5

Data Communication: XML, XSLT, and JSON

An important part of any type of distributed application is how data is pushed around between tiers or layers of the application. Additionally, with Ajax, several concepts are fairly important to know and understand, concepts involved with building distributed heterogeneous environments. Accordingly, in this chapter, you are going to look at:

- **XML** — XML is Extensible Markup Language. It is primarily used for data interchange.
- **XSLT** — XSLT is Extensible Stylesheet Language Transformations. XSLT is designed to take XML data from one format and put it into another format.
- **JSON** — JSON is the JavaScript Object Notation. JSON is a lightweight data interchange format.

When tied together with web services, XML and JSON allow for data interchange between different operating systems and also across the Internet. This is a major change from systems that are heavily tied together and require that each system run a specific operating system merely because of the format the data is communicated in. Another advantage web services provide these data interchange formats is that web services typically run on HTTP. HTTP runs on port 80. Port 80 is a very widely used port and is not blocked, unlike many other ports and protocols, such as Microsoft's Distributed Component Object Model (DCOM) objects.

Trying to put all information about XML, XSLT, and JSON into one chapter will not do any of the subjects justice. This chapter attempts to cover enough information regarding these products so that the reader will have a basic understanding of these technologies as they relate to data communication. However, these topics can each fill a thousand-page book and not completely cover the subject. Wrox offers several good, complete books on the subject of XML and XSLT, including Beginning XML, *Third Edition (Wiley, 2004),* XSLT 2.0 Programmer's Reference, *Third Edition (Wiley, 2004),* X Path 2.0 Programmer's Reference *(Wiley, 2004). You can find these and other titles at* `www.wrox.com`.

This chapter assumes that you are using Visual Studio 2005 and .NET 2.0. Also, you can download the code samples for this chapter (Chapter 5) at `http://beginningajax.com`.

XML

I won't bother you with all of the grand hoopla (and already well covered) talk about how XML will do this or that. Suffice it to say, it is fairly widely used. XML work began at the level of the W3C in 1996. It was formally introduced in February 1998. XML won't wax your floors, and it is not a dessert topping; however, take a quick look at what XML really is and what features it has that are well designed for data transfer and storage:

- **XML is based on Standard Generalized Markup Language (SGML)** — Having a format based on existing international standards, such as SGML, which has been used since 1986, means that you have an existing body of knowledge to draw upon.
- **It has a self-describing format** — The structure and field names are well known.
- **It has textual representation of data** — This textual representation of data allows computer science data structures to be represented textually. These data structures include trees, lists, and records.
- **It is both human- and machine-readable** — This is an improvement over binary/machine data because humans can read it, which allows for simple visual inspection of the data.
- **It has multiple-language support** — It supports information in any human language as well as ASCII and Unicode.
- **It is efficient and consistent** — Its strict syntax and parsing requirements allow the parsing algorithms to perform efficiently and consistently.
- **It is widely applicable** — The hierarchical structure of XML is appropriate for many types of documents.
- **It is platform-independent** — By being text-based, XML is relatively platform-independent and relatively immune to changes in technology.

All that said, XML is not the best solution for everything. XML has several weaknesses that programmers need to be aware of. These are:

- **XML is fairly verbose** — In some cases, an XML representation of data may be redundant. The result may be higher bandwidth and CPU processing needs. With compression, the storage cost and bandwidth needs may be negated; however, the cost may be increased storage CPU processing requirements. In addition, by compressing the data, the advantage of XML being human readable is almost certainly lost.
- **Data storage requirements do not support a wide array of datatypes.** The result is that the numeric *e* value of 2.781828 may not easily be identified as a floating-point value or a string with a width of eight characters. XML Schema and validation routines provide this support; however, XML does not natively provide this information.
- **Mapping to a relational database model may be difficult** — Complex XML layouts may not easily map to a relational database model.

History of XML

XML can trace its heritage back to the mid 1960s with roots in Generalized Markup Language (GML) from IBM. SGML is based on the GML work done at IBM. SGML is a metalanguage that was standardized in

1986 by ISO standard "ISO 8879:1986 Information processing — Text and office systems — Standard Generalized Markup Language (SGML)." After that there were several modifications of the standard to correct some omissions and clear up some loose ends.

In mid-1996, work began on integrating SGML and the web. During the working time, the group had various names for the specification until it finally decided on Extensible Markup Language (XML). This group worked throughout 1997, and on February 10, 1998, the W3C approved the result as a W3C recommendation. This is sometimes referred to as XML 1.0.

Minor revisions of XML 1.0 have taken place. The current revision is known as XML 1.0 Third Edition. This revision was published on February 4, 2004.

The XML 1.1 version was published on February 4, 2004, the same day as the XML 1.0 Third Edition. XML 1.1 has been primarily designed to assist those with unique needs, such as mainframe/host developers. XML 1.1 is not widely implemented at this time.

There is talk and discussion regarding future versions of XML. There is some talk regarding XML 2.0 that would have it eliminate Document Type Definitions (DTDs) from syntax, integration of namespaces, and the addition of other features into the XML standard. In addition, the W3C has had some preliminary discussion regarding the addition of binary encoding into the XML feature set. There is no official work to include binary encoding, merely discussion and investigation, at the time of this writing.

XML Documents

Take a look at what a simple valid XML document looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<menu>
   <menuitem>
      <item>Hamburger</item>
      <item>French Fries</item>
      <item flavor="Chocolate">Milk Shake</item>
      <cost>4.99</cost>
   </menuitem>
</menu>
```

The first line of the file is the *XML declaration*. This line is optional. It contains information regarding the version of XML, typically version 1.0, character encoding, and possibly external dependencies. Next, with XML, there must be one *root element*. In the preceding example, the `<menu>` tag is the root element.

The rest of the document contains a set of nested elements, which can be further divided into attributes and content. An *element* is typically defined with a start tag and an end tag. In the preceding example, you have a start tag of `<item>` and an end tag of `</item>` for your list of menu items. The end tag can be omitted if the start tag is defined as `<item />`. The content of the item is everything that is between the start and end tags. Elements may also contain attributes. *Attributes* are name/value pairs included within the start tag and after the element name. Attribute values must be quoted with a single or double quotation mark. Each attribute name should occur only once within an element. In the preceding code, the item with value of `Milk Shake` has an attribute. The name of the attribute is `flavor` and the value is `"Chocolate"`. In addition to text, elements may contain other elements. In the preceding code example, the `<menuitem>` element contains three individual `<item>` elements.

XML Document Correctness

For an XML document to be considered to be correct, it must meet two requirements. It must be:

- **Well formed** — A *well-formed* document meets all of the XML syntax rules. A parser must refuse to process a document that is not well formed. A well-formed document must conform to the following rules:
    - XML declaration, processing instructions, encoding, and other information may exist in the XML document.
    - Only one root element is specified in the XML document.
    - Elements that are not empty are defined with both a start tag and an end tag.
    - Empty elements may be marked with the empty element tag. An example is `<item />`.
    - Attributes must be enclosed in single or double quotation marks. Attribute names are case-sensitive.
    - Tags may be nested but must not overlap.
    - The document must use the specified encoding. UTF-8 is the default.
    - Elements names are case-sensitive.
- **Valid** — A *valid* document has data that conforms to a set of content rules. These rules describe correct data values and document organizational structure. (For example, the datatypes in an element must match.)
- XML documents that comply with a defined schema are defined as valid. An *XML Schema* is a description of a type of XML document. The schema is typically expressed as constraints on the structure and content of the document. This is above and beyond the basic constraints imposed by XML. You will find out more about schemas a bit later in this chapter.

Try It Out Creating Valid and Well-Formed XML

XML requires that elements be properly next and not overlap. The following example shows invalid XML because of the overlapping of tags with the `<item>` tag. In the example that follows, a tag is open, then a second tag of the same type is opened, the first `<item>` tag is closed, and then the second `<item>` tag is closed.

```
<?xml version="1.0" encoding="UTF-8"?>
<menu>
   <menuitem>
      <item>Hamburger<item></item>
            French Fries</item>
      <item flavor="Chocolate">Milk Shake</item>
   </menuitem>
</menu>
```

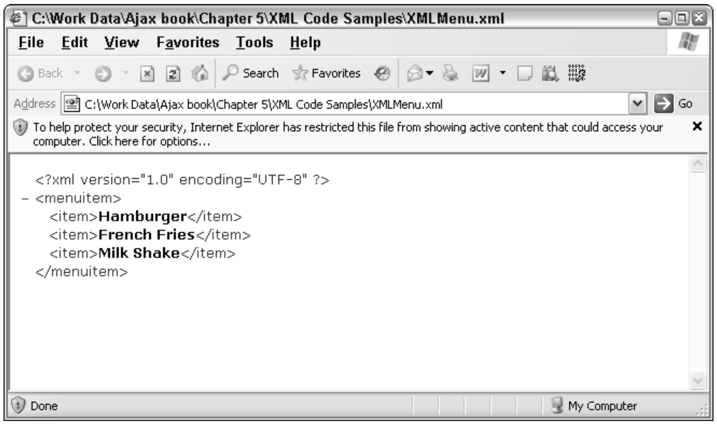
Consider the following code sample that is not well formed. This example contains no root element.

```
<?xml version="1.0" encoding="UTF-8"?>
<item>Hamburger</item>
<item>French Fries</item>
<item flavor="Chocolate">Milk Shake</item>
```

For this example to be well formed, a root element must be added. Now this document is set up so that multiple menu items may be created and is more along the lines of what is desired:

```
<?xml version="1.0" encoding="UTF-8"?>
<menu>
   <menuitem>
      <item>Hamburger</item>
      <item>French Fries</item>
      <item flavor="Chocolate">Milk Shake</item>
   </menuitem>
</menu>
```

Figure 5-1 shows a simple XML example after it has been parsed and displayed in Internet Explorer v6.


C:\Work Data\Ajax book\Chapter 5\XML Code Samples\XMLMenu.xml
File Edit View Favorites Tools Help
Back Search Favorites
Address C:\Work Data\Ajax book\Chapter 5\XML Code Samples\XMLMenu.xml
Go
Done
My Computer

To help protect your security, Internet Explorer has restricted this file from showing active content that could access your computer. Click here for options...

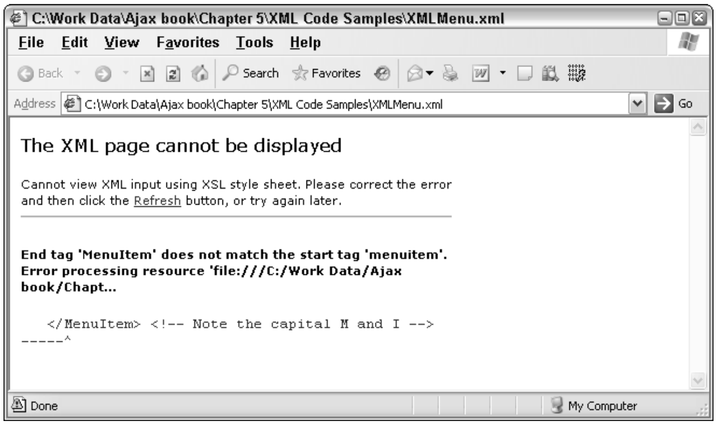
```
<?xml version="1.0" encoding="UTF-8" ?>
- <menuitem>
    <item>Hamburger</item>
    <item>French Fries</item>
    <item>Milk Shake</item>
  </menuitem>
```

Figure 5-1

Now compare the previous example to the following invalid XML:

```
<?xml version='1.0' encoding='UTF-8'?>
<menu>
   <menuitem >
      <item>Hamburger</item>
      <item>French Fries</item>
      <item>Milk Shake</item>
   </MenuItem> <!— Note the capital M and I -->
</menu>
```

Figure 5-2 shows the result in Internet Explorer. In this example, the closing `</MenuItem>` does not match the opening `<menuitem>` because the opening and closing tags do not match from the standpoint of case.


C:\Work Data\Ajax book\Chapter 5\XML Code Samples\XMLMenu.xml
File Edit View Favorites Tools Help
Back Search Favorites
Address C:\Work Data\Ajax book\Chapter 5\XML Code Samples\XMLMenu.xml Go
Done
My Computer

The XML page cannot be displayed

Cannot view XML input using XSL style sheet. Please correct the error and then click the Refresh button, or try again later.

End tag 'MenuItem' does not match the start tag 'menuitem'. Error processing resource 'file:///C:/Work Data/Ajax book/Chapt...

```
   </MenuItem> <!-- Note the capital M and I -->
-----^
```

Figure 5-2

Entity References and Numeric Character References

Within XML is the need to store special characters. These special characters are entity characters and numeric character references.

An *entity* in XML is body of data, typically text, which contains unusual characters. An *entity reference* in XML is a placeholder for an entity. XML contains several predefined entities The entity's name is prefixed with an ampersand (`&`) and ends with a semicolon (`;`). The following is a full list of redefined entities:

- ❑ The ampersand character (`&`) is defined as `&amp;`.
- ❑ The left bracket, or less than, character (`<`) is defined as `&lt;`.
- ❑ The right bracket, or greater than, character (`>`) is defined as `&gt;`.
- ❑ The single quote, or apostrophe, character (`'`) is defined as `&apos;`.
- ❑ The double quote character (`"`) is defined by `&quot;`.

Additional entities may be defined in a document's DTD — see the next section for more on DTDs.

Numeric character references are similar to entities. Instead of merely the `&` followed by the entity name and ending with the semicolon, a `&` and a `#` character are followed by a decimal or hexadecimal character representing a Unicode code point and a semicolon. For example, the `&` character may also be defined as `&#038;`.

DTD

A Document Type Definition (DTD) is the original syntax for XML. A DTD is inherited from SGML. Although a DTD is defined in the XML 1.0 standard, its use is limited for several reasons:

- DTDs lack support for new features of XML. For example, there is no support for namespaces.
- Not all parts of an XML document can be expressed within a DTD.
- DTDs use a syntax inherited from SGML, as opposed to an XML syntax.

XML Schema

The successor to DTDs is XML Schema. Another term for XML Schemas is XML Schema Definition (XSDs). XSDs are better suited for describing XML languages than DTDs. XSDs have some of the following features:

- Have datatyping support
- Allow for detailed constraints on a document's logical structure
- Are required for using XML in validation framework
- Use an XML-based format

Here is an example XML document for a test. In this example, the `MenuSchema` will be used to validate the menu XML document.

```
<?xml version="1.0" encoding="UTF-8"?>
<menu xmlns="http://my-example.com/menus"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://my-example.com/menus Menu2.xsd">
   <menuitem>
      <item>Hamburger</item>
      <cost>1.99</cost>
   </menuitem>
   <menuitem>
      <item>Drink</item>
      <cost>.99</cost>
   </menuitem>
   <menuitem>
      <item>Fries</item>
      <cost>.49</cost>
   </menuitem>
</menu>
```

Here is an example XSD file that will validate that document:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns="http://my-example.com/menus"
xmlns:xs="http://www.w3.org/2001/XMLSchema" targetNamespace="http://my-
example.com/menus" elementFormDefault="qualified"
attributeFormDefault="unqualified">
<xs:element name="menu">
   <xs:annotation>
      <xs:documentation>Menu is the root element</xs:documentation>
   </xs:annotation>
   <xs:complexType>
      <xs:sequence>
         <xs:element name="menuitem" maxOccurs="unbounded">
```

```
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="item" type="xs:string"/>
                     <xs:element name="cost" type="xs:decimal"/>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   </xs:schema>
```

In this example code, the `menuitem` is validated. The `<item>` tag is validated as a string, which is fairly easy to validate, and the `<cost>` tag is validated as a decimal. Within the `<menu>` tag, there are multiple copies of the `<menuitem>` tag that are allowed. Above the `<menuitem>` tag is the `<menu>` tag.

Try It Out Validating XML

Given that there is some XML and the need to validate the XML, take a look at some code to validate the XML in a .NET 2.0 application. In this example, the code creates an `XmlSchemaSet` object. This object holds the contents of the XML Schema file. An `XmlReaderSettings` object is created. This object holds the XML Schema and any other settings necessary for reading the XML feed. The `XmlReader` document is created from XML feed and the `XmlReaderSettings` object. Once the `XmlReader` object is created, any errors that occur when reading the XML document are raised and passed to the `XmlReaderSettings`' `ValidationEventHandler` method. In this example, the user is notified in a label.

ASPX page

```
<form id="form1" runat="server">
<div>
Load Result: <asp:Label ID="lblLoadResult" runat="server" Text=""></asp:Label>
</div>
</form>
```

Code-Behind

```
protected void Page_Load(object sender, EventArgs e)
{
    System.Xml.XmlReaderSettings xmlRDS = new System.Xml.XmlReaderSettings();
    System.Xml.Schema.XmlSchemaSet sc = new System.Xml.Schema.XmlSchemaSet();
    sc.Add("http://my-example.com/menus", Server.MapPath("Menu2.xsd"));
    xmlRDS = new System.Xml.XmlReaderSettings();
    xmlRDS.ValidationEventHandler += new
System.Xml.Schema.ValidationEventHandler(xmlRDS_ValidationEventHandler);
    xmlRDS.ValidationType = System.Xml.ValidationType.Schema;
    xmlRDS.Schemas.Add(sc);
    System.Xml.XmlReader xmlRd =
System.Xml.XmlReader.Create(Server.MapPath("Menu2.xml"), xmlRDS);
    while (xmlRd.Read())
    {
    }
}
private void xmlRDS_ValidationEventHandler(object sender,
System.Xml.Schema.ValidationEventArgs e)
```

```
{
    this.lblLoadResult.Text = "Validation Error: " + Convert.ToString(e.Message) +
"<br />";
}
```

The `XmlReaderSettings` object specifies the settings used in reading in some XML. This object is used with the static `.Create()` method when an `XmlReader` object is created. The properties to note are the `ValidationType`, which is set from an enumeration, and the `Schemas` property, which is based on the `SchemaSet` object. The `XmlSchemaSet` object contains a set of XML Schemas to validate against. If the XML is not valid based on the schema, an `XmlSchemaException()` is generated.

Parsing XML

Two popular types of XML processing exist — the Document Object Model (DOM) and the Simple API for XML (SAX). The key difference between these two approaches is that the first loads the entire XML document into an in-memory data structure, whereas the latter iterates over the XML document one piece at a time in a forward-only, read-only fashion.

DOM Parsing

The Document Object Model (DOM) is an API that allows access to XML and HTML documents and their elements. The DOM is programming-language- and platform-independent.

Typically, XML parsers have been developed that must make use of a tree structure will all elements fully loaded into the parser before any operations occur. As a result, DOM is best used for applications where the document elements are randomly accessed and manipulated.

There are several levels of DOM specification. These specification levels are:

- **Level 0** — A Level 0 DOM contains all of the vendor-specific DOMs that existed before the W3C standardization process.
- **Level 1** — A Level 1 DOM allows for the navigation of documents and modification of their content.
- **Level 2** — A Level 2 DOM contains support for XML namespaces, filtered views, and events.
- **Level 3** — A Level 3 DOM consists of support for:
    - DOM Level 3 Core
    - DOM Level 3 Load and Save
    - DOM Level 3 XPath
    - DOM Level 3 Views and Formatting
    - DOM Level 3 Requirements
    - DOM Level 3 Validation

For further information on the DOM, please refer to Chapter 3.

SAX Parsing

Simple API for XML (SAX) parsing is another form of processing of XML files. A SAX-based parser handles XML as a single stream of data that is available only unidirectionally. As a result, accessing previously used data will result in the XML stream being reread and reparsed.

SAX processing is based on asynchronous events. In this model, as the XML document is read and parsed, events are fired as set up by the program. This is believed to result in faster XML processing than the DOM, because of a much smaller memory footprint compared to using a fully loaded and parsed DOM tree. Truthfully, the speed comparison should be based on a specific program, so generalities like this do not hold up in all situations. The other problem with SAX, which is more significant than the unidirectional issues, is the event-driven programming model. Accurately creating an event-driven program can be very complicated and frustrating.

XML Summary

As you have noticed, XML can be a very complicated topic. In this section, we have attempted to cover the basic topics that you need regarding what XML is, but this is not meant to be a complete reference for XML, just a set of basic information. For more complete information, please reference the books mentioned at the beginning of the chapter. In the next section, we will look at the processing XML with a technology referred to as XSLT.

XSLT

Extensible Stylesheet Language Transformations (XSLT) is an XML-based language used to convert XML from one format to another. These other formats may be a different XML Schema, HTML, plain text, PDF, or some other format. XSLT grew out of the Extensible Stylesheet Language (XSL) development effort within the W3C. The XSLT specification was first published by the W3C as a recommendation on November 16, 1999.

How Processing Occurs

The XSLT language is declarative. Being declarative means that the XSLT stylesheet is made up of a set of templated rules. Each rule in the collection specifies what to add to the resulting output, and the result is then sent to the output. Once an XSLT processor finds a node that meets the processing conditions, instructions within the template rules are processed sequentially.

The XSLT specification defines a transformation in terms of source and results. This keeps from locking a developer into a set of system specific APIs.

XSLT uses the X Path language for identifying the appropriate data in the source tree. X Path also provides a range of functions that assist XSLT processing.

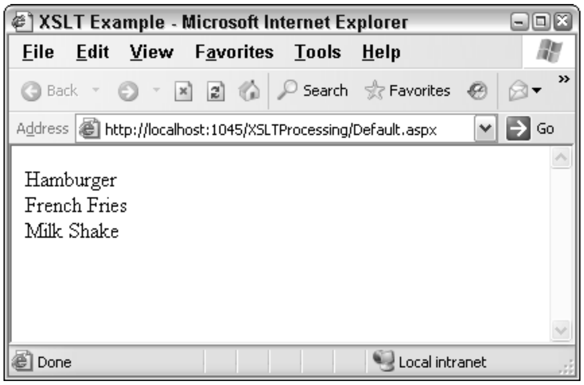
Now take a look at some example XSLT coding. First, consider the following sample XML:

```
<?xml version="1.0" encoding="UTF-8" ?>
<menuitem xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="MenuSchema.xsd">
   <item>Hamburger</item>
   <item>French Fries</item>
   <item>Milk Shake</item>
   <cost>4.99</cost>
</menuitem>
```

Suppose that from this code you want to pull out the elements within the `<item>` tags. The following XSLT code will pull out a list of items.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/1999/xhtml">
   <xsl:template match="/item">
      <xsl:apply-templates />
   </xsl:template>
   <xsl:template match="item">
      <xsl:value-of select="." />
      <br />
   </xsl:template>
   <xsl:template match="cost" />
</xsl:stylesheet>
```

This code works by looking for the matches for the `<item>` tag, pulling them out, and sending them to the output stream. With the .NET Framework, there is an XML Transform control. By setting the control's `DocumentSource` and `TransformSource` properties, the control may be used to easily output the results of an XML file being transformed by an XSLT file. The result of this XML file being transformed by the XSLT file is Figure 5-3.


XSLT Example - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites
Address http://localhost:1045/XSLTProcessing/Default.aspx Go
Done
Local intranet

Hamburger
French Fries
Milk Shake

Figure 5-3

Figure 5-4 shows how XSLT processing occurs at a high level.

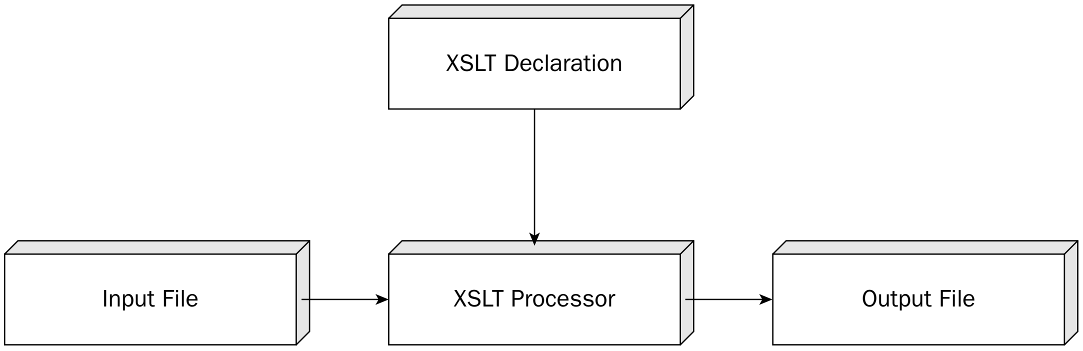


Figure 5-4

XSLT processing occurs in the following steps:

1. The XSLT stylesheet is loaded by the XML parser.
2. The XSLT stylesheet is converted into a tree of nodes. These nodes are also referred to as a *stylesheet tree*.
    a. XSLT stylesheet errors are detected.
    b. `Include` and `import` commands are handled.
3. The XML input is loaded by the parser and converted into a tree of nodes.
4. Whitespace only text nodes are removed from the stylesheet tree.
5. Whitespace-only text nodes are removed from the source tree.
6. The stylesheet tree is supplemented with built-in template rules for default processing.
7. The root node of the source tree is processed along with child nodes.
    a. The template rule that best matches a node is processed.
    b. Template rules are created. Elements are treated as instructions and their semantics are interpreted. Elements and text nodes in the template rules are copied specifically into the result tree.
8. The result tree is serialized, if necessary, according to any provided `xsl:output` instructions.

Built-In Functions

Like many of programming languages, XSLT provides a set of built-in commands. These commands range from string processing to date processing and looping operators, such as a `for` loop structure. In this section, we will look at some of the functions built into XSLT processors. These functions provide the basics upon which many XSLT operations are built.

XSLT <template>

The `<xsl:template>` element is used along with the `match` attribute to build templates and to associate them with XML elements.

```
<xsl:template match="item" />
```

In this example function, the XSLT processor will search for the node named `"item"`. If a match is made, then additional functionality is performed.

XSLT <value-of>

The XSLT `<xsl:value-of>` element is used to extract the value of an XML element. The extracted value is then added to the output's stream of the transformation. Consider the following example:

```
<xsl:value-of select="item" />
```

This code will pull the data from the `item` element of an XML file, and then add that value to the stream of the transform's output.

XSLT <for-each>

The XSLT `<xsl:for-each>` element is used to provide looping support in XSLT. Take a look at the following example:

```
<xsl:for-each select="item" />
   <xsl:value-of select="." />
</xsl:for-each>
```

This example will pull all the `item` nodes from the specified node set and return the value of the current node to the XSLT stream.

XSLT <sort>

The XSLT `<sort>` element is used to assist the `<xsl:for-each>` element in sorting the node set produced by the `<xsl:for-each>` element.

XSLT <if>

The `<xsl:if>` element is used to test the content of an XML file, much like the `if/endif` of a traditional computer programming language. Take a look at the following example:

```
<xsl:if test="cost &gt; 1.00">
..........
</xsl:if>
```

This `if/endif` are often used for conditional tests. With these tests, program execution can be altered based on these tests. If the condition is true, the commands within the `<xsl:if></xsl:if>` are executed. In this example, if the cost node has a value of greater than `1.00`, the code between the `<xsl:if>` and `</xsl:if>` is executed.

XSLT <choose>

The `<xsl:choose>` element is used along with `<xsl:when>` and `<xsl:otherwise>` to perform multiple conditional tests. Consider the following code:

```
<xsl:choose>
   <xsl:when test="expression">
   ...........
   </xsl:when>
   <xsl:otherwise>
```

```
      ...........
      </xsl:otherwise>
   </xsl:choose>
```

In this example, the expression is tested. When the `expression` is true, the code within the `<xsl:when></xsl:when>` tags is executed. If not code is executed, the processing will drop out to the `<xsl:otherwise></xsl:otherwise>` tags and execute within those tags.

Processing with XSLT

Now that you have looked at some of the main XSLT directives, take a look at some more complicated examples of processing XSLT. In this example, you are going to perform some string processing, conditional processing, and mathematical processing in some examples.

Try It Out String Processing

String processing is something that is very common to modern programming languages, and XSLT is no different. It has a number of built-in string processing commands. These commands allow for the searching of strings, returning the location of characters within strings, and other processing functions. You will use the following XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<employees xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <Name>John Jones</Name>
   <Name>Mike Smith</Name>
   <Name>William Skakespeare</Name>
   <Name>Wally McClure</Name>
</employees>
```

You need to perform several operations on this XML code. These operations include getting the length of the name, parsing for the first name of the employee, and parsing for the last name of the employee. The following XSLT file will perform these operations:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/1999/xhtml">
   <xsl:template match="/">
      <table border='1'>
         <tr>
            <th>
               Employee Name:
            </th>
            <th>
               Length of Name:
            </th>
            <th>
               First Name:
            </th>
            <th>
               Last Name:
            </th>
            </tr>
            <xsl:apply-templates />
         </table>
      </xsl:template>
      <xsl:template match="Name">
         <tr>
            <td>
               <xsl:value-of select="." />
            </td>
            <td>
               <xsl:value-of select="string-length(.)" />
            </td>
            <td>
               <xsl:value-of select="substring-before(., ' ')"/>
            </td>
            <td>
               <xsl:value-of select="substring-after(., ' ')"/>
            </td>
         </tr>
      </xsl:template>
   </xsl:stylesheet>
```

The XML and XSLT file can be processed by using an XML control like this:

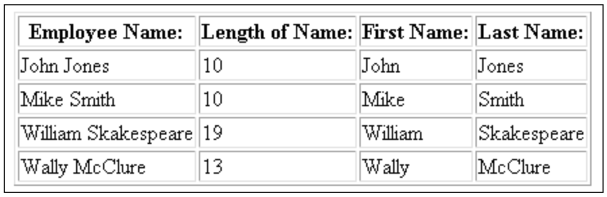
```
   <asp:Xml ID="XMLStringExample" runat="server" DocumentSource="XMLStringTest.xml"
               TransformSource="XMLStringTest.xslt">
```

In this example, a table is created. Each row contains four columns.

- The first column is the name of the employee. This is pulled straight from the `<name>` tag.
- The second column contains the length of the employee's name.
- The third column contains all of the employee names before the first space.
- The final column contains all of the employee names after the first space.

This example assumes that the format for the names is first name, space, and then last name.

Figure 5-5 displays the output of the example XML file being processed by the XSLT file.



Employee Name:	Length of Name:	First Name:	Last Name:
John Jones	10	John	Jones
Mike Smith	10	Mike	Smith
William Skakespeare	19	William	Skakespeare
Wally McClure	13	Wally	McClure

Figure 5-5

For a listing of the methods available for string processing in XSLT, review Appendix A.

Try It Out Numeric Processing

XSLT also possesses the ability to perform standard mathematical operations like many other programming languages. These operations can range from the simple addition and subtraction to ceiling, floor, and rounding.

Take a look at the following XML file:

```
<?xml version="1.0" encoding="utf-8" ?>
<numbers xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <a>9</a>
   <b>24.6</b>
   <c>-1</c>
   <d>4.3</d>
   <e>5</e>
</numbers>
```

You are going to process the numbers in this set and perform several operations on them — displaying data, performing a sum on the nodes, performing a modulo operation, and finally running through a conditional to output a string depending on whether or not the processed value is negative. The XSLT file that follows will process the preceding numeric XML file and output the numeric values.

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/1999/xhtml">
   <xsl:template match="/">
      <table border='1'>
         <tr>
            <th>
               a
            </th>
            <th>
               b
            </th>
            <th>
               c
            </th>
            <th>
               d
            </th>
            <th>
               e
            </th>
            <th>
               sum of numbers
            </th>
            <th>
               a mod e
            </th>
            <th>
               Is (a-b) negative or positive
            </th>
         </tr>
         <xsl:apply-templates />
         </table>
      </xsl:template>
      <xsl:template match="numbers">
         <tr>
            <td>
               <xsl:value-of select="a" />
            </td>
            <td>
               <xsl:value-of select="b" />
            </td>
            <td>
               <xsl:value-of select="c" />
            </td>
            <td>
               <xsl:value-of select="d" />
            </td>
            <td>
               <xsl:value-of select="e" />
            </td>
            <td>
               <xsl:value-of select="sum(*)"/>
            </td>
            <td>
               <xsl:value-of select="a mod e"/>
            </td>
            <td>
               <xsl:choose>
                  <xsl:when test="(a - b) &gt; 0">
                     positive
                  </xsl:when>
                  <xsl:when test="(a - b) &lt; 0">
                     negative
                  </xsl:when>
                  <xsl:otherwise>
                     0
                  </xsl:otherwise>
               </xsl:choose>
            </td>
         </tr>
         <br />
      </xsl:template>
   </xsl:stylesheet>
```

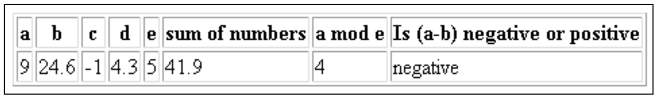
The XML and XSLT file may be processed by using an XML control like this:

```
<asp:Xml ID="XmlNumericExample" runat="server"
DocumentSource="XMLNumericExample.xml"
            TransformSource="XMLNumericExample.xslt">
```

Figure 5-6 shows the output of the numeric XML file after it has been processed by the numeric XSLT file.

- ❑ The first five columns display the values from the XSLT file.
- ❑ The sixth column displays the sum of the numbers.
- ❑ The seventh column displays a modulo e.

- The last column shows a combination of conditional tests, `<xsl:choose>` and `<xsl:when>` tags, as well as testing using less than and greater than commands.



a	b	c	d	e	sum of numbers	a mod e	Is (a-b) negative or positive
9	24.6	-1	4.3	5	41.9	4	negative

Figure 5-6

For more information on the processing functions built into XSLT, please refer to Appendix A on XSLT elements.

Writing Functions in XSLT

The XSLT processor in the .NET Framework and MSXML component support only the XSLT version 1 standard. This standard version makes it a little bit hard to write functions within XSLT. However, these components support the ability to write custom business logic. The ability to write these custom business objects is provided by extending XSLT, using traditional programming languages, such as VBScript, JavaScript, or any .NET-support language. Given the widespread acceptance of JavaScript, these examples use JavaScript to extend XSLT.

XSLT extensions are mostly specific to a given processor. For more comprehensive examples, please refer to the documentation provided by the XSLT processor that is being used.

XSLT 1.0 provides two types of extensions.

- **Extension elements**, which include such things as `xsl:transform`, `xsl:template`, and the like.
- **Extension functions**, which include `string`, `substring`, and the like.

XSLT 1.0 has a template base model. XSLT processors need information to distinguish between static content and extension elements. This can be accomplished by some commands that the processor recognizes. The code that follows shows an example:

```
<xsl:transform version"1.0" xmlns:xsl=http://www.w3.org/1999/XSL/Transform
xmlns:out=http://www.w3.org/1999/xhtml
xmlns:ext=http://exampleurl/extension
extension-element-prefixes="ext">
<xsl:template match="/">
   <out:html>
      <ext:ExampleFunction/>
   </out:html>
</xsl:template>
</xsl:transform>
```

This example shows the very basics of calling an external function in XSLT. The `xml` namespace for output is defined by the `xmlns:out` attribute and the extension's namespace is added through the `xmlns:ext` attribute. The output is formatted for HTML through the `<out:html>` tag, and a call is made to the external `ExampleFunction` through the `<ext:ExampleFunction/>` tag. This example calls out to an external function using the MSXML calling convention.

X Path

The XML Path Language (X Path) is a non-XML syntax for addressing parts of an XML document. X Path has been adopted by developers as a simple query language. X Path is a sequence of steps to get from one set of nodes to another set of nodes. The steps are separated by a slash (`/`) character. Each step is made up of the following parts:

- **Axis Specifier** — The Axis Specifier indicates a navigation direction. The axes available are:
    - child
    - attribute
    - descendant-or-self
    - parent
    - ancestor
    - ancestor-or-self
    - following
    - precending
    - following-sibling
    - self
    - namespace
- **Node Test**
- **Predicate**

In X Path:

- The root element is defined by `/*`.
- All elements are defined as `//*`.
- All top-level elements are defined as `/*/*`.

Therefore, an example X Path expression can look like `/CustomerOrder/Item`. This will select all of the nodes that are named `Item` and a child of `CustomerOrder`.

A more complex expression might be specified as: `/CustomerOrder/Item/following-sibling::*[1]`. This expression selects all elements that are below the `Item` node and indicates the `Item` node is a child of the `CustomerOrder` node.

In X Path, the expression `@` can be used to get at the attribute axis. For example, the expression `//Item[@price > 2]` selects the `Item` nodes that have an attribute of `price` that is greater than the value of 2.

Try It Out X Path Example

In this Try It Out, you take a look at X Path processing using one of the menu examples. In this example, you are going to search for the `<item>` tag of one of the `Menu2.xml` file's `menuitems`. Take a look at the example code in C# and ASP.NET:

```
System.Xml.XPath.XPathDocument document = new
System.Xml.XPath.XPathDocument(Server.MapPath("Menu2.xml"));
System.Xml.XPath.XPathNavigator navigator = document.CreateNavigator();
System.Xml.XPath.XPathNodeIterator nodes =
navigator.Select("//menu/menuitem[cost=.49]/item");
while (nodes.MoveNext())
{
   Response.Write(nodes.Current.Value);
}
```

In this code, the `item` tag of the `menutitem` will be returned if the cost of the item is set to `.49`. The code works by creating an X Path DOM document and loading it with the contents of the `Menu2.xml` file. The next step is to create the `XPathNavigator` object. The X Path expression is passed though the `XPathNavigator`'s `.Select()` method and returns a collection of nodes that can then be iterated through.

Now, take a look at the XML file that you are working with for this example:

```
<?xml version="1.0" encoding="UTF-8"?>
<menu>
   <menuitem>
      <item>Hamburger</item>
      <cost>1.99</cost>
   </menuitem>
   <menuitem>
      <item>Drink</item>
      <cost>.99</cost>
   </menuitem>
   <menuitem>
      <item>Fries</item>
      <cost>.49</cost>
   </menuitem>
</menu>
```

From the XML feed, you can see that the only `menuitem` that matches the cost of `.49` is the `Fries` item and that is the only result that is returned from the sample X Path code.

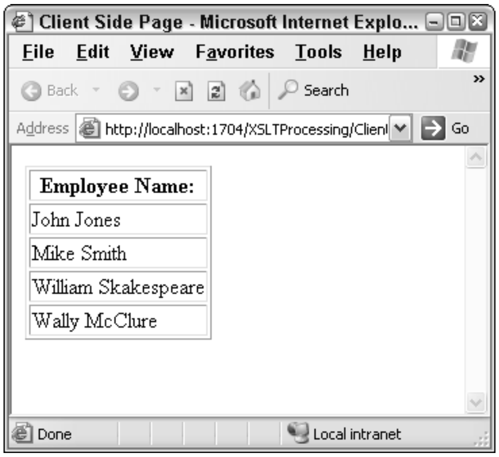
Integrating XML and Ajax

Now that we've covered XML, we come to the inevitable question — how do we integrate this with Ajax? It's actually a fairly simple process if your development team has its own library or needs to debug something that is occurring in an existing library.

Take a quick look at some code. In this example, you are using the Sarissa client-side library to perform the communication back to the server. For the callback to the server, you get a list of people in an XML format. This is put within an XML `DomDocument`. In this example, the code is loaded synchronously, but it could be loaded asynchonrously with a callback. You have created a string holding the XSLT commands. This string is put within an `XSLTProcessor()` object. The final steps are to use the XSLT object to transform the XML `DomDocument` and to then output the data to the browser.

```
<script language="javascript" src="SarissaLibrary.js">
<script language="javascript">
var xsltProc = new XSLTProcessor();
var xsltDoc = Sarissa.getDomDocument();
var xsltStr = "<?xml version=\"1.0\" encoding=\"UTF-8\" ?>" +
    "<xsl:stylesheet version=\"1.0\"
xmlns:xsl=\"http://www.w3.org/1999/XSL/Transform\"
xmlns=\"http://www.w3.org/1999/xhtml\">" +
   "<xsl:template match='/'>" +
      "<table border='1'>" +
         "<tr>" +
            "<th>" +
               "Employee Name:" +
            "</th>" +
         "</tr>" +
         "<xsl:apply-templates />" +
      "</table>" +
   "</xsl:template>" +
   "<xsl:template match=\"Name\">" +
      "<tr>" +
         "<td>" +
            "<xsl:value-of select=\".\" />" +
         "</td>" +
      "</tr>" +
   "</xsl:template>" +
    "</xsl:stylesheet>";
xsltDoc = (new DOMParser()).parseFromString(xsltStr, "text/xml");
xsltProc.importStylesheet(xsltDoc);
function SimpleExample() {
   var xmlDoc = Sarissa.getDomDocument();
   xmlDoc.async = false;
   xmlDoc.load("XMLStringTest.xml");
    var newDocument = xsltProc.transformToDocument(xmlDoc);
    document.write(Sarissa.serialize(newDocument));
}
SimpleExample();
```

Figure 5-7 shows the output of the preceding example.


Client Side Page - Microsoft Internet Explo...
File Edit View Favorites Tools Help
Back Search
Address http://localhost:1704/XSLTProcessing/Client
Go
Done
Local intranet

Employee Name:
John Jones
Mike Smith
William Skakespeare
Wally McClure

Figure 5-7

JSON

As indicated earlier in the chapter, JSON is the JavaScript Object Notation, and it is a lightweight data interchange format. JSON's chief advantage over XML is that the data may be parsed fairly easily using JavaScript's built-in `eval()` method. And although JSON has JavaScript in the name, it may actually be used by various languages.

Layout of JSON

JSON's usefulness is in the area of data interchange, and in some ways it is similar to XML. However, there are some key conceptual differences. Whereas XML is conceptually similar to working with databases and is designed to primarily work with sets of data, JSON is conceptually similar to arrays and collections in procedural programming languages. That means JSON is designed to be easily usable from within a procedural programming language.

JSON is built on the following data structures:

- **Name/value pairs** — This may be called an object, record, structure (struct), HashTable, keyed list, or associated array.
- **List of values** — This list of values is referred to an array in most programming languages.

Take a look at the specific layout of JSON in the following table.

Name	Description
object	`{}`
	`{ members }`
members	`string : value`
	`members , string : value`
array	`[]`
	`[ elements ]`
elements	`value`
	`elements , value`
value	`string, number, object, array, boolean, null`

With JSON, these items take on the following forms:

- An object is a set of name/value pairs. An object begins with a left brace (`{`) and ends with a right brace (`}`). Names are followed by a colon (`:`). Name/value pairs are separated by a comma (`,`).
- An array is a collection of values. Arrays start with a left bracket (`[`) and end with a right bracket (`]`). Values within an array are separated by a comma (`,`).

- A value may be one of several datatypes. If it is a string, it will be contained within double quotation marks. Other datatypes supported within an array are numbers, booleans, null, objects, and arrays.
- A string is a collection of Unicode characters wrapped within double quotation marks. Characters may be escaped by using the forward slash character (`/`).

JSON Example

Take a look at some data encoded in JSON:

```
{'Tables':[{
'Name':'Table1','Rows':[
{'tblStateId':1,'State':'Tennessee'},
{'tblStateId':2,'State':'Alabama'}]}],
'getTable':function(n){return _getTable(n,this);}}
```

This example text displays the textual representation of an ADO.NET dataset. In this JSON object, there is a table with two rows. There are two columns. These columns are `tblStateId` and `State`. Row 1 contains `tblStateId:1` and `State:Tennessee`. Row 2 contains `tblStateid:2` and `State:Alabama`.

Now you can look at the JavaScript code to actually use a JSON object. This JavaScript code runs in Internet Explorer and uses the Sarissa client-side JavaScript library. It will pull data from a local web server and then create an object based on a JSON object.

```
var xmlhttp = new XMLHttpRequest();
function clickme(){
   xmlhttp.onreadystatechange= myHandler
   xmlhttp.open("GET", "GetData.aspx", true);
   xmlhttp.send(null);
}
function myHandler() {
    if ( xmlhttp.readyState == 4 ) // READYSTATE_COMPLETE is 4
    {
    var strObj = xmlhttp.responseText;
    var obj;
    eval("obj=" + strObj);
    for(m in obj)
        alert(m);
    }
}
```

In this example, an `XMLHTTP` object is created and a request is sent to the web server. When the request comes back, an object is created from the returned data and the properties of that object that are available are displayed in a pop-up window to the user.

The example that is shown uses the Sarissa library. This library is discussed in Chapter 9.

Summary

In this chapter, you examined several topics related to data communication.

- **XML** — XML contains the raw data of a data transfer. It is human- and computer-readable in a language-independent format.
- **XSLT** — XSLT is used to transform information from one XML format into another.
- **X Path** — X Path is used to navigate within an XML/XSLT document.
- **JSON** — JSON is a human- and computer-readable data interchange format that is less complex then XML.

These topics form an important basis for heterogeneous communications, such as that used in Ajax and between a web client and a web server. Knowing how XML, XSLT, and JSON are structured can be very valuable as a basis of a custom Ajax library, for getting data between a web client and a web server, or for being able to diagnose a problem in an existing library.

You can find more on the topics discussed in this chapter at the following sites:

- **Wikipedia** — `http://en.wikipedia.org/wiki/XML`
- **"SGML Source" by Charles F. Goldfarb** — `www.sgmlsource.com`
- **World Wide Web Consortium (W3C)** — `www.w3c.org`
- **W3 Schools web site** — `www.w3schools.org`
- **JSON web site** — `www.crockford.com/JSON`