



# *JavaScript*

## Sesión 5 - JavaScript y el Navegador



## Índice

- AJAX
- JSON
- HTML 5
- Almacenando información
  - LocalStorage
- Web Workers
- WebSockets
- Rendimiento



## 5.1 AJAX

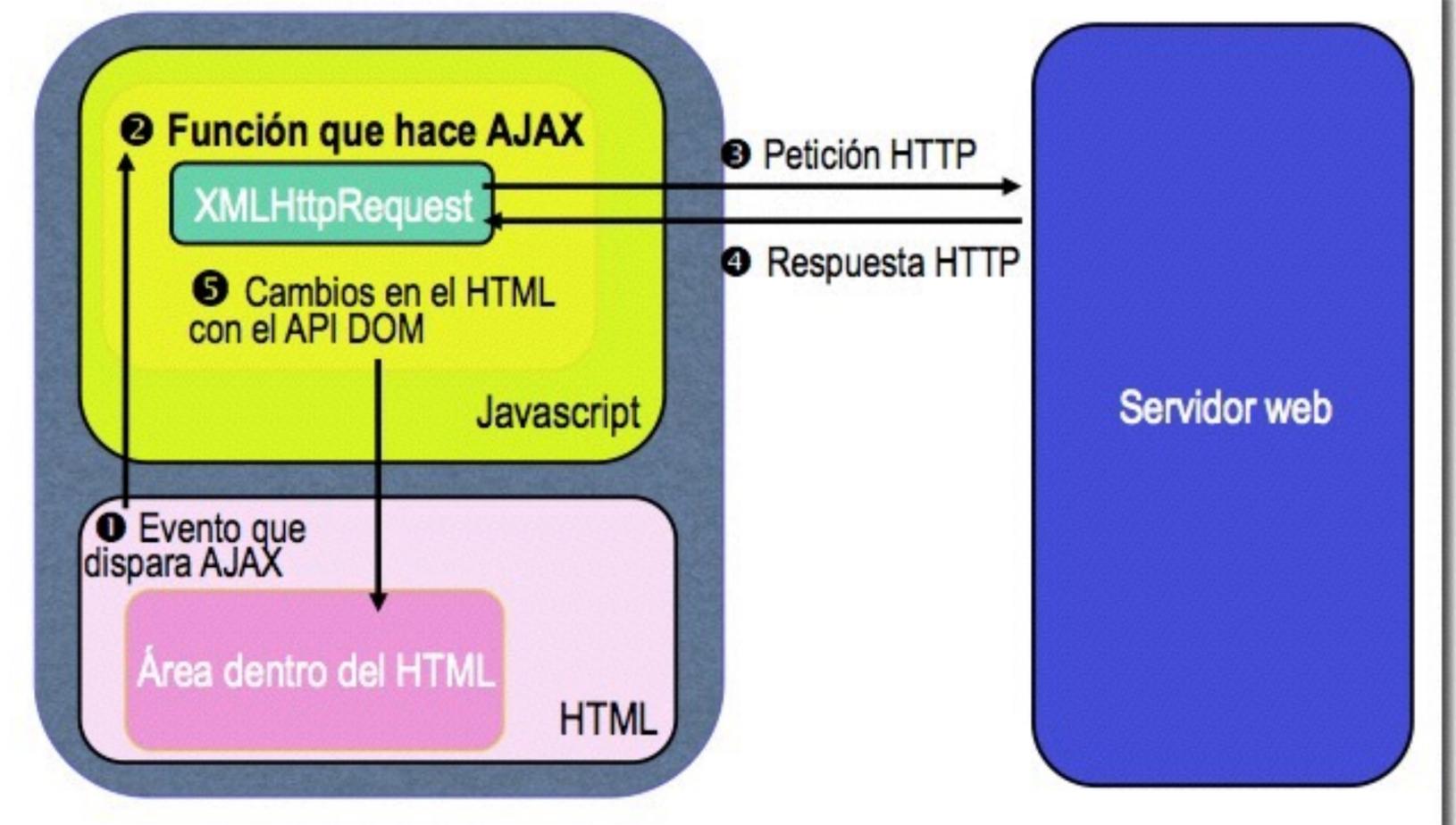
- *Asynchronous JavaScript And XML*
- Técnica (que no lenguaje) que permite realizar peticiones *HTTP* al servidor desde *JavaScript*, y recibir la respuesta sin recargar la página ni cambiar a otra página distinta.
- La información se inserta en la página mediante el uso del API DOM.
- Modo **sandbox** → no se permite realizar peticiones a dominios ajenos
- Objeto `XMLHttpRequest`
  1. lanzar una petición HTTP al servidor mediante `open (getPost, recurso, esAsync)`
  2. enviar la petición mediante `send ()`
  3. recibir la respuesta en la propiedad `responseText`.



## AJAX Síncrono

- Al realizar una petición **síncrona**, AJAX bloquea el navegador y se queda a la espera de la respuesta

```
var xhr = new XMLHttpRequest();  
xhr.open("GET", "fichero.txt", false);  
xhr.send(null);  
alert(xhr.responseText);
```

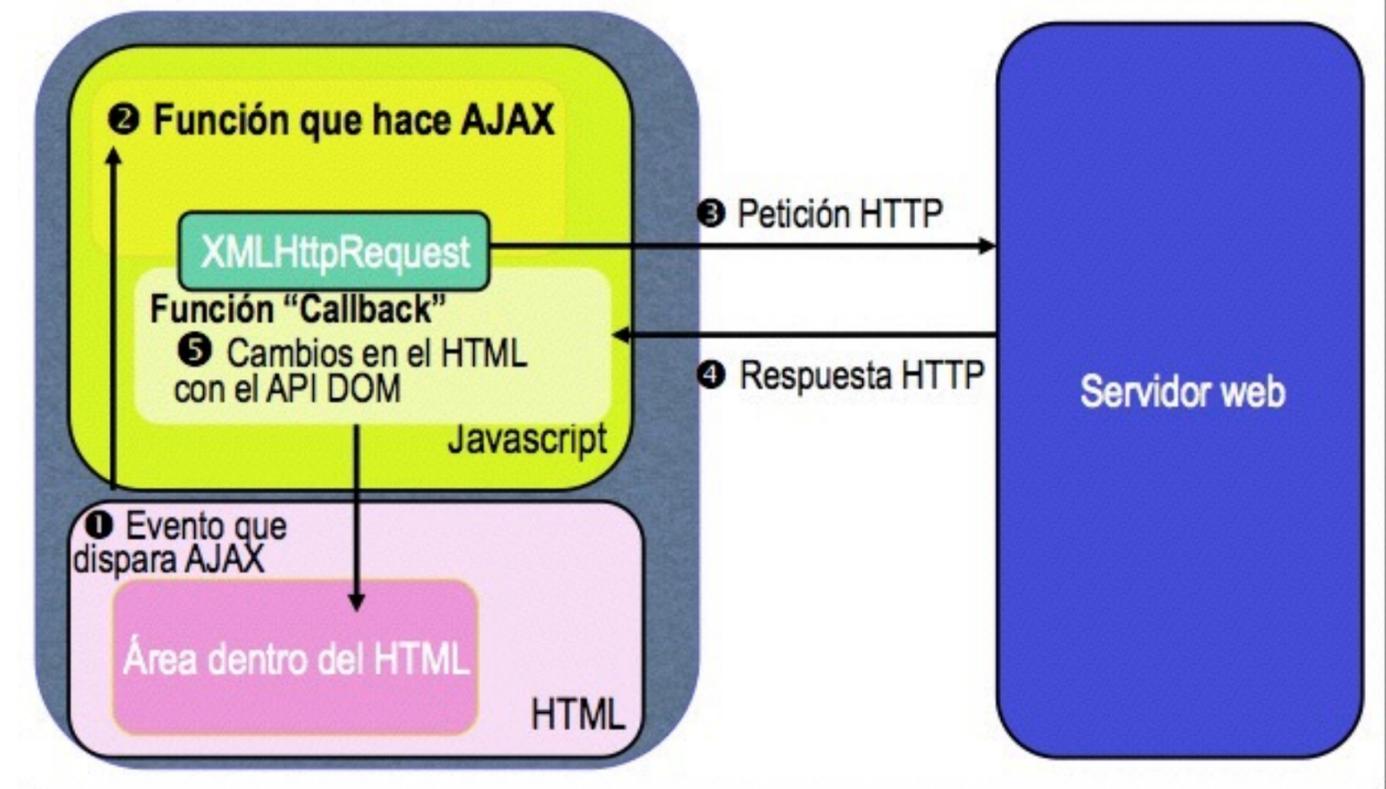




## AJAX Asíncrono

- Tras realizar la petición, el control vuelve al navegador inmediatamente.
- ¿Cuándo está disponible el recurso?
- propiedad `readyState`
- *callback* que ofrece la propiedad `onreadystatechange`.
- Tras abrir conexión, le asignamos el *callback* y después enviamos la petición.

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "fichero.txt", true);
xhr.onreadystatechange = function() {
    if (xhr.readyState === 4) {
        alert(xhr.responseText);
    }
};
xhr.send(null);
```





## readystate y status

- `readystate` → estado de la petición
  - 1: la petición ha fallado
  - 2: sólo ha cargado las cabeceras HTTP
  - 3: esta cargándose
  - 4: petición completa
- `status` → estado de la conexión - código HTTP
  - 200: OK
  - 304: No ha cambiado desde la última petición
  - 404: No encontrado

```
xhr.onreadystatechange = function() {  
    if (xhr.readyState === 4) {  
        var status = xhr.status;  
        if ((status >= 200 && status < 300) || (status === 304)) {  
            alert(xhr.responseText);  
        } else {  
            alert("Houston, tenemos un problema");  
        }  
    }  
};
```



## Enviando datos con POST

- Indicar el método de envío en `open()`
- Tipo de contenido del formulario
- Datos en método `send()`

```
var xhr = new XMLHttpRequest();
xhr.open("POST", "fichero.txt", true);
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
xhr.onreadystatechange = function() {
    // código del manejador
};
xhr.send("heroe=Batman");
```



## 5.2 JSON

- Formato de texto que almacena datos reconocibles como objetos por JavaScript
- Sintaxis de objeto literal

```
{  
  "nombre": "Batman",  
  "email": "batman@heroes.com",  
  "gadgets": ["batmovil", "batarang"],  
  "amigos": [  
    { "nombre": "Robin", "email": "robin@heroes.com"},  
    { "nombre": "Cat Woman", "email": "catwoman@heroes.com"}  
  ]  
}
```



## Objeto JSON

- `JSON.stringify` (`objeto` [, `filtro`] [, `formato`]) → representación *JSON* de un objeto como una cadena
  - serializa el objeto, omitiendo todas las funciones, las propiedades con valores `undefined` y propiedades del prototipo.
- `JSON.parse` (`cadena`) → transforma una cadena *JSON* en un objeto *JavaScript*.

```
var batman = { "nombre": "Batman", "email": "batman@heroes.com" };  
var batmanTexto = JSON.stringify(batman);  
var batmanObjeto = JSON.parse(batmanTexto);
```



## AJAX y JSON

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "heroes.json", true);
xhr.onreadystatechange = function() {
    if (xhr.readyState === 4) {
        var respuesta = JSON.parse(xhr.responseText);
        alert(respuesta.nombre);
    }
};
xhr.send(null);
```

```
var xhr = new XMLHttpRequest();
xhr.open("POST", "fichero.txt", true);
xhr.setRequestHeader("Content-Type", "application/json");
xhr.onreadystatechange = function() {
    // código del manejador
};
xhr.send(JSON.stringify(objeto));
```



## Filtrando campos

- 2º parámetro de `JSON.stringify()`
  - un array con los campos que se incluirán al serializar
  - una función que recibe una clave y un valor, y permite modificar el comportamiento de la operación.

Si para un campo devuelve `undefined` dicho campo no se serializará.

```
var heroe = {
  nombre: "Batman",
  email: "batman@heroes.com",
  gadgets: ["batmovil", "batarang"],
  amigos: [
    { nombre: "Robin", email: "robin@heroes.com" },
    { nombre: "Cat Woman", email: "catwoman@heroes.com" }
  ]
};
```



## Ejemplo Filtrando Campos

```
var nomEmail = JSON.stringify(heroe, ["nombre", "email"]);
var joker = JSON.stringify(heroe, function (clave, valor) {
  switch (clave) {
    case "nombre":
      return "Joker";
    case "email":
      return "joker_" + valor;
    case "gadgets":
      return valor.join(" y ");
    default:
      return valor;
  }
});

console.log(nomEmail); // {"nombre": "Batman", "email": "batman@heroes.com"}
console.log(joker); //
{"nombre": "Joker", "email": "joker_batman@heroes.com", "gadgets": "batmovil y
batarang", "amigos": [{"nombre": "Joker", "email": "joker_robin@heroes.com"},
{"nombre": "Joker", "email": "joker_catwoman@heroes.com"}]}
```



## Indentando el resultado

- Si queremos que el resultado de `stringify` aparezca formateado → tercer parámetro
  - número (entre 1 y 10) → cantidad de espacios utilizados como sangría
  - carácter → utilizado como separador

```
console.log(JSON.stringify(heroe, [ "nombre", "email" ]));  
console.log(JSON.stringify(heroe, [ "nombre", "email" ], 4));  
console.log(JSON.stringify(heroe, [ "nombre", "email" ], "<->" ));
```

```
<top frame>  
{ "nombre": "Batman", "email": "batman@heroes.com" }  
{  
  "nombre": "Batman",  
  "email": "batman@heroes.com"  
}  
<-> "nombre": "Batman",  
<-> "email": "batman@heroes.com"  
>
```



## QUnit y AJAX

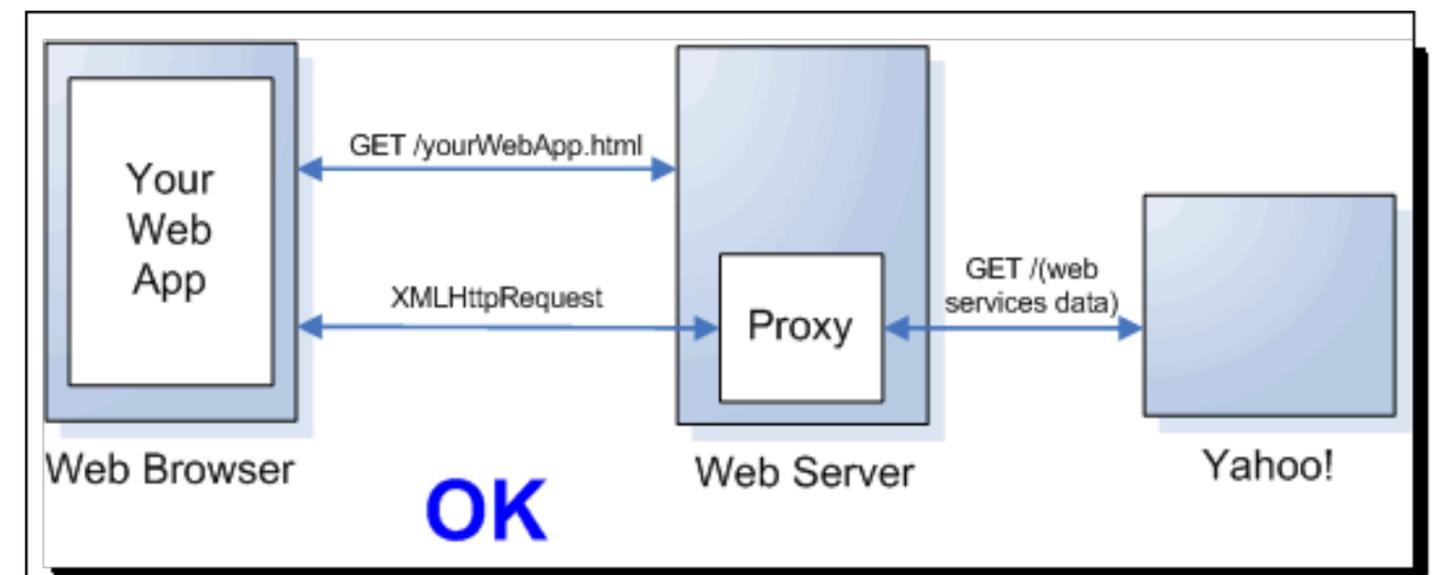
```
QUnit.asyncTest('ajaxHeroes', function (assert) {
    expect(2);

    var xhr = new XMLHttpRequest();
    xhr.open("GET", "heroes.json", true);
    xhr.onreadystatechange = function() {
        if (xhr.readyState === 4) {
            var respuesta = JSON.parse(xhr.responseText);
            assert.equal(respuesta.nombre, "Batman", "Atributo de archivo json");
            assert.equal(respuesta.amigos[0].nombre, "Robin", "Atributo de un hijo que es
array");
            QUnit.start();
        }
    };
    xhr.send(null);
});
```



## Cross-Domain

- Navegador funciona como *sandbox* → restringe las peticiones AJAX
- *Same-Origin Policy* → las peticiones tienen que compartir protocolo, servidor y puerto
  - desde un dominio A se puede realizar una petición a un dominio B, pero no se puede obtener ninguna información de la petición hacia B, ni la respuesta ni siquiera el código de respuesta
- Soluciones *Cross-Domain AJAX*:
  - **Proxy**: crear un servicio en nuestro servidor que *retransmita* la petición al host al que queremos llegar.
  - **CORS**
  - **JSONP**





## CORS (*Cross-Origin Resource Sharing*)

- Si el servidor al que le haces la petición la permite (enviando la cabecera `Access-Control-Allow-Origin`), el navegador también dejará que se realice.
- El servidor tiene que especificar que permite peticiones de otros dominios.
- Pocos dominios permiten su uso (<http://bit.ly> o <http://twitpic.com>).
- Si queremos evitarnos enviar la cabecera, podemos hacer uso de <http://www.corsproxy.com>, el cual fusiona tanto la técnica CORS como el uso de un proxy.



## JSONP (*JSON with Padding*)

- Las restricciones de seguridad no se aplican a la etiqueta `<script>`.
  - Podemos cargar (¡y ejecutar!) *JavaScript* de cualquier origen mediante una petición **GET**.
  - Podríamos cargar en un punto del documento la respuesta del servidor en formato *JSON*, pero no hace nada con esa información

```
<script src="http://api.openbeerdatabase.com/v1/beers.json"></script>
```

- **JSONP** permite definir una función (que es nuestra) que recibirá como parámetro el JSON que responde el servidor.
  - Los servicios que admiten JSONP reciben un parámetro en la petición, (normalmente nombrado *callback*) que especifica el nombre de la función a llamar.

petición `http://api.openbeerdatabase.com/v1/beers.json?callback=miFuncion`

respuesta `miFuncion(respuestaEnJSON);`



## JSONP Dinámico

- Para que la petición se realice al producirse un evento, hemos de crear la etiqueta `<script>` de manera dinámica

<http://jsbin.com/riruc/1/edit?html,js,output>

```
<body>
<script>
  function llamarServicio() {
    var s = document.createElement("script");
    s.src = "http://api.openbeerdatabase.com/v1/beers.json?callback=miFuncion";
    document.body.appendChild(s);
  }
  function miFuncion(json) {
    document.getElementById("resultado").innerHTML = JSON.stringify(json);
  }
</script>
<input type="button" onclick="llamarServicio()" value="JSONP">
<div id="resultado"></div>
</body>
```



## 5.3 HTML 5

- Soporte de vídeo, mediante `<video>` (operaciones `.play()`, `.pause()`, `currentTime = 0;` evento `ended`, `play`, `pause`, ...)
- Soporte de audio
- Elemento `canvas` para la generación de gráficos en 2D, similar a SVG.
- Almacenamiento local y/o offline, mediante el objeto `localStorage`.
- Nuevos elementos de formulario, como tipos `email`, `date`, `number`, ...
- Arrastrar y soltar (*Drag-and-drop*)
- Geolocalización



<http://caniuse.com/>

- *JavaScript* → `getElementsByClassName (claseCSS)`

```
var c1 = document.getElementsByClassName("clase1")
var c12 = document.getElementsByClassName("clase1 clase2");
```



## Detección de Características

- Para detectar una característica, hay que comprobarla:

```
if (document.getElementsByClassName) {  
    // existe, por lo que el navegador lo soporta  
} else {  
    // no existe, con lo que el navegador no lo soporta  
}
```



- **Modernizr** (<http://www.modernizr.com>) → librería de detección de características
- Objeto `Modernizr` que contiene propiedades con las prestaciones soportadas
  - `video`, `localStorage`, etc...

```
if (Modernizr.video) {  
    // usamos el vídeo de HTML5  
} else {  
    // usamos el vídeo Flash  
}
```



## Polyfills

- Si nuestro navegador no soporta la característica deseada, podemos usar *HTML shims* o *polyfills* que reproducen la funcionalidad del navegador.
- **Shim**: librería que ofrece una nueva API a un entorno antiguo mediante los medios que ofrece el entorno.
- **Polyfill**: fragmento de código (o *plugin*) que ofrece la tecnología que esperamos que el navegador ofrezca de manera nativa.
  - Un *polyfill* es un *shim* para el API del navegador.
- Tras comprobar mediante *Modernizr* si el navegador no soporta un *API*, cargar un *polyfill*
- <https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills>
- <http://html5please.com/>
- <http://caniuse.com>



## 5.4 Almacenando Información. Cookies

- Tanto el servidor con *Java* (o cualquier otro lenguaje) como el navegador mediante *JavaScript* pueden acceder a la información almacenada.
  - Sólo pueden almacenar hasta 4KB
  - Se envían y vuelven a recibir con cada petición

- `document.cookie`

```
document.cookie = "nombre=Batman";  
var info = document.cookie;  
document.cookie = "amigo=Robin";
```

- Caducan, normalmente al cerrar el navegador. Si no queremos que caduquen, indicar la fecha de expiración con atributo `expires`

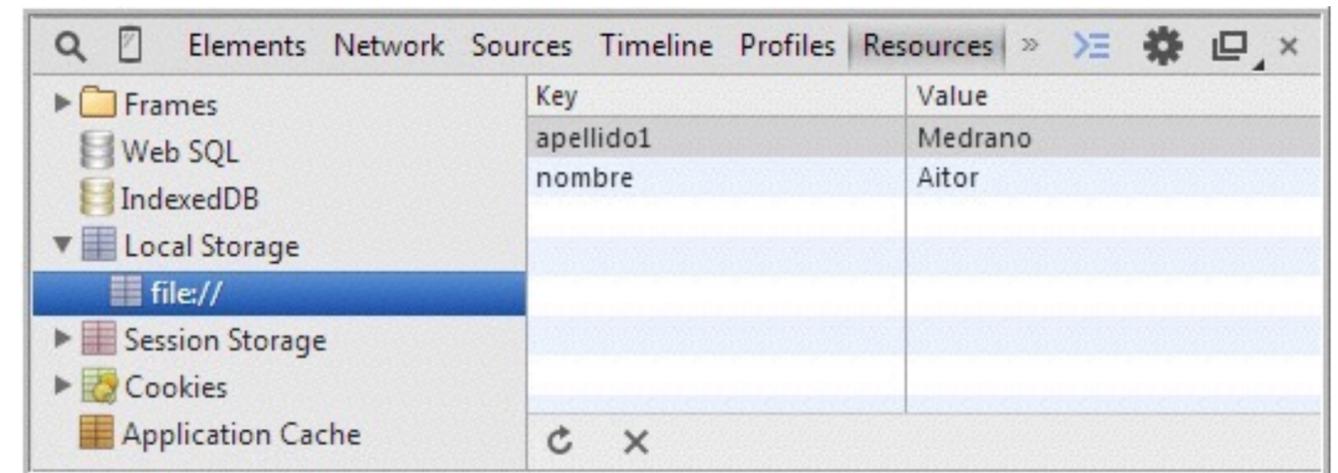
```
document.cookie = "nombre=Batman; expires=Thu Dec 31 2015 00:00:00 GMT+0100 (CET)";
```



## LocalStorage

- Almacén en el navegador mediante un mapa de claves/valor donde podemos almacenar todo el contenido que deseemos.
  - Puede almacenar entre 5 y 10 MB
  - La información no se envía con cada petición
  - No caduca
- Objeto **localStorage**
  - Acceso y modificación mediante propiedades u operaciones `setItem(clave)` y `getItem(clave)`

```
localStorage.nombre = "Aitor";  
localStorage.setItem("apellido1", "Medrano");  
  
console.log(localStorage.nombre);  
console.log(localStorage.getItem("apellido1"));
```





## Operaciones

- Propiedad `length` → tamaño

```
console.log(localStorage.length); // 2
```

- `removeItem(clave)` → eliminar una propiedad

```
localStorage.removeItem(apellido1);  
console.log(localStorage.getItem("apellido1")); // undefined
```

- `clear()` → vacía el almacén

```
localStorage.clear();  
console.log(localStorage.getItem("nombre")); // undefined
```



## Serialización

- Todos los datos se almacenan como cadenas.
- Antes de almacenar cualquier elemento, se ejecuta el método `toString()`

```
localStorage.edad = 35;
console.log(typeof localStorage.getItem("edad")); // "string"
var edad = parseInt(localStorage.getItem("edad"), 10);
```

```
var persona = {
  nombre : "Aitor",
  apellido1 : "Medrano",
  edad : 35
};

localStorage.setItem("persona", persona);
console.log(localStorage.getItem("persona")); // [object Object]
```



## Trabajando con Objetos

- Serializar mediante `JSON.stringify(objeto)`
- Deserializar mediante `JSON.parse(objetoLS)`

```
var persona = {
  nombre : "Aitor",
  apellido1 : "Medrano",
  edad : 35
};

localStorage.setItem("persona", JSON.stringify(persona));
console.log(localStorage.getItem("persona")); // "{\\"nombre\\":\\"Aitor\\", \\"apellido1\\":
\\"Medrano\\", \\"edad\\":35}"
var personaRecuperada = JSON.parse(localStorage.getItem("persona"));
console.log(personaRecuperada); /* [object Object] {
  apellido1: "Medrano",
  edad: 35,
  nombre: "Aitor"
} */
```



## SessionStorage

- `sessionStorage` → objeto que permite almacenar la información con un ciclo de vida asociado a la sesión del navegador
- Al cerrar el navegador, el almacenamiento se vacía.
  - Si se produzca un cierre inesperado por fallo del navegador, los datos se restablecen como si no hubiésemos cerrado la sesión.
- Ambos objetos heredan del interfaz `Storage` por lo que el API es similar.



## IndexedDB

- Estructura más compleja que `localStorage`, similar a una BBDD.
- **WebSQL DB**: wrapper sobre SQLite para interactuar con los datos mediante un interfaz SQL
  - Permite ejecutar sentencias `select`, `insert`, `update` o `delete`.
  - No es un estándar, y sólo hay una implementación, la especificación se ha congelado y ha pasado a un estado de *deprecated*.
  - En la actualidad la soportan la mayor parte de los navegadores excepto *Firefox* y *IE* (<http://caniuse.com/#feat=sql-storage>).
- **IndexedDB**: expone un API como un almacén de objetos
  - Comparte muchos conceptos con una BBDD relacional, tales como base de datos, registro, campo o transacción.
  - No se interactúa mediante el lenguaje *SQL*, sino que se utilizan métodos del mismo modo que se hace mediante *JPA* o *Hibernate*.
  - Soporte en *Firefox*, *Chrome* y *Opera*, y parcial en el resto <http://caniuse.com/#search=IndexedDB>



## 5.5 Web Workers

- *JavaScript* es mono-hilo
- Para hacer tareas en paralelo → AJAX y `setTimeout/setInterval`
- HTML 5 introduce **Web Workers**
  - 2 tipos: dedicados y compartidos (shared)
  - Son hilos de ejecución que se ejecutan en *background* que corren al mismo tiempo (más o menos)
  - Aprovechan las arquitecturas multi-núcleo que ofrece el hardware.
  - El *web worker* se ejecuta en *background* mientras que el hilo principal procesa los eventos del interfaz de usuario.
  - Ejemplo: un *worker* puede procesar una estructura JSON para extraer la información útil a mostrar en el interfaz de usuario.



## Hilo Padre (Dedicado)

- El código que pertenece a un web *worker* reside en un archivo *JavaScript* aparte.
- El hilo padre crea el nuevo *worker* indicando la URI del script en el constructor de *Worker*, el cual se carga de manera asíncrona y lo ejecuta.

```
var worker = new Worker("otroFichero.js");
```

- Para lanzar el worker, el hilo padre le envía un mensaje al hijo mediante el método `postMessage(mensaje)` :

```
worker.postMessage("contenido");
```



## Paso de parámetros

- Se pueden enviar datos de tipos primitivos (cadena, número, booleano, `null` o `undefined`), y estructuras JSON mediante arrays y objetos.
  - No se puede enviar funciones ya que pueden contener referencias al DOM.
- Los hilos del padre y de los workers tienen su propio espacio en memoria
- Los mensajes enviados desde y hacia se pasan por copia, no por referencia → se realiza un proceso de serialización y deserialización de la información.
  - Se desaconseja enviar grandes cantidades de información al worker.



## Callback padre

- El padre registra un *callback* para recibir información del *worker*
- Queda a la espera de recibir un mensaje una vez que el *worker* haya finalizado su trabajo.

```
worker.onmessage = function(evt) {  
    console.log("El worker me ha contestado!");  
    console.log("Y me ha enviado " + evt.data);  
};
```

- El *callback* recibe como parámetro un objeto `evt`, el cual contiene las propiedades:
  - `target`: identifica al *worker* que envió el mensaje. Uso en entorno con múltiples *workers*
  - `data`: contiene el mensaje enviado por el *worker* de vuelta a su padre

```
var worker = new Worker("otroFichero.js");  
  
worker.onmessage = function(evt) {  
    console.log("El worker me ha contestado!");  
    console.log("Y me ha enviado " + evt.data);  
};  
  
worker.postMessage("contenido");
```



## Worker Hijo

- El worker se coloca en un archivo `.js` aparte
- `self` referencia al alcance global (similar a `this`)
- Recibe los mensajes mediante el evento `message` (o propiedad `onmessage`)
- Propiedad `data` para extraer la información.
- Para volver a enviar un mensaje al padre también utiliza el método `postMessage`.

```
self.addEventListener('message', function(evt) {  
    var mensaje = evt.data;  
  
    self.postMessage("Hola " + mensaje);  
});
```



## Rendimiento y Finalización

- Consumen muchos recursos, ya que son hilos a nivel de sistema operativo.
- No conviene crear un gran número de hilos → terminar el web worker una vez ha finalizado su trabajo.
- Gran coste de rendimiento al arrancar y consumen mucha memoria por cada instancia.
- Para que un *worker* termine por si mismo y deseche cualquier tarea pendiente → `close()`

```
self.close();
```

- O si un padre quiere finalizar a un *worker* hijo → `terminate()`

```
worker.terminate();
```



## Seguridad y Restricciones

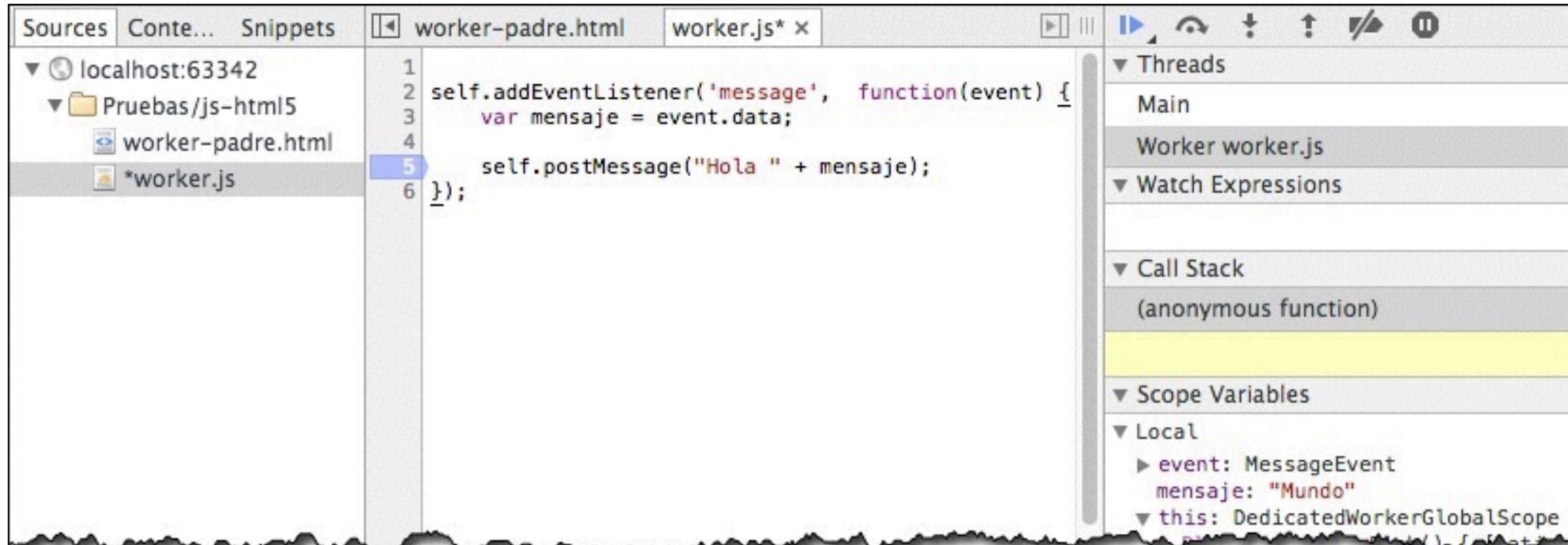
- Dentro de un worker no tenemos acceso a muchos objetos *JavaScript* como `document`, `window` (variables globales), `console`, `parent` ni al *DOM*.
  - ¿Qué podría pasar si múltiples hilos intentaran modificar el mismo elemento?
- Uso: procesar datos y devolver el resultado al padre, el cual sí que puede modificar el DOM.
- Restricciones:
  - Uso reducido de objetos que pueden usar: `XMLHttpRequest` y `localStorage`
  - Acceso a algunas funciones como `setTimeout()`/`clearTimeout()`, `setInterval()`/`clearInterval()`, `navigator`, etc...
  - Se pueden importar otros workers mediante `importScripts`.

```
importScripts("script1.js", "script2.js");
```
  - Siguen la política de *Same-Origin Policy*.
    - un script en <http://www.ejemplo.com> no puede acceder a otro en <https://www.ejemplo.com>.



## Gestión de Errores

- Las *Dev-Tools* permiten depurar el código del worker como si se tratase de cualquier otro código JavaScript.





## error y ErrorEvent

- El padre puede asignar un manejador de eventos sobre el evento `error`
  - Propaga un objeto `ErrorEvent`
    - Propiedades:
      - `filename`: nombre del script que provocó el error.
      - `lineno`: línea del error.
      - `message`: descripción del error.

```
worker.addEventListener('error', function(error) {
  console.log('Error provocado por el worker: ' + error.filename
    + ' en la línea: ' + error.lineno
    + ' con el mensaje: ' + error.message);
});
```



## Casos de Uso

- Trabajar con una librería de terceros con un API síncrona que provoca que el hilo principal tenga que esperar al resultado antes de continuar con la siguiente sentencia.
  - Delegar la tarea a un nuevo worker para beneficiarnos de la capacidad asíncrona.
- Situaciones de chequeo (*polling*) continuo a un destino en *background* y envío de mensaje al hilo principal cuando llega algún dato.
  - Si necesitamos procesar gran cantidad de información devuelta por el servidor, es mejor crear varios *workers* para procesar los datos en porciones que no se solapen.
- Analizar fuentes de vídeo o audio con la ayuda de múltiples web workers, cada uno trabajando en una parte predefinida del problema.



## Shared Worker - Padre

- En los workers dedicados existe una relación directa entre el creador del worker y el propio *worker* mediante una relación 1:1.
- Un *shared worker* puede compartirse entre todas las páginas de un mismo origen.
  - Todas las páginas o scripts de un mismo dominio pueden comunicarse con un único shared worker
  - Permite mantener el mismo estado en diferentes pestañas de la misma aplicación
- Soporte parcial: Sólo en *Firefox, Chrome, Opera* (<http://caniuse.com/#feat=sharedworkers>)
- Se asocian a un puerto mediante la propiedad `port`, para mantener el contacto con el script padre que accede a ellos.

```
var sharedWorker = new SharedWorker('otroScript.js');
sharedWorker.port.onmessage = function(evt) {
    // recibimos los datos del hijo compartido
}
sharedWorker.port.start();
sharedWorker.port.postMessage('datos que queremos enviar');
```



## Shared Worker - Hijo

- Un worker puede escuchar el evento `connect`, el cual se lanza cuando un nuevo cliente intenta conectarse al worker.
- Una vez conectado, sobre el puerto activo, añadiremos un manejador sobre el evento `message`, dentro del cual normalmente contestaremos al hilo padre.

```
self.onconnect = function(evento) {  
    var clientePuerto = event.source;  
  
    clientePuerto.onmessage = function(evento) {  
        var datos = evento.data; // mensaje recibido  
        // ... procesamos los datos  
        clientePuerto.postMessage('datos de respuesta');  
    }  
  
    clientePuerto.start();  
};
```



## 5.6 WebSockets

- Permiten comunicar un cliente con el servidor sin necesidad de peticiones AJAX.
- Técnica de comunicación bidireccional sobre un *socket* TCP, un tipo de tecnología PUSH.
- Casos de uso:
  - Actualización en tiempo real en las aplicaciones de redes sociales
  - Juegos multi-jugador mediante HTML5
  - Aplicaciones de chat online
- Objeto `WebSocket`
- Su constructor recibe un parámetro con la URL que vamos a conectar y otro opcional con información sobre los protocolos a emplear
  - Automáticamente abrirá un conexión temporal al servidor.

```
var ejemploWS = new WebSocket("ws://www.ejemplo.com/servidorSockets", "json");
```



## Estado, Envío y Cierre

- Una vez creada la conexión, podemos consultar su estado → propiedad `status`
  - `0`: conectando, `1`: abierto, `2`: cerrado
- Tras conectar, podemos enviar datos al servidor → `send()`

```
ejemploWS.send("Texto para el servidor");
```

- Los datos que podemos enviar son del tipo `String` (en UTF8), `Blob` o `ArrayBuffer`
- Para finalizar la conexión → `close()`

```
ejemploWS.close();
```
- Antes de cerrar la conexión puede ser necesario comprobar el valor del atributo `bufferedAmount` para verificar que se ha transmitido toda la información.



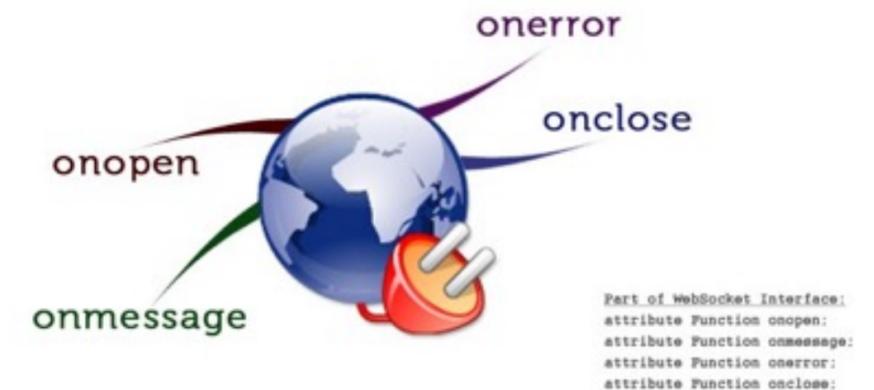
## Eventos

- Al tratarse de una conexión asíncrona, no hay garantía de poder llamar al método `send()` inmediatamente después de haber creado el objeto `WebSocket`.

```
ejemploWS.onopen = function (evt) {  
    ejemploWS.send("Texto para el servidor");  
};
```

```
ejemploWS.onmessage = function (evt) {  
    console.log(evt.data);  
};
```

```
ejemploWS.onerror = function (evt) {  
    console.log("Error:" + evt);  
};
```





## Ejemplo Chat (*Componentes Web*)

```
var websocket;

function connect(nick, sala) {
  if (sala == undefined) {
    websocket = new WebSocket("ws://localhost:8080/cweb-chat/ChatWS");
  } else {
    websocket = new WebSocket("ws://localhost:8080/cweb-chat/ChatWS/" + sala + "?nick=" + nick);
  }
  websocket.onmessage = onMessage;
  websocket.onopen = onOpen;
}

function onOpen(event) {
  document.getElementById("chat").style.display = 'block';
  document.getElementById("login").style.display = 'none';
}

function onMessage(event) {
  items = event.data.split(";");
  document.getElementById("mensajes").innerHTML = "<p><strong>&lt;" + items[0] + "&gt;</strong>" + items[1] + "</p>" +
  document.getElementById("mensajes").innerHTML;
}

function send(texto) {
  websocket.send(texto);
}

window.addEventListener("load", connect, false);
```

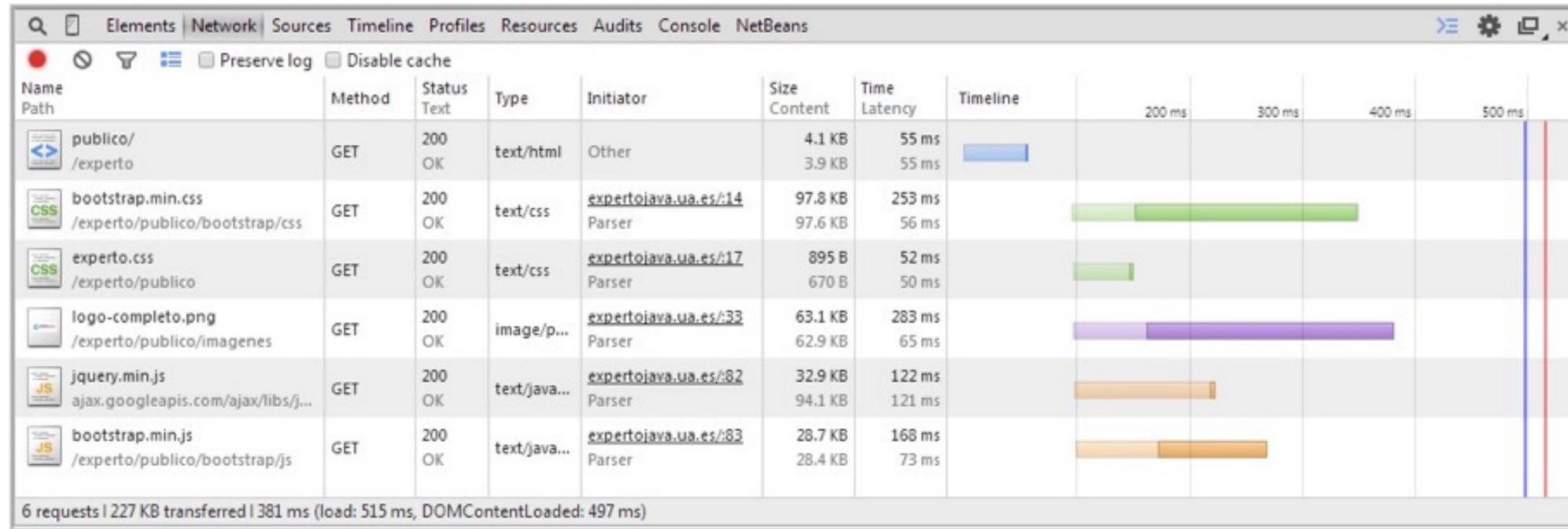


## 5.7 Rendimiento

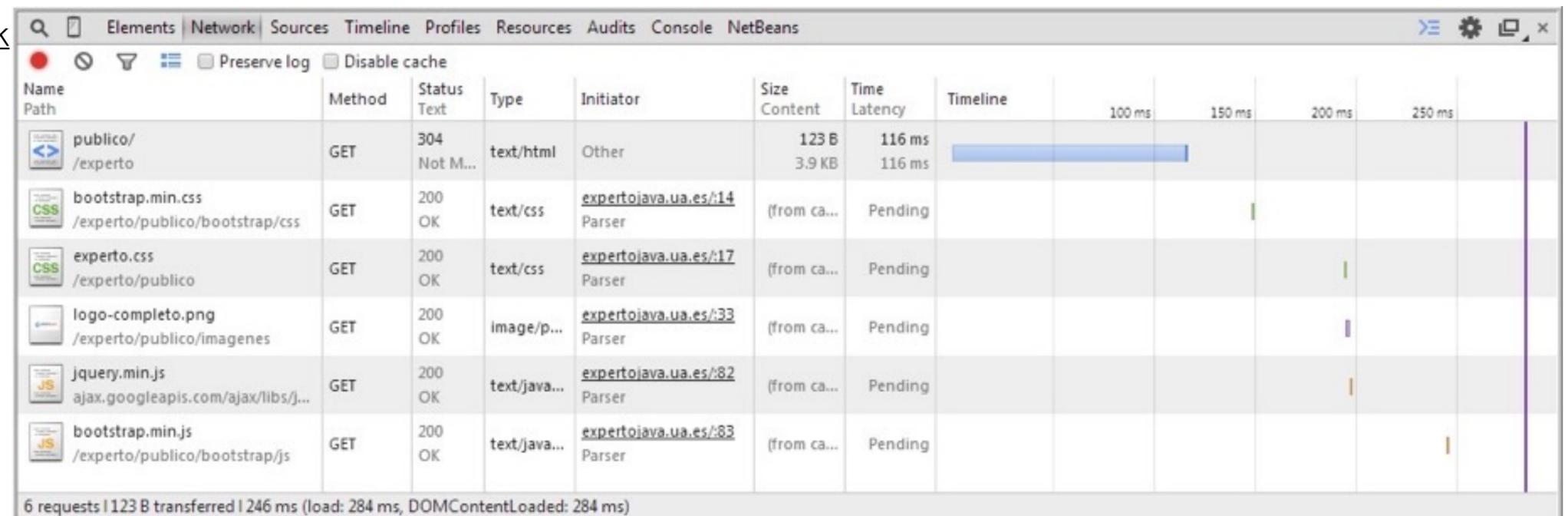
- Identificar los problemas, cuellos de botella, etc..
- Herramientas de **Profiling** → Dev-Tools
  - el botón/pestaña de *Profile*
  - la consola, mediante las funciones `profile()` y `profileEnd()`
  - el código de aplicación, mediante `console.profile()` y `console.profileEnd()`
  - <https://developer.chrome.com/devtools/docs/cpu-profiling>
- **Timers** → `console.time()` y `console.timeEnd()`
- Herramientas de **Análisis de Peticiones** → DevTools → pestaña Network
  - Permite visualizar todos los recursos (`.html`, `.js`, `.css`, imágenes, etc..) que utiliza nuestra página, con su tipo y tamaño, los detalles de las respuestas del servidor, o un timeline con las peticiones de red
  - <https://developer.chrome.com/devtools/docs/network>



## Análisis de Peticiones → <http://expertojava.ua.es/>



<https://developer.chrome.com/devtools/docs/network>





## Reglas I

- **Minificar** el código
  - Eliminar espacios y comentarios para reducir el tamaño
  - *Google Closure Compiler, YUI Compiler*
  - Ofuscar → renombra variables y funciones
- **Reducir el número de archivos**
  - Combinar las librerías en una sola
  - Recomendación **3L**: CDN para librerías famosas + código de aplicación + librerías combinadas en un .js
- Comprobar **calidad del código** mediante validación
  - *JSLint, JSHint*



## Reglas II

- Colocar los enlaces a las **hojas de estilo** antes de los archivos *JavaScript*, para que la página se renderice antes.
- Colocar los enlaces a código **JavaScript** al final de la página, para que primero se cargue todo el contenido *DOM*
- Realizar la carga de la librería *JavaScript* de manera asíncrona incluyendo el atributo **async**:

```
<script src="miLibreriaJS.js" async></script>
```

- Utilizar **funciones con nombre** en vez de anónimas, para que aparezcan en el *Profiler*
- **Aislar** las llamadas **AJAX** y *HTTP* para centrarse en el código *JavaScript*.
  - Conviene crear objetos mock que reduzcan el tráfico de red.
- Dominar la **herramienta Profiler** que usemos en nuestro entorno de trabajo.



## PageSpeed

- <https://developers.google.com/speed/pagespeed/>
- Plugin de Chrome
- Análisis de Rendimiento
- Consejos
- Yahoo → *YSlow*
  - <http://yslow.org/>

Google Developers

Productos > PageSpeed Insights

### PageSpeed Insights g+1 +10m

expertojava.ua.es ANALIZAR

Móvil Ordenador

**70 / 100** Velocidad

**!** Elementos que debes corregir:

- Eliminar el JavaScript que bloquea la visualización y el CSS del contenido de la mitad superior de la página
  - ▶ [Mostrar cómo corregirlo](#)
- Habilitar compresión
  - ▶ [Mostrar cómo corregirlo](#)

Otras webs

- Moodle CV
- Web UA



**¿Preguntas?**