etrieving and Updating Data With CursorAdapters

Doug Hennig

The new CursorAdapter base class added in VFP 8 provides an easy-to-use, consistent interface to data. This month's article shows how to use CursorAdapter to access local data and remote data via ODBC, ADO, and XML, discussing the particular requirements and "gotchas" for each.

As I stated in last month's article, "Adapt Your Cursors to CursorAdapters", one of the most important and exciting new features in VFP 8 is the new CursorAdapter base class. Last month, we looked at the properties, events, and methods of CursorAdapter and discussed the advantages it has over using remote views, SQL passthrough (SPT), ADO, and XML.

Before you start using CursorAdapter, you need to be aware of the specific requirements of the class when you use it to access native data or remote data via ODBC, ADO, and XML. This month's article gives the details for each data source type.

Using Native Data

Even though it's clear that CursorAdapter was intended to standardize and simplify the access to non-VFP data, you can use it as a substitute for Cursor by setting the DataSourceType property to "Native". Why would you do this? Mostly as a look toward the future when your application might be upsized; by simply changing the DataSourceType to one of the other choices (and likely changing a few other properties such as setting connection information), you can easily switch to another database engine such as SQL Server.

When DataSourceType is set to "Native", VFP ignores the DataSource property. SelectCmd must be a SQL SELECT statement, not a USE command or expression, so that means you're always working with the equivalent of a local view rather than the table directly. You're responsible for ensuring that VFP can find any tables referenced in the SELECT statement, so if the tables aren't in the current directory, you either need to set a path or open the database the tables belong to. As usual, if you want the cursor to be updateable, be sure to set the update properties (KeyFieldList, Tables, UpdatableFieldList, and UpdateNameList).

The following example (NativeExample.prg, included in this month's Subscriber Downloads) creates an updateable cursor from the Customer table in the TestData VFP sample database:

```
local loCursor as CursorAdapter, ;
  laErrors[1]
open database (_samples + 'data\testdata')
loCursor = createobject('CursorAdapter')
with loCursor
  .Alias              = 'customercursor'
  .DataSourceType     = 'Native'
  .SelectCmd          = ;
    "select CUST_ID, COMPANY, CONTACT from CUSTOMER " + ;
    "where COUNTRY = 'Brazil'"
  .KeyFieldList       = 'CUST_ID'
  .Tables             = 'CUSTOMER'
  .UpdatableFieldList = 'CUST_ID, COMPANY, CONTACT'
  .UpdateNameList     = 'CUST_ID CUSTOMER.CUST_ID, ' + ;
    'COMPANY CUSTOMER.COMPANY, CONTACT CUSTOMER.CONTACT'
  if .CursorFill()
    browse
    tableupdate(1)
  else
    aerror(laErrors)
    messagebox(laErrors[2])
  endif .CursorFill()
endwith
close databases all
```

Using ODBC

ODBC is actually the most straightforward of the four settings of DataSourceType. You set DataSource to an open ODBC connection handle, set the usual properties, and call CursorFill to retrieve the data. If you fill in KeyFieldList, Tables, UpdatableFieldList, and UpdateNameList, VFP will automatically generate the appropriate UPDATE, INSERT, and DELETE statements to update the backend with any changes. If you want to use a stored procedure instead, set the *Cmd, *CmdDataSource, and *CmdDataSourceType properties appropriately (replace "*" with "Delete", "Insert", and "Update").

Here's an example, taken from ODBCExample.prg, that calls the CustOrderHist stored procedure in the Northwind database that comes with SQL Server to get total units sold by product for a specific customer:

```
local lcConnString, ;
  loCursor as CursorAdapter, ;
  laErrors[1]
lcConnString = 'driver=SQL Server;server=(local); ' + ;
  'database=Northwind;uid=sa;pwd=;trusted_connection=no'
* change password to appropriate value for your database
loCursor = createobject('CursorAdapter')
with loCursor
  .Alias          = 'CustomerHistory'
  .DataSourceType = 'ODBC'
  .DataSource     = sqlstringconnect(lcConnString)
  .SelectCmd      = "exec CustOrderHist 'ALFKI'"
  if .CursorFill()
    browse
  else
    aerror(laErrors)
    messagebox(laErrors[2])
  endif .CursorFill()
endwith
```

Using ADO

Using ADO as the data access mechanism with CursorAdapter has a few more issues than using ODBC:

- DataSource must be set to an ADO RecordSet that has its ActiveConnection property set to an open ADO Connection object.
- If you want to use a parameterized query (which is likely the usual case rather than retrieving all records), you have to pass an ADO Command object that has its ActiveConnection property set to an open ADO Connection object as the fourth parameter to CursorFill. VFP will take care of filling the Parameters collection of the Command object (it parses SelectCmd to find the parameters) for you, but of course the variables containing the values of the parameters must be in scope.
- Using one CursorAdapter with ADO in a DataEnvironment is straightforward: you can set UseDEDataSource to .T. if you wish, then set the DataEnvironment's DataSource and DataSourceType properties as you would with the CursorAdapter. However, this doesn't work if there's more than one CursorAdapter in the DataEnvironment. The reason is that the ADO RecordSet referenced by DataEnvironment.DataSource can only contain a single CursorAdapter's data; when you call CursorFill for the second CursorAdapter, you get a "RecordSet is already open" error. So, if your DataEnvironment has more than one CursorAdapter, you must set UseDEDataSource to .F and manage the DataSource and DataSourceType properties of each CursorAdapter yourself (or perhaps use a DataEnvironment subclass that manages that for you).

The sample code below, taken from ADOExample.prg, shows how to retrieve data using a parameterized query with the help of an ADO Command object. This example also shows the use of the new structured error handling features in VFP 8; the call to the ADO Connection Open method is wrapped in a TRY ... CATCH ... ENDTRY statement to trap the COM error the method will throw if it fails.

```
local loConn as ADODB.Connection, ;
  loCommand as ADODB.Command, ;
  loException as Exception, ;
```

```
  loCursor as CursorAdapter, ;
  lcCountry, ;
  laErrors[1]
loConn = createobject('ADODB.Connection')
with loConn
  .ConnectionString = ;
    'provider=SQLOLEDB.1;data source=(local);' + ;
    'initial catalog=Northwind;uid=sa;pwd=;' + ;
    'trusted_connection=no'
* change password to appropriate value for your database
  try
    .Open()
  catch to loException
    messagebox(loException.Message)
    cancel
  endtry
endwith
loCommand = createobject('ADODB.Command')
loCursor  = createobject('CursorAdapter')
with loCursor
  .Alias          = 'Customers'
  .DataSourceType = 'ADO'
  .DataSource     = createobject('ADODB.RecordSet')
  .SelectCmd      = ;
    'select * from customers where country=?lcCountry'
  lcCountry       = 'Brazil'
  .DataSource.ActiveConnection = loConn
  loCommand.ActiveConnection   = loConn
  if .CursorFill(.F., .F., 0, loCommand)
    browse
  else
    aerror(laErrors)
    messagebox(laErrors[2])
  endif .CursorFill(.F., .F., 0, loCommand)
endwith
```

Using XML

Using XML with CursorAdapters requires some additional things. Here are the issues:

- The DataSource property is ignored.
- The CursorSchema property must be filled in, even if you pass .F. as the first parameter to the CursorFill method, or you'll get an error.
- The SelectCmd property must be set to an expression, such as a user-defined function (UDF) or object method name, that returns the XML for the cursor.
- Changes made to the cursor are converted to a DiffGram, which is XML that contains before and after values for changed fields and records, and placed in the DiffGram property when the update is required.
- In order to write changes back to the data source, UpdateCmdDataSourceType must be set to "XML" and UpdateCmd must be set to an expression (again, likely a UDF or object method) that handles the update. You'll probably want to pass "This.DiffGram" to the UDF so it can send the changes to the data source.

The XML source for the cursor could come from a variety of places. For example, you could call a UDF that converts a VFP cursor into XML using CURSORTOXML() and returns the results:

```
use CUSTOMERS
cursortoxml('customers', 'lcXML', 1, 8, 0, '1')
return lcXML
```

The UDF could call a Web Service that returns a result set as XML. Here's an example that IntelliSense generated for me from a Web Service I created and registered on my own system (the details aren't important; it just shows an example of a Web Service).

```
loWS = newobject('Wsclient', ;
  home() + 'ffc\_webservices.vcx')
loWS.cWSName = 'dataserver web service'
loWS = loWS.SetupClient('http://localhost/' + ;
  'SQDataServer/dataserver.WSDL', 'dataserver', ;
  'dataserverSoapPort')
lcXML = loWS.GetCustomers()
return lcXML
```

It could use SQLXML 3.0 to execute a SQL Server 2000 query stored in a template file on a Web Server (for more information on SQLXML, go to http://msdn.microsoft.com and search for SQLXML). The following code uses an MSXML2.XMLHTTP object to get all records from the Northwind Customers table via HTTP; this will be explained in more detail later.

```
local loXML as MSXML2.XMLHTTP
loXML = createobject('MSXML2.XMLHTTP')
loXML.open('POST', 'http://localhost/northwind/ ' + ;
  'template/getallcustomers.xml, .F.)
loXML.setRequestHeader('Content-type', 'text/xml')
loXML.send()
return loXML.responseText
```

Handling updates is more complicated. The data source must either be capable of accepting and consuming a DiffGram (as is the case with SQL Server 2000) or you'll have to figure out the changes yourself and issue a series of SQL statements (UPDATE, INSERT, and DELETE) to perform the updates.

Here's an example (XMLExample.prg) that uses a CursorAdapter with an XML data source. Notice that both SelectCmd and UpdateCmd call UDFs. In the case of SelectCmd, the customer ID to retrieve is passed to a UDF called GetNWCustomers, which we'll look at in a moment. For UpdateCmd, VFP passed the DiffGram property to SendNWXML, which we'll also look at later.

```
local loCustomers as CursorAdapter, ;
  laErrors[1]
loCustomers = createobject('CursorAdapter')
with loCustomers
  .Alias                  = 'Customers'
  .CursorSchema           = ;
    'CUSTOMERID C(5), COMPANYNAME C(40), ' + ;
    'CONTACTNAME C(30), CONTACTTITLE C(30), ' + ;
    'ADDRESS C(60), CITY C(15), REGION C(15), ' + ;
    'POSTALCODE C(10), COUNTRY C(15), ' + ;
    'PHONE C(24), FAX C(24)'
  .DataSourceType         = 'XML'
  .KeyFieldList           = 'CUSTOMERID'
  .SelectCmd              = 'GetNWCustomers([ALFKI])'
  .Tables                 = 'CUSTOMERS'
  .UpdatableFieldList     = ;
    'CUSTOMERID, COMPANYNAME, CONTACTNAME, ' + ;
    'CONTACTTITLE, ADDRESS, CITY, REGION, ' + ;
    'POSTALCODE, COUNTRY, PHONE, FAX'
  .UpdateCmdDataSourceType = 'XML'
  .UpdateCmd              = 'SendNWXML(This.DiffGram)'
  .UpdateNameList         = ;
    'CUSTOMERID CUSTOMERS.CUSTOMERID, ' + ;
    'COMPANYNAME CUSTOMERS.COMPANYNAME, ' + ;
    'CONTACTNAME CUSTOMERS.CONTACTNAME, ' + ;
    'CONTACTTITLE CUSTOMERS.CONTACTTITLE, ' + ;
    'ADDRESS CUSTOMERS.ADDRESS, ' + ;
    'CITY CUSTOMERS.CITY, ' + ;
    'REGION CUSTOMERS.REGION, ' + ;
    'POSTALCODE CUSTOMERS.POSTALCODE, ' + ;
    'COUNTRY CUSTOMERS.COUNTRY, ' + ;
```

```
    'PHONE CUSTOMERS.PHONE, ' + ;
    'FAX CUSTOMERS.FAX'
  if .CursorFill(.T.)
    browse
  else
    aerror(laErrors)
    messagebox(laErrors[2])
  endif .CursorFill(.T.)
endwith
```

Here's the code for GetNWCustomers. It uses an MSXML2.XMLHTTP object to access a SQL Server 2000 XML template named CustomersByID.xml on a Web server and returns the results. The customer ID to retrieve is passed as a parameter to this code.

```
lparameters tcCustID
local loXML as MSXML2.XMLHTTP
loXML = createobject('MSXML2.XMLHTTP')
loXML.open('POST', 'http://localhost/northwind/' + ;
  'template/customersbyid.xml?customerid=' + tcCustID, ;
  .F.)
loXML.setRequestHeader('Content-type', 'text/xml')
loXML.send()
return loXML.responseText
```

The XML template this code references, CustomersByID.XML, looks like the following:

```
<root xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <sql:header>
    <sql:param name="customerid">
    </sql:param>
  </sql:header>
  <sql:query client-side-xml="0">
    SELECT *
    FROM   Customers
    WHERE CustomerID = @customerid
    FOR XML AUTO
  </sql:query>
</root>
```

Place this file in a virtual directory for the Northwind database (see the sidebar "Setting up SQL Server 2000 XML Access" for details on configuring IIS to work with SQL Server, and for the specific details needed for this article).

SendNWXML looks similar to GetNWCustomers, except it expects to be passed a DiffGram, loads the DiffGram into an MSXML2.DOMDocument object, and passes that object to the Web Server, which will in turn pass it via SQLXML to SQL Server 2000 for processing.

```
lparameters tcDiffGram
local loDOM as MSXML2.DOMDocument, ;
  loXML as MSXML2.XMLHTTP
loDOM = createobject('MSXML2.DOMDocument')
loDOM.async = .F.
loDOM.loadXML(tcDiffGram)
loXML = createobject('MSXML2.XMLHTTP')
loXML.open('POST', 'http://localhost/northwind/', .F.)
loXML.setRequestHeader('Content-type', 'text/xml')
loXML.send(loDOM)
```

To see how this works, run XMLExample.prg. You should see a single record (the ALFKI customer) in a browse window. Change the value in some field, then close the window and run the PRG again. You should see that your change was written to the backend.

Summary

Although the CursorAdapter base class provides a consistent interface to remote data regardless of whether you use ODBC, ADO, or XML, there are some differences in how you set up a CursorAdapter, depending on the data access mechanism chosen. These differences are due to the requirements of the data access mechanisms themselves.

Next month, we'll wind up our examination of CursorAdapter by creating some reusable data classes and discussing how to use CursorAdapters in reports.

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

Setting Up SQL Server 2000 XML Access

In order to access SQL Server 2000 using a URL in a browser or other HTTP client, you have to do a few things. First, you need to download and install SQLXML 3.0 from the MSDN Web site (http://msdn.microsoft.com; do a search for "SQLXML" and then choose the download link).

Next, you have to set up an IIS virtual directory. To do this, choose the Configure IIS Support shortcut in the SQLXML 3.0 folder under Start, Programs in your Windows Taskbar. Expand the node for your server, choose the Web site to work with, and then right-click and choose New, Virtual Directory. In the General tab of the dialog that appears, enter the name of the virtual directory and its physical path. For this article, use "Northwind" as the virtual directory name and "NorthwindTemplates" for the physical directory. Using Windows Explorer, create the physical directory somewhere on your system, and create a subdirectory of it called "Template" (we'll use that subdirectory in a moment). Copy the two templates files included with this month's Subscriber downloads, GetAllCustomers.xml and CustomersByID.xml, to the Template subdirectory.

In the Security tab, enter the appropriate information to access SQL Server, such as the user name and password or the specific authentication mechanism you want to use. In the Data Source tab, choose the SQL Server and, if desired, the database to use. For the example for this article, choose the Northwind database. In the Settings tab, choose the desired settings, but at a minimum, turn on Allow Template Queries and Allow POST.

In the Virtual Names tab, choose "template" from the Type combobox and enter a virtual name (for this article, use "template") and physical path (which should be a subdirectory of the virtual directory; for this article, use "Template") to use for templates. Click on OK.

To test that everything is set up correctly, we'll access SQL Server using the GetAllCustomers.xml template file you copied into in the Template subdirectory. It has the following content:

```
<root xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <sql:query client-side-xml="0">
     SELECT *
     FROM   Customers
     FOR XML AUTO
  </sql:query>
</root>
```

To test this, bring up your browser and type the following URL: http://localhost/northwind/template/getallcustomers.xml. You should see the contents of the Northwind Customers table as XML in your browser. Here's the first part of what you should see:

```
<root xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<Customers CustomerID="ALFKI" CompanyName="Alfreds
Futterkiste" ContactName="Maria Anders"
ContactTitle="Sales Representative"
Address="Obere Str. 57" City="Berlin" PostalCode="12209"
Country="Germany" Phone="030-0074321"
Fax="999-999-9999" />
```

Now you're ready to run the SQLXML examples in this article.