Chapter 7 Building Distributed Applications over HTTP

HTTP is much more than just a protocol used by the Web browser for transmitting HTML. In this chapter I'll show you how to use HTTP generically from a Visual FoxPro client application to execute logic on a Web server that is also running Visual FoxPro. You'll see how to retrieve content and data over HTTP, fire logic on the server and handle bidirectional transfer of data using familiar SQL commands over HTTP. This approach allows you to build rich GUI applications that take advantage of the distributed nature of the Internet without having to conform to a limited HTML interface. Using the techniques described here, you can extend the reach of existing applications across the Internet with minimal amounts of code—using all native Visual FoxPro code.

Hyper thinking

The Web has opened a whole new area of development for building truly distributed applications that can run over widely distributed networks. It has made it possible to build public-access applications at relatively minor cost compared to the infrastructure that was previously required to build this type of distributed application. Much of this has occurred through the use of the Hyper Text Transfer Protocol or HTTP.

Unfortunately, this new medium requires an entirely new approach to application development that focuses heavily on a limited user interface presented in HTML. HTML is a text-based markup language that is essentially line-based and produces output much in the way that ancient word processors like WordStar and WordPerfect of the DOS days did. Compared to typical rich Windows UI applications, even the new 4.0 versions of browsers have a lot of catching up to do in ease of use and usability of the form user interface used for data entry, which is typical for database applications. A lot of things can be done with HTML if you're imaginative, but the end result still leaves a lot to be desired in usability.

The good news is that you *don't have to* build distributed Web applications with HTML. It is absolutely possible to build an application using a rich UI and use the Web simply as a database interface to communicate. The key to making this work is HTTP.

HTTP is based on a client/server model. Typically, the Web browser is the client that requests data from the Web server. The browser is the display mechanism that shows the content served up by the Web server. The server is nothing more than a way station that figures out what type of content to provide to the client. HTTP is the underlying messaging protocol used for all transactions on the World Wide Web. Although the primary use of the protocol is to power the World Wide Web and HTML-based applications, it can do a lot more than plain HTML. Essentially, you can use HTTP to transport any kind of data over the Web, including binary data and even database files!

What if you could build an application using Visual FoxPro on the client side talking to a Web server on the other end of the connection that is also a Visual FoxPro application? Rather than using clumsy HTML you could take advantage of the power of VFP's user interface and ease of data access to build a truly user-friendly application and still get the distributed features promised by HTML-based Web applications.

Microsoft has provided high-level support for various Internet protocols in a system library called WinInet. This library supports relatively simple API interfaces for accessing HTTP, FTP, Gopher and WinSock. I showed the basics of this interface in the last chapter; in this chapter I'll look in much more detail at the HTTP functions that WinInet provides. Microsoft endowed it with a familiar file-based architecture where you can open a connection and then read and write to it directly. This wrapper on top of the WinSock API makes it possible for high-level languages such as Visual FoxPro to directly access the functionality in this system interface.

How can you use HTTP in your applications?

Here's a look at the implications of direct data access over HTTP. The ability to send and receive data in any format you choose gives you the capability to implement your own client/server architecture that can communicate over *any* Internet connection. Here are some useful applications that stand out:

- Any real-time data connection that updates a form from data found on the Web—stock charts or weather information, for example—can simply access an HTTP link and download the data. If you provide timely data to your clients (whether it's financial data or an image captured from your back yard), you can make the data available directly from within a VFP application.

- Software or data updates lend themselves immediately to this technology. Subscription-based services might provide data updates over the Internet using HTTP to download files. The same mechanism can be used to update your actual application files, downloading the update and then running an update program. Technology like this is already in use in many commercial products that prompt you to download updates—your applications easily can do the same.

- Offline applications. Distributed applications often involve taking data offline for processing. The common traveling sales rep example works here: Rep downloads the latest product data for a meeting, goes offline and takes one or multiple orders during the course of the day. At the end of the day the collected data is sent back to the office. Instead of using the internal network via dialup, the tools described here allow this to occur over the Web with a custom application using the Internet as a network.

- Interactive applications that need to communicate with remote Web servers, but don't want to use HTML. Basically any application that runs on the Web could be written with a GUI front end and controlled through these HTTP functions. Some applications simply require a more robust user interface than is possible with HTML today.

- Controlling Web pages from within a client application. For example, it's possible to use search engines from within your applications simply by emulating a browser with native HTTP calls. You can retrieve the result and parse the resulting data into meaningful content. There's tons of information on the Web—it's just a matter of tapping into it. A simple example might be a software registration form—you can feed the values of the form directly to the Web server instead of using an HTML form.

One thing you should keep in mind: Although this approach does not use HTML to display the output, you still need a back-end Web server application to actually provide the data. In the examples here I'll use FoxISAPI as the back end, talking to Microsoft Internet Information Server and providing the logic and data back to the client, but you can talk to any Web server on the Web and use any software on the server that is capable of returning HTTP content.

WinInet with Visual FoxPro

In order to use WinInet you need WinInet.dll in your Windows System directory. This DLL is installed with all recent builds of Windows (SP3 for NT, Win98 and Win95 OSR2) and most of Microsoft's latest Internet tools—if you have Internet Explorer 3.0 or later installed, you have it. Note that you can't simply copy the DLL—at least not the 4.0 version of it—because there are dependencies, and only a full install really works.

! *If you can't rely on people having IE installed, you can use an old version of WinInet.dll that shipped with versions prior to IE 4.0. You can download a copy that's been tested for standalone use from www.west-wind.com/files/wininet.zip. This older file works, with the exception of several features such as proxy support, large HTTPS buffers and using ports other than 80.*

In the following sections I'll introduce the wwIPStuff class, which wraps the WinInet HTTP functionality into an easily accessible interface for application code. The class also provides FTP, SMTP e-mail, domain lookup and a few other features, some of which were discussed in the last chapter. The following code snippets from the class are slightly simplified and edited for length, listing only a few key properties and the relevant snippets used in the examples.

For more detailed code listings and descriptions, see the wwIPStuff.vcx class with the Class Browser, included with the Developer's Download Files at www.hentzenwerke.com.

The easiest way to get started is with the HTTPGet method, which is a simple way to retrieve Web content into a string with a single method call. To use it you can simply do:

```
SET CLASSLIB TO wwIPStuff ADDITIVE
o = CREATE("wwIPStuff")
lcHTML = o.HTTPGet("http://www.west-wind.com/")
```

This retrieves the entire content of the Web page into a string.

The wwIPStuff class implements both a simple HTTPGet interface and a more low-level interface that actually makes the connections with separate method calls, then loads the page and closes the connection. The latter has more flexibility and allows posting data, while the former is a single method call with a single parameter (actually, an optional second parameter specifies the buffer size) that can only retrieve data. Let's have a first look at coding WinInet from Visual FoxPro by looking at the implementation of HTTPGet:

```
********************************************************
DEFINE CLASS wwIPStuff AS CUSTOM
********************************************************
PROTECTED cDLLPath, hHTTPSession, hIPSession, cPostbuffer

*** Custom Properties
cDLLPath = ""

*** Last Error Code
nError = 0
cErrorMsg = ""

*** HTTP Internet Handles
hHTTPSession = 0
hIPSession = 0

cUsername = ""
cPassword = ""

********************************************************
* wwIPStuff :: HTTPGet
*********************************
***  Function: Retrieves an HTTP request from the
***            network and returns a string. Read an
***            HTML or data file across the network.
***    Assume: Blocking call - waits for completion
***            before returning. Only plain HTTP GETs
***            are supported. No passwords. Untested
***            with Proxy Servers - requires Port 80
***            access
***      Pass: tcURL        -  The full URL to retrieve
***                            Only HTTP requests!!!
***            tnBufferSize  -  size of the result(16k)
***    Return: Result string or "" on error
********************************************************
FUNCTION HTTPGet
LPARAMETERS tcUrl, tnBufferSize

tnBufferSize = IIF(!EMPTY(tnBufferSize),;
                 tnBufferSize,16384)

DECLARE INTEGER InternetCloseHandle ;
   IN WININET.DLL INTEGER

DECLARE INTEGER GetLastError IN WIN32API

DECLARE INTEGER InternetOpen ;
   IN WININET.DLL ;
   STRING, INTEGER, STRING, STRING, INTEGER

hInetConnection = ;
   InternetOpen("West Wind Web Connection 2.7",;
   INTERNET_OPEN_TYPE_DIRECT, NULL,NULL,0)

IF hInetConnection = 0
```

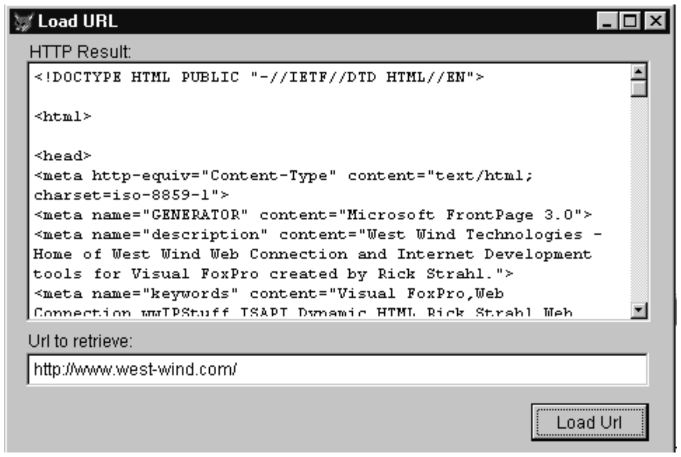
```
   THIS.nError = GetLastError()
   THIS.cErrorMsg = THIS.GetSystemErrorMsg(THIS.nError)
   RETURN ""
ENDIF

THIS.hIPSession = hInetConnection
THIS.WinInetSetTimeout()

SzHead = "Accept: */*" + CR + CR

DECLARE INTEGER InternetOpenUrl ;
   IN WININET.DLL ;
   INTEGER, STRING, STRING, INTEGER,;
   INTEGER,INTEGER

hHTTPResult = InternetOpenUrl(hInetConnection, ;
   tcUrl, szHead,;
   LEN(szHead), INTERNET_FLAG_RELOAD,0);

IF hHTTPResult = 0
   THIS.nError = GetLastError()
   THIS.cErrorMsg = THIS.GetSystemErrorMsg(THIS.nError)
   RETURN ""
ENDIF

DECLARE INTEGER InternetReadFile ;
   IN WININET.DLL ;
   INTEGER, STRING @cBuffer,;
   INTEGER nBuffer, INTEGER @nSizeRead

lcBuffer = SPACE(tnBufferSize)
lnSize = LEN(lcBuffer)

llRetVal = InternetReadFile( ;
   hHTTPResult, @lcBuffer,;
   LEN(lcBuffer), @lnSize)

IF llRetVal = 0
   THIS.nError = GetLastError()
   THIS.cErrorMsg = THIS.GetSystemErrorMsg(THIS.nError)
   RETURN ""
ENDIF

InternetCloseHandle(hHTTPResult)
InternetCloseHandle(hInetConnection)

RETURN (IIF(lnSize > 1, SUBSTR(lcBuffer,1,lnSize),""))
* EOF HTTPGet
```

WinInet works in typical WinAPI style by using handles to connections. A typical connection requires three handles. You first open a handle to your Internet Session, which initializes WinInet and a connection for you. You then use the handle to connect to an actual site, which returns a second handle. The third and final handle is specific to the file or page you want to open and is used to make the read against the open connection. HTTPGet is a high-level function that combines several calls into one for convenience—I'll explain the manual steps further on.

The important thing to remember about handles is that once you create them, you need to get rid of them or you'll incur a handle leak, which will make you run out of handles eventually and leak a small amount of memory for each handle. The wwIPStuff class manages the persistent handles to an extent by using properties and a few simple methods to close them down properly. But always make sure you close a connection before establishing a new one with the same object reference. A good way to make sure of that is to create a new instance of the class for each connection because the Destroy method properly disposes of any open handles.

Time to put this puppy to use. Start with retrieving a simple Web page to a string as shown in **Figure 7.1**. Create a form and add edtHTML and txtUrl fields to the form. Then add a button to load a URL from the Web and add the following code:

```
SET PROCEDURE TO wwIPStuff ADDITIVE
o = CREATE("wwIPStuff")
THISFORM.edtHTML.Value = o.HTTPGet(TRIM(THISFORM.txtUrl.Value),100000)
```



```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">

<html>

<head>
<meta http-equiv="Content-Type" content="text/html;
charset=iso-8859-1">
<meta name="GENERATOR" content="Microsoft FrontPage 3.0">
<meta name="description" content="West Wind Technologies -
Home of West Wind Web Connection and Internet Development
tools for Visual FoxPro created by Rick Strahl.">
<meta name="keywords" content="Visual FoxPro,Web
Connection,wwIPStuff,ISAPI,Dynamic HTML,Rick Strahl,Web
```

http://www.west-wind.com/

***Figure 7.1**. Using WinInet to pull a Web page to a string takes only two lines of code using the wwIPStuff class.*

It's important that the URL you type into txtUrl is fully qualified, including the "http://" protocol prefix. The second parameter specifies the maximum size of the buffer returned to you. The buffer is pre-created and passed to InternetReadFile(), which fills it with the result. The default is 16K, which is enough for typical Web pages, but you can use smaller or larger values as necessary. Small values are useful if you only want to "ping" Web pages to see if they're still running without pulling down a huge Web page. The smaller buffer forces WinInet to read only a small amount of data from the open connection, which makes the request run much faster than a full document buffer returned.

In the example, when you click the button, HTTPGet goes out to the Web page specified and pulls it down into the edit box for viewing.

At first thought it might seem silly to pull down just a Web page—after all, you can use a browser to actually view the data, and you could even view that data in your own form using the WebBrowser control. Using WinInet is much more resource friendly than IE and is a non-

UI operation that doesn't need a form and doesn't use COM, but rather optimized API calls. The simple interface is also easier to use than IE and is not subject to timing issues as IE is.

Once you have the data as a string, you can strip the page of all HTML tags and use it as plain text, using a function called StripHTML(). This useful function is part of wwUtils, which is included in this project.

For example, the above Web page includes a page counter that shows the number of hits. I could use the following code to extract this information from the Web page:

```
SET classlib TO wwIPStuff ADDITIVE
SET PROCEDURE TO wwUtils ADDITIVE

o = CREATE("wwIPStuff")
lcHTML = o.HTTPGet("http://www.west-wind.com",100000)

lcHTML = StripHTML(lcHTML)  && Remove HTML tags
lcCount = Extract(lcHTML,;
                  "This page has been visited",;
                  "times")
IF !EMPTY(lcCount)
  lnCount = VAL(lcCount)
ENDIF
```

StripHTML() and Extract() are a couple of useful functions for parsing data out of strings. They can be found in the wwUtils procedure file available with the Developer's Download Files at www.hentzenwerke.com. I've used similar code to download data from the National Weather Service to retrieve wind information for placement on maps. There's lots of data out there in raw form. If you can get at it via an HTTP request, you can probably put that data to use in your programs. These functions should help.

Retrieving plain HTML can also be useful for checking up on a site to determine whether it's still running, for building a stress tester that continually hammers a site, or using a Web spider that crawls links in a page to collect information for you. Or you can pull Web pages into database memo fields and then display them at a later time with the Web browser control.

Doing data over HTTP

HTML is okay, but it lacks the precision of database data. In the ASP and FoxISAPI server chapters you learned how to build Web request servers that can send HTTP results back to the browser. In most cases you sent back HTML, but of course it's also possible to send back data. **Figure 7.2** shows a form that downloads a customer list based on a name typed in the text box.

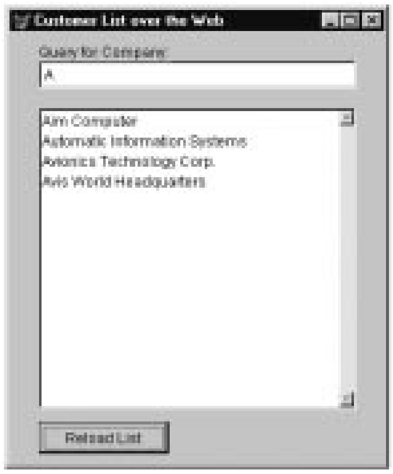

Customer List over the Web
Query for Company:
A
Aim Computer
Automatic Information Systems
Avionics Technology Corp.
Avis World Headquarters
Reload List

Figure 7.2. *Downloading FoxPro table data over the Web involves sending the file as a string from server to client.*

This code goes in the Reload button's Click() event:

```
o = CREATE("wwIPStuff")

*** Retrieve all companies in the txtQueryCompany Field
lcText = o.HTTPGet(;
   "http://localhost/foxisapi/foxisapi.dll?http~CustList1~"+;
   TRIM(THISFORM.txtQueryCompany.value),100000)

IF EMPTY(lcText) OR lcText="FAILED"
   WAIT WINDOW "Invalid HTTP Response..." NOWAIT
   RETURN
ENDIF

tcFilename = SYS(3) + ".txt"
lnHandle = FCREATE(tcFileName)

=FWRITE(lnHandle,lcText)
=FCLOSE(lnHandle)

THISFORM.lstCustList.RowSourceType = 0

CREATE CURSOR TCustList ;
(  CUSTNO      C (8),;
   COMPANY     C (30),;
   CAREOF      C (30) )

APPEND FROM (tcFileName) DELIMITED
ERASE (tcFileName)

THISFORM.lstCustList.RowSourceType = 2
THISFORM.lstCustList.RowSource = "tCustList.company, careof"
THISFORM.lstCustList.Requery
```

This search dialog makes a request against a FoxISAPI Web server back end, requesting the specified customers. The server obliges by sending a list of comma-delimited records as a text string. To quickly see what the server sends, try this URL in the browser:

```
http://localhost/foxisapi/foxisapi.dll/fidemo.httpdemo.process?
                                     method=CustList1&Company=B
```

You'll see the comma-delimited list of records in the browser, minus the line breaks. The server sends this string and the client code simply takes the string, dumps it to a file and then imports it into a temporary cursor. The list box then is assigned a Rowsource pointing at this cursor and displays it in the list.

I implemented the server piece using FoxISAPI because it offers the greatest flexibility when dealing with native FoxPro table data.

Before running any of these demos, make sure you've created the servers in Chapter 5, or at least read through how to build a new FoxISAPI server using the wwFoxISAPI class. The classes shown here were compiled as part of the FIDemo server, and the source code can be found in <InstallPath>\FoxISAPI\http.

The server class that contains the demos is called HTTPDemo in HTTP.prg in the Developer's Download Files at www.hentzenwerke.com. You need to manually add the HTTP.prg file to the FiDemo project and recompile the project. Remember to set the server to SingleUse once you've compiled the project, and then recompile. You must use the updated FoxISAPI.dll from www.hentzenwerke.com in order for the binary transfers (described later in this chapter) to work. If you ran the FoxISAPI examples in Chapter 5 you won't have to do anything further since the HTTP files were already compiled into the server. If you haven't run the Chapter 5 samples, you will have to read through the sections that explain how to compile and configure the FoxISAPI servers.

The FoxISAPI server code for the customer list looks like this (additional server methods in this chapter will be inserted into this class later):

```
*************************************************************
DEFINE CLASS HTTPDemo AS wwFoxISAPI OLEPUBLIC
*************************************************************

********************************************************
* HTTPDemo :: CustList1
*********************************
***  Function: Returns a customer list based on the URL
***            'parameter' passed. Delimited returned.
********************************************************
FUNCTION CustList1

lcCustToFind = Request.QueryString("Company")

lcFile = SYS(3) + ".TXT"

SELECT custno,Company, Careof ;
   FROM (DATAPATH + "TT_Cust") ;
   WHERE Company = lcCustToFind ;
   ORDER BY Company ;
   INTO CURSOR TQuery

COPY TO (lcFile) TYPE DELIMITED

*** Send the Delimited string over the wire
Response.Write(FileToString(lcFile))

ERASE (lcFile)

USE IN TQuery
IF USED("TT_CUST")
   USE IN TT_Cust
ENDIF

ENDFUNC
* CustList1

ENDDEFINE
```

The server code is straightforward: It retrieves the Query String parameter we passed on the URL as Company=B to use as a filter in the SELECT statement that retrieves the actual customer records. The result is copied to a temporary file in delimited form. The file is read back into a string and then simply sent as-is into the HTTP output stream with Response.Write().

Instead of a comma-delimited list, you can also send a file directly. You can simply use **`FILETOSTR(lcDBFName)`** and send that down the HTTP stream! On the other end you pull in the resulting string and dump it back to the file with **`STRTOFILE(lcResult,lcTempDBFName)`**, which you can then attach to the list box to view or import into the cursor as shown above.

Real data over HTTP

Comma-delimited files and files wrapped up as strings are only a start. The problem is that the DBF file format consists of more than one file to contain both the standard data and memo fields. Neither approach shown in the last section works with files that contain memo fields because you actually need to send two files. To get around this problem, you need to use an encoding scheme that knows about Visual FoxPro data. Here are a couple of functions found in wwIPStuff that do just that:

```
********************************************************
* wwIPStuff :: EncodeDBF
*********************************
***  Function: This function encodes a DBF file ready to
***            be sent up to a server using HTTPGetEx in
***            the POST buffer. The file will be URL
***            encoded.
***    Assume: Note you can send a ZIP file here, too!
***            100 byte header on top of file contains
***            5 byte ID (wwDBF) filename (40 bytes) and
***            size(10 bytes) for each
***            file
***      Pass: lcDBF     - Full DBF filename w/ ext
***            llHasMemo - .t. or (.f.)
***    Return: Encoded Buffer or "" on failure
********************************************************
```

```
LPARAMETERS lcDBF, llHasMemo
LOCAL lcBuffer1, lcBuffer2

lcDBF = IIF(TYPE("lcDBF") = "C",UPPER(lcDBF),"")

IF !FILE(lcDBF)
   RETURN ""
ENDIF

lcBuffer1 = File2Var(lcDBF)
lcHeader = "wwDBF" + PADR(justfname(lcDBF),40) + ;
            STR(LEN(lcBuffer1),10)
IF !llHasMemo
  lcHeader = lcHeader + SPACE(50)  && Pad out header
  RETURN lcHeader + lcBuffer1
ENDIF

lcFPT = STRTRAN(LOWER(lcDBF),".dbf",".fpt")

lcBuffer2 = File2Var(lcFPT)
lcHeader = lcHeader + PADR(JUSTFNAME(lcFPT),40) + ;
          STR(LEN(lcBuffer2),10)

RETURN lcHeader + lcBuffer1 + lcBuffer2

********************************************************
* wwIPStuff :: DecodeDBF
*********************************
***  Function: Decodes a buffer that has been encoded
***            with EncodeDBF
***    Assume: The inbound buffer should *NOT* be
***            URLEncoded since the result from an
***            HTTPGET operation will already have been
***            decoded for you!
***      Pass: lcBuffer - String that contains the file
***            lcDBF    - Full Name of DBF to create
***    Return: .t. or .f.
********************************************************

LPARAMETERS lcBuffer,lcDBF
LOCAL lnSeparator

IF LEN(lcBuffer) < 105
   RETURN .F.
ENDIF

lcHeader = SUBSTR(lcBuffer,1,105)
lcFname = TRIM(SUBSTR(lcBuffer,6,40))
lnSize1 = VAL(SUBSTR(lcBuffer,46,10))
lnSize2 = VAL(SUBSTR(lcBuffer,96,10))


*** Use parm or the filename specified in the header
lcDBF = IIF(EMPTY(lcDBF),lcFname,UPPER(lcDBF))

IF lcHeader # "wwDBF"
   WAIT WINDOW NOWAIT "Invalid Decode File Header"
   RETURN .f.
ENDIF
```

```
lcFile1 = ""
lcFile2 = ""

IF lnSize1 > 0
   lcFile1 = SUBSTR(lcBuffer,106,lnSize1)
   IF LEN(lcFile1) < lnSize1
      WAIT WINDOW NOWAIT "Invalid File Size: " + ;
               STR(LEN(lcFile1)) + " of " + STR(lnSize1)
      RETURN .F.
   ENDIF
ENDIF
IF lnSize2 > 0
   lcFile2 = SUBSTR(lcBuffer,106 + lnSize1, lnSize2)
   lnSizex = LEN(lcFile2)
   IF LEN(lcFile2) < lnSize2 - 1
      WAIT WINDOW NOWAIT "Invalid Memo File Size: " +;
                 STR(LEN(lcFile2)) + " of " + STR(lnSize2)
      RETURN .F.
   ENDIF
ENDIF

=File2Var(lcDBF,lcFile1)

IF !EMPTY(lcFile2)
   =File2Var(STRTRAN(lcDBF,".DBF",".FPT"),lcFile2)
ENDIF

RETURN .T.
```

These functions work by taking an input file and adding a header that describes what's contained in the file. The header includes a size for each component and a file name so that the decoding party can figure out the name of the file encoded. The additional benefit of this header is that it's an easy way to tell whether the entire file has been retrieved, by checking the file sizes and comparing them against the decoded strings. If the size doesn't match, an error occurred during the download or the file was incorrectly encoded.

With this code in place, you can send memo data when sending data on the server:

```
********************************************************
* HTTPDemo :: CustList2
*********************************
***  Function: Returns a customer list based on the URL
***            'parameter' passed. This time as file!
***    Assume: wc.dll?http~CustList1~CompanySearchString
********************************************************


FUNCTION CustList2

lcCustToFind = Request.QueryString("Company")

lcFile = SYS(3) + ".DBF"

*** This query includes Memos
SELECT custno,Company, Careof, Address, phone ;
   FROM TT_Cust ;
   WHERE UPPER(Company) = UPPER(lcCustToFind) ;
   ORDER BY Company ;
   INTO DBF (lcFile)
USE

o = CREATE("wwIPStuff")

*** Encode with Memo File
lcText = o.EncodeDBF(lcFile,.T.)

*** Send the binary File string over the wire
Response.Write(lcText)

ERASE (lcFile)

USE IN TT_Cust

ENDFUNC
* CustList1
```

Both client and server use wwIPStuff to handle the encoding on the server and decoding on the client. EncodeDBF() simply takes a file name as input and creates a string from the file. If the llMemo parameter is true, the DBF and FPT are packaged together. On the client side, use DecodeDBF() to unpack the file into DBF and/or Memo files:

```
*** Retrieve all companies starting with "A"
lcText = o.HTTPGet(;
    "http://localhost/foxisapi/foxisapi.dll/fidemo.httpdemo.process?";
    " Method=CustList2&Company=A")

*** Creates the file including Memo
IF !o.DecodeDBF(lcText,"TCustList.dbf")
   RETURN  && DecodeDBF will display nowait WAIT win
ENDIF

*** Do something with the file
USE TCustList
BROWSE
USE

*** Clear out the temporary files
ERASE TCustList.dbf
ERASE TCustList.FPT

RETURN
```

Cool, isn't it? Fewer than 20 lines of code for both the client and server! With this basic technology you can very easily update data over the Web. To make this even more efficient, you could add third-party ZIP control to the Encode and Decode functions and zip the data on the fly once it gets over a certain size, cutting down on the size of the text traveling over the wire.

You can use a standard GUI VFP application to access information that is retrieved on a remote server. This can be a polling-type link that might use a timer to occasionally update data you see in a form, or a by-request operation where the user requests the update from a button click.

Sending data with HTTP POST

Up to now we've only *requested* data from the server, but your application might also need to *update* data on the server. For example, you might have a salesperson log on to the Web and send the sales data she collected over the course of the day back to the home office. To do this, you need a mechanism to send data *to* the Web server from your VFP application.

HTTP has a built-in mechanism for sending data to the server called a POST request (there are several ways to do so, but POST is most commonly used). The most common use of POST requests occurs when you submit an HTML input form on a Web page. The data is encoded (the format is known as URLEncoded) and sent up to the server via the HTTP request initiated by the browser. A typical Web back-end application can then query the posted data to retrieve the form variables as part of the server data sent to the back end. With FoxISAPI and the wwFoxISAPI class I'm using here, the retrieval mechanism is Request.Form().

Typically, only small amounts of data are sent via POST fields, but here again, as with the results returned from an HTTP GET operation, you can send *any* kind and size of data. Although an HTTP POST sends data to the server, POST operations are typically piggybacked on a regular HTTP request so you can also return an HTTP result as part of the same request. In other words, you could post data to simulate an HTML input form *and* retrieve the result page or response that tells whether the request succeeded. The server then has the option of using the input to perform additional logic before creating a result to send back to the client. This two-way communication allows you to build sophisticated server communications with HTTP POST requests.

When I showed the HTTPGet() method, I used the simplified WinInet function *InternetOpenUrl*, which handles most of the opening, loading and retrieving results. When running a POST request, however, you need to use lower-level WinInet functions to add the additional settings required to send a buffer. This means calling three separate methods in the wwIPStuff class: HTTPConnect() to connect to a server, HTTPGetEx() to actually retrieve and/or send the data, and HTTPClose() to shut down the connection. The actual API code also has to take a few extra steps, making a few additional low-level connections through WinInet—in all, it's quite a jumble of DECLARE – API definitions. But the low-level mechanism pays off by providing many other options, such as allowing access to authentication of username and password, secure connections via Secure Sockets Layer (SSL), the HTTP headers and, most importantly, the POST buffer. Here's the code for the three methods:

```
********************************************************
* wwIPStuff :: HTTPConnect
*********************************
***  Function: Connect to an HTTP server.
***    Assume: Sets two handle values in this class. Each
***            instance of this class can only manage
***            one HTTP session at a time. Use this low
***            level function for quick repeated access
***            to HTTP pages.
***      Pass: lcServer   - Server name
***            lcUsername - Optional Username
***            lcPassword - Optional Password
***            llHTTPS    - .T. for secure connections
***    Return: 0 on success or WinAPI Errorcode
********************************************************
LPARAMETER lcServer, lcUsername, lcPassword, llHTTPS
LOCAL lhIP, lhHTTP, lnError, lnHTTPPort

lcServer = IIF(!EMPTY(lcServer), lcServer, THIS.cServer)
lcUsername = TRIM(IIF(!EMPTY(lcUsername), lcUsername, THIS.cUsername))
lcPassword = TRIM(IIF(!EMPTY(lcPassword), lcPassword, THIS.cPassword))

*** Assign Default Ports
IF THIS.nHTTPPort = 0
   lnHTTPPort =   IIF(llHTTPS,INTERNET_DEFAULT_HTTPS_PORT,;
                          INTERNET_DEFAULT_HTTP_PORT)
ELSE
   lnHTTPPort = THIS.nHTTPPort
ENDIF

THIS.lSecureLink = llHTTPS OR THIS.lSecureLink

THIS.cServer = lcServer

THIS.nError = 0
THIS.cErrorMsg = ""

DECLARE INTEGER InternetCloseHandle ;
   IN WinInet.DLL ;
   INTEGER

DECLARE INTEGER GetLastError;
   IN WIN32API

DECLARE INTEGER InternetOpen ;
   IN WININET.DLL ;
   STRING,;
   INTEGER,;
   STRING, STRING, INTEGER

hInetConnection = ;
   InternetOpen("West Wind Web Connection 3.00",;
   THIS.nHTTPConnectType,;
   NULL,NULL,0)


IF hInetConnection = 0
   THIS.nError = GetLastError()
   THIS.cErrorMsg = THIS.GetSystemErrorMsg(THIS.nError)
   RETURN THIS.nError
ENDIF

THIS.hIPSession = hInetConnection
THIS.WinInetSetTimeout()

DECLARE INTEGER InternetConnect ;
   IN WININET.DLL ;
   INTEGER hIPHandle,;
   STRING lpzServer,;
   INTEGER dwPort, ;
   STRING lpzUserName,;
   STRING lpzPassword,;
   INTEGER dwServiceFlags,;
   INTEGER dwReserved,;
   INTEGER dwReserved


lhHTTPSession=;
     InternetConnect(hInetConnection,;
     lcServer,;
     lnHTTPPort,;
     lcUsername,;
     lcPassword,;
     INTERNET_SERVICE_HTTP,;
     0,0)


IF (lhHTTPSession = 0)
   =InternetCloseHandle(hInetConnection)
   THIS.nError = GetLastError()
   THIS.cErrorMsg = THIS.GetSystemErrorMsg()
   RETURN THIS.nError
ENDIF


THIS.hIPSession = hInetConnection
THIS.hHTTPSession = lhHTTPSession

RETURN 0

********************************************************
* wwIPStuff :: HTTPGetEx
*********************************
***  Function: Retrieves an HTTP request from the
***            network and returns a string. Read an
***            HTML or data file across the net.
***    Assume: Blocking call - waits for completion
***            before returning. Use AddPostKey
***            to post data to server
***            Must call HTTPConnect/HTTPClose to
***            manage connection to Server.
***      Pass: tcURL        - URL to retrieve
***            tcBuffer     - HTTP result (by Reference)
***            tnBufferSize - Size of the buffer (ref)
***            tcHeaders    - HTTP Headers sent from
***                           client request. Separate
***                           key:value pairs with CR
***    Return: WinAPI Error Code (check THIS.cErrorMsg)
```

```
*******************************************************
LPARAMETERS tcPage, tcBuffer, tnBufferSize, tcHeaders
LOCAL hHTTPResult, lcOldAlias

tcPage = IIF(EMPTY(tcPage),THIS.cLink,tcPage)
tnBufferSize = IIF(TYPE("tnBufferSize") = "N",;
                 tnBufferSize,0)

lcOldAlias = ALIAS()

THIS.cLink = tcPage

IF USED("wwPostBuffer")
   SELECT wwPostBuffer
   tnPostSize = LEN(wwPostBuffer.cPostBuffer)
   lcPostBuffer = IIF(tnPostSize > 0, wwPostBuffer.cPostBuffer, NULL)
ELSE
   tnPostSize = 0
   lcPostBuffer = NULL
ENDIF

IF USED(lcOldAlias)
   SELECT (lcOldAlias)
ENDIF

THIS.nError = 0
THIS.cErrorMsg = ""

DECLARE INTEGER HttpOpenRequest ;
   IN WININET.DLL ;
   INTEGER hHTTPHandle,;
   STRING lpzReqMethod,;
   STRING lpzPage,;
   STRING lpzVersion,;
   STRING lpzReferer,;
   STRING lpzAcceptTypes,;
   INTEGER dwFlags,;
   INTEGER dwContext

HHTTPResult = HttpOpenRequest(THIS.hHTTPSession,;
   IIF( tnPostSize > 0, "POST","GET"),;
   tcPage,;
   NULL,NULL,NULL,;
   INTERNET_FLAG_RELOAD + IIF(THIS.lSecureLink,INTERNET_FLAG_SECURE,0),0)

IF (hHTTPResult = 0)
   THIS.nError = GetLastError()
   THIS.cErrorMsg = THIS.GetSystemErrorMsg()
   RETURN THIS.nError
ENDIF
THIS.hHTTPSession = hHTTPResult

*** HTTPSendRequest actually goes out and gets the data
*** Note the buffer may not be completely downloaded
*** when you start reading the data below
DECLARE INTEGER HttpSendRequest ;
   IN WININET.DLL ;
   INTEGER hHTTPHandle,;
   STRING lpzHeaders,;
   INTEGER cbHeaders,;
   STRING lpzPost,;
   INTEGER cbPost

IF tnPostSize > 0
  tcHeaders = "Content-Type: application/x-www-form-urlencoded" +;
              IIF(!EMPTY(tcHeaders),CR + tcHeaders,"")
ELSE
  tcHeaders =  IIF(!EMPTY(tcHeaders),tcHeaders,"")
ENDIF

lnRetval = 0
lnRetval = HttpSendRequest(hHTTPResult,;
   tcHeaders,LEN(tcHeaders),;
   lcPostBuffer,tnPostSize)

IF lnRetval = 0
   THIS.nError = GetLastError()
   THIS.cErrorMsg = THIS.GetSystemErrorMsg()
   =InternetCloseHandle(hHTTPResult)
   RETURN THIS.nError
ENDIF


*** Retrieve the HTTP Header
DECLARE INTEGER HttpQueryInfo ;
   IN WININET.DLL ;
   INTEGER hHTTPHandle,;
   INTEGER nType,;
   STRING @cHeaders,;
   INTEGER @cbHeaderSize,;
   STRING cNULL

lcHeaders = SPACE(1024)
lnHeaderSize = 1024
lnRetval = HttpQueryInfo(hHTTPResult,;
                        HTTP_QUERY_RAW_HEADERS_CRLF,;
                        @lcHeaders,@lnHeaderSize,NULL)
THIS.cHTTPHeaders = TRIM(STRTRAN(lcHeaders,CHR(0),""))

*** Call HTTP Event method – 0 means the header is sent
THIS.OnHTTPBufferUpdate(0,0,THIS.cHTTPHeaders)

*** Now Read the actual result data
DECLARE INTEGER InternetReadFile ;
   IN WININET.DLL ;
   INTEGER hHTTPHandle,;
   STRING @lcBuffer,;
   INTEGER cbBuffer,;
   INTEGER @cbBuffer

IF tnBufferSize > 0
    *** Use Fixed Buffer Size
  tcBuffer = SPACE(tnBufferSize)
  lnBufferSize = tnBufferSize
  lnRetval = InternetReadFile(hHTTPResult,;
     @tcBuffer,;
     tnBufferSize,;
     @tnBufferSize)
```

```
ELSE
    *** Build the buffer dynamically
    tcBuffer = ""
    tnSize = 0
    lnRetVal = 0
    lnBytesRead = 1
    lnBufferReads = 0
    DO WHILE .T.
        lcReadBuffer = SPACE(THIS.nHTTPWorkBufferSize)
        lnBytesRead = 0
        lnSize = LEN(lcReadBuffer)

       lnRetval = InternetReadFile(hHTTPResult,;
          @lcReadBuffer,;
          lnSize,;
          @lnBytesRead)

       IF lnRetVal = 1 AND lnBytesRead > 0
          *** Update the input parameters - result buffer and size of buffer
          tcBuffer = tcBuffer + lcReadBuffer
          tnBufferSize = tnBufferSize + lnBytesRead
          lnBufferReads = lnBufferReads + 1
          THIS.OnHTTPBufferUpdate(tnBufferSize,lnBufferReads,@lcReadBuffer)
       ENDIF
        IF (lnRetVal = 1 AND lnBytesRead = 0) OR (lnRetVal = 0)
           EXIT
        ENDIF
    ENDDO
    lnBufferSize = tnBufferSize
ENDIF

IF lnRetval = 0
   THIS.nError = GetLastError()
   THIS.cErrorMsg = THIS.GetSystemErrorMsg()
ENDIF

=InternetCloseHandle(hHTTPResult);

tcBuffer = (IIF(tnBufferSize > 1 AND tnBufferSize <= lnBuffersize, ;
  SUBSTR(tcBuffer,1,tnBufferSize),""))

RETURN THIS.nError
```

```
********************************************************
* wwIPStuff :: HTTPClose
*********************************
***  Function: Closes an HTTP Session.
***    Return: nothing
********************************************************

DECLARE INTEGER InternetCloseHandle ;
   IN WININET.DLL ;
   INTEGER hIPSession

=InternetCloseHandle(THIS.hHTTPSession)
=InternetCloseHandle(THIS.hIPSession)

THIS.hHTTPSession = 0
THIS.hIPSession = 0
RETURN

********************************************************
* wwIPStuff :: AddPostKey
*********************************
***  Function: Adds a key to the post string
***      Pass: tcKey   -  Key to add (or RESET to clear)
***            tcValue -  The value to set it to
***    Return: nothing
********************************************************
LPARAMETERS tcKey, tcValue
LOCAL lcOldAlias
tcKey=IIF(TYPE("tcKey") = "C",tcKey,"")
tcValue=IIF(TYPE("tcValue") = "C",tcValue,"")

lcOldAlias=Alias()

IF !USED("wwPostBuffer")
   *** Use a cursor so we can hold very large buffers
   CREATE CURSOR wwPostBuffer ;
      ( cPostBuffer M)
   APPEND BLANK
ENDIF

SELECT wwPostBuffer

IF tcKey = "RESET"
  REPLACE wwPostBuffer.cPostbuffer WITH ""
  RETURN
ENDIF

IF !EMPTY(tcKey)
   * THIS.cPostBuffer= && No good for buffers over 1meg
  REPLACE wwPostBuffer.cPostBuffer WITH wwPostBuffer.cPostBuffer + tcKey + ;
                   "=" + URLEncode(tcValue) + "&"
ENDIF

IF USED(lcOldAlias)
   SELECT (lcOldAlias)
ENDIF
```

These low-level functions take advantage of a number of customization options for the HTTP request by checking various class properties and parameter values. There is support for security (HTTPS and authentication with username and password), attaching custom HTTP headers to the request (tcHeaders input parameter), the ability to retrieve the HTTP header (cHTTPHeader) from a request in addition to the content, and, most importantly, the ability to send information and data to the Web server by using the POST buffer.

POST data is attached to the request through a special method called AddPostKey. You call this method by supplying a key/value pair:

```
oIP.AddPostKey("Name","Rick Strahl")
```

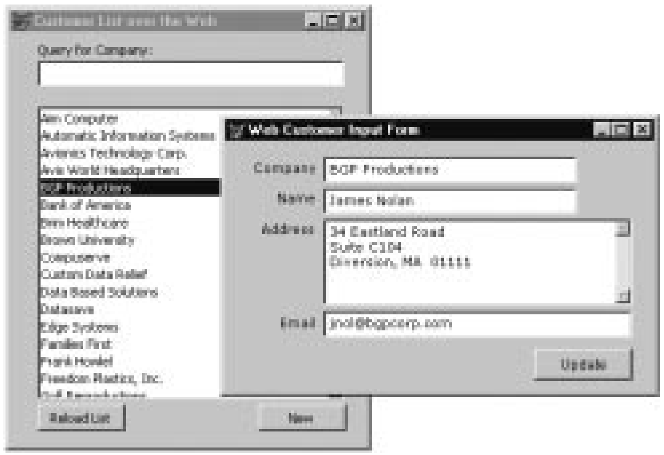
AddPostKey() properly encodes the variable data and formats it to simulate an HTML form submission. POSTed data must be in URLEncoded format, which strips out all "unsafe" characters and converts the input into a plain ASCII string. *Safe* characters in this context include A-Z, a-z and 0-9—all other characters are converted into their hexadecimal ASCII codes: %0D for a carriage return (CHR(13)), for example. This process can be very slow, especially if you do it with Visual FoxPro code on a very large binary file (remember our goal is to send binary file images of DBF files). The URLEncode function (in wwutils.prg) handles the conversion of the string. To make this slow process a lot faster on large strings, the wwIPStuff.dll file contains a routine that does it with C code when the buffer is greater than a couple of thousand bytes.

```
********************************************************
PROCEDURE URLEncode
*******************
***  Function: Encodes a string in URL encoded format
***            for use on URL strings or when passing a
***            POST buffer to wwIPStuff::HTTPGetEx
***      Pass: tcValue  -   String to encode
***    Return: URLEncoded string or ""
********************************************************
LPARAMETER tcValue
LOCAL lcResult, lcChar, lnSize, x

*** Large Buffers use the wwIPStuff function
*** for quicker response
IF LEN(tcValue) > 2048
   LnSize = LEN(tcValue)
   TcValue = PADR(tcValue,lnSize * 3)

   DECLARE INTEGER VFPURLEncode ;
      IN WWIPSTUFF ;
      STRING @cText,;
      INTEGER cInputTextSize

   LnSize = VFPUrlEncode(@tcValue,lnSize)

   IF lnSize > 0
      RETURN SUBSTR(TRIM(tcValue),1,lnSize)
   ENDIF

   RETURN ""
ENDIF
```

```
*** Do it in VFP Code
lcResult = ""

FOR x = 1 TO len(tcValue)
   lcChar = SUBSTR(tcValue,x,1)
   IF ATC(lcChar,"ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789") > 0
      lcResult = lcResult + lcChar
      LOOP
   ENDIF
   IF lcChar = " "
      lcResult = lcResult + "+"
      LOOP
   ENDIF
   *** Convert others to Hex equivalents
   lcResult = lcResult + "%" + RIGHT(TRANSFORM(ASC(lcChar),"@0"),2)
ENDFOR && x = 1 to len(tcValue)

RETURN lcResult
* EOF URLEncode
```

Posting application data

With this wwIPStuff framework code in place, you can send data to the server. The simplest thing you can do is to create a Visual FoxPro form that simulates a Web page. Let's build a short data entry form that allows you to enter new customer information into a database by extending the customer list display sample we used originally. **Figure 7.3** shows the customer entry form that you can access by clicking on a customer in the list or by clicking the New button on the list form.



Aim Computer
Automatic Information Systems
Avionics Technology Corp.
Avis World Headquarters
BGP Productions
Bank of America
Brim Healthcare
Brown University
Compuserve
Custom Data Relief
Data Based Solutions
Datasave
Edge Systems
Families First
Frank Howlel
Freedom Plastics, Inc.

Company: BGP Productions
Name: James Nolan
Address: 34 Eastland Road
Suite C104
Diversion, MA 01111
Email: jnol@bgpcorp.com

Figure 7.3. *Extending the customer list example with the ability to update data on the Web server using HTTP POST to submit the data.*

First bring up the CustList.scx file used in the first data example and add the following code to the list box Valid event to allow viewing specific customer information:

```
DO FORM custinput with TCustList.CustNo
```

Pass down the customer ID so that the customer form can retrieve the entire record from the Web site by using this ID as a parameter. In the CustInput form's Init event you'll find the code responsible for pulling the customer data from the server:

```
* CustInput::Init
LPARAMETER lcCustNo

SET PROCEDURE TO wwUtils ADDITIVE
SET CLASSLIB TO wwIPStuff ADDITIVE

IF !EMPTY(lcCustNo)
   *** Retrieve the full record
   loIP = CREATE("wwIPStuff")
   lcFile =   loIP.HTTPGet(;
"http://localhost/foxisapi/foxisapi.dll/fidemo.httpdemo.Process?";
"Method=GetCust&Custno="+Alltrim(lcCustno),20000)

   IF UPPER(lcFile) = "ERROR"
      WAIT WINDOW NOWAIT lcFile
      RETURN
   ENDIF

   IF loIP.DecodeDbf(lcFile,"TCust.dbf")
      SELECT 0
      USE TCust
      THISFORM.txtCustNo.VALUE = TCust.Custno
      THISFORM.txtCompany.VALUE = TCust.Company
      THISFORM.txtCareof.VALUE = TCust.Careof
      THISFORM.edtADdress.VALUE = TCust.Address
      THISFORM.edtEmail.VALUE = TCust.Email

      USE IN TCust
      ERASE TCust.DBF
      ERASE TCust.FPT
   ELSE
      WAIT WINDOW "Couldn't load Customer " + lcCustNo
      RETURN
   ENDIF
   THISFORM.btnInsert.caption = "Update"
ENDIF
```

If a customer ID was passed in, the code pulls the customer record from the server by downloading the record as a file and storing it to Tcust.dbf/fpt. The `&Custno=" + ALLTRIM(lcCustno)` portion of the URL lets the server know which record to retrieve. The server packages up the file using EncodeDBF() and then sends it down to the client, which uses DecodeDBF() to convert the file back into a DBF called Tcust. The values from this file are then stuffed into the form fields for display. Note that there's some minimal error handling in this code, which requires checking the result HTTP buffer for error string results. The scheme I use typically returns "OK" when all is well, or "Error: Error string" if something didn't work.

You also need to check for a blank string, which means an error occurred in the connection. More on error handling later.

The form can also be updated. If you have a new customer, the txtCustno field will be empty, otherwise it'll contain the existing customer ID. Based on this, the server knows whether to INSERT a new record or UPDATE an existing one. The client-side code that posts the fields to the server looks like this:

```
oIP = CREATE("wwIPStuff")

oIP.HTTPConnect("localhost")

lcOutput = ""
lnSize = 0

oIP.AddPostKey("Company", THISFORM.txtCompany.value)
oIP.AddPostKey("Name", THISFORM.txtCareof.value)
oIP.AddPostKey("Address", THISFORM.edtAddress.value)
oIP.AddPostKey("Email", THISFORM.edtEmail.value)
oIP.AddPostKey("CustId", THISFORM.txtCustNo.Value)

lnResult = oIP.HTTPGetEx(;
"/foxisapi/foxisapi.dll/FiDemo.HTTPDemo.Process?Method=UpdateCust",;
@lcOutput,@lnSize)

*** Check for connection error
IF lnResult # 0
  WAIT WINDOW oIP.cErrorMsg NOWAIT
  RETURN
ENDIF

*** All is well
IF lcOutput = "OK"
   WAIT WINDOW NOWAIT "Customer Info Inserted..."
   RELEASE THISFORM
   RETURN
ENDIF

*** We have an error to display
MESSAGEBOX(lcOutput,48,"Update Error")
```

The key to this request is the AddPostKey() method calls, which are responsible for making the field values available to the Web server. The values are posted through the call to HTTPGetEx(), which also returns a result. Notice how HTTPGetEx() is called by passing in a buffer and size variable by reference:

```
lcOutput = ""
lnSize = 0
lnResult = oIP.HTTPGetEx("…foxisapi.dll…?Method=UpdateCust", @lcOutput,
@lnSize)
```

Here I assign a null string to the buffer, which lets HTTPGetEx() dynamically size the buffer. When HTTPGetEx() returns, it returns an error code (0 if all went well or else an API error). You can check the cErrorMsg property in that case to see the error, or use the

GetSystemErrorMessage() method to retrieve the raw API error. HTTPGetEx() also fills the buffer passed in (lcOutput) with the result data, similar to the way HTTPGet does. lnSize receives the size of the actual result string.

Once the result has been received, it's easy to check for OK or ERROR to see whether the request succeeded or failed. Note here that an update is happening in this request so no significant result is returned, but you could certainly have the Web server return a result data set or an HTML page. That's up to your application implementation.

On the FoxISAPI server side, the code to insert or update the record is also very straightforward (GetCust is listed here, too, for the sake of completeness):

```
********************************************************
* HTTPDemo :: GetCust
*********************************
FUNCTION GetCust

lcCustno = PADL(ALLTRIM(Request.QueryString("CustNo")),8)

lcFileName = Sys(2015) + ".dbf"
SELECT * From (DATAPATH + "TT_CUST") WHERE CustNo = lcCustNo ;
         INTO DBF (lcFileName)
USE
IF _TALLY = 0
   Response.Write("Error: Invalid Customer number" + lcCustNo)
   RETURN
ENDIF

loIP = CREATE("wwIPStuff")
lcFile = loIP.EncodeDBF(lcFileName,.T.)

Response.Write(lcFile)

ERASE lcFileName
ERASE ForceExt(lcFileName,"FPT")

RETURN

********************************************************
* HTTPDemo :: UpdateCust
*********************************
FUNCTION UpdateCust

lcCustId = Request.Form("CustId")


lcCompany = Request.Form("Company")
IF EMPTY(lcCompany)
   Response.Write("Error: Company Field cannot be blank…")
   RETURN
ENDIF


IF EMPTY(lcCustId)
   INSERT INTO (DATAPATH + "TT_CUST") ;
            (Company, careof, email, address, custno) ;
             VALUES ( lcCompany,;
                      Request.Form("Name"),;
                      Request.Form("Email"),;
                      Request.Form("Address"),;
                      SYS(3))
   Response.Write("OK - Insert")
ELSE
  IF !USED("TT_Cust")
     USE (DATAPATH + "TT_CUST") IN 0
  ENDIF
  SELECT TT_Cust
  LOCATE FOR custno = lcCustID
  IF FOUND()
     REPLACE Company WITH lcCompany,;
             Careof WITH  Request.Form("Name"),;
             Email WITH Request.Form("Email"),;
             Address WITH Request.Form("Address")
     Response.Write("OK - Update")
  ELSE
      Response.Write("Error: Customer not found. Not updated")
      RETURN
  ENDIF
ENDIF

Response.Write("OK")

RETURN
```

The update code checks to see if a customer ID was passed—if not, a new record is added with INSERT—otherwise the data updates an existing customer. Request.Form() is used to retrieve all form variables posted. This is really no different from how you would treat input from an HTML page.

There's no error handling here. In a real application you'd probably do some additional checks for valid data and empty field values before inserting them into the database. In this example, the error message should be sent back with a message like this:

```
Response.Write("Error: Invalid Fieldx Value")
```

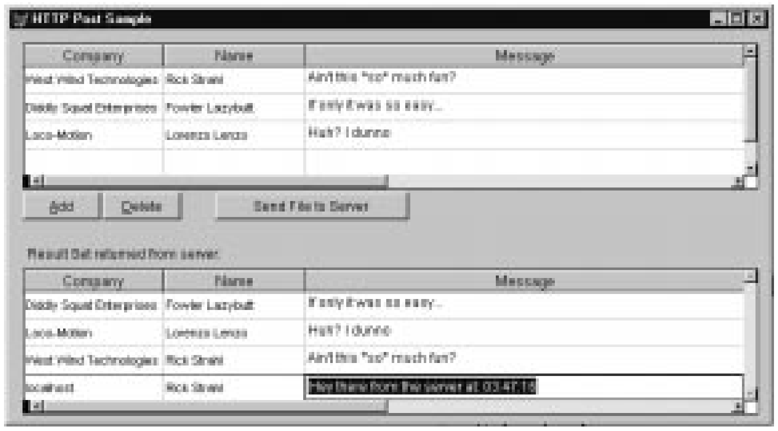
The client handles this with a MessageBox() if "ERROR" is returned as part of the response.

Hey, Mr. Postman, bring me some data

The previous example showed how you can use a Visual FoxPro client application to essentially mimic a browser without using an HTML interface. For many simple operations, or those that can run both on the Web and from within an application, this is probably fine but it requires a fair amount of code. Because you're using Visual FoxPro on both ends of the connection, it's much easier to deal with files passed back and forth. The advantage of this approach is that you can use Visual FoxPro's data binding features rather than having to deal with the data by hand on a field-by-field basis.

Figure 7.4 shows a form containing a grid into which you can type a company, name and message. The idea is that you can dynamically create the table to be sent to the server. Type some data into the grid, and then click Send File to Server to actually POST the data by sending the file as a POST variable called *CustFile*.

The server receives the data and decodes the file back into its DBF form. It then inserts a new record into the table ("Hey there from the server"), re-encodes the file, and sends it back to your VFP form on the client side, which then displays the result in the lower grid.



Company	Name	Message
West Wind Technologies	Rick Strahl	Ain't this "so" much fun?
Diddly Squat Enterprises	Fowler Lazybutt	If only it was so easy...
Loco-Motion	Lorenzo Lenzo	Huh? I dunno

Result Set returned from server:

Company	Name	Message
Diddly Squat Enterprises	Fowler Lazybutt	If only it was so easy...
Loco-Motion	Lorenzo Lenzo	Huh? I dunno
West Wind Technologies	Rick Strahl	Ain't this "so" much fun?
localhost	Rick Strahl	Hey there from the server at 03:47:10

***Figure 7.4**. Sending and receiving data files over HTTP is easy. This form allows entering of data to send to the server, which adds a record and then returns the full result for display in the lower grid.*

Here's the relevant code that goes into the Send button's Click event:

```
o = CREATE("wwIPStuff")

wait window nowait "Selecting data to send..."

*** Clear the result file and the result grid
SELECT TGetDownload
ZAP
THISFORM.Grid2.refresh

*** Select all items from the input cursor
SELECT Company, Name, Message FROM TPostTest ;
    ORDER BY Company, Name ;
    INTO DBF TEMPFILE
USE  && Must close before reading

WAIT WINDOW NOWAIT "Encoding data..."

*** Encode the file and memo
lcFileText = o.EncodeDBF("TempFile.dbf",.T.)

*** Create the Post Buffer
o.AddPostKey("CustFile",lcFileText)

*** Init vars that need to be passed by reference
lcBuffer = ""
lnSize = 0

WAIT WINDOW NOWAIT "Connecting to site..."

lnResult = o.HTTPConnect("localhost")
IF lnResult # 0
   WAIT WINDOW "HTTPConnect error: " + o.cErrorMsg
   RETURN
ENDIF

WAIT WINDOW NOWAIT "Sending data and retrieving result file..."
lnResult =
o.HTTPGetEx("foxisapi/foxisapi.dll/fidemo.httpdemo.process?method=SendCustList"
,;
            @lcBuffer,@lnSize)
IF lnResult # 0
   WAIT WINDOW "HTTPGetEx error: " + o.cErrorMsg
   RETURN
ENDIF

*** Decoding result file from server
lcFileText = o.DecodeDBF(lcBuffer,"TempFile.dbf")

SELECT TGetDownload
ZAP
APPEND FROM TempFile
GO BOTTOM
THISFORM.Grid2.refresh

ERASE TEMPFILE.DBF
ERASE TEMPFILE.FPT

WAIT CLEAR

o.HTTPClose()

RETURN
```

The key to this routine is the following lines of code:

```
*** Encode the file and memo
lcFileText = o.EncodeDBF("TempFile.dbf",.T.)

*** Create the Post Buffer
o.AddPostKey("CustFile",lcFileText)
```

The first line encodes the result from the SELECT statement into a string. The resulting string is then added to the POST request buffer. The file is now transferred to the server as a form variable when HTTPGetEx runs. Simple, huh? You could also send multiple files in a single pass using this mechanism. Simply assign each packaged file to a separate post variable with AddPostKey().

Once the server gets the request, it turns the file back into a string using Request.Form("CustFile") and then uses DecodeDBF() to turn the string back into a DBF/FPT file. Once the file is on disk, the server uses the table, adds a new record to it, then repackages the result with EncodeDBF() and simply sends it back to the client with Response.Write():

```
*********************************
* HTTPDemo :: SendCustList
*********************************
FUNCTION SendCustList

lcFileBuffer = Request.Form("CustFile")
```

```
o = CREATE("wwIPStuff")
IF !o.DecodeDBF(lcFileBuffer,"TTempFile.dbf")
   Response.Write("Error: Invalid File info")
   RETURN
ENDIF

USE TTempFile

INSERT INTO TTempFile(Company, Name, Message) ;
    VALUES (Request.GetServerName(),"Rick Strahl",;
            "Hey there from the server at: "+TIME())

*** Close the file and delete it!
USE In TTempFile

lcFileText = o.EncodeDBF("TTempFile.dbf",.T.)

Response.Write(lcFileText)

ERASE TTempFile.DBF
ERASE TTempFile.FPT

RETURN
ENDFUNC
```

The client then unpacks the file with DecodeDBF() and shows the result in the bottom grid.

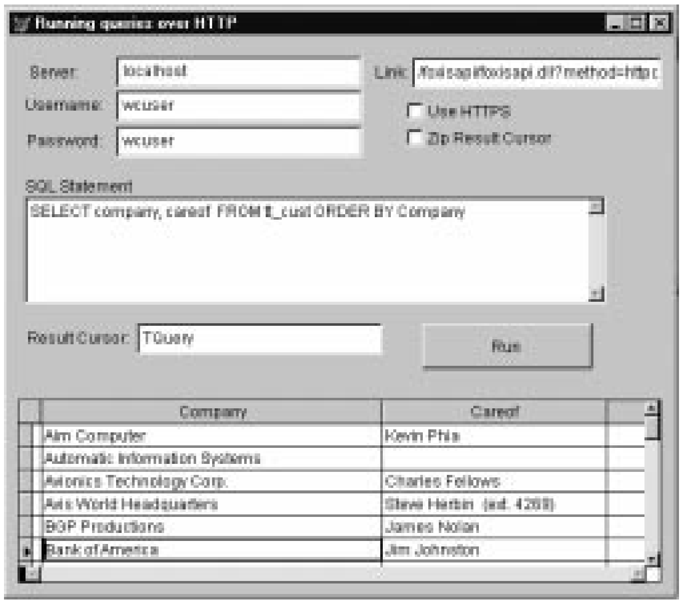
This is not a practical example, but it does demonstrate the whole range of using DBF files for messaging between client and server. Keep in mind that this code is simplified and lacks solid error handling, which I'll discuss further on.

Building in even more functionality!

The framework code I've described above reduces the overhead required to send and receive data files over HTTP to a few lines of code. You'll find that the physical aspects of data transfers are very easy to implement on both the client and server sides. The wwIPStuff class encapsulates so much of the logic that a few lines of code will do the trick. But let's practice our incremental development skills and take this approach one step further.

The wwIPStuff class library contains another class called wwHTTPData, which is derived from aHTTPData. This class wraps up much of the code involved in transferring files and running queries over the Web into a single class that can run a SQL statement over the Web with a single method call. To make this work you need to build two pieces that depend on each other: a server component and a client component, which are implemented as methods of the wwHTTPData class.

Figure 7.5 shows the HTTPSQL sample form. This form lets you run a SQL statement over HTTP against your Web server using various options, including running over HTTPS and zipping the result cursor for transfer.


Running queries over HTTP

Server: localhost
Username: wcuser
Password: wcuser
Link: /foxisapi/foxisapi.dll?method=http:c
☐ Use HTTPS
☐ Zip Result Cursor

SQL Statement
SELECT company, careof FROM tt_cust ORDER BY Company

Result Cursor: TQuery

Company	Careof
Aim Computer	Kevin Phia
Automatic Information Systems	
Avionics Technology Corp.	Charles Fellows
Avis World Headquarters	Steve Herbin (ext. 4268)
BGP Productions	James Nolan
Bank of America	Jim Johnston

Figure 7.5. *The wwHTTPData class allows running SQL statements over the Web from a client application. This form demonstrates most of the options available via class properties.*

I'll start with the client-side code. I want to achieve a SQLPassthrough-type syntax for using the class and method. So to run a query over HTTP the code looks like this:

```
o = CREATE("wwHTTPData")

o.cServername = THISFORM.txtServer.value
o.cHTTPLink = THISFORM.txtLink.value
o.cUserName = THISFORM.txtUserName.value
o.cPassWord = THISFORM.txtPassword.value
o.lIsZipped = THISFORM.chkZipping.value
o.lSecure = THISFORM.chkHTTPS.Value

o.cSQLCursor = TRIM(THISFORM.txtSQLCursor.Value)

*** Run the request!
o.Execute(THISFORM.edtSQL.Value)

IF o.lError
   MESSAGEBOX(o.cErrorMsg,48,THISFORM.Caption)
ELSE
   THISFORM.grdResult.RecordSource = THISFORM.txtSQLCursor.Value
ENDIF
```

The idea is to set a few properties on an object and then run SQL statements against it. The first group of property assignments configures the object more or less generically. Once the properties are set, you run multiple Execute calls. Execute itself takes a SQL statement as a parameter (or you can set the cSQL property and not pass a parameter at all) and is responsible for actually calling the server. Here's how it works:

```
*********************************
* wwHTTPData::Execute  && Client
*********************************
LPARAMETER lcSQL
LOCAL lnSize, lnBuffer, lnResult

THIS.lError = .F.

lcSQL = IIF(TYPE("lcSQL") = "C",lcSQL,THIS.cSQL)

THIS.lError = .F.

IF !INLIST(LOWER(lcSQL),"select","create")
   llNoResultSet = .T.
ELSE
   llNoResultSet = .F.
ENDIF

lnResult = THIS.oHTTP.HTTPConnect(THIS.cServerName, THIS.cUsername, ;
   THIS.cPassWord, THIS.lSecure)
IF lnResult # 0
   THIS.cErrorMsg = THIS.oHTTP.cErrorMsg
   THIS.nError = lnResult
   THIS.lError = .T.
   RETURN .F.
ENDIF

THIS.oHTTP.AddPostKey("txtSQL",lcSQL)
THIS.oHTTP.AddPostKey("txtMaxBufferSize",LTRIM(STR(THIS.nMaxBufferSize)))
IF THIS.lIsZipped
   THIS.oHTTP.AddPostKey("IsZipped","True")
ENDIF

lcbuffer = SPACE(THIS.nMaxBufferSize)
lnSize = THIS.nMaxBufferSize

lnResult = THIS.oHTTP.HTTPGetEx(THIS.cHTTPLink, @lcBUffer, @lnSize)
IF lnResult # 0
   THIS.cErrorMsg = THIS.oHTTP.cErrorMsg
   THIS.nError = lnResult
   THIS.lError = .T.
   RETURN .F.
ENDIF
IF llNoResultSet
   IF EMPTY(lcBuffer)
      RETURN .T.
   ENDIF
ELSE
   IF EMPTY(lcBuffer)
     THIS.cErrorMsg = "No data was returned from this request..."
     THIS.nError = -1
     THIS.lError = .T.
     RETURN .F.
   ENDIF
ENDIF
```

```
IF lcBuffer = "Error"
   THIS.cErrorMsg = lcBuffer
   THIS.nError = -1
   THIS.lError = .T.
   RETURN .F.
ENDIF

IF llNoResultSet
   RETURN .T.
ENDIF

*** Retrieve the file name from the buffer
lcFileName = FORCEEXT(ADDBS(SYS(2023)) + TRIM( SUBSTR(lcBuffer,6,40) ),"dbf")

IF !THIS.oHTTP.DecodeDbf( lcBUffer,
IIF(THIS.lIsZipped,ForceExt(lcFileName,"zip"),lcFileName) )
   THIS.cErrorMsg = "Error: Error Decoding the downloaded file"
   IF AT("401",lcBuffer) > 0 AND ATC("Unauthorized",lcBuffer) > 0
      THIS.cErrorMsg = "Error: Unauthorized access. Check username/password"
   ENDIF
   THIS.nError = -1
   THIS.lError = .T.
   RETURN .F.
ENDIF

IF THIS.lIsZipped
   IF THIS.oHTTP.UnzipFiles(ForceExt(lcFileName,"zip"), ADDBS(SYS(2023))  ) # 0
      ERASE (ForceExt(lcFileName,"*"))
     THIS.cErrorMsg = "Error: Unzipping failed. Most likely you selected too
many messages to download..."
     THIS.nError = -1
     THIS.lError = .T.
      RETURN .T.
   ENDIF
ENDIF

USE (lcFileName) ALIAS THTTPImport
SELECT * FROM THTTPImport WHERE .T. into cursor ( THIS.cSQLCursor )
USE IN THTTPImport
ERASE (forceext(lcFileName,"*"))

RETURN .T.
```

Because you're communicating with a server object of your own design, this code can make a number of assumptions about the data being returned. Thus the error messages are formatted so that errors are caught and assigned to the cErrorMsg property. Notice also that the code checks for empty output—it might actually be okay to not return anything when running a SQL command that doesn't return data, such as anINSERT. Execute handles all packaging and formatting of the SQL statement and unpacks the result cursor that comes back from the server. The result is converted into a cursor by reselecting the data into Cursor and then deleting the temporary download file.

The server component, called S_Execute(), is implemented as another method of the same object. S_Execute takes as input a Web Connection Process object, which can be easily emulated by a wwFoxIsapi class by adding oHTML and oCGI properties that map to Response and Request objects. The idea is that S_Execute() is a self-contained method that can retrieve the input parameters for the SQL statement and return the output of the result cursor directly into the HTTP output stream. S_Execute is called as follows from a wwFoxISAPI request:

```
*** At the Class Definition level (Required for wwHTTPData!!!)
oCGI=.NULL.
oHTML=.NULL.

************************************************************************
* httpDemo :: HTTPData
*********************************
*** Check for validation here
*** Authentication is optional
*** Supported values in the INI -
***  Any        -  Any Authorized user
***    Username   -  A specific user - must match Authenticated User
***   ""         -  None - no Authentication
lcAuthUser = "wcuser"  && THIS.GetAppIniVar("AuthUser","wwHTTPData")

THIS.oCGI = Request      && This code is required for wwFoxISAPI classes
THIS.oHTML = Response    && Add oCGI and oHTML as properties to your object

*** Create Data Object and call Server Side Execute method
***(wrapper for Process Method)
loData = CREATE("wwHTTPData")
loData.S_Execute(THIS, lcAuthUser)
```

Huh? Only five lines of code? Yup—S_Execute() acts as a fully self-contained request handler that sends output directly to the HTTP output stream so no additional Response.Write() calls are necessary.

Note the code to set a username for authentication. This is optional, but highly recommended to provide some added protection for your data. Because you can run SQL statements directly using this class, any VFP client application potentially has access to your server. The authentication can provide added protection to force users to log in prior to accessing data on the server. The request must send the authentication with the cUsername and cPassword properties. This information is then mapped against an NT username. If a username is not found, the command will not be executed. You can also use "", which means "Let everyone on," or "Any", which means "Allow any validated server user account."

You can see how the authentication is handled in S_Execute() among other things:

```
************************************************************************
* Server side Execute
*********************************
***  Function: Generic Execute request handler routine that can
***            be used to pull data from the wire.
***            Typically called from a wwHTTPData client in VFP
***    Assume: Requires Web Connection server request (loProcess)
***      Pass: loProess     -    wwProcess object
***            lcAuthUser   -    User name allowed access (user,"ANY","")
************************************************************************
LPARAMETER loProcess, lcAuthUser
LOCAL lcResultAlias, loEval, lcFileText, lnMaxLength, lcSQL, llUseZip, loHTML,
lcUserName
```

```
THIS.lError = .F.
THIS.cErrorMsg = ""

IF !THIS.Authorize(lcAuthUser, loProcess)
   RETURN
ENDIF

loHTML = loProcess.oHTML

lcFullSQL = loProcess.oCGI.GetFormVar("txtSQL")
lcSQL = LOWER(lcFullSQL,10))
llUseZip = !EMPTY(loProcess.oCGI.GetFormVar("IsZipped"))
lnMaxLength = VAL(loProcess.oCGI.GetFormVar("txtMaxBufferSize"))

IF EMPTY(lcSQL)
   loHTML.Send("Error: No SQL statement to process.")
   RETURN
ENDIF

lcOrigAlias = "wwd_" + SYS(2015)
lcOrigFileName = ADDBS(SYS(2023)) + lcOrigAlias

IF lcSQL = "select"
   lcFullSQL = lcFullSQL + " INTO TABLE " + lcOrigFilename
ENDIF

IF lcSQL # "select" AND lcSQL # "insert" AND lcSQL # "update" AND ;
   lcSQL # "delete" AND lcSQL # "create"
   loHTML.Send("Error: Only SQL commands are allowed.")
   RETURN
ENDIF

*** Turn off user interface functions so any error dialogs
*** will cause an error in the wwEval::Execute() call
*** In particular this will avoid File Open Dialogs
#IF  WWVFPVERSION > 5
  SYS(2335,0)
#ENDIF

loEval = CREATE("wwEval")
loEval.Execute(lcFullSQL)
IF loEval.lError
   loHTML.Send("Error: SQL statement caused an error." + CHR(13) + lcFullSQL)
   RETURN
ENDIF

#IF  WWVFPVERSION > 5
  SYS(2335,1)
#ENDIF

IF lcSQL # "select" AND lcSQL # "create"
   *** If no cursor is returned nothing needs to be returned
   RETURN
ENDIF

*** Otherwise encode the result file
lcResultAlias = Alias()

lcFileName = ADDBS(SYS(2023)) + "wwd_" + SYS(3) + ".dbf"
```

```
*** Now select the result into another cursor that we know by name
*** (CREATED TABLES WE MAY NOT)
SELECT * FROM (lcOrigAlias) INTO DBF (lcFileName)
USE
USE IN (lcResultAlias)

loIP = CREATE("wwIPStuff")

IF llUseZip
   IF loIP.ZipFiles(ForceExt(lcFilename,"zip"),;
                    ForceExt(lcFileName,"*"),9 ) # 0
      Response.Send("Error - unable to zip the file")
      RETURN
   ENDIF
   lcFileText = loIP.EncodeDBF(ForceExt(lcFilename,"zip") )
ELSE
   lcFileText = loIP.EncodeDBF(lcFileName,.T.)
ENDIF

IF EMPTY(lcFileText)
   loProcess.oHTML.Send("Error: File not encoded.")
   ERASE (ForceExt(lcFilename,"*"))
   ERASE (ForceExt(lcOrigFilename,"*"))
   RETURN
ENDIF

IF LEN(lcFileText) >= lnMaxLength
   loProcess.oHTML.Send("Error File is too large to send.")
   ERASE (ForceExt(lcFilename,"*"))
   ERASE (ForceExt(lcOrigFilename,"*"))
   RETURN
ENDIF

loProcess.oHTML.Send(lcFileText)

ERASE (ForceExt(lcFilename,"*"))
ERASE (ForceExt(lcOrigFilename,"*"))
```

The majority of this code deals with formatting and running the SQL statement. The wwEval class is used to safely execute the SQL statement. If an error occurs, the wwEval class's Error method captures the error and passes it back to the calling code without causing any other error handlers to kick off and bail out of the code or generate a runtime error. Also, note the call to SYS(2335), which turns off all user interface functions in a COM object. If a UI operation like a File Open dialog occurs, an error is generated rather than bringing up the dreaded File Open dialog, which could lock your server. This is new in VFP 6.0 and allows getting around some nasty hangup problems that previously couldn't be captured.

The query runs and is dumped to a predetermined temp file. The file is encoded with EncodeDBF() and, if okay, it is sent down the HTTP pipe. The result table can optionally be zipped on the fly (if the third-party DynaZip DLLs are available on the system). Because some commands won't return a result set, their code bypasses all the packaging and they simply return without result output.

Now a word of warning! This class can be very dangerous if not used in a controlled application environment. Because you have full access to the data on your server, a single

`DELETE FROM <your favorite table here>` can wipe out all data in a single stroke. To protect yourself, you have two lines of defense: Hide the URL that you're hitting in your code to make the link non-obvious, and require authentication before allowing access even through your application. But even with these precautions in place, somebody in the know, like a disgruntled employee, might still have access to the data directly. Consider these issues very carefully before you generically implement a SQL request handler. The safest solution is to use embedded code that wraps the request in question with program logic—check the URLs that requests are coming from if necessary, or explicitly ask for authentication from users.

Putting it all together

You now have all the pieces to build an application that can run over the Internet as a client/server application using a totally open, non-proprietary protocol that can access your remote server from any Internet connection. You have full flexibility at the protocol level, and best of all, you can do it all with Visual FoxPro on both the client and server. Think of the flexibility you get with this approach! Whether you want to build offline applications like message readers, or applications that have a more sophisticated front end than HTML can provide, this simple client/server architecture makes it possible to build distributed applications with the tools you already know—and with very little code.

To review, a typical application that runs over HTTP can take advantage of the HTTP connections in the following ways:

- Sending "commands" via the URL's command line to trigger some operation on the server. For example: foxisapi.dll?method=ShowCustData&UserId= DA1111, where DA1111 might be a parameter sent to the server to let it know to work on Customer ID DA1111.
- Retrieving data *from* an HTTP link via HTTPGet() or HTTPGetEx().
- Sending data *to* the server via POSTing data using HTTPGetEx(). Remember that HTTPGetEx() can do both POST and GET operations on a single request, as demonstrated in the last example.
- Simulating a Web browser with a VFP front-end application by simply posting variables to a server.
- Transferring raw data over HTTP and running full-blown SQL statements across the wire if both ends are running Visual FoxPro.

Let's come back to the message board example on my Web site I cited at the beginning of this chapter. My site hosts a message board that is used to post messages for support of Web Connection and general Web programming issues. People access the Web site to view information online, but I recently built an offline reader (see Figure 7.6a) that allows these visitors to use a local VFP application to view that same data. Rather than browsing the Web site, the users can download messages via HTTP and merge them into an existing local file. At the same time, any messages that users want to post can be sent up to the server and merged into the online file for updating. Messages are posted and visible on the Web site as well as available for download.

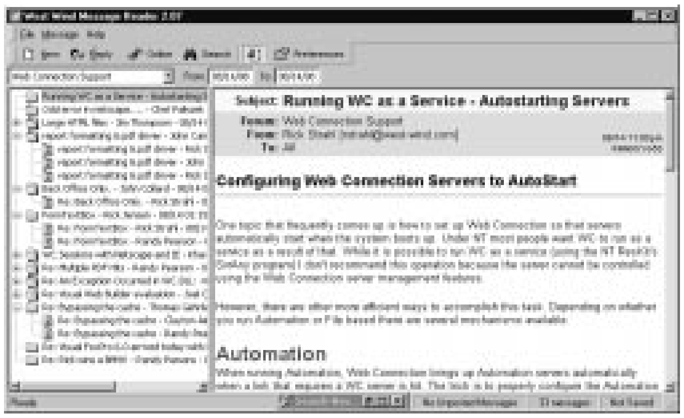
The motivation for this operation is clear: You can build a vastly more efficient UI with a VFP application than on the Web, and the access to the data is much faster. A onetime download, which usually takes only a few seconds, can bring down the data for the entire day immediately, while viewing the messages online can take much more time depending on your connection. Rather than downloading new HTML every time, you load the message content only once—and in compressed format to boot.

In addition to the speed, the client has a lot more flexibility with the data. Users can see unread messages because a local file can flag messages as read. This is impossible online because there's no easy way to attach user information to the online message file, such as which users have read which messages. Locally, it's a simple True/False flag that you can easily query and display with a special image in the Treeview. **Figure 7.6a** (top) shows the GUI version, while **Figure 7.6b** (bottom) shows the Web version.

Here's a look at some key code elements of this application. The following code is a real-world example including error checking, showing how you can put this technology to work:

```
* Form method that handles file downloads
* Pass qualified WHERE clause for the SELECT statement
* Filter must be time zone adjusted (in Download form)
FUNCTION DownLoadMessages
LPARAMETER lcDownLoadFilter

loIP = CREATE("wwIPStuff")

*** Add the Download Filter as a POST key
loIP.AddPostKey("Filter",lcDownLoadFilter)

THISFORM.StatusMessage("Downloading Messages...",,1)

lnResult = loIP.HTTPConnect(wwt_cfg.server)
IF lnResult # 0
   THISFORM.StatusMessage("Error: " + loIP.cErrorMsg)
   RETURN -1
ENDIF

*** Presize the result buffer – only want 500k at a time
lcBUffer = SPACE(500000)
lnSize = LEN(lcBUffer)
lnStartTime = SECONDS()

lnResult = loIP.HTTPGetEx(;
      "/wconnect/wc.dll?wwthreads~Downloadmessages",;
      @lcBUffer,@lnSize)
IF lnResult # 0
   THISFORM.StatusMessage("Error: " + loIP.cErrorMsg)
   RETURN -1
ENDIF

IF lcBUffer = "ERROR - No Records"
   THISFORM.StatusMessage("No messages to download")
   RETURN -1
ENDIF
```

```
IF EMPTY(lcBUffer)
   THISFORM.StatusMessage(;
     "Error: No data was returned by the server...")
   RETURN -1
ENDIF

IF !loIP.DecodeDbf(lcBUffer, "TImport.dbf")
   THIS.StatusMessage("Error: File Import failed. "+;
                      "Too many messages!")
   RETURN -1
ENDIF

*** All went well – Now import the messages
lnReccount = 0

SELE 0
USE TImport
SCAN
   SELECT wwThreads
   LOCATE FOR Msgid = TImport.Msgid
   IF !FOUND()
      SELECT TImport
      SCATTER MEMO MEMVAR
      SELECT wwThreads
      APPEND BLANK
      GATHER MEMO MEMVAR
      lnReccount = lnReccount + 1
   ENDIF
ENDSCAN

USE IN TImport
ERASE TImport.*
*** Refresh the form with the new data
THISFORM.BuildTree()
THISFORM.StatusMessage("Downloaded " + LTRIM(STR(lnReccount)) + ;
  " message(s) in " + STR(SECONDS() - lnStartTime,2) + " seconds")

RETURN lnReccount
```



Subject: **Running WC as a Service - Autostarting Servers**

Forum: Web Connection Support
From: Rick Strahl [rstrahl@west-wind.com]
To: All

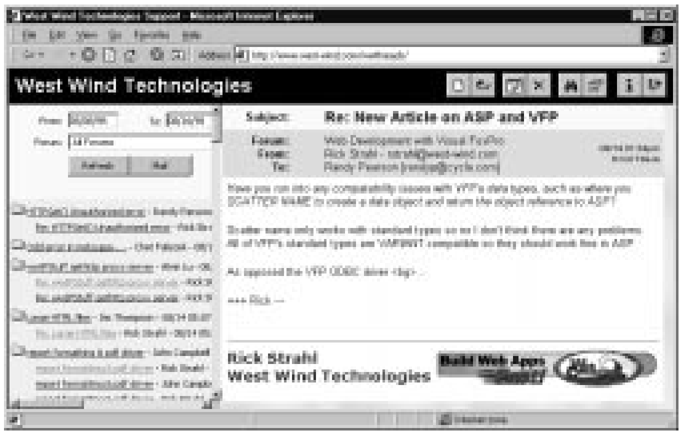
Configuring Web Connection Servers to AutoStart

One topic that frequently comes up is how to set up Web Connection so that servers automatically start when the system boots up. Under NT most people want WC to run as a service as a result of that. While it is possible to run WC as a service (using the NT Resource Kit's SrvAny program) I don't recommend this operation because the server cannot be controlled using the Web Connection server management features.

However, there are other more efficient ways to accomplish this task. Depending on whether you run Automation or File based there are several mechanisms available.

Automation

When running Automation, Web Connection brings up Automation servers automatically when a link that requires a WC server is hit. The trick is to properly configure the Automation



West Wind Technologies

Subject: **Re: New Article on ASP and VFP**

Forum: Web Development with Visual FoxPro
From: Rick Strahl - rstrahl@west-wind.com
To: Randy Pearson [randyp@cycla.com]

Have you run into any compatability issues with VFP's data types, such as where you SCATTER NAME to create a data object and return the object reference to ASP?

Scatter name only works with standard types so no I don't think there are any problems. All of VFP's standard types are VARIANT compatible so they should work fine in ASP.

As opposed the VFP ODBC driver <bg> ...

+++ Rick ---

Rick Strahl
West Wind Technologies

Figures 7.6a and 7.6b*. Which user interface do you think is easier to use? The rich GUI version (top) allows the message views to be configurable to the user so he can see unread messages and threads as flagged. The Web version (bottom) is generic and not customized for each user.*

There are a couple of notable issues here. This code works by sending a request to the server with a POST variable called FILTER, which is a complete WHERE clause to a SELECT statement. Typically the filter contains a date range that is adjusted for time zones, but this code can also be used from a search dialog, simply by passing the appropriate search parameters in the filter expression to provide a Web-based search. This makes this method reusable for all download operations required by the application.

Note that the HTTP download buffer is allocated to a whopping 500K in order to make sure that I can capture a reasonably large file. Even so, 500K will only allow retrieval of approximately three weeks' worth of data—anything more and the download will fail. Because I have a rough idea of traffic, I give the user a warning dialog if he's trying to download more than three weeks of data at once.

Next, notice all the error handling. With these file downloads it's extremely important to check every call that is returned from WinInet. If an error occurs, back out gracefully with a message to the user. You need to decide on the server end what to return in case of an error. In this case the request returns a file, so the code checks for an empty string and then for proper size as specified by the encoding header (this happens in DecodeDBF()).

On the Web Connection server end (almost identical to the wwFoxISAPI code) the code looks like this:

```
* HTTPProcess :: DownloadMessages
FUNCTION DownLoadMessages
LOCAL lcFilter, lcToDate, lcFromDate, lcForum

lcFilter = Request.GetFormVar("Filter")

IF EMPTY(lcFilter)
   lcForum = Request.GetFormVar("Forum")
   lcFromDate = Request.GetFormVar("FromDate")
   lcToDate = Request.GetFormVar("ToDate")

   lcFilter = "Forum='" + PADR(lcForum,30) +"' AND " +;
            "timestamp >= {" + lcFromDate +"} AND "+;
            "timestamp <= {" + lcToDate + "} + 1"
ENDIF

lcFile = SYS(3)
SELECT ThreadId, Msgid, Subject, Message, FromName, FromEmail, To,Forum,;
       TimeStamp ;
       FROM (DATAPATH + "wwThreads") ;
       WHERE &lcFilter ;
       INTO DBF (lcFile)

IF _TALLY < 1
   Response.Send("ERROR - No Records")
   USE
   ERASE (lcFile + ".*")
   RETURN
ENDIF
USE

loIP = CREATE("wwIPStuff")

lcFileText = loIP.EncodeDBF(lcFile + ".dbf",.T.)

IF EMPTY(lcFileText)
   Response.Send("ERROR - File not encoded.")
   ERASE (lcFile + ".*")
   RETURN
ENDIF

IF LEN(lcFileText) >= 500000
   Response.Send("ERROR - File is too large to send.")
   ERASE (lcFile + ".*")
   RETURN
ENDIF

*** This is the actual FILE Send operation!
Response.Send(lcFileText)

ERASE (lcFile + ".*")
```

```
ENDFUNC
* DownLoadMsgs
```

Note the sending of the error messages in the appropriate places. The error messages can be retrieved on the client side and can provide meaningful display in the status bar of the reader. Note, in particular, the size check for the 500K response size. If the encoded file is greater than 500K, a message is sent back rather than attempting (and failing) to send a file that's too large. The client and server sides need to agree on these sizes for this to work properly.

For good measure, here's the client code for uploading messages to the server:

```
FUNCTION UploadMessages

LoIP = CREATE("wwIPStuff")

THISFORM.StatusMessage("Uploading Messages...",,1)

SELECT * FROM wwThreads ;
   WHERE Post AND !DELETED() ;
   INTO DBF TExport

IF _TALLY < 1
   USE
   ERASE TEXPORT.DBF
   ERASE TEXPORT.FPT
   THISFORM.StatusMessage("No messages to upload...")
   LoAPI = CREATE("wwAPI")
   loAPI.Sleep(2000)
ENDIF

THISFORM.StatusMessage("Uploading " + LTRIM(STR(_Tally)) + " Messages...",,1)

USE
lcFileText = loIP.EncodeDBF("TExport.dbf",.T.)

ERASE TEXPORT.DBF
ERASE TEXPORT.FPT



IF EMPTY(lcFileText)
   THISFORM.StatusMessage("Invalid File Info - not uploaded")
   RETURN
ENDIF

loIP.AddPostKey("FileText",lcFileText)

lnResult = loIP.HTTPConnect(wwt_cfg.server)
IF lnResult # 0
   THISFORM.StatusMessage("Error: "+loIP.cErrorMsg)
   RETURN
ENDIF

lcBUffer = SPACE(500000)
lnSize = LEN(lcBuffer)
```

```
lnResult = loIP.HTTPGetEx("/wconnect/wc.dll?wwthreads~UploadMessages",;
                          @lcBuffer,@lnSize)

IF lnResult # 0
   THISFORM.StatusMessage("Error: "+loIP.cErrorMsg)
   RETURN
ENDIF

*** Must check if the Upload went Ok
if lcBuffer # "OK"
   *** No – don't delete messages to post
   THISFORM.StatusMessage("File Upload Failed")
   RETURN
ENDIF

THISFORM.StatusMessage("Deleting Posted Messages...",,1)
DELETE FROM wwThreads WHERE Post

THISFORM.StatusMessage()

RETURN
```

Here the result from the POST operation simply returns OK or an error code, which is irrelevant—it either worked or it didn't.

Don't forget about security!

Sending data over the wire, especially over an open Internet connection, can be dangerous. Remember that it's possible to intercept the data traveling over the Net with a packet analyzer and potentially hijack sensitive information. Your first line of defense is to use HTTPS (Secure HTTP or SSL) to transmit your data to and from the Web server. HTTPS requires a secure certificate on the Web server (see Chapter 10's Security section). Once installed, all communication using HTTPS is encrypted. Unfortunately, the encryption process noticeably slows down communications and you'll want to use it only on requests where security is really required—all others should continue non-secure. The wwIPStuff class supports HTTPS using the low-level HTTP functions by specifying the fourth parameter of .T. for HTTPConnect() or the lSecure property to establish a secure link:

```
FUNCTION HTTPConnect
LPARAMETER lcServer, lcUsername, lcPassword, llSecure
```

Another simple way to protect yourself from unauthorized access is to use passwords. WinInet supports security via standard Web-based *Basic Authentication* or *NT Challenge Response* that uses NT domain security for permissions. You can connect to the server using a specific security context, which exists only while connected to the server over the HTTP connection. wwIPStuff supports this with the second and third parameters to the call to HTTPConnect(). With wwFoxISAPI, you can force a request to password-validate a user with the following code:

```
*** See whether user is already Authenticated
lcUser = Request.GetAuthenticatedUser()
```

```
IF EMPTY(lcUser)
   *** Nope – force Login dialog
   Response. Authenticate()
   RETURN
ENDIF
```

Authenticate() is a method in the FoxISAPI wwResponse class that requests authentication from the Web server via an HTTP header that is returned as a result. The HTTP result looks like this:

```
HTTP/1.0 401 Not Authorized
WWW-Authenticate: basic realm = "localhost"

<HTML><h2>Gotta enter your password to get in!</h2></HTML>
```

WinInet supports navigating this request and sending the username/password to the server and essentially logging the user in. The server request runs again, and this time the user is authenticated and the request can proceed. The client end is automatic with the call to HTTPConnect(). On the server you have to implement the above code to enforce the authentication check.

Another security issue to keep in mind is that links that download data can also be accessed by a browser directly. Although users will likely never see the actual link that is called, it's possible to access the same data link you might use to retrieve data directly via the Location URL line in the browser if a user can guess the URL. Nothing like somebody figuring out your file upload link, and sending you continuous uploads that will fill up and crash your server's hard disk. Consider your URL names carefully and use authentication where applicable to avoid these problems.

Note that both Secure HTTP and authentication don't work with the plain HTTPGet. If you only want to retrieve data securely, you have to use HTTPConnect() and HTTPGetEx().

What about other server tools?

I've used FoxISAPI as the Web server back end in the previous examples. For these examples to work, you need to use the Foxisapi.dll provided with the Developer's Download Files at www.hentzenwerke.com because the update fixes problems with sending binary output to the client. The wwIPStuff code originated from the Web Connection libraries, so it also works safely with Web Connection. If you use another tool such as Active Server Pages, you need to make a few adjustments. I haven't checked out tools like FoxWeb or X-Works—check with the authors to determine whether they support binary output.

With Active Server Pages, the problem is more complicated. VBScript uses double-byte Variants internally, which can cause several problems with binary data. Variant strings are null terminated so when you send a binary string to output they terminate at the NULL. You can get around this problem by using the Response object's BinaryWrite method instead of the basic Write. This method outputs any text string (or Byte Array as the docs call it) as-is, without first converting it to VB's double-byte, wide-character formatting. This will work fine if you create your binary data inside the VBScript code or with one of the built-in objects.

Unfortunately, I've been unable to send binary data created inside Visual FoxPro through an Active Server Page. The problem is that ASP converts the result from a VFP COM object into a Variant, which truncates the result at any NULLs that are encountered. Because VFP cannot return a typed string result (as you can from C, Delphi or VB-created COM objects), the result string is always a double-byte Variant. Sending this Variant with BinaryWrite results in actually sending two bytes per character back to the client, while Write will truncate at NULLs. This limitation means you have to confine yourself to sending non-binary data (like comma-delimited strings or URLEncoded strings) from any VFP COM object.

As a workaround you can use a VFP COM object to send the output. You can pass in the Response object to the method, then use BinaryWrite() from within the Fox object to send the output:

```
Function ASPResponseMethod
LPARAMETER loResponse

lcBinary = FunctionThatReturnsBinary()
loResponse.BinaryWrite(lcBinary)
RETURN
```

This is the only way I've been able to pass binary data back through the ASP engine from VFP. Sorry, but ASP is not a good platform for HTTP messaging with VFP COM objects.

WinInet issues

You should be aware of some issues related to using WinInet's HTTP functions in the context of creating HTTP transfer operations.

Although the wwIPStuff class contains support for connection, receive, and send timeouts (*WinInetSetTimeout()*), a request that hangs in the middle of a connection will not time out according to these values. Instead, a Windows Sockets system default (typically 30-40 seconds) is applied by WinInet. This can be a problem because the WinInet functions don't provide any feedback while in process, and the user might think the request hung. Microsoft calls this "by design" because Web servers typically handle timeouts based on the Web server specified timeout value to allow large requests to complete. I suggest you carefully test your requests under various connection environments to see exactly how your application might be affected by lost or unavailable connections.

Sending huge files in either direction is probably not a good idea unless you have a speedy Internet connection. Bandwidth is always critical, but you also need to think of the system limitations. The process of URLEncoding a string to be sent to the server is slow, especially when using FoxPro code to do the encoding. This is why the online version switches to DLL functions for faster encoding operation. The latest version of wwIPStuff also supports a mechanism of sending data in Multipart Form format, which requires no encoding. This is a brand-new feature which you can read up on in the provided documentation for wwIPStuff. This mechanism can drastically reduce the size of uploaded form data.

Web servers also have a timeout—if you're sending a huge amount of data to the server over a slow link, the server might disconnect you when it's timeout is up, even though data might still be transferring.

The examples in this chapter use fully synchronous access to the Web server, which means you connect, start downloading, and then wait for completion of the request without any feedback. The latest version of wwIPStuff provides an asynchronous version of the HTTP functions that can help, but it's a bit more work to get the result data because you now have to deal with events for download completion, errors, and so on. wwIPStuff implements some workarounds by spying the HTTP header and then using event hooks to let your code know when a new chunk is retrieved, but it's beyond the scope of this discussion. You can examine the cHTTPHeader property and the OnHTTPBufferUpdate() method in the wwIPStuff documentation for more information about how to retrieve status information while transferring data.

In addition to data size, keep in mind that when *sending* data to the server, the URLEncoding process also makes your data much larger—up to three times as big as the original (%OD, for example, for a Carriage Return (Chr(13)). You can get around the size limitations and feedback issues by "chunking" your downloads—break them into smaller files and then request the remaining files successively. It requires some logic and state-keeping to work efficiently.

You can also get some size relief by using a third-party zip control such as DynaZip. DBF data is a prime candidate for compression, yielding 80 percent compression or better for typical VFP tables. The message board uses DynaZip to perform compression, which has resulted in tremendous improvements in transfer times (for example, bringing down 200+ messages over 28.8k in under a minute). But zipped files also tend to be binary content, so although the data might get 80 percent smaller, it becomes more prone to require extensive encoding, which might boost the size back up.

Unrelated to file transfers, I've had a few problems with WinInet when accessing a lot of *different* sites in quick succession. For example, using wwIPStuff to build a Web crawler or to verify site links, you might fire off various requests in quick succession. WinInet handles some operations in the background on separate execution threads. Sometimes when quickly connecting and disconnecting from different sites, some threads don't clean up properly, leaking handles. Furthermore, accessing a link that redirects to another page causes WinInet to leak three handles because the connection is never properly closed—regardless of whether you release the IP handles. Make sure you check typical operation with WinInet to see whether your code is affected by these problems. I expect these to get fixed by Microsoft as WinInet use becomes more prevalent. Neither of these issues should be a problem if you're implementing applications that always connect to the same server.

Summary

Realize that this architecture is *not* meant to replace HTML front-end applications. By using this approach you are requiring all clients to have Win95/NT and the Visual FoxPro runtime, so all the cross-platform and thin-client advantages that HTML brings to the party don't apply. Plain and simple—you're dealing with Fat Client technology by running VFP on the client. But you do gain the ability to take advantage of the Web's distributed environment to make your application reach out and communicate with users everywhere.

If you're building corporate or shrink-wrapped applications that need to communicate from widely spread locations, this is an easy way to link applications to a home site. Whether you're pulling occasional data updates or querying real-time data over the Web, you get the distributed aspect of Web applications without having to bite the HTML bullet.

The big benefit is that you can take advantage of VFP's rich UI features to provide a productive work environment and still provide the distributed, plug-in-from-anywhere connectivity. I use this front end for a number of maintenance operations, from downloading orders from my Web site directly into my Point of Sale application, to checking my error logs on various sites by downloading them to the local machine. In all cases, this VFP front end is only an extension to existing Web applications that are running a full HTML interface. The possibilities are endless, and many companies and popular software packages are already taking advantage of this type of interface. You can do the same from your Visual FoxPro applications!

Pros

- Use a rich user interface instead of limited HTML.
- Use VFP data locally, including data-binding.
- Continue to take advantage of VFP's data access and language features on both ends of the connection.
- Full application environment—not a limited "script-safe" environment that a browser requires.
- Can be used as a data access mechanism for Active Document applications.

Cons

- Fat Client—Visual FoxPro required on the client.
- Potential security issues.