The Ajax transport method

There's more to Ajax than XMLHttp

Skill Level: Intermediate

Mr. Jack D Herrington (jherr@pobox.com)
Senior Software Engineer
Leverage Software, Inc.

06 Jun 2006

Discover three Ajax data transport mechanisms (XMLHttp, script tags, and frames or iframes) and their relative strengths and weaknesses. This tutorial provides code for both the server side and the client side and explains it in detail to provide the techniques you need to put efficient Ajax controls anywhere you need them.

Section 1. Before you start

Learn what to expect from this tutorial and how to get the most out of it.

About this tutorial

With all the hype around Asynchronous JavaScript and XML (Ajax), you would think that more people would know how it works. But all engineers seem to talk about is the `XMLHttp` method of getting data to and from the server. If this method is all you know, then what you can do with Ajax is limited. In reality, there are three ways to get data to and from the server: the `XMLHttp` method, the `<script>` tag method, and the *frame* or *iframe* approach. Unless you know all three of these methods (and their relative strengths and weaknesses), you won't get the full picture. This tutorial shows you all you need to know about the critical transport piece of the Ajax puzzle that fits between the client and the server.

In addition to understanding how the client requests data from the server, there's the

question of what type of data is going across the wire. Most Ajax articles talk about Extensible Markup Language (XML), but you can actually move plain text, Hypertext Markup Language (HTML) pages, or JavaScript code. You have a lot of reasons for thinking outside the XML box.

In this tutorial, you will:

- Create the database back end for the example.
- Build a set of server-side pages that provide access to the database data.
- Build a set of pages based on `XMLHttp` that use the data services.
- Build a set of pages based on iframes that use the data services.
- Build a set of pages based on `<script>` tags that use the data services.

Objectives

In this tutorial, you will learn the three primary methods of Ajax transport and their use in practice with PHP and JavaScript code.

Prerequisites

This tutorial assumes a basic knowledge of XML, HTML, and the JavaScript and PHP programming languages. To run the examples in this tutorial, you need:

- A PHP server that has access to a server running MySQL.
- A Web browser. (I recommend either Mozilla Firefox or Microsoft® Internet Explorer V6.)

System requirements

To run the examples in this tutorial, you will need to install the Apache Web server as well as PHP. You will also need access to a Web browser, such as Mozilla Firefox.

Section 2. Laying the groundwork

Download the code
All the HTML, PHP, and SQL source code used in this tutorial is available for download from the Downloads section.

What is Ajax anyway, and why is it important? Ajax is simply a way of updating Web pages on the fly without going back to the server for a completely new page. Looking at it sequentially, the client first requests the page from the server. The server sends back an HTML page with embedded JavaScript code. That JavaScript code then requests more data from the server -- typically using XML -- and updates the page dynamically. You might trigger the script on a timer or based on user input. Exactly how the script gets the data from the server is what you'll learn here.

Most books and articles on Ajax reference one method of Ajax transport: the `XMLHttp` request. This is for two reasons: First, this method is perhaps the most elegant, and second, it's the latest method. It's not so new that you must worry about browser compatibility, however. If a browser doesn't support `XMLHttp`, it's unlikely that the browser is particularly good for dynamic HTML (DHTML) work, and it might not be worth supporting.

The `XMLHttp` object is very flexible, but for security reasons, it's restricted in exactly where it can retrieve data. This restriction might not seem important, but it's a real problem if you want to perform tasks such as place your Ajax controls on a blog page from another server. For that, you must look at an alternative method: the lowly `<script>` tag. And let's not forget the third option: Using a frame or inline frame (or *iframe*) to move data to and from the server. This approach isn't particularly recent, but it is worth knowing about so that you can identify it in practice.

This tutorial contains three separate sections of code. At the bottom level is the database, including the schema and data. On top of that are the server pages that expose the data in the database in various forms. Layered on top of that is a set of HTML pages that access that data through a variety of mechanisms.

In this section, you start by setting up your database. Then, you create the PHP pages that access the data.

Build the database

What is a Web application without a database? For this tutorial, you use a simple one-table book database in MySQL. Listing 1 shows that database and defines its records.

Listing 1. The database

```
CREATE TABLE books (
  id MEDIUMINT NOT NULL AUTO_INCREMENT,
  author TEXT,
  title TEXT,
  PRIMARY KEY ( id ) );

INSERT INTO books VALUES (
  null,
  "Jack Herrington",
  "Code Generation in Action" );
INSERT INTO books VALUES (
  null,
  "Jack Herrington",
  "Podcasting Hacks" );
INSERT INTO books VALUES (
  null,
  "Jack Herrington",
  "PHP Hacks" );
```

Listing 1 doesn't show anything too complicated: just a single table with three fields, a unique identifier, a title, and an author. Exposing this table to the client requires several different Web pages that each export a different format for the data. The sections that follow detail each format type and show the PHP server code that outputs the data from the database in the given format.

Build the text service

The PHP code shown in Listing 2 connects to the database and dumps the contents as a text string with carriage returns between each record.

Listing 2. The text formatted service

```
<?php
require_once("DB.php");

header( "Content-type: text" );

$dsn = 'mysql://root:password@localhost/ajaxdb';
$db =& DB::Connect( $dsn, array() );
if (PEAR::isError($db)) { die($db->getMessage()); }

$res = $db->query( "SELECT author, title FROM books" );
while( $res->fetchInto( $row ) )
{
  echo($row[0].' - '.$row[1]."\n");
}
?>
```

To test this service, you can run it on the command line using the `php` interpreter, as Listing 3 shows.

Listing 3. Run the text formatted service

```
% php text.php
Jack Herrington - Code Generation in Action
Jack Herrington - Podcasting Hacks
Jack Herrington - PHP Hacks
%
```

This example might not seem terribly important, but remember that in the Ajax world, anything that isn't XML or an XML-compliant stream is considered text. So the vCard you want to move around or the base64-encoded stream is simply text. You must know how to move non-XML data around.

Build the XML service

XML is the most common source for Ajax work. You create the simplest version of XML for this example using the code in Listing 4.

Listing 4. Xml.php

```
<php
header( 'Content-type: text/xml' );
?>
  <books>
    <?php
      require_once("DB.php");

      $dsn = 'mysql://root:password@localhost/ajaxdb';
      $db =& DB::Connect( $dsn, array() );
      if (PEAR::isError($db)) { die($db->getMessage()); }

        $res = $db->query( "SELECT author, title FROM books" );
        while( $res->fetchInto( $row ) )
        {
    ?>
    <book><author><?php echo($row[0]) ?>
      </author><title><?php echo($row[1]) ?>
      </title></book>
    <?php
    }
    ?>
  </books>
```

Notice that at the top of the script, you set the `Content-type` header field to the `text/xml` MIME type. This type is critical for Ajax work. Unless the MIME type is properly set, the XML document won't be returned at the end of an `XMLHttp` request.

Listing 5 shows the output of Listing 4 on the command line.

Listing 5. Output from xml.php

```
% php xml.php
<books>
  <book><author>Jack Herrington</author><title>Code Generation in Action</title></book>
  <book><author>Jack Herrington</author><title>Podcasting Hacks</title></book>
  <book><author>Jack Herrington</author><title>PHP Hacks</title></book>
</books>
%
```

When working with the iframe solution, you can't send XML directly. Instead, you must encode the XML as HTML text. Listing 6 shows this encoding.

Listing 6. Xml_text.php

```
<?php
header( 'Content-type: text/html' );
  ob_start();
?>
  <books>
    <?php
      require_once("DB.php");

      $dsn = 'mysql://root:password@localhost/ajaxdb';
      $db =& DB::Connect( $dsn, array() );
      if (PEAR::isError($db)) { die($db->getMessage()); }

        $res = $db->query( "SELECT author, title FROM books" );
      while( $res->fetchInto( $row ) )
      {
    ?>
    <book><author><?php echo($row[0]) ?>
      </author><title><?php echo($row[1]) ?>
        </title></book>
    <?php
    }
    ?>
  </books>
<?php
$xml = ob_get_clean();
$xml = preg_replace( '/\>/', '>', $xml );
$xml = preg_replace( '/\</', '<', $xml );
echo( $xml );
?>
```

In this case, the content type is set to `html`, and all the XML content is captured using the `ob_start()` and `ob_get_clean()` methods. Then, the XML is encoded as an HTML string by converting the < and > symbols to `<` and `>`. You can see the output of this script in Listing 7.

Listing 7. Output from xml_text.php

```
% php xml_text.php
  <books>
    <book><author>Jack Herrington</author>
      <title>Code Generation in Action</title></book>
    <book><author>Jack Herrington</author>
      <title>Podcasting Hacks</title></book>
    <book><author>Jack Herrington</author>
      <title>PHP Hacks</title></book>
  </books>
%
```

But working with a custom XML format like this is only part of the XML story. You might want to use other XML standards in your application.

Build the RSS and RDF service

The Rich Site Summary (RSS) and Resource Description Framework (RDF) syndication standards have two major benefits: They are XML based, which makes them ideal for Ajax, and they are easy to create. Listing 8 shows the code that creates an RSS feed from the database table.

Listing 8. Rss.php

```
<?php
header( 'Content-type: text/xml' );
?>
  <rss version="0.91">
    <channel>
      <title>Book List</title>
      <description>My book list</description>
      <?php
        require_once("DB.php");

        $dsn = 'mysql://root:password@localhost/ajaxdb';
        $db =& DB::Connect( $dsn, array() );
        if (PEAR::isError($db)) { die($db->getMessage()); }

          $res = $db->query( "SELECT author, title, id FROM books" );
        while( $res->fetchInto( $row ) )
        {
      ?>
      <item><description>A book by <?php echo($row[0]) ?>
        </description><title><?php echo($row[1]) ?>
          </title>
        <link>http://myhost/book.php?id=<?php echo($row[2]) ?></link>
      </item>
      <?php
      }
      ?>
    </channel>
  </rss>
```

When you run rss.php on the command line, you see the RSS output in Listing 9.

Listing 9. Output from rss.php

```
% php rss.php
  <rss version="0.91">
    <channel>
      <title>Book List</title>
        <description>My book list</description>
        <item><description>A book by Jack Herrington</description>
          <title>Code Generation in Action</title>
          <link>http://myhost/book.php?id=1</link>
        </item>
        <item><description>A book by Jack Herrington</description>
          <title>Podcasting Hacks</title>
          <link>http://myhost/book.php?id=2</link>
        </item>
        <item><description>A book by Jack Herrington</description>
          <title>PHP Hacks</title>
          <link>http://myhost/book.php?id=3</link>
        </item>
    </channel>
  </rss>
%
```

The great part about using RSS for your Ajax data is that you can do things with one action. Not only do you get the data to your JavaScript code, but you also create a syndication feed that people can subscribe to using their RSS readers. And RSS readers are becoming ubiquitous. The shipping Firefox Web browser supports RSS feeds natively, and Internet Explorer version 7 will also support RSS.

But what if you have extra data that you send along with each item? You can simply add extra tags, as in Listing 10.

Listing 10. Rssextra.php

```
<?php
header( 'Content-type: text/xml' );
?>
  <rss version="0.91">
    <channel>
      <title>Book List</title>
      <description>My book list</description>
      <?php
        require_once("DB.php");

        $dsn = 'mysql://root:password@localhost/ajaxdb';
        $db =& DB::Connect( $dsn, array() );
        if (PEAR::isError($db)) { die($db->getMessage()); }

          $res = $db->query( "SELECT author, title, id FROM books" );
        while( $res->fetchInto( $row ) )
        {
      ?>
      <item><description>A book by <?php echo($row[0]) ?>
        </description><title><?php echo($row[1]) ?>
        </title>
        <link>http://myhost/book.php?id=<?php echo($row[2]) ?></link>
```

```
        <author><?php echo($row[0]) ?></author>
        <id><?php echo($row[2]) ?></id>
      </item>
      <?php
      }
      ?>
    </channel>
  </rss>
```

When you run rssextra.php on the command line, you can see the extra tags with the author and ID data, as in Listing 11.

Listing 11. Output from rssextra.php

```
% php rssextra.php
  <rss version="0.91">
    <channel>
      <title>Book List</title>
        <description>My book list</description>
        <item><description>A book by Jack Herrington</description>
          <title>Code Generation in Action</title>
          <link>http://myhost/book.php?id=1</link>
          <author>Jack Herrington</author>
          <id>1</id>
        </item>
        <item><description>A book by Jack Herrington</description>
          <title>Podcasting Hacks</title>
          <link>http://myhost/book.php?id=2</link>
          <author>Jack Herrington</author>
          <id>2</id>
        </item>
        <item><description>A book by Jack Herrington</description>
          <title>PHP Hacks</title>
          <link>http://myhost/book.php?id=3</link>
          <author>Jack Herrington</author>
          <id>3</id>
        </item>
    </channel>
  </rss>
%
```

As it turns out, you can add extra tags to RSS feeds that most readers simply ignore. But if you don't feel comfortable with that, you can always use RDF, which is a similar syndication format that allows you to add extra tags using another name space without violating the specification. Listing 12 shows the code for the RDF feed.

Listing 12. Rdf.php

```
<?php
  header( 'Content-type: text/xml' );
  echo( '<?xml version="1.0" ?>'."\n" );
?>
  <rdf xmlns:mc="http://mysite.com">
    <channel>
```

```
      <title>Book List</title>
      <description>My book list</description>
      <link>http://mysite.com/</list>
    </channel>
    <?php
      require_once("DB.php");

      $dsn = 'mysql://root:password@localhost/ajaxdb';
      $db =& DB::Connect( $dsn, array() );
      if (PEAR::isError($db)) { die($db->getMessage()); }

        $res = $db->query( "SELECT author, title, id FROM books" );
      while( $res->fetchInto( $row ) )
      {
    ?>
    <item><description>A book by <?php echo($row[0]) ?>
      </description><title><?php echo($row[1]) ?>
      </title>
      <link>http://mysite.com/book.php?id=<?php echo($row[2]) ?></link>
      <mc:author><?php echo($row[0]) ?></mc:author>
        <mc:id><?php echo($row[2]) ?></mc:id>
    </item>
    <?php
    }
    ?>
  </rdf>
```

When you run rdf.php, you get the RDF output in Listing 13.

Listing 13. Output from rdf.php

```
% php rdf.php
<?xml version="1.0" ?>
  <rdf xmlns:mc="http://mysite.com">
    <channel>
      <title>Book List</title>
      <description>My book list</description>
      <link>http://mysite.com/</list>
    </channel>
    <item><description>A book by Jack Herrington</description>
      <title>Code Generation in Action</title>
      <link>http://mysite.com/book.php?id=1</link>
      <mc:author>Jack Herrington</mc:author>
      <mc:id>1</mc:id>
    </item>
    <item><description>A book by Jack Herrington</description>
      <title>Podcasting Hacks</title>
      <link>http://mysite.com/book.php?id=2</link>
      <mc:author>Jack Herrington</mc:author>
      <mc:id>2</mc:id>
    </item>
    <item><description>A book by Jack Herrington</description>
      <title>PHP Hacks</title>
      <link>http://mysite.com/book.php?id=3</link>
      <mc:author>Jack Herrington</mc:author>
      <mc:id>3</mc:id>
    </item>
  </rdf>
%
```

With this script, you defined an extra name space in the `<rdf>` tag called `mc`. It doesn't really matter whether that name space connects to a real site. What matters is that with that name space, you defined two additional tags -- `<mc:author>` and `<mc:id>` -- that add the author and ID data to each item.

Build the JavaScript (JSON) service

The JavaScript code format is gaining popularity in Ajax development. It's often referred to as the *JavaScript Object Notation format,* or JSON, but in reality, it's simply JavaScript. Listing 14 shows the first version of this data service script.

Listing 14. Js.php

```
<?php
  require_once("DB.php");

  header( "Content-type: text/javascript" );

    $dsn = 'mysql://root:password@localhost/ajaxdb';
    $db =& DB::Connect( $dsn, array() );
    if (PEAR::isError($db)) { die($db->getMessage()); }

      $res = $db->query( "SELECT author, title FROM books" );
      $items = array();
    while( $res->fetchInto( $row ) )
    {
      $items []= "{ author: '".$row[0]."', title: '".$row[1]."' }";
    }
?>
[ <?php echo( join( ",\n", $items ) ); ?> ];
```

Notice that you set the MIME type to `text/javascript` so that the browser recognizes it. Run the script to return the output in Listing 15.

Listing 15. Output from js.php

```
% php js.php
  [ { author: 'Jack Herrington', title: 'Code Generation in Action' },
  { author: 'Jack Herrington', title: 'Podcasting Hacks' },
  { author: 'Jack Herrington', title: 'PHP Hacks' } ];
%
```

This output is simply an array that contains one associative array (or *hash table*) for each record. This type of output is great for evaluations using the JavaScript `eval` function.

Unfortunately, this example won't work using a `<script>` tag, because this JavaScript code does nothing beyond create an array. To be useful in a `<script>` tag, the JavaScript code must invoke a method or function with the data, as in

Listing 16.

Listing 16. Jsadd.php

```
<?php
  require_once("DB.php");

  header( "Content-type: text/javascript" );

    $dsn = 'mysql://root:password@localhost/ajaxdb';
    $db =& DB::Connect( $dsn, array() );
    if (PEAR::isError($db)) { die($db->getMessage()); }

      $res = $db->query( "SELECT author, title FROM books" );
      $items = array();
    while( $res->fetchInto( $row ) )
    {
      $items []= "{ author: '".$row[0]."', title: '".$row[1]."' }";
    }
?>
addData( 'jsadd.php', [ <?php echo( join( ",\n", $items ) ); ?> ] );
```

This code is much the same as Listing 14 with the exception that the JavaScript code calls the `addData` function with the data. When you run jsadd.php on the command line, you get the output in Listing 17.

Listing 17. Output from jsadd.php

```
% php jsadd.php
  addData( 'jsadd.php', [ { author: 'Jack Herrington',
  title: 'Code Generation in Action' },
    { author: 'Jack Herrington', title: 'Podcasting Hacks' },
    { author: 'Jack Herrington', title: 'PHP Hacks' } ] );
%
```

This code is great for `<script>` tags, but what about iframes? To pass JavaScript data to an iframe, the JavaScript data must be encoded as HTML, as in Listing 18.

Listing 18. Js_text.php

```
<?php
  require_once("DB.php");

  header( "Content-type: text/html" );

    $dsn = 'mysql://root:password@localhost/ajaxdb';
    $db =& DB::Connect( $dsn, array() );
    if (PEAR::isError($db)) { die($db->getMessage()); }

      $res = $db->query( "SELECT author, title FROM books" );
      $items = array();
    while( $res->fetchInto( $row ) )
    {
```

```
      $items []= "{ author: '".$row[0]."', title: '".$row[1]."' }";
    }
?>
[ <?php echo( join( ",\n", $items ) ); ?> ];
```

The only change here from Listing 14 is that the MIME type has changed to `html`. Listing 19 shows the output of js_text.php.

Listing 19. Output from js_text.php

```
% php js_text.php
  [ { author: 'Jack Herrington', title: 'Code Generation in Action' },
  { author: 'Jack Herrington', title: 'Podcasting Hacks' },
  { author: 'Jack Herrington', title: 'PHP Hacks' } ];
%
```

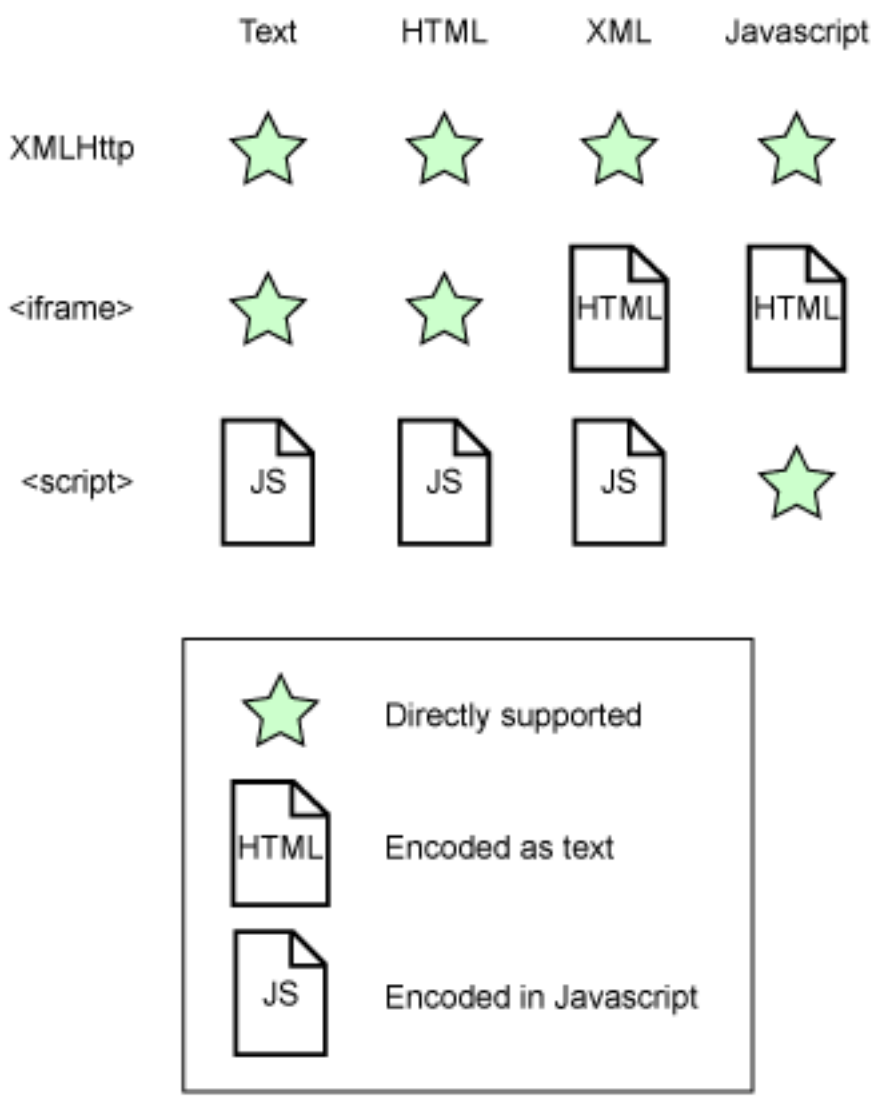
The real advantage of JavaScript data in the Ajax world is speed. It can take up to half a second to parse through XML on a reasonably fast machine, while the equivalent amount of data encoded as JavaScript is evaluated into arrays and associative arrays in just a few milliseconds.

The next step in this process is to build the clients for all this data.

Section 3. Building the client

Now that you have all the data feeds ready for the client, you must tackle how to read the data from the server using the three Ajax methods. Figure 1 shows the data formats and how the different transport methods consume them.

Figure 1. The Ajax transport compatibility chart



On its face, the `XMLHttp` transport method seems the best choice. However, because of the security issues in the browser, it's important that you know the two alternative methods, as well. The sections that follow cover the transport mechanisms and show the client-side code they use to consume each data type.

The XMLHttp client

The `XMLHttp` method is an easy way to move data to and from your Web server. It supports the `GET` and `POST` methods, so you can transfer large volumes of data. In addition, all current browsers support it.

Note: Internet Explorer requires some fiddling to get to an `XMLHttp`-equivalent

object, but the new version of Internet Explorer -- version 7 -- will support the standard method that Firefox, Opera, Safari, and others support. To solve one issue with Safari caching `XMLHttp` requests, add a random value as a URL parameter to avoid the cache.

The big advantage of the `XMLHttp` method is its simplicity. The big disadvantage is the security restriction on domains. If your page comes from www.mysite.com, your script cannot then request data from www.cnn.com or even data.mysite.com. You can only make requests to www.mysite.com, which means that creating an RSS reader that resides just on the client is impossible. You need a proxy page on the server that retrieves data from www.cnn.com and returns it through www.mysite.com.

But suppose that the security restriction is just fine with you. How do you get the data?

Use the text service through XMLHttp

The DHTML code in Listing 20 retrieves data from the text.php page and displays it in a browser.

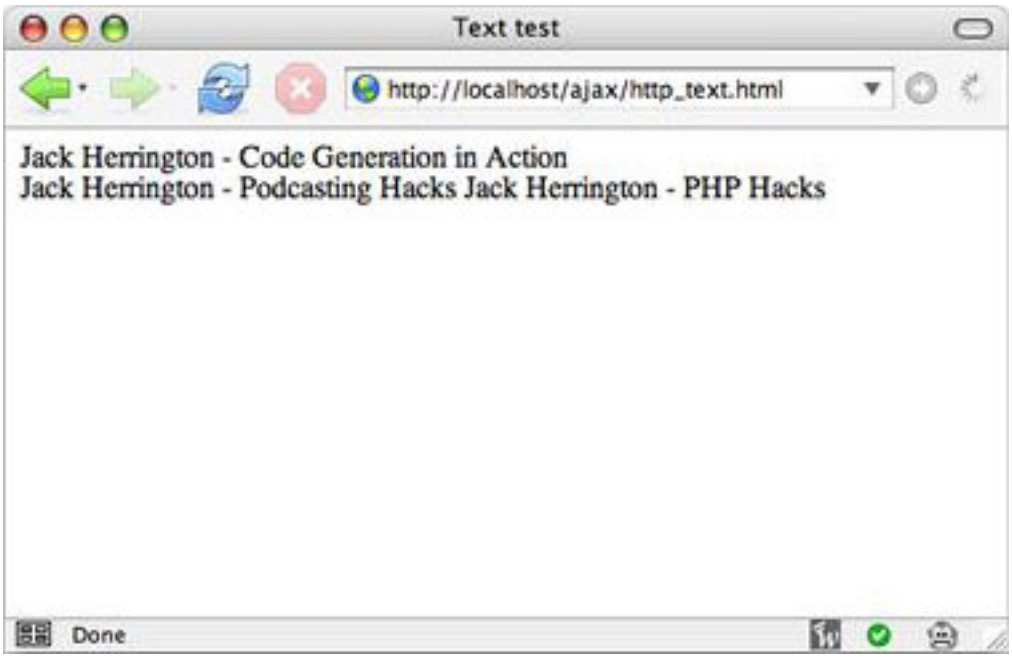
Listing 20. Http_text.html

```
<html>
  <title>Text test</title>
    <head>
      <script>
        var req = null;
        function processReqChange()
        {
          if (req.readyState == 4 && req.status == 200 )
          {
            var dobj = document.getElementById( "dataDiv" );
            var text = req.responseText;
            text = text.replace( "\n", "<br/>" );
            dobj.innerHTML = text;
          }
        }

        function loadUrl( url )
        {
          if(window.XMLHttpRequest) {
            try { req = new XMLHttpRequest();
            } catch(e) { req = false; }
          }
          else if(window.ActiveXObject)
          {
            try { req = new ActiveXObject("Msxml2.XMLHTTP");
            } catch(e) {
            try { req = new ActiveXObject("Microsoft.XMLHTTP");
            } catch(e) { req = false; }
          }
        }
```

```
          if(req) {
            req.onreadystatechange = processReqChange;
            req.open("GET", url, true);
            req.send("");
          }
        }

        var url = window.location.toString();
        url = url.replace( /\/.*?$/, "sources/text.php" );
        loadUrl( url );
      </script>
    <body>
      <div id="dataDiv">
      </div>
    </body>
</html>
```

Figure 2 shows the result.

Figure 2. The http_text test page



This is the simplest possible `XMLHttp` request. The `loadUrl` function creates an `XMLHttpRequest` object, and then uses the `open` function to start the data download. When the process is complete, the `processReqChange` function is called to check the status of the request. If the request is complete, the function adds the text to the HTML document through the `dataDiv` element. The regular expression changes the carriage return in the text to a `<br>` tag, so that the text flows nicely instead of being on one long line.

Use the HTML service through XMLHttp

Another popular method that uses the `XMLHttp` object is to request updated HTML markup data from the server. The new Atlas framework from Microsoft uses this method. The idea is that when you updated sections of the page, the page calls back to the server for new HTML, then updates the sections of the page that contain that HTML fragment. Listing 21 shows this process.

Listing 21. Http_html.html

```
<html>
  <title>HTML test</title>
    <head>
      <style>
        td { border: 1px solid #666; padding: 3px; }
      </style>
      <script>
        var req = null;
        function processReqChange()
        {
          if (req.readyState == 4 && req.status == 200 )
          {
            var dobj = document.getElementById( "dataDiv" );
            dobj.innerHTML = req.responseText;
          }
        }

        function loadUrl( url )
        {
          ...
        }

        var url = window.location.toString();
        url = url.replace( /\/.*?$/, "sources/html.php" );
          loadUrl( url );
      </script>
    <body>
      <div id="dataDiv">
      </div>
    </body>
</html>
```

The code for `loadUrl` is the same as that in Listing 20. The big difference here is that the `processReqChange` function uses the `responseText` function to fill the `innerHTML` of the `dataDiv` element, thereby updating the content of the page.

Use the XML and RSS services through XMLHttp

By far, one of the most popular uses of the `XMLHttp` object is to transfer XML of a given format to and from a browser. Because the request object returned has an XML Document Object Model (DOM) element built in, you can use it to search through the XML for new data.

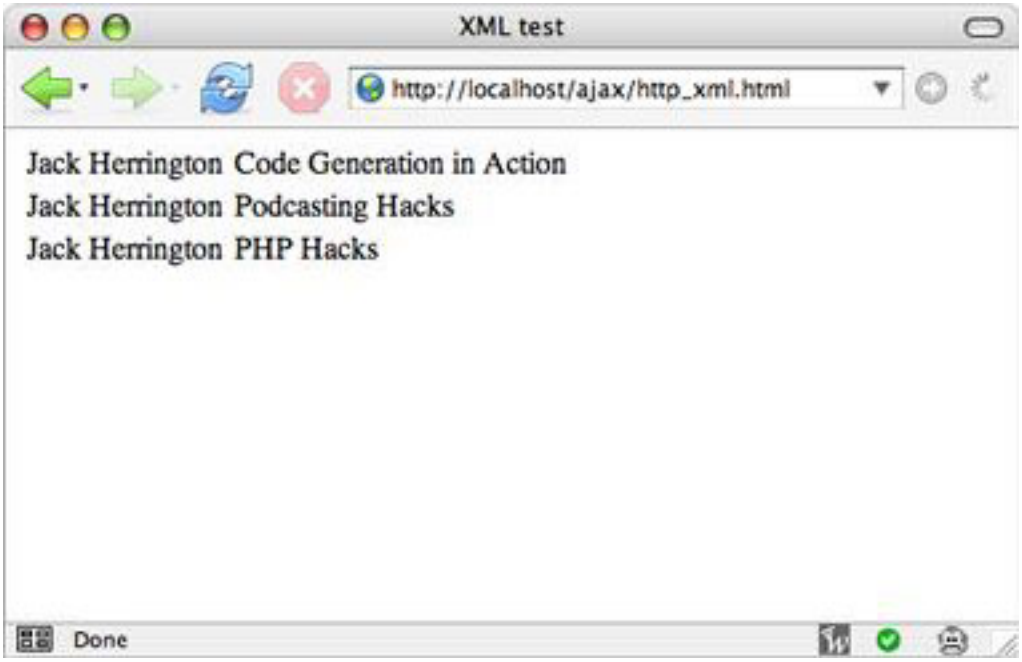
Listing 22 shows DHTML code that requests the book data from the server in the custom XML format.

Listing 22. Http_xml.html

```
<html>
  <title>XML test</title>
    <head>
      <script>
        var req = null;
        function processReqChange()
        {
          if (req.readyState == 4 && req.status == 200 && req.responseXML )
          {
            var dobj = document.getElementById( "dataBody" );

            var nl = req.responseXML.getElementsByTagName( 'book' );
            for( var i = 0; i < nl.length; i++ )
            {
                   var nli = nl.item( i );
              var elAuthor = nli.getElementsByTagName( 'author' );
              var author = elAuthor.item(0).firstChild.nodeValue;
              var elTitle = nli.getElementsByTagName( 'title' );
              var title = elTitle.item(0).firstChild.nodeValue;

              var elTr = document.createElement( 'tr' );
              dobj.appendChild( elTr );

              var elAuthorTd = document.createElement( 'td' );
              elAuthorTd.innerHTML = author;
              elTr.appendChild( elAuthorTd );

              var elTitleTd = document.createElement( 'td' );
              elTitleTd.innerHTML = title;
              elTr.appendChild( elTitleTd );
            }
          }
        }

        function loadXMLDoc( url )
        {
          ...
        }

            var url = window.location.toString();
            url = url.replace( /\/.*?$/, "sources/xml.php" );
            loadXMLDoc( url );
      </script>
    <body>
      <table cellspacing="0" cellpadding="3">
         <tbody id="dataBody">
         </tbody>
      </table>
    </body>
</html>
```

(The `loadXMLDoc` function is the same as the `loadUrl` from previous examples and is omitted for brevity's sake.) The tricky work is done in the `processReqChange` function, where the JavaScript code uses the `getElementsByTagName` function to find the `<book>`, `<title>` and `<author>` tags in the XML. Figure 3 shows the output of the code.

Figure 3. The finished http_xml page



Listing 23 shows the `processReqChange` code that parses the RSS version of the page.

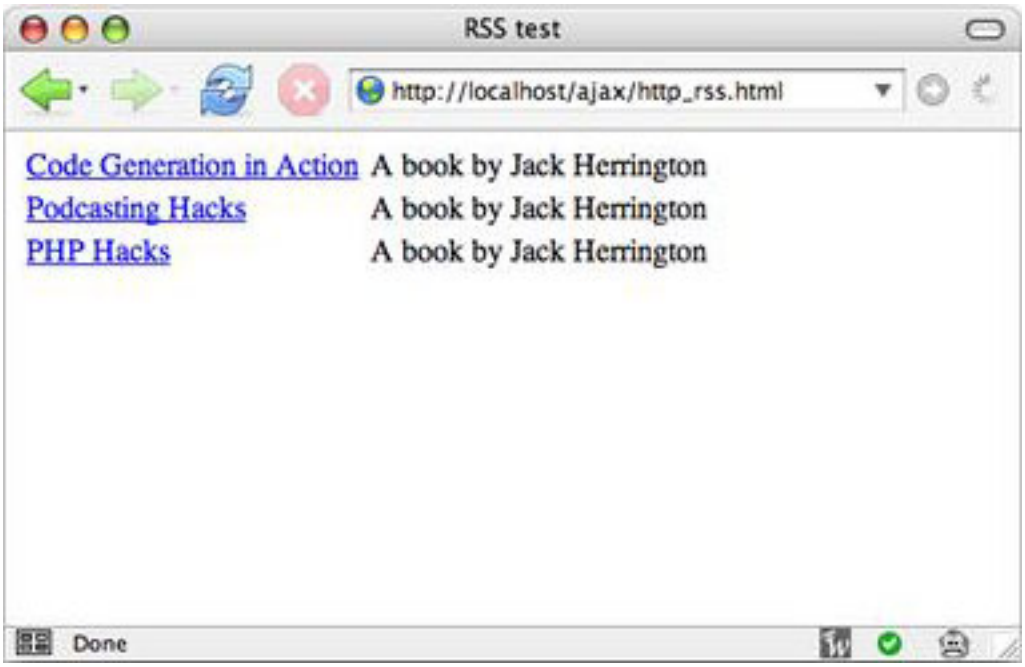
Listing 23. The processReqChange from http_rss.html

```
...
function processReqChange()
{
  if (req.readyState == 4 && req.status == 200 && req.responseXML )
  {
    var dobj = document.getElementById( "dataBody" );

    var nl = req.responseXML.getElementsByTagName( 'item' );
    for( var i = 0; i < nl.length; i++ )
    {
      var nli = nl.item( i );
      var elDescription = nli.getElementsByTagName( 'description' );
      var description = elDescription.item(0).firstChild.nodeValue;
      var elTitle = nli.getElementsByTagName( 'title' );
      var title = elTitle.item(0).firstChild.nodeValue;
      var elLink = nli.getElementsByTagName( 'link' );
      var link = elLink.item(0).firstChild.nodeValue;

      var elTr = document.createElement( 'tr' );
      dobj.appendChild( elTr );

      var elTitleTd = document.createElement( 'td' );
      elTr.appendChild( elTitleTd );

      var elLink = document.createElement( 'a' );
      elLink.innerHTML = title;
      elLink.href = link;
      elTitleTd.appendChild( elLink );
```

```
      var elDescriptionTd = document.createElement( 'td' );
      elDescriptionTd.innerHTML = description;
      elTr.appendChild( elDescriptionTd );
      }
    }
  }
    ...
```

The big difference here is that the code finds an addition link value and uses it to create an `<anchor>` tag around the title in the table. The result is shown in Figure 4.

Figure 4. The finished http_rss.html page



The advantage of using RSS for the transport is twofold. First, RSS clients can monitor the RSS; second, you can apply this HTML and JavaScript code to any RSS-formatted feed as long as the data comes from the same domain as the one that served the page.

Use JavaScript code and the JSON service through XMLHttp

To finish your exploration of the `XMLHttp` transport method, you'll use JavaScript-formatted data. The first example (Listing 24) shows the `processReqChange` function that handles data coming in from the js.php page that returns JavaScript code (or JSON, if you want to be trendy).

Listing 24. The processReqChange from the http_js.html page

```
...
  function processReqChange()
  {
    if (req.readyState == 4 && req.status == 200 && req.responseText )
    {
      var books = eval( req.responseText );

      var dobj = document.getElementById( "dataBody" );
      for( var b in books )
      {
      var elTr = document.createElement( 'tr' );
      dobj.appendChild( elTr );

      var elTitleTd = document.createElement( 'td' );
      elTitleTd.innerHTML = books[b].author;
      elTr.appendChild( elTitleTd );

      var elDescriptionTd = document.createElement( 'td' );
      elDescriptionTd.innerHTML = books[b].title;
      elTr.appendChild( elDescriptionTd );
      }
    }
  }
...
```

Note how much easier it is to parse the JavaScript code returned from js.php than it is to parse XML. You simply run the `eval` function on the code and take the returned value, which is the array of hash tables, then walk through them using a standard JavaScript `for` loop.

Another alternative for transferring JavaScript code is to use the type of JavaScript code shown in Listing 16, which calls a JavaScript function during evaluation. The JavaScript code that parses this function is in Listing 25.

Listing 25. The processReqChange and addData from http_jsadd.html

```
...
  function addData( url, books )
  {
    var dobj = document.getElementById( "dataBody" );
    for( var b in books )
    {
      var elTr = document.createElement( 'tr' );
      dobj.appendChild( elTr );

      var elTitleTd = document.createElement( 'td' );
      elTitleTd.innerHTML = books[b].author;
      elTr.appendChild( elTitleTd );

      var elDescriptionTd = document.createElement( 'td' );
      elDescriptionTd.innerHTML = books[b].title;
      elTr.appendChild( elDescriptionTd );
    }
  }

  function processReqChange()
  {
    if (req.readyState == 4 && req.status == 200 && req.responseText )
```

```
        eval( req.responseText );
    }
...
```

As you can see, the code that was in the `processReqChange` function has now moved into the `addData` function.

This roundup of the uses of `XMLHttp` should give you a nice recipe book for using the `XMLHttp` object in your application with a variety of data formats.

Section 4. The iframe client

A more traditional (if you can call it that) method of transporting data to and from the server after the page loads is to use a `<frame>` or `<iframe>` tag. I strongly recommend using an invisible frame (the `<iframe>` tag), because it's easier and doesn't affect the layout of the page the way a frame does.

The advantage of a frame is that you can use it in the ever-diminishing number of browsers that don't support the `XMLHttp` object. You can also use a frame to add items to the user's page history so that the back button actually works on your Ajax page. Unfortunately, the disadvantages are numerous. Using frames for transport works easily only for transmitting HTML, although with some hacking you can get JavaScript code and XML across the wire, as well.

With the iframe approach, you can get data to the server in either of two ways. The first way is through the URL arguments associated with the `src` attribute on the `<iframe>` tag. The second way is to create a `<form>` tag with associated `<input>` elements inside the iframe document, then use the `submit()` method on the `<form>` tag to `POST` or `GET` data to the server. This tutorial demonstrates the `src` attribute.

Use the HTML service through an iframe

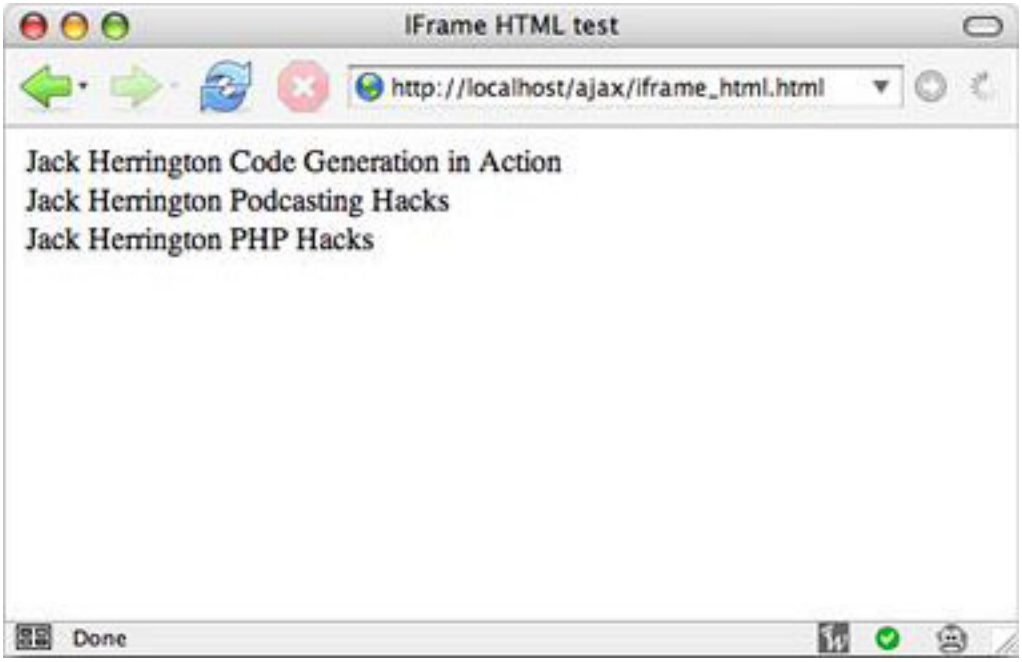
The code in Listing 26 demonstrates the use of an iframe to load the data in the html.php page.

Listing 26. Iframe_html.html

```
<html>
 <title>Iframe HTML test</title>
   <head>
    <script>
     function iframe_loaded( frame )
     {
       var dobj = document.getElementById( "dataDiv" );
       dobj.innerHTML = frame.contentWindow.document.body.innerHTML;
     }

       var urlsToLoad = [];

       function processRequests()
       {
        for( var u in urlsToLoad )
        {
         var sObj = document.createElement( 'iframe' );
         sObj.src = urlsToLoad[ u ];
         sObj.onload = function() { iframe_loaded( sObj ); };
         if ( sObj.attachEvent != null )
          sObj.attachEvent( 'onload', function() { iframe_loaded( sObj
); } );
         sObj.style.visibility = 'hidden';
         sObj.style.display = 'none';
         document.body.appendChild( sObj );
        }
       }

       function loadUrl( url )
       {
        urlsToLoad.push( url );
       }

       if ( window.addEventListener )
        window.addEventListener( 'load', processRequests, 0 );
       else
        window.attachEvent( 'onload', processRequests );

       var url = window.location.toString();
       url = url.replace( /\/.*?$/, "sources/html.php" );
       loadUrl( url );
    </script>
   <body>
    <div id="dataDiv"></div>
   </body>
</html>
```

If you point your browser to iframe_html.php, you see the result in Figure 5.

Figure 5. The iframe_html.html page



So, how does this page work? First, the code calls the `loadUrl` function, which puts the URL request in a queue of requests that to execute after the page loads. The `processRequests` function is called when the page loads; its job is to create the `<iframe>` tags for each request.

To verify when the iframe is loaded and the data is ready, the code sets the `onload` function of the iframe and uses the `attachEvent` function. This function works for both Internet Explorer and Firefox. The `iframe_loaded` function uses the DOM to get the contents of the page, and then sets the contents of the `dataDiv` element to the contents of the iframe.

Use the JavaScript service through an iframe

With the easy tasks out of the way, it's time to hack the `iframe` method to get it to support JavaScript code. The first hack occurs on the server side, where you encode the JavaScript code as HTML. The second hack comes in how you get to the text of the iframe document that holds the JavaScript code. Listing 27 shows this hack.

Listing 27. The iframe_loaded function in iframe_js.html

```
...
function iframe_loaded( frame )
{
  var books = null;
```

```
  if ( frame.contentDocument )
    books = eval( frame.contentDocument.body.textContent );
  else
    books = eval( frame.contentWindow.document.body.innerText );

  var dobj = document.getElementById( "dataBody" );
  for( var b in books )
  {
    var elTr = document.createElement( 'tr' );
    dobj.appendChild( elTr );

    var elTitleTd = document.createElement( 'td' );
    elTitleTd.innerHTML = books[b].author;
    elTr.appendChild( elTitleTd );

    var elDescriptionTd = document.createElement( 'td' );
    elDescriptionTd.innerHTML = books[b].title;
    elTr.appendChild( elDescriptionTd );
  }
}
...
```

These hacks are required primarily for Internet Explorer, which causes a window to appear when it attempts to download a JavaScript file that is returned with the `text/javascript` MIME type. Outside these hacks, the `iframe` method uses an `eval` method similar to that used in the `XMLHttp` version of the JavaScript reading code.

Use the XML service through iframes

The toughest thing to move through the `iframe` method is XML. Like moving JavaScript code, the hack is on both server and client. First, the XML must be encoded as an HTML string. Then, the client must use the XML processor inside the client to turn the text string inside the `iframe` method into an XML document. This hack is shown in the `iframe_loaded` function in Listing 28.

Listing 28. The iframe_loaded function in iframe_xml.html

```
...
function iframe_loaded( frame )
{
  var dobj = document.getElementById( "dataBody" );

  var xmlDoc = null;
  if( frame.contentDocument )
  {
    // Mozilla/Firefox
    var xml = frame.contentDocument.body.textContent;
    xmlDoc = new DOMParser().parseFromString( xml, "text/xml" );
  }
  else
  {
    var xml = frame.contentWindow.document.body.innerText;
    xmlDoc = new ActiveXObject( 'MSXML2.DOMDocument' );
```

```
    xmlDoc.loadXML( xml );
  }

  var nl = xmlDoc.getElementsByTagName( 'book' );
  for( var i = 0; i < nl.length; i++ )
  {
    var nli = nl.item( i );
    var elAuthor = nli.getElementsByTagName( 'author' );
    var author = elAuthor.item(0).firstChild.nodeValue;
    var elTitle = nli.getElementsByTagName( 'title' );
    var title = elTitle.item(0).firstChild.nodeValue;

    var elTr = document.createElement( 'tr' );
    dobj.appendChild( elTr );

    var elAuthorTd = document.createElement( 'td' );
    elAuthorTd.innerHTML = author;
    elTr.appendChild( elAuthorTd );

    var elTitleTd = document.createElement( 'td' );
    elTitleTd.innerHTML = title;
    elTr.appendChild( elTitleTd );
  }
}
...
```

Unfortunately, Internet Explorer and Firefox have two different ways of parsing the XML in a string into an XML DOM. That's where the check on `frame.contentDocument` comes into play. After the XML is turned into a DOM, the code is similar to the `XMLHttp` pages that read through the XML DOM and create the HTML.

Section 5. The script tag client

The third and final approach is to use `<script>` tags to download data encoded in JavaScript code from the server. You can send any data type -- text, XML, or JavaScript-formatted data structures -- over the wire using a `<script>` tag.

The advantage of the `<script>` tag approach is huge: You can request data from any server. This benefit is demonstrated through services such as Google Maps. The `<script>` tag approach is the only way to allow people who use your Web site to copy and paste Ajax code fragments from your site and have them run on their own sites while accessing your data. You can post large quantities of data through the `<script>` tag.

One disadvantage, however, is that formats other than JavaScript require some JavaScript encoding to pass from the server to the client. Another, far more

significant disadvantage is that only the GET protocol is available through this approach.

Use the JavaScript service through <script> tags

The code that uses a <script> tag to access data from the jsadd.php page is in Listing 29.

Listing 29. The script.html page

```
<html>
  <title>Script tag test #2</title>
    <head>
      <script>
        function addData( url, books )
        {
          var dobj = document.getElementById( "dataBody" );
          for( var b in books )
          {
            var elTr = document.createElement( 'tr' );
            dobj.appendChild( elTr );

            var elTitleTd = document.createElement( 'td' );
            elTitleTd.innerHTML = books[b].author;
            elTr.appendChild( elTitleTd );

            var elDescriptionTd = document.createElement( 'td' );
            elDescriptionTd.innerHTML = books[b].title;
            elTr.appendChild( elDescriptionTd );
          }
        }

        var urlsToLoad = [];

        function processRequests()
        {
          for( var u in urlsToLoad )
          {
            var sObj = document.createElement( 'script' );
            sObj.src = urlsToLoad[ u ];
            document.body.appendChild( sObj );
          }
        }

        function loadDoc( url )
        {
          urlsToLoad.push( url );
        }

        if ( window.addEventListener )
          window.addEventListener( 'load', processRequests, 0 );
        else
          window.attachEvent( 'onload', processRequests );

        var url = window.location.toString();
        url = url.replace( /\/.*?$/, "sources/jsadd.php" );
        loadDoc( url );
      </script>
```

```
    <body>
      <table cellspacing="0" cellpadding="3">
        <tbody id="dataBody">
        </tbody>
          </table>
    </body>
</html>
```

This code is similar in form to the iframe approach. It uses the `onload` handler to add `<script>` tags to the document `<body>` tag. The dynamically added `<script>` tags then request the jsadd.php page, which includes JavaScript code that calls the `addData` function. That function then adds the data to the display using the `createElement()` and `appendChild()` methods, which you've seen throughout this tutorial.

Overall, the `<script>` tag approach is the most flexible and the easiest method for getting data from the server dynamically, although I recommend spending a little time on the server side putting together code that takes data structures in the server language and translates them into JavaScript structures with proper encoding. Many of the dynamic programming languages include packages for this task. (The JSON Web site (http://json.org) is a good resource, as well. See Resources.)

Section 6. Summary

Venture forward to Web V2.0 when you transport data with Ajax

This tutorial demonstrated three unique methods to get data from your server without reloading the page. From my own experience, I recommend either using the `XMLHttp` or the `<script>` tag method. In either case, I prefer to use JavaScript as the transport language, because it's far easier and faster for the client to interpret.

As to the method of transport, I recommend the `<script>` tag approach, because it allows users on your site to use View Source functionality on your page, grab your code, and use it on their own pages to display your data in their context. As long as it's attributed appropriately, this borrowing can give your site viral reach, and Web V2.0 is all about viral marketing.

That said, if this tutorial is anything, it's a demonstration of just how many ways to handle simply the data-transport portion of the Ajax equation. You must decide for

yourself the most appropriate method for your Web V2.0 applications.

Downloads

Description	Name	Size	Download method
HTML source files for Ajax transformation	x-ajaxtrans-tutorial-html_files.zip	8KB	HTTP
SQL and PHP source files	x-ajaxtrans-tutorial-source_files.zip	5KB	HTTP

Information about download methods

Resources

Learn

- For more information about the PHP programming language, check out the PHP home page.
- Visit the XML page on the World Wide Web Consortium (W3C) site, an excellent spot for learning about XML standards and technologies.
- At JSON.org, learn what you need to know about the JavaScript Object Notation and find references to server libraries for creating JSON-encoded data.
- For more information about RSS 1.0, read the RSS 1.0 specification. For more information about RSS 2.0, read the RSS 2.0 specification.
- Explore RDF as defined on the W3C site's RDF page.
- Visit the developerWorks XML zone to expand your XML skills.
- Stay current with developerWorks technical events and Webcasts.

Get products and technologies

- Build your next development project with IBM trial software, available for download directly from developerWorks.

Discuss

- Participate in developerWorks blogs and get involved in the developerWorks community.

About the author

Mr. Jack D Herrington
A senior software engineer with more than 20 years of experience, Jack Herrington is the author of three books: *Code Generation in Action, Podcasting Hacks,* and *PHP Hacks.* He is also the author of more than 30 articles. You can contact him at jherr@pobox.com.