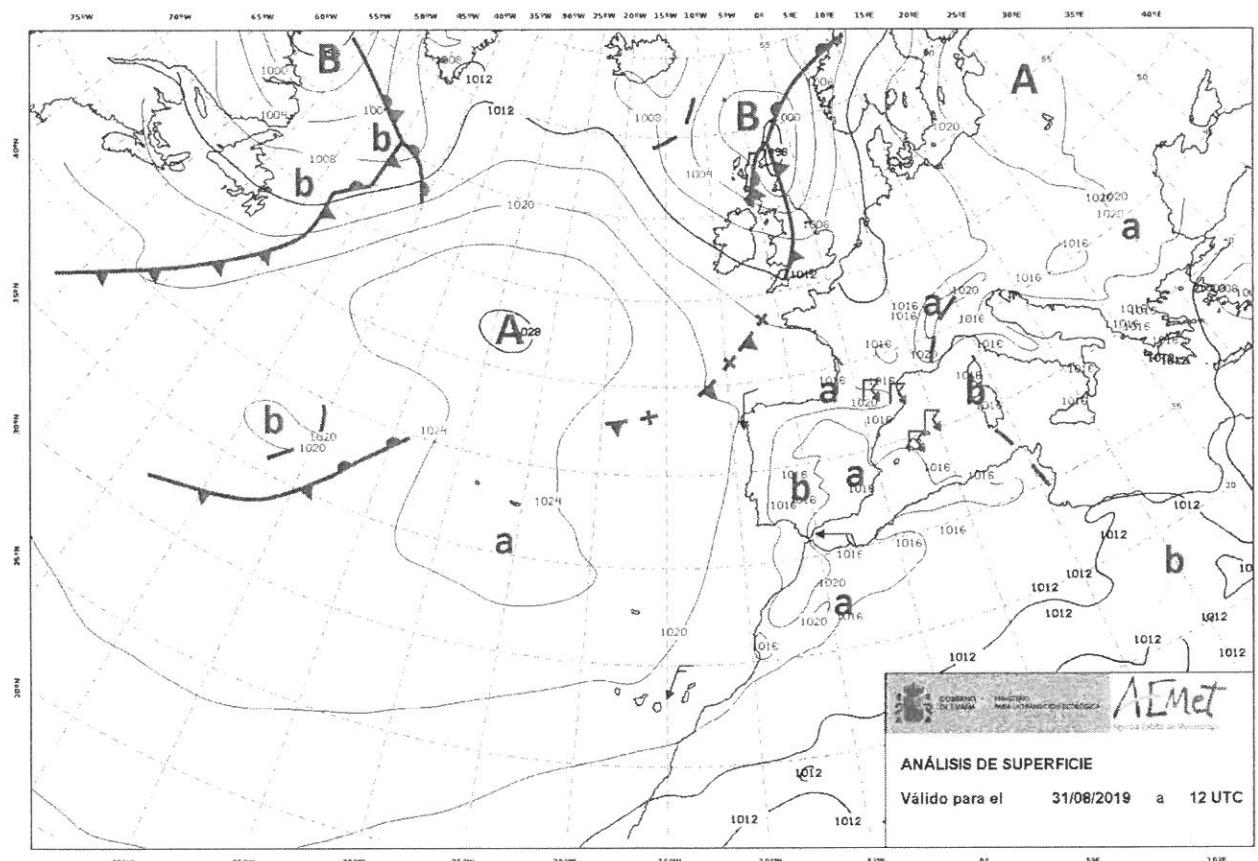


Figura 10.6: Aspecto inicial de la aplicación del mapa meteorológico.

- Tras pegar nuestra API y pulsar el botón **Cargar mapa**, solicitaremos el mapa meteorológico del día en la dirección:

<https://opendata.aemet.es/opendata/api/mapasygraficos/analisis>

- Se quitarán los controles de formulario y se mostrará (en horizontal) el mapa.
- Hay que recordar que debemos enviar vía GET un parámetro llamado `apikey`, con nuestra clave (la cual se obtiene del formulario).
- Los datos que llegan, nos ofrecen un objeto JSON donde la propiedad llamada `datos`, es una URL a la imagen del mapa. La petición http nos proporciona ese enlace, pero no el mapa en sí.
- Por ello, hay una segunda petición (a esa URL devuelta por la propiedad `datos`) que es la que nos proporciona el mapa



©AEMET. Autorizado el uso de la información y su reproducción citando a AEMET como autora de la misma

Figura 10.7: Mapa meteorológico obtenido de la API de datos abiertos de la AEMET.

SOLUCIÓN: PRÁCTICA 10.4

Código HTML (`index.html`), simplemente se encarga de mostrar la portada dentro de un contenedor `main`, además de cargar el código CSS y el JavaScript:

```
<!DOCTYPE html>
<html lang="es">
<head>
```

```

<meta charset="UTF-8">
<title>Mapa meteorológico del día</title>
<link rel="stylesheet" href="estilos.css">
</head>

<body>
<main>
    <h1>Imagen del día de la AEMET</h1>
    <form action="#">
        <label for="apikey">Pega tu API Key</label><br>
        <textarea name="apikey" id="apikey" cols="30" rows="10">
        </textarea><br>
        <button>Cargar mapa</button>
    </form>
</main>
<script src="accion.js"></script>
</body>
</html>

```

El CSS es muy simple, solo se encarga de rotar el mapa y redimensionarlo:

```

main{
    text-align: center;
}
main img{
    transform:rotate(90deg);
    width:70%;
}

```

En el código JavaScript, la función **procesarFetch** se encarga de todo el trabajo. En este caso hemos decidido colocar el método **fetch** en una función **async** a fin de mostrar cómo utilizarla con la instrucción **await**. Es un buen ejemplo para usar la API Fetch de este modo porque hay dos peticiones http encadenadas: una para el enlace al mapa (en formato JSON) y otra para obtener el mapa en sí (formato binario).

Los parámetros de la petición realizados en el objeto de la petición, se han hecho siguiendo las especificaciones de AEMET. Hay que tener en cuenta que la documentación no se basa en el API de Fetch sino en JavaScript usando el objeto **XMLHttpRequest** y también se explica cómo hacer la petición usando la librería **jQuery**. No es difícil adaptar ese código a Fetch (de hecho, es más fácil esta API).

La primera petición obtiene os datos en JSON, se convierten usando el método **json()**. La segunda petición usa el método **blob**. Cada proceso y petición se realiza esperando la finalización de las promesas anteriores logrando un código muy sencillo de entender.

```

/**
 * Función que realiza la petición fetch y coloca el mapa en el documento
 * en el elemento indicado

```

```
* @param {string} apikey
* @param {Element} elemento Contenedor de la imagen y el formulario
* */
async function procesarFetch(apikey,elemento){
    var form=elemento.querySelector("form");
    var headers = new Headers({
        "cache-control": "no-cache"
    });
    var conf={
        method:"GET",
        mode:"cors",
        headers:headers,
    }
    try{
        const resp1=await fetch("https://opendata.aemet.es/opendata/api/
mapasygraficos/analisis?api_key="+
                        apikey,conf) ;
        const data1=await resp1.json();
        const resp2=await fetch(data1.datos,conf);
        const mapa=await resp2.blob();
        const img=document.createElement("img");
        img.setAttribute("src",URL.createObjectURL(mapa));
        //Añadimos la imagen y quitamos el formulario
        elemento.removeChild(form);
        elemento.appendChild(img);
    }
    catch(error){
        let p=document.createElement("p");
        p.innerHTML=<strong>Error al cargar el mapa:</strong> "+error;
        elemento.appendChild(p);
    }
}

// Código principal dentro del evento load
// para asegurar la carga previa de los componentes
window.addEventListener("load",(ev)=>{
    let main=document.querySelector("main");
    let apiKey=document.querySelector("textarea");
    let boton=document.querySelector("button");

    boton.addEventListener("click",(ev)=>{
        ev.preventDefault();
        procesarFetch(apiKey.value,main);
    });
});
```

10.10 PRÁCTICAS RECOMENDADAS

Práctica 10.5: Cambio de moneda

- La URL <https://exchangeratesapi.io/> proporciona una API para obtener cambios de moneda está documentado en la propia página. En la propia página aparece un ejemplo de devolución de datos:

```
{
  "base": "EUR",
  "date": "2018-04-08",
  "rates": {
    "CAD": 1.565,
    "CHF": 1.1798,
    "GBP": 0.87295,
    "SEK": 10.2983,
    "EUR": 1.092,
    "USD": 1.2234,
    ...
  }
}
```

- La propiedad **base** hace referencia a la moneda base del cambio, en el ejemplo anterior se tomará 1 euro como base. Así, por ejemplo podremos obtener el cambio de 1 EUR=1.565 GBP (Libras)

Conversión de monedas

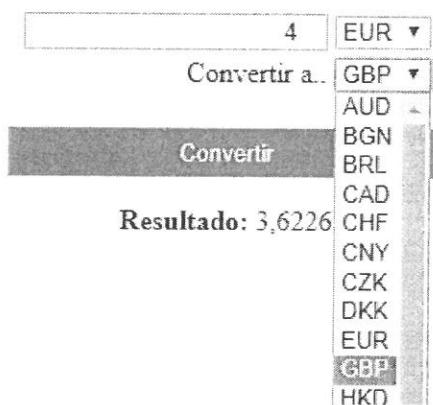


Figura 10.8: Ejemplo de conversión de monedas

- Para obtener la última fecha de cambio, se usa: <https://api.exchangeratesapi.io/latest>
- Podemos enviar el parámetro **symbols** para que nos devuelva solo los cambios a monedas concretas. Por ejemplo: <https://api.exchangeratesapi.io/latest?symbols=USD,GBP> nos consigue los cambios de Euro a Dólares de EEUU y a Libras Esterlinas.

- También podemos cambiar la base de la moneda con el parámetro **base** al que podemos indicar una moneda diferente del Euro.
- Gracias a esta API, creamos el formulario de la Figura 10.8. Este formulario deberá permitir realizar cualquier cambio de moneda. Para llenar los cuadros de lista, necesitaremos hacer una primera petición y mostrar todas las monedas.

Práctica 10.6: Imagen de la NASA

- La NASA tiene un servicio que permite mostrar lo que llaman la “Imagen del día”. Ese servicio se puede invocar desde JavaScript usando la URL: <https://api.nasa.gov/planetary/apod>
- A esa dirección (mediante GET) le podemos pasar un parámetro con la fecha llamado **date** en este formato “**YYYY-MM-DD**” por ejemplo “**2019-05-01**”. Además se pasa otro parámetro llamado **api_key** que es una clave que nos pasa la NASA para poder usar su API.
- Si no se pasa el parámetro **date**, se nos da la imagen del día actual.
- La api key (la clave) la podemos obtener si rellenamos nuestros datos en la dirección: <https://api.nasa.gov/index.html#apply-for-an-api-key>

Get Your API Key

Sign up for an application programming interface (API) key to access and use web services available on the Data.gov developer network.

* Required fields

The form consists of several input fields:

- First Name**: An input field with a red asterisk indicating it is required.
- Last Name**: An input field with a red asterisk indicating it is required.
- Email**: An input field with a red asterisk indicating it is required.
- How will you use the APIs?** (optional): A text area for describing the intended use of the APIs.
- Signup**: A large, dark button at the bottom of the form.

Figura 10.9: Formulario de alta en la página de la NASA para obtener nuestro API Key

- Al llenar nuestro nombre, apellido y email nos darán la clave (un código numérico muy grande).
- Sabiendo esos datos prueba una petición a ver si llega el archivo JSON y observa el formato. A partir de ahí, crea una página que haga lo siguiente:

- Que inicialmente muestre la imagen del día y un control por debajo permita cambiar la fecha.
- Al cambiar la fecha se muestra la imagen de ese día.
- El color de fondo será negro.

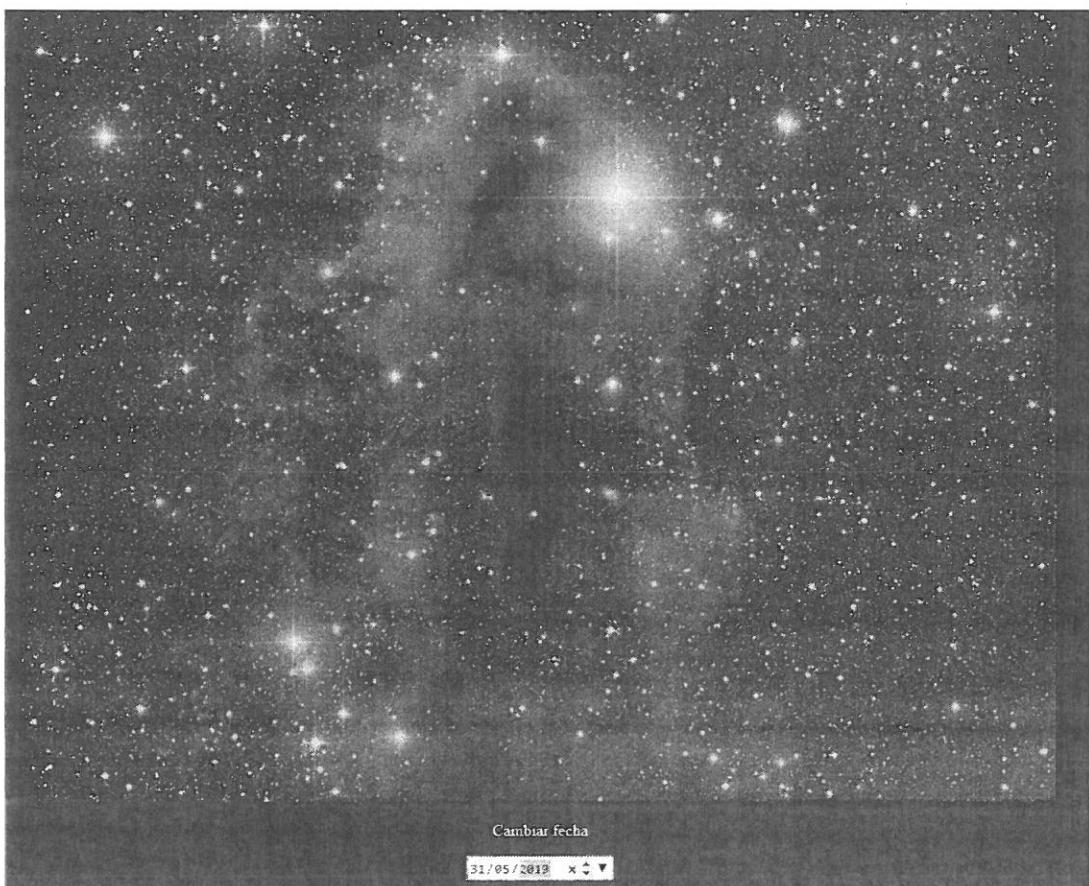


Figura 10.10: Imagen del día de la NASA para la fecha 31/05/2019

Práctica 10.7: Imágenes desde la nube

- El servicio picsum permite utilizar imágenes libremente. Posee una API al respecto. En este ejercicio podemos usar esta sintaxis:

<https://picsum.photos/300/300?image=1>

- Este enlace obtiene la imagen número 1 a una resolución de 300 x 300 píxeles.
 - Si añadimos el parámetro blur (no hace falta indicar valor en él), tendremos:
- <https://picsum.photos/300/300?image=1?blur>
- La imagen sale desenfocada.
- Sabiendo esto, genera una página que muestre 200 imágenes a 300 por 300 de resolución.
 - Solo deben caber imágenes 4 en horizontal. Si la pantalla es pequeña, entonces, las imágenes se muestran más verticales. No debe haber scroll horizontal. Las imágenes deben quedar centradas en la ventana.

- Si se hace clic en una imagen, esta se desenfoca
- Además aparece desenfocada durante una semana.
- Si se hace clic en otra, entonces esa será la desenfocada la semana siguiente.
- Si se hace clic en la que ya está desenfocada, entonces quedará sin desenfocar.
- Usa las peticiones AJAX para enfocar y desenfocar.



Figura 10.11: Aplicación con la lista de fotos de **picsum**, habiendo marcado la segunda para desenfocar.

Práctica 10.8: Animación vistosa

- Basándote en la API de **randomuser.me** (véase Práctica 10.2 y Práctica 10.3), consigue crear una página web que muestre 50 caras aleatorias consiguiendo que cambien 100 veces, de forma aleatoria, cada dos décimas de segundo.
- Es una animación muy llamativa cuyo efecto se basa en que no haya un exceso de retraso al conseguir las fotos. Lo ideal es cargar las fotos por adelantado.
- Lo lógico es hacer una petición a esa API que devuelva muchas personas (por ejemplo 1000) y almacenarlas en un array. Luego el array se puede desordenar de forma aleatoria cada dos o tres décimas .
- El control de las décimas lo llevan los intervalos.



Figura 10.12: Una muestra de la página de la animación vistosa. Las fotos deben ir cambiando continuamente 100 veces. Luego se detiene la animación

10.11 RESUMEN DE LA UNIDAD

- AJAX es una tecnología que permite la comunicación cliente/servidor de forma dinámica mediante JavaScript a través de peticiones y respuestas http. Permite la carga de datos asíncrona hacia un determinado componente de la aplicación.
- Los datos recibidos por las peticiones AJAX pueden ser de todo tipo: texto, binario, XML, etc. Actualmente la preferencia es JSON.
- Los servicios de Internet pueden aplicar políticas CORS para protegerse de peticiones procedentes de otros dominios. Solo podemos recoger datos de peticiones a otros dominios si su política de CORS lo acepta.
- AJAX facilita la creación de aplicaciones enriquecidas y de nuevos modelos de interfaz de usuario muy dinámicos y compatibles con todo tipo de tecnologías back-end. Hay algunas desventajas relativas a la dificultad de los navegadores para indexar páginas con contenidos obtenidos mediante peticiones AJAX o el no poder crear marcadores con el estado actual de una página.
- El objeto **XMLHttpRequest** es el que tradicionalmente se usa para gestionar las peticiones y respuestas AJAX. Actualmente, es más recomendable el uso de la API Fetch, basada en el método global **fetch**.
- El método **fetch** es el encargado de realizar las peticiones utilizando la URL destino y un objeto (llamado **request** u objeto de petición) que contiene los parámetros concretos de http para la petición.
- El resultado del método **fetch** es una promesa que, si se resuelve correctamente, retorna un objeto resultado de la petición (**response**) que contiene las cabeceras http y los datos de la respuesta.
- El objeto de respuesta posee métodos para procesar los datos que contiene, en función de su tipo (texcto, binario, JSON, etc.), lo cual genera una nueva promesa cuyo resultado positivo son los datos correctamente convertidos al formato deseado.
- Las cabeceras http se gestionan con un tipo de objeto llamado **Headers** que permite crear y modificarlas.
- El envío de datos permite utilizar todos los verbos http (GET, POST, PUT, etc.) y enviar los datos en los formatos habituales: codificados por URL o tipo **form data**. Para los datos de tipo form data existe un objeto llamado **FormData** que permite gestionarlos y crearlos de forma cómoda.

10.12 TEST DE REPASO

1. **¿Qué es CORS?**
 - a) Un grupo de música irlandesa.
 - b) Un comando http como GET o POST.
 - c) Un mecanismo que permite dar o no permiso a peticiones http de otros dominios.
 - d) Un tipo de cifrado http que facilita la autenticación del cliente hacia el servidor y viceversa.

2. **¿Qué objeto clásico en JavaScript está relacionado con el manejo de AJAX?**
 - a) XMLHttpRequest.
 - b) XMLHttpRequest.
 - c) XMLHttpRequest.
 - d) XMLHttpRequest.

3. **¿Qué objeto clásico en JavaScript está relacionado con el manejo de AJAX?**
 - a) XMLHttpRequest.
 - b) XMLHttpRequest.
 - c) XMLHttpRequest.
 - d) XMLHttpRequest.

4. **¿Qué parámetros utiliza el método fetch para realizar peticiones?**
 - a) Una URL y el comando http de envío de petición
 - b) Una URL y un objeto de tipo Headers
 - c) Una URL y un objeto de tipo Response
 - d) Una URL y un objeto de tipo Request

5. **¿Cómo podemos saber que el resultado de una petición AJAX tiene código "ok"?**
 - a) Mediante la propiedad status del objeto de respuesta.
 - b) Mediante la propiedad statusText del objeto de respuesta.
 - c) Mediante la propiedad ok del objeto de respuesta.
 - d) Las dos primeras respuestas son correctas.
 - e) Las tres primeras respuestas son correctas.

6. **¿Qué resulta de invocar al método fetch?**
 - a) Un objeto Promise
 - b) Un objeto Response
 - c) Un objeto Request
 - d) Un objeto Headers

7. **¿Cómo se recoge el objeto de respuesta (Response) de las peticiones fetch?**
 - a) Mediante la función callback del método then
 - b) Mediante el resultado del operador await aplicado a fetch
 - c) Las dos respuestas anteriores son correctas

8. **¿Cómo se recoge el objeto de respuesta (Response) de las peticiones fetch?**
 - a) Mediante la función callback del método then.
 - b) Mediante el resultado del operador await aplicado a fetch.
 - c) Las dos respuestas anteriores son correctas.
 - d) Ninguna es correcta, el objeto Response se usa como segundo parámetro del método fetch.

9. **¿Qué propiedad hay que modificar para indicar que nuestra petición usando el comando http PUT?**
 - a) Propiedad method del objeto Request
 - b) Propiedad type del objeto Request
 - c) Propiedad method del objeto Response
 - d) Propiedad type del objeto Response

10. **¿Qué tipo de objeto resulta de usar el método json() de los objetos de respuesta?**
 - a) JSON
 - b) String
 - c) Object
 - d) Promise

- 11.- ¿Cómo podemos saber si los datos recibidos son de tipo JSON?**
- Analizando la cabecera Content-Type de la propiedad headers del objeto de respuesta.
 - Analizando la cabecera MIME-Type de la propiedad headers del objeto de respuesta.
 - Analizando la cabecera Content-Type de la propiedad headers del objeto de petición.
 - Analizando la cabecera MIME-Type de la propiedad headers del objeto de petición.
 - Analizando la propiedad type del objeto de respuesta.
- 12.- ¿Qué método del objeto de respuesta se usa para procesar datos binarios?**
- binary()
 - bin()
 - data()
 - blob()
- 13.- ¿Qué método del objeto de petición se usa para indicar que nuestra petición es de tipo CORS ?**
- type
 - method
 - mode
 - cors
- 14.- ¿Qué método permite añadir una nueva cabecera a un objeto de tipo Headers ?**
- append
 - add
 - header
 - new
- 15.- ¿Qué método permite añadir una nueva cabecera a un objeto de tipo FormData ?**
- append
 - add
 - header
- 16.- ¿Qué método permite obtener el valor de una cabecera concreta de un objeto Headers?**
- has
 - get
 - is
 - obtain
- 17.- ¿Qué método permite obtener el valor de una cabecera concreta de un objeto FormData?**
- has
 - get
 - is
 - obtain
- 18.- ¿En qué propiedad del objeto de petición se colocan los datos enviados por una petición GET?**
- method
 - body
 - headers
 - En ninguna, se añaden a la URL
- 19.- ¿En qué propiedad del objeto de petición se colocan los datos enviados por una petición POST?**
- method
 - body
 - headers
 - En ninguna, se añaden a la URL
- 20.- Queremos enviar con nuestra petición AJAX, de tipo POST, los datos procedentes de un formulario asociado a un objeto llamado *form*. ¿Cómo podemos crear un objeto FormData, llamado *fd*, con esos datos?**
- let fd=FormData(form);
 - let fd=new FormData(form);
 - let fd=form.FormData();
 - let fd=form.getFormData();