Primeros pasos con AJAX

6.1. Breve historia de AJAX

La historia de AJAX está íntimamente relacionada con un objeto de programación llamado `XMLHttpRequest`. El origen de este objeto se remonta al año 2000, con productos como Exchange 2000, Internet Explorer 5 y Outlook Web Access.

Todo comenzó en 1998, cuando **Alex Hopmann** y su equipo se encontraban desarrollando la entonces futura versión de Exchange 2000. El punto débil del servidor de correo electrónico era su cliente vía web, llamado OWA (*Outlook Web Access*).

Durante el desarrollo de OWA, se evaluaron dos opciones: un cliente formado sólo por páginas HTML estáticas que se recargaban constantemente y un cliente realizado completamente con HTML dinámico o DHTML. Alex Hopmann pudo ver las dos opciones y se decantó por la basada en DHTML. Sin embargo, para ser realmente útil a esta última le faltaba un componente esencial: "algo" que evitara tener que enviar continuamente los formularios con datos al servidor.

Motivado por las posibilidades futuras de OWA, Alex creó en un solo fin de semana la primera versión de lo que denominó XMLHTTP. La primera demostración de las posibilidades de la nueva tecnología fue un éxito, pero faltaba lo más difícil: incluir esa tecnología en el navegador Internet Explorer.

Si el navegador no incluía XMLHTTP de forma nativa, el éxito del OWA se habría reducido enormemente. El mayor problema es que faltaban pocas semanas para que se lanzara la última beta de Internet Explorer 5 previa a su lanzamiento final. Gracias a sus contactos en la empresa, Alex consiguió que su tecnología se incluyera en la librería MSXML que incluye Internet Explorer.

De hecho, el nombre del objeto (`XMLHTTP`) se eligió para tener una buena excusa que justificara su inclusión en la librería XML de Internet Explorer, ya que este objeto está mucho más relacionado con HTTP que con XML.

6.2. La primera aplicación

6.2.1. Código fuente

La aplicación AJAX completa más sencilla consiste en una adaptación del clásico *"Hola Mundo"*. En este caso, una aplicación JavaScript descarga un archivo del servidor y muestra su contenido sin necesidad de recargar la página.

Código fuente completo:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
```

```
<title>Hola Mundo con AJAX</title>

<script type="text/javascript">
function descargaArchivo() {
  // Obtener la instancia del objeto XMLHttpRequest
  if(window.XMLHttpRequest) {
    peticion_http = new XMLHttpRequest();
  }
  else if(window.ActiveXObject) {
    peticion_http = new ActiveXObject("Microsoft.XMLHTTP");
  }

  // Preparar la funcion de respuesta
  peticion_http.onreadystatechange = muestraContenido;

  // Realizar peticion HTTP
  peticion_http.open('GET', 'http://localhost/holamundo.txt', true);
  peticion_http.send(null);

  function muestraContenido() {
    if(peticion_http.readyState == 4) {
      if(peticion_http.status == 200) {
        alert(peticion_http.responseText);
      }
    }
  }
}

window.onload = descargaArchivo;
</script>

</head>
<body></body>
</html>
```

En el ejemplo anterior, cuando se carga la página se ejecuta el método JavaScript que muestra el contenido de un archivo llamado `holamundo.txt` que se encuentra en el servidor. La clave del código anterior es que la petición HTTP y la descarga de los contenidos del archivo se realizan sin necesidad de recargar la página.

6.2.2. Análisis detallado

La aplicación AJAX del ejemplo anterior se compone de cuatro grandes bloques: instanciar el objeto `XMLHttpRequest`, preparar la función de respuesta, realizar la petición al servidor y ejecutar la función de respuesta.

Todas las aplicaciones realizadas con técnicas de AJAX deben instanciar en primer lugar el objeto `XMLHttpRequest`, que es el objeto clave que permite realizar comunicaciones con el servidor en segundo plano, sin necesidad de recargar las páginas.

La implementación del objeto `XMLHttpRequest` depende de cada navegador, por lo que es necesario emplear una discriminación sencilla en función del navegador en el que se está ejecutando el código:

```
if(window.XMLHttpRequest) {  // Navegadores que siguen los estándares
  peticion_http = new XMLHttpRequest();
}
else if(window.ActiveXObject) {  // Navegadores obsoletos
  peticion_http = new ActiveXObject("Microsoft.XMLHTTP");
}
```

Los navegadores que siguen los estándares (Firefox, Safari, Opera, Internet Explorer 7 y 8) implementan el objeto `XMLHttpRequest` de forma nativa, por lo que se puede obtener a través del objeto `window`. Los navegadores obsoletos (Internet Explorer 6 y anteriores) implementan el objeto `XMLHttpRequest` como un objeto de tipo ActiveX.

Una vez obtenida la instancia del objeto `XMLHttpRequest`, se prepara la función que se encarga de procesar la respuesta del servidor. La propiedad `onreadystatechange` del objeto `XMLHttpRequest` permite indicar esta función directamente incluyendo su código mediante una función anónima o indicando una referencia a una función independiente. En el ejemplo anterior se indica directamente el nombre de la función:

```
peticion_http.onreadystatechange = muestraContenido;
```

El código anterior indica que cuando la aplicación reciba la respuesta del servidor, se debe ejecutar la función `muestraContenido()`. Como es habitual, la referencia a la función se indica mediante su nombre sin paréntesis, ya que de otro modo se estaría ejecutando la función y almacenando el valor devuelto en la propiedad `onreadystatechange`.

Después de preparar la aplicación para la respuesta del servidor, se realiza la petición HTTP al servidor:

```
peticion_http.open('GET', 'http://localhost/prueba.txt', true);
peticion_http.send(null);
```

Las instrucciones anteriores realizan el tipo de petición más sencillo que se puede enviar al servidor. En concreto, se trata de una petición de tipo `GET` simple que no envía ningún parámetro al servidor. La petición HTTP se crea mediante el método `open()`, en el que se incluye el tipo de petición (`GET`), la URL solicitada (`http://localhost/prueba.txt`) y un tercer parámetro que vale `true`.

Una vez creada la petición HTTP, se envía al servidor mediante el método `send()`. Este método incluye un parámetro que en el ejemplo anterior vale `null`. Más adelante se ven en detalle todos los métodos y propiedades que permiten hacer las peticiones al servidor.

Por último, cuando se recibe la respuesta del servidor, la aplicación ejecuta de forma automática la función establecida anteriormente.

```
function muestraContenido() {
  if(peticion_http.readyState == 4) {
    if(peticion_http.status == 200) {
      alert(peticion_http.responseText);
    }
  }
}
```

La función `muestraContenido()` comprueba en primer lugar que se ha recibido la respuesta del servidor (mediante el valor de la propiedad `readyState`). Si se ha recibido alguna respuesta, se comprueba que sea válida y correcta (comprobando si el código de estado HTTP devuelto es igual a `200`). Una vez realizadas las comprobaciones, simplemente se muestra por pantalla el contenido de la respuesta del servidor (en este caso, el contenido del archivo solicitado) mediante la propiedad `responseText`.

6.2.3. Refactorizando la primera aplicación

La primera aplicación AJAX mostrada anteriormente presenta algunas carencias importantes. A continuación, se refactoriza su código ampliándolo y mejorándolo para que se adapte mejor a otras situaciones. En primer lugar, se definen unas variables que se utilizan en la función que procesa la respuesta del servidor:

```
var READY_STATE_UNINITIALIZED = 0;
var READY_STATE_LOADING = 1;
var READY_STATE_LOADED = 2;
var READY_STATE_INTERACTIVE = 3;
var READY_STATE_COMPLETE = 4;
```

Como se verá más adelante, la respuesta del servidor sólo puede corresponder a alguno de los cinco estados definidos por las variables anteriores. De esta forma, el código puede utilizar el nombre de cada estado en vez de su valor numérico, por lo que se facilita la lectura y el mantenimiento de las aplicaciones.

Además, la variable que almacena la instancia del objeto `XMLHttpRequest` se va a transformar en una variable global, de forma que todas las funciones que hacen uso de ese objeto tengan acceso directo al mismo:

```
var peticion_http;
```

A continuación, se crea una función genérica de carga de contenidos mediante AJAX:

```
function cargaContenido(url, metodo, funcion) {
  peticion_http = inicializa_xhr();

  if(peticion_http) {
    peticion_http.onreadystatechange = funcion;
    peticion_http.open(metodo, url, true);
    peticion_http.send(null);
  }
}
```

La función definida admite tres parámetros: la URL del contenido que se va a cargar, el método utilizado para realizar la petición HTTP y una referencia a la función que procesa la respuesta del servidor.

En primer lugar, la función `cargaContenido()` inicializa el objeto `XMLHttpRequest` (llamado `xhr` de forma abreviada). Una vez inicializado, se emplea el objeto `peticion_http` para establecer la función que procesa la respuesta del servidor. Por último, la función `cargaContenido()` realiza la petición al servidor empleando la URL y el método HTTP indicados como parámetros.

La función `inicializa_xhr()` se emplea para encapsular la creación del objeto `XMLHttpRequest`:

```
function inicializa_xhr() {
  if(window.XMLHttpRequest) {
    return new XMLHttpRequest();
  }
  else if(window.ActiveXObject) {
    return new ActiveXObject("Microsoft.XMLHTTP");
  }
}
```

La función `muestraContenido()` también se refactoriza para emplear las variables globales definidas:

```
function muestraContenido() {
  if(peticion_http.readyState == READY_STATE_COMPLETE) {
    if(peticion_http.status == 200) {
      alert(peticion_http.responseText);
    }
  }
}
```

Por último, la función `descargaArchivo()` simplemente realiza una llamada a la función `cargaContenido()` con los parámetros adecuados:

```
function descargaArchivo() {
  cargaContenido("http://localhost/holamundo.txt", "GET", muestraContenido);
}
```

A continuación se muestra el código completo de la refactorización de la primera aplicación:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
<title>Hola Mundo con AJAX, version 2</title>

<script type="text/javascript" language="javascript">

var READY_STATE_UNINITIALIZED=0;
var READY_STATE_LOADING=1;
var READY_STATE_LOADED=2;
var READY_STATE_INTERACTIVE=3;
var READY_STATE_COMPLETE=4;

var peticion_http;

function cargaContenido(url, metodo, funcion) {
  peticion_http = inicializa_xhr();

  if(peticion_http) {
    peticion_http.onreadystatechange = funcion;
    peticion_http.open(metodo, url, true);
    peticion_http.send(null);
  }
}
```

```
function inicializa_xhr() {
  if(window.XMLHttpRequest) {
    return new XMLHttpRequest();
  }
  else if(window.ActiveXObject) {
    return new ActiveXObject("Microsoft.XMLHTTP");
  }
}

function muestraContenido() {
  if(peticion_http.readyState == READY_STATE_COMPLETE) {
    if(peticion_http.status == 200) {
      alert(peticion_http.responseText);
    }
  }
}

function descargaArchivo() {
  cargaContenido("http://localhost/holamundo.txt", "GET", muestraContenido);
}

window.onload = descargaArchivo;

</script>

</head>
<body></body>
</html>
```

Ejercicio 11

A partir de la página web proporcionada, añadir el código JavaScript necesario para que:

1. Al cargar la página, el cuadro de texto debe mostrar por defecto la URL de la propia página.
2. Al pulsar el botón "Mostrar Contenidos", se debe descargar mediante peticiones AJAX el contenido correspondiente a la URL introducida por el usuario. El contenido de la respuesta recibida del servidor se debe mostrar en la zona de "Contenidos del archivo".
3. En la zona "Estados de la petición" se debe mostrar en todo momento el estado en el que se encuentra la petición (No inicializada, cargando, completada, etc.)
4. Mostrar el contenido de todas las cabeceras de la respuesta del servidor en la zona "Cabeceras HTTP de la respuesta del servidor".
5. Mostrar el código y texto de estado de la respuesta del servidor en la zona "Código de estado".

6.3. Métodos y propiedades del objeto XMLHttpRequest

El objeto `XMLHttpRequest` posee muchas otras propiedades y métodos diferentes a las manejadas por la primera aplicación de AJAX. A continuación se incluye la lista completa de todas las propiedades y métodos del objeto y todos los valores numéricos de sus propiedades.

Las propiedades definidas para el objeto `XMLHttpRequest` son:

Propiedad	Descripción
`readyState`	Valor numérico (entero) que almacena el estado de la petición
`responseText`	El contenido de la respuesta del servidor en forma de cadena de texto
`responseXML`	El contenido de la respuesta del servidor en formato XML. El objeto devuelto se puede procesar como un objeto DOM
`status`	El código de estado HTTP devuelto por el servidor (`200` para una respuesta correcta, `404` para "No encontrado", `500` para un error de servidor, etc.)
`statusText`	El código de estado HTTP devuelto por el servidor en forma de cadena de texto: "OK", "Not Found", "Internal Server Error", etc.

Los valores definidos para la propiedad `readyState` son los siguientes:

Valor	Descripción
`0`	No inicializado (objeto creado, pero no se ha invocado el método `open`)
`1`	Cargando (objeto creado, pero no se ha invocado el método `send`)
`2`	Cargado (se ha invocado el método `send`, pero el servidor aún no ha respondido)
`3`	Interactivo (se han recibido algunos datos, aunque no se puede emplear la propiedad `responseText`)
`4`	Completo (se han recibido todos los datos de la respuesta del servidor)

Los métodos disponibles para el objeto `XMLHttpRequest` son los siguientes:

Método	Descripción
`abort()`	Detiene la petición actual
`getAllResponseHeaders()`	Devuelve una cadena de texto con todas las cabeceras de la respuesta del servidor
`getResponseHeader("cabecera")`	Devuelve una cadena de texto con el contenido de la cabecera solicitada
`onreadystatechange`	Responsable de manejar los eventos que se producen. Se invoca cada vez que se produce un cambio en el estado de la petición HTTP. Normalmente es una referencia a una función JavaScript
`open("metodo", "url")`	Establece los parámetros de la petición que se realiza al servidor. Los parámetros necesarios son el método HTTP empleado y la URL destino (puede indicarse de forma absoluta o relativa)
`send(contenido)`	Realiza la petición HTTP al servidor
`setRequestHeader("cabecera", "valor")`	Permite establecer cabeceras personalizadas en la petición HTTP. Se debe invocar el método `open()` antes que `setRequestHeader()`

El método `open()` requiere dos parámetros (método HTTP y URL) y acepta de forma opcional otros tres parámetros. Definición formal del método `open()`:

```
open(string metodo, string URL [,boolean asincrono, string usuario, string password]);
```

Por defecto, las peticiones realizadas son asíncronas. Si se indica un valor `false` al tercer parámetro, la petición se realiza de forma síncrona, esto es, se detiene la ejecución de la aplicación hasta que se recibe de forma completa la respuesta del servidor.

No obstante, las peticiones síncronas son justamente contrarias a la filosofía de AJAX. El motivo es que una petición síncrona *congela* el navegador y no permite al usuario realizar ninguna acción hasta que no se haya recibido la respuesta completa del servidor. La sensación que provoca es que el navegador se ha *colgado* por lo que no se recomienda el uso de peticiones síncronas salvo que sea imprescindible.

Los últimos dos parámetros opcionales permiten indicar un nombre de usuario y una contraseña válidos para acceder al recurso solicitado.

Por otra parte, el método `send()` requiere de un parámetro que indica la información que se va a enviar al servidor junto con la petición HTTP. Si no se envían datos, se debe indicar un valor igual a `null`. En otro caso, se puede indicar como parámetro una cadena de texto, un array de bytes o un objeto XML DOM.

6.4. Utilidades y objetos para AJAX

Una de las operaciones más habituales en las aplicaciones AJAX es la de obtener el contenido de un archivo o recurso del servidor. Por tanto, se va a construir un objeto que permita realizar la carga de datos del servidor simplemente indicando el recurso solicitado y la función encargada de procesar la respuesta:

```
var cargador = new net.CargadorContenidos("pagina.html", procesaRespuesta);
```

La lógica común de AJAX se encapsula en un objeto de forma que sea fácilmente reutilizable. Aplicando los conceptos de objetos de JavaScript, funciones constructoras y el uso de prototype, es posible realizar de forma sencilla el objeto cargador de contenidos.

El siguiente código ha sido adaptado del excelente libro "Ajax in Action", escrito por Dave Crane, Eric Pascarello y Darren James y publicado por la editorial Manning.

```
var net = new Object();

net.READY_STATE_UNINITIALIZED=0;
net.READY_STATE_LOADING=1;
net.READY_STATE_LOADED=2;
net.READY_STATE_INTERACTIVE=3;
net.READY_STATE_COMPLETE=4;

// Constructor
net.CargadorContenidos = function(url, funcion, funcionError) {
  this.url = url;
  this.req = null;
  this.onload = funcion;
  this.onerror = (funcionError) ? funcionError : this.defaultError;
  this.cargaContenidoXML(url);
}

net.CargadorContenidos.prototype = {
```

```
  cargaContenidoXML: function(url) {
    if(window.XMLHttpRequest) {
      this.req = new XMLHttpRequest();
    }
    else if(window.ActiveXObject) {
      this.req = new ActiveXObject("Microsoft.XMLHTTP");
    }

    if(this.req) {
      try {
        var loader = this;
        this.req.onreadystatechange = function() {
          loader.onReadyState.call(loader);
        }
        this.req.open('GET', url, true);
        this.req.send(null);
      } catch(err) {
        this.onerror.call(this);
      }
    }
  },

  onReadyState: function() {
    var req = this.req;
    var ready = req.readyState;
    if(ready == net.READY_STATE_COMPLETE) {
      var httpStatus = req.status;
      if(httpStatus == 200 || httpStatus == 0) {
        this.onload.call(this);
      }
      else {
        this.onerror.call(this);
      }
    }
  },

  defaultError: function() {
    alert("Se ha producido un error al obtener los datos"
      + "\n\nreadyState:" + this.req.readyState
      + "\nstatus: " + this.req.status
      + "\nheaders: " + this.req.getAllResponseHeaders());
  }
}
```

Una vez definido el objeto `net` con su método `CargadorContenidos()`, ya es posible utilizarlo en las funciones que se encargan de mostrar el contenido del archivo del servidor:

```
function muestraContenido() {
  alert(this.req.responseText);
}

function cargaContenidos() {
  var cargador = new net.CargadorContenidos("http://localhost/holamundo.txt",
muestraContenido);
}
```

```
window.onload = cargaContenidos;
```

En el ejemplo anterior, la aplicación muestra un mensaje con los contenidos de la URL indicada:

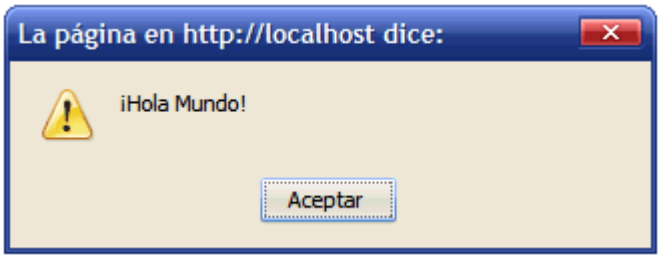

Figura 7.1. Mensaje mostrado cuando el resultado es exitoso

Por otra parte, si la URL que se quiere cargar no es válida o el servidor no responde, la aplicación muestra el siguiente mensaje de error:

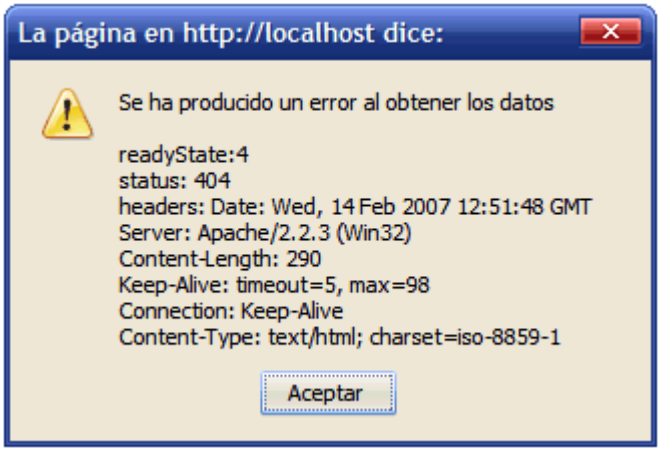

Figura 7.2. Mensaje mostrado cuando el resultado es erróneo

El código del cargador de contenidos hace un uso intensivo de objetos, JSON, funciones anónimas y uso del objeto `this`. Seguidamente, se detalla el funcionamiento de cada una de sus partes.

El primer elemento importante del código fuente es la definición del objeto `net`.

```
var net = new Object();
```

Se trata de una variable global que encapsula todas las propiedades y métodos relativos a las operaciones relacionadas con las comunicaciones por red. De cierto modo, esta variable global simula el funcionamiento de los *namespaces* ya que evita la colisión entre nombres de propiedades y métodos diferentes.

Después de definir las constantes empleadas por el objeto `XMLHttpRequest`, se define el constructor del objeto `CargadorContenidos`:

```
net.CargadorContenidos = function(url, funcion, funcionError) {
  this.url = url;
  this.req = null;
  this.onload = funcion;
  this.onerror = (funcionError) ? funcionError : this.defaultError;
  this.cargaContenidoXML(url);
}
```

Aunque el constructor define tres parámetros diferentes, en realidad solamente los dos primeros son obligatorios. De esta forma, se inicializa el valor de algunas variables del objeto, se comprueba si se ha definido la función que se emplea en caso de error (si no se ha definido, se emplea una función genérica definida más adelante) y se invoca el método responsable de cargar el recurso solicitado (`cargaContenidoXML`).

```
net.CargadorContenidos.prototype = {
  cargaContenidoXML:function(url) {
    ...
  },
  onReadyState:function() {
    ...
  },
  defaultError:function() {
    ...
  }
}
```

Los métodos empleados por el objeto `net.cargaContenidos` se definen mediante su prototipo. En este caso, se definen tres métodos diferentes: `cargaContenidoXML()` para cargar recursos de servidor, `onReadyState()` que es la función que se invoca cuando se recibe la respuesta del servidor y `defaultError()` que es la función que se emplea cuando no se ha definido de forma explícita una función responsable de manejar los posibles errores que se produzcan en la petición HTTP.

La función `defaultError()` muestra un mensaje de aviso del error producido y además muestra el valor de algunas de las propiedades de la petición HTTP:

```
defaultError:function() {
  alert("Se ha producido un error al obtener los datos"
    + "\n\nreadyState:" + this.req.readyState
    + "\nstatus: " + this.req.status
    + "\nheaders: " + this.req.getAllResponseHeaders());
}
```

En este caso, el objeto `this` se resuelve al objeto `net.cargaContenidos`, ya que es el objeto que contiene la función anónima que se está ejecutando.

Por otra parte, la función `onReadyState` es la encargada de gestionar la respuesta del servidor:

```
onReadyState: function() {
  var req = this.req;
  var ready = req.readyState;
  if(ready == net.READY_STATE_COMPLETE) {
    var httpStatus = req.status;
    if(httpStatus == 200 || httpStatus == 0) {
      this.onload.call(this);
    } else {
      this.onerror.call(this);
    }
  }
}
```

Tras comprobar que la respuesta del servidor está disponible y es correcta, se realiza la llamada a la función que realmente procesa la respuesta del servidor de acuerdo a las necesidades de la aplicación.

```
this.onload.call(this);
```

El objeto `this` se resuelve como `net.CargadorContenidos`, ya que es el objeto que contiene la función que se está ejecutando. Por tanto, `this.onload` es la referencia a la función que se ha definido como responsable de procesar la respuesta del servidor (se trata de una referencia a una función externa).

Normalmente, la función externa encargada de procesar la respuesta del servidor, requerirá acceder al objeto `XMLHttpRequest` que almacena la petición realizada al servidor. En otro caso, la función externa no será capaz de acceder al contenido devuelto por el servidor.

Como ya se vio en los capítulos anteriores, el método `call()` es uno de los métodos definidos para el objeto `Function()`, y por tanto disponible para todas las funciones de JavaScript. Empleando el método `call()` es posible obligar a una función a ejecutarse sobre un objeto concreto. En otras palabras, empleando el método `call()` sobre una función, es posible que dentro de esa función el objeto `this` se resuelva como el objeto pasado como parámetro en el método `call()`.

Así, la instrucción `this.onload.call(this);` se interpreta de la siguiente forma:

- El objeto `this` que se pasa como parámetro de `call()` se resuelve como el objeto `net.CargadorContenidos`.
- El objeto `this.onload` almacena una referencia a la función externa que se va a emplear para procesar la respuesta.
- El método `this.onload.call()` ejecuta la función cuya referencia se almacena en `this.onload`.
- La instrucción `this.onload.call(this);` permite ejecutar la función externa con el objeto `net.CargadorContenidos` accesible en el interior de la función mediante el objeto `this`.

Por último, el método `cargaContenidoXML` se encarga de enviar la petición HTTP y realizar la llamada a la función que procesa la respuesta:

```
cargaContenidoXML:function(url) {
  if(window.XMLHttpRequest) {
    this.req = new XMLHttpRequest();
  }
  else if(window.ActiveXObject) {
    this.req = new ActiveXObject("Microsoft.XMLHTTP");
  }
  if(this.req) {
    try {
      var loader=this;
      this.req.onreadystatechange = function() {
        loader.onReadyState.call(loader);
      }
```

```
      this.req.open('GET', url, true);
      this.req.send(null);
    } catch(err) {
      this.onerror.call(this);
    }
  }
}
```

En primer lugar, se obtiene una instancia del objeto `XMLHttpRequest` en función del tipo de navegador. Si se ha obtenido correctamente la instancia, se ejecutan las instrucciones más importantes del método `cargaContenidoXML`:

```
var loader = this;
this.req.onreadystatechange = function() {
  loader.onReadyState.call(loader);
}
this.req.open('GET', url, true);
this.req.send(null);
```

A continuación, se almacena la instancia del objeto actual (`this`) en la nueva variable `loader`. Una vez almacenada la instancia del objeto `net.cargadorContenidos`, se define la función encargada de procesar la respuesta del servidor. En la siguiente función anónima:

```
this.req.onreadystatechange = function() { ... }
```

En el interior de esa función, el objeto `this` no se resuelve en el objeto `net.CargadorContenidos`, por lo que no se puede emplear la siguiente instrucción:

```
this.req.onreadystatechange = function() {
  this.onReadyState.call(loader);
}
```

Sin embargo, desde el interior de esa función anónima si es posible acceder a las variables definidas en la función exterior que la engloba. Así, desde el interior de la función anónima sí que es posible acceder a la instancia del objeto `net.CargadorContenidos` que se almacenó anteriormente.

En el código anterior, no es obligatorio emplear la llamada al método `call()`. Se podría haber definido de la siguiente forma:

```
var loader=this;
this.req.onreadystatechange = function() {
  // loader.onReadyState.call(loader);
  loader.onReadyState();
}
```

En el interior de la función `onReadyState`, el objeto `this` se resuelve como `net.ContentLoader`, ya que se trata de un método definido en el prototipo del propio objeto.

Ejercicio 12

La página HTML proporcionada incluye una zona llamada *ticker* en la que se deben mostrar noticias generadas por el servidor. Añadir el código JavaScript necesario para:

1. De forma periódica cada cierto tiempo (por ejemplo cada segundo) se realiza una petición al servidor mediante AJAX y se muestra el contenido de la respuesta en la zona reservada para las noticias.
2. Además del contenido enviado por el servidor, se debe mostrar la hora en la que se ha recibido la respuesta.
3. Cuando se pulse el botón "Detener", la aplicación detiene las peticiones periódicas al servidor. Si se vuelve a pulsar sobre ese botón, se reanudan las peticiones periódicas.
4. Añadir la lógica de los botones "Anterior" y "Siguiente", que detienen las peticiones al servidor y permiten mostrar los contenidos anteriores o posteriores al que se muestra en ese momento.
5. Cuando se recibe una respuesta del servidor, se resalta visualmente la zona llamada *ticker*.
6. Modificar la aplicación para que se reutilice continuamente el mismo objeto `XMLHttpRequest` para hacer las diferentes peticiones.

6.5. Interacción con el servidor

6.5.1. Envío de parámetros con la petición HTTP

Hasta ahora, el objeto `XMLHttpRequest` se ha empleado para realizar peticiones HTTP sencillas. Sin embargo, las posibilidades que ofrece el objeto `XMLHttpRequest` son muy superiores, ya que también permite el envío de parámetros junto con la petición HTTP.

El objeto `XMLHttpRequest` puede enviar parámetros tanto con el método GET como con el método POST de HTTP. En ambos casos, los parámetros se envían como una serie de pares clave/valor concatenados por símbolos `&`. El siguiente ejemplo muestra una URL que envía parámetros al servidor mediante el método GET:

```
http://localhost/aplicacion?parametro1=valor1&parametro2=valor2&parametro3=valor3
```

La principal diferencia entre ambos métodos es que mediante el método POST los parámetros se envían en el cuerpo de la petición y mediante el método GET los parámetros se concatenan a la URL accedida. El método GET se utiliza cuando se accede a un recurso que depende de la información proporcionada por el usuario. El método POST se utiliza en operaciones que crean, borran o actualizan información.

Técnicamente, el método GET tiene un límite en la cantidad de datos que se pueden enviar. Si se intentan enviar más de 512 bytes mediante el método GET, el servidor devuelve un error con código `414` y mensaje `Request-URI Too Long` (*"La URI de la petición es demasiado larga"*).

Cuando se utiliza un elemento `<form>` de HTML, al pulsar sobre el botón de envío del formulario, se crea automáticamente la cadena de texto que contiene todos los parámetros que se envían al servidor. Sin embargo, el objeto `XMLHttpRequest` no dispone de esa posibilidad y la cadena que contiene los parámetros se debe construir manualmente.

A continuación se incluye un ejemplo del funcionamiento del envío de parámetros al servidor. Se trata de un formulario con tres campos de texto que se validan en el servidor mediante AJAX. El código HTML también incluye un elemento `<div>` vacío que se utiliza para mostrar la respuesta del servidor:

```
<form>
  <label for="fecha_nacimiento">Fecha de nacimiento:</label>
  <input type="text" id="fecha_nacimiento" name="fecha_nacimiento" /><br/>

  <label for="codigo_postal">Codigo postal:</label>
  <input type="text" id="codigo_postal" name="codigo_postal" /><br/>

  <label for="telefono">Telefono:</label>
  <input type="text" id="telefono" name="telefono" /><br/>

  <input type="button" value="Validar datos" />
</form>

<div id="respuesta"></div>
```

El código anterior produce la siguiente página:

Figura 7.3. Formulario de ejemplo

El código JavaScript necesario para realizar la validación de los datos en el servidor se muestra a continuación:

```
var READY_STATE_COMPLETE=4;
var peticion_http = null;

function inicializa_xhr() {
  if(window.XMLHttpRequest) {
    return new XMLHttpRequest();
  }
  else if(window.ActiveXObject) {
    return new ActiveXObject("Microsoft.XMLHTTP");
  }
}

function crea_query_string() {
  var fecha = document.getElementById("fecha_nacimiento");
  var cp = document.getElementById("codigo_postal");
  var telefono = document.getElementById("telefono");

  return "fecha_nacimiento=" + encodeURIComponent(fecha.value) +
         "&codigo_postal=" + encodeURIComponent(cp.value) +
         "&telefono=" + encodeURIComponent(telefono.value) +
         "&nocache=" + Math.random();
}

function valida() {
  peticion_http = inicializa_xhr();
  if(peticion_http) {
```

```
    peticion_http.onreadystatechange = procesaRespuesta;
    peticion_http.open("POST", "http://localhost/validaDatos.php", true);

    peticion_http.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    var query_string = crea_query_string();
    peticion_http.send(query_string);
  }
}

function procesaRespuesta() {
  if(peticion_http.readyState == READY_STATE_COMPLETE) {
    if(peticion_http.status == 200) {
      document.getElementById("respuesta").innerHTML = peticion_http.responseText;
    }
  }
}
```

La clave del ejemplo anterior se encuentra en estas dos líneas de código:

```
peticion_http.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
peticion_http.send(query_string);
```

En primer lugar, si no se establece la cabecera `Content-Type` correcta, el servidor descarta todos los datos enviados mediante el método POST. De esta forma, al programa que se ejecuta en el servidor no le llega ningún parámetro. Así, para enviar parámetros mediante el método POST, es obligatorio incluir la cabecera `Content-Type` mediante la siguiente instrucción:

```
peticion_http.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
```

Por otra parte, el método `send()` es el que se encarga de enviar los parámetros al servidor. En todos los ejemplos anteriores se utilizaba la instrucción `send(null)` para indicar que no se envían parámetros al servidor. Sin embargo, en este caso la petición si que va a enviar los parámetros.

Como ya se ha comentado, los parámetros se envían en forma de cadena de texto con las variables y sus valores concatenados mediante el símbolo `&` (esta cadena normalmente se conoce como *"query string"*). La cadena con los parámetros se construye manualmente, para lo cual se utiliza la función `crea_query_string()`:

```
function crea_query_string() {
  var fecha = document.getElementById("fecha_nacimiento");
  var cp = document.getElementById("codigo_postal");
  var telefono = document.getElementById("telefono");

  return "fecha_nacimiento=" + encodeURIComponent(fecha.value) +
         "&codigo_postal=" + encodeURIComponent(cp.value) +
         "&telefono=" + encodeURIComponent(telefono.value) +
         "&nocache=" + Math.random();
}
```

La función anterior obtiene el valor de todos los campos del formulario y los concatena junto con el nombre de cada parámetro para formar la cadena de texto que se envía al servidor. El uso de la función `encodeURIComponent()` es imprescindible para evitar problemas con algunos caracteres especiales.

La función `encodeURIComponent()` reemplaza todos los caracteres que no se pueden utilizar de forma directa en las URL por su representación hexadecimal. Las letras, números y los caracteres `- _ . ! ~ * ' ( )` no se modifican, pero todos los demás caracteres se sustituyen por su equivalente hexadecimal.

Las sustituciones más conocidas son las de los espacios en blanco por `%20`, y la del símbolo `&` por `%26`. Sin embargo, como se muestra en el siguiente ejemplo, también se sustituyen todos los acentos y cualquier otro carácter que no se puede incluir directamente en una URL:

```
var cadena = "cadena de texto";
var cadena_segura = encodeURIComponent(cadena);
// cadena_segura = "cadena%20de%20texto";

var cadena = "otra cadena & caracteres problemáticos / : =";
var cadena_segura = encodeURIComponent(cadena);
// cadena_segura =
"otra%20cadena%20%26%20caracteres%20problem%C3%A1ticos%20%2F%20%3A%20%3D";
```

JavaScript incluye una función contraria llamada `decodeURIComponent()` y que realiza la transformación inversa. Además, también existen las funciones `encodeURI()` y `decodeURI()` que codifican/decodifican una URL completa. La principal diferencia entre `encodeURIComponent()` y `encodeURI()` es que esta última no codifica los caracteres `; / ? : @ & = + $ , #`:

```
var cadena = "http://www.ejemplo.com/ruta1/index.php?parametro=valor con ñ y &";
var cadena_segura = encodeURIComponent(cadena);
// cadena_segura =
"http%3A%2F%2Fwww.ejemplo.com%2Fruta1%2Findex.php%3Fparametro%3Dvalor%20con%20%C3%B1%20y%20%26";

var cadena_segura = encodeURI(cadena);
// cadena_segura = "http://www.ejemplo.com/ruta1/
index.php?parametro=valor%20con%20%C3%B1%20y%20";
```

Por último, la función `crea_query_string()` añade al final de la cadena un parámetro llamado `nocache` y que contiene un número aleatorio (creado mediante el método `Math.random()`). Añadir un parámetro aleatorio adicional a las peticiones GET y POST es una de las estrategias más utilizadas para evitar problemas con la caché de los navegadores. Como cada petición varía al menos en el valor de uno de los parámetros, el navegador está obligado siempre a realizar la petición directamente al servidor y no utilizar su cache. A continuación se muestra un ejemplo de la *query string* creada por la función definida:

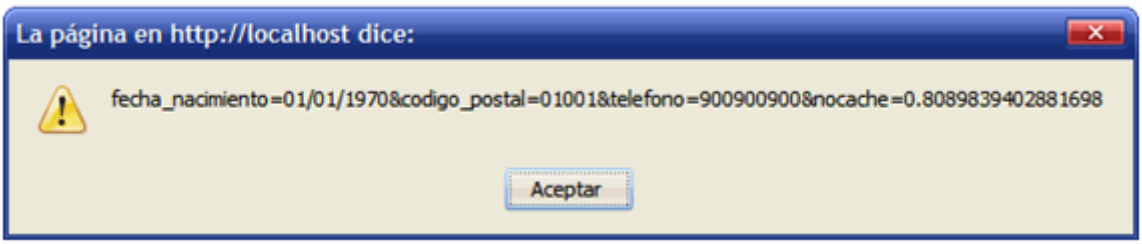

Figura 7.4. Query String creada para el formulario de ejemplo

En este ejemplo sencillo, el servidor simplemente devuelve el resultado de una supuesta validación de los datos enviados mediante AJAX:

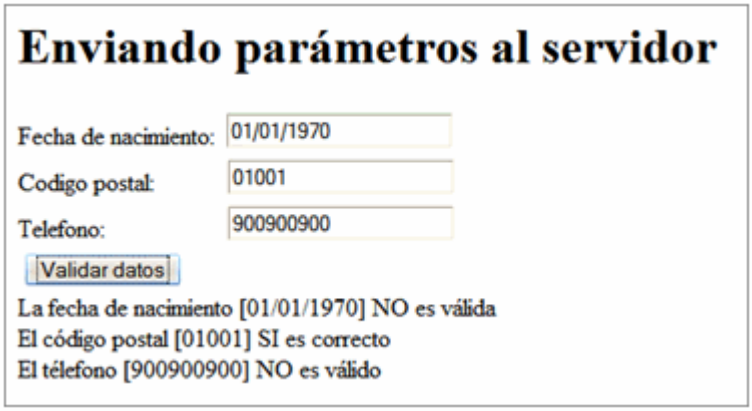

Figura 7.5. Mostrando el resultado devuelto por el servidor

En las aplicaciones reales, las validaciones de datos mediante AJAX sólo se utilizan en el caso de validaciones complejas que no se pueden realizar mediante el uso de código JavaScript básico. En general, las validaciones complejas requieren el uso de bases de datos: comprobar que un nombre de usuario no esté previamente registrado, comprobar que la localidad se corresponde con el código postal indicado, etc.

Ejercicio 13

Un ejemplo de validación compleja es la que consiste en comprobar si un nombre de usuario escogido está libre o ya lo utiliza otro usuario. Como es una validación que requiere el uso de una base de datos muy grande, no se puede realizar en el navegador del cliente. Utilizando las técnicas mostradas anteriormente y la página web que se proporciona:

1. Crear un script que compruebe con AJAX y la ayuda del servidor si el nombre escogido por el usuario está libre o no.
2. El script del servidor se llama `compruebaDisponibilidad.php` y el parámetro que contiene el nombre se llama `login`.
3. La respuesta del servidor es `"si"` o `"no"`, en función de si el nombre de usuario está libre y se puede utilizar o ya ha sido ocupado por otro usuario.
4. A partir de la respuesta del servidor, mostrar un mensaje al usuario indicando el resultado de la comprobación.

6.5.2. Refactorizando la utilidad net.CargadorContenidos

La utilidad diseñada anteriormente para la carga de contenidos y recursos almacenados en el servidor, solamente está preparada para realizar peticiones HTTP sencillas mediante GET. A continuación se refactoriza esa utilidad para que permita las peticiones POST y el envío de parámetros al servidor.

El primer cambio necesario es el de adaptar el constructor para que se puedan especificar los nuevos parámetros:

```
net.CargadorContenidos = function(url, funcion, funcionError, metodo, parametros,
contentType) {
```

Se han añadido tres nuevos parámetros: el método HTTP empleado, los parámetros que se envían al servidor junto con la petición y el valor de la cabecera `content-type`.

A continuación, se sustituye la instrucción `this.req.open('GET', url, true);` por esta otra:

```
this.req.open(metodo, url, true);
```

El siguiente paso es añadir (si así se indica) la cabecera `Content-Type` de la petición:

```
if(contentType) {
  this.req.setRequestHeader("Content-Type", contentType);
}
```

Por último, se sustituye la instrucción `this.req.send(null);` por esta otra:

```
this.req.send(parametros);
```

Así, el código completo de la solución refactorizada es el siguiente:

```
var net = new Object();

net.READY_STATE_UNINITIALIZED=0;
net.READY_STATE_LOADING=1;
net.READY_STATE_LOADED=2;
net.READY_STATE_INTERACTIVE=3;
net.READY_STATE_COMPLETE=4;

// Constructor
net.CargadorContenidos = function(url, funcion, funcionError, metodo, parametros,
contentType) {
  this.url = url;
  this.req = null;
  this.onload = funcion;
  this.onerror = (funcionError) ? funcionError : this.defaultError;
  this.cargaContenidoXML(url, metodo, parametros, contentType);
}

net.CargadorContenidos.prototype = {
  cargaContenidoXML: function(url, metodo, parametros, contentType) {
    if(window.XMLHttpRequest) {
      this.req = new XMLHttpRequest();
    }
    else if(window.ActiveXObject) {
      this.req = new ActiveXObject("Microsoft.XMLHTTP");
    }

    if(this.req) {
      try {
        var loader = this;
        this.req.onreadystatechange = function() {
          loader.onReadyState.call(loader);
        }
        this.req.open(metodo, url, true);
        if(contentType) {
          this.req.setRequestHeader("Content-Type", contentType);
        }
        this.req.send(parametros);
        } catch(err) {
          this.onerror.call(this);
        }
```

```
    }
  },

  onReadyState: function() {
    var req = this.req;
    var ready = req.readyState;
    if(ready == net.READY_STATE_COMPLETE) {
      var httpStatus = req.status;
      if(httpStatus == 200 || httpStatus == 0) {
        this.onload.call(this);
      }
      else {
        this.onerror.call(this);
      }
    }
  },

  defaultError: function() {
    alert("Se ha producido un error al obtener los datos"
      + "\n\nreadyState:" + this.req.readyState
      + "\nstatus: " + this.req.status
      + "\nheaders: " + this.req.getAllResponseHeaders());
  }
}
```

6.6. Aplicaciones complejas6

6.6.1. Envío de parámetros mediante XML

La flexibilidad del objeto `XMLHttpRequest` permite el envío de los parámetros por otros medios alternativos a la tradicional *query string*. De esta forma, si la aplicación del servidor así lo requeire, es posible realizar una petición al servidor enviando los parámetros en formato XML.

A continuación se modifica el ejemplo anterior para enviar los datos del usuario en forma de documento XML. En primer lugar, se modifica la llamada a la función que construye la *query string*:

```
function valida() {
  peticion_http = inicializa_xhr();
  if(peticion_http) {
    peticion_http.onreadystatechange = procesaRespuesta;
    peticion_http.open("POST", "http://localhost/validaDatos.php", true);
    var parametros_xml = crea_xml();
    peticion_http.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    peticion_http.send(parametros_xml);
  }
}
```

Seguidamente, se crea la función `crea_xml()` que se encarga de construir el documento XML que contiene los parámetros enviados al servidor:

```
function crea_xml() {
  var fecha = document.getElementById("fecha_nacimiento");
  var cp = document.getElementById("codigo_postal");
```

```
  var telefono = document.getElementById("telefono");

  var xml = "<parametros>";
  xml = xml + "<fecha_nacimiento>" + fecha.value + "<\/fecha_nacimiento>";
  xml = xml + "<codigo_postal>" + cp.value + "<\/codigo_postal>";
  xml = xml + "<telefono>" + telefono.value + "<\/telefono>";
  xml = xml + "<\/parametros>";
  return xml;
}
```

El código de la función anterior emplea el carácter \ en el cierre de todas las etiquetas XML. El motivo es que las etiquetas de cierre XML y HTML (al contrario que las etiquetas de apertura) se interpretan en el mismo lugar en el que se encuentran, por lo que si no se incluyen esos caracteres \ el código no validaría siguiendo el estándar XHTML de forma estricta.

El método `send()` del objeto `XMLHttpRequest` permite el envío de una cadena de texto y de un documento XML. Sin embargo, en el ejemplo anterior se ha optado por una solución intermedia: una cadena de texto que representa un documento XML. El motivo es que no existe a día de hoy un método robusto y que se pueda emplear en la mayoría de navegadores para la creación de documentos XML completos.

6.6.2. Procesando respuestas XML

Además del envío de parámetros en formato XML, el objeto `XMLHttpRequest` también permite la recepción de respuestas de servidor en formato XML. Una vez obtenida la respuesta del servidor mediante la propiedad `petición_http.responseXML`, es posible procesarla empleando los métodos DOM de manejo de documentos XML/HTML.

En este caso, se modifica la respuesta del servidor para que no sea un texto sencillo, sino que la respuesta esté definida mediante un documento XML:

```
<respuesta>
  <mensaje>...</mensaje>
  <parametros>
    <telefono>...</telefono>
    <codigo_postal>...</codigo_postal>
    <fecha_nacimiento>...</fecha_nacimiento>
  </parametros>
</respuesta>
```

La respuesta del servidor incluye un mensaje sobre el éxito o fracaso de la operación de validación de los parámetros y además incluye la lista completa de parámetros enviados al servidor.

La función encargada de procesar la respuesta del servidor se debe modificar por completo para tratar el nuevo tipo de respuesta recibida:

```
function procesaRespuesta() {
  if(peticion_http.readyState == READY_STATE_COMPLETE) {
    if(peticion_http.status == 200) {
      var documento_xml = peticion_http.responseXML;
      var root = documento_xml.getElementsByTagName("respuesta")[0];
```

```
      var mensajes = root.getElementsByTagName("mensaje")[0];
      var mensaje = mensajes.firstChild.nodeValue;

      var parametros = root.getElementsByTagName("parametros")[0];

      var telefono =
parametros.getElementsByTagName("telefono")[0].firstChild.nodeValue;
      var fecha_nacimiento =
parametros.getElementsByTagName("fecha_nacimiento")[0].firstChild.nodeValue;
      var codigo_postal =
parametros.getElementsByTagName("codigo_postal")[0].firstChild.nodeValue;

      document.getElementById("respuesta").innerHTML = mensaje + "<br/>" + "Fecha
nacimiento = " + fecha_nacimiento + "<br/>" + "Codigo postal = " + codigo_postal +
"<br/>" + "Telefono = " + telefono;
    }
  }
}
```

El primer cambio importante es el de obtener el contenido de la respuesta del servidor. Hasta ahora, siempre se utilizaba la propiedad `responseText`, que devuelve el texto simple que incluye la respuesta del servidor. Cuando se procesan respuestas en formato XML, se debe utilizar la propiedad `responseXML`.

El valor devuelto por `responseXML` es un documento XML que contiene la respuesta del servidor. Como se trata de un documento XML, es posible utilizar con sus contenidos todas las funciones DOM que se vieron en el capítulo correspondiente a DOM.

Aunque el manejo de repuestas XML es mucho más pesado y requiere el uso de numerosas funciones DOM, su utilización se hace imprescindible para procesar respuestas muy complejas o respuestas recibidas por otros sistemas que exportan sus respuestas internas a un formato estándar XML.

El mecanismo para obtener los datos varía mucho según cada documento XML, pero en general, se trata de obtener el valor almacenado en algunos elementos XML que a su vez pueden ser descendientes de otros elementos. Para obtener el primer elemento que se corresponde con una etiqueta XML, se utiliza la siguiente instrucción:

```
var elemento = root.getElementsByTagName("nombre_etiqueta")[0];
```

En este caso, se busca la primera etiqueta `<nombre_etiqueta>` que se encuentra dentro del elemento `root` (en este caso se trata de la raíz del documento XML). Para ello, se buscan todas las etiquetas `<nombre_etiqueta>` del documento y se obtiene la primera mediante `[0]`, que corresponde al primer elemento del array de elementos.

Una vez obtenido el elemento, para obtener su valor se debe acceder a su primer nodo hijo (que es el nodo de tipo texto que almacena el valor) y obtener la propiedad `nodeValue`, que es la propiedad que guarda el texto correspondiente al valor de la etiqueta:

```
var valor = elemento.firstChild.nodeValue;
```

Normalmente, las dos instrucciones anteriores se unen en una sola instrucción:

```
var tfno = parametros.getElementsByTagName("telefono")[0].firstChild.nodeValue;
```

Ejercicio 14

Normalmente, cuando se valida la disponibilidad de un nombre de usuario, se muestra una lista de valores alternativos en el caso de que el nombre elegido no esté disponible. Modificar el ejercicio de comprobación de disponibilidad de los nombres para que permita mostrar una serie de valores alternativos devueltos por el servidor.

El script del servidor se llama `compruebaDisponibilidadXML.php` y el parámetro que contiene el nombre se llama `login`. La respuesta del servidor es un documento XML con la siguiente estructura:

Si el nombre de usuario está libre:

```
<respuesta>
  <disponible>si</disponible>
</respuesta>
```

Si el nombre de usuario está ocupado:

```
<respuesta>
  <disponible>no</disponible>
  <alternativas>
    <login>...</login>
    <login>...</login>
    ...
    <login>...</login>
  </alternativas>
</respuesta>
```

Los nombres de usuario alternativos se deben mostrar en forma de lista de elementos (`<ul></ul>`).

Modificar la lista anterior para que muestre enlaces para cada uno de los nombres alternativos. Al pinchar sobre el enlace de un nombre alternativo, se copia en el cuadro de texto del login del usuario.

6.6.3. Parámetros y respuestas JSON

Aunque el formato XML está soportado por casi todos los lenguajes de programación, por muchas aplicaciones y es una tecnología madura y probada, en algunas ocasiones es más útil intercambiar información con el servidor en formato JSON.

JSON es un formato mucho más compacto y ligero que XML. Además, es mucho más fácil de procesar en el navegador del usuario. Afortunadamente, cada vez existen más utilidades para procesar y generar el formato JSON en los diferentes lenguajes de programación del servidor (PHP, Java, C#, etc.)

El ejemplo mostrado anteriormente para procesar las respuestas XML del servidor se puede reescribir utilizando respuestas JSON. En este caso, la respuesta que genera el servidor es mucho más concisa:

```
{
mensaje: "...",
```

```
parametros: {telefono: "...", codigo_postal: "...", fecha_nacimiento: "..." }
}
```

Considerando el nuevo formato de la respuesta, es necesario modificar la función que se encarga de procesar la respuesta del servidor:

```
function procesaRespuesta() {
  if(http_request.readyState == READY_STATE_COMPLETE) {
  if(http_request.status == 200) {
    var respuesta_json = http_request.responseText;
    var objeto_json = eval("("+respuesta_json+")");

    var mensaje = objeto_json.mensaje;

    var telefono = objeto_json.parametros.telefono;
    var fecha_nacimiento = objeto_json.parametros.fecha_nacimiento;
    var codigo_postal = objeto_json.parametros.codigo_postal;

    document.getElementById("respuesta").innerHTML = mensaje + "<br>" + "Fecha
nacimiento = " + fecha_nacimiento + "<br>" + "Codigo postal = " + codigo_postal +
"<br>" + "Telefono = " + telefono;
    }
  }
}
```

La respuesta JSON del servidor se obtiene mediante la propiedad `responseText`:

```
var respuesta_json = http_request.responseText;
```

Sin embargo, esta propiedad solamente devuelve la respuesta del servidor en forma de cadena de texto. Para trabajar con el código JSON devuelto, se debe transformar esa cadena de texto en un objeto JSON. La forma más sencilla de realizar esa conversión es mediante la función `eval()`, en la que deben añadirse paréntesis al principio y al final para realizar la evaluación de forma correcta:

```
var objeto_json = eval("("+respuesta_json+")");
```

Una vez realizada la transformación, el objeto JSON ya permite acceder a sus métodos y propiedades mediante la notación de puntos tradicional. Comparado con las respuestas XML, este procedimiento permite acceder a la información devuelta por el servidor de forma mucho más simple:

```
// Con JSON
var fecha_nacimiento = objeto_json.parametros.fecha_nacimiento;
```

```
// Con XML
var parametros = root.getElementsByTagName("parametros")[0];
var fecha_nacimiento =
parametros.getElementsByTagName("fecha_nacimiento")[0].firstChild.nodeValue;
```

También es posible el envío de los parámetros en formato JSON. Sin embargo, no es una tarea tan sencilla como la creación de un documento XML. Así, se han diseñado utilidades específicas para la transformación de objetos JavaScript a cadenas de texto que representan el objeto en formato JSON. Esta librería se puede descargar desde el sitio web www.json.org.

Para emplearla, se añade la referencia en el código de la página:

```
<script type="text/javascript" src="json.js"></script>
```

Una vez referenciada la librería, se emplea el método `stringify` para realizar la transformación:

```
var objeto_json = JSON.stringify(objeto);
```

Además de las librerías para JavaScript, están disponibles otras librerías para muchos otros lenguajes de programación habituales. Empleando la librería desarrollada para Java, es posible procesar la petición JSON realizada por un cliente:

```
import org.json.JSONObject;
...
String cadena_json = "{propiedad: valor, codigo_postal: otro_valor}";
JSONObject objeto_json = new JSONObject(cadena_json);
String codigo_postal = objeto_json.getString("codigo_postal");
```

Ejercicio 15

Rehacer el ejercicio 14 para procesar respuestas del servidor en formato JSON. Los cambios producidos son:

1) El script del servidor se llama `compruebaDisponibilidadJSON.php` y el parámetro que contiene el nombre se llama `login`.

2) La respuesta del servidor es un objeto JSON con la siguiente estructura:

El nombre de usuario está libre:

```
{ disponible: "si" }
```

El nombre de usuario está ocupado:

```
{ disponible: "no", alternativas: ["...", "...", ..., "..."] }
```

6.7. Seguridad

La ejecución de aplicaciones JavaScript puede suponer un riesgo para el usuario que permite su ejecución. Por este motivo, los navegadores restringen la ejecución de todo código JavaScript a un entorno de ejecución limitado y prácticamente sin recursos ni permisos para realizar tareas básicas.

Las aplicaciones JavaScript no pueden leer ni escribir ningún archivo del sistema en el que se ejecutan. Tampoco pueden establecer conexiones de red con dominios distintos al dominio en el que se aloja la aplicación JavaScript. Además, un script sólo puede cerrar aquellas ventanas de navegador que ha abierto ese script.

La restricción del acceso a diferentes dominios es más restrictiva de lo que en principio puede parecer. El problema es que los navegadores emplean un método demasiado simple para diferenciar entre dos dominios ya que no permiten ni subdominios ni otros protocolos ni otros puertos.

Por ejemplo, si el código JavaScript se descarga desde la siguiente URL: http://www.ejemplo.com/scripts/codigo.js, las funciones y métodos incluidos en ese código no pueden acceder a los recursos contenidos en los siguientes archivos:

- http://www.ejemplo.com:8080/scripts/codigo2.js
- https://www.ejemplo.com/scripts/codigo2.js
- http://192.168.0.1/scripts/codigo2.js
- http://scripts.ejemplo.com/codigo2.js

Afortunadamente, existe una forma de solucionar parcialmente el problema del acceso a recursos no originados exactamente en el mismo dominio. La solución se basa en establecer el valor de la propiedad `document.domain`

Así, si el código alojado en http://www.ejemplo.com/scritps/codigo1.js establece la variable `document.domain = "ejemplo.com";` y por otra parte, el código alojado en http://scripts.ejemplo.com/codigo2.js establece la variable `document.domain = "ejemplo.com";` los recursos de ambos códigos pueden interactuar entre sí.

La propiedad `document.domain` se emplea para permitir el acceso entre subdominios del dominio principal de la aplicación. Evidentemente, los navegadores no permiten establecer cualquier valor en esa propiedad, por lo que sólo se puede indicar un valor que corresponda a una parte del subdominio completo donde se encuentra el script.