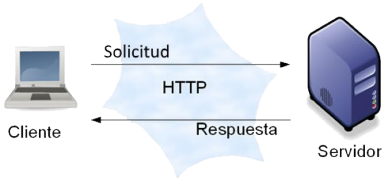
INTRODUCCIÓN COMUNICACIÓN ASINCRONA

INTRODUCCIÓN

El HTTP son las siglas de “Hypertext Transfer Protocol“, el protocolo de comunicación que permite las transferencias de información en la WEB (World Wide Web).

El HTTP funciona como un protocolo de solicitud-respuesta (request-response) entre un cliente y un servidor.

Un cliente puede ser un navegador WEB (Microsoft Edge, Safari, Mozzilla, Google Chrome, …) ejecutado desde un ordenador y un servidor puede ser un chip ESP8266 que aloja una o varias páginas WEB. Cuando el cliente envía una solicitud HTTP al servidor, el servidor envía la respuesta al cliente.



MODELOS DE COMUNICACIÓN

Hasta ahora, la interacción entre el navegador y el servidor seguía un modelo de comunicación petición-respuesta poco satisfactorio para el usuario, donde la actualización de la información presentada requería continuas recargas de la página, apareciendo la sensación de ser un contenido o información demasiado estático.

Este modelo ha sido mejorado gracias a la aparición de AJAX, que definen un conjunto de técnicas que mediante procesos en segundo plano permiten mejorar el comportamiento del modelo anterior, eliminando la necesidad de recarga de la página, apareciendo incluso la posibilidad de realizar cambios o actualizaciones parciales en el documento.

¿CÓMO FUNCIONA EL CLIENTE-SERVIDOR?

Podemos referirnos a esta relación de "camaradería" entre las dos partes del desarrollo web: la comunicación entre un cliente y un servidor. Para que nos entendamos, existe un agente cliente que realiza

peticiones de contenido (las webs en nuestro caso) y un agente servidor que a parte de almacenar los archivos de la web los puede "leer" o "interpretar".

La función del cliente es visualizar un contenido para su lectura interpretando los archivos e información que le ha pedido a un servidor y que este le ha enviado.

La función por parte del servidor es atender o escuchar (concepto técnico) las peticiones/solicitudes de clientes, que en un momento dado pueden ser uno o muchísimos de forma simultánea.

¿QUIÉN ES EL CLIENTE ?

El cliente es el navegador web que tienes instalado y con el que visualizas una página. Un navegador tiene la capacidad de interpretar varios lenguajes de programación especialmente creados para la visualización y maquetación de contenido.

Gracias a los lenguajes que puede interpretar, el navegador es capaz de por ejemplo:

- Construir botones a partir de una etiqueta de HTML.
- Cambiar el diseño de una parte de una parte de la web con tan solo pasar el ratón por encima.
- Mostrar un color distinto en enlaces a páginas que ya has visitado almacenados en su memoria (cache).
- Adaptar textos y elementos gráficos según el tamaño de la ventana de tu navegador con CSS.

¿QUIÉN ES EL SERVIDOR?

El servidor es un software especializado para tal función instalado en un ordenador que se pasa el día encendido. El servidor se pasa las horas literalmente a la espera de recibir un contacto/llamada/petición a través de internet.

El servidor, entre sus funciones principales tiene:

- Alojar ficheros de todo tipo, desde imagenes a ficheros de programación o base de datos.
- Interpretar ficheros con programación específica de servidor y construir un fichero en base a esta
- Enviar un mensaje de respuesta de tamaño variable, ya sea de error (código 404 por ejemplo) o de ok (código 200), con el contenido indicado por la programación que un administrador web.

¿CÓMO SE COMUNICAN EL CLIENTE Y EL SERVIDOR?

Cuando el servidor recibe una petición por parte de un cliente (puede estar en cualquier parte del mundo), realiza una serie de acciones normalmente programadas en un lenguaje de programación de servidor, y envía una respuesta http con el contenido de respuesta y unas cabeceras de comunicación, ya sean error o de ok todo ha ido bien.

Si por alguna razón el servidor estuviera "caído", es decir, apagado o fuera de servicio, el cliente recibiría una respuesta del proveedor de servicios de internet o de la empresa encargada de dar alojamiento a ese equipo. Esto se conoce comúnmente como código 404 con el mensaje relacionado: la página no existe o ya no está disponible.

FLUJO DE DATOS ENTRE CLIENTE Y SERVIDOR

El proceso comienza escribiendo en la barra de direcciones del navegador la URL de la página web que queremos ver. En segundo lugar, el navegador envía el mensaje a través de internet al ordenador remoto (servidor), de acuerdo con un protocolo estandarizado, solicitando la página (archivo) index.php.

El servidor web recibe el mensaje, comprueba que se trata de una petición válida, y al ver que la extension es “php” solicita al intérprete de PHP (que es otro programa que se ejecuta en el servidor web) que le envíe el archivo.

El intérprete PHP lee desde el disco duro del servidor el archivo index.php y empieza a procesar las instrucciones (código de programación) que contenga dicho archivo. Decimos que el intérprete PHP “ejecuta” los comandos contenidos en el archivo y, eventualmente, se comunica con un gestor de base de datos (ejemplos de ellos pueden ser MySql, Oracle, SQL Server, etc.).

Una vez el intérprete PHP termina de ejecutar el código contenido en el archivo y ha recibido toda la información necesaria del gestor de base de datos, envía los resultados al servidor web. El servidor web envía la página al cliente que la había solicitado y el navegador muestra en pantalla la información que le envía el servidor web.

La arquitectura informática de Cliente / Servidor tiene como principio fundamental la agrupación de todo tipo de aplicaciones en dos grupos fundamentales, que sería básicamente dividir aquellas aplicaciones en las que se ofrecen distintos Servicios específicos y funcionalidades acordes, mientras que por otro lado están aquellos programas que hacen uso de los servicios mencionados.

ORIGEN AJAX

AJAX (Asynchronous JavaScript and XML) es una técnica que acelera la interacción con la web de manera drástica, puesto que permite actualizar elementos de una página sin tener que cargarla entera. Esto permite ahorrar mucho ancho de banda puesto que, utilizado convenientemente, se producen más llamadas al servidor pero este devuelve exactamente el contenido que se necesita actualizar.

Con Ajax, se hace posible realizar peticiones al servidor y obtener respuesta de este en segundo plano (sin necesidad de recargar la página web completa) y usar esos datos para, a través de JavaScript, modificar los contenidos de la página creando efectos dinámicos y rápidos.

AJAX actúa a modo de capa intermediaria entre el navegador y el servidor, sirviéndose del lenguaje JavaScript como elemento conector entre las diferentes tecnologías que permiten cubrir todo los aspectos del desarrollo de una aplicación web como son los siguientes:

- **Presentación:** HTML5, CSS3, Bootstrap
- **Gestión e interacción contenido:** Document Object Model (DOM)
- **Intercambio y manipulación datos:** XML, JSON, XSLT
- **Comunicación asíngrona:** objeto SMLHttpRequest

Google hace desde siempre un uso importante de AJAX en sus webs: Gmail, Google Maps, Google Suggest (al autocompletar las búsquedas), etc.

Sin utilizar librerías externas (como Jquery, que simplifica enormemente el uso de AJAX), el código mínimo sería el siguiente (personalizándolo según el objetivo):

```
<html>
<head>
<script>
        function hacerAlgoConAJAX(valor) {
          var xmlhttp = new XMLHttpRequest();
          xmlhttp.onreadystatechange = function() {
            if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
              document.getElementById("idParrafo").innerHTML = xmlhttp.responseText;
            }
          }
          xmlhttp.open("GET", "getDesdeServidor.php?parametro=" + valor, true);
          xmlhttp.send();  }
</script>
</head>
<body>
```

```
<input type="text" onkeyup="hacerAlgoConAJAX(this.value)">
<br /> <span id="idParrafo"></span>
</body>
</html>
```

Necesitaríamos además un script getDesdeServidor.php que generase la salida deseada (echo) para recuperar desde JavaScript y actualizar el elemento necesario (p, span, div, etc.).

En este caso se realiza la llamada asíncrona (es decir, se realiza la actualización de la página) cuando se pulsa una tecla en una caja de texto. Otro evento típico para desencadenar la actualización serían el onChange de un SELECT, para cambiar algo al seleccionar un elemento, por ejemplo, actualizar el contenido de otra lista sincronizada (marca/modelo, provincia/concello, etc.)

JSON

JSON (JavaScript Object Notation) es un formato de texto muy sencillo utilizado para intercambio de datos. *E*s comúnmente utilizado para transmitir datos en aplicaciones web (por ejemplo: enviar algunos datos desde el servidor al cliente, así estos datos pueden ser mostrados en páginas web, o vice versa). Es una notación subconjunto de la utilizada en JavaScript, aunque hoy en día se utiliza desde cualquier lenguaje y plataforma, sobre todo como alternativa a XML. Sus características básicas son:

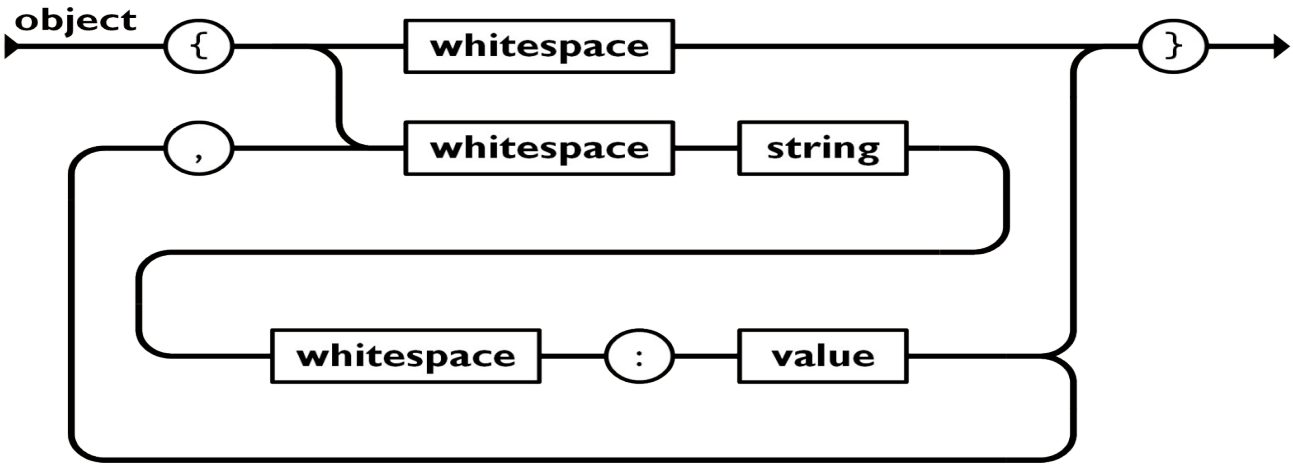
- Los datos se almacenan en pares clave/valor, separados por el carácter dos puntos :
- Las claves y los valores de tipo cadena van encerradas entre comillas dobles
- Los datos se separan por comas ,
- Los arrays se almacenan entre corchetes [ ]
- Los objetos se almacenan entre llaves { }

Fuente: http://www.json.org/json-es.html

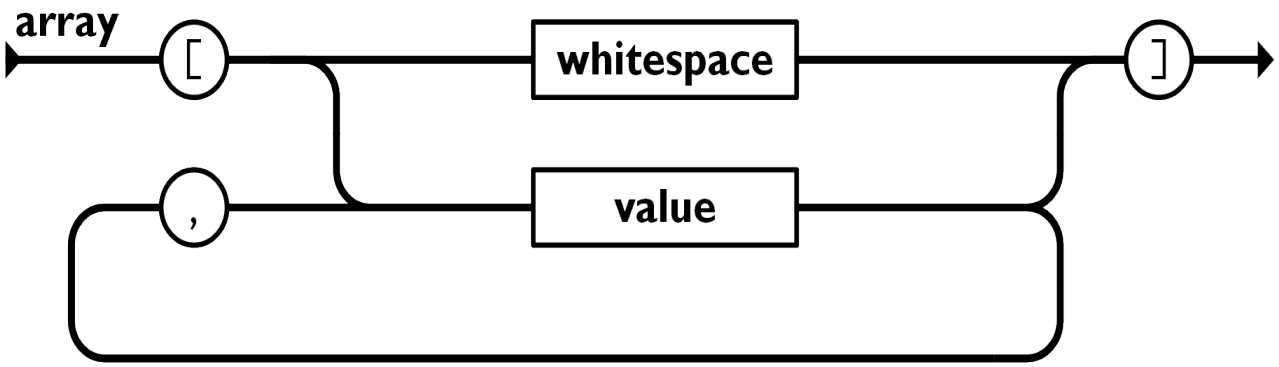
Los JSON son cadenas - útiles cuando se quiere transmitir datos a través de una red. Debe ser convertido a un objeto nativo de JavaScript cuando se requiera acceder a sus datos. Ésto no es un problema, dado que JavaScript posee un objeto global JSON que tiene los métodos disponibles para convertir entre ellos.

Un objeto JSON puede ser almacenado en su propio archivo, que es básicamente sólo un archivo de texto con una extension .json, y una MIME type de application/json.

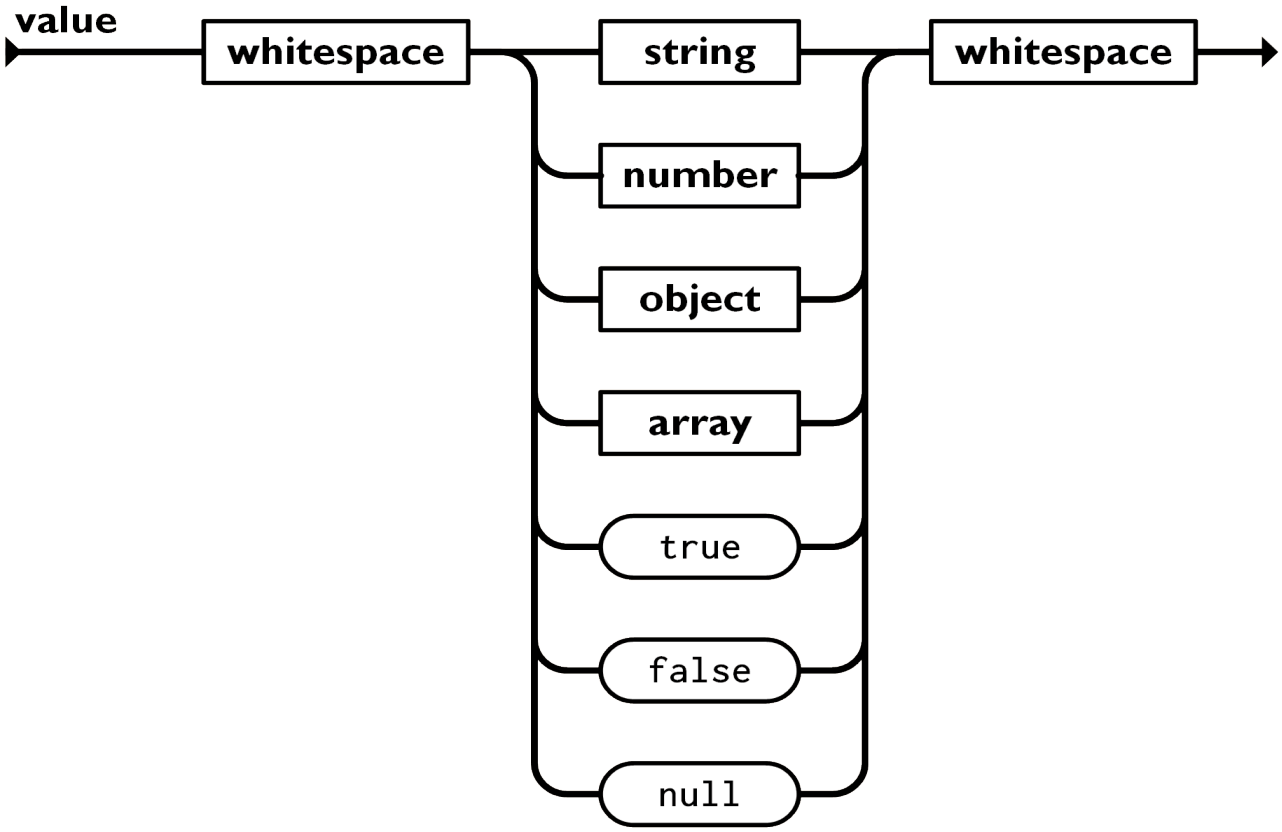
Un objeto es un conjunto desordenado de pares nombre/valor. Un objeto comienza con {l lave de apertura y termine con } llave de cierre. Cada nombre es seguido por :dos puntos y los pares nombre/valor están separados por ,coma.



Un array es una colección de valores. Un array comienza con [corchete izquierdo y termina con]corchete derecho. Los valores se separan por ,coma.



Un valor puede ser una cadena de caracteres con comillas dobles, o un número, o true o false o null, o un objeto o un arreglo. Estas estructuras pueden anidarse.



Generalmente un archivo JSON ocupa menos espacio que su análogo XML, y es mucho más sencillo de analizar (en XML necesitamos un “parser” (analizador) XML en cambio para JSON basta con una sencilla función).

- JSON es sólo un formato de datos — contiene sólo propiedades, no métodos.
- JSON requiere usar comillas dobles para las cadenas y los nombres de propiedades. Las comillas simples no son válidas.
- Una coma o dos puntos mal ubicados pueden producir que un archivo JSON no funcione. Se debe ser cuidadoso para validar cualquier dato que se utilizar (aunque los JSON generados por computador tienen menos probabilidades de tener errores, mientras el programa generador trabaje adecuadamente). Es posible validar JSON utilizando una aplicación como JSONLint.
- JSON Puede tomar la forma de cualquier tipo de datos que sea válido para ser incluido en un JSON, no sólo arrays u objetos. Así, por ejemplo, una cadena o un número único podrían ser objetos JSON válidos.
- A diferencia del código JavaScript en que las propiedades del objeto pueden no estar entre comillas, en JSON, sólo las cadenas entre comillas pueden ser utilizadas como propiedades.

FTECH

Fetch es el nombre de una nueva API para Javascript con la cuál podemos realizar peticiones HTTP asíncronas utilizando promesas y de forma que el código sea un poco más sencillo. La forma de realizar una petición es muy sencilla, básicamente se trata de llamar a fetch y pasarle por parámetro la URL de la petición a realizar:

```
const promise = fetch("/robots.txt");
promise.then(function(response) {
  /* ... */
});
```

El fetch() devolverá una ***promise*** que será aceptada cuando reciba una respuesta y sólo será rechazada si hay un fallo de red o si por alguna razón no se pudo completar la petición.

El modo más habitual de manejar las promesas es utilizando ***.then()***, aunque también se puede utilizar ***async/await***. Esto se suele reescribir de la siguiente forma, que queda mucho más simple y evitamos constantes o variables temporales de un solo uso:

```
fetch("/robots.txt")
  .then(function(response) {
    /** Código que procesa la respuesta **/
  });
```

Al método **.then()** se le pasa una función callback donde su parámetro response es el objeto de respuesta de la petición que hemos realizado. En su interior realizaremos la lógica que queramos hacer con la respuesta a nuestra petición.

Opciones de fetch()

A la función fetch(), al margen de la url a la que hacemos petición, se le puede pasar un segundo parámetro de opciones de forma opcional, un object con opciones de la petición HTTP:

```
const options = {
  method: "GET"
};

fetch("/robots.txt", options)
  .then(response => response.text())
  .then(data => {
    /** Procesar los datos **/
  });
```

Un poco más adelante, veremos como trabajar con la respuesta response, pero vamos a centrarnos ahora en el parámetro opcional options de la petición HTTP. En este objeto podemos definir varios detalles:

Campo	Descripción
method	Método HTTP de la petición. Por defecto,GET. Otras opciones:HEAD,POST, etc...
headers	Cabeceras HTTP. Por defecto,{}.
body	Cuerpo de la petición HTTP. Puede ser de varios tipos:String,FormData,Blob, etc...
credentials	Modo de credenciales. Por defecto,omit. Otras opciones:same-origin einclude.

Lo primero, y más habitual, suele ser indicar el método HTTP a realizar en la petición. Por defecto, se realizará un GET, pero podemos cambiarlos a **HEAD**, **POST**, **PUT** o cualquier otro tipo de método. En segundo lugar, podemos indicar objetos para enviar en el body de la petición, así como modificar las cabeceras en el campo headers:

```
const options = {
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify(jsonData)
};
```

En este ejemplo, estamos enviando una petición **POST**, indicando en la cabecera que se envía contenido **JSON** y en el cuerpo de los datos, enviando el objeto **jsonData**, codificado como texto mediante stringify().

Por último, el campo **credentials** permite modificar el modo en el que se realiza la petición. Por defecto, el valor **omit** hace que no se incluyan credenciales en la petición, pero es posible indicar los valores **same-origin**, que incluye las credenciales si estamos sobre el mismo dominio, o **include** que incluye las credenciales incluso en peticiones a otros dominios

La respuesta Response

Si volvemos a nuestro ejemplo de la petición con **fetch**, observaremos que en el primer **.then()** tenemos un objeto response. Se trata de la respuesta que nos llega del servidor web al momento de recibir nuestra petición:

```
fetch("/users.txt", options)
  .then(response => response.text())
  .then(data => {
    /** Procesar los datos **/
  });
```

Aunque en este ejemplo, simplemente estamos utilizando una función flecha que hace un **return** implícito de la promesa que devuelve el método .text(), dicho objeto response tiene una serie de propiedades y métodos que pueden resultarnos útiles al implementar nuestro código.
Por el lado de las propiedades, tenemos las siguientes:

Propiedad	Descripción
.status	Código de error HTTP de la respuesta (100-599).
.statusText	Texto representativo del código de error HTTP anterior.
.ok	Devuelve true si el código HTTP es 200 (o empieza por 2).
.headers	Cabeceras de la respuesta.
.url	URL de la petición HTTP.

Las propiedades .**status** y **statusText** nos devuelven el código de error HTTP de la respuesta en formato numérico y cadena de texto respectivamente. La propiedad .ok que nos devuelve true si el código de error de la respuesta es un valor del rango 2xx, es decir, que todo ha ido correctamente. Así pues, tenemos una forma práctica y sencilla de comprobar si todo ha ido bien al realizar la petición:

```
fetch("/users.txt")
  .then(response => {
    if (response.ok)
      return response.text()
  })
```

Por último, tenemos la propiedad .headers que nos devuelve las cabeceras de la respuesta y la propiedad .url que nos devuelve la URL completa de la petición que hemos realizado.

Procesando la respuesta

Por otra parte, la instancia **response** también tiene algunos métodos interesantes, la mayoría de ellos para procesar mediante una promesa los datos recibidos y facilitar el trabajo con ellos**:**

Método	Descripción
.text()	Devuelve una promesa con el texto plano de la respuesta.
.json()	Idem, pero con un objeto json. Equivale a usar JSON.parse().
.blob()	Idem, pero con un objeto Blob (binary large object).
.arrayBuffer()	Idem, pero con un objeto ArrayBuffer (buffer binario puro).
.formData()	Idem, pero con un objeto FormData (datos de formulario).
.clone()	Crea y devuelve un clon de la instancia en cuestión.
Response.error()	Devuelve un nuevo objeto Response con un error de red asociado.
Response.redirect(url, code)	Redirige a una url, opcionalmente con un code de error.

Observa que en los ejemplos anteriores hemos utilizado response.text(). Este método indica que queremos procesar la respuesta como datos textuales, por lo que dicho método devolverá una con los datos en texto plano, facilitando trabajar con ellos de forma manual:

```
fetch("/robots.txt")
  .then(response => response.text())
  .then(data => console.log(data));
```

Observa que en este fragmento de código, tras procesar la respuesta con response.text(), devolvemos una con el contenido en texto plano. Esta se procesa en el segundo .then(), donde gestionamos dicho contenido almacenado en data.

Ten en cuenta que tenemos varios métodos similares para procesar las respuestas. Por ejemplo, el caso anterior utilizando el método response.json() en lugar de response.text() sería equivalente al siguiente fragmento:

```
fetch("/contents.txt")
  .then(response => response.text())
  .then(data => {
    const json = JSON.parse(data);
    console.log(json);
  });
```

Como se puede ver, con ***response.json()*** nos ahorraríamos tener que hacer el ***JSON.parse()*** de forma manual, por lo que el código es algo más directo.

Se puede gestionar las promesas de dos formas distintas, usando .**then** o usando **async/await,** pero antes de explicar las características de cada uno de los dos métodos, vamos a definir lo que es una **promise**

¿Qué son las promises?

Como su propio nombre indica, una promesa es algo que, en principio pensamos que se cumplirá, pero en el futuro pueden ocurrir varias cosas:

- La promesa se cumple (promesa resuelta)
- La promesa no se cumple (promesa rechazada)
- La promesa se queda en un estado incierto indefinidamente (promesa pendiente)

Las promesas en Javascript se representan a través de un object, y cada promesa estará en un estado concreto: **pendiente**, **aceptada** o **rechazada**. Además, cada promesa tiene los siguientes métodos:

Métodos	Descripción
.then(resolve)	Ejecuta la función callback resolve cuando la promesa se cumple.
.catch(reject)	Ejecuta la función callback reject cuando la promesa se rechaza.
.then(resolve,reject)	Método equivalente a las dos anteriores en el mismo .then().
.finally(end)	Ejecuta la función callback end tanto si se cumple como si se rechaza.

La forma general de tratar una promesa es utilizando el **.then()** con un sólo parámetro, tal y como se muestra en el siguiente ejemplo, puesto que muchas veces lo único que nos interesa es realizar una acción cuando la promesa se cumpla, aunque podemos hacer uso del método catch() para actuar cuando se rechaza una promesa.

```
fetch("/users.txt")
  .then(function(response) {
    /* Código a realizar cuando se cumpla la promesa */
  })
  .catch(function(error) {
    /* Código a realizar cuando se rechaza la promesa */
  });
```

Si queremos encadenar promesas:

```
fetch("/users.txt)
  .then(response => {
    return response.text(); // Devuelve una promesa
  })
  .then(data => {
    console.log(data);
  })
  .catch(error => { /* Código a realizar cuando se rechaza la promesa */ });
```

No olvides indicar el **return** para poder encadenar las siguientes promesas con **.then()**. Tras un .catch() también es posible encadenar .then() para continuar procesando promesas.

El código anterior se puede simplificar utilizando las funciones flecha

```
fetch("/users.txt")
  .then(response => response.text())
  .then(data => console.log(data))
  .finally(() => console.log("Terminado."))
  .catch(error => console.error(data));
```

Además que hemos añadido el método .finally() para añadir una función callback que se ejecutará tanto si la promesa se cumple o se rechaza, lo que nos ahorrará tener que repetir la función en el .then() como en el .catch().

Algo muy importante a tener en cuenta, es que el código que ejecutamos en el interior de un .then() es código asíncrono no bloqueante:

- **Asíncrono:** Porque probablemente no se ejecuterá de inmediato, sino que tardará en ejecutarse.
- **No bloqueante:** Porque mientras espera ser ejecutado, no bloquea el resto del programa.

Esto significa que cuando llegamos a un **.then(),** el sistema no se bloquea, sino que deja la función «pendiente» hasta que se cumpla la promesa, pero mientras, continua procesando el resto del programa.

Observa el siguiente ejemplo:

```
fetch("//users.txt")
  .then(response => response.text())
  .then(data => {
    console.log("Código asíncrono");
  });

console.log("Código síncrono")
```

Aunque el console.log("Código asíncrono") figure unas líneas antes del console.log("Código síncrono"), se mostrará más tarde. Esto ocurre porque el console.log() del interior del .then() no ocurre inmediatamente, y al no ser bloqueante, se continua con el resto del programa hasta que se ejecute, que lo retomará.

Crear promesas

A continuación vamos a mostrar un ejemplo en el que se crea una promesa. En este caso al objeto **new Promise()** se le pasa por parámetro una función con dos callbacks:

- El primer callback, **resolve**, lo utilizaremos cuando ***se cumpla la promesa***.
- El segundo callback, **reject**, lo utilizaremos cuando ***se rechace la promesa***.

```
const doTask = (iterations) => {
  return new Promise( (resolve, reject) => {
    const numbers = [];

    for (let i = 0; i < iterations; i++) {
      const number = 1 + Math.floor(Math.random() * 6);
      numbers.push(number);
      if (number === 6) {
        reject({
          error: true,
          message: "Se ha sacado un 6"
        });
      }
    }
```

```
    resolve({
      error: false,
      value: numbers
    });
  })
};

doTask(10)
  .then(result => console.log("Tiradas correctas: ", result.value))
  .catch(err => console.error("Ha ocurrido algo: ", err.message));
```

También puedes consultar el ejemplo ***EjemploManejoPromesa*** subido a la plataforma

Para entender el funcionamiento de todos estos conceptos es necesario entender que es la sincronía en un lenguaje de programación

¿Qué es la asincronía?

La asincronía es uno de los conceptos principales que rige el mundo de Javascript. Cuando comenzamos a programar, normalmente realizamos tareas de forma síncrona, llevando a cabo tareas secuenciales que se ejecutan una detrás de otra, de modo que el orden o flujo del programa es sencillo y fácil de observar en el código:

```
primera_funcion();   // Tarea 1: Se ejecuta primero
segunda_funcion();   // Tarea 2: Se ejecuta cuando termina primera_funcion()
tercera_funcion();   // Tarea 3: Se ejecuta cuando termina segunda_funcion()
```

Sin embargo, en el mundo de la programación, tarde o temprano necesitaremos realizar operaciones **asíncronas**, especialmente en ciertos lenguajes como Javascript, donde tenemos que realizar tareas que tienen que esperar a que ocurra un determinado suceso que no depende de nosotros, y reaccionar realizando otra tarea sólo cuando dicho suceso ocurra.

***Nota:** La programación asíncrona es una técnica que permite a tu programa iniciar una tarea de larga duración y seguir respondiendo a otros eventos mientras esta tarea se ejecuta, en lugar de tener que esperar hasta que esta tarea haya terminado.*
El código síncrono se lee y ejecuta desde la primera hasta la última línea, en orden. El código asíncrono, en cambio, no respetará necesariamente el orden de las líneas de código y puede tener funciones que se leen y ejecutan de forma simultánea o aleatoria.

Lenguaje no bloqueante

Para comprender mejor el concepto de non blocking en JavaScript, es esencial conocer el funcionamiento del event loop (bucle de eventos). El event loop es un mecanismo que controla cómo se manejan las tareas en JavaScript. Cuando un programa ejecuta una tarea que llevará un tiempo significativo para completarse, en lugar de detener toda la ejecución y esperar a que termine, el event loop permite que el programa siga adelante con otras tareas disponibles. Esto asegura que el programa sea más receptivo y que los usuarios no experimenten bloqueos o retrasos frustrantes.

¿Cómo puede JavaScript ejecutar tareas sin bloquear?

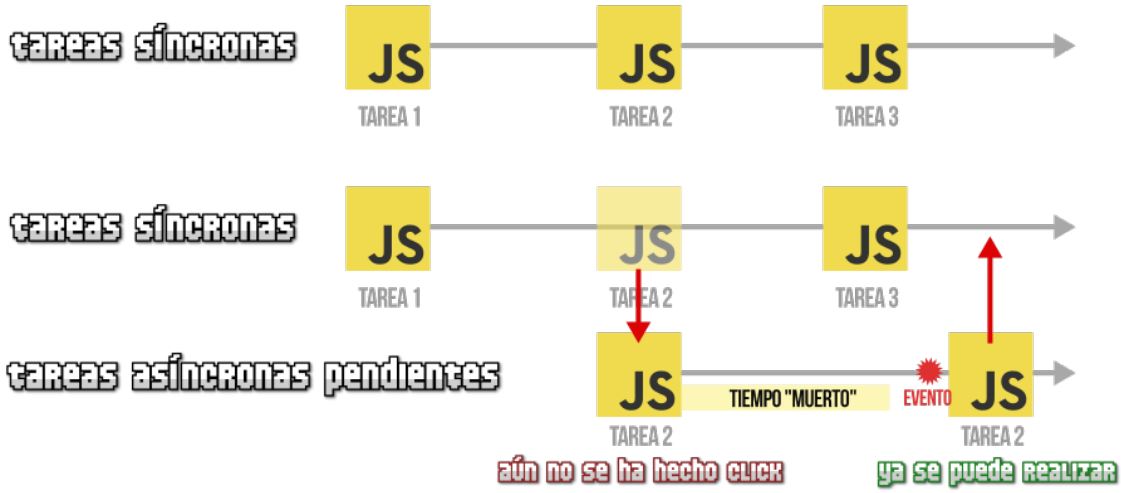
JavaScript es un lenguaje de programación de un solo subproceso, lo que significa que ejecuta una tarea a la vez. Sin embargo, gracias al modelo de concurrencia basado en el **event loop**, puede lograr el efecto de ejecución no bloqueante. Cuando se llama una función asíncrona, la tarea se coloca en una cola de tareas pendientes y el programa continúa con la ejecución de otras tareas disponibles. Una vez que la tarea asíncrona se completa, se procesa el resultado y se le notifica al programa.

En JavaScript, uno de los métodos comunes para lograr el comportamiento no bloqueante es el uso de **funciones asíncronas y callbacks**. Las funciones asíncronas le permiten a los desarrolladores escribir código que parece síncrono, pero que en realidad se ejecuta de forma asíncrona, lo que evita bloquear el hilo principal del programa. Por otro lado, los **callbacks** son **funciones que se pasan como argumentos a otras funciones y se ejecutan una vez que se completa una operación asíncrona.**

Ventajas del non blocking en JavaScript

- **Mayor capacidad de respuesta:** las aplicaciones que utilizan técnicas non blocking pueden responder rápidamente a las acciones del usuario, lo que mejora significativamente la experiencia del usuario y evita la sensación de «congelamiento» en la interfaz.
- **Eficiencia en el uso de recursos:** al permitir que el programa realice múltiples tareas a la vez, se optimiza el uso de recursos del sistema y se evita el desperdicio de tiempo de CPU.
- **Escalabilidad:** las aplicaciones non blocking son más escalables, lo que significa que pueden manejar una mayor cantidad de solicitudes y datos sin sacrificar el rendimiento.
- **Evita bloqueos:** las técnicas non blocking garantizan que el programa nunca se bloquee, lo que significa que las tareas críticas pueden ejecutarse sin problemas y sin afectar a otras operaciones.

Cuando hablamos de Javascript, habitualmente nos referimos a él como un lenguaje **no bloqueante**. Con esto queremos decir que las tareas que realizamos no se quedan bloqueadas esperando ser finalizadas, y por consiguiente, evitando proseguir con el resto de tareas.



Imaginemos que la segunda_funcion() del ejemplo anterior realiza una tarea que depende de otro factor, como por ejemplo un click de ratón del usuario. Si hablasemos de un lenguaje bloqueante, hasta que el usuario no haga click, Javascript no seguiría ejecutando las demás funciones, sino que se quedaría bloqueado esperando a que se terminase esa segunda tarea:

Pero como Javascript es un lenguaje no bloqueante, lo que hará es mover esa tarea a una lista de tareas pendientes a las que irá «prestándole atención» a medida que lo necesite, pudiendo continuar y retomar el resto de tareas a continuación de la segunda.

Pero esto no es todo. Ten en cuenta que pueden existir múltiples tareas asíncronas, dichas tareas puede que terminen realizándose correctamente (o puede que no) y ciertas tareas pueden depender de otras, por lo que deben respetar un cierto orden. Además, es muy habitual que no sepamos previamente cuanto tiempo va a tardar en terminar una tarea, por lo que necesitamos un mecanismo para controlar todos estos factores: ***las promesas***

Ejemplos de tareas asíncronas

En Javascript no todas las tareas son asíncronas, pero hay ciertas tareas que si lo son, y probablemente se entiendan mejor con ejemplos reales:

- Un **fetch()** a una URL para obtener un archivo .json.
- Un **play()** de un .mp3 que creamos mediante un new Audio().
- Una tarea programada con **setTimeout()** que se ejecutará en el futuro.

Todos estos ejemplos se realizan mediante tareas asíncronas, ya que realizan un procedimiento que podría bloquear la ejecución del resto del programa al tardar mucho: la descarga de un fichero grande desde un servidor lento, una conexión a internet muy lenta, un dispositivo saturado a la hora de comunicarse con el sensor del móvil, etc...

¿Cómo gestionar la asincronía?

Teniendo en cuenta el punto anterior, debemos aprender a buscar mecanismos para dejar claro en nuestro código Javascript, que ciertas tareas tienen que procesarse de forma asíncrona para quedarse a la espera, y otras deben ejecutarse de forma síncrona.
En Javascript existen varias formas de gestionar la asincronía. Aquí citamos algunas de ellas:

- Funciones callbacks: En este caso estamos usando una función callback para pasársela como parámetro a setTimeout(), que es otra función. Este tipo de funciones podemos usarlas como un primer intento de manejar la sincronía

```
setTimeout(() => {
  console.log("Código asíncrono.");
}, 2000);

console.log("Código síncrono.");
```

- Promesas
- Async/Await

Promesas usando async/await

- **La palabra clave await:** Lo que hace await es detener la ejecución y no continuar. En el caso de una promesa con async/await, se espera a que se resuelva la promesa y hasta que no lo haga no continúa. Aquí tenemos un código bloqueante a diferencia del **.then()**
- **La palabra clave async:** Para poder utilizar el await dentro de una función, sólo tenemos que definirla como asíncrona y para ello debemos utilizar async, y al llamarla utilizar el await

Comprobamos el funcionamiento con este ejemplo:

- Creamos una función request(url) que definimos con async
- Llamamos a fetch utilizando await para esperar y resolver la promesa
- Comprobamos si todo ha ido bien usando response.ok
- Llamamos a response.text() utilizando await y devolvemos el resultado

```
const request = async (url) => {
  const response = await fetch(url);
  if (!response.ok)
```

```
    throw new Error("WARN", response.status);
  const data = await response.text();
  return data;
}

const resultOk = await request("/robots.txt");
const resultError = await request("/nonExistentFile.txt");
```

Una vez hecho esto, podemos llamar a nuestra función request y almacenar el resultado, usando nuevamente await.

Bibliografía:

- ***Aprender a programar web (Javier Gómez)***
- ***Desarrollo web en entorno cliente (Ed. Garceta)***
- ***Lenguajesjs.com***