Tema 12. Comunicaciones en HTML5

Contenido

1) Introducción

A la hora de hacer una página Web, es recomendable aprovechar al máximo los recursos que tenemos y en la medida de lo posible, evitar el tráfico innecesario entre el cliente y el servidor, para no cargar demasiado las vías de comunicación, ni ralentizar la carga de nuestra página. Es aquí, donde van a tener especial importancia las comunicaciones que HTML5 nos ofrece, ya que van a permitir agilizar las comunicaciones necesarias para nuestra página Web.

Por un lado trataremos la ***Mensajería Web***, que nos permitirá intercambiar información entre los diferentes documentos de nuestra Web, evitando así muchas conexiones innecesarias con el servidor. Y en segundo lugar, trataremos los ***WebSockets***, mediante los cuales, estableceremos conexiones directas con el servidor para agilizar dicho tráfico.

2) Mensajería Web

La mensajería Web, o la HTML5 Web Messaging, es otra de las novedosas opciones que ofrece el estándar HTML5. En este caso, se trata de una API de JavaScript, que permite intercambiar información entre documentos en el mismo contexto de navegación de una forma segura. De esta manera, el navegador será el encargado de la comunicación entre documentos, y evitamos así muchas conexiones con el servidor.

Hoy en día, el intercambio de información entre documentos del mismo o diferente dominio es muy habitual, y no es un problema fácil de solucionar. Podemos acceder al DOM para encontrar la información relativa a la página y evitar así tráfico de información, pero los navegadores modernos no permiten acceder a él desde cualquier sitio, debido a su política de seguridad. Por lo tanto, en relación a este problema aparecen nuevos mecanismos para ejecutar dicho intercambio de información, y uno de ellos es el Web Messaging que ofrece HTML5.

¡IMPORTANTE!
En el siguiente enlace podéis encontrar la recomendación del W3C sobre Mensajería Web: Web Messaging API

a) La API de Web Messaging

Para realizar el envío y recepción de la información entre los diferentes documentos, contamos con la API de JavaScript. En ella, disponemos del método ***postMessage()***, que será el encargado de realizar el envío de un documento a otro, su funcionamiento es el siguiente:

postMessage(mensaje, dominio, puertos)

- *El primer parámetro*, es el mensaje que va a ser enviado. Éste puede ser una cadena de texto, un objeto, un array...
- *El segundo parámetro*, es el dominio de la ventana de destino. Si no se desea especificar un origen en concreto puede utilizarse un asterisco como comodín, sin embargo esto deja enormes agujeros de seguridad. Por otra parte, si el origen es el mismo que el del emisor puede utilizarse una barra diagonal ("/"), pero no siempre funciona.
- *El tercer parámetro*, es opcional, y definirá los puertos del navegador a utilizar. Deberá de contener un objeto *MessagePort* que veremos más adelante.

A parte de enviar la información, deberemos de recibirla en el destino. Para ello, contamos con el evento ***message*** que se ejecutará en el momento que se reciba un mensaje. Por lo tanto, debemos tener la página de destino escuchando dicho evento, y esperando la recepción de información. Para dicho evento, existen una serie de atributos que son los siguientes:

evento.**data**: Contiene el mensaje enviado por el emisor.

evento.**origin**: Contiene el dominio del documento emisor.

evento.**lastEventId**: Contiene el identificador del actual evento de mensaje.

evento.**source**: Contiene una referencia al objeto *window* del emisor.

evento.**ports**: Es un array que contiene los puertos a utilizar enviados por el emisor.

b) Ejemplo con Web Messaging

Vamos a ver ahora mediante este ejemplo, el funcionamiento del intercambio de información entre documentos. Vamos a disponer de dos páginas, *EjemploM1.html* y *EjemploM2.html*, entre las cuales haremos el intercambio de mensajes. En la primera de ellas, disponemos de un *iFrame*, sobre el cual tenemos cargada la segunda página. Los mensajes serán enviados desde la primera hasta la segunda, y esta segunda responderá sobre la primera.

Vamos por lo tanto con la primera de ellas, el código HTML será el siguiente:

EjemploM1.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo Mensajeria</title>

    <style>
        #info{
            width:350px;
            border: 2px solid blue;
            text-align:center;
            padding-bottom:5px;
        }
    </style>
</head>

<body>

    <iframe id="contenedor" src="EjemploM2.html" height="300" width="350"></iframe>

    <div id="info">

    <p>Mensaje para enviar: <input id="mensaje" type="text">
    <input id="btnEnvio" type="button" value="Enviar"></p>

    <span id="estado"></span>

    </div>

</body>
</html>
```

Hemos creado un elemento *iFrame*, sobre el cual cargamos nuestra segunda página *EjemploM2.html*. Por otro lado, tenemos un elemento *DIV* que contiene, un elemento de entrada, donde escribiremos el mensaje que queramos enviar, y junto a él, un botón, que será el encargado de realizar el envío. También disponemos de un elemento *SPAN*, que será donde vayamos escribiendo la respuesta que nos envíe nuestro segundo documento. El formato, se lo hemos aplicado utilizando la etiqueta *STYLE* como vemos.

Una vez tenemos la estructura de nuestra página, vamos a ver las funciones de JavaScript que harán uso de la API para el intercambio de información entre documentos. En primer lugar, cuando termina de cargar la página, comprobaremos si nuestro navegador acepta el uso de *Web Messaging*. En caso de que no lo acepte, lanzaremos una alerta comunicándolo, pero en caso de que lo acepte, aparte de la alerta asignaremos los eventos necesarios a los elementos correspondientes. El código sería el siguiente:

```
18      window.onload = function ()
19      {
20          if (window.postMessage)
21          {
22              alert("El navegador soporta WebMessaging");
23              document.getElementById("btnEnvio").onclick = enviar;
24              window.addEventListener("message",recibir,false);
25          }
26          else
27          {
28              alert("Lo sentimos, el navegador no soporta WebMessaging");
29          }
30      }
```

Vemos que si el navegador soporta el uso de *Web Messaging*, aparte de la alerta, definimos que cuando hagamos clic sobre el botón de envío, se ejecute la función *enviar()*. Y por otro lado, ponemos la página escuchando el evento *message*, para que cuando reciba algún mensaje ejecute la función *recibir()*.

Por tanto, este será el código de la función *enviar()* que se ejecutará cuando pulsemos el botón de envío:

```
32      function enviar()
33      {
34          var mnsg = document.getElementById("mensaje").value;
35          var protocol = window.location.protocol;
36          var host = window.location.host;
37          var org = protocol+"//"+host;
38          document.getElementById("contenedor").contentWindow.postMessage(mnsg, org);
39      }
```

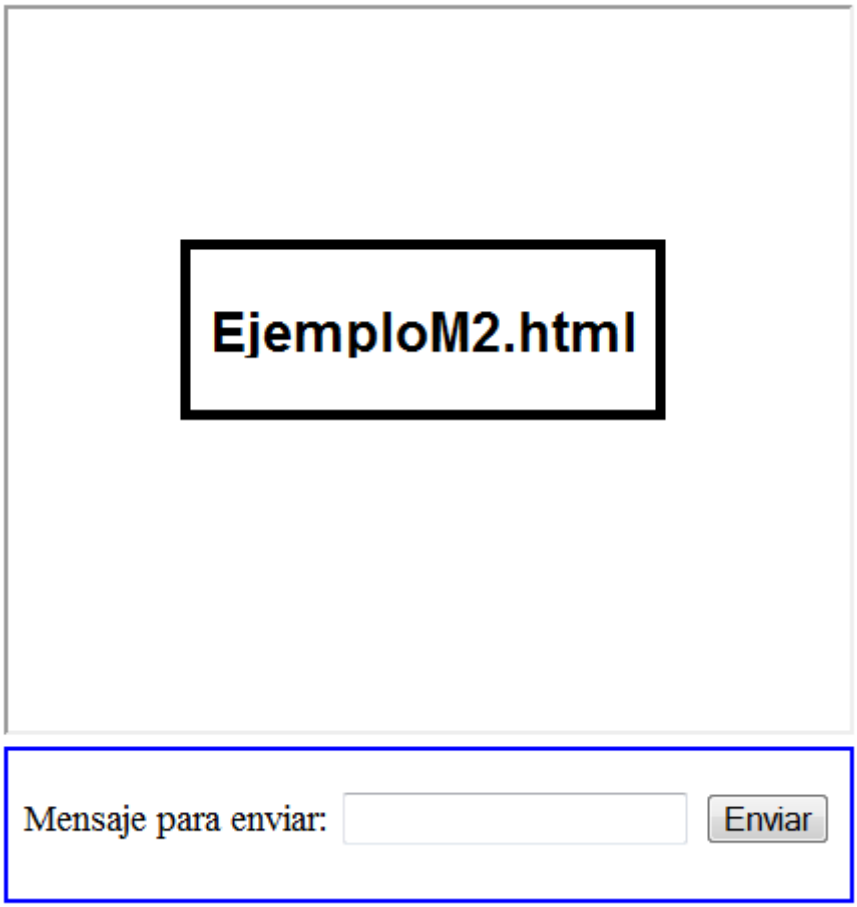
En primer lugar, recogemos el valor del campo de entrada donde escribimos el mensaje a enviar. Luego, definimos el dominio de la ventana de destino, que como vemos será la misma que la de esta página. En lugar de definir así el dominio de destino, al tratarse del mismo que el del emisor, podremos poner simplemente una barra diagonal ("/"), pero en algunos casos puede dar problemas. Una vez disponemos de los datos necesarios, ejecutamos el envío del mensaje mediante el método *postMessage()*.

Por último, la función que se ejecutará cuando se reciba un mensaje será *recibir()*, y tendrá el siguiente código:

```
41      function recibir(event)
42      {
43          document.getElementById("estado").innerHTML = event.data;
44      }
```

Lo único que hacemos, es recoger el mensaje recibido mediante el atributo *event.data* del propio evento que ha lanzado la función, y colocarlo en el elemento *SPAN* que hemos definido para ello.

Si vemos esta primera página en ejecución, el resultado sería el siguiente:



Tenemos el *iFrame* donde irá cargada nuestra segunda página, y debajo el campo de entrada y el botón para el envío de los mensajes. El elemento *SPAN* que hemos definido para colocar la respuesta de la segunda página, está vacío de momento.

Una vez definida nuestra primera página, vamos a ver cómo hemos definido la segunda. El código HTML sería el siguiente:

EjemploM2.html

```html
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo Mensajeria</title>

    <style>
        #mensajes{
            width:320px;
            height:200px;
            border:2px solid black;
            padding:5px;
        }
    </style>
</head>

<body>
    <h2>Mensajes recibidos en Destino</h2>
    <div id="mensajes"></div>

</body>
</html>
```

Vemos como en este caso, únicamente tenemos un encabezado y un elemento *DIV* donde iremos colocando los mensajes que se reciban desde nuestra primera página. El formato esta vez también, se lo hemos aplicado mediante la etiqueta *STYLE*.

Una vez definida la estructura, vamos a ver el contenido JavaScript. El código sería el siguiente:

```javascript
window.addEventListener("message",leer,false);

function leer(event)
{
    document.getElementById("mensajes").innerHTML += event.data+"<br>";

    var mnsg = "OK mensaje: "+event.data;
    event.source.postMessage(mnsg, event.origin);
}
```

Vemos que lo primero que hacemos es poner a la página escuchando al evento *message*, para que cuando reciba un mensaje se ejecute la función *leer()*. Esta función como vemos, recoge el mensaje recibido mediante el atributo *event.data* propio del evento que lanza la función, y lo añade al *DIV* con identificador "mensajes" definido para ello. Por otro lado, renvía el mensaje recibido con un "OK" delante a la página que se lo ha enviado, para así confirmar la recepción del mismo.

Esta respuesta, se envía mediante el método *postMessage()*, pero necesitamos ejecutarlo utilizando la referencia de la ventana de destino correcta, por eso, utilizamos *event.source* para ejecutar el método, y en el parámetro de dominio del destino colocamos el *event.origin* que es el dominio del elemento que nos ha enviado el mensaje, y por tanto el receptor de la respuesta.

Con las dos páginas ya en funcionamiento, el resultado sería el siguiente:

Vemos como en este caso, ya está cargada la página *EjemploM2.html* sobre nuestro *iFrame*. Hemos enviado el mensaje "Hola, ¿qué tal?" desde nuestra primera página, lo ha recibido la página cargada sobre el *iFrame*, y lo ha publicado en su contenedor destinado para ello. A la vez, ha devuelto una respuesta que se ha publicado en el elemento *SPAN* como vemos.

c) Canales y puertos con Web Messaging

Hemos visto el funcionamiento de la API de *Web Messaging*, pero nos falta un apartado importante dentro de esta. Aparte de lo que hemos visto, existe un objeto llamado ***MessageChannel***, para crear canales de comunicación. Cada objeto de este tipo, dispone de dos puertos (***port1*** y ***port2***) que son objetos ***MessagePort*** y que podemos utilizar para intercambiar información con otros documentos a través de ellos.

Por lo tanto, en el momento que creamos un objeto de tipo *MessageChannel*, disponemos de dos puertos a través de los cuales podemos enviar o recibir mensajes como hemos visto con anterioridad. Estos puertos disponen de los siguientes elementos:

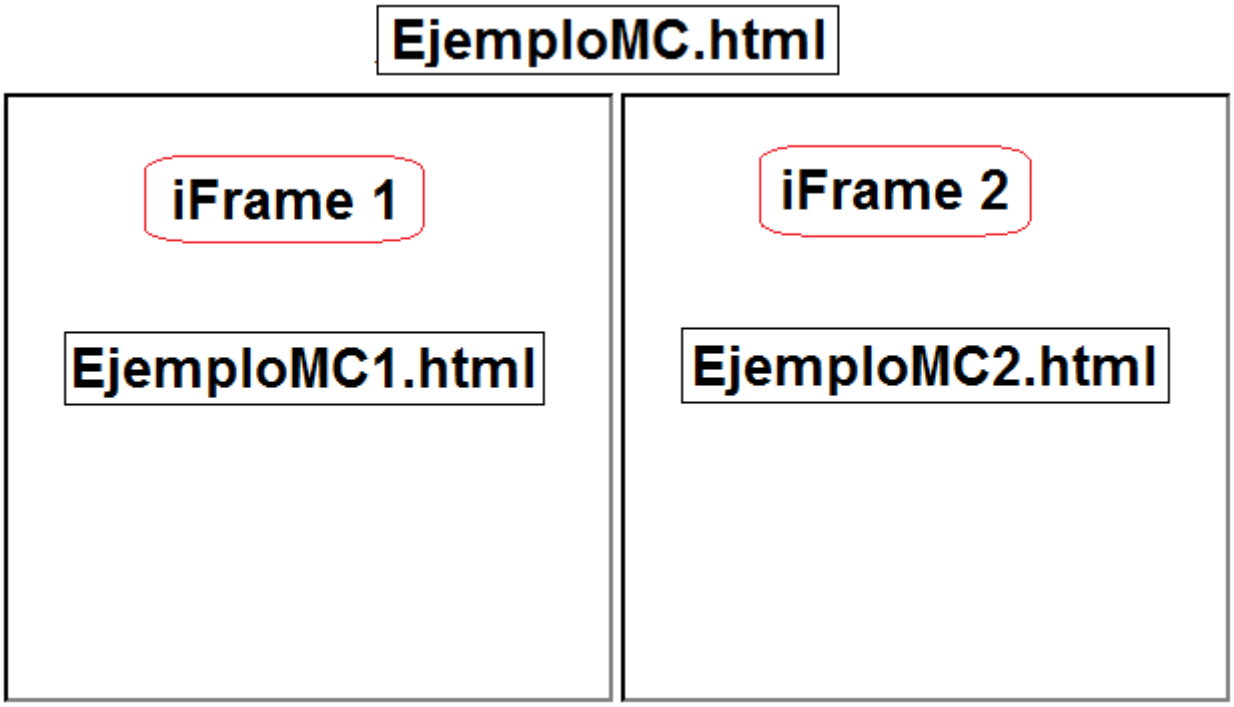
Método ***postMessege(mensaje)***: Función que envía el mensaje por el puerto en cuestión.

Método ***start()***: Función que abre el puerto para que comience el envío de mensajes a través de él.

Método ***stop()***: Función que cierra el puerto para que termine el envío de mensajes a través de él.

Evento ***message***: Evento que se ejecuta cuando se recibe un mensaje por el puerto en cuestión.

Vamos a ver todo esto más claro con el siguiente ejemplo. Vamos a tener en nuestra página dos *iFrames*, y en cada uno de ellos, vamos a tener cargadas dos páginas diferentes e independientes entre sí. En una determinada situación, una de ellas necesitará información que contiene la otra. Para ello, la página principal hará de intermediaria para ponerlas en contacto y facilitar los medios de transmisión. La estructura por lo tanto será algo como esto:



La página contenida en el *iFrame 1* será la que necesite información de la página contenida en el *iFrame 2*.

Vamos a comenzar viendo el código de la página *EjemploMC1.html*, que será el siguiente:

EjemploMC1.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo Mensajeria Canal</title>

    <style>
        #datos{
            width:250px;
            min-height:100px;
            border:2px solid red;
            padding:10px;
        }
    </style>

</head>

<script type="text/javascript">

    function comenzar(){

        var canal = new MessageChannel();

        var protocol = window.location.protocol;
        var host = window.location.host;
        var org = protocol+"//"+host;

        window.parent.postMessage("¡Receptor listo!", org, [canal.port2]);

        canal.port1.addEventListener("message",leer,false);

        canal.port1.start();
    }

    function leer(event)
    {
        document.getElementById("datos").innerHTML += event.data+"<br>";
    }

</script>

<body>
    <h2>Receptor de datos
    <input type="button" onclick="comenzar()" value="Solicitar datos"></h2>

    <div id="datos"></div>

</body>
</html>
```

Vemos que esta página juega el papel de receptor de datos, ya que será quien necesite los datos de nuestra segunda página, y tendrá que recogerlos. Por lo tanto, únicamente tenemos un título y un botón, y un elemento *DIV* con identificador "datos", que será donde vayamos colocando los datos que se reciban. En el momento que pulsemos el botón, será cuando comience todo el proceso de petición de los datos.

En ese momento, se ejecutará la función *comenzar()*, que como vemos, lo primero que hace es crear un canal de comunicaciones. Necesitaremos los dos puertos, ya que por uno nos pondremos a

escuchar cualquier recepción de información, y el otro lo utilizará nuestra segunda página para enviar los datos solicitados.

Una vez creado dicho canal, vamos a enviar un mensaje a la página principal, para indicarle que la página ya está lista y que necesita los datos de la segunda página. Para ello, el dominio será el mismo que el de esta página, pero aparte, le enviamos como tercer parámetro, el objeto *MessagePort* de nuestro segundo puerto, ya que este será, como hemos dicho, el canal que utilice la segunda página para enviar la información.

Una vez hecho esto, ponemos el puerto uno a escuchar cualquier recepción de mensajes, y ejecutamos el método *start()* para que comience la recepción. Por lo tanto, esta página queda a la espera de información.

Por otro lado, cuando esta página reciba algún mensaje, se ejecutará la función *leer()*, que lo único que hará, será publicar dicho mensaje en el *DIV* creado para ello.

El código de la página principal *EjemploMC.html*, será el siguiente:

EjemploMC.html

```
<html>
<head>
    <title>Ejemplo Mensajeria Canal</title>
</head>

<script type="text/javascript">

    window.onload = function ()
    {
        if (window.postMessage)
        {
            alert("El navegador soporta WebMessaging");
            window.addEventListener("message",recibir,false);
        }
        else
        {
            alert("Lo sentimos, el navegador no soporta WebMessaging");
        }
    }

    function recibir(event)
    {
        if(confirm("El receptor solicita los datos del emisor..."))
        {
            var protocol = window.location.protocol;
            var host = window.location.host;
            var org = protocol+"//"+host;

            var cad = "Aceptada transferencia...";
            var contenedor = document.getElementById("contenedor2");
            contenedor.contentWindow.postMessage(cad, org, event.ports);
        }
    }

</script>

<body>

    <iframe id="contenedor1" src="EjemploMC1.html" height="300" width="300"></iframe>

    <iframe id="contenedor2" src="EjemploMC2.html" height="300" width="300"></iframe>

</body>
</html>
```

Esta página principal, dispone de los dos *iFrames* con las dos páginas cargadas, y como hemos dicho, hará de intermediaria entre ambas, para que puedan transmitirse datos entre ellas. Por eso, en el momento que se carga esta página, se comprueba a ver si el navegador soporta *Web Messaging*, y en caso afirmativo, la página se pone escuchar el evento *message*, para que cuando reciba un mensaje ejecute la función *recibir()*.

El mensaje que recibirá, será el que le envíe la primera página comunicándole que está lista y que necesita los datos de la segunda. Por lo tanto, esta tendrá que ponerse en contacto con la segunda para comunicárselo. Como vemos, la función recibir(), lo primero que hace es preguntar a ver si aceptamos la transferencia o no. En caso negativo, no hará nada, pero en caso afirmativo, enviará un mensaje a la segunda página.

En este mensaje, le comunicará que la transferencia ha sido aceptada, el origen como vemos será el mismo que el de la página principal, y como tercer parámetro, le enviará el puerto que ha recibido desde la primera página, y que en este caso lo contiene el atributo *event.ports*.

Y por último, el código de la página *EjemploMC2.html*, será el siguiente:

EjemploMC2.html

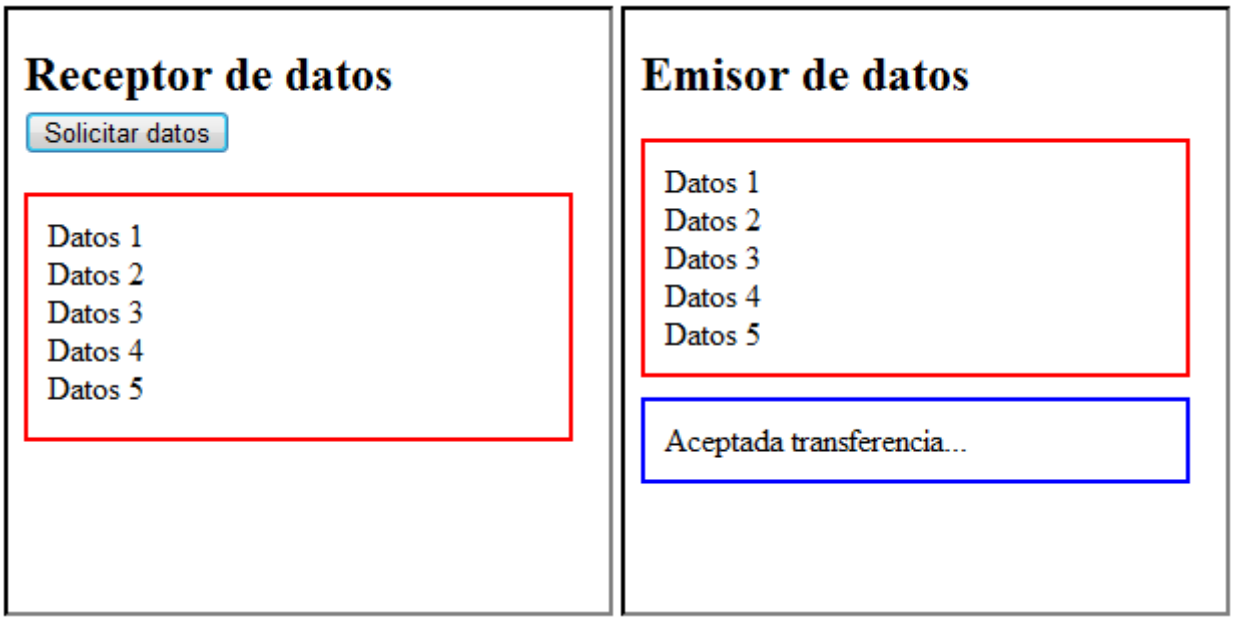
```
<!DOCTYPE html>
<html>
<head>
    <title>Ejemplo Mensajeria Canal</title>

    <style>
        #informacion{
            width:250px;
            border:2px solid red;
            padding:10px;
        }
        #estado{
            width:250px;
            border:2px solid blue;
            padding:10px;
            margin:10px 0px;
        }
    </style>

</head>

<script type="text/javascript">

    window.addEventListener("message",leer,false);

    function leer(event)
    {
        document.getElementById("estado").innerHTML = event.data+"<br>";

        for(i=1;i<=5;i++)
        {
            var mnsg = document.getElementById("info"+i).innerHTML;
            event.ports[0].postMessage(mnsg);
        }
    }

</script>

<body>
    <h2>Emisor de datos</h2>
    <div id="informacion">
        <span id="info1">Datos 1</span><br>
        <span id="info2">Datos 2</span><br>
        <span id="info3">Datos 3</span><br>
        <span id="info4">Datos 4</span><br>
        <span id="info5">Datos 5</span><br>
    </div>
    <div id="estado">Esperando...</div>
</body>
</html>
```

En este caso, disponemos de un título, y dos elementos DIV diferentes. En el primero de ellos, con identificador "informacion", disponemos de la información que la primera página necesita, y en el segundo, con identificador "estado", mostramos el estado de la transferencia de datos. Imaginaros que los datos que vamos a transferir, han sido consultados a una base de datos de servidor, y la primera página, en lugar de solicitárselos otra vez al servidor, los recogerá de aquí, evitando así conexiones innecesarias.

Por lo tanto, esta página estará escuchando el evento *message*, y cuando reciba un mensaje ejecutará la función *leer()*. El mensaje recibido será el que la página principal le envíe, para que se ponga en contacto con la página uno y le envíe los datos. En el momento que recibe dicho mensaje, cambia el estado de la transferencia, y publica el mensaje que la página principal le ha enviado, y aparte envía uno por uno los datos solicitados a la primera página. Esta transferencia, la hace a través del puerto que la página uno le ha facilitado, y que en este caso está en la primera posición del array *event.ports*.

De esta manera, estará enviando los datos directamente a la primera página, y esta los irá publicando en el *DIV* correspondiente.

Una vez que tenemos todo en funcionamiento, el resultado será el siguiente:



Tenemos cargadas las dos páginas en los *iFrames* de la página principal. En el momento que pulsamos en el botón de la primera, esta envía un mensaje a la principal para comunicarle que está lista y que necesita datos, de la misma manera le facilita el puerto que la segunda página deberá utilizar para el envío de la información. Esta página principal, al recibir el mensaje, consulta si se acepta la transferencia, y en caso positivo le envía un mensaje a la segunda página para comunicarle que la transferencia ha sido aceptada y facilitarle el puerto de transmisión que deberá utilizar. Cuando la segunda página recibe el mensaje, utilizando el puerto correspondiente le envía los datos solicitados a la primera página.

De esta manera, hemos comunicado dos páginas independientes y se han enviado datos entre ellas.

3) WebSockets

Los *WebSockets*, son una de las novedades más potentes que HTML5 trae consigo. Esta especificación, define una API que permite a las páginas Web, utilizar el protocolo WebSocket para establecer conexiones directas y *full-duplex (pueden recibir y transmitir al mismo tiempo)*, con el servidor remoto. Estas conexiones, reducen muchísimo el tráfico innecesario en la red, ya que se establece una conexión directa para transmisión y recepción de datos.

Mediante los *WebSockets* por tanto, podemos establecer conexiones bidireccionales, permanentes y en tiempo real entre el cliente y el servidor mediante el navegador Web. Está diseñado para ser implementado en los navegadores y servidores Web, pero puede ser utilizado perfectamente en cualquier aplicación de tipo cliente-servidor.

La utilización de estos Sockets o hilos de comunicación, nos dan una mayor rapidez y nos facilitan el intercambio de datos entre el cliente y el servidor. Esta forma de comunicación y transmisión de datos, es especialmente importante en el caso de aplicaciones que deban de funcionar a tiempo real, ya que la velocidad de envío y recepción de datos es muchísimo más rápida y ágil, ya que estamos utilizando una conexión directa con el servidor.

> ¡IMPORTANTE!
> En el siguiente enlace podéis encontrar la recomendación del W3C sobre WebSockets: WebSocket API
> El protocolo utilizado en los WebSockets, está siendo normalizado por la IETF, y podéis encontrar toda la información sobre ello en el siguiente enlace: IETF
> Kaazing publica la primera guía definitiva para losWebSocket de HTML5:
> Kaazing, pioneros de WebSocket de HTML5, ha anunciado ya la guía definitiva sobre este tema. Es un libro técnico liderado por Vanessa Wang y Peter Moskovits, y ya está disponible para su compra. Leer noticia

a) La API de WebSockets

Como hemos explicado, tenemos la posibilidad de establecer conexiones directas entre el cliente y el servidor utilizando los *WebSockets*. Para ello, necesitamos utilizar un lenguaje de Script, que en este caso será Javascript, desde donde tendremos disponible la API que vamos a utilizar para establecer este tipo de conexiones, y para realizar la transferencia de los datos.

Lo primero que haremos, será comprobar si nuestro navegador soporta o no los *WebSockets*, y para ello utilizaremos el elemento ***window.WebSocket*** que será *true* en caso de soportarlos, y *false* en caso contrario. Lo haremos de la siguiente manera:

```
if (window.WebSocket)
{
    alert("El navegador soporta WebSocket");
}
else
{
    alert("Lo sentimos, el navegador no soporta WebSocket");
}
```

Una vez que sabemos si nuestro navegador está preparado para este tipo de conexiones, deberemos de establecer la conexión con el servidor de WebSocket. Esto lo haremos creando un nuevo objeto de tipo WebSocket, el cuál recibirá como parámetro la dirección del servidor que vayamos a utilizar para ello. Lo haremos de la siguiente manera:

```
ws = new WebSocket();
```

En el momento que creamos ese objeto, pasándole la dirección del servidor que vayamos a utilizar, estamos estableciendo la conexión WebSocket con dicho servidor. Por lo tanto, lo que tenemos que hacer una vez establecida, es controlar los eventos que puedan ir sucediendo a lo largo de dicha comunicación. Los eventos de que disponemos para los *WebSockets* son los siguientes:

open: Evento que se ejecuta cuando se ha establecido la conexión.

close: Evento que se ejecuta cuando se termina la conexión.

message: Evento que se ejecuta cuando cliente recibe un mensaje del servidor.

error: Evento que se ejecuta cuando se da un error en la conexión.

Deberemos de controlar estos eventos, y definir las funciones que se deberán lanzar en el momento que se vayan dando.

Por lo tanto, ya tenemos establecida y controlada la conexión, solo nos faltan los atributos y métodos de que dispone el objeto WebSocket y que harán posible la transferencia y recepción de datos entre cliente y servidor. Los que vamos a utilizar serán los siguientes:

close(): Método que detiene la conexión entre cliente y servidor.

send(datos): Método que ejecuta el envío de los datos al servidor.

readyState: Atributo que define el estado de la conexión. Los valores que puede tener son los siguientes: *0(conectando)*, *1(conectado)*, *2(cerrando)* y *3(cerrado)*.

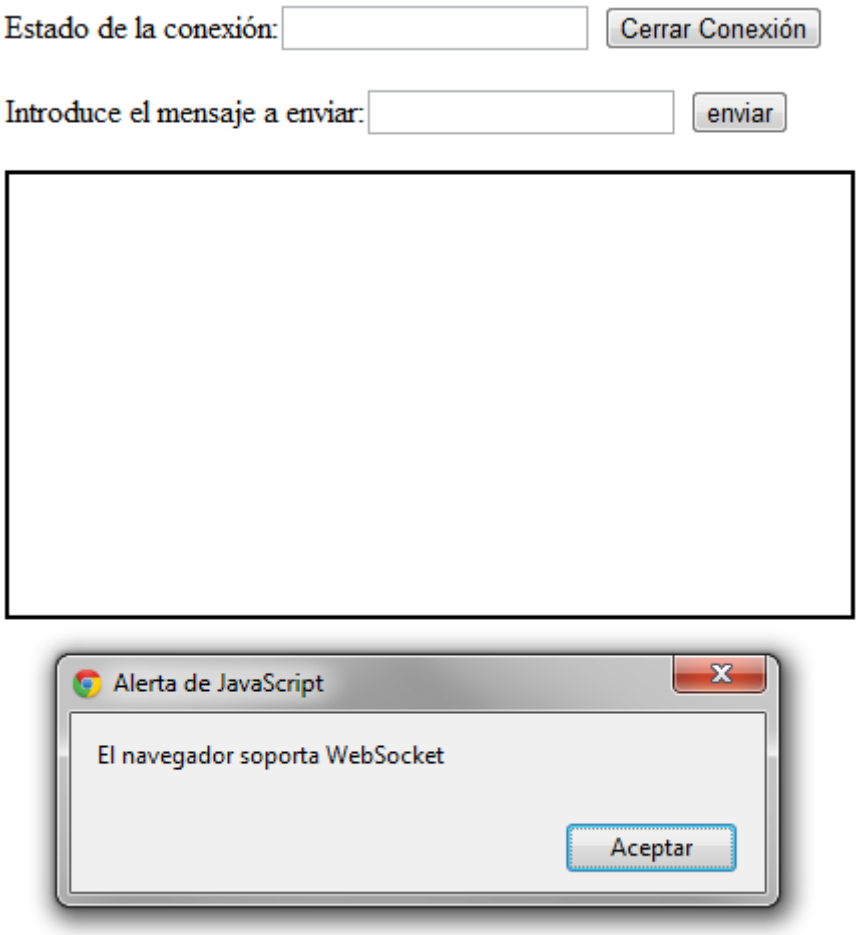
Estos son los métodos y atributos de que dispone el objeto WebSocket, pero necesitamos conocer otro atributo, que es parte del propio evento y que es el que contiene la información o los datos que recibimos del servidor. Este atributo, *evento.**data***, contiene o bien la información que nos envía el servidor, o bien la información sobre el error generado.

Con todo esto, ya tenemos suficiente para crear una aplicación que se conecte a un servidor WebSocket, envíe información al mismo y reciba respuestas. Vamos a verlo en el siguiente ejemplo.

b) Ejemplo con WebSockets

En este ejemplo, vamos a crear una página que haga uso de los *WebSockets*, para conectarse a un servidor que responde con la misma información que recibe. Por lo tanto, podremos enviar la información que queramos, y en ese momento recibiremos como respuesta esa misma información. El servidor que utilizaremos será el siguiente: ***ws://echo.websocket.org/***, que trabajará de la manera que hemos explicado.

El resultado que queremos conseguir es el siguiente:



En el momento que carga la página, comprueba que el navegador soporta *WebSockets* y lo comunica mediante una alerta. A continuación, establecerá una conexión con el servidor indicado como veremos más adelante. En primer lugar, mostraremos el estado de la conexión en el campo de texto que vemos, y en el *DIV* con borde negro, iremos mostrando los mensajes tanto de conexión, como de error, como la información que enviamos al servidor y que recibimos del mismo. Tenemos la opción de enviar el contenido que coloquemos en el campo de entrada, asi como de cerrar la conexión. Vamos a ir poco a poco viendo cómo realizamos todos estos procesos.

En primer lugar, crearemos la estructura de la página. Este será el código que utilizaremos:

EjemploWS.html

```
<!DOCTYPE HTML>
<html>
<head>
    <title>Ejemplo WebSocket</title>

    <style>
        .con{color:#2E2E2E;}
        .env{color:#4B8A08;}
        .rec{color:#0174DF;}
        .err{color:#8A0829;}

        #contenedor{
            border:2px solid black;
            width:400px;
            min-height:200px;
            padding:10px;
        }
    </style>
</head>

<body>

<p>Estado de la conexión:<input type="text" id="estado">
<input type="button" value="Cerrar Conexión" onclick="cierre()"></p>

<p>Introduce el mensaje a enviar:<input type="text" id="entrada">
<input type="button" value="enviar" onclick="envio()"></p>

<div id="contenedor"></div>

</body>
</html>
```

Vemos como hemos creado, el campo de entrada con identificador "estado", para mostrar el estado de la conexión y a su lado, el botón de cierre de conexión, que ejecutará la función *cierre()*. Hemos creado también, el campo de entrada con identificador "entrada", que será donde escribamos la información que queramos mandar al servidor, y el botón de envío, que ejecutará la función *envio()*. Para terminar, hemos creado un *DIV* con identificador "contenedor", que será donde vayamos escribiendo todos los mensajes tanto de conexión, como de error, como los datos que intercambiaremos con el servidor. Mediante la etiqueta *STYLE*, le hemos aplicado el formato deseado a los elementos de nuestra página.

Una vez creada la estructura de nuestra página, vamos a ir viendo las funciones que crearemos en JavaScript. En primer lugar, cuando termina de cargar la página, deberemos por un lado comprobar si nuestro navegador soporta *WebSockets*, y en ese caso, establecer la conexión con el servidor y activar los eventos relacionados con los *WebSockets* que hemos estudiado con anterioridad. El código será el siguiente:

```
23      window.onload = function ()
24      {
25          if (window.WebSocket)
26          {
27              alert("El navegador soporta WebSocket");
28              establecerConexion();
29              estado();
30          }
31          else
32          {
33              alert("Lo sentimos, el navegador no soporta WebSocket");
34          }
35      }
36
37      function establecerConexion()
38      {
39          ws = new WebSocket("ws://echo.websocket.org/");
40          ws.onopen = abrir;
41          ws.onclose = cerrar;
42          ws.onmessage = enviar;
43          ws.onerror = error;
44      }
```

Cuando carga la página, mediante el objeto *window.WebSocket* comprobamos si nuestro navegador soporta *WebSockets*. En caso de que no los acepte, lo comunicaremos mediante una alerta, pero en el caso de que los acepte, aparte de comunicarlo mediante otra alerta, ejecutaremos las funciones *establecerConexion()* y *estado()*. La primera de ellas como vemos, establece la conexión con el servidor indicado, y define que funciones que se deben ejecutar en el momento en que suceda cada uno de los eventos definidos. Y la segunda, actualizará el estado de la conexión como veremos más adelante.

En el momento que establecemos la conexión, sucede el evento *open*, por lo tanto se ejecutará la función *abrir()*, que será la siguiente:

```
46      function abrir()
47      {
48          var cadena = "<p class='con'>Conexion establecida</p>";
49          document.getElementById("contenedor").innerHTML += cadena;
50      }
```

Vemos que esta función, lo único que hace es añadir un mensaje sobre el *DIV* con identificador "contenedor", donde indica que la conexión se ha establecido.

Una vez cargada la página y establecida la conexión, vamos a ver la función *cierre()*, que se ejecutará cuando pulsemos el botón definido para ello, y que se encargará de cerrar la conexión. El código será el siguiente:

```
81      function cierre()
82      {
83          ws.close();
84      }
```

Lo único que se deberá hacer para cerrar una conexión, será ejecutar el método de *close()* del objeto WebSocket creado.

En ese momento, saltará el evento *close*, y por tanto se ejecutará la función *cerrar()* como hemos definido al principio. El código de esta función será el siguiente:

```
52      function cerrar()
53      {
54          var cadena = "<p class='con'>Conexion cerrada</p>";
55          document.getElementById("contenedor").innerHTML += cadena;
56      }
```

Vemos que esta función, lo único que hace es añadir un mensaje sobre el *DIV* con identificador "contenedor", donde indica que la conexión se ha cerrado.

Hemos visto ya, cómo establecer y cerrar una conexión con el servidor correspondiente, ahora vamos a ver cómo se da el intercambio de información entre ellos. En el momento que introducimos algún tipo de información en el campo de entrada con identificador "entrada", y pulsamos en el botón "enviar", se ejecutará la función *envio()*, cuyo código será el siguiente:

```
75      function envio()
76      {
77          var enviado = document.getElementById("entrada").value;
78          ws.send(enviado);
79      }
```

Cogemos el contenido del elemento de entrada indicado, y hacemos uso del método *send()* del objeto WebSocket para enviarlo al servidor.

En el momento que el servidor responda, saltará el evento *message*, y por tanto como hemos definido con anterioridad, se ejecutará la función *enviar()*. Esta función, mostrará tanto el contenido que hemos enviado, como el que recibimos como respuesta por parte del servidor. El código será el siguiente:

```
58      function enviar(event)
59      {
60          var enviado = document.getElementById("entrada").value;
61
62          var cadena = "<p class='env'>Mensaje enviado: "+enviado+"</p>";
63          document.getElementById("contenedor").innerHTML += cadena;
64
65          var cadena2 = "<p class='rec'>Mensaje recibido: "+event.data+"</p>";
66          document.getElementById("contenedor").innerHTML += cadena2;
67      }
```

Vemos como en primer lugar toma el contenido del elemento de entrada donde hemos escrito el mensaje a enviar, y lo añade sobre el *DIV* con identificador "contenedor". Después, añade también sobre dicho elemento *DIV*, el contenido del atributo *event.data* propio del evento que ha lanzado la función, que contiene la respuesta del servidor.

Como hemos indicado antes, hemos creado un apartado donde iremos mostrando el estado de la conexión, esto lo irá actualizando la función *estado()* que hemos creado, y que se ejecutará cuando carguemos la página, como hemos dicho antes. El código de esta función será el siguiente:

```
86      function estado()
87      {
88          switch(ws.readyState)
89          {
90              case 0:
91                  var estadoT = "Conectando...";
92                  break;
93              case 1:
94                  var estadoT = "Conectado";
95                  break;
96              case 2:
97                  var estadoT = "Desconectando...";
98                  break;
99              case 3:
100                 var estadoT = "Desconectado";
101                 break;
102             default:
103                 var estadoT = "Desconocido";
104                 break;
105         }
106
107         document.getElementById("estado").value = estadoT ;
108         setTimeout(estado,100);
109     }
```

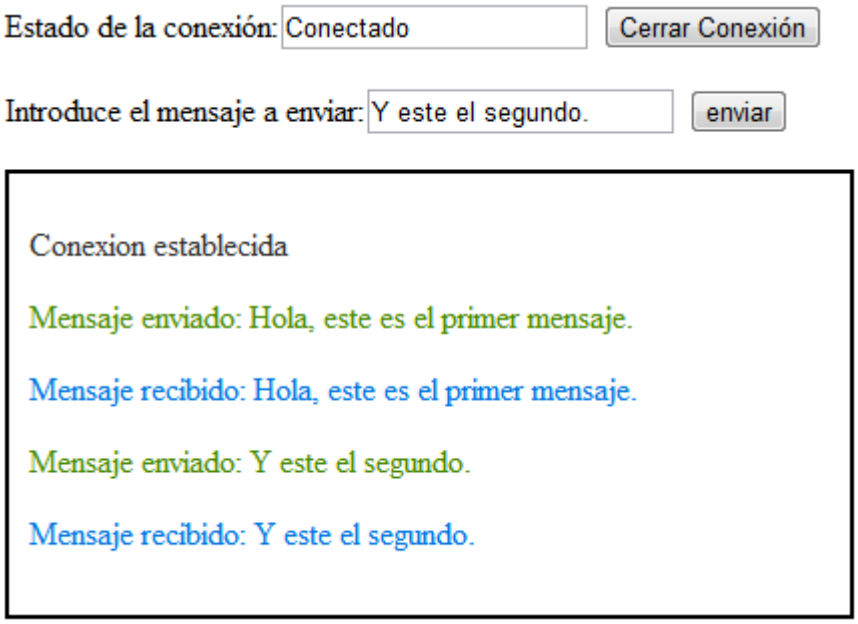
En función del valor del atributo *readyState* propio del objeto WebSocket, mostraremos un estado u otro en el campo de entrada con identificador "estado". Esta función, la iremos ejecutando cada 100 ms para controlar los cambios de estado de la conexión.

Por último, cuando sucede algún error, se lanzará el evento *error* y por tanto se ejecutará la función *error()* que hemos definido. El código de esta función será el siguiente:

```
69      function error(event)
70      {
71          var cadena = "<p class='err'>Ha sucedido un error: "+event.data+"</p>";
72          document.getElementById("contenedor").innerHTML += cadena;
73      }
```

Vemos que esta función, lo único que hace es añadir un mensaje sobre el *DIV* con identificador "contenedor", donde mostrará el error ocurrido. La información de dicho error, la tendrá el atributo *event.data*, propio del evento que ha lanzado la función.

Si lo vemos en plena conexión, el resultado sería el siguiente:



Vemos que la conexión está establecida, y así lo refleja el estado de la conexión. En este caso, hemos enviado dos mensajes, y vemos como la respuesta del servidor es devolvernos el mismo mensaje que le enviamos.

c) Servidores para WebSocket

Hasta ahora, hemos estado viendo como programar nuestra página para que se conecte a un servidor WebSocket e intercambie información con él utilizando hilos o sockets. En el ejemplo, hemos trabajado con un servidor en concreto, que cuando recibía alguna información, respondía con el mismo contenido. Pues bien, para que todo funcione correctamente, el servidor debe de estar preparado y programado para recibir conexiones y darles respuesta a las informaciones recibidas. Para ello, deberemos de instalar o añadir el paquete para el uso de *WebSockets* en el servidor con el que estamos trabajando, o bien trabajar con uno que este preparado para ello. Tendremos también la opción de programar a nuestro gusto, la respuesta que dicho servidor debe de dar en función del mensaje o la información recibida.

A continuación vamos a ver una serie de recursos que podemos utilizar para instalar y poner en funcionamiento nuestro servidor para *WebSockets*:

phpwebsocket: WebSockets basado en PHP. Podemos descargarlo y ver su funcionamiento desde **aquí**.

jWebSocket: WebSockets basado en Java. Podemos descargarlo y ver su funcionamiento desde **aquí**.

web-socket-ruby: WebSockets basado en Ruby. Podemos descargarlo y ver su funcionamiento desde **aquí**.

Socket IO-node: Podemos descargarlo y ver su funcionamiento desde **aquí**.

NodeJS: Utilizado para los servidores múltiples. Podemos descargarlo y ver su funcionamiento desde **aquí**.

Kaazing WebSocket Gateway: Puerta de enlace WebSocket basado en Java. Podemos descargarlo y ver su funcionamiento desde **aquí**.

pyWebSocket: Basado en Phyton, extensión para servidores Apache. Podemos descargarlo y ver su funcionamiento desde **aquí**.

Netty: Marco de red basado en Java que incluye soporte para WebSockets. Podemos descargarlo y ver su funcionamiento desde **aquí**.

websocket-for-Python: WebSockets basado en Phyton. Podemos descargarlo y ver su funcionamiento desde **aquí**.

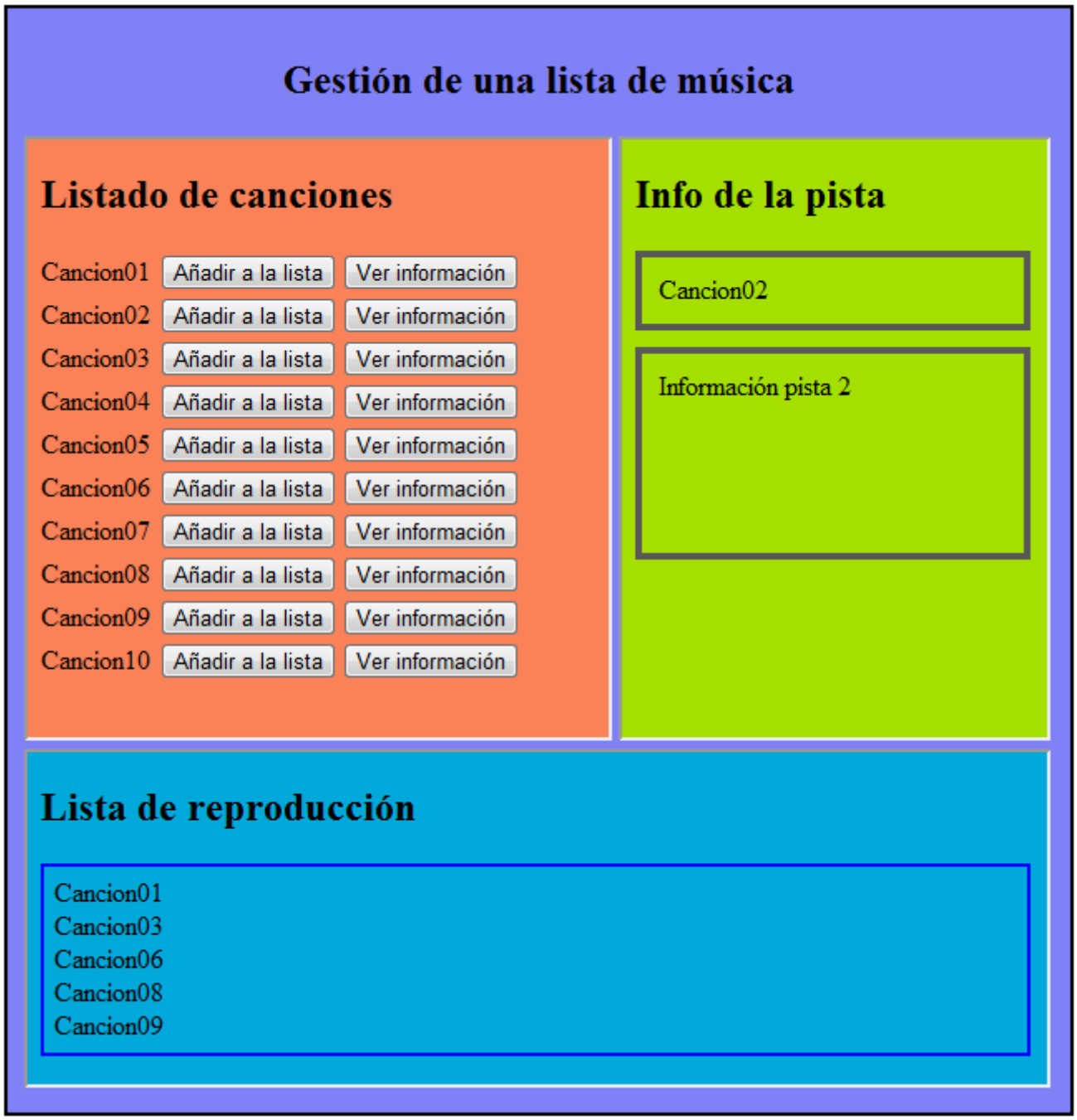
¡IMPORTANTE!
Como vemos, todas estas opciones están basadas en algún lenguaje de servidor, y por tanto necesitamos conocer dicho lenguaje para sacarle partido realmente. Aunque en la mayoría, se ofrece una ayuda para poder empezar a trabajar con cualquiera de ellas.

Ejercicios

Ejercicio 1: Crear un sistema para gestionar una lista de reproducción de música

Vamos a crear un sistema para gestionar una lista de reproducción de música. La página tendrá cuatro documentos diferentes, uno principal con tres *iFrames*, y sobre ellos estarán cargados los otros tres documentos. El primero de ellos, dispondrá de la lista completa de canciones, así como de la información de las mismas, aunque esta última se mantendrá oculta. Sobre el segundo documento, mostraremos el nombre y la información de la canción seleccionada. Y el tercero, irá almacenando la lista de reproducción, en función de cuáles sean las canciones que vayamos seleccionando en el documento que contiene la lista completa.

Por lo tanto, el documento que muestra la información y el documento que almacena la lista de reproducción, necesitarán información que contiene el documento que muestra la lista completa de canciones. En ese proceso de envío de información entre ellos, el documento principal hará de intermediario. El objetivo será conseguir algo similar a lo que vemos en la siguiente imagen:



Pasos a seguir

1. En primer lugar, estructuraremos la página para conseguir el formato que vemos en la imagen. Como hemos dicho, serán cuatro documentos diferentes. El primero de ellos, contendrá tres *iFrames*, y en cada uno de ellos estará cargado cada uno de los otros tres documentos.
2. Una vez estructurada la página, debemos poner a escuchar a las páginas los eventos necesarios. La página del listado completo, deberá disponer de medios para poder enviar información a cualquiera de las otras dos páginas cargadas sobre los *iFrames*, y en este trabajo, la página principal hará de intermediaria.

3. Una vez tenemos abiertos todos los medios para la transferencia de información, deberemos crear las funciones que se ejecutarán cuando pulsemos en los botones de añadir y ver, y que ejecutarán la transferencia de la información necesaria.
4. Cada uno de las páginas que reciba información, deberá tener definido qué hacer cuando le lleguen los mensajes.