Searching With Web Services

Doug Hennig

Web Services make it easy to access functions on other computers. Several Web sites now provide a Web Service to search their archives for text. However, every Web Service has a different interface and returns results in a different format. This month, Doug shows how to create a wrapper to provide a consistent way to search for the same text from many Web Services.

Web Services are going to change the way applications are deployed. Rather than monolithic apps sitting on workstations or servers, functionality is going to be pushed out to Web servers. For example, an order entry application could submit a shipping order to UPS via one Web Service, and track the status of the shipment with another Web Service. Although you could do that before using the UPS Web site, you had to do HTML scraping to make the results understandable to the application. With Web Services, it's like calling a function on your local machine, except the function actually sits on someone else's server somewhere on the Internet. While security and availability are currently concerns, I believe these problems will be solved over the long run.

One of the cool things about VFP 7 is that it makes it easy to create your own Web Services. First, you have to install the Microsoft SOAP Toolkit, which comes on the VFP 7 CD. Then, you simply create a COM DLL (mark one of the classes in your project as OLEPUBLIC and build a DLL) and invoke the Web Services Wizard (available from the Tools, Wizards menu item). The wizard will generate the files necessary to deploy the Web Service, including a WSDL file that describes the interface of your class. It'll even create an IIS virtual directory and configure it for you. And when you rebuild your DLL after making changes, a project hook will automatically update the Web Service files. It doesn't get much easier than that! For details on creating your own Web Services, see Whil Hentzen's articles in the February and March 2002 issues of FoxTalk.

So far, the majority of the publicly available VFP-based Web Services are search engines for Web sites that contain content related to VFP:

- ProFox (http://www.leafe.com): Ed Leafe's email-based message board for FoxPro developers. The WSDL file is located at http://www.leafe.com/profoxws.wsdl.
- FoxWiki (http://fox.wikis.com): Steven Black created this incredible resource, which provides thousands of documents on every conceivable subject. Information about the Web Service is available at http://fox.wikis.com/wc.dll?Wiki~WikiWebServices and you can find the WSDL file at http://fox.wikis.com/wikiwebservice.wsdl.
- Universal Thread (http://www.universalthread.com): This forum, created by Michel Fournier, is where thousands of VFP developers hang out every day, sharing ideas and files. For documentation on the Web Service, see http://www.universalthread.com/WebService.asp; the WSDL file is located at http://www.universalthread.com/universalthread.wsdl.

In addition to these VFP-related sites, Google recently released a Web Service to search anywhere on the Internet. Their implementation is limited and kind of goofy; thanks to Hector Correa for posting some sample code on the FoxWiki showing how to access it in VFP (http://fox.wikis.com/wc.dll?Wiki~GoogleWebServiceUsingVFP~VFP). If you want to use the Google Web Service, you have to sign up for the service (there's no charge) and download the WSDL file (part of the documentation and samples found at http://www.google.com/apis)

Some Web Services are easier to use than others. For example, searching the FoxWiki for all references to "Stonefield" is easy:

```
loWS     = newobject('WSClient', ;
  home() + 'FFC\_WebServices.vcx')
loSearch = loWS.SetupClient('http://fox.wikis.com/' + ;
  'wikiwebservice.wsdl', '', '')
lcXML    = loSearch.GetTextSearch('Stonefield')
xmltocursor(lcXML)
browse
```

This code uses the WSClient class that comes with VFP and provides a wrapper for the MSSOAP.SOAPClient COM object, which provides access to a Web Service. The SetupClient method expects to be passed the URL for a WSDL file, and it create a "proxy" object, one that has the same interface as the Web Service so you can do what looks like local calls. However, behind the scenes, when you call a method of the proxy object, the call is packaged up into a SOAP wrapper, which is passed to the Web Service. (For more details on how Web Services work behind the scenes, search the MSDN Web site (http://msdn.microsoft.com) for "SOAP" or "Web Services".) In this case, we're calling the GetTextSearch method of the Web Service on the FoxWiki Web site, the return value of which is a VFP cursor packaged as XML. So, we can use XMLTOCURSOR() to convert it back to a cursor and browse the results.

ProFox is just as easy, but note the different method name and parameters (it requires a date range), and the results are formatted differently:

```
loWS     = newobject('WSClient', ;
  home() + 'FFC\_WebServices.vcx')
loSearch = loWS.SetupClient('http://www.leafe.com/' + ;
  'profoxws.wsdl', '', '')
lcXML    = loSearch.SearchTextLinks('Stonefield, ;
  {^2002-01-01}, date(), .F.)
xmltocursor(lcXML)
browse
```

Both the Universal Thread and Google are considerably more complex. The Universal Thread requires you to login and it returns a unique ID value that you must pass on each method call. However, you don't pass it as a parameter: it must be embedded in the SOAP header (see the documentation for this Web Service for details). Google doesn't require a login; you pass a unique ID assigned to you when you sign up for the service on every method call. However, it returns an array of results that you have to dig the content out of, so you can't just use a simple XMLTOCURSOR() call.

Because every service is different, both in terms of how you call it and what the results look like, researching the solution to a problem by searching multiple services can be a pain. To make it easier, I created a wrapper for this.

Data-Driving the Searches

I originally created a hard-coded wrapper program to do Web Service searching, but once I added code to search more than one Web Service, I realized that it would be better to data-drive the process. In other words, a table contains information about each of the Web Services to search and some generic code processes that table. This way, adding another Web Service to the search is as simple as adding a new record to the table.

Because each Web Service has a different way to call it and returns different results, I decided to put the specific code needed for each Web Service into a memo field in the table. This code is executed using the new EXECSCRIPT() function in VFP 7.

The table that defines the Web Services to search and how to access them is called WSSEARCH.DBF; its structure is shown in Table 1.

Field Name	Type/Size	Description
ORDER	N(2, 0)	The order in which to process the records.
ACTIVE	L	.T. if this Web Service should be searched (you can set it to .F. to temporarily disable searching a Web Service that you know is unavailable).
NAME	C(30)	The descriptive name of the Web Service.
WSDL	M	The URL for the Web Service's WSDL file.
SEARCH	M	The code to call the Web Service's search method. This code is passed the Web Service proxy object, the search string, the start and end dates, and the contents of the USERNAME and PASSWORD fields. It should return either an empty string or the results in an XML string that will create a cursor with TITLE, URL, and CONTENTS columns.
USERNAME	C(10)	The user name to access the Web Service.
PASSWORD	C(10)	The password.

Table 1. The structure of the WSSEARCH table.

Here's an example of a WSSEARCH record. The FoxWiki Web Services record has http://fox.wikis.com/wikiwebservice.wsdl for WSDL, blank values in USERNAME and PASSWORD, and the following code in SEARCH:

```
lparameters toWS, tcSearch, tdBegin, tdEnd, tcUserName, ;
  tcPassword
local lcXML
lcXML = toWS.GetTextSearch(tcSearch)
if not empty(lcXML)
  xmltocursor(lcXML, '_WikiResults')
  if reccount() = 0
    lcXML = ''
  else
    select NAME as TITLE, LINK as URL, '' as CONTENTS ;
      from _WikiResults into cursor _Results
    cursortoxml('_Results', 'lcXML', 1, 0, 0, '1')
    use in _Results
  endif reccount() = 0
  use in _WikiResults
endif not empty(lcXML)
return lcXML
```

This code calls the GetTextSearch method of the Web Service, passing it the text to search for. Note that the start and end dates and the user name and password aren't used by this code, since the FoxWiki Web Service doesn't need anything except the search string. If the return value from GetTextSearch isn't empty, the code uses XMLTOCURSOR() to convert the results into a cursor, and if we have any results, uses a SQL SELECT command to create a cursor of the desired structure (TITLE for the title of the document, URL for the URL of the document, and CONTENTS for the contents of the document) and CURSORTOXML() to convert it back to an XML string. It then returns either the updated XML or an empty string if the search failed.

The WSSearch Class

The WSSearch class, defined in WSSEARCH.PRG, is based on the Session base class. It has three custom properties: cErrorMessage, which contains an error message if something went wrong (such as we couldn't initialize the MSSoap object), cProcessingMessage, which contains a string indicating the progress of the search, and aResults, an array property that contains the results of the search. The Init method opens the WSSEARCH table, the Destroy method closes it, and the Error method simply sets cErrorMessage to MESSAGE() so we can capture the error message.

The Search method does all the work. It expects to be passed three parameters: the search string and the beginning and ending dates for the search. It places the results of each Web Service search in a row in aResults and returns the number of rows in the array. The first column of aResults contains the Web Service name, the second contains .T. if the search succeeded, and the third contains the XML returned from the Web Service if it succeeded or the text of the error message if it failed (such as the Web Service not being available).

Here's the first part of this method's code. If any of the parameters are the wrong data type or empty, it sets cErrorMessage appropriately and returns –1.

```
function Search(tcSearch, tdBegin, tdEnd)
local lnResults, ;
  lcName, ;
  loWS, ;
  loSearch, ;
  llOK, ;
  lcMessage, ;
  lcXML
with This

* Ensure we have valid parameters.

  if vartype(tcSearch) <> 'C' or empty(tcSearch)
    .cErrorMessage = 'Invalid search string'
    return -1
```

```
  endif vartype(tcSearch) <> 'C' ...
  if vartype(tdBegin) <> 'D' or empty(tdBegin)
    .cErrorMessage = 'Invalid starting date'
    return -1
  endif vartype(tdBegin) <> 'D' ...
  if vartype(tdEnd) <> 'D' or empty(tdEnd)
    .cErrorMessage = 'Invalid starting date'
    return -1
  endif vartype(tdEnd) <> 'D' ...
```

Next, the code initializes a variable that contains the number of Web Services processed and starts a SCAN loop that processes active records in the WSSEARCH table. For each Web Service record, Search adds a new row to the aResults array and sets the cProcessingMessage property to indicate it's trying to access the Web Service.

```
  lnResults = 0
  scan for ACTIVE

* Add a new row to the array and set the processing
* message.

    .cErrorMessage = ''
    lcName         = alltrim(NAME)
    lcCode         = SEARCH
    lcUserName     = alltrim(USERNAME)
    lcPassword     = alltrim(PASSWORD)
    lnResults      = lnResults + 1
    dimension .aResults[lnResults, 3]
    .aResults[lnResults, 1] = lcName
    .cProcessingMessage     = 'Accessing ' + lcName
```

Next, the code instantiates a subclass of WSClient (we'll look at that subclass later) and calls the SetupClient method to create the proxy object for the Web Service. If it failed (likely because you have no Internet connection or the Web Service is currently unavailable), Search sets a flag and grabs the error message from the WSClient object.

```
    loWS         = createobject('WSClient3')
    loWS.cWSName = lcName
    loSearch     = loWS.SetupClient(alltrim(WSDL), '', ;
      '')
    lcMessage    = loWS.cErrorMessage
    llOK         = vartype(loSearch) = 'O' and ;
      empty(lcMessage)
```

If Search succeeded in create the Web Service proxy, it updates cProcessingMessage to indicate that it's searching the Web Service and calls the code contained in the SEARCH memo, passing it the proxy object, the search string, the starting and ending dates, and the user name and password. The return value from that code will either be XML containing the search results or empty if an error occurred or the search failed, so Search places the result or error message into the array and flags whether the search was successful or not.

```
    if llOK
      .cProcessingMessage = 'Searching ' + lcName
      lcXML = execscript(lcCode, loSearch, tcSearch, ;
        tdBegin, tdEnd, lcUserName, lcPassword)
      .aResults[lnResults, 2] = not empty(lcXML) and ;
        '<?xml' $ lcXML
      do case
        case not empty(lcXML)
          .aResults[lnResults, 3] = lcXML
        case not empty(.cErrorMessage)
          .aResults[lnResults, 3] = .cErrorMessage
        otherwise
          .aResults[lnResults, 3] = 'No results'
      endcase
```

If Search couldn't create the proxy object, an appropriate message is put into the third column of the current array row.

```
    else
      .aResults[lnResults, 3] = lcMessage
    endif llOK
```

Finally, when Search has processed all of the Web Services, it sets cProcessingMessage to "done", clears the error message (since any error messages will now be in aResults) and returns the number of rows in aResults.

```
  endscan for ACTIVE
  .cProcessingMessage = 'Done'
  .cErrorMessage      = ''
endwith
return lnResults
```

WSClient3

VFP comes with two classes that provide a wrapper for the Microsoft SOAP client object: WSClient and WSClient2, both located in _WebServices.vcx in the FFC subdirectory of the VFP home directory. WSClient is fine for interactive uses (from the Command window for example), but WSClient2 is a better choice for applications because it doesn't use MESSAGEBOX() to display warnings when there are problems.

I've discovered a couple of slight problems with WSClient2. First, if the proxy object can't be created, WSClient2 doesn't store the error message because the ErrorAlert method, called from the Error method, doesn't do anything (it has an IF .F. statement that short-circuits the behavior of the method). Second, SetupClient doesn't properly clean up after itself; it changes the current work area and doesn't restore it afterward. Finally, CheckWSDBF, which is actually defined in WSClient and is called from several methods, opens a table the class needs to record information in if it isn't already open, but doesn't select it if it is. The methods that call it expect that the desired table is selected, and don't work properly if not.

To work around these issues, I created a subclass of WSClient2 called WSClient3, also defined in WSSEARCH.PRG. The ErrorAlert method of WSClient3 stores the message passed to it by Error in a new cErrorMessage property so anything using this class can determine what went wrong. The SetupClient method saves the current work area, uses DODEFAULT() to do the usual behavior, and then restores the work area. Finally, CheckWSDBF selects the appropriate table if it's already open.

```
define class WSClient3 as WSClient2 of ;
  (home() + 'ffc\_webservices.vcx')
  cErrorMessage = ''

  function ErrorAlert(tcMessage)
    This.cErrorMessage = tcMessage
  endfunc

  function SetupClient(tcURI, tcService, tcPort)
    local lnSelect, ;
      luReturn
    lnSelect = select()
    luReturn = dodefault(tcURI, tcService, tcPort)
    select (lnSelect)
    return luReturn
  endfunc

  function CheckWSDBF
    local llReturn
    if not empty(This.cWSAlias) and used(This.cWSAlias)
      select (This.cWSAlias)
      llReturn = .T.
    else
      llReturn = dodefault()
    endif not empty(This.cWSAlias) ...
```

```
    return llReturn
  endfunc
enddefine
```

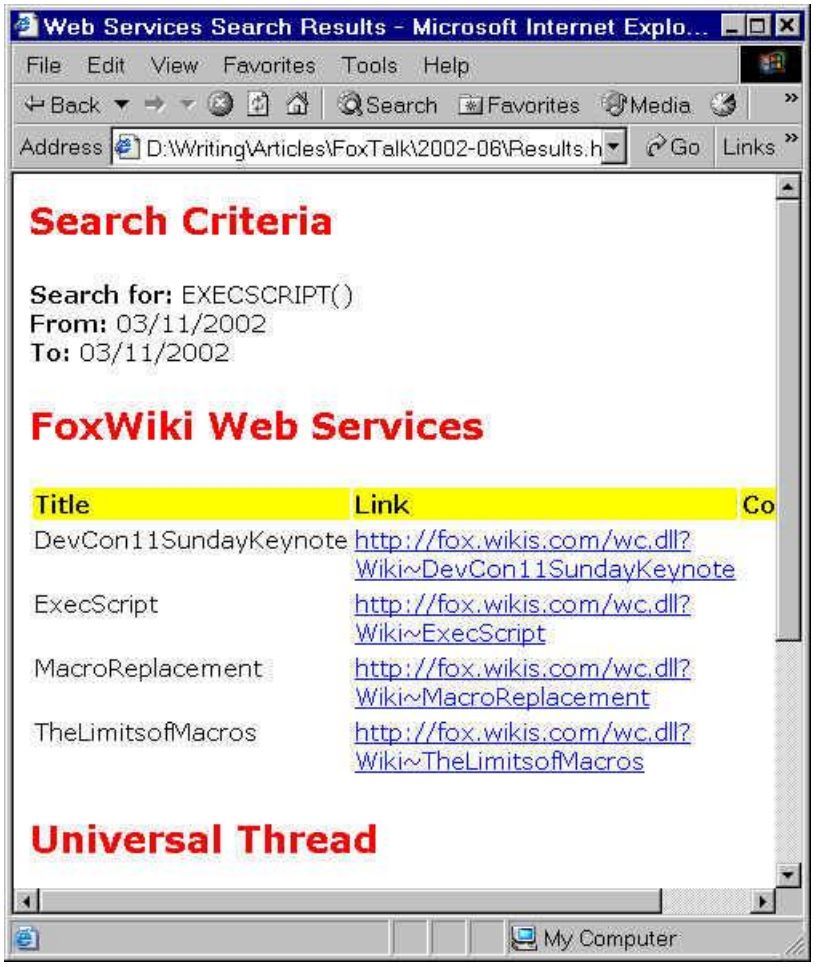
Check it Out

To make it easy to call WSSearch, I created a simple front-end form for it. WSSEARCH.SCX has a text box for the search string and DateTimePicker ActiveX controls for the starting and ending dates. When you click on the Search button, it uses WSSearch to perform the searching and then converts the results to an HTML document using the following code:

```
set textmerge on to Results.html noshow
\\<html>
\<head>
\<title>Web Services Search Results</title>
\</head>
\<body>
\<h1>Search Criteria</h1>
\<b>Search for:</b> <<lcSearchFor>><br>
\<b>From:</b> <<ldStart>><br>
\<b>To:</b> <<ldEnd>>
for lnI = 1 to lnResults
  \<h1><<alltrim(.oSearch.aResults[lnI, 1])>></h1>
  if .oSearch.aResults[lnI, 2]
    xmltocursor(.oSearch.aResults[lnI, 3], 'TEMP')
    \<table>
    \<tr><th>Title</th><th>Link</th>
    \\<th>Contents</th></tr>
    scan
      \<tr><td valign="top"><<alltrim(TITLE)>></td>
      \\<td valign="top"><a href="<<alltrim(URL)>>">
      \\<<alltrim(URL)>></a></td>
      lcContents = strtran(alltrim(CONTENTS), ccCRLF, ;
        '<br>')
      lcContents = strtran(lcContents, ccCR, '<br>')
      lcContents = strtran(lcContents, ccLF, '<br>')
      \\<td valign="top"><<lcContents>></td></tr>
    endscan
    use
    \</table>
  else
    \<p><<.oSearch.aResults[lnI, 3]>>
  endif .oSearch.aResults[lnI, 2]
next lnI
\</body>
\</html>
\
set textmerge to
```

The resulting document has live links to each of the search result topics; Figure 1 shows the results of a search for “EXECSCRIPT()”.

Figure 1. Search results.



Of course, before you can actually use this, you'll need to fill in your Universal Thread user name and password and your Google user ID in the appropriate fields in WSSEARCH.DBF, and put a copy of GoogleSearch.WSDL in the same directory as the other files for this month's article.

This form (and the class it uses) makes it easy to look for solutions to problems you're sure other VFP developers have solved or to search for information on new technologies (like how to use Web Services) without having to manually go to each search engine's Web site. I hope you find this as useful as I have!

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and Stonefield Query, co-author of "What's New in Visual FoxPro 7.0" and "The Hacker's Guide to Visual FoxPro 7.0", both from Hentzenwerke Publishing, and author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's Pros Talk Visual FoxPro series. He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals", both from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Certified Professional (MCP). Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com