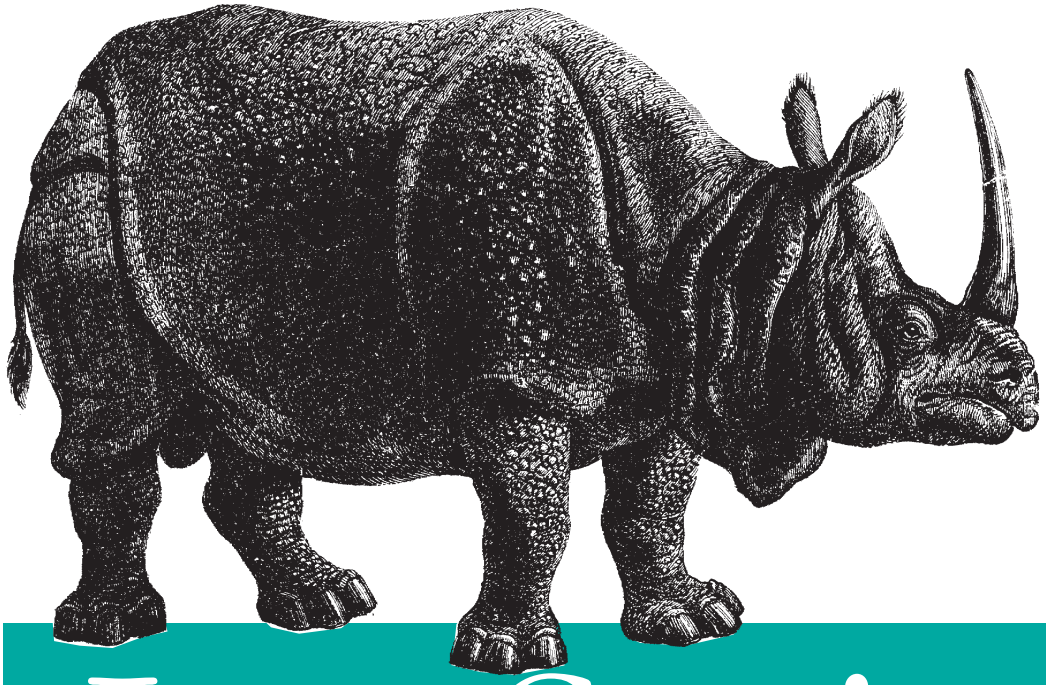
Activate Your Web Pages

5th Edition
Includes Ajax and DOM Scripting



JavaScript

The Definitive Guide

O'REILLY®

David Flanagan

JavaScript: The Definitive Guide, Fifth Edition
by David Flanagan

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: Debra Cameron
Production Editor: Sanders Kleinfeld
Copyeditor: Mary Anne Weeks Mayo
Proofreader: Sanders Kleinfeld
Indexer: Ellen Troutman-Zaig
Cover Designer: Edie Freedman
Interior Designer: David Futato
Illustrators: Robert Romano and Jessamyn Read

Printing History:

August 1996:	Beta Edition.
January 1997:	Second Edition.
June 1998:	Third Edition.
January 2002:	Fourth Edition.
August 2006:	Fifth Edition.

ISBN-10: 0-596-10199-6
ISBN-13: 978-0-596-10199-2
[M]

CHAPTER 21

JavaScript and XML

The most important feature of the Ajax web application architecture is its ability to script HTTP with the XMLHttpRequest object, which was covered in Chapter 20. The X in "Ajax" stands for XML, however, and for many web applications, Ajax's use of XML-formatted data is its second most important feature.

This chapter explains how to use JavaScript to work with XML data. It starts by demonstrating techniques for obtaining XML data: loading it from the network, parsing it from a string, and obtaining it from XML *data islands* within an HTML document. After this discussion of obtaining XML data, the chapter explains basic techniques for working with this data. It covers use of the W3C DOM API, transforming XML data with XSL stylesheets, querying XML data with XPath expressions, and serializing XML data back to string form.

This coverage of basic XML techniques is followed by two sections that demonstrate applications of those techniques. First, you'll see how it is possible to define HTML templates and automatically expand them, using the DOM and XPath, with data from an XML document. Second, you'll see how to write a web services client in JavaScript using the XML techniques from this chapter.

Finally, the chapter concludes with a brief introduction to E4X, which is a powerful extension to the core JavaScript language for working with XML.

21.1 Obtaining XML Documents

Chapter 20 showed how to use the XMLHttpRequest object to obtain an XML document from a web server. When the request is complete, the `responseXML` property of the XMLHttpRequest object refers to a Document object that is the parsed representation of the XML document. This is not the only way to obtain an XML Document object, however. The subsections that follow show how you can create an empty XML document, load an XML document from a URL without using XMLHttpRequest, parse an XML document from a string, and obtain an XML document from an XML data island.

As with many advanced client-side JavaScript features, the techniques for obtaining XML data are usually browser-specific. The following subsections define utility functions that work in both Internet Explorer (IE) and Firefox.

21.1.1 Creating a New Document

You can create an empty (except for an optional root element) XML Document in Firefox and related browsers with the DOM Level 2 method `document.implementation.createDocument()`. You can accomplish a similar thing in IE with the ActiveX object named MSXML2.DOMDocument. Example 21-1 defines an `XML.newDocument()` utility function that hides the differences between these two approaches. An empty XML document isn't useful by itself, but creating one is the first step of the document loading and parsing techniques that are shown in the examples that follow this one.

Example 21-1. Creating an empty XML document

```
/**
 * Create a new Document object. If no arguments are specified,
 * the document will be empty. If a root tag is specified, the document
 * will contain that single root tag. If the root tag has a namespace
 * prefix, the second argument must specify the URL that identifies the
 * namespace.
 */
XML.newDocument = function(rootTagName, namespaceURL) {
    if (!rootTagName) rootTagName = "";
    if (!namespaceURL) namespaceURL = "";

    if (document.implementation && document.implementation.createDocument) {
        // This is the W3C standard way to do it
        return document.implementation.createDocument(namespaceURL,
                                                       rootTagName, null);
    }
    else { // This is the IE way to do it
        // Create an empty document as an ActiveX object
        // If there is no root element, this is all we have to do
        var doc = new ActiveXObject("MSXML2.DOMDocument");

        // If there is a root tag, initialize the document
        if (rootTagName) {
            // Look for a namespace prefix
            var prefix = "";
            var tagname = rootTagName;
            var p = rootTagName.indexOf(':');
            if (p != -1) {
                prefix = rootTagName.substring(0, p);
                tagname = rootTagName.substring(p+1);
            }

            // If we have a namespace, we must have a namespace prefix
            // If we don't have a namespace, we discard any prefix
            if (namespaceURL) {
```

Example 21-1. Creating an empty XML document (continued)

```
                if (!prefix) prefix = "a0"; // What Firefox uses
            }
            else prefix = "";

            // Create the root element (with optional namespace) as a
            // string of text
            var text = "<" + (prefix?(prefix+":"):"") +  tagname +
                (namespaceURL
                 ?(" xmlns:" + prefix + '="' + namespaceURL +'"')
                 :"") +
                "/>";
            // And parse that text into the empty document
            doc.loadXML(text);
        }
        return doc;
    }
};
```

21.1.2 Loading a Document from the Network

Chapter 20 showed how to use the XMLHttpRequest object to dynamically issue HTTP requests for text-based documents. When used with XML documents, the `responseXML` property refers to the parsed representation as a DOM Document object. XMLHttpRequest is nonstandard but widely available and well understood, and is usually the best technique for loading XML documents.

There *is* another way, however. An XML Document object created using the techniques shown in Example 21-1 can load and parse an XML document using a less well-known technique. Example 21-2 shows how it is done. Amazingly, the code is the same in both Mozilla-based browsers and in IE.

Example 21-2. Loading an XML document synchronously

```
/**
 * Synchronously load the XML document at the specified URL and
 * return it as a Document object
 */
XML.load = function(url) {
    // Create a new document with the previously defined function
    var xmldoc = XML.newDocument( );
    xmldoc.async = false;  // We want to load synchronously
    xmldoc.load(url);      // Load and parse
    return xmldoc;         // Return the document
};
```

Like XMLHttpRequest, this `load( )` method is nonstandard. It differs from XMLHttpRequest in several important ways. First, it works only with XML documents; XMLHttpRequest can be used to download any kind of text document. Second, it is not restricted to the HTTP protocol. In particular, it can be used to read files from

the local filesystem, which is helpful during the testing and development phase of a web application. Third, when used with HTTP, it generates only GET requests and cannot be used to POST data to a web server.

Like XMLHttpRequest, the `load( )` method can be used asynchronously. In fact, this is the default method of operation unless `async` property is set to `false`. Example 21-3 shows an asynchronous version of the `XML.load( )` method.

Example 21-3. Loading an XML document asynchronously

```
/**
 * Asynchronously load and parse an XML document from the specified URL.
 * When the document is ready, pass it to the specified callback function.
 * This function returns immediately with no return value.
 */
XML.loadAsync = function(url, callback) {
    var xmldoc = XML.newDocument( );

    // If we created the XML document using createDocument, use
    // onload to determine when it is loaded
    if (document.implementation && document.implementation.createDocument) {
        xmldoc.onload = function( ) { callback(xmldoc); };
    }
    // Otherwise, use onreadystatechange as with XMLHttpRequest
    else {
        xmldoc.onreadystatechange = function( ) {
            if (xmldoc.readyState == 4) callback(xmldoc);
        };
    }

    // Now go start the download and parsing
    xmldoc.load(url);
};
```

21.1.3 Parsing XML Text

Sometimes, instead of parsing an XML document loaded from the network, you simply want to parse an XML document from a JavaScript string. In Mozilla-based browsers, a DOMParser object is used; in IE, the `loadXML( )` method of the Document object is used. (If you paid attention to the `XML.newDocument( )` code in Example 21-1, you've already seen this method used once.)

Example 21-4 shows a cross-platform XML parsing function that works in Mozilla and IE. For platforms other than these two, it attempts to parse the text by loading it with an XMLHttpRequest from a `data:` URL.

Example 21-4. Parsing an XML document

```
/**
 * Parse the XML document contained in the string argument and return
 * a Document object that represents it.
```

Example 21-4. Parsing an XML document (continued)

```
 */
XML.parse = function(text) {
    if (typeof DOMParser != "undefined") {
        // Mozilla, Firefox, and related browsers
        return (new DOMParser()).parseFromString(text, "application/xml");
    }
    else if (typeof ActiveXObject != "undefined") {
        // Internet Explorer.
        var doc = XML.newDocument();  // Create an empty document
        doc.loadXML(text);            // Parse text into it
        return doc;                   // Return it
    }
    else {
        // As a last resort, try loading the document from a data: URL
        // This is supposed to work in Safari. Thanks to Manos Batsis and
        // his Sarissa library (sarissa.sourceforge.net) for this technique.
        var url = "data:text/xml;charset=utf-8," + encodeURIComponent(text);
        var request = new XMLHttpRequest();
        request.open("GET", url, false);
        request.send(null);
        return request.responseXML;
    }
};
```

21.1.4 XML Documents from Data Islands

Microsoft has extended HTML with an `<xml>` tag that creates an XML data island within the surrounding "sea" of HTML markup. When IE encounters this `<xml>` tag, it treats its contents as a separate XML document, which you can retrieve using `document.getElementById()` or other HTML DOM methods. If the `<xml>` tag has a `src` attribute, the XML document is loaded from the URL specified by that attribute instead of being parsed from the content of the `<xml>` tag.

If a web application requires XML data, and the data is known when the application is first loaded, there is an advantage to including that data directly within the HTML page: the data is already available, and the web application does not have to establish another network connection to download the data. XML data islands can be a useful way to accomplish this. It is possible to approximate IE data islands in other browsers using code like that shown in Example 21-5.

Example 21-5. Getting an XML document from a data island

```
/**
 * Return a Document object that holds the contents of the <xml> tag
 * with the specified id. If the <xml> tag has a src attribute, an XML
 * document is loaded from that URL and returned instead.
 *
 * Since data islands are often looked up more than once, this function caches
 * the documents it returns.
 */
```

Example 21-5. Getting an XML document from a data island (continued)

```
XML.getDataIsland = function(id) {
    var doc;

    // Check the cache first
    doc = XML.getDataIsland.cache[id];
    if (doc) return doc;

    // Look up the specified element
    doc = document.getElementById(id);

    // If there is a "src" attribute, fetch the Document from that URL
    var url = doc.getAttribute('src');
    if (url) {
        doc = XML.load(url);
    }
    // Otherwise, if there was no src attribute, the content of the <xml>
    // tag is the document we want to return. In Internet Explorer, doc is
    // already the document object we want. In other browsers, doc refers to
    // an HTML element, and we've got to copy the content of that element
    // into a new document object
    else if (!doc.documentElement) {// If this is not already a document...

        // First, find the document element within the <xml> tag. This is
        // the first child of the <xml> tag that is an element, rather
        // than text, comment, or processing instruction
        var docelt = doc.firstChild;
        while(docelt != null) {
            if (docelt.nodeType == 1 /*Node.ELEMENT_NODE*/) break;
            docelt = docelt.nextSibling;
        }

        // Create an empty document
        doc = XML.newDocument( );

        // If the <xml> node had some content, import it into the new document
        if (docelt) doc.appendChild(doc.importNode(docelt, true));
    }

    // Now cache and return the document
    XML.getDataIsland.cache[id] = doc;
    return doc;
};
XML.getDataIsland.cache = {}; // Initialize the cache
```

This code does not perfectly simulate XML data islands in non-IE browsers. The HTML standard requires browsers to parse (but ignore) tags such as <xml> that they don't know about. This means that browsers don't discard XML data within an <xml> tag. It also means that any text within the data island is displayed by default. An easy way to prevent this is with the following CSS stylesheet:

```
<style type="text/css">xml { display: none; }</style>
```

Another incompatibility is that non-IE browsers treat the content of XML data islands as HTML rather than XML content. If you use the code in Example 21-5 in Firefox, for example, and then serialize the resulting document (you'll see how to do this later in the chapter), you'll find that the tag names are all converted to uppercase because Firefox thinks they are HTML tags. In some cases, this may be problematic; in many other cases, it is not. Finally, notice that XML namespaces break if the browser treats the XML tags as HTML tags. This means that inline XML data islands are not suitable for things like XSL stylesheets (XSL is covered in more detail later in this chapter) because those stylesheets always use namespaces.

If you want the network benefits of including XML data directly in an HTML page, but don't want the browser incompatibilities that come with using XML data islands and the `<xml>` tag, consider encoding your XML document text as a JavaScript string and then parsing the document using code like that shown in Example 21-4.

21.2 Manipulating XML with the DOM API

The previous section showed a number of ways to obtain parsed XML data in the form of a Document object. The Document object is defined by the W3C DOM API and is much like the HTMLDocument object that is referred to by the `document` property of the web browser.

The following subsections explain some important differences between the HTML DOM and the XML DOM and then demonstrate how you can use the DOM API to extract data from an XML document and display that data to a user by dynamically creating nodes in the browser's HTML document.

21.2.1 XML Versus HTML DOM

Chapter 15 explained the W3C DOM but focused on its application in client-side JavaScript to HTML documents. In fact, the W3C designed the DOM API to be language-neutral and focused primarily on XML documents; its use with HTML documents is through an optional extension module. In Part IV, notice that there are separate entries for Document and HTMLDocument, and for Element and HTMLElement. HTMLDocument and HTMLElement are extensions of the core XML Document and Element objects. If you are used to manipulating HTML documents with the DOM, you must be careful not to use HTML-specific API features when working with XML documents.

Probably the most important difference between the HTML and XML DOMs is that the `getElementById()` method is not typically useful with XML documents. In DOM Level 1, the method is actually HTML-specific, defined only by the HTMLDocument interface. In DOM Level 2, the method is moved up a level to the Document interface, but there is a catch. In XML documents, `getElementById()` searches for elements with the specified value of an attribute whose *type* is "id". It is not

sufficient to define an attribute *named* "id" on an element: the name of the attribute does not matter—only the type of the attribute. Attribute types are declared in the DTD of a document, and a document's DTD is specified in the DOCTYPE declaration. XML documents used by web applications often have no DOCTYPE declaration specifying a DTD, and a call to `getElementById()` on such a document always returns `null`. Note that the `getElementsByTagName()` method of the Document and Element interfaces works fine for XML documents. (Later in the chapter, I'll show you how to query an XML document using powerful XPath expressions; XPath can be used to retrieve elements based on the value of any attribute.)

Another difference between HTML and XML Document objects is that HTML documents have a `body` property that refers to the `<body>` tag within the document. For XML documents, only the `documentElement` property refers to the top-level element of the document. Note that this top-level element is also available through the `childNodes[]` property of the document, but it may not be the first or only element of that array because an XML document may also contain a DOCTYPE declaration, comments, and processing instructions at the top level.

There is also an important difference between the XML Element interface and the HTMLElement interface that extends it. In the HTML DOM, standard HTML attributes of an element are made available as properties of the HTMLElement interface. The `src` attribute of an `<img>` tag, for example, is available through the `src` property of the HTMLImageElement object that represents the `<img>` tag. This is not the case in the XML DOM: the Element interface has only a single `tagName` property. The attributes of an XML element must be explicitly queried and set with `getAttribute()`, `setAttribute()`, and related methods.

As a corollary, note that special attributes that are meaningful on any HTML element are meaningless on all XML elements. Recall that setting an attribute named "id" on an XML element does not mean that that element can be found with `getElementById()`. Similarly, you cannot style an XML element by setting its style attribute. Nor can you associate a CSS class with an XML element by setting its class attribute. All these attributes are HTML-specific.

21.2.2 Example: Creating an HTML Table from XML Data

Example 21-7 defines a function named `makeTable()` that uses both the XML and HTML DOMs to extract data from an XML document and insert that data into an HTML document in the form of a table. The function expects a JavaScript object literal argument that specifies which elements of the XML document contain table data and how that data should be arranged in the table.

Before looking at the code for `makeTable()`, let's first consider a usage example. Example 21-6 shows a sample XML document that's used here (and elsewhere throughout this chapter).

Example 21-6. An XML datafile

```
<?xml version="1.0"?>
<contacts>
  <contact name="Able Baker"><email>able@example.com</email></contact>
  <contact name="Careful Dodger"><email>dodger@example.com</email></contact>
  <contact name="Eager Framer" personal="true"><email>framer@example.com</email></contact>
</contacts>
```

The following HTML fragment shows how the `makeTable( )` function might be used with that XML data. Note that the schema object refers to tag and attribute names from this sample datafile:

```
    <script>
    // This function uses makeTable( )
    function displayAddressBook( ) {
        var schema = {
            rowtag: "contact",
            columns: [
                { tagname: "@name", label: "Name" },
                { tagname: "email", label: "Address" }
            ]
        };

        var xmldoc = XML.load("addresses.xml");  // Read the XML data
        makeTable(xmldoc, schema, "addresses");  // Convert to an HTML table
    }
    </script>

    <button onclick="displayAddressBook( )">Display Address Book</button>
    <div id="addresses"><!--table will be inserted here --></div>
```

The implementation of `makeTable( )` is shown in Example 21-7.

Example 21-7. Building an HTML table from XML data

```
/**
 * Extract data from the specified XML document and format it as an HTML table.
 * Append the table to the specified HTML element. (If element is a string,
 * it is taken as an element ID, and the named element is looked up.)
 *
 * The schema argument is a JavaScript object that specifies what data is to
 * be extracted and how it is to be displayed. The schema object must have a
 * property named "rowtag" that specifies the tag name of the XML elements that
 * contain the data for one row of the table. The schema object must also have
 * a property named "columns" that refers to an array. The elements of this
 * array specify the order and content of the columns of the table. Each
 * array element may be a string or a JavaScript object. If an element is a
 * string, that string is used as the tag name of the XML element that contains
 * table data for the column, and also as the column header for the column.
 * If an element of the columns[] array is an object, it must have one property
 * named "tagname" and one named "label". The tagname property is used to
 * extract data from the XML document and the label property is used as the
 * column header text. If the tagname begins with an @ character, it is
 * an attribute of the row element rather than a child of the row.
```

Example 21-7. Building an HTML table from XML data (continued)

```
 */
function makeTable(xmldoc, schema, element) {
    // Create the <table> element
    var table = document.createElement("table");

    // Create the header row of <th> elements in a <tr> in a <thead>
    var thead = document.createElement("thead");
    var header = document.createElement("tr");
    for(var i = 0; i < schema.columns.length; i++) {
        var c = schema.columns[i];
        var label = (typeof c == "string")?c:c.label;
        var cell = document.createElement("th");
        cell.appendChild(document.createTextNode(label));
        header.appendChild(cell);
    }
    // Put the header into the table
    thead.appendChild(header);
    table.appendChild(thead);

    // The remaining rows of the table go in a <tbody>
    var tbody = document.createElement("tbody");
    table.appendChild(tbody);

    // Now get the elements that contain our data from the xml document
    var xmlrows = xmldoc.getElementsByTagName(schema.rowtag);

    // Loop through these elements. Each one contains a row of the table.
    for(var r=0; r < xmlrows.length; r++) {
        // This is the XML element that holds the data for the row
        var xmlrow = xmlrows[r];
        // Create an HTML element to display the data in the row
        var row = document.createElement("tr");

        // Loop through the columns specified by the schema object
        for(var c = 0; c < schema.columns.length; c++) {
            var sc = schema.columns[c];
            var tagname = (typeof sc == "string")?sc:sc.tagname;
            var celltext;
            if (tagname.charAt(0) == '@') {
                // If the tagname begins with '@', it is an attribute name
                celltext = xmlrow.getAttribute(tagname.substring(1));
            }
            else {
                // Find the XML element that holds the data for this column
                var xmlcell = xmlrow.getElementsByTagName(tagname)[0];
                // Assume that element has a text node as its first child
                var celltext = xmlcell.firstChild.data;
            }
            // Create the HTML element for this cell
            var cell = document.createElement("td");
            // Put the text data into the HTML cell
            cell.appendChild(document.createTextNode(celltext));
```

Example 21-7. Building an HTML table from XML data (continued)

```
            // Add the cell to the row
            row.appendChild(cell);
        }

        // And add the row to the tbody of the table
        tbody.appendChild(row);
    }

    // Set an HTML attribute on the table element by setting a property.
    // Note that in XML we must use setAttribute( ) instead.
    table.frame = "border";

    // Now that we've created the HTML table, add it to the specified element.
    // If that element is a string, assume it is an element ID.
    if (typeof element == "string") element = document.getElementById(element);
    element.appendChild(table);
}
```

21.3 Transforming XML with XSLT

Once you've loaded, parsed, or otherwise obtained a Document object representing an XML document, one of the most powerful things you can do with it is transform it using an XSLT stylesheet. XSLT stands for XSL Transformations, and XSL stands for Extensible Stylesheet Language. XSL stylesheets are XML documents and can be loaded and parsed in the same way that any XML document can. A tutorial on XSL is well beyond the scope of this book, but Example 21-8 shows a sample stylesheet that can transform an XML document like the one shown in Example 21-6 into an HTML table.

Example 21-8. A simple XSL stylesheet

```
<?xml version="1.0"?><!-- this is an xml document -->
<!-- declare the xsl namespace to distinguish xsl tags from html tags -->
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html"/>

  <!-- When we see the root element, output the HTML framework of a table -->
  <xsl:template match="/">
    <table>
      <tr><th>Name</th><th>E-mail Address</th></tr>
      <xsl:apply-templates/>  <!-- and recurse for other templates -->
    </table>
  </xsl:template>

  <!-- When we see a <contact> element... -->
  <xsl:template match="contact">
    <tr> <!-- Begin a new row of the table -->
      <!-- Use the name attribute of the contact as the first column -->
      <td><xsl:value-of select="@name"/></td>
```

Example 21-8. A simple XSL stylesheet (continued)

```
      <xsl:apply-templates/>  <!-- and recurse for other templates -->
    </tr>
  </xsl:template>

  <!-- When we see an <email> element, output its content in another cell -->
  <xsl:template match="email">
    <td><xsl:value-of select="."/></td>
  </xsl:template>
</xsl:stylesheet>
```


XSLT transforms the content of an XML document using the rules in an XSL stylesheet. In the context of client-side JavaScript, this is usually done to transform the XML document into HTML. Many web application architectures use XSLT on the server side, but Mozilla-based browsers and IE support XSLT on the client side, and pushing the transform off the server and onto the client can save server resources and network bandwidth (because XML data is usually more compact than the HTML presentation of that data).

Many modern browsers can style XML using either CSS or XSL stylesheets. If you specify a stylesheet in an `xml-stylesheet` processing instruction, you can load an XML document directly into the browser, and the browser styles and displays it. The requisite processing instruction might look like this:

```
<?xml-stylesheet href="dataToTable.xml" type="text/xsl"?>
```

Note that the browser performs this kind of XSLT transformation automatically when an XML document containing an appropriate processing instruction is loaded into the browser display window. This is important and useful, but it is not the subject matter of this section. What I explain here is how to use JavaScript to dynamically perform XSL transformations.

The W3C has not defined a standard API for performing XSL transformations on DOM Document and Element objects. In Mozilla-based browsers, the XSLTProcessor object provides a JavaScript XSLT API. And in IE, XML Document and Element objects have a `transformNode()` method for performing transformations. Example 21-9 shows both APIs. It defines an XML.Transformer class that encapsulates an XSL stylesheet and allows it to be used to transform more than one XML document. The `transform()` method of an XML.Transformer object uses the encapsulated stylesheet to transform a specified XML document and then replaces the content of a specified DOM element with the results of the transformation.

Example 21-9. XSLT in Mozilla and Internet Explorer

```
/**
 * This XML.Transformer class encapsulates an XSL stylesheet.
 * If the stylesheet parameter is a URL, we load it.
 * Otherwise, we assume it is an appropriate DOM Document.
 */
```

Example 21-9. XSLT in Mozilla and Internet Explorer (continued)

```
XML.Transformer = function(stylesheet) {
    // Load the stylesheet if necessary.
    if (typeof stylesheet == "string") stylesheet = XML.load(stylesheet);
    this.stylesheet = stylesheet;

    // In Mozilla-based browsers, create an XSLTProcessor object and
    // tell it about the stylesheet.
    if (typeof XSLTProcessor != "undefined") {
        this.processor = new XSLTProcessor( );
        this.processor.importStylesheet(this.stylesheet);
    }
};

/**
 * This is the transform( ) method of the XML.Transformer class.
 * It transforms the specified xml node using the encapsulated stylesheet.
 * The results of the transformation are assumed to be HTML and are used to
 * replace the content of the specified element.
 */
XML.Transformer.prototype.transform = function(node, element) {
    // If element is specified by id, look it up.
    if (typeof element == "string") element = document.getElementById(element);

    if (this.processor) {
        // If we've created an XSLTProcessor (i.e., we're in Mozilla) use it.
        // Transform the node into a DOM DocumentFragment.
        var fragment = this.processor.transformToFragment(node, document);
        // Erase the existing content of element.
        element.innerHTML = "";
        // And insert the transformed nodes.
        element.appendChild(fragment);
    }
    else if ("transformNode" in node) {
        // If the node has a transformNode( ) function (in IE), use that.
        // Note that transformNode( ) returns a string.
        element.innerHTML = node.transformNode(this.stylesheet);
    }
    else {
        // Otherwise, we're out of luck.
        throw "XSLT is not supported in this browser";
    }
};

/**
 * This is an XSLT utility function that is useful when a stylesheet is
 * used only once.
 */
XML.transform = function(xmldoc, stylesheet, element) {
    var transformer = new XML.Transformer(stylesheet);
    transformer.transform(xmldoc, element);
}
```

At the time of this writing, IE and Mozilla-based browsers are the only major ones that provide a JavaScript API to XSLT. If support for other browsers is important to you, you might be interested in the AJAXSLT open-source JavaScript XSLT implementation. AJAXSLT originated at Google and is under development at *http://goog-ajaxslt.sourceforge.net*.

21.4 Querying XML with XPath

XPath is a simple language that refers to elements, attributes, and text within an XML document. An XPath expression can refer to an XML element by its position in the document hierarchy or can select an element based on the value of (or simple presence of) an attribute. A full discussion of XPath is beyond the scope of this chapter, but Section 21.4.1 presents a simple XPath tutorial that explains common XPath expressions by example.

The W3C has drafted an API for selecting nodes in a DOM document tree using an XPath expression. Firefox and related browsers implement this W3C API using the `evaluate()` method of the Document object (for both HTML and XML documents). Mozilla-based browsers also implement `Document.createExpression()`, which compiles an XPath expression so that it can be efficiently evaluated multiple times.

IE provides XPath expression evaluation with the `selectSingleNode()` and `selectNodes()` methods of XML (but not HTML) Document and Element objects. Later in this section, you'll find example code that uses both the W3C and IE APIs.

If you wish to use XPath with other browsers, consider the open-source AJAXSLT project at *http://goog-ajaxslt.sourceforge.net*.

21.4.1 XPath Examples

If you understand the tree structure of a DOM document, it is easy to learn simple XPath expressions by example. In order to understand these examples, though, you must know that an XPath expression is evaluated in relation to some *context* node within the document. The simplest XPath expressions simply refer to children of the context node:

```
contact            // The set of all <contact> tags beneath the context node
contact[1]         // The first <contact> tag beneath the context
contact[last()]    // The last <contact> child of the context node
contact[last()-1]  // The penultimate <contact> child of the context node
```

Note that XPath array syntax uses 1-based arrays instead of JavaScript-style 0-based arrays.

The "path" in the name XPath refers to the fact that the language treats levels in the XML element hierarchy like directories in a filesystem and uses the "/" character to separate levels of the hierarchy. Thus:

```
contact/email      // All <email> children of <contact> children of context
/contacts          // The <contacts> child of the document root (leading /)
contact[1]/email   // The <email> children of the first <contact> child
contact/email[2]   // The 2nd <email> child of any <contact> child of context
```

Note that `contact/email[2]` evaluates to the set of `<email>` elements that are the second `<email>` child of any `<contact>` child of the context node. This is not the same as `contact[2]/email` or `(contact/email)[2]`.

A dot (`.`) in an XPath expression refers to the context element. And a double-slash (`//`) elides levels of the hierarchy, referring to any descendant instead of an immediate child. For example:

```
.//email         // All <email> descendants of the context
//email          // All <email> tags in the document (note leading slash)
```

XPath expressions can refer to XML attributes as well as elements. The `@` character is used as a prefix to identify an attribute name:

```
@id            // The value of the id attribute of the context node
contact/@name  // The values of the name attributes of <contact> children
```

The value of an XML attribute can filter the set of elements returned by an XPath expression. For example:

```
contact[@personal="true"]  // All <contact> tags with attribute personal="true"
```

To select the textual content of XML elements, use the `text()` method:

```
contact/email/text()  // The text nodes within <email> tags
//text()              // All text nodes in the document
```

XPath is namespace-aware, and you can include namespace prefixes in your expressions:

```
//xsl:template     // Select all <xsl:template> elements
```

When you evaluate an XPath expression that uses namespaces, you must, of course, provide a mapping of namespace prefixes to namespace URLs.

These examples are just a survey of common XPath usage patterns. XPath has other syntax and features not described here. One example is the `count()` function, which returns the number of nodes in a set rather than returning the set itself:

```
count(//email)    // The number of <email> elements in the document
```

21.4.2 Evaluating XPath Expressions

Example 21-10 shows an XML.XPathExpression class that works in IE and in standards-compliant browsers such as Firefox.

Example 21-10. Evaluating XPath expressions

```
/**
 * XML.XPathExpression is a class that encapsulates an XPath query and its
 * associated namespace prefix-to-URL mapping. Once an XML.XPathExpression
 * object has been created, it can be evaluated one or more times (in one
 * or more contexts) using the getNode( ) or getNodes( ) methods.
 *
 * The first argument to this constructor is the text of the XPath expression.
 *
 * If the expression includes any XML namespaces, the second argument must
 * be a JavaScript object that maps namespace prefixes to the URLs that define
 * those namespaces. The properties of this object are the prefixes, and
 * the values of those properties are the URLs.
 */
XML.XPathExpression = function(xpathText, namespaces) {
    this.xpathText = xpathText;    // Save the text of the expression
    this.namespaces = namespaces;  // And the namespace mapping

    if (document.createExpression) {
        // If we're in a W3C-compliant browser, use the W3C API
        // to compile the text of the XPath query
        this.xpathExpr =
            document.createExpression(xpathText,
                                      // This function is passed a
                                      // namespace prefix and returns the URL.
                                      function(prefix) {
                                          return namespaces[prefix];
                                      });
    }
    else {
        // Otherwise, we assume for now that we're in IE and convert the
        // namespaces object into the textual form that IE requires.
        this.namespaceString = "";
        if (namespaces != null) {
            for(var prefix in namespaces) {
                // Add a space if there is already something there
                if (this.namespaceString) this.namespaceString += ' ';
                // And add the namespace
                this.namespaceString += 'xmlns:' + prefix + '="' +
                    namespaces[prefix] + '"';
            }
        }
    }
};

/**
 * This is the getNodes( ) method of XML.XPathExpression. It evaluates the
 * XPath expression in the specified context. The context argument should
 * be a Document or Element object. The return value is an array
 * or array-like object containing the nodes that match the expression.
 */
XML.XPathExpression.prototype.getNodes = function(context) {
    if (this.xpathExpr) {
```

Example 21-10. Evaluating XPath expressions (continued)

```
        // If we are in a W3C-compliant browser, we compiled the
        // expression in the constructor. We now evaluate that compiled
        // expression in the specified context.
        var result =
            this.xpathExpr.evaluate(context,
                                    // This is the result type we want
                                    XPathResult.ORDERED_NODE_SNAPSHOT_TYPE,
                                    null);

        // Copy the results we get into an array.
        var a = new Array(result.snapshotLength);
        for(var i = 0; i < result.snapshotLength; i++) {
            a[i] = result.snapshotItem(i);
        }
        return a;
    }
    else {
        // If we are not in a W3C-compliant browser, attempt to evaluate
        // the expression using the IE API.
        try {
            // We need the Document object to specify namespaces
            var doc = context.ownerDocument;
            // If the context doesn't have ownerDocument, it is the Document
            if (doc == null) doc = context;
            // This is IE-specific magic to specify prefix-to-URL mapping
            doc.setProperty("SelectionLanguage", "XPath");
            doc.setProperty("SelectionNamespaces", this.namespaceString);

            // In IE, the context must be an Element not a Document,
            // so if context is a document, use documentElement instead
            if (context == doc) context = doc.documentElement;
            // Now use the IE method selectNodes() to evaluate the expression
            return context.selectNodes(this.xpathText);
        }
        catch(e) {
            // If the IE API doesn't work, we just give up
            throw "XPath not supported by this browser.";
        }
    }
}


/**
 * This is the getNode() method of XML.XPathExpression. It evaluates the
 * XPath expression in the specified context and returns a single matching
 * node (or null if no node matches). If more than one node matches,
 * this method returns the first one in the document.
 * The implementation differs from getNodes() only in the return type.
 */
XML.XPathExpression.prototype.getNode = function(context) {
    if (this.xpathExpr) {
        var result =
```

Example 21-10. Evaluating XPath expressions (continued)

```
            this.xpathExpr.evaluate(context,
                                    // We just want the first match
                                    XPathResult.FIRST_ORDERED_NODE_TYPE,
                                    null);
        return result.singleNodeValue;
    }
    else {
        try {
            var doc = context.ownerDocument;
            if (doc == null) doc = context;
            doc.setProperty("SelectionLanguage", "XPath");
            doc.setProperty("SelectionNamespaces", this.namespaceString);
            if (context == doc) context = doc.documentElement;
            // In IE call selectSingleNode instead of selectNodes
            return context.selectSingleNode(this.xpathText);
        }
        catch(e) {
            throw "XPath not supported by this browser.";
        }
    }
};

// A utility to create an XML.XPathExpression and call getNodes( ) on it
XML.getNodes = function(context, xpathExpr, namespaces) {
    return (new XML.XPathExpression(xpathExpr, namespaces)).getNodes(context);
};

// A utility to create an XML.XPathExpression and call getNode( ) on it
XML.getNode  = function(context, xpathExpr, namespaces) {
    return (new XML.XPathExpression(xpathExpr, namespaces)).getNode(context);
};
```

21.4.3 More on the W3C XPath API

Because of the limitations in the IE XPath API, the code in Example 21-10 handles only queries that evaluate to a document node or set of nodes. It is not possible in IE to evaluate an XPath expression that returns a string of text or a number. This is possible with the W3C standard API, however, using code that looks like this:

```
// How many <p> tags in the document?
var n = document.evaluate("count(//p)", document, null,
                          XPathResult.NUMBER_TYPE, null).numberValue;
// What is the text of the 2nd paragraph?
var text = document.evaluate("//p[2]/text( )", document, null,
                             XPathResult.STRING_TYPE, null).stringValue;
```

There are two things to note about these simple examples. First, they use the `document.evaluate( )` method to evaluate an XPath expression directly without compiling it first. The code in Example 21-10 instead used `document.createExpression( )` to compile an XPath expression into a form that could be reused. Second, notice that

these examples are working with HTML <p> tags in the document object. In Firefox, XPath queries can be used on HTML documents as well as XML documents.

See Document, XPathExpression, and XPathResult in Part IV for complete details on the W3C XPath API.

21.5 Serializing XML

It is sometimes useful to *serialize* an XML document (or some subelement of the document) by converting it to a string. One reason you might do this is to send an XML document as the body of an HTTP POST request generated with the XMLHttpRequest object. Another common reason to serialize XML documents and elements is for use in debugging messages!

In Mozilla-based browsers, serialization is done with an XMLSerializer object. In IE, it is even easier: the xml property of an XML Document or Element object returns the serialized form of the document or element.

Example 21-11 shows serialization code that works in Mozilla and IE.

Example 21-11. Serializing XML

```
/**
 * Serialize an XML Document or Element and return it as a string.
 */
XML.serialize = function(node) {
    if (typeof XMLSerializer != "undefined")
        return (new XMLSerializer( )).serializeToString(node);
    else if (node.xml) return node.xml;
    else throw "XML.serialize is not supported or can't serialize " + node;
};
```

21.6 Expanding HTML Templates with XML Data

One key feature of IE's XML data islands is that they can be used with an automatic templating facility in which data from a data island is automatically inserted into HTML elements. These HTML templates are defined in IE by adding datasrc and datafld ("fld" is short for "field") attributes to the elements.

This section applies the XML techniques seen earlier in the chapter and uses XPath and the DOM to create an improved templating facility that works in IE and Firefox. A template is any HTML element with a datasource attribute. The value of this attribute should be the ID of an XML data island or the URL of an external XML document. The template element should also have a foreach attribute. The value of this attribute is an XPath expression that evaluates to the list of nodes from which XML data will be extracted. For each XML node returned by the foreach expression, an expanded copy of the template is inserted into the HTML document. The template is expanded by finding all elements within it that have a data attribute. This attribute is

another XPath expression to be evaluated in the context of a node returned by the foreach expression. This data expression is evaluated with XML.getNode( ), and the text contained by the returned node is used as the content of the HTML element on which the data attribute was defined.

This description becomes much clearer with a concrete example. Example 21-12 is a simple HTML document that includes an XML data island and a template that uses it. It has an onload( ) event handler that expands the template.

Example 21-12. An XML data island and HTML template

```
<html>
<!-- Load our XML utilities for data islands and templates -->
<head><script src="xml.js"></script></head>
<!-- Expand all templates on document load -->
<body onload="XML.expandTemplates( )">

<!-- This is an XML data island with our data -->
<xml id="data" style="display:none"> <!-- hidden with CSS -->
  <contacts>
    <contact name="Able Baker"><email>able@example.com</email></contact>
    <contact name="Careful Dodger"><email>dodger@example.com</email></contact>
    <contact name="Eager Framer"><email>framer@example.com</email></contact>
  </contacts>
</xml>

<!-- These are just regular HTML elements -->
<table>
<tr><th>Name</th><th>Address</th></tr>
<!-- This is a template. Data comes from the data island with id "data". -->
<!-- The template will be expanded and copied once for each <contact> tag -->
<tr datasource="#data" foreach="//contact">
<!-- The "name" attribute of the <contact> is inserted into this element -->
<td data="@name"></td>
<!-- The content of the <email> child of the <contact> goes in here -->
<td data="email"></td>
</tr> <!-- end of the template -->
</table>
</body>
</html>
```

A critical piece of Example 21-12 is the onload event handler, which calls a function named XML.expandTemplates( ). Example 21-13 shows the implementation of this function. The code is fairly platform-independent, relying on basic Level 1 DOM functionality and on the XPath utility functions XML.getNode( ) and XML.getNodes( ) defined in Example 21-10.

Example 21-13. Expanding HTML templates

```
/*
 * Expand any templates at or beneath element e.
 * If any of the templates use XPath expressions with namespaces, pass
```

Example 21-13. Expanding HTML templates (continued)

```
 * a prefix-to-URL mapping as the second argument as with XML.XPathExpression( )
 *
 * If e is not supplied, document.body is used instead. A common
 * use case is to call this function with no arguments in response to an
 * onload event handler. This automatically expands all templates.
 */
XML.expandTemplates = function(e, namespaces) {
    // Fix up arguments a bit.
    if (!e) e = document.body;
    else if (typeof e == "string") e = document.getElementById(e);
    if (!namespaces) namespaces = null; // undefined does not work

    // An HTML element is a template if it has a "datasource" attribute.
    // Recursively find and expand all templates. Note that we don't
    // allow templates within templates.
    if (e.getAttribute("datasource")) {
        // If it is a template, expand it.
        XML.expandTemplate(e, namespaces);
    }
    else {
        // Otherwise, recurse on each of the children. We make a static
        // copy of the children first so that expanding a template doesn't
        // mess up our iteration.
        var kids = []; // To hold copy of child elements
        for(var i = 0; i < e.childNodes.length; i++) {
            var c = e.childNodes[i];
            if (c.nodeType == 1) kids.push(e.childNodes[i]);
        }

        // Now recurse on each child element
        for(var i = 0; i < kids.length; i++)
            XML.expandTemplates(kids[i], namespaces);
    }
};

/**
 * Expand a single specified template.
 * If the XPath expressions in the template use namespaces, the second
 * argument must specify a prefix-to-URL mapping
 */
XML.expandTemplate = function(template, namespaces) {
    if (typeof template=="string") template=document.getElementById(template);
    if (!namespaces) namespaces = null; // Undefined does not work

    // The first thing we need to know about a template is where the
    // data comes from.
    var datasource = template.getAttribute("datasource");

    // If the datasource attribute begins with '#', it is the name of
    // an XML data island. Otherwise, it is the URL of an external XML file.
    var datadoc;
    if (datasource.charAt(0) == '#')   // Get data island
```

Example 21-13. Expanding HTML templates (continued)

```
        datadoc = XML.getDataIsland(datasource.substring(1));
    else                                // Or load external document
        datadoc = XML.load(datasource);

    // Now figure out which nodes in the datasource will be used to
    // provide the data. If the template has a foreach attribute,
    // we use it as an XPath expression to get a list of nodes. Otherwise,
    // we use all child elements of the document element.
    var datanodes;
    var foreach = template.getAttribute("foreach");
    if (foreach) datanodes = XML.getNodes(datadoc, foreach, namespaces);
    else {
        // If there is no "foreach" attribute, use the element
        // children of the documentElement
        datanodes = [];
        for(var c=datadoc.documentElement.firstChild; c!=null; c=c.nextSibling)
            if (c.nodeType == 1) datanodes.push(c);
    }

    // Remove the template element from its parent,
    // but remember the parent, and also the nextSibling of the template.
    var container = template.parentNode;
    var insertionPoint = template.nextSibling;
    template = container.removeChild(template);

    // For each element of the datanodes array, we'll insert a copy of
    // the template back into the container. Before doing this, though, we
    // expand any child in the copy that has a "data" attribute.
    for(var i = 0; i < datanodes.length; i++) {
        var copy = template.cloneNode(true);           // Copy template
        expand(copy, datanodes[i], namespaces);        // Expand copy
        container.insertBefore(copy, insertionPoint);  // Insert copy
    }

    // This nested function finds any child elements of e that have a data
    // attribute. It treats that attribute as an XPath expression and
    // evaluates it in the context of datanode. It takes the text value of
    // the XPath result and makes it the content of the HTML node being
    // expanded. All other content is deleted.
    function expand(e, datanode, namespaces) {
        for(var c = e.firstChild; c != null; c = c.nextSibling) {
            if (c.nodeType != 1) continue;  // elements only
            var dataexpr = c.getAttribute("data");
            if (dataexpr) {
                // Evaluate XPath expression in context.
                var n = XML.getNode(datanode, dataexpr, namespaces);
                // Delete any content of the element
                c.innerHTML = "";
                // And insert the text content of the XPath result
                c.appendChild(document.createTextNode(getText(n)));
            }
            // If we don't expand the element, recurse on it.
```

Example 21-13. Expanding HTML templates (continued)

```
            else expand(c, datanode, namespaces);
        }
    }

    // This nested function extracts the text from a DOM node, recursing
    // if necessary.
    function getText(n) {
        switch(n.nodeType) {
        case 1: /* element */
            var s = "";
            for(var c = n.firstChild; c != null; c = c.nextSibling)
                s += getText(c);
            return s;
        case 2: /* attribute*/
        case 3: /* text */
        case 4: /* cdata */
            return n.nodeValue;
        default:
            return "";
        }
    }

};
```

21.7 XML and Web Services

Web services represent an important use for XML, and SOAP is a popular web service protocol that is entirely XML-based. In this section, I'll show you how to use the XMLHttpRequest object and XPath queries to make a simple SOAP request to a web service.

Example 21-14 is JavaScript code that constructs an XML document representing a SOAP request and uses `XMLHttpRequest` to send it to a web service. (The web service returns the conversion rate between the currencies of two countries.) The code then uses an XPath query to extract the result from the SOAP response returned by the server.

Before considering the code, here are some caveats. First, details on the SOAP protocol are beyond the scope of this chapter, and this example demonstrates a simple SOAP request and SOAP response without any attempt to explain the protocol or the XML format. Second, the example does not use Web Services Definition Language (WSDL) files to look up web service details. The server URL, method, and parameter name are all hardcoded into the sample code.

The third caveat is a big one. The use of web services from client-side JavaScript is severely constrained by the same-origin security policy (see Section 13.8.2). Recall that the same-origin policy prevents client-side scripts from connecting to, or accessing

data from, any host other than the one from which the document that contains the script was loaded. This means that JavaScript code for accessing a web service is typically useful only if it is hosted on the same server as the web service itself. Web service implementors may want to use JavaScript to provide a simple HTML-based interface to their services, but the same-origin policy precludes the widespread use of client-side JavaScript to aggregate the results of web services from across the Internet onto a single web page.

In order to run Example 21-14 in IE, you can relax the same-origin security policy. Select **Tools → Internet Options → Security** and then click on the **Internet** tab in the resulting dialog. Scroll through the list of security options to find one named **Access data sources across domains**. This option is usually (and should be) set to **disabled**. In order to run this example, change it to **prompt**.

To allow Example 21-14 to run in Firefox, the example includes a call to the Firefox-specific `enablePrivilege()` method. This call prompts the user to grant enhanced privileges to the script so that it can override the same-origin policy. This works when the example is run from a `file:` URL in the local filesystem but does not work if the example is downloaded from a web server (unless the script has been digitally signed, which is beyond the scope of this book).

With those caveats out of the way, let's move on to the code.

Example 21-14. Querying a web service with SOAP

```
/**
 * This function returns the exchange rate between the currencies of two
 * countries. It determines the exchange rate by making a SOAP request to a
 * demonstration web service hosted by XMethods (http://www.xmethods.net).
 * The service is for demonstration only and is not guaranteed to be
 * responsive, available, or to return accurate data. Please do not
 * overload XMethod's servers by running this example too often.
 * See http://www.xmethods.net/v2/demoguidelines.html
 */
function getExchangeRate(country1, country2) {
    // In Firefox, we must ask the user to grant the privileges we need to run.
    // We need special privileges because we're talking to a web server other
    // than the one that served the document that contains this script. UniversalXPConnect
    // allows us to make an XMLHttpRequest to the server, and
    // UniversalBrowserRead allows us to look at its response.
    // In IE, the user must instead enable "Access data sources across domains"
    // in the Tools->Internet Options->Security dialog.
    if (typeof netscape != "undefined") {
        netscape.security.PrivilegeManager.
                enablePrivilege("UniversalXPConnect UniversalBrowserRead");
    }

    // Create an XMLHttpRequest to issue the SOAP request. This is a utility
    // function defined in the last chapter.
```

Example 21-14. Querying a web service with SOAP (continued)

```
    var request = HTTP.newRequest( );

    // We're going to be POSTing to this URL and want a synchronous response
    request.open("POST", "http://services.xmethods.net/soap", false);

    // Set some headers: the body of this POST request is XML
    request.setRequestHeader("Content-Type", "text/xml");

    // This header is a required part of the SOAP protocol
    request.setRequestHeader("SOAPAction", '""');

    // Now send an XML-formatted SOAP request to the server
    request.send(
        '<?xml version="1.0" encoding="UTF-8"?>' +
        '<soap:Envelope' +
        '  xmlns:ex="urn:xmethods-CurrencyExchange"' +
        '  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"' +
        '  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"' +
        '  xmlns:xs="http://www.w3.org/2001/XMLSchema"' +
        '  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">' +
        '   <soap:Body ' +
        '     soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">'+
        '      <ex:getRate>' +
        '         <country1 xsi:type="xs:string">' + country1 + '</country1>' +
        '         <country2 xsi:type="xs:string">' + country2 + '</country2>' +
        '      </ex:getRate>' +
        '   </soap:Body>' +
        '</soap:Envelope>'
        );

    // If we got an HTTP error, throw an exception
    if (request.status != 200) throw request.statusText;

    // This XPath query gets us the <getRateResponse> element from the document
    var query = "/s:Envelope/s:Body/ex:getRateResponse";

    // This object defines the namespaces used in the query
    var namespaceMapping = {
        s:  "http://schemas.xmlsoap.org/soap/envelope/",  // SOAP namespace
        ex: "urn:xmethods-CurrencyExchange" // the service-specific namespace
    };

    // Extract the <getRateResponse> element from the response document
    var responseNode=XML.getNode(request.responseXML, query, namespaceMapping);

    // The actual result is contained in a text node within a <Result> node
    // within the <getRateReponse>
    return responseNode.firstChild.firstChild.nodeValue;
}
```

21.8 E4X: ECMAScript for XML

ECMAScript for XML, better known as E4X, is a standard extension* to JavaScript that defines a number of powerful features for processing XML documents. At the time of this writing, E4X is not widely available. Firefox 1.5 supports it, and it is also available in version 1.6 of Rhino, the Java-based JavaScript interpreter. Microsoft does not plan to support it in IE 7, and it is not clear when or whether other browsers will add support.

Although it is an official standard, E4X is not yet widely enough deployed to warrant full coverage in this book. Despite its limited availability, though, the powerful and unique features of E4X deserve some coverage. This section presents an overview-by-example of E4X. Future editions of this book may expand the coverage.

The most striking thing about E4X is that XML syntax becomes part of the JavaScript language, and you can include XML literals like these directly in your JavaScript code:

```
// Create an XML object
var pt =
    <periodictable>
      <element id="1"><name>Hydrogen</name></element>
      <element id="2"><name>Helium</name></element>
      <element id="3"><name>Lithium</name></element>
    </periodictable>;

// Add a new element to the table
pt.element += <element id="4"><name>Beryllium</name></element>;
```

The XML literal syntax of E4X uses curly braces as escape characters that allow you to place JavaScript expressions within XML. This, for example, is another way to create the XML element just shown:

```
pt = <periodictable></periodictable>;              // Start with empty table
var elements = ["Hydrogen", "Helium", "Lithium"];  // Elements to add
// Create XML tags using array contents
for(var n = 0; n < elements.length; n++) {
    pt.element += <element id={n+1}><name>{elements[n]}</name></element>;
}
```

In addition to this literal syntax, you can also work with XML parsed from strings. The following code adds another element to your periodic table:

```
pt.element += new XML('<element id="5"><name>Boron</name></element>');
```

When working with XML fragments, use `XMLList()` instead of `XML()`:

```
pt.element += new XMLList('<element id="6"><name>Carbon</name></element>' +
                          '<element id="7"><name>Nitrogen</name></element>');
```

* E4X is defined by the ECMA-357 standard. You can find the official specification at *http://www.ecma-international.org/publications/standards/Ecma-357.htm*.

Once you have an XML document defined, E4X defines an intuitive syntax for accessing its content:

```
var elements = pt.element;    // Evaluates to a list of all <element> tags
var names = pt.element.name;  // A list of all <name> tags
var n = names[0];             // "Hydrogen": content of <name> tag 0.
```

E4X also adds new syntax for working with XML objects. The `..` operator is the descendant operator; you can use it in place of the normal `.` member-access operator:

```
// Here is another way to get a list of all <name> tags
var names2 = pt..name;
```

E4X even has a wildcard operator:

```
// Get all descendants of all <element> tags.
// This is yet another way to get a list of all <name> tags.
var names3 = pt.element.*;
```

Attribute names are distinguished from tag names in E4X using the `@` character (a syntax borrowed from XPath). For example, you can query the value of an attribute like this:

```
// What is the atomic number of Helium?
var atomicNumber = pt.element[1].@id;
```

The wildcard operator for attribute names is `@*`:

```
// A list of all attributes of all <element> tags
var atomicNums = pt.element.@*;
```

E4X even includes a powerful and remarkably concise syntax for filtering a list using an arbitrary predicate:

```
// Start with a list of all elements and filter it so
// it includes only those whose id attribute is < 3
var lightElements = pt.element.(@id < 3);

// Start with a list of all <element> tags and filter so it includes only
// those whose names begin with "B". Then make a list of the <name> tags
// of each of those remaining <element> tags.
var bElementNames = pt.element.(name.charAt(0) == 'B').name;
```

E4X defines a new looping statement for iterating through lists of XML tags and attributes. The `for/each/in` loop is like the `for/in` loop, except that instead of iterating through the properties of an object, it iterates through the values of the properties of an object:

```
// Print the names of each element in the periodic table
// (Assuming you have a print( ) function defined.)
for each (var e in pt.element)  {
    print(e.name);
}

// Print the atomic numbers of the elements
for each (var n in pt.element.@*) print(n);
```

In E4X-enabled browsers, this `for/each/in` loop is also useful for iterating through arrays.

E4X expressions can appear on the left side of an assignment. This allows existing tags and attributes to be changed and new tags and attributes to be added:

```
// Modify the <element> tag for Hydrogen to add a new attribute
// and a new child element so that it looks like this:
//
// <element id="1" symbol="H">
//   <name>Hydrogen</name>
//   <weight>1.00794</weight>
// </element>
//
pt.element[0].@symbol = "H";
pt.element[0].weight = 1.00794;
```

Removing attributes and tags is also easy with the standard `delete` operator:

```
delete pt.element[0].@symbol; // delete an attribute
delete pt..weight;            // delete all <weight> tags
```

E4X is designed so that you can perform most common XML manipulations using language syntax. E4X also defines methods you can invoke on XML objects. Here, for example, is the `insertChildBefore( )` method:

```
pt.insertChildBefore(pt.element[1],
                     <element id="1"><name>Deuterium</name></element>);
```

Note that the objects created and manipulated by E4X expressions are XML objects. They are not DOM Node or Element objects and do not interoperate with the DOM API. The E4X standard defines an optional XML method `domNode( )` that returns a DOM Node equivalent to the XML object, but this method is not implemented in Firefox 1.5. Similarly, the E4X standard says a DOM Node can be passed to the `XML( )` constructor to obtain the E4X equivalent of the DOM tree. This feature is also unimplemented in Firefox 1.5, which restricts the utility of E4X for client-side JavaScript.

E4X is fully namespace-aware and includes language syntax and APIs for working with XML namespaces. For simplicity, though, the examples shown here have not illustrated this syntax.