CHAPTER 20

Introducing XML

In This Chapter

Writing XML documents

Working with XML and ADO

Understanding the benefits of using XML

Unless you've been living on the moon for the last several years, you've undoubtedly heard of XML. It's been hailed as the future of the Web, the foundation for e-commerce and the universal information exchange medium. In reality, XML is merely a language used to describe information. In this chapter, I'll talk about how to create an XML document, how XML works with ADO, and why you might want to use XML in your applications.

Documenting Information

XML stands for Extensible Markup Language. It is part of the family of languages developed from the *Standardized General Purpose Markup Language* (SGML) that includes the various dialects of *HyperText Markup Language* (HTML). These languages are used to describe the structure of a document, but not the actual information contained in the document.

Tagging information

All SGML languages, including XML and HTML, are based on the concept of tags. A *tag* is formed by inserting a keyword inside a less than and greater than symbol pair (<>), such as `<HEADER1>`. Most tags work in pairs such as `<HEADER1>` and `</HEADER1>`, where the slash in front of the keyword is used to mark the end of that tag pair. In XML, this combination is known as an *element*. An element is used to mark the beginning and end of a piece of information. Sometimes the information is a single value, while at other times, it may be a collection of elements.

Elements may be nested and the information inside inherits the characteristics of all of the outer elements. In HTML, for instance, the tag `<I>` identifies a block of text that should be

displayed in italics, while the tag <B> identifies a block of text that should be displayed in bold. Thus, the following HTML statement will display the text in both bold and italics:

```
You may display text in <B>bold</B>, <I>italics</I>, and both
<B><I>bold and italics</I></B> by using different combinations
of tags.
```

Adding attributes

SGML also allows you to refine the meaning of a tag by including one or more attributes inside the tag. For instance, the <IMG> tag is used in HTML to mark the place where an image will be placed. Yet knowing that an image is to be placed in a document isn't sufficient unless you know which image is to be used. This information is specified with the SRC attribute. Thus, the following tag would display the image CJ&Sam.JPG in a Web page:

```
<IMG SRC="CJ&Sam.JPG">
```

Many tags support multiple attributes. The following tag not only specifies the name of the image to be displayed, but its height and width:

```
<IMG SRC="CJ&Sam.JPG" HEIGHT="640" WIDTH="480">
```

Grouping and formatting tags

Tags are grouped together in a document. With very few exceptions, how the tags are placed in the document doesn't matter, as long as the order of the tags remains constant. Thus, this set of tags

```
You may display text in <B>bold</B>, <I>italics</I>, and both
<B><I>bold and italics</I></B> by using different combinations
of tags.
```

and this set of tags

```
You may display text in
<B>bold</B>,
<I>italics</I>,
and both
<B><I>bold and italics</I></B>
by using different combinations of tags.
```

and even this set of tags

```
You may display text in
     <B>bold</B>,
     <I>italics</I>,
and both
     <B><I>bold and italics</I></B>
by using different combinations of tags.
```

have exactly the same meaning and will result in exactly the same display. This allows you to format your documents and make them easy to read without impacting the meaning of the information contained in the document.

Using XML tags

Unlike the HTML generic tags, which describe how information is to be formatted in a document, XML uses meaningful tags that describe the information they contain. A tag can refer to a single field or to a collection of fields, which correspond to a field name in a table, a database table, or a hierarchical view that is constructed from multiple database tables.

> Note
>
> **Helpful hierarchies:** Theoretically, it is possible to describe any combination of data in a hierarchy. In fact, some of the earliest databases, such as IBM's Information Management System (IMS), were based on a hierarchical data model. When extracting data from a database to send to another application, it is often useful to arrange it as a hierarchy. This eliminates redundant information and makes it easier for the receiving application to reformat the data to fit its own database structures.

A Simple XML document

Figure 20-1 shows a diagram of my family. It describes the family name (Freeze), the father (me), the mother (Jill), our children (Christopher and Samantha), and our pets (Kokomo, Pixel, Terry, Cali, Dusty and Raymond).

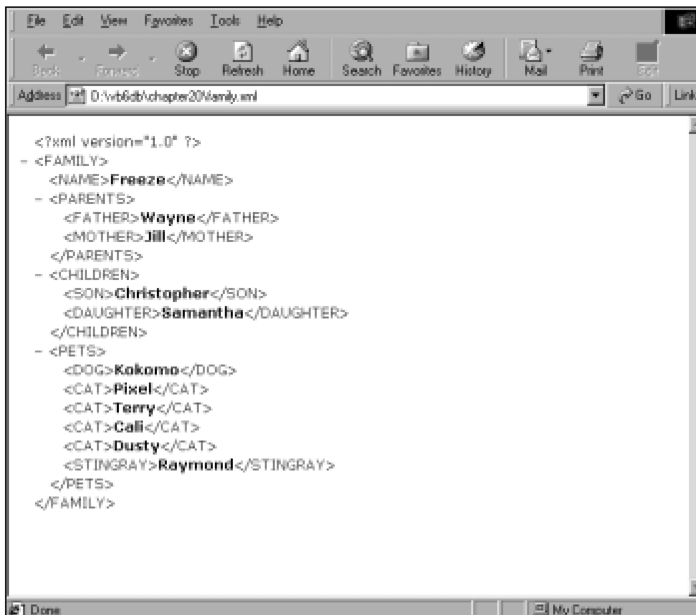


Figure 20-1: My family

Listing 20-1 contains the SQL statements that would create a database to hold this information. The database consists of four tables: one with the family name, one with information about the parents, another with the family's children, and a fourth with information about the family's pets.

Listing 20-1: A family database

```
Create Table Family (
   Name Char(32))

Create Table Parents (
   Name Char(32),
   Parent Char(32),
   Type Char(32))

Create Table Children (
   Name Char(32),
   Child Char(32),
   Type Char(32))

Create Table Pets (
   Name Char(32),
   Pet Char(32),
   Type Char(32))
```

You can also easily translate this information into an XML document (see Listing 20-2). Notice that the database tables had to flatten the information to fit into three distinct tables, while the XML document maintained the original hierarchical structure.

Listing 20-2: An XML document for the Freeze family

```
<?xml version="1.0"?>
<FAMILY>
   <NAME>Freeze</NAME>
   <PARENTS>
  <FATHER>Wayne</FATHER>
         <MOTHER>Jill</MOTHER>
   </PARENTS>
   <CHILDREN>
      <SON>Christopher</SON>
      <DAUGHTER>Samantha</DAUGHTER>
   </CHILDREN>
   <PETS>
      <DOG>Kokomo</DOG>
      <CAT>Pixel</CAT>
```

```
        <CAT>Terry</CAT>
        <CAT>Cali</CAT>
        <CAT>Dusty</CAT>
        <STINGRAY>Raymond</STINGRAY>
    </PETS>
</FAMILY>
```

You can find the FAMILY.XML document as \VB6DB\Chapter20\XML\FAMILY.XML on the CD-ROM.

The XML tags are used either to identify a single item, such as the name of the father or mother, or to identify a collection of tags that are logically grouped together, such as the children or pets.

Some of the latest generation of Web browsers, such as Internet Explorer 5.0, can display XML files directly (see Figure 20-2). If you look at Figure 20-2 carefully, you'll see a minus sign (-) in front of the <FAMILY>, <PARENTS>, <CHILDREN>, and <PETS> tags, in addition to the raw HTML. By clicking on the minus sign next to a tag, you can hide all of the tags below it in the hierarchy. The minus sign will change to a plus sign (+), which you can click on to show the tags again.

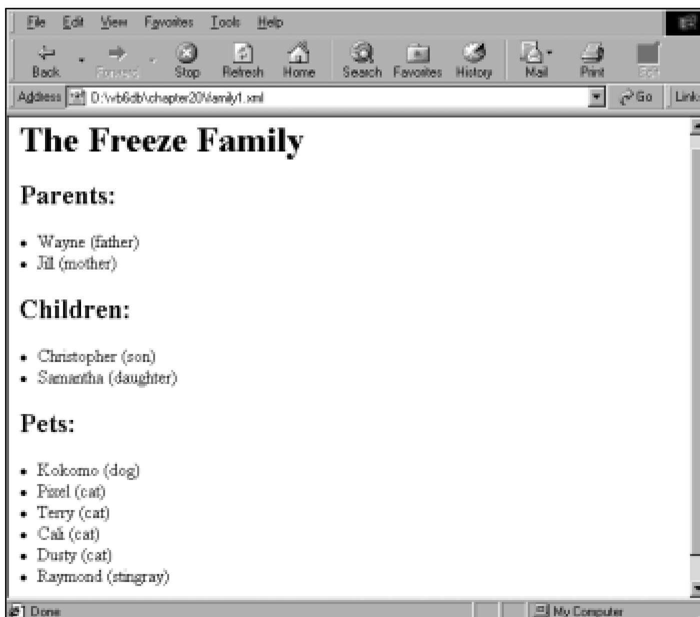


Figure 20-2: Viewing XML data in Internet Explorer 5.0

XML attributes

Another way to include information in an XML document is to use attributes. Like element names, you determine which attributes you want to include in the document. I've taken the XML document from Listing 20-2 and rewritten it to move the information about the type of family member to an attribute within an element, as shown in Listing 20-3.

Listing 20-3: An XML document for the Freeze family

```
<?xml version="1.0"?>
<FAMILY Name="Freeze">
<PARENT Type="Father">Wayne</PARENT>
   <PARENT Type="Mother">Jill</PARENT>
   <CHILD Type="Son">Christopher</CHILD>
   <CHILD Type="Daughter">Samantha</CHILD>
   <PET Type="Dog">Kokomo</PET>
   <PET Type="Cat">Pixel</PET>
   <PET Type="Cat">Terry</PET>
   <PET Type="Cat">Cali</PET>
   <PET Type="Cat">Dusty</PET>
   <PET Type="Stingray">Raymond</PET>
</FAMILY>
```

Using attributes lets you include more information in a single element. The downside is that it makes the document more complicated to read; it may also hide some hierarchy information.

For the most part, you should use attributes to hold information about your data. For example, the following element uses the attribute `Currency` to describe the type of currency used in the value for `LISTPRICE`. This is a good example of using attributes to clarify the information for a particular data value.

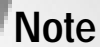
```
<LISTPRICE Currency="USD">29.99</LISTPRICE>
```

Tip

Extendable elements: If you're not sure whether to code a value as an element or an attribute, you should probably code the value as an element. This approach is more flexible, which is important if you plan to include new elements and attributes in the future.

Writing XML Documents

The rules for XML are fairly straightforward, and if you want, you can create your XML documents in any old text editor, including Notepad. However, for the most part, you should use an alternate way to load a file. Probably the best way is to write a program that generates the document for you. By doing this you can ensure that the document is generated properly. See Chapter 21 for one method you might use in Visual Basic.



Extensive XML: I highly recommend the *XML Bible* by Elliotte Rusty Harold, published by IDG Books. This well-written book goes into all of the details you should know before you build an XML-based application.

Creating an XML document

At the top of an XML document is a header tag, which describes information about the document (see Listing 20-4). Typically, all you will code is the version level, though the XML standard defines several other attributes that you may want to include.

Listing 20-4: A minimal XML document

```
<?xml version="1.0"?>
<root_element_tag>
</root_element_tag>
```

Following the header tag is the root level element. Every XML document is a hierarchy, with one and only one root element. In Listing 20-4, the root level element is `<root_element_tag>` and `</root_element_tag>`, while in Listing 20-2 the root level element is `<FAMILY>` and `</FAMILY>`.

Identifying XML elements

Unlike HTML, you must create the XML elements you use in your document. For instance, I created the following element to identify my son:

```
<SON>Christopher</SON>
```

The tag <SON> marks the beginning of a block of data, while the tag </SON> marks the end of the tag. You can nest pairs of tags within each other, like this:

```
<CHILDREN>
   <SON>Christopher</SON>
   <DAUGHTER>Samantha</DAUGHTER>
</CHILDREN>
```

The <CHILDREN> tag marks the collection of elements that comprises the children in my family. Elements can and should be nested to describe groups of data. A group of data might be something as simple as identifying a record, a collection of records, or a table.

The following element

```
<CHILDREN></CHILDREN>
```

can be written as

```
<CHILDREN/>
```

In this case, it means that there is no information associated with that particular element.

In order to be considered a well-formed XML document, the start and stop tags must not overlap. They need to be closed in the reverse order in which they were opened. The following statement is legal in HTML, but will cause an error in XML:

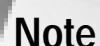
```
<B><I>Hi Jill</B></I>
```

Thus, you need to rewrite the statement as follows:

```
<B><I>Hi Jill</I></B>
```

Creating XSL Style Sheets

The Extensible Style Sheet Language (XSL) is used to format an XML document. An XSL Style Sheet is similar in concept to a Cascading Style Sheet for an HTML document. Both tools allow you to create a template that can be used to provide a common formatting to a document.



Note

And that ain't all: This is an overly abbreviated section that describes what you can do with just a few XSL statements. XSL is a very rich language, which can be used to format some very complex documents.

The basic approach used in the XSL is that the first element in the document applies to the root element of the XML document. The rest of the elements on the XML document are formatted recursively from the root element. All XSL elements have a pre-

fix of `xsl:`.XSL is derived from SMGL and XML, so it is a tag-oriented language that has to follow the same basic rules used for any XML document. Unlike some other languages, XSL is case sensitive, so all tags must be entered using lowercase characters. Figure 20-3 shows the same FAMILY.XML document from Listing 20-2, but it uses the style codes from the XSL document shown in Listing 20-5.

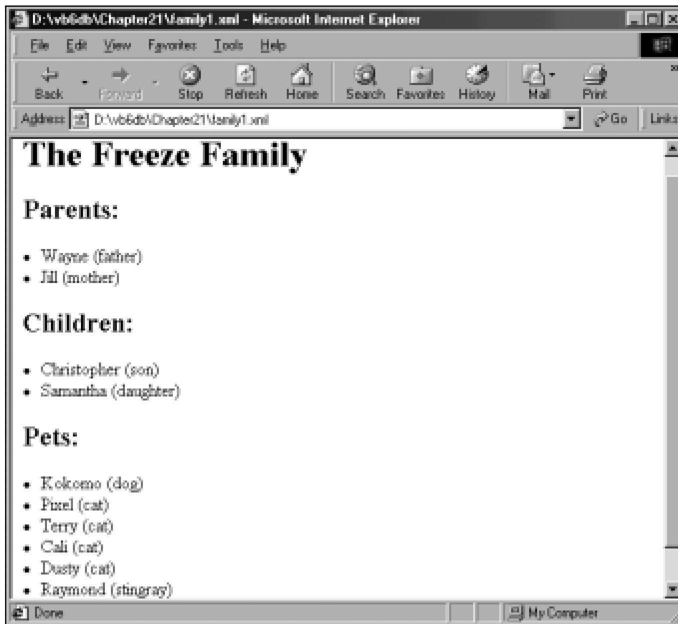


Figure 20-3: Formatting an XML document with an XSL style sheet

Listing 20-5: **An XSL style sheet for the FAMILY.XML document**

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<xsl:template match="/">
     <html>
     <head>
     </head>
     <body>
     <xsl:apply-templates/>
     </body>
     </html>
```

Continued

Listing 20-5 *(continued)*

```
</xsl:template>

<xsl:template match="FAMILY">
     <h1>
     The
     <xsl:value-of select="NAME"/>
     Family
     </h1>
     <xsl:apply-templates/>
</xsl:template>

<xsl:template match="PARENTS">
     <p>
     <h2>Parents:</h2>
     <xsl:apply-templates/>
     </p>
</xsl:template>

<xsl:template match="FATHER">
     <li>
     <xsl:value-of select="."/>
     (father)
     </li>
</xsl:template>

<xsl:template match="MOTHER">
     <li>
     <xsl:value-of select="."/>
     (mother)
     </li>
</xsl:template>

<xsl:template match="CHILDREN">
     <p>
     <h2>Children:</h2>
     <xsl:apply-templates/>
     </p>
</xsl:template>

<xsl:template match="SON">
     <li>
     <xsl:value-of select="."/>
     (son)
     </li>
</xsl:template>

<xsl:template match="DAUGHTER">
     <li>
```

```
        <xsl:value-of select="."/>
        (daughter)
        </li>
   </xsl:template>

   <xsl:template match="PETS">
        <p>
        <h2>Pets:</h2>
        <xsl:apply-templates/>
        </p>
   </xsl:template>

   <xsl:template match="DOG">
        <li>
        <xsl:value-of select="."/>
        (dog)
        </li>
   </xsl:template>

   <xsl:template match="CAT">
        <li>
        <xsl:value-of select="."/>
        (cat)
        </li>
   </xsl:template>

   <xsl:template match="STINGRAY">
        <li>
        <xsl:value-of select="."/>
        (stingray)
        </li>
   </xsl:template>

   </xsl:stylesheet>
```

The XSL style sheet is found on the CD-ROM in the `\VB6DB\CHAPTER20\FAMILY.XSL` file. I've also included a modified version of the FAMILY.XML file, called FAMILY1.XML, that will use this style sheet to format the information.

The xsl:stylesheet element

The `xsl:stylesheet` element is the root element of an XSL style sheet. The primary attribute to this element is the `xlsnm:xsl`, which is used to specify the name space for the document. The name space identifies all of the elements and their attributes. This is a URL reference to a Web site like the one below. However, most XML vendors only verify the name; they don't actually look up the document on the Web.

```
http://www.w3.org/TR/WD-xsl
```

Name spaces are used in XML to help clarify a reference. For instance, if you have a tag called name, it may have many different meanings depending on the context in which it is used. By adding the name space and a colon in front of the element, you can clarify which name space the element was used from. In fact, the `xsl:` in front of the `xsl:stylesheet` element identifies the element as belonging to the xsl name space.

The xsl:template element

The `xsl:template` element is the fundamental element to control how a particular part of your XML document is formatted. Consider the following fragment from the FAMILY.XSL document:

```
<xsl:template match="STINGRAY">
    <li>
    <xsl:value-of select="."/>
    (stingray)
    </li>
</xsl:template>
```

This code will process the following `STINGRAY` element from the XML document:

```
<STINGRAY>Raymond</STINGRAY>
```

The `match` attribute identifies the XML element that will be processed by this XSL element. Note that there are many other ways to associate the XSL template with an XML element. The `match` attribute is just one of the easier ones to use.

Once the particular template element has been defined, the information inside the element will be displayed and any XSL elements will be executed. In this case, the following information will be output:

```
<li>Raymond (stingray)</li>
```

Where the text `<li>` comes directly from this element, the word `Raymond` will be generated by the `<xsl:value-of>` element, followed by the text `(stingray)` and `</li>`. There is no reason that I couldn't have included other HTML elements, such as an `IMG` element, to display a picture of a stingray or a hyperlink to another document. Also, you should note that the information is processed in order of how it is listed. Thus, you can perform tasks before and after any other XSL tags you wish to use.

Note

Office-oriented: The `xsl:template` element is similar to the concept of Style, used in Microsoft Office. In a typical Word document for example, you may use specific styles for headers, body text, tables, etc. . If you want to change the font used in a header, all you have to do is update the style with the new font. Then all of the headers in the document will automatically be updated. Without styles, you would have to manually update each individual header.

The xsl:value-of element

The xsl:value-of element returns the value of an element. If you use the select attribute and specify a period as the value, then the value of the current element will be returned. Thus, for this XSL template

```
<xsl:template match="DAUGHTER">
    <li>
    <xsl:value-of select="."/>
    (daughter)
    </li>
</xsl:template>
```

and this XML element

```
   <DAUGHTER>Samantha</DAUGHTER>
```

the <xsl:value- of select="."/> will return the value Samantha.

The xsl:apply-templates element

The xsl:apply-templates element processes templates for all of the elements within the current element. In the following XSL template

```
<xsl:template match="PARENTS">
    <p>
    <h2>Parents:</h2>
    <xsl:apply-templates/>
    </p>
</xsl:template>
```

an HTML header would be created containing the word Parents, and then the FATHER and MOTHER elements beneath the PARENTS tag would be processed from the XML document fragment shown below:

```
   <PARENTS>
    <FATHER>Wayne</FATHER>
      <MOTHER>Jill</MOTHER>
   </PARENTS>
```

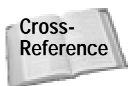
Any information output by processing the templates associated with these elements would follow the information output before the <xsl-apply-templates> element was reached in the XSL template. After all of the templates have been processed, the </p> tag would be output.c

Other XML tools

There are a few other tools that you should be aware of when creating an XML document. These tools help ensure that your XML documents are created properly, as well as help other users understand your document.

XML parsers

One tool that you'll find valuable is an XML parser. Given the nature of the XML language, it can be a real pain for you to write code to convert the XML elements into something a bit more meaningful, much less verify that the XML document you received is well formed.



Microsoft has released an XML parser called MSXML that you can call from your program. I'll talk about this parser in more detail in Chapter 21.

Document Type Definitions

Another feature of XML is the ability to create a set of rules that governs how an XML document is created. These rules are known as *Document Type Definitions,* or DTD. This information can be useful when creating XML documents, since it ensures that you can't create an invalid document. You don't have to include a DTD with your XML document, and in practice, many tools, such as Internet Explorer 5.0, don't bother to use it even if you do include it.

XLinks and XPointers

The *Extensible Linking Language* (XLL) helps you locate XML resources outside the local document. XLL has two main parts: XLinks and XPointers. XLL is similar in concept to an HTML link. An XPointer is a way to identify a location in an XML document, while an XLink uses a URL, and perhaps an Xpointer, to locate a section of a document.

Working with XML and ADO

ADO has the ability to save information from a `Recordset` object into an XML file. This makes it easy to create ADO documents that can be sent to other applications. But while the file is formatted according to XML rules, there are a few unique characteristics that you should understand.

Creating an XML File with ADO

Consider the following **Select** statement:

```
Select CustomerId, Name
From Customers
Where State = 'MD'
```

You can easily use it to populate a `Recordset` with data from the sample database and save the results to an XML file with the statements in Listing 20-6.

Listing 20-6: The Command1_Click event in SaveXML

```
Private Sub Command1_Click()

Dim db As ADODB.Connection
Dim rs As ADODB.Recordset

Set db = New ADODB.Connection
db.Open "provider=sqloledb;data source=Athena;" & _
      "initial catalog=VB6DB", "sa", ""

Set rs = New ADODB.Recordset
rs.Open "Select CustomerId, Name From Customers " & _
      "Where State='MD'", db, adOpenForwardOnly, adLockReadOnly

rs.Save App.Path & "\results.xml", adPersistXML

rs.Close

db.Close

End Sub
```

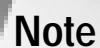
The SaveXML program and a copy of the `Results.XML` file can be found on the CD-ROM in the `\VB6DB\Chapter20\SaveXML` directory.

Looking at the XML file

The XML file will contain all of the information necessary to reconstruct the `Recordset`, including a description of each `Recordset` and each row of information retrieved from the database (see Listing 20-7).

Listing 20-7: A sample XML file created by ADO

```
<xml xmlns:s='uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882'
   xmlns:dt='uuid:C2F41010-65B3-11d1-A29F-00AA00C14882'
   xmlns:rs='urn:schemas-microsoft-com:rowset'
   xmlns:z='#RowsetSchema'>

<s:Schema id='RowsetSchema'>

   <s:ElementType name='row' content='eltOnly'
         rs:CommandTimeout='30'>

      <s:AttributeType name='CustomerId' rs:number='1'
         s:writeunknown='true'>

         <s:datatype dt:type='int' dt:maxLength='4'
               rs:precision='10'
            rs:fixedlength='true' rs:maybenull='false'/>

      </s:AttributeType>

      <s:AttributeType name='Name' rs:number='2'
         rs:nullable='true' rs:writeunknown='true'>

         <s:datatype dt:type='string' rs:dbtype='str'
            dt:maxLength='64'/>

      </s:AttributeType>

      <s:extends type='rs:rowbase'/>

   </s:ElementType>

</s:Schema>

<rs:data>

   <z:row CustomerId='84' Name='Fred Price'/>
   <z:row CustomerId='205' Name='Joseph Bell'/>
   <z:row CustomerId='385' Name='Kali Carlisle'/>

</rs:data>

</xml>
```



It looks different on disk: While the XML file generated by ADO is human readable, it isn't very printer friendly. Therefore, I've restructured how the elements are displayed in Listing 20-5, without changing any of the content.

The root element of the XML document is the `xml` element. It is used to define the name spaces that are used within the document. Four name spaces are typically defined:

- ✦ `s` defines the schema
- ✦ `dt` defines data types
- ✦ `rs` defines `Recordset` information
- ✦ `z` contains row information

The XML file is broken into two main elements: the schema element and the recordset element. The schema element is denoted by the tag `<s:schema>`, while the recordset element is denoted by the `<rs:data>` tag.

In the `s:schema` element, global recordset information is defined in the `s:ElementType` element. Within this element, information about each of the columns is defined. The `s:AtrributeType` element is used to define the column name that will be used in the recordset's `z:row` element, along with its data type.

Then each row of data in the recordset is listed in the `rs:data` element. This is the most understandable element in the file. Each row in the recordset is identified with the `z:row` element name. Within the row, each column is listed as an attribute, with its corresponding data value.

Understanding the Benefits of Using XML

So, now you know that XML is a way that you can intelligently describe data. But why would you want to use it in your applications? In short, not all applications will benefit from XML, but many will.

Data interchange

XML is based on the concept that a document is the best way to exchange information between organizations. All kinds of documents are used in a business: purchase orders, invoices, contracts, product specifications, and so on. A document created by one organization must then be read and understood by another. Documents contain two parts: the framework for presenting the information and the information itself.

Consider a product catalog. It consists of a number of individual specification sheets. For a group of similar products, each of the categories in the specification sheet is the same, inviting the reader to make apples-to-apples comparisons between products.

There are many different ways to exchange data between applications, each with its limitations and problems. In order to understand why you would use XML, you should understand the more traditional methods of data interchange.

Binary files

Binary files typically contain a raw dump of the data from your application. No information about the data structure of the file is contained in the file, and usually a custom program needs to be written to read the information from the file and reformat it into something that the receiving application can use.

Once the file is created, it is transported to another computer via a network, floppy disk, or some other method, then loaded into the second computer. Any errors or problems encountered on the remote computer are usually returned back to the first computer, using the same technique that was used to send the data in the first place. However, this technique means that there is a delay between the time the data is created in the sending application and the time before the receiving application has posted the changes. In many cases this doesn't matter, but in some cases it can be a big problem.

There are other drawbacks to using binary files. Different computers use different ways of storing numeric data. Intel computers, for instance, store a 16-bit integer low byte then high byte, while other types of computers may store the same value high byte then low byte. This means that you may have to transform the actual values in order to accurately process the file.

Text files

Text files are also known as *flat files.* These files usually contain a formatted dump of data from the sending application. When dealing with mainframe-based applications, these files typically use a fixed data format, where each field occupies the same column positions in each line of the file.

PC-based applications typically use delimited files, such as *Comma Separated Value* (CSV) files or tab delimited files, where each field in a line is separated from the next field by a special character, such as a comma or tab. Each record in the file is separated from the next by a carriage return, line feed, or carriage return line feed pair.

Like binary files, text files usually don't contain information about the data itself. Sometimes the first line of the file will contain the names of the fields, but other information, such as data type and size information, is almost never included. Also, like binary files, text files need to be transported to the remote computer either through a network or through removable media, such as a floppy disk or magnetic tape.

COM components

Using COM to exchange data is merely a matter of building a COM component in the receiving application that can be called by the sending application as the data is created. This makes is easier to send the data, since the receiving application is

able to process the data as it is received. There is no problem parsing the various fields that are being transported, since each field is stored in its own property with its own specific data type.

One problem encountered when using COM for data interchange revolves around the real-time nature of COM. As the data is generated in the sending application, the receiving application must be available to process it. If necessary, you can use a message queue between the COM components that allows the two applications to process data at their own speed. This prevents the sending application from transmitting data faster than the receiving application can process it.

You don't have to worry about byte order or any other compatibility issues like that because the COM specification ensures that those problems can't arise. Of course, using COM components means that you have to run in a Windows environment. While there are techniques that allow you to run COM on non-Windows systems, they generally aren't as stable as using COM natively.

Separating content from formatting

It is possible to build generic Web pages that load and display the information found in an XML document, however it is often desirable to create a program that reads the HTML document and extracts meaningful information from the document. By using an XML document, you can easily extract the information you need.

Better searches

It is possible to perform better searches of an XML file when compared to an unstructured file. You can use the tags to ignore data you don't want to search. For instance, if you have an XML file containing a list of books, you can search only the author tags if you're looking for a book written by a particular author. While this may not make a big difference when searching for someone named Elizabeth Thornberry, it may help to eliminate false hits when searching for titles written by Red Book.

Local access

For all practical purposes, you could use an XML document as a primitive database. All of the information needed to describe the data is kept within the document, so it would be easy to retrieve data from a remote database server and save it locally in an XML file which could be used as input to various tools, such as a report writer or statistical analysis program. Because you have a local copy of the data, you don't impact the database server's performance when you process the data as you try various report formats or create complex statistical analysis programs.

Easily compressed

Because XML tags and data are stored using normal common ASCII characters, the data is easily compressed. Thus tools, such as a dial-up modem that automatically includes data compression helps to compensate for the fact that XML documents

are much larger than a straight binary file. Note that even with compression, the XML document will most likely still be larger than the equivalent binary file, but using data compression goes a long way toward eliminating the extra overhead.

Vendor independence

XML is independent of any particular vendor and has the benefit of being incorporated into many different products. Thus, it is easily incorporated into applications. Unlike comma separated value files that have to be parsed, there are many different XML parsers available to help you read an XML document. So as long as you can agree on the elements in the XML document, it doesn't matter whether a COBOL program running on an IBM mainframe or a Visual Basic program running on Windows 2000 processes the XML document.

Industry acceptance

Many industry-specific groups are being formed to determine the elements and organization for document exchange. This is very important, since it will allow you to exchange information with other applications and not worry if, for example, someone chooses to call someone's name `FIRST_NAME`, `FIRSTNAME` or `FNAME`.

There are a number of industry groups currently working on standardizing XML tags so that organizations can exchange documents with the knowledge that they will understand the structures used in the documents.

Thoughts on Using XML

Not every application will benefit from XML. Specifically, adding XML support to closed applications (for example, those applications that don't share data with other applications) is probably a waste of time and money. There are better ways to share data within an application, such as a database or a COM object. However, if you have an open application that needs to exchange data with other applications, XML may be the solution for you.

It's my opinion that XML is one of the most over-hyped technologies in the market today. Every vendor is scrambling to demonstrate their commitment to XML and claim that their products are XML-enabled. In reality, XML is only a tool for facilitating data exchange. The real power of XML comes from the fact that many industries are defining XML-based standards for information interchange. Having a standard that describes how to submit a purchase order for automotive parts or return information about your checkbook is what is really important.

Summary

In this chapter you learned:

✦ how XML documents are formatted.

✦ about XML tags and attributes.

✦ how to translate a database design into an XML document.

✦ how to create an XML document with an editor.

✦ how to view an XML document with Internet Explorer 5.

✦ about XSL style sheets.

✦ how ADO saves recordsets in XML format.

✦ about the benefits of using XML