

AJAX HACKS

Tips & Tools for Creating Responsive Web Sites



O'REILLY®

Bruce W. Perry

Ajax Hacks™

by Bruce Perry

Copyright © 2006 O'Reilly Media, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North,
Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

Editors: Simon St.Laurent

Production Editor: Mary Anne Weeks Mayo

Copyeditor: Rachel Wheeler

Indexer: XXX

Cover Designer: Hanna Dyer

Interior Designer: David Futato

Illustrators: Robert Romano, Jessamyn

Read, and Lesley Borash

Printing History:

March 2006:

First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. The *Hacks* series designations, *Baseball Hacks*, the image of an umpire's indicator, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Small print: The technologies discussed in this publication, the limitations on these technologies that technology and content owners seek to impose, and the laws actually limiting the use of these technologies are constantly changing. Thus, some of the hacks described in this publication may not work, may cause unintended harm to systems on which they are used, or may not be consistent with applicable user agreements. Your use of these hacks is at your own risk, and O'Reilly Media, Inc. disclaims responsibility for any damage or expense resulting from their use. In any event, you should take care that your use of these hacks does not violate any applicable laws, including copyright laws.



This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 0-596-10169-4

[M]



HACK

#1

Detect Browser Compatibility with the Request Object

Use JavaScript to set up Microsoft's and the Mozilla-based browsers' different request objects.

Browser compatibility is an important consideration. You have to make sure the “engine” behind Ajax’s server handshake is properly constructed, but you can never predict which browsers your users will favor.

The programming tool that allows Ajax applications to make HTTP requests to a server is an object that you can use from within JavaScript code. In the world of Firefox and Netscape (as well as Safari and Opera), this object is named `XMLHttpRequest`. However, continuing with the tradition established by IE 5.0, recent vintages of Internet Explorer implement the software as an ActiveX object named `Microsoft.XMLHTTP` or `Mxml2.XMLHTTP`.



`Microsoft.XMLHTTP` and `Mxml2.XMLHTTP` refer to different versions of software components that are a part of Microsoft XML Core Services (MSXML). Here’s what our contributing IE expert says on this matter:

“If you use `Microsoft.XMLHTTP`, the `ActiveXObject` wrapper will try to initialize the last known good version of the object that has this program (or “prog”) ID. This object, in theory, could be MSXML 1.0, but almost no one these days has that version because it has been updated via Windows Update, IE 6, or another means. MSXML 1.0 was very short-lived. If you use `MSXML2.XMLHTTP`, that signifies to the wrapper to use at least MSXML 2.0 libraries. Most developers do not need to use a specific version of MSXML, such as `MSXML2.XMLHTTP.4.0` or `MSXML2.XMLHTTP.5.0`.”

Although Microsoft and the engineers on the Mozilla project have chosen to implement this object differently, we will refer to the ActiveX and `XMLHttpRequest` objects simply as “request objects” throughout this book, because they have very similar functionality.

As a first step in using Ajax, you must check if the user’s browser supports either one of the Mozilla-based or ActiveX-related request objects, and then properly initialize the object.

Using a Function for Checking Compatibility

Wrap the compatibility check inside a JavaScript function, then call this function before you make any HTTP requests using the object. For exam-

ple, in Mozilla-based browsers such as Netscape 7.1 and Firefox 1.5 (as well as in Safari 2.0 and Opera 8.5), the request object is available as a property of the top-level `window` object. The reference to this object in JavaScript code is `window.XMLHttpRequest`. The compatibility check for these browser types looks like this:

```
if(window.XMLHttpRequest){
    request = new XMLHttpRequest();
    request.onreadystatechange=handleResponse;
    request.open("GET",theURL,true);
    request.send(null);
}
```

The JavaScript variable `request` is to a top-level variable that will refer to the request object.



As an alternative model, the open-source library *Prototype* uses object-oriented JavaScript to wrap the request object into its own object, as in the object `Ajax.Request` (see Chapter 6).

If the browser supports `XMLHttpRequest`, then:

1. `if(window.XMLHttpRequest)` returns true because the `XMLHttpRequest` is not null or undefined.
2. The object will be instantiated with the new keyword.
3. Its `onreadystatechange` event listener (see the section “XMLHttpRequest” earlier in this chapter) will be defined as a function named `handleResponse()`.
4. The code calls the request object’s `open()` and `send()` methods.

What about Internet Explorer users?



Microsoft Internet Explorer–related blogs mentioned, at the time this book went to publication, that IE 7 would support a native `XMLHttpRequest` object.

In this case, the `window.XMLHttpRequest` object will not exist in the browser object model. Therefore, another branch of the `if` test is necessary in your code:

```
else if (window.ActiveXObject){
    request=new ActiveXObject("Microsoft.XMLHTTP");
    if (! request){
        request=new ActiveXObject("Msxml2.XMLHTTP");
    }
}
```

Detect Browser Compatibility with the Request Object

```
    }  
    if(request){  
        request.onreadystatechange=handleResponse;  
        request.open(reqType,url,true);  
        request.send(null);  
    }  
}
```

This code fragment tests for the existence of the top-level window object `ActiveXObject`, thus signaling the use of Internet Explorer. The code then initializes the request using two of a number of possible ActiveX program IDs (here, `Microsoft.XMLHTTP` and `Mxml2.XMLHTTP`).

You can get even more fine-grained when testing for different versions of the IE request object, such as `Mxml2.XMLHTTP.4.0`. In the vast majority of cases, however, you will not be designing your application based on various versions of the MSXML libraries, so the prior code will suffice.

The code then makes one final check for whether the request object has been properly constructed (`if(request){...}`).

Given three chances, if the request variable is still null or undefined, your browser is really out of luck when it comes to using the request object for Ajax!

Here's an example of an entire compatibility check:

```
/* Wrapper function for constructing a request object.  
Parameters:  
reqType: The HTTP request type, such as GET or POST.  
url: The URL of the server program.  
asynch: Whether to send the request asynchronously or not. */  
  
function httpRequest(reqType,url,asynch){  
    //Mozilla-based browsers  
    if(window.XMLHttpRequest){  
        request = new XMLHttpRequest();  
    } else if (window.ActiveXObject){  
        request=new ActiveXObject("Mxml2.XMLHTTP");  
        if (! request){  
            request=new ActiveXObject("Microsoft.XMLHTTP");  
        }  
    }  
    //the request could still be null if neither ActiveXObject  
    //initialization succeeded  
    if(request){  
        initReq(reqType,url,asynch);  
    } else {  
        alert("Your browser does not permit the use of all "+  
            "of this application's features!");  
    }  
}
```

```

    }
  }
  /* Initialize a request object that is already constructed */
  function initReq(reqType,url,bool){
    /* Specify the function that will handle the HTTP response */
    request.onreadystatechange=handleResponse;
    request.open(reqType,url,bool);
    request.send(null);
  }
}

```

“Use the Request Object to POST Data to the Server” [Hack #2] shows how to implement a POST request with XMLHttpRequest.



HACK
#2

Use the Request Object to POST Data to the Server

Step beyond the traditional mechanism of posting your user’s form values.

This hack uses the POST HTTP request method to send data, communicating with the server without disrupting the user’s interaction with the application. It then displays the server’s response to the user. The difference between this hack’s approach to posting data and the typical form-submission method is that with Ajax, the page is not altered or refreshed when the application connects with the server to POST it the data. Thus, the user can continue to interact with the application without waiting for the interface to be rebuilt in the browser.

Imagine that you have a web portal in which several regions of the page or view provide the user with a variety of services. If one of these regions involves posting data, the entire application might have a more responsive feel if the POST request happens in the background. This way, the entire page (or segments of it) does not have to be refreshed in the browser.

The example web page used in this hack is a simple one. It requests users to enter their first and last names, gender, and country of origin, and then click a button to POST the data. Figure 1-1 shows what the web page looks like in a browser window.

Here’s the code for the HTML page:

```

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
  <script type="text/javascript" src="/parkerriver/js/hack2.js"></script>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Send a data tidbit</title>
</head>
<body>
<h3>A Few Facts About Yourself...</h3>
<form action="javascript:void%200" onsubmit="sendData();return false">

```



Figure 1-1. Please Mister POST man

```

<p>First name: <input type="text" name="firstname" size="20"> </p>
<p>Last name: <input type="text" name="lastname" size="20"> </p>
<p>Gender: <input type="text" name="gender" size="2"> </p>
<p>Country of origin: <input type="text" name="country" size="20"> </p>
<p><button type="submit">Send Data</button></p>
</form>
</body>
</html>

```



You may be wondering about the weird-looking form `action="javascript:void%20"` part. Because we are calling JavaScript functions when the form is submitted, we do not want to give the action attribute anything but a JavaScript URL that has no return value, such as `"javascript:void 0"`. We have to encode the space between `void` and `0`, which is where the `%20` comes in. If JavaScript is disabled in the user's browser, clicking the submit button on the form has no effect because the action attribute does not point to a valid URL. In addition, certain HTML validators will display warnings if you use `action=""`. Another way of writing this code is to include the function calls as part of the `window.onload` event handler in the JavaScript `.js` file, which is the approach used by most hacks in this book.

The first code element of interest is the `script` tag, which imports the JavaScript code (in a file named `hack2.js`). The form tag's `onsubmit` attribute specifies a function called `sendData()`, which in turn formats the data for a POST request (by calling another function, `setQueryString()`) and sends

the data to the server. For brevity's sake, we've saved the description of checking for blank fields for a later hack ("Validate a Text Field or textarea for Blank Fields" [Hack #22]), but web applications should take this step before they hit the server.

The *hack2.js* file defines the necessary JavaScript. Here is the `setQueryString()` function:

```
function setQueryString(){
    queryString="";
    var frm = document.forms[0];
    var numberElements = frm.elements.length;
    for(var i = 0; i < numberElements; i++) {
        if(i < numberElements-1) {
            queryString += frm.elements[i].name+"="+
                encodeURIComponent(frm.elements[i].value)+"&";
        } else {
            queryString += frm.elements[i].name+"="+
                encodeURIComponent(frm.elements[i].value);
        }
    }
}
```

This function formats a POST-style string out of all the form's input elements. All the name/value pairs are separated by an & character, except for the pair representing the last input element in the form. The entire string might look like:

```
firstname=Bruce&lastname=Perry&gender=M&country=USA
```

Now you have a string you can use in a POST HTTP request. Let's look at the JavaScript code that sends the request. Everything starts with the `sendData()` function. The code calls this function in the HTML form tag's `onsubmit` attribute:

```
var request;
var queryString; //will hold the POSTed data
function sendData(){
    setQueryString();
    var url="http://www.parkerriver.com/s/sender";
    httpRequest("POST",url,true);
}

/* Initialize a request object that is already constructed.
Parameters:
    reqType: The HTTP request type, such as GET or POST.
    url: The URL of the server program.
    isAsynch: Whether to send the request asynchronously or not. */
function initReq(reqType,url,isAsynch){
    /* Specify the function that will handle the HTTP response */
    request.onreadystatechange=handleResponse;
    request.open(reqType,url,isAsynch);
```

```
    /* Set the Content-Type header for a POST request */
    request.setRequestHeader("Content-Type",
        "application/x-www-form-urlencoded; charset=UTF-8");
    request.send(queryString);
}

/* Wrapper function for constructing a request object.
Parameters:
reqType: The HTTP request type, such as GET or POST.
url: The URL of the server program.
asynch: Whether to send the request asynchronously or not. */

function httpRequest(reqType,url,asynch){
    //Mozilla-based browsers
    if(window.XMLHttpRequest){
        request = new XMLHttpRequest();
    } else if (window.ActiveXObject){
        request=new ActiveXObject("Msxml2.XMLHTTP");
        if (! request){
            request=new ActiveXObject("Microsoft.XMLHTTP");
        }
    }
    //the request could still be null if neither ActiveXObject
    //initialization succeeded
    if(request){
        initReq(reqType,url,asynch);
    } else {
        alert("Your browser does not permit the use of all "+
            "of this application's features!");
    }
}
}
```

The purpose of the `httpRequest()` function is to check which request object the user's browser is associated with (see "Detect Browser Compatibility with the Request Object" [Hack #1]). Next, the code calls `initReq()`, whose parameters are described in the comment just above the function definition.

The code `request.onreadystatechange=handleResponse;` specifies the event-handler function that deals with the response. We'll look at this function a little later. The code then calls the request object's `open()` method, which prepares the object to send the request.

Setting Headers

The code can set any request headers after calling `open()`. In our case, we have to create a Content-Type header for a POST request.



Firefox required the additional Content-Type header; Safari 1.3 did not. (We were using Firefox 1.02 at the time of writing this hack.) It is a good idea to add the proper header because in most cases the server is expecting it from a POST request.

Here's the code for adding the header and sending the POST request:

```
request.setRequestHeader("Content-Type",  
    "application/x-www-form-urlencoded; charset=UTF-8");  
request.send(queryString);
```

If you enter the raw `queryString` value as a parameter, the method call looks like this:

```
send("firstname=Bruce&lastname=Perry&gender=M&country=USA");
```

Ogling the Result

Once your application POSTs data, you want to display the result to your users. This is the responsibility of the `handleResponse()` function. Remember the code in the `initReq()` function:

```
request.onreadystatechange=handleResponse;
```

When the request object's `readyState` property has a value of 4, signifying that the object's operations are complete, the code checks the HTTP response status for the value 200. This value indicates that the HTTP request has succeeded. The `responseText` is then displayed in an alert window. This is somewhat anticlimactic, but I thought I'd keep this hack's response handling simple, because so many other hacks do something more complex with it!

Here is the relevant code:

```
//event handler for XMLHttpRequest  
function handleResponse(){  
    if(request.readyState == 4){  
        if(request.status == 200){  
            alert(request.responseText);  
        } else {  
            alert("A problem occurred with communicating between "+  
                "the XMLHttpRequest object and the server program.");  
        }  
    }  
}  
} //end outer if
```

Figure 1-2 shows what the alert window looks like after the response is received.

The server component returns an XML version of the POSTed data. Each parameter name becomes an element name, with the parameter value as the

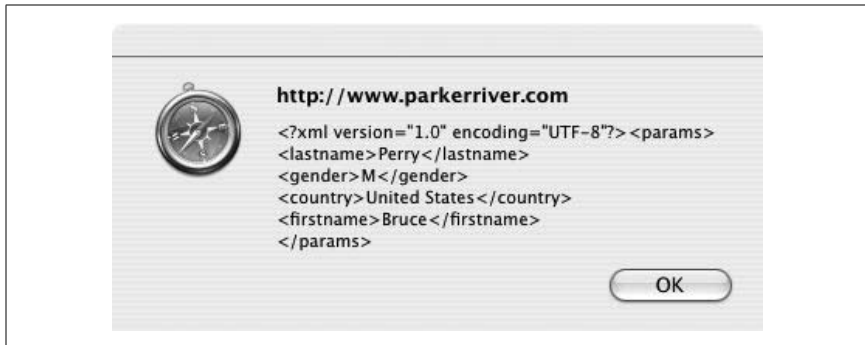


Figure 1-2. Alert! Server calling...

element content. This POSTed data is nested within `params` tags. The component is a Java servlet. The servlet is not the main focus of this hack, but here's some code anyway, for the benefit of readers who are curious about what is happening on the server end:

```
protected void doPost(HttpServletRequest request,
                      HttpServletResponse response) throws
                      ServletException, IOException {
    Map reqMap = request.getParameterMap();
    String val=null;
    String tag = null;
    StringBuffer body = new StringBuffer("<params>\n");
    boolean wellFormed = true;
    Map.Entry me = null;
    for(Iterator iter= reqMap.entrySet().iterator();iter.hasNext();) {
        me=(Map.Entry) iter.next();
        val= ((String[])me.getValue())[0];
        tag = (String) me.getKey();
        if (! XMLUtils.isWellFormedXMLName(tag)){
            wellFormed=false; break;
        }
        body.append("<").append(tag).append(">").
            append(XMLUtils.escapeBodyValue(val)).
            append("</").append(tag).append(">\n");
    }
    if(wellFormed) {
        body.append("</params>");
        sendXML(response,body.toString());
    } else {
        sendXML(response,"<notWellFormedParams />");
    }
}
```

The code uses `XMLUtils`, a Java class from the Jakarta Commons Betwixt open source package, to check whether the parameter names are well

formed, as well as whether the parameter values contain invalid XML content and thus have to be escaped. If for some reason the component is POSTed data that contains nonwell-formed parameter names (such as na< > me instead of name), the servlet returns an empty XML element reporting this condition.



HACK #3

Use Your Own Library for XMLHttpRequest

Break out the code that initializes the request object and sends requests to its own JavaScript file.

To cleanly separate the concerns of big Ajax applications, create a separate file that manages the XMLHttpRequest object, then import that file into every web page that needs it. At the very least, this ensures that any necessary changes regarding how the code sets up the request object have to be made only in this file, as opposed to every JavaScript file that uses Ajax-style requests.

This hack stores all the request object–related code in a file called *http_request.js*. Any web page that uses XMLHttpRequest can then import this file in the following way:

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

Here's the code for the file, including all the comments:

```
var request = null;
/* Wrapper function for constructing a request object.
Parameters:
  reqType: The HTTP request type, such as GET or POST.
  url: The URL of the server program.
  async: Whether to send the request asynchronously or not.
  respHandle: The name of the function that will handle the response.
Any fifth parameters, represented as arguments[4], are the data a
POST request is designed to send. */
function httpRequest(reqType,url,async,respHandle){
  //Mozilla-based browsers
  if(window.XMLHttpRequest){
    request = new XMLHttpRequest();
  } else if (window.ActiveXObject){
    request=new ActiveXObject("Msxml2.XMLHTTP");
    if (! request){
      request=new ActiveXObject("Microsoft.XMLHTTP");
    }
  }
  //very unlikely, but we test for a null request
  //if neither ActiveXObject was initialized
  if(request) {
    //if the reqType parameter is POST, then the
    //5th argument to the function is the POSTed data
```

```
        if(reqType.toLowerCase() != "post") {
            initReq(reqType,url,asynch,respHandle);
        } else {
            //the POSTed data
            var args = arguments[4];
            if(args != null && args.length > 0){
                initReq(reqType,url,asynch,respHandle,args);
            }
        }
    } else {
        alert("Your browser does not permit the use of all "+
            "of this application's features!");
    }
}
/* Initialize a request object that is already constructed */
function initReq(reqType,url,bool,respHandle){
    try{
        /* Specify the function that will handle the HTTP response */
        request.onreadystatechange=respHandle;
        request.open(reqType,url,bool);
        //if the reqType parameter is POST, then the
        //5th argument to the function is the POSTed data
        if(reqType.toLowerCase() == "post") {
            request.setRequestHeader("Content-Type",
                "application/x-www-form-urlencoded; charset=UTF-8");
            request.send(arguments[4]);
        } else {
            request.send(null);
        }
    } catch (errv) {
        alert(
            "The application cannot contact "+
            "the server at the moment. "+
            "Please try again in a few seconds.\n"+
            "Error detail: "+errv.message);
    }
}
```

The applications that use this code call the `httpRequest()` function with four or five (with POST requests) parameters. You see lots of examples of calling this function in the other hacks. Here's another:

```
var _url = "http://www.parkerriver.com/s/sender";
var _data="first=Bruce&last=Perry&middle=W";
httpRequest("POST",_url,true,handleResponse,_data);
```

The code comments describe the meaning of each of these parameters. The last parameter represents the data that accompanies a POST request.



A POST HTTP request includes the POSTed data beneath the request-header information. A GET request, on the other hand, appends parameter names/values onto the URL.

If the code is not using POST, the client code uses only the first four parameters. The fourth parameter can be either the name of a function that is declared in the client code (i.e., a response-handling function that appears outside of the *http_request.js* file) or a function literal. The latter option involves defining a function inside a function call, which is often awkward and difficult to read. However, it is sensible in situations in which the HTTP response handling is short and simple, as in:

```
var _url = "http://www.parkerriver.com/s/sender";  
//a debugging setup  
httpRequest("POST", _url, true, function(){alert(request.responseText);});
```

`httpRequest()` initiates the same browser detection and setup of XMLHttpRequest for Internet Explorer and non-Microsoft browsers as described in “Detect Browser Compatibility with the Request Object” [Hack #1]. `initReq()` handles the second step of setting up the request object: specifying the `onreadystatechange` event handler and calling the `open()` and `send()` methods to make an HTTP request. The code traps any errors or exceptions thrown by these request method calls using a `try/catch` statement. For example, if the code calls `open()` with a URL specifying a different host than that used to download the enclosing web page, the `try/catch` statement catches the error and pops up an alert window.

Finally, as long as the web page imports *http_request.js*, the `request` variable is available to code external to the imported file; `request` is, in effect, a global variable.



`request` is thus reserved as a variable name because local variables that use the `var` keyword will supercede (with unintentional consequences) the globally used `request`, as in the following example:

```
function handleResponse(){  
    //supercedes the imported request variable  
    var request = null;  
    try{  
        if(request.readyState == 4){  
            if(request.status == 200){...
```



HACK

#4

Receive Data as XML

Ajax and server programs provide a DOM Document object that's ready to go.

Many technologies currently exchange data in Extensible Markup Language format, mostly because XML is a standardized and extensible format widely supported by the software world. Thus, different parties can use existing, well-known technologies to generate, send, and receive XML, without having to adapt to the software tools used by the parties with whom they are exchanging the XML data.

An example is a Global Positioning System (GPS) device that can share the data it has recorded about, say, a hike or a bike ride with a location-aware web application. You just stick the USB cable attached to the GPS device into a USB computer port, launch software that sends the device data to the Web, and that's it. The data format is usually an XML language that has been defined already for GPS software. The web application and the GPS device “speak the same language.”

Although this book is not the place for an extensive introduction to XML, you have probably seen these text files in one form or another. XML is used as a “meta” language that describes and categorizes specific types of information. XML data starts with an optional XML declaration (e.g., `<?xml version="1.0" encoding="UTF-8"?>`), followed by a root element and zero or more child elements. An example is:

```
<?xml version="1.0" encoding="UTF-8"?>
<gps>
  <gpsMaker>Garmin</gpsMaker>
  <gpsDevice>
    Forerunner 301
  </gpsDevice>
</gps>
```

Here, `gps` is the root element, and `gpsMaker` and `gpsDevice` are child elements.

Ajax and the request object can receive data as XML, which is very useful for handling web-services responses that use XML. Once the HTTP request is complete, the request object has a property named `responseXML`. This object is a DOM Document object that your Ajax application can use. Here's an example:

```
function handleResponse(){
  if(request.readyState == 4){
    if(request.status == 200){
      var doc = request.responseXML;
    }
  }
}
```

In the previous code sample, the `doc` variable is a DOM Document object, offering a similar API to a browser's display page. This hack receives XML from a server, then initiates a little DOM programming with the Document object to pull some information out of the XML.



If you just want to see the raw XML text, use the request's `responseText` property instead.

The HTML file for this hack is basically the same as the one used in “Use the Request Object to POST Data to the Server” [Hack #2], but a `div` element is added at the end, where the code displays information about the returned XML. Here's the code for the HTML page:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
  <script type="text/javascript" src="js/hack3.js"></script>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Receive XML response</title>
</head>
<body>
<h3>A Few Facts About Yourself...</h3>
<form action="javascript:void%200" onsubmit="sendData();return false">
  <p>First name: <input type="text" name="firstname" size="20"> </p>
  <p>Last name: <input type="text" name="lastname" size="20"> </p>
  <p>Gender: <input type="text" name="gender" size="2"> </p>
  <p>Country of origin: <input type="text" name="country" size="20"> </p>
  <p><button type="submit">Send Data</button></p>
  <div id="docDisplay"></div>
</form>
</body>
</html>
```

Figure 1-3 shows what the page looks like before the user enters any information.

The JavaScript code in the `hack3.js` file POSTs its data to a server application, which sends back a response in XML format. The field validation step [Hack #22] has been skipped for the sake of brevity, but web applications using forms should always implement this task.

Like other examples in this chapter, the server program echoes the parameter names and values back to the client, as in `<params><firstname>Bruce</firstname></params>`. “Use the Request Object to POST Data to the Server” [Hack #2] shows some of the code for the server component that puts together

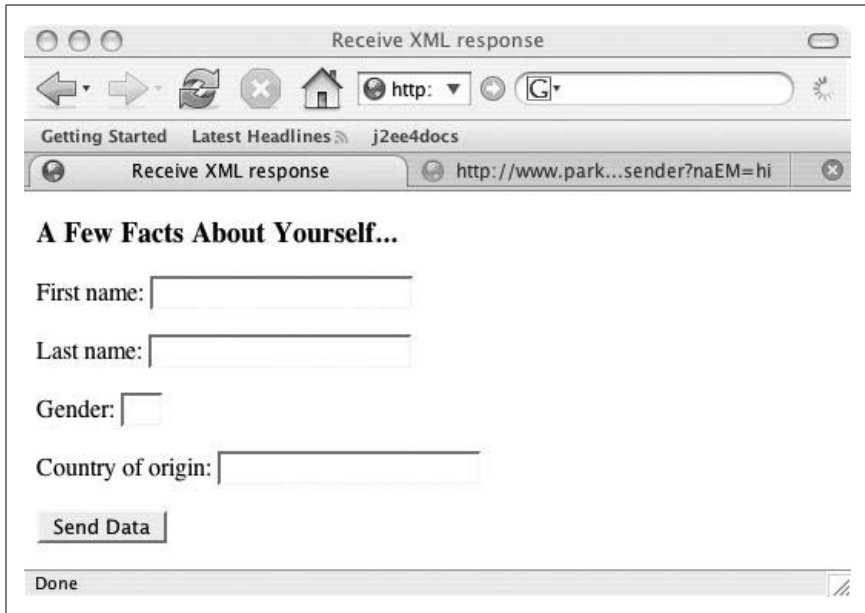


Figure 1-3. All set up to receive XML

the return value. This technique suits our purpose for showing a simple example of programming XML in an Ajax application:

```
var request;
var queryString; //will hold the POSTed data

function sendData(){
    setQueryString();
    var url="http://www.parkerriver.com/s/sender";
    httpRequest("POST",url,true);
}
//event handler for XMLHttpRequest
function handleResponse(){
    if(request.readyState == 4){
        if(request.status == 200){
            var doc = request.responseXML;
            var info = getDocInfo(doc);
            stylizeDiv(info,document.getElementById("docDisplay"));
        } else {
            alert("A problem occurred with communicating between "+"
                "the XMLHttpRequest object and the server program.");
        }
    }
} //end outer if
}

/* Initialize a request object that is already constructed */
function initReq(reqType,url,bool){
```

```

    /* Specify the function that will handle the HTTP response */
    request.onreadystatechange=handleResponse;
    request.open(reqType,url,bool);
    request.setRequestHeader("Content-Type",
        "application/x-www-form-urlencoded; charset=UTF-8");
    /* Only works in Mozilla-based browsers */
    //request.overrideMimeType("text/xml");
    request.send(queryString);
}

/* Wrapper function for constructing a request object.
Parameters:
reqType: The HTTP request type, such as GET or POST.
url: The URL of the server program.
asynch: Whether to send the request asynchronously or not. */
function httpRequest(reqType,url,asynch){
    //Snipped...See Hack #1
}

function setQueryString(){
    queryString="";
    var frm = document.forms[0];
    var numberElements = frm.elements.length;
    for(var i = 0; i < numberElements; i++) {
        if(i < numberElements-1) {
            queryString += frm.elements[i].name+"="+
                encodeURIComponent(frm.elements[i].value)+"&";
        } else {
            queryString += frm.elements[i].name+"="+
                encodeURIComponent(frm.elements[i].value);
        }
    }
}

/* Provide the div element's content dynamically. We can add
style information to this function if we want to jazz up the div */
function stylizeDiv(bdyTxt,div){
    //reset DIV content
    div.innerHTML="";
    div.style.backgroundColor="yellow";
    div.innerHTML=bdyTxt;
}

/* Get information about an XML document via a DOM Document object */
function getDocInfo(doc){
    var root = doc.documentElement;
    var info = "<h3>Document root element name: <h3 />"+ root.nodeName;
    var nds;
    if(root.hasChildNodes()) {
        nds=root.childNodes;
        info+= "<h4>Root node's child node names/values:<h4 />";
        for (var i = 0; i < nds.length; i++){
            info+= nds[i].nodeName;
            if(nds[i].hasChildNodes()){
                info+= " : \"+nds[i].firstChild.nodeValue+"\<br />";
            }
        }
    }
}

```

```
        } else {  
            info+= " : Empty<br />";  
        }  
    }  
}  
return info;  
}
```



Mozilla Firefox can use the `request.overrideMimeType()` function to force the interpretation of the response stream as a certain mime type, as in `request.overrideMimeType("text/xml")`. Internet Explorer's request object does not have this function. This function call does not work with Safari 1.3, either.

After the code POSTs its data and receives a response, it calls a method named `getDocInfo()`, which builds a string that displays some information about the XML document and its child or subelements:

```
var doc = request.responseXML;  
var info = getDocInfo(doc);
```

The `getDocInfo()` function gets a reference to the root XML element (`var root = doc.documentElement;`); it then builds a string specifying the name of the root element and information about any of its child nodes or elements, such as the child node name and value. The code then feeds this information to the `stylizeDiv()` method. The `stylizeDiv()` method uses the `div` element at the end of the HTML page to dynamically display the gathered information:

```
function stylizeDiv(bdyTxt,div){  
    //reset div content  
    div.innerHTML="";  
    div.style.backgroundColor="yellow";  
    div.innerHTML=bdyTxt;  
}
```

Figure 1-4 shows what the web page looks like after the application receives the XML response.



The text nodes that the application shows are newline characters in the returned XML.

The core DOM API offered by the browser's JavaScript implementation provides developers with a powerful tool for programming complex XML return values.

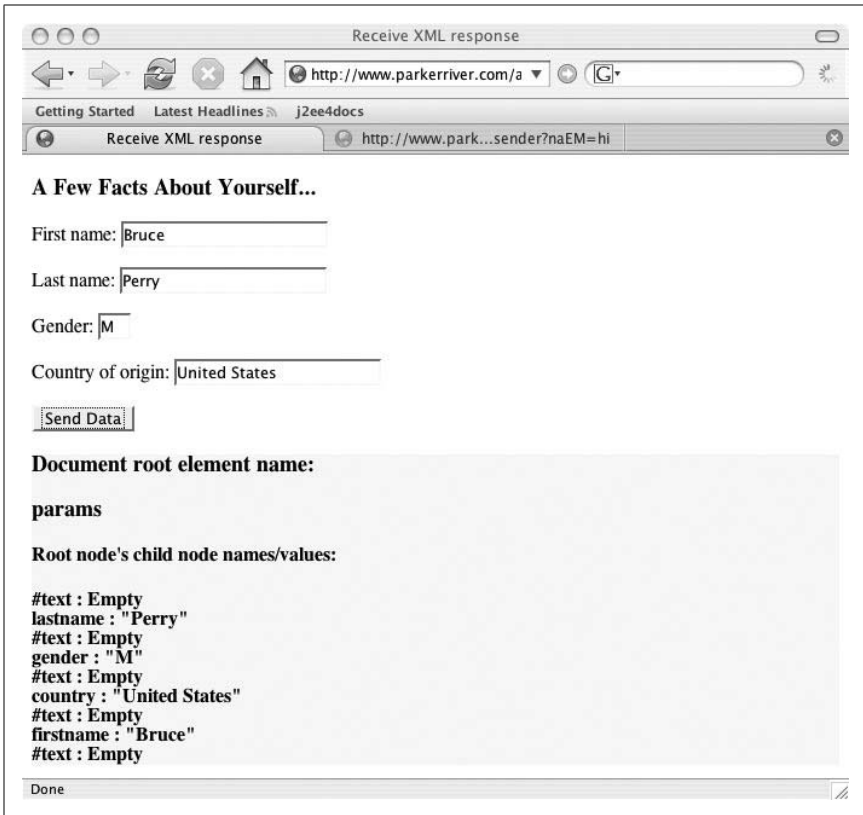


Figure 1-4. Delving into XML return values



HACK #5

Get Plain Old Strings

Manage weather readings, stock quotes, web page scrapings, or similar non-XML data as plain old strings.

The request object has the perfect property for web applications that do not have to handle server return values as XML: `request.responseText`. This hack asks the user to choose a stock symbol, and the server returns the stock price for display. The code handles the return value as a string.



A variation to this program in the next hack requires the stock prices to be handled as numbers. These are old prices that a server component stores for certain stock symbols, not *live* quotes that you would obtain from a commercial web service or by HTML scraping. For an example of that mechanism, see “Use XMLHttpRequest to Scrape a Energy Price from a Web Page” [Hack #39].

First, here is the HTML for the web page. It imports JavaScript code from a file named *hack9.js*:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/1999/REC-html401-19991224/strict.dtd">
<html>
<head>
  <script type="text/javascript" src="js/hack9.js"></script>
  <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  <title>Choose a stock</title>
</head>
<body>
<h3>Stock prices</h3>
<form action="javascript:void%200" onsubmit=
  "getStockPrice(this.stSymbol.value);return false">
  <p>Enter stock symbol: <input type="text" name=
    "stSymbol" size="4"><span id="stPrice"></span></p>
  <p><button type="submit">Get Stock Price</button></p>
</form>
</body>
</html>
```

Figure 1-5 shows the web page as displayed in Firefox. The user enters a symbol such as “GRMN” (case insensitive) and clicks the Get Stock Price button; the JavaScript then fetches the associated stock price and displays it within a span element to the right of the text field.



Figure 1-5. Instantaneously displaying a stock price

The function that sets the request process in motion is `getStockPrice()`. This function takes the value of the text field named `stSymbol` and returns the associated stock price (it uses the request object to talk to a server component, which fetches the actual stock price). Here is the JavaScript code:

```
var request;
```

```
var symbol; //will hold the stock symbol

function getStockPrice(sym){
  symbol=sym;
  if(sym){
    var url="http://localhost:8080/parkerriver/s/stocks?symbol="+sym;
    httpRequest("GET",url,true);
  }
}

//event handler for XMLHttpRequest
function handleResponse(){
  if(request.readyState == 4){
    if(request.status == 200){
      /* Grab the result as a string */
      var stockPrice = request.responseText;
      var info = "&#171;The price is: $" +stockPrice+"&#187;";
      document.getElementById("stPrice").style.fontSize="0.9em";
      document.getElementById("stPrice").style.
      backgroundColor="yellow";
      document.getElementById("stPrice").innerHTML=info;

    } else {
      alert("A problem occurred with communicating between "+
        "the XMLHttpRequest object and the server program.");
    }
  } //end outer if
}

/* See Hack #1 for the httpRequest() code;
it is snipped here for the sake of brevity. */
```

The function `getStockPrice()` wraps a call to the function `httpRequest()`, which is responsible for setting up the request object. If you have already read through some of this chapter's other hacks, you will recognize the `handleResponse()` function as enclosing much of the interesting action.



“Detect Browser Compatibility with the Request Object” [Hack #1] and “Use Your Own Library for XMLHttpRequest” [Hack #3] explain the `httpRequest()` function in more detail.

If the request is complete (i.e., if `request.readyState` has a value of 4) and the HTTP response status is 200 (meaning that the request has succeeded), the code grabs the server response as the `request.responseText` property value. The code then uses DOM scripting to display the stock price with some CSS style-related attributes:

```
document.getElementById("stPrice").style.fontSize="0.9em";
document.getElementById("stPrice").style.backgroundColor="yellow";
document.getElementById("stPrice").innerHTML =info;
```

The style attributes make the font size a little bit smaller than the user's preferred browser font size and specify yellow as the background color of the text display. The `innerHTML` property of the `span` element is set to the stock price within double angle brackets.