

# Comunicación con clientes ricos y AJAX

## Índice

1 Servlets y clientes ricos.....	2
1.1 HTTP Tunneling.....	2
1.2 Paso de objetos.....	2
1.3 Envío de datos en la petición.....	3
2 Servlets y Javascript: AJAX.....	4
2.1 Petición HTTP desde Javascript.....	5
2.2 Lectura de la respuesta.....	6
2.3 Aplicaciones de AJAX.....	7
2.4 Frameworks AJAX: jMaki.....	7

## 1. Servlets y clientes ricos

Vamos a ver a continuación cómo establecer una comunicación entre clientes ricos y *servlets* utilizando el protocolo HTTP. Este protocolo es la interfaz que nos ofrecen los servidores para comunicarnos con los componentes web que contengan, por lo tanto podremos utilizarlo desde un cliente rico Java para comunicarnos con los *servlets* o JSP alojados en el servidor. Además tiene la ventaja de que al ser un protocolo web estándar nos va a permitir realizar las comunicaciones sin problemas, ya que los firewalls intermedios normalmente dejarán pasar las conexiones de este tipo. Aunque los navegadores web normalmente utilizan este protocolo simplemente para obtener contenido HTML, a continuación veremos que podemos utilizarlo para intercambiar cualquier tipo de contenido.

### 1.1. HTTP Tunneling

Si queremos establecer una comunicación entre un cliente rico y un *servlet*, deberemos establecer desde el cliente una conexión con la URL en la que está publicado el *servlet*. La información se intercambiará a través de la conexión HTTP con dicha URL, y esto es lo que se denomina *HTTP tunneling*. Lo que haremos será construir en el cliente un objeto URL con la URL del *servlet* que queramos utilizar, y abrir una conexión con dicha URL para realizar la petición y recibir el resultado generado por el *servlet*. Al realizar la conexión es conveniente indicar que no utilice la caché, para que siempre se solicite al servidor que genere nuevamente el contenido:

```
URLConnection con = url.openConnection();
con.setUseCaches(false);
```

Ahora podemos establecer las cabeceras de la petición que queramos. De esta forma podemos proporcionar información sencilla mediante pares (*clave, valor*):

```
con.setRequestProperty("header", "valor");
```

Entonces ya podemos crear la conexión con la URL:

```
InputStream in = con.getInputStream();
```

Es en este momento cuando se establece la conexión con el *servlet* y éste genera el resultado. Mediante el flujo de entrada que obtenemos podremos leer desde nuestro applet el contenido generado. Este flujo de entrada podrá ser de cualquier tipo, según el tipo de contenido generado por el *servlet*. Podemos leer directamente la secuencia de bytes, transformarlo a un flujo de caracteres, o bien a un flujo de procesamiento más complejo (por ejemplo `DataInputStream`).

### 1.2. Paso de objetos

También podremos hacer que el servlet nos devuelva objetos. Si el servlet serializa un objeto y lo escribe en la respuesta (utilizando un `ObjectOutputStream`), desde el cliente podremos utilizar un `ObjectInputStream` para leer dicho objeto:

```
ObjectInputStream ois = new ObjectInputStream(in);
MiClase obj = (MiClase)ois.readObject();
```

Por último, una vez leído todo el contenido, cerraremos el flujo de entrada:

```
in.close();
```

Para que el servlet devuelva un objeto deberá especificar como tipo del contenido generado el siguiente tipo MIME:

```
application/x-java-serialized-object
```

Y lo único que tendrá que hacer entonces será obtener un `ObjectOutputStream` a partir del flujo de salida de la respuesta, y escribir el objeto en él:

```
MiClase result = generaObjetoResultante();
response.setContentType("application/x-java-serialized-object");
ObjectOutputStream oos = new
ObjectOutputStream(response.getOutputStream());
oos.writeObject(result);
oos.flush();
```

Hemos visto que el applet podrá realizar una petición al servlet, y como resultado podrá devolvernos cualquier tipo de datos, pudiendo incluso enviarnos un objeto. A continuación veremos que también es posible enviar datos desde el cliente al servidor utilizando esta petición HTTP.

### 1.3. Envío de datos en la petición

Para poder enviar información al servidor podremos utilizar el bloque de contenido del mensajes de petición HTTP. Para ello la conexión que abrimos con la URL debe ser de lectura/escritura:

```
URLConnection con = url.openConnection();
con.setUseCaches(false);
con.setDoOutput(true);
```

A continuación crearemos un array de bytes donde escribiremos la información que enviaremos al servidor:

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
```

Podemos usar distintos tipos de flujos sobre este objeto, según vayamos a escribir caracteres (`PrintWriter`), datos codificados en binario (`DataOutputStream`) u objetos serializados (`ObjectOutputStream`), por ejemplo.

Una vez escrito este contenido, deberemos establecer una cabecera con la longitud de

dicho contenido:

```
con.setRequestProperty("Content-Length",
String.valueOf(baos.size()));
```

Por cuestión de compatibilidad entre navegadores, será conveniente establecer manualmente el tipo de contenido:

```
connection.setRequestProperty("Content-Type",
"application/x-java-serialized-object");
```

Tipos MIME que podemos utilizar comunmente para intercambiar información son:

application/x-www-form-urlencoded	Se envían los datos codificados de la misma forma en la que son codificados por un formulario HTML con método POST.
text/plain	Se envía como contenido texto ASCII.
application/octet-stream	Se envía como contenido datos binarios. Dentro de la secuencia de bytes podremos codificar la información como queramos. Por ejemplo, podemos codificar de forma binaria un objeto serializado, utilizando un <code>DataOutputStream</code> .
application/x-java-serialized-object	Se envía como contenido un objeto Java serializado.

En este momento ya podremos enviar los datos al flujo de salida de la conexión:

```
baos.writeTo(connection.getOutputStream());
```

Una vez hecho esto ya podemos obtener el flujo de entrada de la conexión, para realizar la petición y obtener contenido que genera el servlet como respuesta, de la misma forma que lo hacíamos en el caso anterior.

#### Nota:

Si no abrimos el flujo de entrada (ni intentamos consultar ninguna otra propiedad de las respuesta que genera el *servlet*), la petición no se llegará a realizar y por lo tanto el *servlet* no recibirá los datos que hayamos escrito para ser enviados)

## 2. Servlets y Javascript: AJAX

AJAX (*Asynchronous Javascript And Xml*) no es una tecnología, sino una técnica de desarrollo que combina una serie de tecnologías para hacer que un navegador en el lado del cliente pueda obtener información del servidor sin tener que recargar la página que se esté mostrando. Las tecnologías que se utilizan son:

- **HTML** y **CSS** para presentar la información.

- **XML** para intercambiar información con el servidor.
- **DOM** y **Javascript** para analizar esta información.

Esta técnica consistirá en hacer una petición HTTP utilizando el objeto `XMLHttpRequest` de *Javascript*. Con esto obtendremos del servidor un documento XML con la información requerida, que podremos analizar con el DOM (también de *Javascript*) Una vez tengamos los datos obtenidos del servidor, podemos actualizarlos en el documento HTML utilizando nuevamente el DOM.

## 2.1. Petición HTTP desde Javascript

Lo primero que deberemos hacer es obtener el objeto `XMLHttpRequest` que nos permita hacer una petición HTTP desde *Javascript*. Deberemos distinguir entre IE y Firefox:

```
function verMensajes() {
  //Preparar objeto para lanzar petición
  if (window.XMLHttpRequest) { //Firefox, etc
    petición = new XMLHttpRequest();
  } else if (window.ActiveXObject) { //Explorer
    petición = new ActiveXObject("Microsoft.XMLHTTP");
  }
}
```

Una vez obtenido este objeto, deberemos especificar a qué función se deberá llamar cuando se reciba la respuesta HTTP del servidor. Especificaremos esta función *callback* de la siguiente forma:

```
//a quien llamar cuando el servidor responda
petición.onreadystatechange = atenderPetición;
```

Por último, efectuaremos la petición propiamente dicha, especificando el tipo de petición, la URL a la que vamos a conectar, y si la llamada es asíncrona. Si no fuese asíncrona, el programa quedaría bloqueado hasta recibir la respuesta. Por lo tanto, deberemos poner este último parámetro siempre a `true` para evitar que la web queda bloqueada mientras se obtiene la información:

```
//lanzar la petición propiamente dicha
petición.open("GET",
"http://localhost:7001/ChatXml/chat/listaMensajesXml.jsp", true);
petición.send(null);
}
```

NOTA: Por motivo de seguridad, sólo se permitirá hacer peticiones mediante el objeto `XMLHttpRequest` al servidor del que se ha descargado esta página.

Una vez se reciba la respuesta, se llamará a la función especificada como *callback*. Deberemos fijarnos en el estado en el que se encuentra la petición:

0	No inicializada
1	Cargando
2	Cargada

3	Interactiva
4	Completada

El significado de cada uno de estos estados es:

- **No inicializada.** Se ha creado el objeto de la petición HTTP pero ésta todavía no se ha efectuado.
- **Cargando.** Se ha realizado la petición HTTP y se está esperando a recibir la respuesta.
- **Cargada.** La respuesta HTTP se ha recibido, pero todavía no se puede acceder al modelo DOM que representa al documento obtenido.
- **Interactiva.** El modelo DOM se ha empezado a construir pero todavía no está completo. Podremos consultar las partes del modelo que ya se hayan construido.
- **Completada.** El modelo DOM del documento obtenido está completo.

Sólo nos interesará el caso en el que la petición haya sido completada. Además también deberemos comprobar que la respuesta del servidor no nos haya devuelto un código de error. Si todo ha ido bien, el estado de la respuesta debería ser 200 OK:

```
function atenderPetición() {
    if (petición.readyState == 4) {
        //analizar respuesta
        if (petición.status!=200) {
            alert("ha habido un error");
            return;
        }
    }
}
```

En este punto tendremos el contenido de la respuesta en las propiedades `petición.responseText` y `petición.responseXML`. La primera de ellas nos ofrecerá la respuesta como una cadena de texto, mientras que la segunda es un objeto del tipo `XMLDocument` al que podremos acceder mediante el DOM.

## 2.2. Lectura de la respuesta

Normalmente utilizaremos `responseText` cuando queramos incluir directamente en el HTML el contenido devuelto por el servidor, y `responseXML` cuando estemos intercambiando estructuras de datos.

A continuación se muestra un ejemplo en el que se analiza el documento XML obtenido para incluir la información formateada en el HTML:

```
//mostrar mensajes
var areaMensajes = document.getElementById("mensajesChat");
var textoHTML = "";

docXml = petición.responseXML;
var raiz = docXml.getElementsByTagName('mensajes');
mensajes = raiz[0].getElementsByTagName('mensaje');
for(i=0;i<mensajes.length;i++) {
    var nick =
```

```
mensajes[i].getElementsByTagName('nick').item(0).firstChild.data;
    var texto =
mensajes[i].getElementsByTagName('texto').item(0).firstChild.data;
    textoHTML += "<strong>&lt;" + nick +
"&gt;</strong> " + texto + "<br/>";
    }

    areaMensajes.innerHTML = textoHTML;
}
}
```

En el caso de este ejemplo, para intercambiar la información estaríamos utilizando un documento XML con el siguiente formato:

```
<mensajes>
<mensaje>
  <nick>Ana</nick>
  <texto>Hola</texto>
</mensaje>
<mensaje>
  <nick>Jose</nick>
  <texto>Que tal?</texto>
</mensaje>
</mensajes>
```

Podremos invocar la función que efectúa la petición HTTP de diferentes formas. Podemos utilizar un temporizador para hacer que la información se actualice periódicamente, o bien podemos hacer que se invoque como respuesta a algún evento del usuario (por ejemplo cuando pulse un botón).

Como alternativa a XML, resulta de especial interés el formato JSON. Se trata de un lenguaje ligero de intercambio de información, que puede utilizarse en lugar de XML (que resulta considerablemente más pesado) para estas aplicaciones AJAX. De hecho, en Javascript puede leerse este tipo de formato simplemente utilizando el método `eval()`.

### 2.3. Aplicaciones de AJAX

---

Esta técnica tiene numerosas aplicaciones, como por ejemplo:

- *Validación de formularios*: Podemos validar la información introducida en un formulario en el lado del servidor antes de enviarlo. Por ejemplo, al registrarnos en una web podemos comprobar si un *login* ya está utilizado.
- *Autocompletar campos*: Mientras escribimos en un campo de un formulario, podemos consultar en el servidor las coincidencias con lo que llevamos escrito para que nos muestre una lista con las posibilidades existentes.
- *Mostrar datos actualizados*: Podemos mantener información de última hora en la web sin tener que recargar. Por ejemplo, podemos ver el marcador de un partido de fútbol que se está jugando en este momento, la clasificación de la Fórmula 1, o información de la bolsa.

## 2.4. Frameworks AJAX: jMaki

---

Hemos visto hasta ahora los fundamentos de AJAX y como utilizar esta técnica programando directamente JavaScript a bajo nivel. Sin embargo, esta forma de trabajar puede resultar compleja, sobretodo si no se cuenta con suficiente experiencia en JavaScript, y puede dar lugar a grandes cantidades de código mal estructurado y difícil de entender, haciendo la aplicación poco mantenible.

Por este motivo, y debido al éxito que AJAX ha cosechado, aparecen multitud de librerías y *frameworks* destinados a facilitarnos el uso de esta técnica en nuestras aplicaciones. Tenemos por ejemplo librerías JavaScript como Scriptaculous o DOJO, que nos ofrecen una serie de componentes de la interfaz de usuario predefinidos, con los que añadir contenido dinámico a nuestras webs (AJAX, animaciones, etc). También podemos encontrar componentes basados en AJAX proporcionados por diferentes proveedores, como Google (mapas, búsqueda) o Yahoo, que podemos introducir en nuestras páginas utilizando las librerías necesarias.

Aquí encontramos el problema de que cada proveedor ofrece su propia API JavaScript para utilizar los componentes AJAX que proporciona, lo cual nos forzaría a aprender cada una de estas APIs para poder introducir los correspondientes componentes en nuestra web. Para evitar este problema aparecen *frameworks* como jMaki que tratan de definir un modelo único de acceso a los diferentes tipos de componentes.

Pasaremos a continuación a estudiar jMaki con más detalle. Este *framework* está soportado por los principales entornos de desarrollo (Eclipse, NetBeans), y nos permitirá introducir en nuestras páginas componentes AJAX proporcionados por diferentes proveedores (Google, Yahoo, DOJO, etc) simplemente mediante el uso de *tags* en nuestros JSPs.

### 2.4.1. Instalación en Eclipse

---

Vamos a comenzar viendo como instalar en Eclipse el *plugin* que nos permita trabajar con jMaki.

Lo primero que deberemos hacer es ir al menú *Help > Software Updates ....* Aquí añadiremos el repositorio en el que se encuentra este *plugin*:

```
https://ajax.dev.java.net/eclipse
```

Una vez hecho esto, desplegaremos el contenido de este repositorio, seleccionaremos el elemento *jMaki Ajax*, y pulsamos sobre el botón *Install ...*

Con esto habremos instalado el *plugin*. Ahora cuando estemos editando un JSP, abriremos la ventana *Snippets* y en ella veremos la paleta de componentes (*widgets*) jMaki que podemos incorporar en nuestra web, simplemente arrastrándolos sobre el lugar del JSP en el que queramos incluirlo.



## 2.4.2. Componentes de los widgets

A la hora de desarrollar nuestra web simplemente utilizaremos etiquetas jMaki en nuestros JSPs. Cuando esta página se genere, dichas etiquetas se transformarán en elementos HTML y llamadas a librerías JavaScript que puedan ser correctamente interpretadas por el navegador. La forma de renderizar estos *widgets* se define mediante 3 ficheros en jMaki. Estos ficheros son:

- `component.html`: Define una plantilla del código HTML que compondrá el *widget* en la página.
- `component.js`: Define una clase JavaScript que hará de adaptador entre la llamada al *widget* desde la *taglib* y la librería concreta que implementa las funcionalidades del *widget* que estemos utilizando. Esta capa es la que nos va a permitir tener una interfaz de acceso única a *widgets* proporcionados por diferentes proveedores.
- `component.css`: Incluye los componentes de estilo necesarios para mostrar el *widget*.

Cada *widget* estará definido en un directorio distinto, que contendrá, por lo menos, estos tres ficheros. El directorio asociado a cada *widget* corresponde a su nombre, de forma similar a lo que ocurre con los paquetes en Java, y todos estos directorios estarán bajo la carpeta `resources`. Por ejemplo, si un *widget* se llama `google.map`, estará en un directorio `/resources/google/map`, dentro de nuestra carpeta web.

## 2.4.3. Tipos de widgets

Dentro de jMaki podemos encontrar diversos tipos de *widgets*. Podemos encontrar diferentes *widgets* de un mismo tipo, muchas veces proporcionados por diferentes proveedores. A continuación mostramos una tabla con los modelos generales de *widgets* existentes y una serie de *widgets* concretos de cada tipo.

Tipo	Widgets
Menú	Yahoo Menu, jMaki Menu, jMaki Tab Menu, jMaki Accordion Menu
Tabla	Yahoo Datatable, Dojo Table
Árbol	Yahoo Tree, Dojo Tree
Desplegable	Dojo Combobox
Pestañas	jMaki Dynamic Container, Dojo Accordion, Dojo Tabbedview, Yahoo Tabbedview, Spry Accordion
Fisheye	Dojo Fisheye
Cajón	Dojo Drawer
Mapas	Yahoo Map, Google Map
Validación	Spry validation

Todos los *widgets* de jMaki utilizan para su configuración una serie de atributos comunes. Estos atributos son:

- `id`: Identificador de un elemento, que podremos utilizar para referenciarlo posteriormente.
- `label`: Etiqueta que se mostrará en el *widget*, como por ejemplo el título que figurará en las pestañas, elementos de menús, nodos de árbol o columnas de una tabla.
- `href`: Sirve para indicar una URL que haga de enlace, y que al pulsar sobre ella se acceda a la dirección a la que apunta.
- `include`: Con este atributo especificaremos también una URL, pero en lugar de poner el enlace lo que se hará será incluir el contenido de dicha URL en el contenedor correspondiente.
- `action`: Nos permite establecer una comunicación entre elementos de diferentes *widgets* (por ejemplo podríamos añadir una acción en un botón para que se inserten filas en una tabla, o hacer que al pulsar sobre un icono se abra una determinada pestaña).
- `service`: Indica la URL a la que se conectará el *widget* mediante AJAX para obtener o enviar la información necesaria.

Hemos mostrado los atributos generales, pero cada tipo de *widget* tiene además su propio modelo de datos. A continuación pasaremos a ver algunos ejemplos concretos. Para obtener más información sobre el modelo de datos concreto de cada tipo de *widget* y sobre como establecer una comunicación entre ellos, podemos consultar la documentación en:

<https://ajax.dev.java.net/>

#### 2.4.4. Ejemplo: Múltiples pestañas

Vamos a ver en primer lugar un ejemplo de como crear una página con varias pestañas, cada una de ellas con un contenido diferente. Utilizaremos para ello el *widget* `Tabbedview` de Yahoo. Arrastraremos este widget sobre nuestro JSP y modificaremos el texto generado de la siguiente forma:

```
<a:widget name="yahoo.tabbedview"
  value="{items:[
    {label : 'A', content : 'Contenido de la pestaña A'},
    {id : 'tabB', label : 'B', include : 'pagB.jsp ',
lazyLoad : true },
    {label : 'C', include : 'pagC.jsp', selected : true,
iframe: true}
  ]
}" />
```

Deberemos crear dos nuevos JSPs: `pagB.jsp` y `pagC.jsp`, con el contenido que tendrán las pestañas B y C respectivamente. Hemos de hacer notar el uso de los atributos `label` e `include`. Con el primero de ellos indicamos el título de cada pestaña. El segundo nos permite indicar el contenido que tendrá, cogiendo este contenido de otro JSP.

### 2.4.5. Ejemplo: Google Maps

Vamos a ver ahora como podemos incluir mapas dinámicos en nuestra aplicación (Google Maps o Yahoo Maps). Para ello arrastraremos el *widget* Maps de Google sobre nuestro JSP. Veremos un código como el siguiente:

```
<a:widget name="google.map"
  args="{ centerLat : 37.4041960114344,
         centerLon : -122.008194923401 }" />
```

Podemos ver que con esto podemos incluir de forma sencilla un mapa que indique una determinada dirección, especificando las coordenadas de dicha dirección como parámetro. Pero también podemos utilizar el mapa junto a otro *widget* que nos permita obtener las coordenadas a partir de la dirección del lugar. Para hacer esto arrastraremos sobre el JSP el *widget* Geocoder de Yahoo:

```
<a:widget name="yahoo.geocoder" service="/xhp" />
```

En este caso podemos ver que este *widget* necesita un parámetro *service* en el que se indica el servicio al que deberá conectar el *widget* mediante AJAX para obtener las coordenadas a partir del texto de la dirección. Esta conexión debe hacerse a través de un proxy incluido en jMaki, que deberemos mapear como servlet en nuestra aplicación.

Este servlet está ya implementado en la clase `XmlHttpProxyServlet`, por lo que en Eclipse deberemos indicar que cree un nuevo servlet, pero utilizando una clase ya existente (esto lo haremos en el mismo asistente de crear nuevo servlet). Mapearemos este servlet a la dirección que hemos especificado en el *widget*: `/xhp`. En el `web.xml` quedará de la siguiente forma:

```
<servlet>
  <description></description>
  <display-name>XmlHttpProxyServlet</display-name>
  <servlet-name>XmlHttpProxyServlet</servlet-name>
<servlet-class>com.sun.jmaki.services.XmlHttpProxyServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>XmlHttpProxyServlet</servlet-name>
  <url-pattern>/xhp</url-pattern>
</servlet-mapping>
```

Si ahora probamos la página, veremos que al introducir una dirección en el campo de texto del Geocoder, se nos muestra esta dirección en el mapa.

### 2.4.6. Ejemplo: Autocompletar texto

Para terminar, vamos a ver un ejemplo de como autocompletar texto con AJAX utilizando jMaki. Tendremos un campo de texto, en el que conforme escribamos nos mostrará las posibles coincidencias con las que podríamos autocompletar. Vamos a utilizar el *widget* Autocompleter de Scriptaculous. Arrastramos el widget sobre nuestro JSP y lo

modificamos como se muestra a continuación:

```
<a:widget name="scriptaculous.autocompleter" service="lista.jsp" />
```

En este caso, en `service` especificamos la página de la que sacará las opciones con las que autocompletar. Estas opciones se definirán como una lista HTML. Por ejemplo, podríamos definir `lista.jsp` como se muestra a continuación:

```
<ul>
<li>Seleccionar <span class="selectme">A</span></li>
<li>Seleccionar <span class="selectme">B</span></li>
</ul>
```

Es posible que esto no nos funcione debido a un *bug* en algunas versiones de jMaki. Para corregirlo deberemos abrir el fichero `component.js` correspondiente a este *widget* (estará en el directorio `/resources/scriptaculous/autocompleter`). En la línea 12 debe declarar una variable `service`, pero por error podría aparecer `servicei`. Si es así deberemos corregirlo y dejarlo como se muestra a continuación:

```
var service = wargs.service;
```

De especial interés es también el fichero `component.htm`. Aquí se especifica la plantilla utilizada para mostrar este componente en el HTML. Podemos ver que tiene un campo de texto, donde introduciremos el contenido a autocompletar. Este *widget* envía al servicio un parámetro con el texto introducido hasta el momento, para que éste pueda devolvernos las coincidencias con ese texto. Este parámetro recibe como nombre el nombre que le hayamos dado al campo de texto. Sin embargo, podemos ver que el campo de texto no tiene asignado ningún nombre (falta el atributo `name`), por lo que será buena idea añadirlo:

```
<input id="{uuid}" name="texto" type="text"/>
```

Con esto, nuestra página `lista.jsp` recibirá un parámetro `texto` con el texto introducido hasta el momento. Entonces tendrá que buscar las coincidencias con ese texto (por ejemplo elementos en nuestra base de datos que comiencen de esa manera), y devolverlas formateadas como una lista HTML.

