Planteamiento

Ahora que ya hemos estirado y calentado mediante katas nuestro conocimiento y habilidad con JavaScript vamos a estudiar y trabajar una de sus características esenciales que tiene que ver con la esencia de lo que conocemos como desarrollo web: la **asincronía**.

Introducción

Si le preguntamos a cualquier usuario que cree que sucede cuando solicita una página web nos estará explicando qué es la asincronía de la forma más plana y sencilla.

Cuando un usuario solicita una página web puede ser:

- Escribiendo la url en la barra del navegador
- Accediendo a través de un click en un enlace

En ambos casos lo que sucede es una serie de acciones en las diferentes capas de la estructura OSI que acabarán, en el mejor de los casos, con la renderización en el navegador de la página solicitada.

Sin embargo desde que se hace la solicitud hasta que la página se presenta en el navegador transcurre un tiempo, es decir, no es inmediato y eso es precisamente lo que es la asincronía en el mundo web: **la diferencia temporal desde que se hace una petición de datos hasta que estos están disponibles**.

HTTP

La asincronía debida al protocolo http es la más evidente y de la que todos somos conscientes. La otra que veremos y con la que trabajaremos es intrínseca al desarrollo web y su gestión no ha sido una práctica extendida hasta hace pocos años. Veremos el por qué más adelante.

Repasando conceptos básicos y sin entrar en mucha profundidad, cuando se solicita una página web el retardo acumulado hasta que podemos visualizar el contenido se debe a:

- Contacto con el servicio de DNS
- Resolución del nombre de dominio en relación a la IP
- Devolución de la IP
- Contacto con el servidor cuya dirección IP hemos obtenido
- Devolución del fichero solicitado (si existe) o cabecera de error
- Interpretación y renderizado del fichero obtenido o mensaje de error

Este proceso no debería tardar más de un segundo en ningún caso con una conexión de internet de alta velocidad (como las que tenemos desde hace tiempo) y con unos equipos con capacidad de procesamiento alta (como los que tenemos desde hace tiempo). En

países o contextos con conexiones lentas y dispositivos móviles antiguos el proceso de petición, carga y renderizado puede alargarse más tiempo.

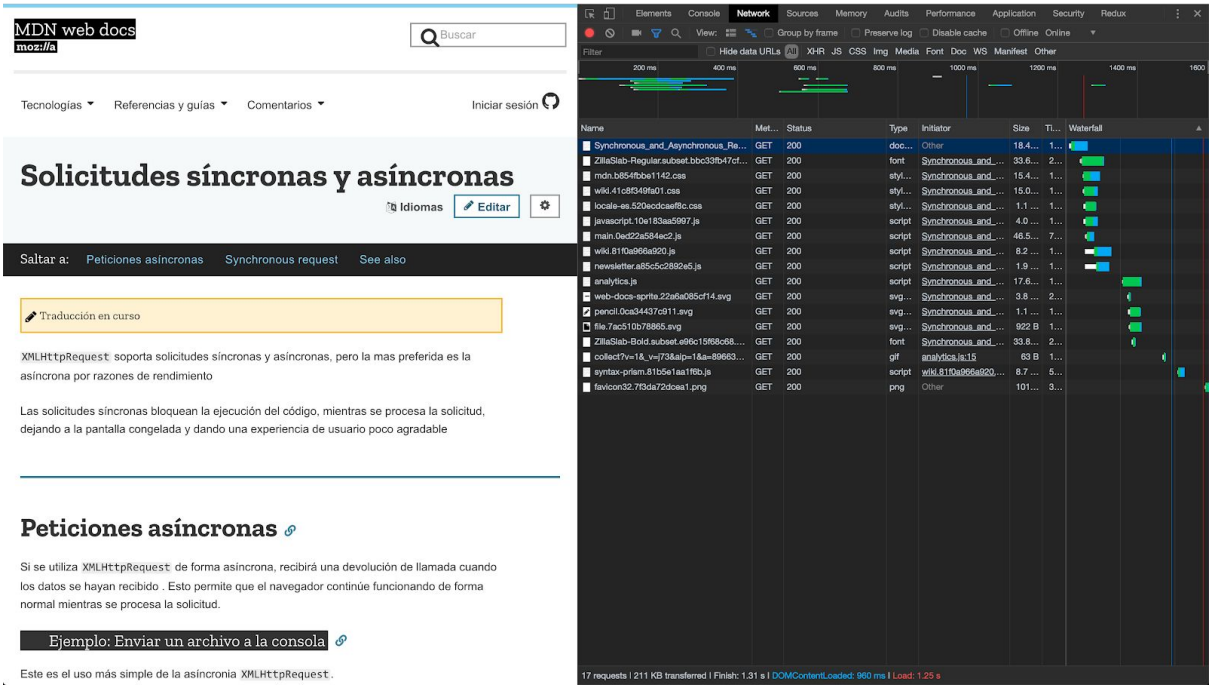
Los ficheros que pueden solicitarse a un servidor pueden ser imágenes, ficheros de vídeo y audio, elementos de stream y en general cualquiera aceptado por las políticas del servidor, pero el caso que vamos a utilizar es el esencial y básico: los ficheros html.

- http: Hypertext Transfer Protocol
- html: HyperText Markup Language

Estos dos elementos van muy de la mano: protocolo y especificación de lenguaje y son los que juntos cimentan **Internet** como la conocemos.

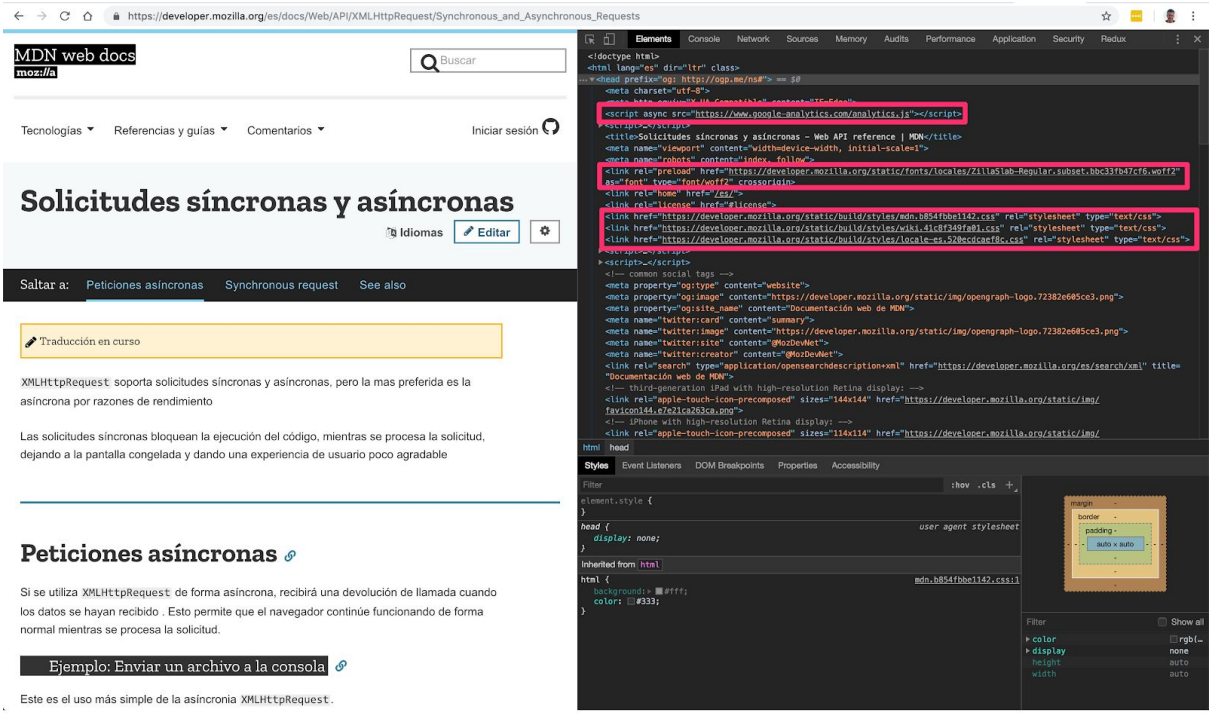
Cuando un navegador recibe la respuesta a su solicitud (que suele ser el archivo index.html en la raíz del servidor de sitios como Google, la UOC o cualquier otro) lo que sucede es que se interpreta su contenido empezando por la cabecera del **<head>** que es donde se encuentran los metadatos.

Recordad que lo que recibe el navegador es un fichero html. En el momento de recepción no tenemos ni estilos, ni fuentes, ni imágenes ni JavaScript. Vamos a ver cómo todo eso se solicita y en qué orden.



En esta imágen de la pestaña network podéis ver cada una de las solicitudes que tiene lugar en forma de cascada. Pensad que esto es lo que sucede la primera vez que solicitáis la página. Para las posteriores la caché interna del navegador puede evitar algunas solicitudes innecesarias si los assets de la página no han cambiado:

Revisemos este ejemplo de fichero html:



Ahora tenemos el html y con él cierta meta información de la página tanto para el control de la presentación de la página, para el indexado en buscadores como para la ubicación de todos los assets relacionados con la correcta visualización de la página solicitada.

Según el orden secuencial del código, los rectángulos rojos resaltan:

- Archivo js para el control de la analítica web
- Tipo de fuente externo (una no incluída en las fuentes de sistema)
- Estilos css

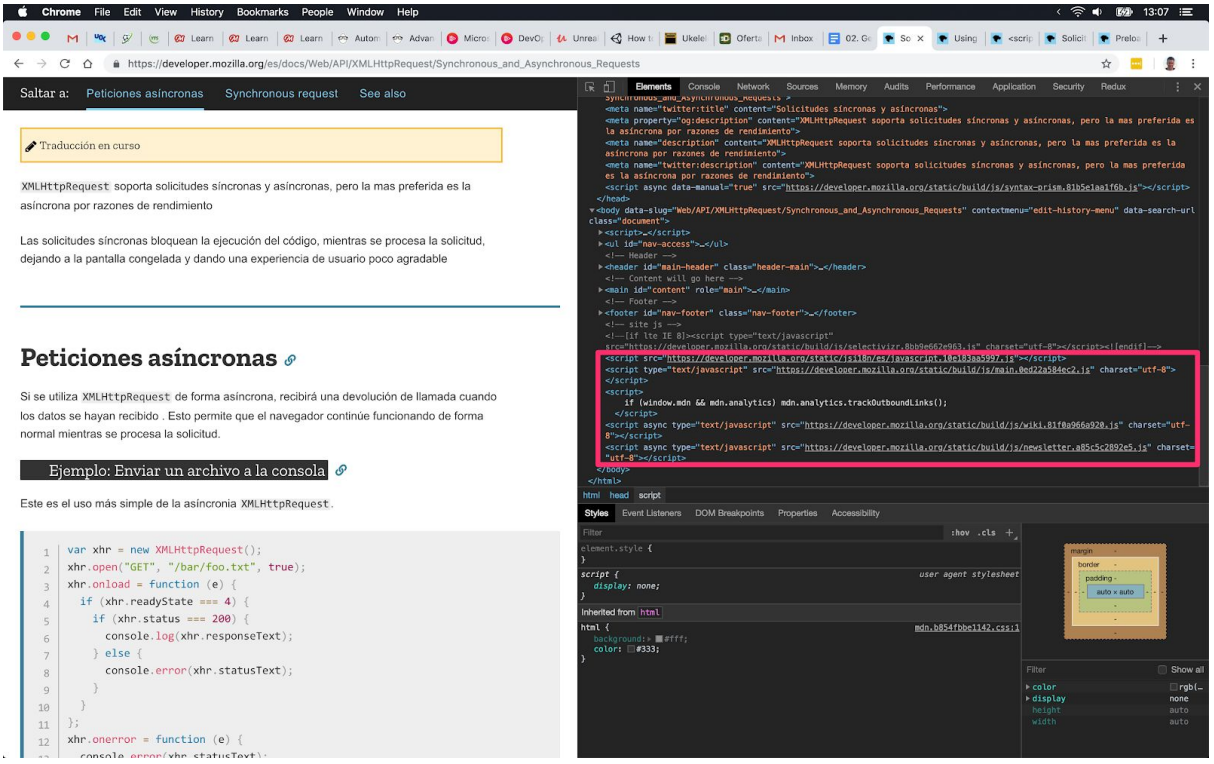
Lo que hará el navegador es entender y preparar la estructura de etiquetas de html para presentarla en el navegador y posteriormente realizar las solicitudes de assets de la web. Para simplificar la comprensión vamos a imaginar que todos los assets se piden a la vez de forma simultánea (aunque esto es únicamente cierto en servidor con http2).

La primera solicitud es la del fichero js de analytics y contiene el tag async en la etiqueta. Al solicitar un fichero js por defecto se bloqueará el hilo principal del navegador y el renderizado de la web quedará bloqueado hasta que el fichero haya llegado y se haya ejecutado.

¿Imagináis qué pasaría si por algún motivo ese fichero no existiera o se perdiera por el camino o hubiera cambiado de nombre? El navegador aparecerá en blanco y no respondería, esperando que llegue el fichero y poder ejecutarlo. Ésta no es para nada una buena experiencia de usuario ni planificación del camino crítico de la página.

Para evitar esto se pone el tag **async** que permite que el hilo principal de renderizado del navegador no quede bloqueado y el usuario pueda tener una respuesta visual inmediata e ininterrumpida.

Históricamente no se hacía así puesto que la propiedad **async** es propia de **HTML5**. Lo que se hacía anteriormente era incluir la solicitud del fichero al final de la etiqueta <body> por lo que el navegador renderizaba y luego lanzaba la request por lo que el contenido ya estaba renderizado en el navegador y el usuario podía consumir e interactuar con la información. Podemos ver esto en la página de MDN para la carga de ficheros JavaScript:



En el caso de la petición de una fuente externa podría suceder que el contenido se renderizara previamente a recibir e interpretar el archivo del tipo de fuente con el fallback definido en los estilos.

```
html {
  font-family: Lato, "Lucida Grande", Tahoma, Sans-Serif;
}
```

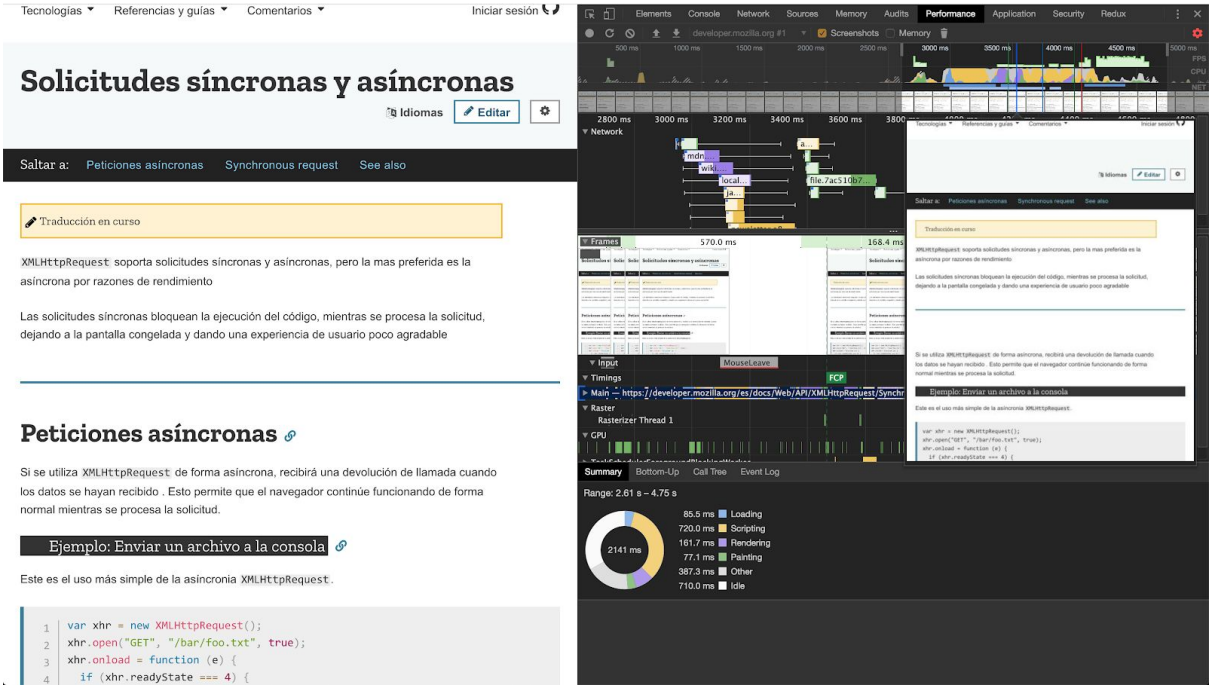
Al recibir la fuente se aplicará el estilo y habrá un salto en la representación de la página ajustando los textos, espaciado y estilos con los datos de la nueva fuente recién descargada.

De forma parecida a **async** para <script>s de **JavaScript** tenemos preload en la segunda request para la fuente. Con esto le decimos al navegador que no es un asset esencial pero

que lo necesitaremos para presentar correctamente el contenido. El navegador interpreta y hace su magia con los metadatos especificados.

En el último rectángulo vemos la petición de los ficheros css con los estilos de los diferentes elementos de la página web. Este es el caso más sencillo de todos. En caso de existir y recibirlos correctamente se aplicarán, en caso contrario veremos un error en consola conforme no se han podido descargar los archivos y, potencialmente, veremos un renderizado que no se corresponde con el diseño que esperaba transmitir quien implementó el sitio.

Si queréis indagar un poco más en lo que sucede mientras se realizan todas estas peticiones y cómo las gestiona el navegador podéis revisar el pestaña **Performance** de las dev tools y poner a grabar mientras recargais una página web:



Del monolito web al frontend y backend y SPAs

Hasta aquí hemos revisado lo que sucede a modo estructural con el protocolo y el fichero html recibido pero no a modo de aplicación.

Si nos remontamos cronológicamente a los inicios del desarrollo web veremos que por aquel entonces teníamos páginas web estáticas formadas por varios ficheros de html, css y, en el mejor de los casos, scripts muy sencillos de **JavaScript** para control de formularios o alguna funcionalidad muy primitiva.

En esa época las páginas se implementaban una por una en base a su contenido. Mantener la estructura del sitio y los estilos de forma coherente es un trabajo poco tecnificado y potencialmente tedioso y repetitivo, muy proclive a fallos, errores e inconsistencias.

Conforme la tecnología se asentaba, el tamaño de los sitios aumentaba y la necesidad de servir contenidos de forma dinámica aparecía, un nuevo paradigma surgió en base a generar contenidos de forma dinámica en el servidor.

Lenguajes de servidor como **PHP** o **Ruby** permitían definir en el lado del servidor plantillas de html que en base a variables de sesión, usuario o de petición presentaba unos datos u otros permitiendo el dinamismo de las páginas:

```
<title><?php print $PAGE_TITLE;?></title>

<?php if ($CURRENT_PAGE == "Index") { ?>
      <meta name="description" content="" />
      <meta name="keywords" content="" />
<?php } ?>

<link rel="stylesheet" type="text/css"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.min.
css">
<style>
      #main-content {
            margin-top:20px;
      }
      .footer {
            font-size: 14px;
            text-align: center;
      }
</style>
```

Este código nos recuerda a html pero con variables. El servidor insertaba en los fragmentos de código *<?php ?>* las variables relativas o datos procesados de cada petición y devolvía el html con los datos solicitados. El html recibido era estático pero su generación era dinámica.

Este paradigma es el que se conoce como **monolito** porque no existe una diferencia real entre la parte de cliente y la de servidor. Estructura html, estilos css y la gestión y procesado de datos desde la base de datos hasta su representación conviven en un único ecosistema.

Durante este tiempo la especialización entre los **especialistas en presentación y manipulación de datos** fue creciendo dando lugar a perfiles **frontend** y **backend**. Las tecnologías avanzaron en complejidad y, cada vez, resultaba más difícil seguir siendo **webmaster**, el actual **fullstack**, para acabar teniendo que tender profesionalmente a uno u otro perfil. La vida es demasiado corta para ser un buen especialista en más de una cosa.

Con el incremento de la potencia de los equipos, la parte más ligera de la manipulación de datos se desplazó del servidor al cliente aunque eso implicaba tener que enviar ciertos

datos sensibles con los consecuentes problemas de seguridad y privacidad. Estas peticiones a demandas desde el cliente al servidor son la parte asíncrona que vamos a tratar en esta parte de la asignatura.

Esta migración de carga computacional al cliente propició la aparición del paradigma de aplicaciones SPA (Single Page Applications). Pasamos de tener páginas estáticas generadas dinámicamente en el servidor a tener una aplicación en el cliente que realizaba peticiones a demanda contra el servidor. Esta es la situación actual en la que la mayoría de sitios ya no pueden considerarse páginas web como tal sino como aplicaciones, en muchos casos habilitadas por frameworks como **Angular** o librerías como **React** o **Vue**.

Interfaz y APIs

Nosotros no vamos a llegar a desarrollar una SPA. Nos vamos a quedar en un paso anterior, con un sitio que solicite datos de forma dinámica al servidor. Primero vamos a repasar las diferentes interfaces y APIs de **JavaScript** y de navegador que nos hemos encontrado a lo largo del tiempo para realizar peticiones asíncronas:

- XMLHttpRequest
- jQuery y Ajax
- Fetch

Nota: vamos a enfocar tanto la teoría como la práctica exclusivamente en el consumo de datos de una API. Aunque es posible realizar otras acciones (CRUD) vamos a limitar la dificultad de las explicaciones y los ejercicios a la petición de datos.

XMLHttpRequest

Inicialmente sólo disponíamos de la interfaz XMLHttpRequest para poder realizar consultas asíncronas desde cliente. Estaba poco optimizada (no en su funcionamiento sino en su sintaxis) puesto que era poco usada ya que no había una necesidad generalizada de este tipo de consultas al servidor y, por ello, la construcción de una petición era realmente tediosa:

```
// 1. Create a new XMLHttpRequest object
let xhr = new XMLHttpRequest();

// 2. Configure it: GET-request for the URL /article/.../load
xhr.open('GET', '/article/xmlhttprequest/example/load');

// 3. Send the request over the network
xhr.send();

// 4. This will be called after the response is received
xhr.onload = function() {
```

```
  if (xhr.status != 200) { // analyze HTTP status of the response
    alert(`Error ${xhr.status}: ${xhr.statusText}`); // e.g. 404: Not
Found
  } else { // show the result
    alert(`Done, got ${xhr.response.length} bytes`); // responseText is
the server
  }
};

xhr.onprogress = function(event) {
  if (event.lengthComputable) {
    alert(`Received ${event.loaded} of ${event.total} bytes`);
  } else {
    alert(`Received ${event.loaded} bytes`); // no Content-Length
  }

};

xhr.onerror = function() {
  alert("Request failed");
};
```

Fuente: https://javascript.info/xmlhttprequest

Cierto es que todo esto podría implementarse en una función y reutilizarse a conveniencia parametrizando las partes de la petición que quisiéramos hacer variables pero no dejaba de ser un método poco optimizado y muy proclive a errores.

jQuery y Ajax

De nuevo sin entrar en muchos detalles pero JQuery es una librería que ofrecía una interfaz común para las diferentes implementaciones de la especificación ECMAScript en diferentes navegadores.

Como quizás recordéis hubo una época en la que había una guerra abierta entre navegadores, básicamente **Internet Explorer** contra **Netscape Navigator**. Su implementación del motor de JavaScript no era igual y uno, como desarrollador y usuario, podía tener una experiencia parcial o sesgada en caso de visitar un sitio con uno u otro navegador.

Un grupo de brillantes y pioneros desarrolladores trabajaron en una librería llamada **jQuery** que al ser incluida como dependencia de un proyecto, habilitaba una capa de traducción que unificaba los accesos a las diferentes APIs de los navegadores eliminando la problemática de desarrollar para uno u otro.

Uno de los métodos que ofrece jQuery es ajax que permite simplificar la realización de peticiones con un código similar a este:

```
$.ajax({
  method: "POST",
  url: "some.php",
  data: { name: "John", location: "Boston" }
})
  .done(function( msg ) {
    alert( "Data Saved: " + msg );
  });
```

Fuente: http://api.jquery.com/jquery.ajax/

Estaremos de acuerdo en que la propuesta de **jQuery** no solo ayudaba a unificar diferentes interfaces sino que simplifica mucho a nivel de desarrollo la realización de peticiones.

Si visitamos el repositorio de jQuery en Github podremos ver como su implementación hace uso del interfaz XMLHttpRequest (extracto de código):

```
jQuery.ajaxSettings.xhr = function() {
      try {
            return new window.XMLHttpRequest();
      } catch ( e ) {}
};
```

Fetch

La evolución y maduración de **JavaScript** como lenguaje así como las continuas actualizaciones de la especificación y las progresivas implementaciones de los navegadores hizo más superficial la necesidad de **jQuery** como capa de traducción.

Sin embargo muchos desarrolladores habían desarrollado dependencia a su uso por la facilidad y simplicidad de algunos de sus métodos y la resistencia a olvidar viejos hábitos.

Consejo: evitad el uso de jQuery. Su propósito inicial ya no tiene razón de ser y los casos de uso justificados son escasos y muy puntuales (especialmente en aplicaciones antiguas).

La API fetch (más info) es el equivalente del método **ajax** de **jQuery** pero dentro de la especificación de los navegadores. **jQuery** (a mi modo de ver) ya no tiene un lugar en el desarrollo moderno pero durante mucho tiempo marcó el rumbo que debería seguir **JavaScript** para alcanzar la madurez y estandarización que tiene a día de hoy.

```
// url (required), options (optional)
```

```
fetch('https://davidwalsh.name/some/url', {
      method: 'get'
}).then(function(response) {

}).catch(function(err) {
      // Error :(
});
```

Fuente: https://davidwalsh.name/fetch

A valorar la conveniencia de esta API por encima si cabe de la propuesta por jQuery al no requerir de la importación y uso de una librería adicional.

Nota: los tres métodos son válidos para usar en fase de desarrollo pero os recomiendo encarecidamente que uséis fetch puesto que es el más moderno y extendido. En caso de necesitar dar retrocompatibilidad a navegadores antiguos que no tengan incorporada esta API podéis añadir en vuestro proyecto el polyfill de fetch sobre el objeto window.

Control de flujo en el código

Estas tres maneras de realizar peticiones incorporan la gestión de la asincronía de formas diferentes:

- **xhr.onload()** para **XMLHttpRequest**
- **.done()** para **jQuery.ajax**
- **.then()** y .**catch()** para **fetch**

En los dos primeros casos hablamos de **callbacks** en el segundo de la resolución de **promises**.

Callbacks

Un callback es simplemente una función a ejecutar una vez se cumpla una situación o circunstancia.

En el caso de las peticiones asíncronas el callback se ejecuta cuando los datos llegan devueltos desde el servidor.

Nota: en el siguiente módulo también revisaremos este concepto para los eventListeners sobre los elementos del DOM.

Cuando hablemos de callbacks pensamos en una función normal pero que será invocada en un momento concreto que no podemos controlar directamente, sólo podemos indicar que, cuando esa situación se produzca, se ejecute dicha función.

Callback ejemplo de XMLHttpRequest:

```
xhr.onprogress = function(event) {
  if (event.lengthComputable) {
    alert(`Received ${event.loaded} of ${event.total} bytes`);
  } else {
    alert(`Received ${event.loaded} bytes`); // no Content-Length
  }
};
```

Callback ejemplo de jQuery.ajax:

```
.done(function( msg ) {
    alert( "Data Saved: " + msg );
  });
```

Promises

Las promesas son una manera más sofisticada de implementar la gestión de la asincronía. La lógica detrás de las promesas a nivel de especificación es altamente compleja.

Nota: os recomiendo indagar en su conceptualización hasta donde os dé vuestro interés y capacidad en los enlaces que os dejo al final del documento pero, por suerte o por desgracia, no necesitamos entender su implementación para entender cómo usarlos.

Según la definición de MDN de Promise:

Una promesa representa un valor que puede estar disponible ahora, en el futuro, o nunca.

Esta definición es tan abstracta, genérica y sencilla que cuesta hacerse a la idea de la complejidad que lleva implícita su desarrollo a nivel de especificación.

Aprovechando los textos encontrados en MDN, podemos entender una promesa como uno de los tipos básicos de JavaScript (como Number, Array u Object) pero que almacena el estado de una petición, pudiendo ser:

- **pendiente** (pending): estado inicial, no cumplida o rechazada.
- **cumplida** (fulfilled): significa que la operación se completó satisfactoriamente.
- **rechazada** (rejected): significa que la operación falló.

Una promesa pendiente puede ser cumplida con un valor, o rechazada con una razón (error). Cuando una de estas dos posibilidades sucede, el método **then** o **catch** es ejecutado.

Las promesas fueron diseñadas de este modo ya que el lanzar una petición no se debe bloquear el hilo de ejecución hasta que sus datos (o error) sean recibidos. Una promesa permite mantener una estructura de código y ejecución síncrona y secuencial aunque haya procesos asíncronos en marcha. Estos conceptos son los que hacen complicada la implementación a nivel de lenguaje pero fácil su uso ya que toda la complejidad cognitiva de lo que sucede a nivel asíncrono (y que nos resulta tan complicado de conceptualizar a los seres humanos) queda escondida detrás de la API **fetch**.

Revisando el ejemplo anterior:

```
// url (required), options (optional)
fetch('https://davidwalsh.name/some/url', {
      method: 'get'
}).then(function(response) {

}).catch(function(err) {
      // Error :(
});
```

La gestión de la asincronía se realiza de un modo conceptualmente muy sencillo:

- al ser ejecutada la primera línea, fetch crea una promesa en estado **pendiente**
- cuando se resuelva esta podrá pasar a un estado **completado** o **rechazado**
- si se **rechaza** se ejecutará el método catch con la función definida como callback que recibirá como parámetro el error (err) encontrado
- si se **completa** se ejecutará el método then con la función definida como callback que recibirá como parámetro la respuesta de la petición

Async / await

Async y await conforman un método alternativo de representar el trabajo con promesas. Hace exactamente lo mismo esto:

```
function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
        throw new Error(response.status);
      }
    })
}
```

Que esto:

```
async function loadJson(url) { // (1)
  let response = await fetch(url); // (2)

  if (response.status == 200) {
    let json = await response.json(); // (3)
    return json;
  }

  throw new Error(response.status);
}
```

Ejemplo original de: https://javascript.info/async-await

Otro ejemplo para hacer más comprensible lo que sucede:

```
async function foo() {
  console.log("hi");
  return 1;
}

async function bar() {
  const result = await foo();
  console.log(result);
}

bar();
console.log("lo");
```

Código original: https://stackoverflow.com/a/42833744

El proceso será:

- Creación de la promesa al ejecutar *const result = await foo();*
- Ejecución del código secuencia del foo() *console.log("hi")*
- Espera para la resolución de la promesa (aunque no hay petición asíncrona)
- Salto al hilo principal para invocar *console.log("lo")*
- Resolución de la promesa, result obtiene el resultado y se invoca *console.log(result)*

Otro ejemplo más:

```
function getUser(name){
 fetch(`https://api.github.com/users/${name}`)
  .then(function(response) {
    return response.json();
  })
```

```
  .then(function(json) {
    console.log(json);
  });
};

//get user data
getUser('yourUsernameHere');
```

Se convierte en esto:

```
async function getUserAsync(name)
{
  let response = await fetch(`https://api.github.com/users/${name}`);
  let data = await response.json()
  return data;
}

getUserAsync('yourUsernameHere')
  .then(data => console.log(data));
```

Código original: https://dev.to/shoupn/javascript-fetch-api-and-using-asyncawait-47mp

Es posible que encontréis más fácil un método que otro así que no dudéis en usarlo a conveniencia.

Notas sobre asincronía en JavaScript

Todos los métodos que hemos visto han gestionado la asincronía con callbacks y promesas. Esta es la manera adecuada porque por la naturaleza de JavaScript solo existe un hilo de ejecución y todo lo que parezca simultáneo será en realidad concurrente (más info).

Puesto que solo existe un hilo de ejecución y éste condiciona la presentación de datos al usuario, bloquear el hilo para realizar una petición bloquearía la interacción del usuario con nuestro sitio.

Como vimos en el primer apartado, cargar un fichero de **JavaScript** en el <head> del fichero html puede bloquear totalmente la carga de la página. Lo mismo puede suceder si una vez cargada la página realizamos una petición que bloquea el hilo principal: la página estaría renderizada pero toda interacción quedaría pendiente a que se resuelva nuestra petición.

Recursos

A continuación os dejo la lista de recursos a revisar para la **PAC2**:

- [YDKJS] Async & Performance
    - Introducción
    - Conceptos básicos
    - Callbacks
    - Promesas
- [Eloquent JS] HTTP and Forms (solo lo relativo al protocolo)
- [Eloquent JS] Asynchronous Programming
- [javascript.info] The JavaScript language: Promises, async/await

Videos:

- Revisiting Async - Kyle Simpson
- Advanced Async and Concurrency Patterns in JavaScript

Nota: Kyle Simpson tiene una opinión muy fuerte sobre los temas que trata y suele estar enfrentado contra algunas decisiones tomadas sobre la especificación de JavaScript. Intentad abstraeros de todo lo que sea opinión y gentratos unicamente es lo que sean hechos y teoría objetiva.