CHAPTER 22

Integrating XML with Internet Information Server Applications

In This Chapter

Requesting information using the Document Object Model

Building a simple Web page

Updating customer information

In this chapter, I'll show you how to use the Document Object Model by building a Web page using VBScript. That Web page will communicate with a Web server application built with Internet Information Server (IIS) Applications.

Requesting Information

How often have you wanted to import a particular piece of information via the Web into your program for analysis? Perhaps you're interested in getting a stock quote on a periodic rate or following mortgage rates? Maybe you want to download information about how well the Orioles are playing.

By defining an XML document for requesting information and another document to contain the response, you can build a new breed of server that responds to XML requests for information. The fact that you can leverage existing HTTP technologies, such as Web servers and Web development tools, makes it easier to build these applications.

Getting Customer Information With XML

In this chapter, I'm going to focus on how to build an XML client program that requests information from an XML Server program. I've decided to build a Web page using a little VBScript as the client and an IIS Application as the server (see Figure 22-1). This application supports two basic types of requests: retrieving information about a customer and updating information about a customer.

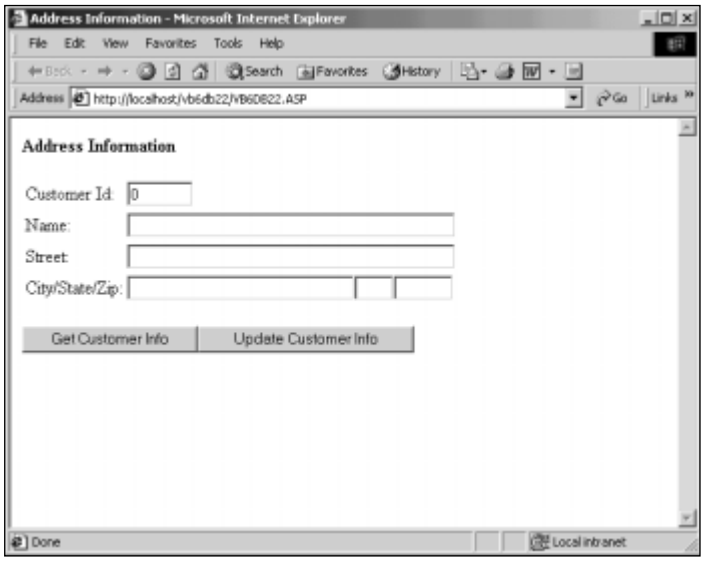

Figure 22-1: Running the XML Server application

Building the Simple Web Page

The sample Web page shown in Figure 22-1 is a fairly simple Web page that uses a table to line up the captions and the fields I use to display the data, as you can see in Listing 22-1. It is broken into three main sections: the `<head>`, the `<body>`, and the `<script>`. While I've left the tags for `<script>` in Listing 22-1, I omitted the code, since I'll discuss it later in this section.

Listing 22-1: **HTML for the XML client Web page**

```
<html>
   <head>
      <title>Address Information</title>
   </head>
```

```
    <body>
       <strong>Address Information</strong>
       <form align="left" name="AddressInfo">
          <table border = "0">
             <tr>
                <td>Customer Id:</td>
                <td><input type="text" name="CustomerId" size="6"
                   value="0"></td>
             </tr>
             <tr>
                <td>Name:</td>
                <td><input type="text" name="Name" size="45"
                      value=""></td>
             </tr>
             <tr>
                <td>Street:</td>
                <td><input type="text" name="Street" size="45"
                      value=""></td>
             </tr>
             <tr>
                <td>City/State/Zip:</td>
                <td>
                   <input type="text" name="City" size="30" value="">
                   <input type="text" name="State" size="2" value="">
                   <input type="text" name="Zip" size="5" value="">
                </td>
             </tr>
          </table>
       </form>
       <button onClick="GetCustomerInfo()">Get Customer Info</button>
       <button onClick="UpdateCustomerInfo()">Update Customer
             Info</button>
    </body>
    <script language="VBScript">

    </script>
 </html>
```

Note that the form declaration differs from most Web pages that use forms. In this case, I don't need attributes that describe how to send the form data to the server. Specifically, I didn't code the `action` and `method` attributes. This is because I simply don't need them. The routines that will handle the conversion to XML will also handle the interactions with the Web server.

At the bottom of the form, I declared buttons that will call the `GetCustomerInfo` and `UpdateCustomerInfo` VBScript routines. This is where the actual work of converting the information from the form into an XML document and sending it to the server for processing takes place.

Requesting Customer Information

Retrieving customer information involves determining how the request and response XML documents should look and then building the code to process the documents.

Defining the XML documents

The `GetCustomerInfo` script routine takes the `CustomerId` field from the form on the Web page and assembles the XML document shown in Listing 22-2. This document defines the `GETCUSTOMERINFO` element to identify the request. Within the `GETCUSTOMERINFO` element are one or more `CUSTOMER` elements with the `CustomerId` attribute coded. This attribute specifies the customer you want to retrieve.

Listing 22-2: A sample request for customer information

```
<?xml version="1.0"?>
<GETCUSTOMERINFO>
   <CUSTOMER CustomerId="0"/>
</GETCUSTOMERINFO>
```

Listng 22-3 shows how the server should respond to the request. I use the same basic document that was used to request the customer's information, but I expand the `CUSTOMER` element to include elements for the Name, Street, City, State, and Zip fields from the Customers table. I also include another attribute called `Get`, which indicates the status of the request. A value of OK means that the information was retrieved properly. Otherwise, `Get` will contain an error message.

Listing 22-3: A sample response to the request for customer information

```
<GETCUSTOMERINFO>
   <CUSTOMER CustomerId="0" Get="OK">
```

```
      <NAME>
         Dexter Valentine
      </NAME>
      <STREET>
         3250 Second Ave.
      </STREET>
      <CITY>
         San Francisco
      </CITY>
      <STATE>
         CA
      </STATE>
      <ZIP>
         94115
      </ZIP>
   </CUSTOMER>
</GETCUSTOMERINFO>
```

Requesting a customer

Pressing the Get Customer Info button on the Web page will trigger the `GetCustomerInfo` VBScript routine in the `<script>` section of the HTML document (see Listing 22-4). This routine performs three separate tasks. First, it must create an XML document similar to the one shown in Listing 22-2. Next, it must take the document and transmit it to the Web server. Finally, it must take the response document (see Listing 22-3) from the Web server and fill in the various fields on the form.

Listing 22-4: The GetCustomerInfo routine in XML Client

```
Sub GetCustomerInfo()

Set XMLReq = CreateObject("MSXML2.DOMDocument")
Set p = XMLReq.createProcessingInstruction("xml", _
      "version=""1.0""")
XMLReq.appendChild p

Set node = XMLReq.createElement("GETCUSTOMERINFO")
Set subnode = XMLReq.createElement("CUSTOMER")
subnode.setAttribute "CustomerId", _
      Document.AddressInfo.CustomerId.Value

node.appendChild subnode
```

Continued

Listing 22-4 *(continued)*

```
XMLReq.appendChild node

MsgBox XMLReq.xml

set http=CreateObject("MSXML2.XMLHTTP")
http.open "Post", _
      "http://athena/VB6DB22/VB6DB22.ASP?wci=GetCustomer",
false
http.setRequestHeader "Content-Type", "text/xml"
http.send XMLReq

Set XMLResp = CreateObject("MSXML2.DOMDocument")
XMLResp.LoadXML http.responsetext

MsgBox XMLResp.xml

Set nl = XMLResp.getElementsByTagName("CUSTOMER")
i = 0
Do While (i < nl.length) And (nl(i).getAttribute("CustomerId")
<> _
       Document.AddressInfo.CustomerId.Value)
  i = i + 1
Loop

If i < nl.length Then
   If nl(i).getAttribute("Get") = "OK" Then
      Set nx = nl(i).getElementsByTagName("NAME")
      Document.AddressInfo.Name.Value = nx(0).text

      Set nx = nl(i).getElementsByTagName("STREET")
      Document.AddressInfo.Street.Value = nx(0).text

      Set nx = nl(i).getElementsByTagName("CITY")
      Document.AddressInfo.City.Value = nx(0).text

      Set nx = nl(i).getElementsByTagName("STATE")
      Document.AddressInfo.State.Value = nx(0).text

      Set nx = nl(i).getElementsByTagName("ZIP")
      Document.AddressInfo.Zip.Value = nx(0).text

   Else
      MsgBox "The customer wasn't found: " & _
         nl(i).getAttribute("Get")

Else
   MsgBox "The customer wasn't found."
```

```
End If

End Sub
```

The format's changed to protect the guilty: I admit it. I've reformatted all of the HTML and script code from the actual documents to make them more readable. However, changing the formatting does not change how the Web page works.

Building the request document

The `GetCustomerInfo` routine begins by creating a `DOMDocument` object called `XMLReq` to hold the XML document I want to send to the server. Note that I can't use the normal `Dim` and `Set` statements to create the object, because VBScript can't reference the object libraries directly from code. The only way to create an object in VBScript is to use the `CreateObject` function.

After creating the base document, I add the `<?xml version="1.0"?>` element by using the `createProcessingInstruction` and `appendChild` methods. While this isn't absolutely necessary, since the MSXML parser is smart enough to figure out how your document is structured without it, it is good form to include this element in case you choose to use a different XML server in the future.

Once the XML document is initialized, I create the `GETCUSTOMERINFO` element that really defines this document by using the `createElement` method. This returns an object reference to an `XMLDOMElement` object, which I save in the variable called `node`. Then I create another `XMLDOMElement` object for `CUSTOMER` in the variable `subnode`. I use the `setAttribute` method to create the `CustomerId` attribute with the value from the `CustomerId` field in the form. Then I connect the `subnode` object to the `node` object by using the `node.appendChild` method. Next, I use the `XMLReq.appendChild` method to link the `node` object to the root document.

I should point out that the order in which I append the processing and element instructions to the root object is important. All of the objects stored below a particular hierarchy are stored in the order where they were inserted. Thus, if you want element A to be displayed before element B when the XML document is generated, you must append element A before you append element B. Since I want the `GETCUSTOMERINFO` element to follow the processing instruction element, I have to append the processing instruction first.

After creating the document, I use the `MsgBox` statement (see Figure 22-2) and the `XMLReq.xml` method to display the document to the user. While this wouldn't be included in a production version of this application, it allows the programmer to see the XML request before it is sent.

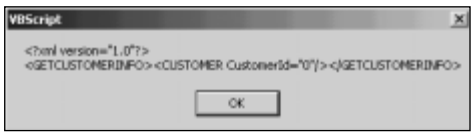

Figure 22-2: Viewing the XML GETCUSTOMERINFO request document

Sending the request document

In the next section of the routine, I create an `XMLHttp` object called `http` to perform the actual data transfer. After creating `http`, I use the `open` method to establish an HTTP connection to the Web server. I specify that I want to perform an HTTP Post operation to send the document, and I include the URL of the program that will process the request. Finally, I choose not to do an asynchronous transfer. This means that the `send` method won't return until a response has been received from the Web server. This approach simplifies the programming involved, since I don't have to enable the `onreadystatechange` event to determine when the response document has been received.

Before I send the document, I use the `setRequestHeader` method to set the Content-Type HTTP header explicitly to `text/xml`. While this isn't important in this application, since both sides are expecting XML documents to be transferred, it may be important in other situations where different processing paths may be taken depending on the document type.

When the code reaches the `send` method, a warning message may be displayed to the user letting them know that the Web page is accessing external information (see Figure 22-3). You can configure the Web browser to allow programs to disable this error message by changing the security level to low for the particular zone that you are accessing.

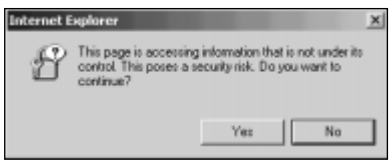

Figure 22-3: Getting permission to end the XML document

Do I really want to do this?: Changing the security level in your browser to allow you to use the `send` method in your Web page allows any Web page in the same content zone to use this function. Before you change this option, be sure you really want to take this security risk.

After using the `send` method to transmit the document, I create a new `DOMDocument` object that will hold the response from the Web server. Then I use the `LoadXML`

method to create the document from the `http.responsetext` property and then use the `MsgBox` statement to display the response document to the user (see Figure 22-4).

Figure 22-4: Viewing the GETCUSTOMERINFO response document

Displaying the response document

Displaying the information is merely a matter of working your way through the response from the Web server and extracting the information you want to display. This is easier said than done, however. You need to traverse the document hierarchy to find the `CUSTOMER` element that matches the CustomerId value from the form. Then you need to determine if the request was successful. Once this is done, you can take the information associated with the request and update the form.

In this case, I begin by creating a `nodeList` object that contains all of the `CUSTOMER` elements using the `getElementsByTagName` method. Since it is possible that the `nodeList` object may have more than one `CUSTOMER` element, I'll set the variable `i` to zero and use a `Do While` loop to check each of the elements to find the first one that matches the CustomerId value from the form.

When the loop finishes, the variable `i` will either point to the proper element or it will contain a value that is one larger than the number of elements in the `nodeList` object. (Remember that the `nodeList` object is a zero-based collection, so if it contains only one element, the element will have an index value of zero while the collection has a `length` of one.)

Next, I check the value of `i` to see if it is less than the length of the collection and issue the appropriate message if it isn't. Then I can see if the value of the `Get` attribute is `OK`. If it isn't, I need to issue the appropriate error message.

If everything worked correctly, I can retrieve the information for each of the fields on the form by creating a new `nodeList` object by searching for a particular element within the current node (`nl(i)`). Since the format of the document allows only one element with a particular name within the `CUSTOMER` element, I can safely access the first value in the returned `nodeList` since I know it must be the only element in the list. Then I can use the `text` property to extract the value of the `XMLDOMText` node below it and save it in the appropriate field on the form. This results in the updated Web page shown in Figure 22-5.

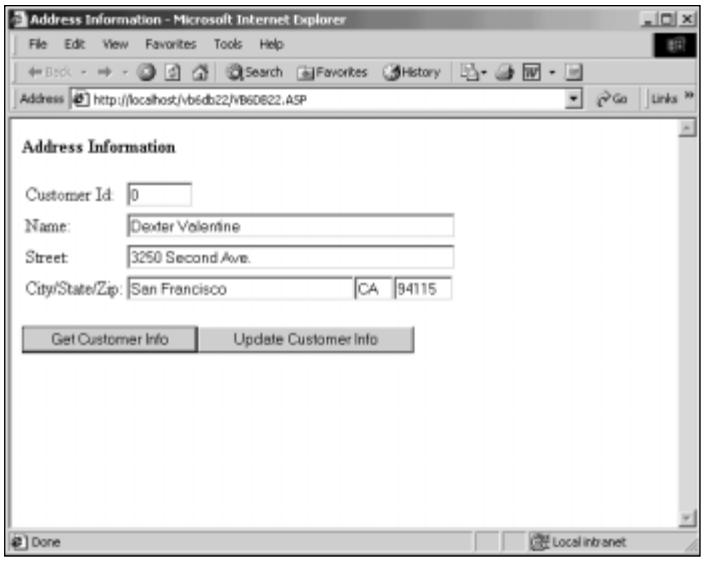

Figure 22-5: Viewing the customer's information

Getting a customer from the database

Now that you understand the client side, it's time to dig into the server side. Since this is an IIS Application, it responds to requests sent to an Internet Information Server (IIS) Web server. In this case, it must respond to an XML document that is transmitted using the `Post` method. It must parse the incoming XML document to determine the information that is requested and then construct a new XML document with the appropriate response.

The `GetCustomer_Respond` event in the XML Server program is triggered any time someone requests a document using the following URL:

```
http://Athena/VB6DB22/VB6DB22.ASP?wci=GetCustomer
```

This URL points to a computer called `Athena` and looks in the `VB6DB22` directory for the file called `VB6DB22.ASP`. It passes the `wci=GetCustomer` parameter to the file, which will trigger the `GetCustomer_Respond` event in the IIS Application (see Listing 22-5).

Listing 22-5: **The GetCustomer_Respond event in XML Server**

```
Private Sub GetCustomer_Respond()

Dim attr As IXMLDOMAttribute
```

```
Dim el As IXMLDOMElement
Dim nl As IXMLDOMNodeList
Dim node As IXMLDOMElement
Dim p As IXMLDOMProcessingInstruction
Dim subnode As IXMLDOMElement
Dim subsubnode As IXMLDOMElement
Dim XMLReq As DOMDocument
Dim XMLResp As DOMDocument
Dim z() As Byte

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

z = Request.BinaryRead(10000)
Set XMLReq = New DOMDocument
XMLReq.loadXML StrConv(z, vbUnicode)
Set nl = XMLReq.getElementsByTagName("CUSTOMER")

Set XMLResp = New DOMDocument
Set p = XMLResp.createProcessingInstruction("xml", _
   "version=""1.0""")
XMLResp.appendChild p

Set node = XMLResp.createElement("GETCUSTOMERINFO")
XMLResp.appendChild node

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

Set rs = New ADODB.Recordset
Set rs.ActiveConnection = db

For Each el In nl
   rs.Source = "Select * From Customers Where CustomerId = " & _
         el.getAttribute("CustomerId")
   rs.Open

   If Not ((rs.BOF) And (rs.EOF)) Then
      Set subnode = XMLResp.createElement("CUSTOMER")
      Set attr = XMLResp.createAttribute("CustomerId")
      attr.Text = rs("CustomerId").Value
      subnode.Attributes.setNamedItem attr
      Set attr = XMLResp.createAttribute("Get")
      attr.Text = "OK"
      subnode.Attributes.setNamedItem attr
```

Continued

Listing 22-5 *(continued)*

```
        node.appendChild subnode

        Set subsubnode = XMLResp.createElement("NAME")
        subsubnode.Text = rs("Name").Value
        subnode.appendChild subsubnode

        Set subsubnode = XMLResp.createElement("STREET")
        subsubnode.Text = rs("Street").Value
        subnode.appendChild subsubnode

        Set subsubnode = XMLResp.createElement("CITY")
        subsubnode.Text = rs("City").Value
        subnode.appendChild subsubnode

        Set subsubnode = XMLResp.createElement("STATE")
        subsubnode.Text = rs("State").Value
        subnode.appendChild subsubnode

        Set subsubnode = XMLResp.createElement("ZIP")
        subsubnode.Text = rs("Zip").Value
        subnode.appendChild subsubnode

    Else
        Set subnode = XMLResp.createElement("CUSTOMER")
        Set attr = XMLResp.createAttribute("CustomerId")
        attr.Text = el.getAttribute("CustomerId")
        subnode.Attributes.setNamedItem attr

        Set attr = XMLResp.createAttribute("Get")
        attr.Text = "Not found"
        subnode.Attributes.setNamedItem attr

        node.appendChild subnode

    End If

    rs.Close

Next el

XMLResp.Save Response

db.Close

Set XMLResp = Nothing
Set XMLReq = Nothing
Set rs = Nothing
```

```
Set db = Nothing

End Sub
```

Preparing to respond to the request

After declaring a whole lot of local variables, I begin processing by using the `Request.BinaryRead` to get the input document into a byte array. Next, I create a new instance of the `DOMDocument` object that will hold the request, and use the `loadXML` method to build the document object hierarchy. Note that I used the `StrConv` function to convert the ASCII encoded string into Unicode before loading it with the `loadXML` method. Finally, I use the `getElementsByTagName` method to create a list of all of the `CUSTOMER` elements.

Then I create a new instance of the response document (`XMLResp`) and initialize it with the standard XML version information. Next, I will append a `GETCUSTOMERINFO` object that will contain the individual `CUSTOMER` elements that form the response.

In order to access the database, I create a new instance of the `ADODB.Connection` object and use the `Open` method to log onto the database server. Then I create a new instance of the `ADODB.Recordset` object and set the `ActiveConnection` property to the `Connection` object I just opened.

Building the response

After all of the prep work, I use a `For Each` loop to access each `CUSTOMER` element in the `nodeList` collection. Using the information from the `CustomerId` attribute, I build a **Select** statement to retrieve information about the specified CustomerId value and then open the `Recordset` object.

If the `Recordset` object contains at least one record (`Not (rs.BOF And rs.EOF)` is `True`), I'll create a new `CUSTOMER` element node using the `createElement` method, and set the `CustomerId` to the current value of CustomerId and the `Get` attribute to `OK`.

For each of the fields that I want to return, I create a new element node and assign it the value from the corresponding database field. Then I add it to the `CUSTOMER` element I created earlier. After I add all of the elements, I close the `Recordset` object.

If the **Select** didn't return any rows, I'll create a `CUSTOMER` element with the `CustomerId` and `Get` attributes as before, but rather than assigning the `Get` attribute a value of `OK`, I'll return `"Not found"`. Afterwards, I'll close the `Recordset` object for the particular `CUSTOMER` element and repeat the `For Each` loop until I'm out of `CUSTOMER` elements to process.

Finally, I'll use the `XMLResp.Save` method against the `Response` object. This will automatically take the XML document stored in the document object model and output the XML tags to the HTTP return stream. Once the document is returned, I can close the database connection and destroy the various objects I created while processing this request.

Updating Customer Information

You've seen one way to handle a transaction using XML documents to carry the request and the response. This is the basic way most XML data exchanges will occur. It doesn't matter if the document exchange returns information or performs a function. As long as the proper information is contained in the document, it really doesn't matter.

However, the `GetCustomerInfo` and `GetCustomer_Respond` routines are based on documents that are element-oriented. Each individual field is stored in a separate element. In the update process, I choose to store each field as an element of the `CUSTOMER` element.

Defining the update XML documents

When requesting an update, you need to include all of the fields that need to be updated in the requesting document. By using attributes instead of elements, you can get a slightly smaller document which probably won't make much of a difference in the long run, but it does result in a flatter hierarchy which can be easier to process with your application program.

Listing 22-6 contains a sample XML document that would be transmitted from the XML client to the XML server to update a particular value. Each of the fields to be updated are stored in a separate attribute, and the `CustomerId` attribute is used to identify the customer's information in the database.

Listing 22-6: An XML document containing update information

```
<?xml version="1.0"?>
<UPDATECUSTOMERINFO>
   <CUSTOMER CustomerId="0" Name="Dexter Valentine"
      Street="3250 Second Ave." City="San Francisco"
      State="CA" Zip="94115"/>
</UPDATECUSTOMERINFO>
```

The document to return the status of the update is based on the same document that was used to request the update (see Listing 22-7). The main differences are that the individual attributes containing the data to be updated are not returned, while a new attribute called `Update` is added that will report the status of the update.

Listing 22-7: An XML document containing the results of the update

```
<?xml version="1.0"?>
<UPDATECUSTOMERINFO>
   <CUSTOMER CustomerId="0" Update="OK"/>
</UPDATECUSTOMERINFO>
```

Requesting an update

Clicking on the Update Customer Info button will trigger the `UpdateCustomerInfo` routine shown in Listing 22-8. This routine begins by creating an object called `XMLReq`, which will hold the XML request document and insert the XML version processing instruction.

Listing 22-8: The UpdateCustomerInfo routine in XML Client

```
Sub UpdateCustomerInfo()

Set XMLReq = CreateObject("MSXML2.DOMDocument")
Set p = XMLReq.createProcessingInstruction("xml",
"version=""1.0""")
XMLReq.appendChild p

Set node = XMLReq.createElement("UPDATECUSTOMERINFO")
Set subnode = XMLReq.createElement("CUSTOMER")

subnode.setAttribute "CustomerId", _
      Document.AddressInfo.CustomerId.Value
subnode.setAttribute "Name", Document.AddressInfo.Name.Value
subnode.setAttribute "Street",
Document.AddressInfo.Street.Value
subnode.setAttribute "City", Document.AddressInfo.City.Value
subnode.setAttribute "State", Document.AddressInfo.State.Value
```

Continued

Listing 22-8 *(continued)*

```
subnode.setAttribute "Zip", Document.AddressInfo.Zip.Value
node.appendChild subnode

XMLReq.appendChild node

MsgBox XMLReq.xml

set http=CreateObject("MSXML2.XMLHTTP")
http.Open "Post", _
   "http://athena/VB6DB22/VB6DB22.ASP?wci=UpdateCustomer",
false
http.setRequestHeader "Content-Type", "text/xml"
http.send XMLReq
Set XMLResp = CreateObject("MSXML2.DOMDocument")
XMLResp.LoadXML http.responsetext

MsgBox XMLResp.xml

End Sub
```

Next, I create the `UPDATECUSTOMERINFO` and `CUSTOMER` elements, which will hold the request. Then I can use the `setAttribute` method to add the various attribute values to the `CUSTOMER` element. Note that the `setAttribute` method will automatically create the `XMLDOMAttribute` object for the attribute if it doesn't exist and automatically append it to the element. If the attribute object already exists, then this method will merely update the value.

Before I send the document to the server, I display it using a MsgBox statement (see Figure 22-6). Then I use the same technique I used earlier to send the request to the XML server and wait for its response. When I receive the response, I display the response to the user to let them know if the update was successful or not (see Figure 22-7).

Figure 22-6: Displaying the update request

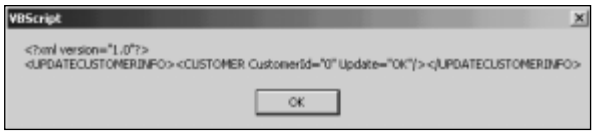

Figure 22-7: Displaying a successful update

Processing an update

On the server side, the `UpdateCustomer_Respond` event will be triggered when an XML document arrives (see Listing 22-9). It uses the same process that the `GetCustomer_Respond` method used to receive the XML document, initialize the return XML document, and open a database connection. I also select all of the elements named `CUSTOMER` and save them in a `nodeList` object. However, from this point on, the two routines differ significantly.

Listing 22-9: **The UpdateCustomer_Respond in XML Server**

```
Private Sub UpdateCustomer_Respond()

On Error Resume Next

Dim attr As IXMLDOMAttribute
Dim el As IXMLDOMElement
Dim nl As IXMLDOMNodeList
Dim node As IXMLDOMElement
Dim p As IXMLDOMProcessingInstruction
Dim parm As ADODB.Parameter
Dim subnode As IXMLDOMElement
Dim subsubnode As IXMLDOMElement
Dim XMLReq As DOMDocument
Dim XMLResp As DOMDocument
Dim z() As Byte

Dim db As ADODB.Connection
Dim cmd As ADODB.Command

z = Request.BinaryRead(10000)
Set XMLReq = New DOMDocument
XMLReq.loadXML StrConv(z, vbUnicode)

Set nl = XMLReq.getElementsByTagName("CUSTOMER")

Set XMLResp = New DOMDocument
```

Continued

Listing 22-9 *(continued)*

```
Set p = XMLResp.createProcessingInstruction("xml", _
      "version=""1.0""")
XMLResp.appendChild p

Set node = XMLResp.createElement("UPDATECUSTOMERINFO")
XMLResp.appendChild node

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

Set cmd = New ADODB.Command
Set cmd.ActiveConnection = db
cmd.CommandText = "Update Customers Set Name=?, Street=?, " & _
      "City=?, State=?, Zip=? Where CustomerId=?"

Set parm = cmd.CreateParameter("Name", adVarChar, adParamInput, 64)
cmd.Parameters.Append parm

Set parm = cmd.CreateParameter("Street", adVarChar, _
      adParamInput, 64)
cmd.Parameters.Append parm

Set parm = cmd.CreateParameter("City", adVarChar, adParamInput, 64)
cmd.Parameters.Append parm

Set parm = cmd.CreateParameter("State", adChar, adParamInput, 2)
cmd.Parameters.Append parm

Set parm = cmd.CreateParameter("Zip", adInteger, adParamInput, 4)
cmd.Parameters.Append parm

Set parm = cmd.CreateParameter("CustomerId", adInteger, _
   adParamInput, 4)
cmd.Parameters.Append parm

For Each el In nl
   cmd.Parameters("CustomerId").Value = _
      el.getAttribute("CustomerId")
   cmd.Parameters("Name").Value = el.getAttribute("Name")
   cmd.Parameters("Street").Value = el.getAttribute("Street")
   cmd.Parameters("City").Value = el.getAttribute("City")
   cmd.Parameters("State").Value = el.getAttribute("State")
   cmd.Parameters("Zip").Value = el.getAttribute("Zip")
   db.Errors.Clear
   cmd.Execute
```

```
    Set subnode = XMLResp.createElement("CUSTOMER")
    subnode.setAttribute "CustomerId", _
       cmd.Parameters("CustomerId").Value
    If db.Errors.Count = 0 Then
       subnode.setAttribute "Update", "OK"

    Else
       subnode.setAttribute "Update", db.Errors.Item(0).Description

    End If
    node.appendChild subnode

Next el

XMLResp.Save Response

Set XMLResp = Nothing
Set XMLReq = Nothing

db.Close
Set cmd = Nothing
Set db = Nothing

End Sub
```

I chose to create a parameterized `Command` object, which uses the **Update** statement to change the contents of the database. So, after creating a new instance of the `Command` object, I create an **Update** statement listing each of the fields I want to update and assigning them a value of question mark (`?`). The question mark is really a placeholder that will be replaced with the parameters associated with the `Command` object.

Then I create the `Parameter` objects for the `Command` object using the `CreateParameter` method. I specify the name, data type, direction, and length for each parameter as I create it, then I `Append` it to the `Command`'s `Parameters` collection. Note that I create the `Parameter` objects in the same order that the question marks appear. This is very important, since the only way to associate a parameter with the statement is the order of the parameters.

Once I've done all of this setup work, I'm ready to use a `For Each` loop to process the list of `CUSTOMER` elements. I use the `getAttribute` method to return the value of each of the attributes from the XML request document and save it as the value in the corresponding `Parameter` object.

After defining the parameters, I clear the `Connection` object's `Errors` collection and `Execute` the command. Then I create the `CUSTOMER` element in the return document and set the `CustomerId` property. If there were no database errors (`db.Errors.Count = 0`), I'll set the `Get` attribute to `OK`; otherwise, I'll set the `Get` attribute to the `Description` property from the first element in the `Errors` collection.

Now I `Append` the `CUSTOMER` element to the `XMLResp` document and retrieve the next node in the `nodeList` object. This process will continue until all of the elements in the `nodeList` object have been processed. I end the routine by saving the `XMLResp` object to the `Response` stream, closing the database connection and destroying the objects I used in this routine.

Thoughts about Programming XML Documents

Without XML at your disposal, getting information across the Web programmatically can be difficult. You have to build complicated programs that will download a Web page containing the information you want, and then try to parse it looking for the proper value. In addition, you have to update your program each time someone updates the format of the Web page. However, XML offers an easier solution.

Using XML it is reasonable to build a pair of applications that communicate with each other via the Internet using XML documents. The client program may be a traditional Visual Basic program, or perhaps a JavaScript-based Web page, that generates an XML document containing a request for information. This request is then passed to a Web server, which receives the XML document, decodes it, and returns the data to the client program. Finally, the client program extracts the information it wants from the return XML document.

The nice thing about this approach is that you can use any tools you want on the client and the server side. The only issue is that both programs must agree on the elements in the XML documents that are exchanged. But as long as XML is used in the middle, the details of the programs on each side aren't important.

The programs can be coded in Visual Basic, VBScript, Java, C++, or even COBOL for that matter. They can run on Windows 2000 Server, Solaris, Linux, or OS/390. The point is, as long as the XML is properly constructed, you have a vendor-independent solution.

You should consider the application I built here merely as a toy to explore what you can do with a little XML, HTML, and a Visual Basic program. I'm not saying that you should run out and convert all of your applications to XML anytime you need to pass information. However, XML is the wave of the future, and anything you can do now to learn more about how to use it will make your life easier in the future.

One of the problems with message queues is that you need a way to pass information between the client application and the transaction server. While you can pass persisted objects back and forth, they can be a pain to develop and debug. Since XML is human readable, it is easier to debug (trust me—debugging a COM+ based message queue application can be a real nightmare) without losing any of the flexibility of objects it would replace.

You could easily combine tools such as an IIS Application, COM+ transactions, message queues, and XML to build a complex, high-performance application that accepts vendor-neutral requests coded in XML. This allows you to develop clients for different platforms, including such operating systems as Linux, Solaris, Macintosh, and even the occasional OS/390 IBM mainframe.

Summary

In this chapter you learned:

- how to design XML documents to request and receive information from a Web server.
- how to build an IIS Application that sends an XML request to a Web server.
- how to build an IIS Application to parse an XML document.
- how to update a database using an XML document.

✦ ✦ ✦