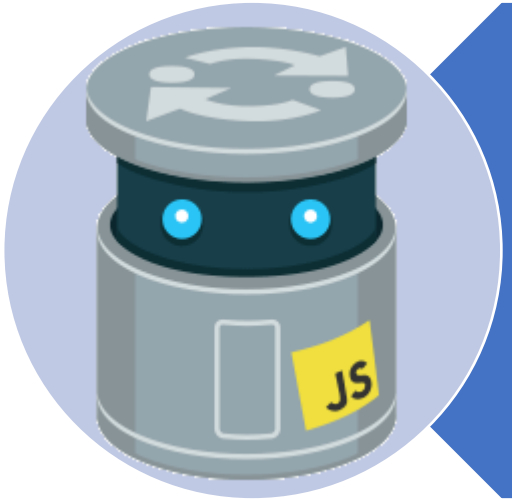
Jorge García Flores

Desarrollo Web en Entorno Cliente

Tema 14

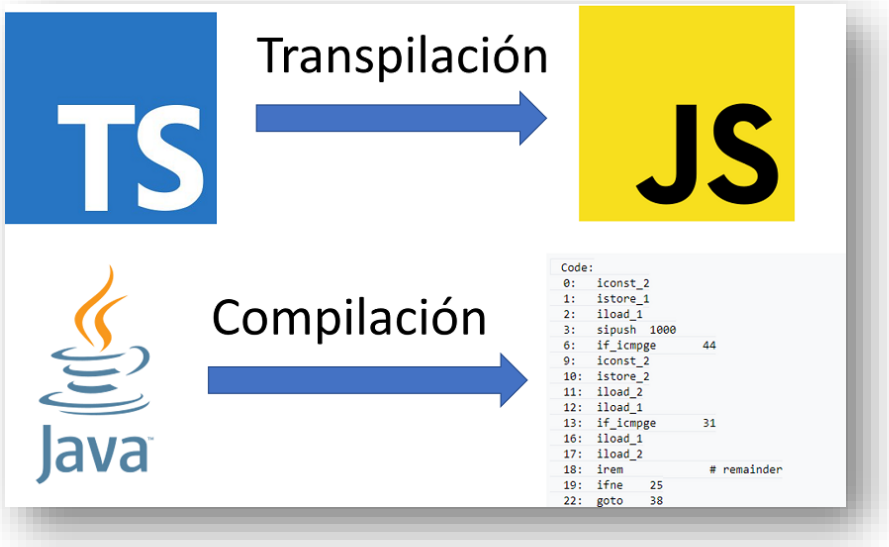


Import. Excepciones. AJAX

import

Un transpilador es un tipo especial de compilador que traduce de un lenguaje fuente a otro lenguaje fuente, de un nivel de abstracción parecido.

Se diferencia de los compiladores tradicionales en que estos reciben como entrada ficheros conteniendo código fuente y generan código máquina del más bajo nivel.



La sentencia *`import`* se definió en el ES6 y se usa para importar variables, constantes y funciones que están en otros ficheros JS, llamados módulos externos, y que han sido, de manera explícita exportados, anteponiendo la palabra reservada *`export`*.

Está implementada en muchos transpiladores, como *Typescript* y *Babel*, y en *bundles* como *Rollup* y *Webpack*.

El *bundling* o empaquetamiento es el proceso de agrupar todas las dependencias y componentes de una aplicación en un solo archivo, conocido como *bundle*. Este archivo bundle contiene todo el código necesario para que la aplicación funcione correctamente, incluyendo las dependencias y los componentes.

import

Para hacer uso de esta característica, en el que se va a hacer uso de código JS ubicado en otro fichero hay que colocar, preferiblemente al principio del mismo, la sentencia *import*:

La sentencia *import* tiene la siguiente sintaxis:

```
import defaultExport from "nombre-fichero-modulo";
import * as nombre from "nombre-fichero-modulo";
import {nombre-del-export } from "nombre-fichero-modulo";
import {nombre-del-export as alias } from "nombre-fichero-modulo";
import {nombre-del-export1,nombre-del-export2 } from "nombre-fichero-modulo";
import {export1, export2 as alias2 , [...] } from "nombre-fichero-modulo";
import defaultExport, { export [ , [...] ] } from "nombre-fichero-modulo";
import defaultExport, * as nombre from "nombre-fichero-modulo";
import "module-name";
```

Cualquiera de las líneas presentadas anteriormente puede utilizarse, pero cada una tiene un significado diferente.

import

Para que las definiciones, constantes, objetos, funciones, clases, etc. puedan exportarse hay que poner la palabra reservada *export* delante de cada una de ellas.

Ejemplo:

```
export const mensaje='Hola';
export function saludo(){console.log('Hola');}
```

También puede exportarse todo a la vez, utilizando la palabra reservada *export* seguida de llaves { } y dentro el cuerpo de las llaves aquello que se quiera exportar.

Ejemplo:

```
export {
        mensaje,
        saludo()
}
```

import

En el código HTML, donde se carga el fichero script de JS que contiene los *import*, hay que indicar que es un módulo, mediante el atributo *type="module"*.

Ejemplo, el fichero se llama *app.js* y utiliza código exportado desde otro fichero, el llamado *exports.js*; el fichero HTML que daría así:

```
<script src="./app.js" type="module"></script>
```

Desde el fichero HTML no se indica o menciona el fichero *exports.js*.

En cambio, en el fichero *app.js* hay que realizar la importación mediante la sentencia *import*:

```
import * from "./exports.js";
```

import

Ejemplo, se quiere hacer uso de una función llamada *saludo()*, definida en el fichero *componentes.js*, para importar únicamente esta función y poderse utilizarla en otro fichero JS, se importa y, a continuación, ya puede utilizarse en este fichero:

```
import { saludo } from './componentes.js';

saludo();
```

En el fichero *componentes.js* hay que anteponer la palabra *export* a la función para que se pueda exportar:

```
export function saludo()
{
   console.log('Hola');
}
```

import

Además, se puede crear una exportación por defecto mediante la sentencia *export default* antepuesta a aquello que se quiere exportar.

Solo puede haber una exportación por defecto en un fichero, por lo que no puede aparecer más de una sentencia *export default*.

En el fichero de JS, en el que se quiere utilizar el código exportado por defecto, se utiliza la sentencia *import* con *defaultExport*, así, cualquiera de las siguientes sería válida:

```
import defaultExport from "nombre-fichero-modulo";
import defaultExport, { export [ , [...] ] } from "nombre-fichero-modulo";
import defaultExport, * as nombre from "nombre-fichero-modulo";
```

import

Ejemplo, supóngase que se quiere exportar por defecto una función llamada *saludo()*, definida en el fichero *componentes.js*, en el fichero *componentes.js* donde está definida dicha función se pondría:

```
export default function saludo()
{
   console.log('Hola');
}
```

Y en el fichero que se quiere importar:

```
import saludo from "./componentes.js";
```

Se puede redefinir su nombre en el fichero importado, por lo que También sería válido:

```
import miSaludo from "./componentes.js";
```

import

La sentencia *export default* antepuesta a una definición de constante o variable dará error. En este caso, primero se hace la definición y, a continuación, en otra línea de código se coloca *export default* seguido del nombre de lo que se quiere exportar por defecto.

Ejemplo:

```
const nombre='Jorge';
export default nombre;
```

Por último, hay que tener en cuenta que cuando se importa un fichero, el contenido que hay en él se ejecuta.

Ejemplo, el siguiente código, sacará por consola la palabra *Jorge*, cuando se importe el fichero donde está contenido:

```
const nombre='Jorge';
console.log(nombre);
export default nombre;
```

excepciones

Las excepciones provienen, normalmente, de errores, como los que se generan por imprevistos que ocurren durante la ejecución de un programa, produciendo situaciones anómalas que impiden o alteran el comportamiento o flujo normal del mismo.

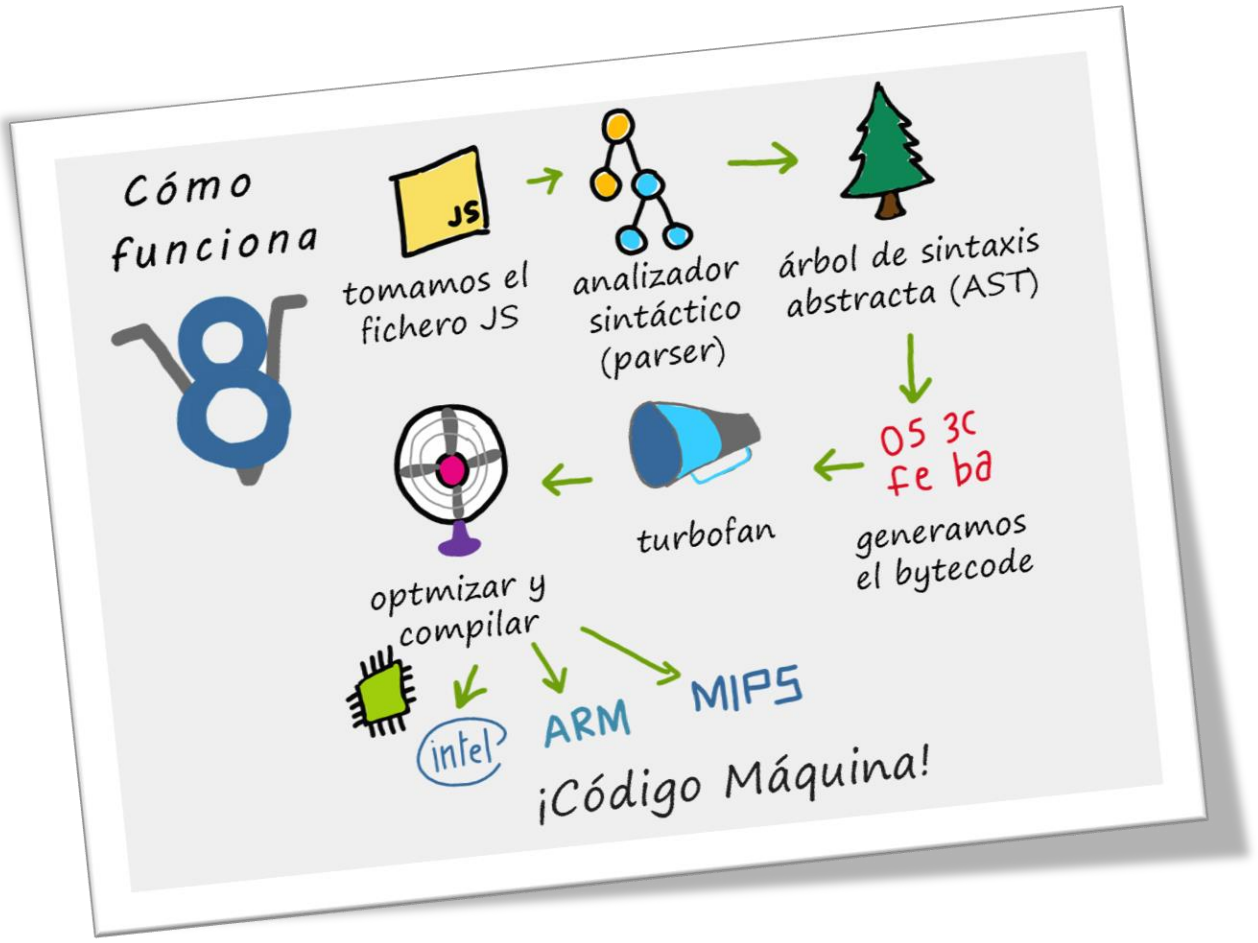
Las excepciones permiten tener un código destinado a manejar los errores, separado de la lógica de aplicación del programa.

Al producirse una anomalía, un error, JavaScript crea un objeto de tipo *Error* con dos propiedades: *name* y *message*, y desciende en la pila de ejecución hasta encontrar un manejador que puede tratar la excepción, el cual toma el control en ese momento de la aplicación.

El manejador de la excepción puede acceder al objeto *Error* creado y realizar, mediante código, acciones en respuesta al mismo, como puede ser: intentar recuperar el error y volver a un flujo normal del programa, detener de forma segura el programa y evitar situaciones inconsistentes, como, por ejemplo, el que se produce al cancelar una compra.

Si no hay ningún manejador que se haga cargo de la excepción, ésta llega hasta la consola de errores y el objeto *Error* es mostrado en la misma y el programa finaliza.

excepciones



No hay que confundir los errores de sintaxis, *SyntaxError*, que se deben a que no se ha utilizado correctamente la sintaxis del lenguaje JS, con las excepciones.

En el primer caso, el código no llega a ejecutarse ya que el analizador sintáctico del motor de JS detecta que el código contiene errores (sintácticos) y lo muestra en la consola.

```
Uncaught SyntaxError: Invalid or unexpected token    VM99:1
```

try catch

Las excepciones se manejan usando las sentencias *try…catch*, que consisten en los siguientes bloques:

- Un bloque *try* (intentar), que contiene una o más instrucciones. Si en este bloque se produce algún error, se detiene la ejecución, se construye el error y se lanza una excepción que será tratada por las sentencias del bloque *catch*, si está definido, si no lo está la excepción se propaga en la pila de ejecución, como haría normalmente una excepción. En caso de que no se produzca error, el bloque *catch* se ignora.

- Ninguno o un bloque *catch* (atrapar) que especifica qué hacer si una excepción es lanzada en el bloque *try*, o en alguna función o método invocado dentro del bloque *try*.

- Ninguno o un bloque El bloque *finally* (finalmente) que se ejecuta inmediatamente después de *try…catch*, se haya o no producido una excepción. Normalmente sirve para hacer tareas relacionas con la limpieza o depuración de código, como cerrar archivos abiertos que se quedarían abiertos al producirse una excepción, escribir en algún repositorio de tipo *log*, que permita comprobar el funcionamiento del código,…

try catch

La sintaxis de *try…catch*, es la siguiente:

```
try {
      [...sentencias]
} catch ([error]) {
      [...sentencias]
} finally {
      [...sentencias]
}
```

En la sentencia catch se puede especificar que reciba un objeto de tipo error, indicándoselo a continuación entre paréntesis, como si fuera un argumento de una función.

Esto permite acceder al tipo de error y a su descripción, y en base a ello ejecutar unas instrucciones u otras.

throw

Existe la posibilidad de lanzar una excepción de manera intencionada desde el código mediante la palabra reservada *throw* (lanzar).

Esto puede ser útil, por ejemplo, si un *catch* atrapa un tipo de excepción, una vez comprobado su tipo, que no se quiere manejar en ese *catch*, sino en otro que es más adecuado, por lo que se quiere que se siga propagando hasta que lo alcance, o si se detecta que una variable tiene un valor que no es invalido, pero no es adecuado para el funcionamiento de la aplicación, como, por ejemplo, un ingreso en una cuenta bancaria que fuera negativo (no sería ingreso, debería ser un abono)…

La sintaxis de la expresión *throw* es la siguiente:

```
throw expresión;
```

Donde expresión es la expresión que se lanzará.

throw

Se puede lanzar cualquier expresión, no solo expresiones de un tipo específico.

El siguiente código lanza excepciones de distintos tipos:

```
throw 'Error2';   // tipo String
throw 42;         // tipo Number
throw true;       // tipo Boolean
throw {function() {return "¡Soy un objeto!";}};
```

Se puede especificar un objeto cuando se lanza una excepción y posteriormente, acceder a las propiedades de dicho objeto desde el bloque *catch*.

throw

Ejemplo:

```
function excepcion(mensaje) {
    this.mensaje=mensaje;
    this.nombre='excepcionUsuario';
}
excepcion.prototype.aCadena=function() {
    return this.nombre + ' : '  + this.mensaje;
}

const correo='pepe';

try{
    if(!(/@/.test(correo)))
        throw new excepcion('La dirección no es un e-mail válido');
}catch(e){console.error(e.aCadena());}
```

throw

Además, se puede crear un objeto de tipo *Error* y lanzar dicho objeto a través de *throw*.

La sintaxis del constructor del objeto *Error* es la siguiente:

```
Error()
Error(mensaje)
Error(mensaje, opciones)
Error(mensaje, nombre_de_archivo)
Error(mensaje, nombre_de_archivo, numero_de_linea)
```

mensaje es una descripción del error.

opciones es un objeto en el que se definen otras características como la causa (cause) del error.

nombre_de_archivo es el valor de la propiedad *fileName* del objeto *Error*. Por defecto, es el nombre del archivo que contiene el código que llamó al constructor *Error()*.

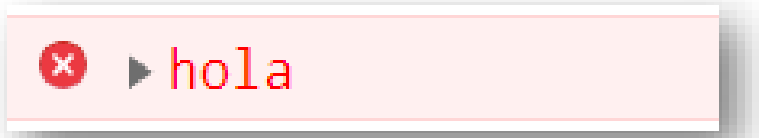
numero_de_linea es el valor de la propiedad *lineNumber* en el del objeto Error. Por defecto, es el número de línea que contiene la invocación del constructor *Error()*.

throw

No obstante, solo las propiedades *Error.prototype.message* y *Error.prototype.name*, que se corresponden con el nombre dado al error y la descripción del mismo, son lo estándar, el resto de propiedades dependen del navegador o motor de JS que implementen.

Ejemplo:

```
try{
    try {
         throw "hola";
    } catch (err) {
        throw new Error('Nuevo mensaje de error', {cause: err});}
}catch (err) {console.error(err.cause);}
```



AJAX (*Asynchronous JavaScript And XML* o JavaScript asíncrono y XML), es una técnica que permite la comunicación con otros servicios para obtener, añadir, modificar y borrar información sin tener que recargar la página web desde el servidor de manera asíncrona (sin detener el código a espera de la respuesta).

AJAX es un término que describe dos capacidades que presentan los navegadores desde hace años, pero que habían sido ignoradas por muchos desarrolladores Web, hasta que hace poco surgieron aplicaciones como *Gmail*, *Google suggest* y *Google Maps*.

Las dos capacidades son:

- Hacer peticiones al servidor sin tener que volver a cargar la página.
- Analizar y trabajar con documentos XML.

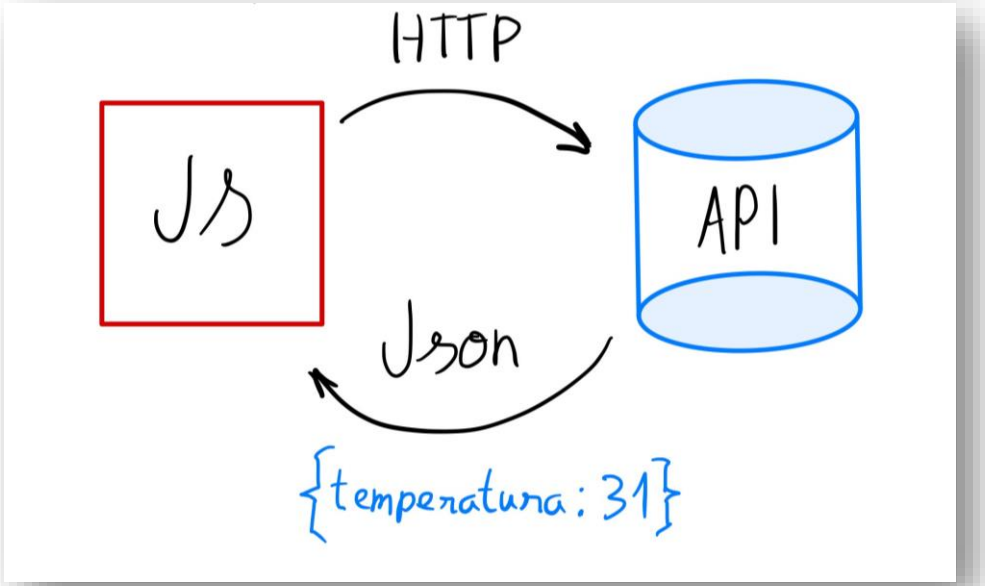
ajax

Permite acceder a una cantidad ilimitada de información por parte de otros servicios (estadísticas, bases de datos, cálculos…) que una vez tratada enriquecerán la navegación con estructuras nuevas de HTML o elementos que mejoran la experiencia.

Por ejemplo, si se está haciendo un buscador de viajes, informar del tiempo hará en el destino cuando se indique la fecha del viaje mejoraría la experiencia de usuario.

En este caso preguntaría a un servidor externo (mediante una API) dicha información para después crear un apartado dentro de la web con la temperatura.

Esas predicciones del tiempo no están en el código original, se preguntan y se obtienen sin recargar la página.



XMLHttpRequest

Para realizar una petición HTTP usando JS es necesario crear una instancia de una clase que provea esta funcionalidad.

Esta clase fue inicialmente introducida en Internet Explorer como un objeto ActiveX, llamado *XMLHTTP*.

Posteriormente, Mozilla, Safari y otros navegadores lo siguieron, implementando una clase *XMLHttpRequest* que soportaba los métodos y las propiedades del objeto ActiveX original.

Para utilizar AJAX hay crear una instancia de esa clase:

```
if (window.XMLHttpRequest) { // Mozilla, Safari, ...
    http_request = new XMLHttpRequest();
}
else if (window.ActiveXObject) { // IE
    http_request = new ActiveXObject("Microsoft.XMLHTTP");
}
```

ActiveX es un entorno para definir componentes de software utilizables de forma independiente del lenguaje de programación.

ActiveX fue presentado en 1996 por Microsoft como una evolución de sus tecnologías *Component Object Model* (COM) y *Object Linking and Embedding* (OLE).

XMLHttpRequest

A continuación, se indica que función se encargará de procesar la respuesta del servidor a la petición enviada.

Esto se realiza asignando el nombre de la función de que se va a utilizar a la propiedad *onreadystatechange* de la instancia:

```
http_request.onreadystatechange = ()=>{
    // procesar la respuesta
};
```

Después, se realiza la petición mediante los métodos *open()* y *send()* de la clase *XMLHttpRequest*:

```
http_request.open('GET','http://www.example.org/algun.archivo',true);
http_request.send();
```

XMLHttpRequest.open

El primer parámetro de la llamada a *`open()`* de *`XMLHttpRequest`* es el método *HTTP request* que se va a emplear, este puede ser GET, POST, HEAD o cualquier otro método que se quiera usar y sea aceptado por el servidor.

El nombre del método se escribe en mayúsculas, sino algunos navegadores podrían no procesar la petición.

El segundo parámetro es la URL de la página que se está pidiendo. Como medida de seguridad, no es posible acceder a páginas de dominios de terceras personas, por ello, se debe conocer el dominio exacto de todas las páginas a las que se quiere acceder o se obtendrá un error de permiso denegado.

El tercer parámetro establece si la petición es asíncrona, si se establece a *`true`* la ejecución de la función de JS sin detenerse a la espera de la respuesta del servidor, en caso de que se ponga a *`false`*, el código se detendrá hasta obtener la respuesta del servidor.

XMLHttpRequest.send

El método *send* de *XMLHttpRequest* puede recibir un parámetro en forma de cadena de texto, que indica cualquier información que se quiera enviar al servidor utilizando el método POST para la petición.

Pero, previamente, es necesario cambiar el tipo MIME en la cabecera de la petición a *'application/x-www-form-urlencoded'*

Ejemplo:

```
const datos="name=jorge&user=admin";
http_request.setRequestHeader('Content-Type','application/x-www-form-urlencoded');
http_request.open('POST','http://www.example.org/algun.archivo',true);
http_request.send(datos);
```

XMLHttpRequest.readyState

Se puede comprobar el estado de una petición, o el estado de un objeto *`XMLHttpRequest`* a través de una propiedad de dicho objeto llamada *`readyState`*.

Esta propiedad puede contener un número entero del 0 al 4 que indica el estado en el que se encuentra una petición.

La lista de valores, junto con su significado, de la propiedad *`readyState`* son:

- ***0*** conexión con el servidor no iniciada
- ***1*** conexión con el servidor establecida
- ***2*** petición recibida en el servidor
- ***3*** el servidor está procesando la respuesta
- ***4*** la petición se completó y la respuesta está disponible

XMLHttpRequest.status

Existe otra propiedad de *`XMLHttpRequest`* llamada *`status`* que contiene un código numérico entero enviado por el servidor y que indica el tipo de respuesta dada a la petición.

Puede tomar valores como:

- ***200*** indicando que se generó la respuesta y no hubo problemas.
- ***404*** algo fue mal y no se encontró y generó la respuesta.
- ***500*** algo ajeno fue mal en el servidor y no se generó la respuesta.

Esta propiedad tiene un equivalente, pero con el tipo de respuesta dada a la petición en forma de cadena de texto, por ejemplo: "*`OK`*", "*`Not found`*", "*`Internal Server Error`*"…

Se recomienda repasar estos códigos:

https://www.w3schools.com/tags/ref_httpmessages.asp

XMLHttpRequest.responseText
XMLHttpRequest.responseXML

La propiedad *`responseText`* de *`XMLHttpRequest`* contiene la respuesta del servidor, en forma de cadena de texto, cuando se complete la petición.

Es muy frecuente que la respuesta, a pesar de estar en formato texto, sea realmente una cadena JSON, por lo que puede procesarse con el método *`parse()`* del objeto *`JSON`*.

De igual manera, la propiedad *`responseXML`* de *`XMLHttpRequest`* contiene la respuesta del servidor, en formato XML.

La información contenida en formato XML puede procesarse mediante funciones de JavaScript que interactúan con el DOM, como las ya estudiadas en anteriores temas.

ejemplo

En el siguiente código de ejemplo, se lanza una petición al servidor asíncrona; a media que vaya cambiando el estado de la petición se irá invocando a la función *alertContents* que solo procesará la respuesta cuando haya finalizado.

Observa que solo se procesa la respuesta en formato texto cuando la propiedad *readyState* del objecto de tipo *XMLHttpRequest* tiene el valor 4, indicando que ha finalizado.

En ese momento se comprueba si todo fue OK o hubo problemas, mediante la propiedad *status* del objecto de tipo *XMLHttpRequest*.

```
http_request.onreadystatechange=alertContents;
http_request.open('GET',url,true);
http_request.send();

function alertContents(){
  console.log(http_request.readyState);
  if (http_request.readyState==4) {
     if (http_request.status==200)
        console.log(http_request.responseText);
     else
        alert('Hubo problemas con la petición.');
  }
}
```

ejemplo

El siguiente código es un ejemplo muy sencillo de una petición completa a una API expresada por lo que contuviera la constante *api_call_xml*.

El resultado de dicha a petición saldrá por la consola del navegador mediante la sentencia *console.log(http_request.responseXML)*

```
const http_request=new XMLHttpRequest();
http_request.onreadystatechange=()=>{
  if (http_request.readyState==4){
    if (http_request.status==200){
        console.log("Respuesta: ");
        console.log(http_request.responseXML);
      }
    }
};

http_request.open('GET',api_call_xml,true);
http_request.send();
```

XMLHttpRequest.responseType
XMLHttpRequest.response

En la actualizad, se usa la propiedad *`response`* de *`XMLHttpRequest`* para obtener la respuesta del servidor cuando se completa la petición.

El formato de la respuesta debe especificarse previamente mediante la propiedad *`responseType`* de *`XMLHttpRequest`*, si no se especifica, por defecto, se esperará una cadena de texto.

Los formatos de respuesta que se pueden especificar son:

- ***`""`*** o ***`"text"`*** por defecto, obtiene una cadena de texto.
- ***`"arraybuffer"`*** especifica un *ArrayBuffer* (no es un array de JS) para datos binarios.
- ***`"document"`*** especifica un documento XML o un documento HTML, en base al tipo MIME del dato recibido.

XMLHttpRequest.responseType
XMLHttpRequest.response

- ***"blob"*** especifica un Blob para datos binarios.
- ***"json"*** especifica un JSON, que será automáticamente analizado.

Ejemplo:

```
const http_request=new XMLHttpRequest();
http_request.onreadystatechange=()=>{
  if (http_request.readyState==4){
    if (http_request.status==200)
      console.log(http_request.response);
  }};

http_request.open('GET',url,true);
http_request.responseType='json';
http_request.send();
```

XMLHttpRequest.setRequestHeader

El objeto *XMLHttpRequest* permite tanto enviar cabeceras personalizadas como leer cabeceras de la respuesta.

Existen 3 métodos para las cabeceras HTTP:

setRequestHeader(name, value) asigna la cabecera de la solicitud con los valores *name* y *value* pasados como argumento. Una vez que una cabecera es asignada no se puede modificar, por lo que llamadas posteriores a este método solo agregan información a la cabecera, no la sobrescriben.

Ejemplo:

```
http_request.setRequestHeader('Content-Type', 'application/json');
```

XMLHttpRequest.setRequestHeader

El siguiente código es el envío, en formato JSON, de los datos contenidos en la variable *pedido* al servidor para que los procese, mediante el código contenido en el archivo *compra.php*.

```
http_request = new XMLHttpRequest();
if (http_request)
{
    const pedido_json = JSON.stringify(pedido);
    http_request.onreadystatechange = ()=>{
        if (http_request.readyState == 4) {
            if (http_request.status == 200)
                console.log("Respuesta del Servidor: ", http_request.responseText);
            else
                console.log("Ha habido un error...");
        }
    };
    http_request.open("POST", "compra.php", true);
    http_request.setRequestHeader("Content-type", "application/json");
    http_request.send(pedido_json);
}
else
    alert('Fallo :( No es posible crear una instancia XMLHTTP');
```

XMLHttpRequest.getResponseHeader

getResponseHeader(name) devuelve la cabecera de la respuesta con el nombre *name* pasado como argumento, excepto *Set-Cookie* y *Set-Cookie2*.

Ejemplo:

```
console.log( http_request.getResponseHeader('Content-Type'));
```

```
application/json; charset=utf-8
```

XMLHttpRequest. getAllResponseHeaders

getAllResponseHeaders() devuelve todas las cabeceras de la respuesta, excepto por *Set-Cookie* y *Set-Cookie2*. Las cabeceras se devuelven como una sola línea de texto.

El salto de línea entre las cabeceras siempre es un *\r\n*, por lo que por estos caracteres se pueden dividir en cabeceras individuales.

El separador entre el nombre y el valor siempre es dos puntos seguido de un espacio "*:* ".

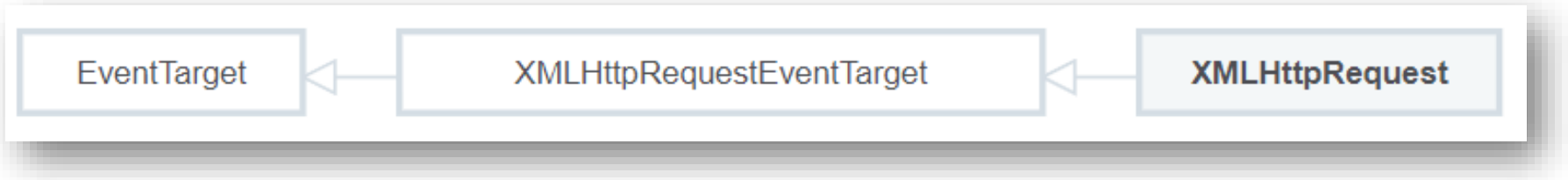
Ejemplo:

```
const cabeceras=http_request.getAllResponseHeaders()
  .split('\r\n')
  .reduce((resultado, procesando) => {
    let [nombre, valor] = procesando.split(': ');
    resultado[nombre] = valor;
    return resultado;
  }, {});
```

XMLHttpRequest.upload

El objeto *XMLHttpRequest* dispone de una propiedad llamada *upload* que contiene un objeto que se puede útil para monitorear el progreso de una carga.

Este objeto también es un *XMLHttpRequestEventTarget*, esto es, una interfaz que describe los manejadores de eventos que puede implementar un objeto *XMLHttpRequest*, por lo que se le puede asociar manejadores del evento (*event listeners*).



Según la especificación los manejadores de eventos deben adjuntarse después de que el método *open* de *XMLHttpRequest* sea llamado; sin embargo, los navegadores no funciona bien si se hace de esta manera, por lo que es altamente recomendable hacerlo justo antes de invocar a *open*.

XMLHttpRequestEventTarget

Los eventos que admite el objeto *XMLHttpRequest* son los siguientes:

abort se activa cuando se cancela una solicitud, por ejemplo, porque el programa llamó al método *XMLHttpRequest.abort()*, este método cancela la solicitud si ya se ha enviado. También es accesible a través de un manejador de eventos sobre la propiedad *XMLHttpRequest.onabort*.

error se activa cuando la solicitud encuentra un error. También es accesible a través de un manejador de eventos sobre la propiedad *XMLHttpRequest.onerror*.

Load se activa cuando una transacción (envío de datos, como puede ser el envío de un fichero) XMLHttpRequest se completa con éxito. También es accesible a través de un manejador de eventos sobre la propiedad *XMLHttpRequest.onload*.

Loadend se activa cuando se completa una solicitud, ya sea con éxito, después de *load*, o sin éxito, después de *abort* o *error*. También es accesible a través de un manejador de eventos sobre la propiedad *XMLHttpRequest.onloadend*.

XMLHttpRequestEventTarget

Loadstart se activa cuando una solicitud ha comenzado a cargar datos. También es accesible a través de un manejador de eventos sobre la propiedad *XMLHttpRequest.onloadstart*.

progress se activa periódicamente cuando una solicitud recibe más datos. También es accesible a través de un manejador de eventos sobre la propiedad *XMLHttpRequest.onprogress*.

La función manejadora del evento recibe un objeto *evento* que tiene dos propiedades *event.loaded*, que indica la cantidad de datos transferidos y *event.total*, que indica la cantidad de datos que serán transferidos.

timeout se activa cuando el progreso finaliza debido a que expira el tiempo preestablecido. También es accesible a través de un manejador de eventos sobre la propiedad *XMLHttpRequest.ontimeout*.
Se puede acceder al tiempo preestablecido mediante la propiedad *XMLHttpRequest.timeout*.

Para conocer todas las propiedades y métodos de *XMLHttpRequestEventTarget* se puede consultar la documentación en:
https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequestEventTarget

XMLHttpRequestEventTarget

Ejemplos:

```
const http_request=new XMLHttpRequest();
http_request.upload.onprogress = e=>{
   const progress = Math.round((e.loaded * 100.0) / e.total);
   console.log("Subiendo: " + progress + "%");
};
```

```
const http_request=new XMLHttpRequest();
http_request.upload.loadstart= e=>{
   console.log("Empezando la transferencia");
};
```

FormData

El objeto *FormData* proporciona una manera sencilla de construir un conjunto de parejas clave-valor que representan los campos de un formulario y sus valores, que pueden ser enviados fácilmente como argumento del método *XMLHttpRequest.send*.

FormData utiliza el mismo formato que usaría un formulario si el tipo de codificación fuera "*multipart/form-data*".

A modo de recordatorio: cuando se realiza una solicitud POST se debe codificar los datos que forman el cuerpo de la solicitud de alguna manera. Los formularios HTML proporcionan tres métodos de codificación:

- ***application/x-www-form-urlencoded*** es el formato por defecto, también pude usarse *multipart/form-data* pero es menos eficiente.
- ***multipart/form-data*** este método de codificación debe usarse si el formulario incluye algún campo *<input type="file">*
- ***text/plain*** fue introducido por HTML 5 y solo es útil para depuración.

FormData

Para cambiar la codificación de los datos del cuerpo de la solicitud del objeto *FormData* se utiliza la etiqueta *enctype* que admite como valores: *multipart/form-data*, *application/x-www-form-urlencoded* y *text/plain*.

Ejemplo desde HTML sería:

```
<form id="formulario" method="post" enctype="multipart/form-data">
  <div>
    <label for="nombre">Nombre:</label>
    <input type="text" id="nombre" name="nombre">
  </div>
  <input type="submit" value="Enviar">
</form>
```

Desde JS no se debe cambiar la cabecera con el método *setRequestHeader* ya que solo se estaría especificando el tipo de datos, pero realmente no se estaría codificando los mismos a ese formato; es el propio objeto *FormData* el que debe codificar adecuadamente los datos mediante el atributo *enctype*:

```
const formData = new FormData();
//formData.enctype="application/x-www-form-urlencoded";
formData.enctype="multipart/form-data"
```

FormData

El constructor del objeto *FormData* puede recibir, con carácter opcional, como argumento un elemento HTML de tipo *<form>*.

Si se utiliza un argumento *<form>*, *FormData* se rellenará con las claves-valores actuales del formulario usando la propiedad de nombre de cada elemento para las claves y su valor para los valores.

Ejemplo:

```
<form id="formulario" name="formulario">
  <div>
    <label for="nombre">Nombre:</label>
    <input type="text" id="nombre" name="nombre">
  </div>
  <input type="submit" value="Enviar">
</form>
```

```
const formData=new FormData(document.querySelector('#formulario'));
```

FormData

El objeto *FormData* dispone de los siguientes métodos:

- ***append(name, value)*** agrega un campo al formulario con el nombre *name* y el valor *value*.
- ***append(name, blob, fileName)*** agrega un campo del tipo *<input type="file">*, el tercer argumento *fileName* establece el nombre del archivo.

El segundo argumento es un objeto *blob* y representa un objeto de tipo fichero de datos crudos inmutables. Para conocer más acerca de los objetos blob se puede consultar la documentación en: https://developer.mozilla.org/en-US/docs/Web/API/Blob

- ***delete(name)*** elimina el campo de nombre *name*.
- ***get(name)*** devuelve el valor del campo con el nombre *name*.
- ***has(name)*** en caso de que exista el campo con el nombre *name*, devuelve *true*, de lo contrario *false*.

FormData

- ***getAll(name)*** devuelve un array con todos los valores asociados a los campos con el nombre *name*.

- ***keys()*** devuelve un iterador que al recorrerlo permite obtener los nombres asociados a los campos del objeto *FormData*.

- ***values()*** devuelve un iterador que al recorrerlo permite obtener los valores asociados al objeto *FormData*.

- ***entries()*** devuelve un iterador que al recorrerlo permite obtener los nombres y sus valores asociados al objeto *FormData*, en cada iteración se obtiene un array de dos elementos asociados al par clave y su valor asociado.

Ejemplo:

```
const formData = new FormData();
formData.append('nombre', 'Jorge');
formData.append('apellido', 'Garcia');

for(let campo of formData.entries())
   console.log(campo[0]+ ', '+ campo[1]);
```

FormData

Ejemplo:

```
const formData = new FormData();

formData.append("username", "jorge");
formData.append("accountnum", 123456);

const content = '<a id="a"><b id="b">hola</b></a>';
const blob = new Blob([content], {type:"text/xml"});
formData.append("archivo", blob);

const http_request=new XMLHttpRequest();
request.open("POST", "http://foo.com/submitform.php");
request.send(formData);
```

Bibliografía y recursos online

- https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Statements/import
- https://262.ecma-international.org/6.0/#sec-imports
- https://programadorwebvalencia.com/cursos/javascript/ajax/
- https://developer.mozilla.org/es/docs/Web/Guide/AJAX
- https://www.w3schools.com/xml/ajax_xmlhttprequest_response.asp
- https://www.w3schools.com/tags/ref_httpmessages.asp
- https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequestEventTarget
- https://developer.mozilla.org/en-US/docs/Web/API/Blob