



Desarrollo de aplicaciones multiplataforma

Elaborado por: José Antonio Sánchez

Módulo 3

Programación HTML5

SESIÓN 7

12. Web Workers

Los navegadores ejecutan las aplicaciones en un único thread

Si JavaScript está ejecutando una tarea muy pesada, que se traduce en tiempo de procesado, el rendimiento del navegador se ve afectado

Los Web workers se introdujeron con la idea de simplificar la ejecución de threads en el navegador

Un worker permite crear un entorno en el que un bloque de código JavaScript puede ejecutarse de manera paralela sin afectar al thread principal del navegador

12. Web Workers

Los Web workers se ejecutan en un subproceso aislado por tanto, es necesario que el código que ejecutan se encuentre en un archivo independiente

Lo primero que se tiene que hacer es crear un nuevo objeto Worker en la página principal

```
var worker = new Worker('task.js');
```

Si el archivo especificado existe, el navegador generará un nuevo subproceso de Worker que descargará el archivo JavaScript de forma asíncrona

El Worker no comenzará a ejecutarse hasta que el archivo se haya descargado completamente

12.1. API de transferencia de mensajes

Antes de comenzar a utilizar los Worker, es necesario conocer el protocolo de paso de mensajes

Este protocolo es utilizado en otras APIs como WebSocket y Server-Sent Event

El API de transferencia de mensajes es una manera muy simple de enviar cadenas de caracteres entre un origen (o un dominio) a un destino

Por ejemplo se puede utilizar para enviar información a una ventana abierta como popup, o a un iframe dentro de la página, aún cuando tiene como origen otro dominio

12.1. API de transferencia de mensajes

La comunicación entre un Worker y su página principal se realiza mediante un modelo de evento y el método `postMessage()`

En función del navegador o de la versión, `postMessage()` puede aceptar una cadena o un objeto JSON como argumento único

Siempre podemos utilizar los métodos `JSON.stringify` y `JSON.parse` para la transferencia de objetos entre el thread principal y los Worker

12.1. API de transferencia de mensajes

En la tarea principal:

```
var worker = new Worker('doWork.js');  
worker.postMessage('Hello World');
```

En el Worker (doWork.js):

```
self.addEventListener('message', function(event) {  
    self.postMessage(event.data);  
}, false);
```

Al ejecutar `postMessage()` desde la página principal, el Worker es capaz de obtener este mensaje escuchando al evento “message” y puede acceder a los datos del mensaje a través de la propiedad “data” del evento

12.1. API de transferencia de mensajes

En la tarea principal:

```
var worker = new Worker('doWork.js');  
worker.postMessage('Hello World');
```

En el Worker (doWork.js):

```
self.addEventListener('message', function(event) {  
    self.postMessage(event.data);  
}, false);
```

`postMessage()` también sirve para transferir datos de vuelta al thread principal

12.1. API de transferencia de mensajes

Los mensajes que se transfieren entre el origen y los Worker se **copian**, no se pasan por referencia.

El objeto se serializa al transferirlo al Worker y, posteriormente, se anula la serialización en la otra fase del proceso

El origen y el Worker **no comparten la misma instancia**, por lo que el resultado final es la creación de un duplicado en cada transferencia

12.1. API de transferencia de mensajes

En el documento principal

```
<button onclick="sayHI()">Say HI</button>
<button onclick="unknownCmd()">Send unknown command</button>
<button onclick="stop()">Stop worker</button>
<output id="result"></output>

<script>
  function sayHI() { worker.postMessage({'cmd': 'start', 'msg': 'Hi'});}
  function stop() { worker.postMessage({'cmd': 'stop', 'msg': 'Bye'});}
  function unknownCmd() { worker.postMessage({'cmd': 'foobard', 'msg': '???'});}

  var worker = new Worker('doWork.js');
  worker.addEventListener('message', function(e) {
    document.getElementById('result').textContent = e.data;
  }, false);
</script>
```

12.1. API de transferencia de mensajes

En el doWork.js

```
this.addEventListener('message', function(e) {  
    var data = e.data;  
    switch (data.cmd) {  
        case 'start':  
            this.postMessage('WORKER STARTED: '+data.msg);  
            break;  
        case 'stop':  
            this.postMessage('WORKER STOPPED: '+data.msg+'. (buttons will no longer work)');  
            this.close(); // Terminates the worker.  
            break;  
        default:  
            this.postMessage('Unknown command: '+data.msg);  
    }  
}, false);
```

12.2. Uso de Web Workers

Un Worker es una manera ejecutar código JavaScript de manera paralela al proceso principal, sin interferir con el navegador

Al igual que con el resto de funcionalidades de HTML5, debemos comprobar su disponibilidad en el navegador en el que ejecutamos la aplicación:

```
if(Modernizr.webworkers) {  
    alert('El explorador soporta Web workers');  
} else {  
    alert('El explorador NO soporta Web workers');  
}
```

Crear nuevo Worker es muy sencillo. Tan sólo tenemos que crear una nueva instancia del objeto Worker, indicando como parámetro del constructor el fichero JavaScript que contiene el código que debe ejecutar el Worker.

12.2. Uso de Web Workers

Para crear un nuevo Worker tan sólo tenemos que crear una nueva instancia del objeto Worker, indicando como parámetro el fichero JavaScript que contiene el código que debe ejecutar el Worker

```
var worker = new Worker('my_worker.js');
```

La manera de comunicarnos con el nuevo Worker es a través API de transferencia de mensajes

```
worker.postMessage('Hello World');
```

Este método únicamente acepta un parámetro, la cadena de texto a enviar al Worker

12.2. Uso de Web Workers

Por otra parte, la manera de recibir mensajes originados en el Worker es definiendo un escuchador para el evento message

```
worker.addEventListener('message', function(event) {  
    alert(event.data);  
}, false);
```

12.2. Uso de Web Workers

En el código del Worker

Dentro de un Worker necesitamos comunicarnos con el thread principal, tanto para recibir los datos de los mensajes como para nuevos datos de vuelta

Para ello, añadimos un escuchador para el evento “message”, y enviamos los datos de vuelta utilizando API de transferencia de mensajes

```
this.addEventListener('message', function(e) {  
    postMessage("I'm done!");  
});
```


12.2. Uso de Web Workers

En el código del Worker

La visibilidad de un Worker es mucho más reducida, por ejemplo, la palabra reservada “this” no hace referencia al objeto window, sino al Worker en sí mismo

Los Workers tienen restringido el acceso a las siguientes funciones:

- ✓ DOM (no es seguro para el subproceso)
- ✓ Objeto window
- ✓ Objeto document
- ✓ Objeto parent

12.2. Uso de Web Workers

En el código del Worker

Debido al comportamiento de ejecución en paralelo de los Web workers, éstos solo pueden acceder a algunas funciones (según especificación):

- ✓ Enviar datos con `postMessage` y aceptar mensajes entrantes a través del evento `onmessage`
- ✓ `close`, para terminar con el Worker actual
- ✓ Realizar peticiones Ajax
- ✓ Utilizar las funciones de tiempo `setTimeout()/clearTimeout()` y `setInterval()/clearInterval()`.
- ✓ Las siguientes funciones de JavaScript: `eval`, `isNaN`, `escape`, etc.
- ✓ WebSockets
- ✓ EventSource
- ✓ Bases de datos Web SQL, IndexedDB
- ✓ Web Workers

12.3. Subworkers

Los Workers tienen la capacidad de generar Workers secundarios (subtareas)

A la hora de utilizar estos Subworkers, y antes de poder devolver el resultado final al hilo principal, es necesario que todos los procesos hayan terminado

```
var pendingWorkers = 0, results = {};  
onmessage = function (event) {  
    var data = JSON.parse(event.data), worker = null;  
    pendingWorkers = data.length;  
    for (var i = 0; i < data.length; i++) {  
        worker = new Worker('subworker.js');  
        worker.postMessage(JSON.stringify(data[i]));  
        worker.onmessage = storeResult(event);  
    }  
}
```

12.3. Subworkers

```
function storeResult(event) {  
    var result = JSON.parse(event.data);  
  
    pendingWorkers--;  
    if (pendingWorkers <= 0) {  
        postMessage(JSON.stringify(results));  
    }  
}
```

12.4. Gestión de errores

Si se produce un error mientras se ejecuta un Worker, se activa un evento error

La interfaz incluye tres propiedades útiles para descubrir la causa del error

- ✓ filename: el nombre de la secuencia de comandos del Worker que causó el error
- ✓ lineno: el número de línea donde se produjo el error
- ✓ message: una descripción significativa del error

12.4. Gestión de errores

```
function onError(error) {  
    document.getElementById('error').textContent = [  
        'ERROR: Line ', error.lineno, ' in ', error.filename, ': ',  
        error.message].join("");  
}  
  
function onMsg(event) {  
    document.getElementById('result').textContent = event.data;  
}  
  
var worker = new Worker('workerWithError.js');  
worker.addEventListener('message', onMsg, false);  
worker.addEventListener('error', onError, false);
```

12.5. Seguridad

Debido a las restricciones de seguridad de Google Chrome, los Workers no se ejecutarán de forma local (por ejemplo, desde file://) en las últimas versiones del navegador

Uncaught Error: SECURITY_ERR: DOM Exception 18

Para ejecutar tu aplicación desde el esquema file://, ejecuta Chrome con el conjunto de marcadores --allow-file-access-from-files

No se puede cargar una secuencia de comandos desde una URL **data:** o una URL **javascript:**. Tampoco, una página https: puede iniciar secuencias de comandos con una URL **http:**.

12.6. Ejercicio práctico

Crear un Web worker que dado un número entero, calcule todos los números primos comprendidos entre 1 y dicho número

Proporcionaremos a este Worker un número entero, y devolverá un array con todos los números primos encontrados. Mostrar el listado de números primos en el documento principal

WebWorkers

Introduce un número entero mayor que 1:

Tiempo empleado: 1ms

Primos: 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101
103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197
199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311
313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431
433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541 547 557
563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661
673 677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797 809
811 821 823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937
941 947 953 967 971 977 983 991 997

13. XMLHttpRequest2

Un objeto XMLHttpRequest se utiliza para intercambiar datos con un servidor

```
var req = new XMLHttpRequest();  
req.onload = reqListener;  
req.open("get", "yourFile.txt", true);  
req.send();
```

El objeto (jqXHR) que devuelve una llamada “\$.ajax()” de jQuery es un superconjunto del objeto nativo del navegador XMLHttpRequest

13. XMLHttpRequest2

XMLHttpRequest2 introduce una gran cantidad de nuevas funciones que ponen fin a los problemas de nuestras aplicaciones web:

- ✓ Solicitudes de origen cruzado
- ✓ Eventos de progreso de subidas
- ✓ Compatibilidad con subida/bajada de datos binarios
- ✓ ...

Esto permite a AJAX trabajar en coordinación con muchas de las API HTML5, como API de FileSystem, el API de Web Audio y WebGL

13. XMLHttpRequest2

Recuperar archivos como “blob” binario era muy complicado con XHR

El truco que se ha documentado mucho implicaba anular el tipo mime con un conjunto de caracteres definido por el usuario:

```
var xhr = new XMLHttpRequest();  
xhr.open('GET', '/path/to/image.png', true);  
xhr.overrideMimeType('text/plain; charset=x-user-defined');
```

13. XMLHttpRequest2

```
xhr.onreadystatechange = function(e) {  
    if (this.readyState == 4 && this.status == 200) {  
        var binStr = this.responseText;  
        for (var i = 0, len = binStr.length; i < len; ++i) {  
            var c = binStr.charCodeAt(i);  
            var byte = c & 0xff; // byte at offset i  
        }  
    }  
};  
xhr.send();
```

Lo que se obtiene realmente en `responseText` es una cadena binaria que representa el archivo de imagen, no el tipo “blob”

13.1. Formato de respuesta

Las nuevas propiedades de XMLHttpRequest (responseType y response) permiten indicar al navegador el formato en el que queremos que nos devuelva los datos

- ✓ xhr.responseType: **antes** de enviar una solicitud, establece el tipo de respuesta a *text*, *arraybuffer*, *blob* o *document*. La respuesta predeterminada es *text*
- ✓ xhr.response: **después** de una solicitud correcta, la propiedad response de xhr contendrá los datos solicitados como DOMString, ArrayBuffer, Blob o Document (en función del valor establecido en responseType)

Con esto, podemos recuperar la imagen como Blob en lugar de como una cadena

13.1. Formato de respuesta

```
window.URL = window.URL || window.webkitURL; // Take care of vendor prefixes.
```

```
var xhr = new XMLHttpRequest();  
xhr.open('GET', '/path/to/image.png', true);  
xhr.responseType = 'blob'; // Use Blob instead of ArrayBuffer
```

```
xhr.onload = function(e) {  
    if (this.status == 200) {  
        var blob = this.response;  
        var img = document.createElement('img');  
        img.onload = function(e) {  
            window.URL.revokeObjectURL(img.src); // Clean up after yourself  
        };  
        img.src = window.URL.createObjectURL(blob);  
        document.body.appendChild(img);  
        // more code  
    }  
};  
xhr.send();
```

13.2. Envío de datos

XMLHttpRequest limitaba a enviar datos DOMString o Document (XML)

Se ha rediseñado el método “send()” para aceptar todos estos tipos:

- ✓ DOMString
- ✓ Document
- ✓ FormData
- ✓ Blob
- ✓ File
- ✓ ArrayBuffer

13.2. Envío de datos

Envío de la cadena de datos: `xhr.send(domstring)`

```
sendText('test string');  
function sendTextNew(txt) {  
    var xhr = new XMLHttpRequest();  
    xhr.open('POST', '/server', true);  
    xhr.responseText = 'text';  
    xhr.onload = function(e) {  
        if (this.status == 200) {  
            console.log(this.response);  
        }  
    };  
    xhr.send(txt);  
}
```


13.2. Envío de datos

Envío de formularios: xhr.send(formdata)

```
function sendForm() {  
    var formData = new FormData();  
    formData.append('username', 'johndoe');  
    formData.append('id', 123456);  
  
    var xhr = new XMLHttpRequest();  
    xhr.open('POST', '/server', true);  
    xhr.onload = function(e) { ... };  
    xhr.send(formData);  
}
```

13.2. Envío de datos

Envío de formularios: xhr.send(formdata)

Los objetos FormData se pueden inicializar a partir de un elemento HTMLFormElement de la página

```
<form id="myform" name="myform" action="/server">  
  <input type="text" name="username" value="johndoe">  
  <input type="number" name="id" value="123456">  
  <input type="submit" onclick="return sendForm(this.form);">  
</form>
```

13.2. Envío de datos

Envío de formularios: xhr.send(formdata)

```
function sendForm(form) {  
    var formData = new FormData(form);  
    formData.append('secret_token', '1234567890');  
    var xhr = new XMLHttpRequest();  
    xhr.open('POST', form.action, true);  
    xhr.onload = function(e) { ... };  
    xhr.send(formData);  
    return false; // Prevent page from submitting.  
}
```

13.2. Envío de datos

Envío de formularios: xhr.send(formdata)

FormData también puede incluir subidas de archivos. Simplemente se añade el archivo/s y el navegador construirá una solicitud multipart/form-data cuando se ejecute send()

```
function uploadFiles(url, files) {  
    var formData = new FormData();  
    for (var i = 0, file; file = files[i]; ++i) {  
        formData.append(file.name, file);  
    }  
    var xhr = new XMLHttpRequest();  
    xhr.open('POST', url, true);  
    xhr.onload = function(e) { ... };  
    xhr.send(formData); // multipart/form-data automatic  
}
```

13.2. Envío de datos

Envío de formularios: `xhr.send(formdata)`

Podremos llamar a la función de subida cuando se añada un archivo

```
document.querySelector('input[type="file"]').addEventListener('change', function(e) {  
    uploadFiles('/server', this.files);  
}, false);
```

13.2. Envío de datos

Envío de archivos o blob: xhr.send(blobOrFile)

Se puede crear un nuevo blob desde cero con el API BlobBuilder y se sube al servidor

Además también se puede configurar un controlador para informar al usuario sobre el progreso de la subida:

```
// Take care of vendor prefixes. BlobBuilder = window.MozBlobBuilder ||  
    window.WebKitBlobBuilder || window.BlobBuilder;
```

```
var bb = new BlobBuilder();  
bb.append('hello world');  
upload(bb.getBlob('text/plain'));
```

13.2. Envío de datos

Envío de archivos o blob: xhr.send(blobOrFile)

```
function upload(blobOrFile) {  
    var xhr = new XMLHttpRequest();  
    xhr.open('POST', '/server', true);  
    xhr.onload = function(e) { ... };  
  
    var progressBar = document.querySelector('progress'); // Listen to the upload progress  
    xhr.upload.onprogress = function(e) {  
        if (e.lengthComputable) {  
            progressBar.value = (e.loaded / e.total) * 100;  
            progressBar.textContent = progressBar.value; // Unsupported browsers  
        }  
    };  
    xhr.send(blobOrFile);  
}
```

13.2. Envío de datos

Envío de fragmento de bytes: xhr.send(arraybuffer)

```
function sendArrayBuffer() {  
    var xhr = new XMLHttpRequest();  
    xhr.open('POST', '/server', true);  
    xhr.onload = function(e) { ... };  
  
    var uint8Array = new Uint8Array([1, 2, 3]);  
  
    xhr.send(uint8Array.buffer);  
}
```


13.3. Ejemplo. Descargar y guardar archivos en el sistema de archivos

Supongamos que tienes una galería de imágenes y quieres recuperar un grupo de imágenes para, a continuación, guardarlas localmente con el sistema de archivos HTML5

Una forma de conseguir esto sería solicitar imágenes como ArrayBuffer, crear un Blob a partir de los datos y escribir el blob con FileWriter:

```
window.requestFileSystem = window.requestFileSystem ||  
    window.webkitRequestFileSystem;  
  
function onError(e) {  
    console.log('Error', e);  
}
```

13.3. Ejemplo. Descargar y guardar archivos en el sistema de archivos

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/path/to/image.png', true);
xhr.responseType = 'arraybuffer';
xhr.onload = function(e) {
    window.requestFileSystem(TEMPORARY, 1024 * 1024, function(fs) {
        fs.root.getFile('image.png', {create: true}, function(fileEntry) {
            fileEntry.createWriter(function(writer) {
                writer.onwrite = function(e) { ... };
                writer.onerror = function(e) { ... };
                var bb = new BlobBuilder();
                bb.append(xhr.response);
                writer.write(bb.getBlob('image/png'));
            }, onError);
        }, onError);
    }, onError);
};
xhr.send();
```

13.3. Ejemplo. Dividir un archivo y subir cada fragmento

Con las API de archivo, podemos minimizar el trabajo necesario para subir un archivo de gran tamaño

La técnica es dividir el archivo que se va a subir en varios fragmentos, crear un XHR para cada parte y unir los fragmentos en el servidor

```
window.BlobBuilder = window.MozBlobBuilder || window.WebKitBlobBuilder ||  
    window.BlobBuilder;
```

```
function upload(blobOrFile) {  
    var xhr = new XMLHttpRequest();  
    xhr.open('POST', '/server', true);  
    xhr.onload = function(e) { ... };  
    xhr.send(blobOrFile);  
}
```

13.3. Ejemplo. Dividir un archivo y subir cada fragmento

```
document.querySelector('input[type="file"]').addEventListener('change', function(e) {  
    var blob = this.files[0];  
    const BYTES_PER_CHUNK = 1024 * 1024; // 1MB chunk sizes.  
    const SIZE = blob.size;  
    var start = 0;  
    var end = BYTES_PER_CHUNK;  
    while(start < SIZE) {  
        if ('mozSlice' in blob) var chunk = blob.mozSlice(start, end);  
        else var chunk = blob.webkitSlice(start, end);  
        upload(chunk);  
        start = end;  
        end = start + BYTES_PER_CHUNK;  
    }  
}, false);
```