The Essentials of XMLHttpRequest and XML Programming with Java

Ajax on Java™

O'REILLY®

Steven Douglas Olson

Ajax on Java
by Steven Douglas Olson

Copyright © 2007 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*.

Editor: Mike Loukides
Executive Editor: Mike Loukides
Production Editor: Lydia Onofrei
Copyeditor: Rachel Wheeler
Proofreader: Lydia Onofrei
Indexer: Johnna VanHoose Dinse
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrators: Robert Romano and Jessamyn Read

Printing History:

February 2007: First Edition.

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

ISBN-10: 0-596-10187-2
ISBN-13: 978-0-596-10187-9
[M]

CHAPTER 4

XML and JSON for Ajax

Do you really need XML for an Ajax application? The previous chapter showed that you don't always need XML. In particular, if you only have one data point, XML is overkill. But the fact is, most web applications deal with multiple data points: usernames, passwords, addresses, cities, states, zip codes, etc. How will you decipher those fields when they're sent back from the server?

In some cases, passing a string of delimited values may seem like the simplest approach, but using XML has advantages. For one thing, XML is self-documenting. During debugging, you can look at the XML string and see exactly what goes where; that is a luxury you won't have with a string of comma-separated values.

Another reason for using XML is that an XML parser is built into most browsers. The parsing work has already been done for you; all you have to do is leverage the built-in parser. Sure, you could pass the data in other formats—Java properties files, comma or tab-separated values, YAML files, or a cute custom format that you've designed yourself—but then you would have to write your own parser in JavaScript.

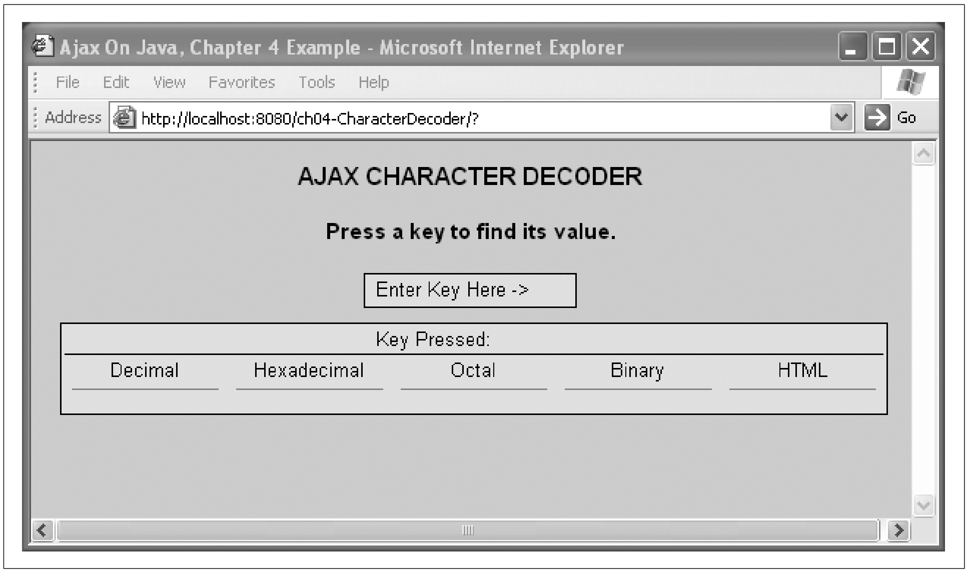
There is another good way to send data to and from the server: JavaScript Object Notation (JSON). We will discuss JSON toward the end of this chapter.

The Character Decoder

The example in this chapter is similar to the one in the previous chapter, but instead of the server returning one data point, it's going to return five. Retuning a small collection of data shows what happens when you go beyond a single data point and illustrates why most Ajax applications need XML or some other way to structure the data that is passed from the server to the client.

Figure 4-1 shows how the user interface of the application will look when we're done. The design is simple enough: we send a character to the server using

XMLHttpRequest( ), and the server responds with a String containing the five conversions in XML format (decimal, octal, hexadecimal, binary, and HTML). The callback( ) function in the client then calls a function to parse the XML and populate the fields in the browser.


Ajax On Java, Chapter 4 Example - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://localhost:8080/ch04-CharacterDecoder/?
Go

AJAX CHARACTER DECODER

Press a key to find its value.

Enter Key Here ->

Key Pressed:

Decimal	Hexadecimal	Octal	Binary	HTML

Figure 4-1. The complete Ajax Character Decoder example

Now it starts to get fun. One keystroke fills in all the data fields, and although it doesn't look like much is going on from the user's perspective, from the programmer's perspective we know that the application is communicating with the server without a clunky submit or reload or any waiting for the page to refresh.

Setting Up a Simple XML Document

Before we delve into the code, we need to make some decisions. We're going to return data using XML, but how should that XML be structured? What should our XML response look like? We don't want anything complex, so we'll aim to create an XML document that looks like this:

```
<converted-values>
    <decimal>97</decimal>
    <hexadecimal>0x61</hexadecimal>
    <octal>0141</octal>
    <hyper>&amp;0x61;</hyper>
    <binary>1100001B</binary>
</converted-values>
```

With this format, the browser can use its document object model (DOM) parser to index and retrieve the data.

There are many ways to create this XML document. For the sake of simplicity, we'll first use a `StringBuffer` to wrap the data with XML tags. Later, we'll look at other ways to create the XML document.

When I talk about XML formatting, I'm referring to the server wrapping the data in XML. The client receives the XML-formatted string in the `HTTPResponse` and parses it for the individual data fields. The client passes data through the request using either `HTTPPost( )` or `HTTPGet( )`. There is no reason for the client to send XML data to the server, because the data is already wrapped in the request as name/value pairs.

Using a Servlet to Build an XML Document

Let's start by looking at the servlet code that wraps the data in XML. This servlet is shown in Example 4-1.

Example 4-1. The AjaxResponseServlet

```
/*
 * Converts a character to hex, decimal, binary, octal, and HTML, then
 * wraps each of the fields with XML and sends them back through the response.
 */
package com.AJAXbook.servlet;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;


public class AjaxResponseServlet extends HttpServlet {

    public void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
        // key is the parameter passed in from the JavaScript
        // variable named url (see index.html)
        String key = req.getParameter("key");
        StringBuffer returnXML = null;
        if (key != null) {
            // extract the first character from key
            // as an int, then convert that int to a String
            int keyInt = key.charAt(0);
            returnXML = new StringBuffer("\r\n<converted-values>");
            returnXML.append("\r\n<decimal>"+
                             Integer.toString(keyInt)+"</decimal>");
            returnXML.append("\r\n<hexadecimal>0x"+
                             Integer.toString(keyInt,16)+"</hexadecimal>");
            returnXML.append("\r\n<octal>0"+
                             Integer.toString(keyInt,8)+"</octal>");
```

Example 4-1. The AjaxResponseServlet (continued)

```
            returnXML.append("\r\n<hyper>&amp;0x"+
                            Integer.toString(keyInt,16)+";</hyper>");
            returnXML.append("\r\n<binary>"+
                            Integer.toString(keyInt,2)+"B</binary>");
            returnXML.append("\r\n</converted-values>");

            // set up the response
            res.setContentType("text/xml");
            res.setHeader("Cache-Control", "no-cache");
            // write out the XML string
            res.getWriter().write(returnXML.toString( ));
        }
        else {
            // if key comes back as a null, return a question mark
            res.setContentType("text/xml");
            res.setHeader("Cache-Control", "no-cache");
            res.getWriter( ).write("?");
        }
    }
}
```

This code is similar to the code from Chapter 3. The only thing that has been added is the code to wrap the data with XML tags:

```
returnXML = new StringBuffer("\r\n<converted-values>");
returnXML.append("\r\n<decimal>"+
                Integer.toString(keyInt)+"</decimal>");
returnXML.append("\r\n<hexadecimal>0x"+
                Integer.toString(keyInt,16)+"</hexadecimal>");
returnXML.append("\r\n<octal>0"+
                Integer.toString(keyInt,8)+"</octal>");
returnXML.append("\r\n<hyper>&amp;0x"+
                Integer.toString(keyInt,16)+";</hyper>");
returnXML.append("\r\n<binary>"+
                Integer.toString(keyInt,2)+"B</binary>");
returnXML.append("\r\n</converted-values>");
```

This code simply sets up a `StringBuffer` called `returnXML`. We then convert the incoming value to decimal, hex, etc.; wrap it with an appropriate XML tag; and append it to the buffer. When we've finished all five conversions and added the closing tag (`</converted-values>`), we send the response back to the Ajax client using `res.getWriter().write( )`. We return a question mark (without any XML wrapping) if the key we received was `null`.

Other Ways to Build the XML Document

Building an XML document by appending to a `StringBuffer` is a common approach, but it's far from ideal, particularly if you need to generate a large document programmatically. Fortunately, there are alternatives.

JDOM

One option is to use the JDOM library to write the XML. Download the *jdom.jar* file from *http://www.jdom.org*, and put it in your application's *WEB-INF/lib* directory. Then, instead of writing to a `StringBuffer`, use JDOM to build the XML, as shown in Example 4-2.

Example 4-2. Using JDOM to create the XML document

```
// additional imports needed for JDOM
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.output.XMLOutputter;

public String createJdomXML(int key) throws IOException {
    Document document = new Document( );
    // create root node
    Element root = new org.jdom.Element("converted-values");
    document.setRootElement(root);

    // create your node
    org.jdom.Element element = new org.jdom.Element("decimal");
    // add content to the node
    element.addContent(Integer.toString(key));
    // add your node to root
    root.addContent(element);

    element = new org.jdom.Element("hexadecimal");
    element.addContent("0x" + Integer.toString(key, 16));
    root.addContent(element);
    element = new org.jdom.Element("octal");
    element.addContent("0" + Integer.toString(key, 8));
    root.addContent(element);
    element = new org.jdom.Element("hyper");
    element.addContent("&0x" + Integer.toString(key, 16));
    root.addContent(element);
    element = new org.jdom.Element("binary");
    element.addContent(Integer.toString(key, 2) + "B");
    root.addContent(element);

    // output JDOM document as a String of bytes
    XMLOutputter outputter = new XMLOutputter( );
    return outputter.outputString(document);
}
```

In the preceding code, we first create a `Document` (`org.jdom.Document`), then an `Element` named `root` with the `String` `"converted-values"` as its value. That element becomes the root of the XML document. Here's what the document looks like at this point:

```
<converted-values>
</converted-values>
```

To add child elements to the root, we create new elements and add them to the root element. The code that creates the `decimal` element and adds it to the root element looks like this:

```
org.jdom.Element element = new org.jdom.Element("decimal");
element.addContent(Integer.toString(key));
root.addContent(element);
```

We repeat this process until we've added all the elements to the root. Then we use an `XMLOutputter` to format the document into a `String` that we can send back to the client. The JDOM XML document now looks like this (with linefeeds and spaces added for readability):

```
<?xml version="1.0" encoding="UTF-8"?>
<converted-values>
    <decimal>97</decimal>
    <hexadecimal>0x61</hexadecimal>
    <octal>0141</octal>
    <hyper>&amp;0x61</hyper>
    <binary>1100001B</binary>
</converted-values>
```

dom4j

dom4j is an XML library similar in intent to JDOM. After downloading dom4j from *http://www.dom4j.org/download.html* and installing it in your application's *WEB-INF/lib* directory, you can use it to create your XML document. As shown in Example 4-3, we create a document, add a root element to the document, add the elements and data to the root, and then return the document in a `String`.

Example 4-3. Using dom4j to create the XML document

```
// additional imports for dom4j
import org.dom4j.Document;
import org.dom4j.DocumentHelper;
import org.dom4j.DocumentException;
import org.dom4j.Element;
import org.dom4j.io.OutputFormat;
import org.dom4j.io.XMLWriter;

...

public String createDom4jXML(int key) throws IOException {
    Document document = DocumentHelper.createDocument( );
    Element root = document.addElement("converted-values");

    Element element = root.addElement("decimal").addText(
            Integer.toString(key));
    element = root.addElement("hexadecimal").addText(
            "0x" + Integer.toString(key, 16));
    element = root.addElement("octal").addText("0" + Integer.toString(key, 8));
    element = root.addElement("hyper").addText("&0x" + Integer.toString(key, 16));
```

Example 4-3. Using dom4j to create the XML document (continued)

```
    element = root.addElement("binary").addText(Integer.toString(key, 2) + "B");
    StringBuffer xmlDoc = null;

    StringWriter sw = new StringWriter( );
    OutputFormat outformat = OutputFormat.createPrettyPrint( );
    XMLWriter writer = new XMLWriter(sw, outformat);
    writer.write(document);
    writer.close( );
    xmlDoc = sw.getBuffer( );

    return xmlDoc.toString( );
}
```

The dom4j library uses the static method `DocumentHelper.createDocument( )` to create the XML document. The method `root.addElement( )` puts a child element on the root element, and `addText( )` puts the data in the elements. The `OutputFormat` class is then used to format the `XMLDocument`, so the document looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<converted-values>
    <decimal>97</decimal>
    <hexadecimal>0x61</hexadecimal>
    <octal>0141</octal>
    <hyper>&amp;0x61</hyper>
    <binary>1100001B</binary>
</converted-values>
```

This step can be skipped, because it only formats the document for readability by adding linefeeds and spaces. Since humans shouldn't need to read this document (unless you are debugging), you won't need the formatting.

To use dom4j without the formatting, simply replace these two lines:

```
OutputFormat outformat = OutputFormat.createPrettyPrint( );
XMLWriter writer = new XMLWriter(sw, outformat);
```

with this line:

```
XMLWriter writer = new XMLWriter(sw);
```

SAX

SAX, the Simple API for XML, provides another way to create an XML document for an Ajax application. It may be faster than JDOM or dom4J, because it doesn't require building a DOM tree for your document. Start by initializing a `StringWriter` and a `StreamResult`. Initialize the `StreamResult` with the `StreamWriter`, then get a `SAXTransformerFactory` and get a `TransformerHandler` from that. The `TransformerHandler` allows you to create an XML document by starting a document and appending elements and data to the `TransformerHandler`. Example 4-4 shows how it works.

Example 4-4. Using SAX to write out the XML document

```
// additional imports for writing XML with SAX
import java.io.*;
import org.xml.sax.helpers.AttributesImpl;
import javax.xml.transform.sax.SAXTransformerFactory;
import javax.xml.transform.sax.TransformerHandler;

public String createSAXXML(int key) {
    Writer writer = new StringWriter( );
    StreamResult streamResult = new StreamResult(writer);

    SAXTransformerFactory transformerFactory =
            (SAXTransformerFactory) SAXTransformerFactory.newInstance( );
    try {
        String data = null;
        TransformerHandler transformerHandler =
                transformerFactory.newTransformerHandler( );

        transformerHandler.setResult(streamResult);
        // start the document
        transformerHandler.startDocument( );
        // list all the attributes for element
        AttributesImpl attr = new AttributesImpl( );
        // start writing elements
        // every start tag and end tag has to be defined explicitly
        transformerHandler.startElement(null,null, "converted-values", null);
        transformerHandler.startElement(null,null,"decimal",null);
        data = Integer.toString(key, 10);
        transformerHandler.characters(data.toCharArray(),0,data.length( ));

        transformerHandler.endElement(null,null,"decimal");

        transformerHandler.startElement(null,null,"hexadecimal",null);
        data = "0x" + Integer.toString(key, 16);
        transformerHandler.characters(data.toCharArray(),0,data.length( ));

        transformerHandler.endElement(null,null,"hexadecimal");
        transformerHandler.startElement(null,null,"octal",null);
        data = "0" + Integer.toString(key, 8);
        transformerHandler.characters(data.toCharArray(),0,data.length( ));

        transformerHandler.endElement(null,null,"octal");
        transformerHandler.startElement(null,null,"binary",null);
        data = Integer.toString(key, 2)+"B";
        transformerHandler.characters(data.toCharArray(),0,data.length( ));

        transformerHandler.endElement(null,null,"binary");
        transformerHandler.startElement(null,null,"hyper",null);
        data = "&0x" +Integer.toString(key, 16);
        transformerHandler.characters(data.toCharArray(),0,data.length( ));

        transformerHandler.endElement(null,null,"hyper");
        transformerHandler.endElement(null,null, "converted-values");
```

Example 4-4. Using SAX to write out the XML document (continued)

```
        transformerHandler.endDocument( );
        transformerHandler.setResult(streamResult);
    } catch (Exception e) {
        return null;
    }
    return writer.toString( );
}
```

After calling `startDocument( )` to begin the document, we must create the elements and add data to them. We create an element by calling `startElement( )`:

```
transformerHandler.startElement(null,null,"binary",null)
```

The third element is the only element needed to set up the XML tag, `<binary>`.

The actual `startElement( )` method declaration looks like this:

```
public void startElement(String uri, String localName, String
qName, Attributes atts)
```

The `uri` parameter is used for the namespace, but since this example does not use a namespace, a `null` is passed in.

The second parameter, `localName`, is also used for the namespace and not needed in this example.

The third parameter, `qName`, is the qualified name.

The last parameter, `atts`, is used when the element has attributes; pass in `null` if attributes are not used, as in this case.

To put the data after the element tag, we set a `String`, `data`, to the desired value:

```
data = Integer.toString(key, 2)+"B";
```

Then we convert the data to a `CharArray` and pass it into the `characters( )` method. The second and third parameters show where processing starts and stops in the `CharArray`:

```
transformerHandler.characters(data.toCharArray(),0,data.length( ));
```

Finally, we terminate the element with a call to `endElement( )`:

```
transformerHandler.endElement(null,null,"binary");
```

Each element is created with `startElement( )`, `characters( )`, and `endElement( )`. When all of the elements for the documents have been completed, a call to `endDocument( )` is executed and the result is sent to the `StreamResult` that was set up at the start of the method:

```
transformerHandler.endElement(null,null, "converted-values");
transformerHandler.endDocument( );
transformerHandler.setResult(streamResult);
```

Finally, the `StreamResult` is converted to a `String` by calling `toString()` on the `StringWriter`. The `StreamResult` wraps the `StringWriter` that was set up at the beginning of this method:

```
return writer.toString();
```

The `String` can then be returned to the calling method.

Using SAX is purportedly a faster and less memory-intensive way to create XML documents than using DOM-based libraries such as JDOM and dom4j. If your testing shows that speed is an issue, or if the SAX API is more natural for your application, you should consider using it.

There are other ways to create an XML document. For example, the Apache project's Element Construction Set (ECS) allows you to create an XML document, but there is no method to add data to the document at this time, so for this application ECS is not useful.

Back on the Client: Mining the XML

Here's where the fun begins. The client code in Example 4-5 shows how to mine the data fields from the XML document that the server sends.

Example 4-5. The client code

```
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<head>
    <STYLE type="text/css">
        .borderless {  color:black; text-align:center; background:powderblue;
                border-width:0;border-color:green;  }
    </STYLE>

    <title>function</title>
    <SCRIPT language="JavaScript" type="text/javascript">
        var req;

        function convertToXML( ) {
            var key = document.getElementById("key");
            var keypressed = document.getElementById("keypressed");
            keypressed.value = key.value;
            var url = "/ajaxcodeconverter-lab2/response?key=" + escape(key.value);
            if (window.XMLHttpRequest) {
                req = new XMLHttpRequest( );
            }
            else if (window.ActiveXObject) {
                req = new ActiveXObject("Microsoft.XMLHTTP");
            }
            req.open("Get",url,true);
            req.onreadystatechange = callback;
            req.send(null);
        }
```

Example 4-5. The client code (continued)

```
        function nonMSPopulate( ) {
            xmlDoc = document.implementation.createDocument("","", null);
            var resp = req.responseText;
            var parser = new DOMParser( );
            var dom = parser.parseFromString(resp,"text/xml");

            decVal = dom.getElementsByTagName("decimal");
            var decimal = document.getElementById('decimal');
            decimal.value=decVal[0].childNodes[0].nodeValue;

            hexVal = dom.getElementsByTagName("hexadecimal");
            var hexadecimal = document.getElementById('hexadecimal');
            hexadecimal.value=hexVal[0].childNodes[0].nodeValue;

            octVal = dom.getElementsByTagName("octal");
            var octal = document.getElementById('octal');
            octal.value=octVal[0].childNodes[0].nodeValue;

            hyperVal = dom.getElementsByTagName("hyper");
            var hyper = document.getElementById('hyper');
            hyper.value=hyperVal[0].childNodes[0].nodeValue;

            binaryVal = dom.getElementsByTagName("binary");
            var bin = document.getElementById('bin');
            bin.value=binaryVal[0].childNodes[0].nodeValue;
        }

        function msPopulate( ) {
            var resp = req.responseText;

            var xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
            xmlDoc.async="false";
            xmlDoc.loadXML(resp);

            nodes=xmlDoc.documentElement.childNodes;

            dec = xmlDoc.getElementsByTagName('decimal');
            var decimal = document.getElementById('decimal');
            decimal.value=dec[0].firstChild.data;

            hexi = xmlDoc.getElementsByTagName('hexadecimal');
            var hexadecimal = document.getElementById('hexadecimal');
            hexadecimal.value=hexi[0].firstChild.data;

            oct = xmlDoc.getElementsByTagName('octal');
            var octal = document.getElementById('octal');
            octal.value=oct[0].firstChild.data;

            bin = xmlDoc.getElementsByTagName('binary');
            var binary = document.getElementById('bin');
            binary.value=bin[0].firstChild.data;
```

Example 4-5. The client code (continued)

```
            hypertextml = xmlDoc.getElementsByTagName('hyper');
            var hyper = document.getElementById('hyper');
            hyper.value=hypertextml[0].firstChild.data;
        }

        function callback( ) {
            if (req.readyState==4) {
                if (req.status == 200) {

                    if (window.XMLHttpRequest) {
                        nonMSPopulate( );
                    }
                    else if (window.ActiveXObject) {
                        msPopulate( );
                    }
                }
            }
            clear( );
        }

        function clear( ) {
            var key = document.getElementById("key");
            key.value="";
        }

    </SCRIPT>
</head>

<body>
    <H1>
        <CENTER>AJAX CHARACTER DECODER</CENTER>
    </H1>
    <H2>
        <CENTER>Press a key to find its value.</CENTER>
    </H2>
    <form name="form1" action="/ajaxcodeconverter-lab2" method="get">

    <table border="2" bordercolor="black" bgcolor="lightblue" valign="center"
           align="center">
        <tr>
            <td align="center">
                Enter Key Here ->
                <input type="text" id="key" name="key" maxlength="1" size="1"
                       onkeyup="convertToXML( );">
            </td>
        </tr>
    </table>
    <br>
```

Example 4-5. The client code (continued)

```
    <table class="borderless" border="1" valign="center" align="center">
        <tr>
            <td align="center" colspan="5">
                Key Pressed:&nbsp;
                <input class="borderless" type="text" readonly id="keypressed"
                       maxlength="1" size="1">
            </td>
        </tr>
        <tr>
            <td align="center">&nbsp;&nbsp;Decimal&nbsp;&nbsp;</td>
            <td align="center">Hexadecimal</td>
            <td align="center">&nbsp;&nbsp;&nbsp;Octal&nbsp;&nbsp;&nbsp;</td>
            <td align="center">
                &nbsp;&nbsp;&nbsp;&nbsp;Binary&nbsp;&nbsp;&nbsp;&nbsp;
            </td>
            <td align="center">
                &nbsp;&nbsp;&nbsp;&nbsp;HTML&nbsp;&nbsp;&nbsp;&nbsp;
            </td>
        </tr>
        <tr>
            <td align="center"><input class="borderless" type="text" readonly
                id="decimal" maxlength="6" size="6"></td>
            <td align="center"><input class="borderless" type="text" readonly
                id="hexadecimal" maxlength="6" size="6"></td>
            <td align="center"><input class="borderless" type="text" readonly
                id="octal" maxlength="6" size="6"></td>
            <td align="center"><input class="borderless" type="text" readonly
                id="bin" maxlength="8" size="8"></td>
            <td align="center"><input class="borderless" type="text" readonly
                id="hyper" maxlength="6" size="6"></td>
        </tr>
    </table>

    </form>
</body>
</html>
```

Wow, that's a chunk of code. Let's break it up into two parts: first we'll look at the XML parsing, then at writing the data to the fields.

XML Parsing with JavaScript

Remember, it all starts with our `callback( )` function in JavaScript:

```
    function callback( ) {
        if (req.readyState==4) {
            if (req.status == 200) {
                if (window.XMLHttpRequest) {
                    nonMSPopulate( );
                }
                else if (window.ActiveXObject) {
                    msPopulate( );
                }
            }
            clear( );
        }
    }
```

Remember how `convertToXML( )` had to determine whether the browser was Internet Explorer or something else? We have the same problem again here. When `callback( )` is invoked, it must check to see whether we are running `ActiveXObject` (Internet Explorer) or `XMLHttpRequest` (all other major browsers).

If the browser is Internet Explorer, we run `msPopulate( )` to strip out the data from the XML. Otherwise, we run `nonMSPopulate( )`. What's the difference? It has to do with how we get an XML parser and with the API that parser presents to us. Firefox, Mozilla, and Safari all use `new DOMParser( )` to get a built-in parser that can parse XML, and it's rumored that Opera will support this soon. Internet Explorer, on the other hand, uses `new ActiveXObject("Microsoft.XMLDOM")` to get the Microsoft XML parser.

Although Ajax works on most browsers in their current released states, it's entirely fair to say that the problem of cross-browser compatibility is the Achilles' heel of web applications.

Populating the Form on a Microsoft Browser

The `msPopulate( )` function is reproduced in Example 4-6.

Example 4-6. The msPopulate() function

```
function msPopulate( ) {
    var resp = req.responseText;

    var xmlDoc=new ActiveXObject("Microsoft.XMLDOM");
    xmlDoc.async="false";
    xmlDoc.loadXML(resp);

    dec = xmlDoc.getElementsByTagName('decimal');
    var decimal = document.getElementById('decimal');
    decimal.value=dec[0].firstChild.data;

    hexi = xmlDoc.getElementsByTagName('hexadecimal');
    var hexadecimal = document.getElementById('hexadecimal');
    hexadecimal.value=hexi[0].firstChild.data;

    oct = xmlDoc.getElementsByTagName('octal');
    var octal = document.getElementById('octal');
    octal.value=oct[0].firstChild.data;

```

Example 4-6. The msPopulate() function (continued)

```
20      bin = xmlDoc.getElementsByTagName('binary');
21      var binary = document.getElementById('bin');
22      binary.value=bin[0].firstChild.data;
23
24      hypertextml = xmlDoc.getElementsByTagName('hyper');
25      var hyper = document.getElementById('hyper');
26      hyper.value=hypertextml[0].firstChild.data;
27  }
```

Here, we use the built-in browser functions that I have touted as a programmer power play. We start by getting the `ActiveXObject` called `Microsoft.XMLDOM` (line 4). Next, we load the response from the servlet into the XML document (line 6).

Now we can mine the document for the data. First we get the data between the `<decimal></decimal>` tags. To do this, we first retrieve the XML data field information, by calling `getElementsByTagName(elementName)` (line 8); this function returns the array of child nodes belonging to the element (parent node) associated with the given tag. After we get the array of child nodes, we can reference the first element in the child node by calling `firstChild`. We then obtain the value of the child node by referencing the `data` field. So, to sum it up, we get our decimal value from `dec[0].firstChild.data`.

If you are parsing a document that contains multiple tags with the same tag name, you can access any of the values by indexing the array. For example, if you had another `<decimal>value</decimal>` entry, you would index it with this call: `dec[1].firstChild.data`.

After obtaining the decimal value from the XML, line 10 updates the `decimal` form element. We've now retrieved one value from the XML and displayed it on the page. We continue on in this fashion until all of our data fields are updated with the values retrieved from the XML DOM sent from the servlet.

Populating the Form on Other Browsers

The code for handling non-Microsoft browsers, shown in Example 4-7, is similar; there are some minor differences in the parser API, but that's about it.

Example 4-7. The nonMSPopulate() function

```
1  function nonMSPopulate( ) {
2      var resp = req.responseText;
3      var parser = new DOMParser( );
4      var dom = parser.parseFromString(resp,"text/xml");
5
6      decVal = dom.getElementsByTagName("decimal");
7      var decimal = document.getElementById('decimal');
8      decimal.value=decVal[0].childNodes[0].nodeValue;
9
```

Example 4-7. The nonMSPopulate() function (continued)

```
10      hexVal = dom.getElementsByTagName("hexadecimal");
11      var hexadecimal = document.getElementById('hexadecimal');
12      hexadecimal.value=hexVal[0].childNodes[0].nodeValue;
13
14      octVal = dom.getElementsByTagName("octal");
15      var octal = document.getElementById('octal');
16      octal.value=octVal[0].childNodes[0].nodeValue;
17
18      hyperVal = dom.getElementsByTagName("hyper");
19      var hyper = document.getElementById('hyper');
20      hyper.value=hyperVal[0].childNodes[0].nodeValue;
21
22      binaryVal = dom.getElementsByTagName("binary");
23      var bin = document.getElementById('bin');
24      bin.value=binaryVal[0].childNodes[0].nodeValue;
25  }
```

We create a new `DOMParser` (line 3), then we create a DOM on line 4 from the XML string that we received from the servlet. Next, we get the element between the `<decimal></decimal>` tags (line 6) by calling `dom.getElementByTagName("decimal")`. After retrieving the `decimal` form element from our HTML document, we retrieve the value sent to us in the XML and use it to set the appropriate field:

```
decimal.value=decVal[0].childNodes[0].nodeValue;
```

The reference to `decVal[0]` gets the data between the `<decimal></decimal>` tags. If we had two sets of `<decimal></decimal>` tags, we would reference the second set as `decVal[1]`.

This is what the data should look like in the response sent from the servlet:

```
<converted-values>
    <decimal>97</decimal>
    <hexadecimal>0x61</hexadecimal>
    <octal>0141</octal>
    <hyper>&amp;0x61;</hyper>
    <binary>1100001B</binary>
</converted-values>
```

Building the Application

Now that we have reviewed the code, let's build it and try it out. It's very similar to the example in the previous chapter, and because I don't like to overwrite my work, I put the new application in its own directory tree. The directory structure I used is shown in Figure 4-2.

This is just a guide so you can see how I built the sample. You can name your directories differently as long as you know how to configure the application server properly.

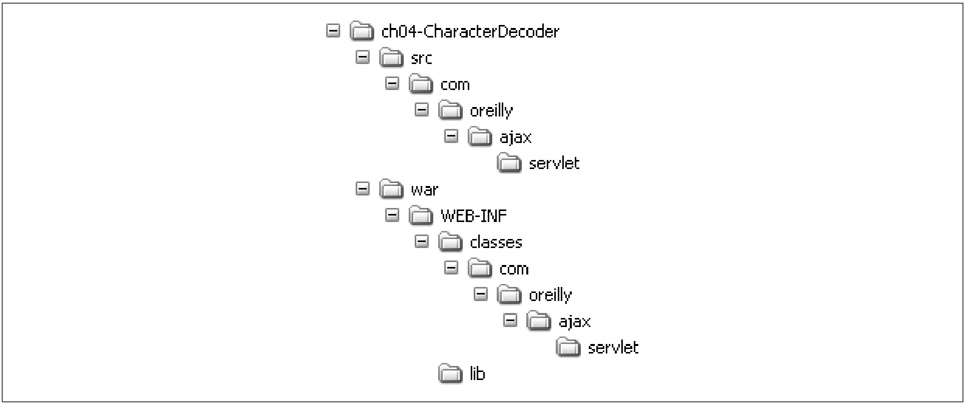


Figure 4-2. Directory structure for Ajax Character Decoder (second version)

The *web.xml* file for the project is presented in Example 4-8.

Example 4-8. web.xml

```
<!DOCTYPE web-app
    PUBLIC  "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
    <servlet>
        <servlet-name>AjaxResponseServlet</servlet-name>
        <servlet-class>
            com.AJAXbook.servlet.AjaxResponseServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>AjaxResponseServlet</servlet-name>
        <url-pattern>/response</url-pattern>
    </servlet-mapping>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

The *build.xml* file for the project is shown in Example 4-9.

Example 4-9. build.xml

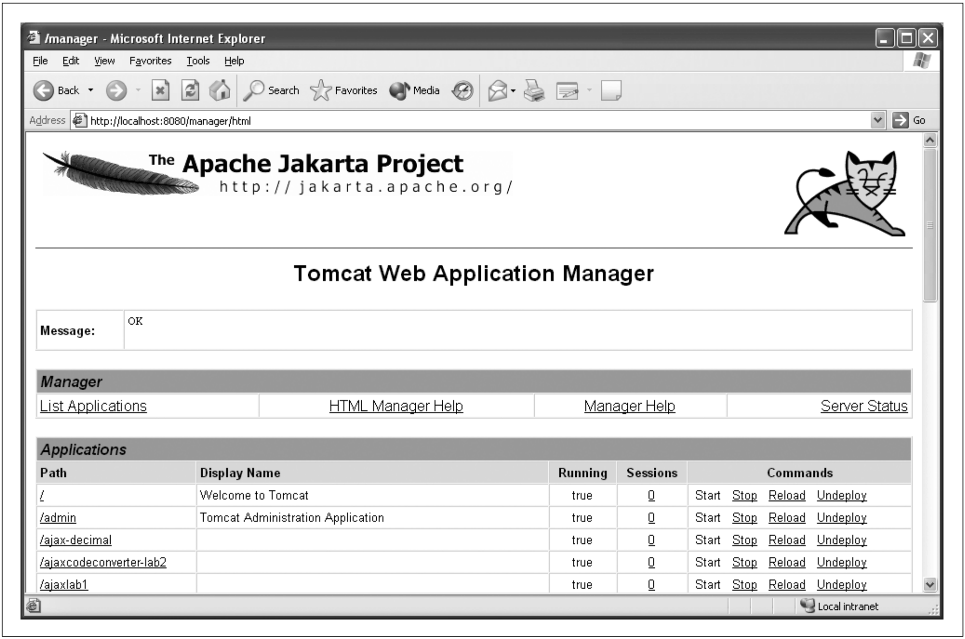
```
<?xml version="1.0"?>
<project name="AJAX-CODECONVERTER " default="compile" basedir=".">

    <property environment="env"/>
    <property name="src.dir" value="src"/>
    <property name="test.dir" value="test"/>
    <property name="war.dir" value="war"/>
```

Example 4-9. build.xml (continued)

```
    <property name="db.dir" value="db"/>
    <property name="class.dir" value="${war.dir}/WEB-INF/classes"/>
    <property name="test.class.dir" value="${test.dir}/classes"/>
    <property name="lib.dir" value="${war.dir}/WEB-INF/lib"/>
    <property name="webapp.dir"
              value="${env.TOMCAT_HOME}/webapps/ajaxcodeconverter-lab2"/>

    <path id="ajax.class.path">
        <fileset dir="${lib.dir}">
            <include name="*.jar"/>
        </fileset>
    </path>

    <target name="testenv">
        <echo message="env.TomcatHome=${env.TOMCAT_HOME}"/>
        <echo message="env.ANT_HOME=${env.ANT_HOME}"/>
    </target>

    <target name="init">
        <mkdir dir="${class.dir}"/>
        <mkdir dir="${test.class.dir}"/>
    </target>

    <target name="compile" depends="init"
            description="Compiles all source code.">
        <javac srcdir="${src.dir}" destdir="${class.dir}" debug="on"
               classpathref="ajax.class.path"/>
    </target>

    <target name="clean" description="Erases contents of classes dir">
        <delete dir="${class.dir}"/>
        <delete dir="${test.class.dir}"/>
    </target>

    <target name="deploy" depends="compile"
            description="Copies the contents of web-app to destination dir">
        <copy todir="${webapp.dir}">
            <fileset dir="${war.dir}"/>
        </copy>
    </target>

</project>
```

Running the Application on Tomcat

If you are running your application on the Tomcat server, you can use the Tomcat Web Application Manager (accessible through the URL *http://localhost:8080/manager/html*) to check whether it was deployed. The application manager is shown in Figure 4-3. As you can see in this figure, two versions of our Ajax converter have been deployed successfully and are currently running. The directory in which you installed the application becomes part of the path.


/manager - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media
Address http://localhost:8080/manager/html
Go
The Apache Jakarta Project
http://jakarta.apache.org/
Local intranet

Tomcat Web Application Manager

Message:	OK

Manager			
List Applications	HTML Manager Help	Manager Help	Server Status

Applications

Path	Display Name	Running	Sessions	Commands
/	Welcome to Tomcat	true	0	Start Stop Reload Undeploy
/admin	Tomcat Administration Application	true	0	Start Stop Reload Undeploy
/ajax-decimal		true	0	Start Stop Reload Undeploy
/ajaxcodeconverter-lab2		true	0	Start Stop Reload Undeploy
/ajaxlab1		true	0	Start Stop Reload Undeploy

Figure 4-3. Tomcat Web Application Manager

Click the link under Applications to open a browser window that accesses the directory of your application. Then click the “index.html” link to see the application.

Passing Data with JSON

Now that you’ve seen how to use XML as the data vehicle, we must talk about some of the problems with XML. One major drawback is speed. XML requires two tags per data point, plus extra tags for parent nodes and so on. All this extra data in transmission slows down the data exchange between the client and server. You can easily end up with a long document that contains only a few bytes’ worth of data. Constructing the document can also be a rather elaborate process that requires a lot of memory on the server.

Fortunately, there is another way to send data to the client that is easier to parse and more compact. That alternative is JSON (pronounced Jason). JSON objects are typically smaller than the equivalent XML documents, and working with them is more memory-efficient.

The other great benefit of JSON is that you can parse it with JavaScript’s `eval()` function. You don’t need other libraries, and you don’t need to worry as much about cross-browser functionality. As long as your browser has JavaScript enabled and supports the `eval()` function, you will be able to interpret the data.

You may still need to use XML (and now you know how), but if you have the option, there are compelling reasons to use JSON. In most cases, you will be better off with a JSON implementation.

This is our data object represented in JSON:

```
{"conversion":{
"decimal": "120",
"hexadecimal": "78",
"octal": "170",
"hyper": "&amp;0x78",
"binary": "1111000B"}
}
```

There are programmatic ways to build the JSON object, but to keep it simple we'll use a `StringBuffer` again and glue together the strings that will form the conversion object. Example 4-10 illustrates how we build the data object in the servlet.

Example 4-10. AjaxJSONServlet.java

```
package com.oreilly.ajax.servlet;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class AjaxResponseServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    public void doGet(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
        // key is the parameter passed in from the JavaScript
        // variable named url (see index.html)
        String key = req.getParameter("key");
        if (key != null) {
            // extract the first character from key
            // as an int, then convert that int to a String
            int keyInt = key.charAt(0);
            // set up the response
            res.setContentType("text/xml");
            res.setHeader("Cache-Control", "no-cache");
            // write out the XML string
            String outString = createStringBufferJSON(keyInt);
            res.getWriter( ).write(outString);
        }
        else {
            // if key comes back as a null, return a question mark
            res.setContentType("text/xml");
            res.setHeader("Cache-Control", "no-cache");
            res.getWriter( ).write("?");
        }
    }
```

Example 4-10. AjaxJSONServlet.java (continued)

```
    public String createStringBufferJSON(int keyInt) {
        StringBuffer returnJSON = new StringBuffer("\r\n{\"conversion\":{");
        returnJSON.append("\r\n\"decimal\": \""+
                          Integer.toString(keyInt)+"\",");
        returnJSON.append("\r\n\"hexadecimal\": \""+
                          Integer.toString(keyInt,16)+"\",");
        returnJSON.append("\r\n\"octal\": \""+
                          Integer.toString(keyInt,8)+"\",");
        returnJSON.append("\r\n\"hyper\": \"&0x"+
                          Integer.toString(keyInt,16)+"\",");
        returnJSON.append("\r\n\"binary\": \""+
                          Integer.toString(keyInt,2)+"B\"");
        returnJSON.append("\r\n}}");
        return returnJSON.toString( );
    }
}
```

That wasn't all that different from creating an XML document with a `StringBuffer`.

An alternative approach is to use the JSON library. Download *json_simple.zip* from *http://www.JSON.org/java/json_simple.zip*, and unzip it. Copy the *json_simple.jar* file from the *lib* directory into your *WEB-INF/lib* directory, and then add to it the import from `json_simple`:

```
import org.json.simple.JSONObject;
```

Now the code from Example 4-10 can be written as shown in Example 4-11.

Example 4-11. Writing JSON support with the json_simple library

```
public String createJSONwithJSONsimple(int keyInt) {
    JSONObject obj = new JSONObject( );
    JSONObject obj2 = new JSONObject( );

    obj2.put("decimal",Integer.toString(keyInt));
    obj2.put("hexadecimal",Integer.toString(keyInt,16));
    obj2.put("octal",Integer.toString(keyInt,8));
    obj2.put("hyper","&0x"+Integer.toString(keyInt,16));
    obj2.put("binary",Integer.toString(keyInt,2)+"B");

    obj.put("conversion",obj2);
    return(obj.toString( ));
}
```

The first `JSONObject`, `obj`, encapsulates the second object to product the result. The JSON object built with *json_simple.jar* looks like this:

```
{"conversion":{"decimal":"103","hyper":"&0x67","octal":"147","hexadecimal":"67",
"binary":"1100111B"}}
```

You can nest objects with the `json_simple` library. As a matter of fact, you can nest objects in a JSON array. You'll learn more about JSON arrays and JSON objects in Chapter 5.

Another JSON library, called `jsontools`, is located at *http://developer.berlios.de/projects/jsontools/*. The manual for the `jsontools` project does a really great job of explaining JSON and how to use the `jsontools` library, so you may want to download it if you're looking for a good primer on JSON.

Also, don't forget to look at the documentation at *http://www.JSON.org*.

Changing the JavaScript for JSON

Now let's look at the client JavaScript code. I've replaced two functions with one: I removed `msPopulate( )` and `nonMSPopulate( )`, and now all browsers use the `populateJSON( )` function shown in Example 4-12.

Example 4-12. The populateJSON() function

```
function populateJSON( ) {
    var jsonData = req.responseText;

    var myJSONObject = eval('(' + jsonData + ')');

    var decimal = document.getElementById('decimal');
    decimal.value=myJSONObject.conversion.decimal;

    var hexadecimal = document.getElementById('hexadecimal');
    hexadecimal.value=myJSONObject.conversion.hexadecimal;

    var octal = document.getElementById('octal');
    octal.value=myJSONObject.conversion.octal;

    var binary = document.getElementById('bin');
    binary.value=myJSONObject.conversion.binary;

    var hyper = document.getElementById('hyper');
    hyper.value=myJSONObject.conversion.hyper;
}
```

Summary

Remember that Ajax isn't a technology: it's a group of ideas that, used together, have proven very powerful. When you combine form manipulation with JavaScript, asynchronous callbacks with `XMLHTTPRequest`, and built-in XML with JSON parsers, you have something revolutionary—even though the individual pieces have been around for a while.

Combine these client-side technologies with Java's established server-side technologies, such as servlets, Struts, and JSF, and you've got a really powerful basis for building a new generation of interactive web applications.