UD10. AJAX

1. COMUNICACIÓN CLIENTE/SERVIDOR

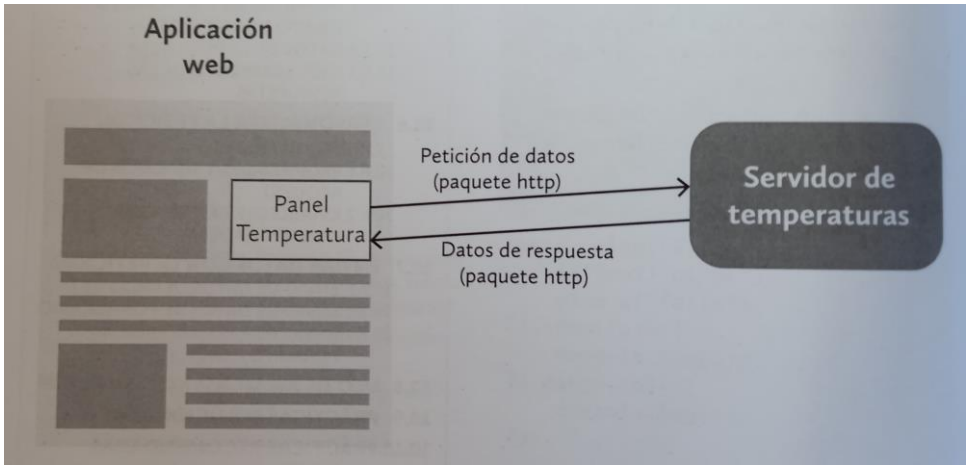
En la primera unidad expusimos el funcionamiento general de las aplicaciones web. Hemos diferencia lo que se ejecuta en el lado del cliente (front-end) y lo que se ejecuta en el lado del servidor (back-end)

La explicación entonces 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 al 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 en la 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, 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.



2. INTRODUCCIÓN A AJAX

2.1. ¿QUÉ ES AJAX?

Asynchronus JavaScript and XML. 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 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 IE 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 esta 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 cargando 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, pero, en realidad, se pueden enviar los datos en muchos otros formatos.

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 estáticas 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 rellenando) y de relleno 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.

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 paginarlo para no provocar una gran espera en la llegada de 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 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 de 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 utilizar protección anti **CORS**, de la que hablaremos más adelante.

3. PETICIONES AJAX

3.1. USO DE UNA API PARA ACCEDER A UN SERVICIO WEB

API (**Application Programming Interface**), 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 usado 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 autentificarnos antes de resolver la petición de datos. Esa autenticación suele implicar el envío de claves o tokens al servidor.

3.2. CORS

CORS (Cross-Origin Resource Sharing). 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 estas 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 *, 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.

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 este 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 2 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 le 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 IE).

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 del resto de esta unidad.

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://alumno.net/servicios/nifaleatorio.php")
.then(resolver=>{
        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 lo 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("https://alumno.net/servicios/nifaleatorio.php")
.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
>
```

5. MANIPULAR LA RESPUESTA. OBJETO RESPONSE

Como hemos visto, 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 ahora las capacidades de este objeto para poder utilizar de la mejor forma posible los datos de respuesta de la petición realizada.

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 respuesta http. Contiene los datos en sí de la respuesta.

status	Devuelve el código de respuesta de la petición http. Un valor 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 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.
type	Indica el tipo de respuesta de la petición. Posibles valores: **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.

[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

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.

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**) son una convención aceptada por todos los servidores que indica el tipo de datos.

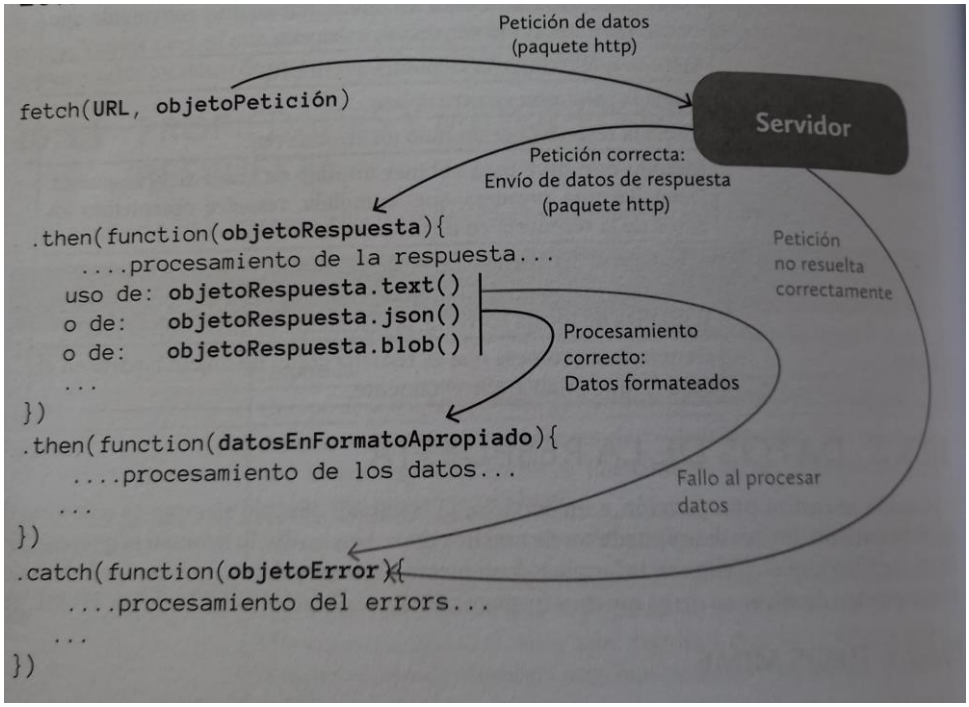
Su formato habitual se basa en indicar un tipo, seguido de un subtipo separados por una barra de dividir. Ejemplos de tipos MIME habituales usados en peticiones AJAX son:

- **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.

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

5.3. PROCESAMIENTO DE LAS RESPUESTAS



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

1) Invocar al a función fetch indicando la URL a la que realizamos la petición y un objeto **request**, si es el caso, en el que matizaremos aspectos avanzados de la petición con el método http a utilizar (**GET**, **POST**, **PUT**, **DELETE**, etc).
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 procesaremos 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.

5.3.1. MANIPULAR RESPUESTAS EN FORMATO TEXTO

El tipo de datos más sencillo es el texto. Para entender cómo funcionan, e n la dirección https://jorgesanchez.net/servicios/poesia.php se permite obtener el texto de un poema aleatorio. Cada vez que se realiza una petición 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=erro});
</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 rellenar 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.

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.

Si tuviésemos una dirección que permitiese obtener en formato JSON un NIF aleatorio. Supongamos que la respuesta fuese del tipo:

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

Devuelve un array de objetos (aunque solo devuelve un elemento del array) en el cada elemento es un objeto con 2 propiedades: tipo y número. Para crear una aplicación que requiere un NIF aleatorio y lo 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://...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.

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 respueta a una petición son los 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 de 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 HTML.

Veamos cómo poner en práctica 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");
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 mian
})
.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.

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 posibles a nuestros intereses.

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, 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: • **same-origin**. Solo se acepta la 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.
cache	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: • **default**. Valor habitual. El navegador comprueba si la respuesta a la petición está en la 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 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 el código 504 de petición.
redirect	Indica cómo se deben procesar las respuestas en el caso de que procedan de redirecciones. Posibilidades: • **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) En peticiones AJAX, solo las 2 primeras interesan
credentials	Permite especificar si se admiten cookies en la petición. Posibilidades: • **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	Almacena una cadena de integridad creada con un algoritmo hash de criptografía (se admite sha256, sha384, sha512), para validar la integridad de la petición.

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

```
fetch("https://direccionanovalida.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.

6.2. ESTABLECER LA CABERA. 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":"uft-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: objeto.**append**('Content-Type','application/json'); Si el nombre de cabecera http no es correcta, se provoca una excepción de tipo **TypeError**.
delete	Retira una cabecera del objeto de cabecera. Ejemplo: objeto.**delete**('Content-Type');
get	Devuelve el valor de una cabecera en el objeto. Por ejemplo: objeto.**get**('Content-Type'); Podría retornar, por ejemplo '***application/json***' Si la cabecera de la que se pide el valor no está en el objeto, devuelve **null**.
set	Modifica una cabecera existente y le otorga un nuevo valor: objeto.**set**('Content-Type', 'text/xml'); 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**.
has	Método que comprueba si el objeto de cabeceras contiene una entrada concreta. De ser así, devuelve true. objeto.**has**('Content-Type');

entries	Devuelve un objeto iterable que permite recorrer todas las cabeceras del objeto:
	
	for(let [clave,valor] of ObjetoHeader]{
	console.log(clave +", "+ valor);
	}

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

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 de 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 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. Esa 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 recibirlos, dependiendo de lo que el servicio es capaz de aceptar.

7.1. ENVÍO DE DATOS USANDO GET

En el caso de 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 10 **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 10 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://alumno.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.createElemnet("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.

7.2. ENVÍO DE DATOS DE FORMULARIO

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://direccionanovalida.php",{
method: "post",
        headers: {'Content-Type':'application/x-www-form-urlencoded'},
        body: " nif:123456789A&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 **&**. 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 este formato.

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

```
body: "nif:123456789A&tipo=dni"
```

7.2.2. USO DE OBJETOS DE 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 parámetro/clave se hace con el método **append**:

```
form.append("nif","12345678A");
form.append("tipo","dni");
```
Si quisiéramos borrar un parámetro:

```
form.delete("tipo");
```
Si quisiéramos modificar:

```
form.set("nif","9876532B");
```
Saber si hemos grabado un valor lo hace el método has, que devuelve true o false.

```
console.log(form.has("nif"); //Escribe true
```
Obtener un 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","Gema");
form.append("edad","50");
form.append("profesión","profe");
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://alumno.net/servicios/nifaleatorio.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 2 propiedades: ***codigo*** que devuelve el código de error (vale 0 si no hay erro) 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");
form.append("edad","50");
fetch("https://alumno.net/servicios/nifaleatorio.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://alumno.net/servicios/nifaleatorio.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>
let form=document.querySelector("form");
```

```
let capaRes=document.getElementById("resultado");
let 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 códgio, 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 los muestra en una capa (**capaRes**).

Para que el manejo del usuario de esta página sea más cómodo y permite 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.

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 disponile 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

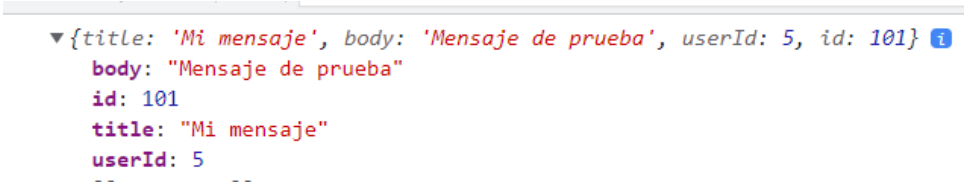
Si añadimos el número de post: https://jsonplaceholder.typicode.com/posts/5 nos devuelve el post número 5 (hay 100 para probar).

Mediante peticiones tipo POST podemos añadir nuevos datos. En realidad no se añaden (Es un servicio para practicar, no es real), pero parece como que sí lo hace. 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***".

```
<script>
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)});
</script>
```

El resultado de la petición es:



Realmente no hay un post con número 101, se nos muestran estos datos simulando la que sería 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.

8. USO DE AWAIT/ASYN 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 y facilita 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 2 notaciones utilizar para nuestras peticiones.