Data Strategies in VFP: Introduction

By Doug Hennig

Overview

There are a lot of ways you can access non-VFP data (such as SQL Server) in VFP applications: remote views, SQL passthrough, ADO, XML ... This document will examine the pros and cons of different mechanisms and discuss when it's appropriate to use a particular strategy. We'll also look at an exciting new technology in VFP called CursorAdapter that will make accessing remote data much easier than it is in earlier versions.

Introduction

More and more VFP developers are storing their data in something other than VFP tables, such as SQL Server or Oracle. There are a lot of reasons for this, including fragility of VFP tables (both perceived and actual), security, database size, and corporate standards. Microsoft has made access to non-VFP data easier with every release, and has even been encouraging it with the inclusion of MSDE (Microsoft Data Engine, a free, stripped-down version of SQL Server) on the VFP 7 CD.

However, accessing a backend database has never been quite as easy as using VFP tables. In addition, there are a variety of mechanisms you can use to do this:

- Remote views, which are based on ODBC connections.
- SQL passthrough (SPT) functions, such as SQLCONNECT(), SQLEXEC(), and SQLDISCONNECT(), which are also based on ODBC connections.
- ActiveX Data Objects, or ADO, which provide an object-oriented front end to OLE DB providers for database engines.
- XML, which is a lightweight, platform-independent, data transport mechanism.

Which mechanism should you choose? The answer (as it is for the majority of VFP questions <g>) is "it depends". It depends on a number of factors, including your development team's experience and expertise, your infrastructure, the needs of the application, anticipated future requirements, and so on.

Let's take a look at each of these mechanisms, including their advantages and disadvantages, some techniques to be aware of, and my opinion of where each fits in the overall scheme of things. We'll also take a look at what I think is one of the biggest new features in VFP 8, the CursorAdapter class, and how it makes using remote data access via ODBC, ADO, or XML much easier and consistent.

Remote Views

Like a local view, a remote view is simply a pre-defined SQL SELECT statement that's defined in a database container. The difference is that a remote view accesses data via ODBC (even if the data it's accessing is VFP) rather than natively.

You can create a remote view either programmatically using the CREATE SQL VIEW command or visually using the View Designer. In both cases, you need to specify an ODBC connection to use. The connection can either be an ODBC data source (DSN) set up on your system or a Connection object that's already been defined in the same database. Here's an example that creates a remote view to the Customers table of the sample Northwind database that comes with SQL Server (we'll use this database extensively in our examples). This was excerpted from code generated by GENDBC; there's actually a lot more code that sets the various properties of the connection, the view, and the fields.

```
CREATE CONNECTION NORTHWINDCONNECTION ;
   CONNSTRING "DSN=Northwind SQL; UID=sa; PWD=testdb; " + ;
   "DATABASE=Northwind; TRUSTED CONNECTION=No"
CREATE SQL VIEW "CUSTOMERSVIEW" ;
   REMOTE CONNECT "NorthwindConnection" ;
   AS SELECT * FROM dbo.Customers Customers
DBSetProp('CUSTOMERSVIEW', 'View', 'UpdateType', 1)
DBSetProp('CUSTOMERSVIEW', 'View', 'WhereType', 3)
DBSetProp('CUSTOMERSVIEW', 'View', 'FetchMemo', .T.)
DBSetProp('CUSTOMERSVIEW', 'View', 'SendUpdates', .T.)
DBSetProp('CUSTOMERSVIEW', 'View', 'Tables', 'dbo.Customers')
DBSetProp('CUSTOMERSVIEW.customerid', 'Field', 'KeyField', .T.)
DBSetProp('CUSTOMERSVIEW.customerid', 'Field', 'Updatable', .F.)
DBSetProp('CUSTOMERSVIEW.customerid', 'Field', 'UpdateName', ;
   'dbo.Customers.CustomerID')
DBSetProp('CUSTOMERSVIEW.companyname', 'Field', 'Updatable', .T.)
DBSetProp('CUSTOMERSVIEW.companyname', 'Field', 'UpdateName', ;
   'dbo.Customers.CompanyName')
```

One of the easiest ways you can upsize an existing application is using the Upsizing Wizard to create SQL Server versions of your VFP tables, then create a new database (for example, REMOTE.DBC) and create remote views in that database that have the same names as the tables they're based on. That way, the code to open a remote view will be exactly the same as that to open a local table except you'll open a different database first. For example:

```
if oApp.lUseLocalData
  open database Local
else
  open database Remote
endif
use CUSTOMERS
```

If you're using cursor objects in the DataEnvironment of forms and reports, you have a little bit of extra work to do because those objects have a reference to the specific database you had selected when you dropped the views into the DataEnvironment. To handle this, put code similar to the following into the BeforeOpenTables method of the DataEnvironment:

```
local loObject
for each loObject in This.Objects
  if upper(loObject.BaseClass) = 'CURSOR' and not empty(loObject.Database)
    loObject.Database = iif(oApp.lUseLocalData, 'local.dbc', 'remote.dbc')
  endif upper(loObject.BaseClass) = 'CURSOR' ..
next loObject
```

One thing to be aware of: When you open a view, VFP attempts to lock the view's records in the DBC, even if it's only briefly. This can cause contention in busy applications where several users might try to open a form at the same time. Although there are workarounds (copying the DBC to the local workstation and using that one or, in VFP 7 and later, using SET REPROCESS SYSTEM to increase the timeout for lock contention), it's something you'll need to plan for.

There's quite a controversy over whether remote views are a good thing or not. If you're interested in reading about the various sides of the arguments, check out these links:

http://fox.wikis.com/wc.dll?Wiki~RemoteViews~VFP

http://fox.wikis.com/wc.dll?Wiki~MoreOnRemoteViews~VFP

http://fox.wikis.com/wc.dll?Wiki~Client/ServerDataAccessTechniques~VFP

http://fox.wikis.com/wc.dll?Wiki~Client/ServerTechniquesPerformance~VFP

Advantages

The advantages of remote views are:

- You can use the View Designer to create a remote view visually. Okay, until VFP 8, which fixes many of the known problems with the View Designer (especially with complex views involving more than two tables), this wasn't necessarily an advantage <g>, but even in earlier versions, simple views can be created very quickly and easily. It's great to visually see all the fields in the underlying tables, easily set up the various parts of the SQL SELECT statement using a friendly interface, and quickly set properties of the view using checkboxes or other UI elements.

- From a language point-of-view, remote views act just like tables. As a result, they can be used anywhere: you can USE them, add them to the DataEnvironment of a form or report, bind them to a grid, process them in a SCAN loop, and so forth. With some of the other technologies, specifically ADO and XML, you have to convert the result set to a VFP cursor before you can use it in many places in VFP.

- It's easier to convert an existing application to use remote views, especially if it already uses local views, than using any of the other techniques.

- Because you can add a remote view to the DataEnvironment of a form or report, you can take advantage of the visual support the DE provides: dragging and dropping fields or the entire cursor to automatically create controls, easily binding a control to a field by selecting it from a combobox in the Properties Window, and so on. Also, depending on the settings of the

AutoOpenTables and OpenViews properties, VFP will automatically open the remote views for you.

- It's easy to update the backend with changes: assuming the properties of the view have been set up properly, you simply call TABLEUPDATE(). Transaction processing and update conflict detection are built-in.

- Remote views are easy to use in a development environment: just USE and then BROWSE.

Disadvantages

The disadvantages of remote views are:

- Remote views live in a DBC, so that one more set of files you have to maintain and install on the client's system.

- Since a remote view's SQL SELECT statement is pre-defined, you can't change it on the fly. Although this is fine for a typical data entry form, it can be an issue for queries and reports. You may have to create several views from the same set of data, each varying in the fields selected, structure of the WHERE clause, and so forth.

- You can't call a stored procedure from a remote view, so a remote view needs direct access to the underlying tables. This may be an issue with your application's database administrators; some DBAs believe that data access should only be via stored procedures for security and other reasons. Also, because they are precompiled on the backend, stored procedures often perform significantly faster than SQL SELECT statements.

- When you use TABLEUPDATE() to write changes in the view to the backend database, you have little ability (other than by setting a few properties) to control how VFP does the update.

- As is the case with local views, if you use a SELECT * view to retrieve all the fields from a specific table and the structure of that table on the backend changes, the view is invalid and must be recreated.

- They work with ODBC only, so they're stuck in the "client-server" model of direct data connections. They can't take advantage of the benefits of ADO or XML.

- As I mentioned earlier, DBC lock contention is an issue you need to handle.

- Like other types of VFP cursors, you can't pass the result set outside the current data session, let alone to another application or tier. This limitation alone pretty much makes them useless in an n-tier application.

- Until VFP 8, which allows you to specify the connection handle to use when you open a remote view with the USE statement, you had little ability to manage the connections used by your application.

- The connection information used for a remote view is hard-coded in plain text in the DBC (notice the user name and password in the code shown earlier). That means that a hacker can easily discover the keys to your backend kingdom (such as the user name and password) using nothing more complicated than Notepad to open the DBC. This isn't much of an issue starting in VFP 7 because it allows you to specify a connection string when you open a remote view with the USE command, meaning that you can dynamically assemble the server, user name, and password, likely from encrypted information, just before opening the view.

Basically, it comes down to a control issue: remote views make it easy to work with backend data, but at the expense of limiting the control you have over them.

When to Use

Remote views are really only suited to client-server, 2-tier applications where you have a direct connection to the data. I believe that in the long run, most applications will be more flexible, robust, and have a longer shelf life if they're developed using an n-tier design, so remote views are best suited to those cases where you're upsizing an existing application and don't want to redesign/redevelop it or when your development team doesn't have much experience with other technologies.

SQL Passthrough

VFP provides a number of functions, sometimes referred to as SQL passthrough (or SPT) functions, which allow you to access a backend database. SQLCONNECT() and SQLSTRINGCONNECT() make a connection to the backend database engine; the difference between these two functions is that SQLCONNECT() requires an existing ODBC Data Source (DSN) that's been defined on the user's system, while SQLSTRINGCONNECT() simply passes the necessary information directly to ODBC, known as a DSNless connection. As you may guess, SQLDISCONNECT() disconnects from the backend. SQLEXEC() sends a command, such as a SQL SELECT statement, to the database engine, typically (but not necessarily, depending on the command) putting the returned results into a VFP cursor.

Here's an example that opens a connection to the Northwind database (this assumes there's a DSN called "Northwind" that defines how to connect to this database), retrieves a single customer record and displays it in a BROWSE window, and then closes the connection.

```
lnHandle = sqlconnect('Northwind')
if lnHandle > 0
  sqlexec(lnHandle, "select * from customers where customerid = 'ALFKI'")
  browse
  sqldisconnect(lnHandle)
else
  aerror(laErrors)
  messagebox('Could not connect: ' + laErrors[2])
endif lnHandle > 0
```

To use a DSNless connection instead, replace the SQLCONNECT() statement with the following (replace the name of the server, user ID, and password with the appropriate values):

```
lnHandle = sqlstringconnect('Driver=SQL Server;Server=(local);' + ;
  Database=Northwind;uid=sa;pwd=whatever')
```

The DispLogin SQL setting controls whether or not ODBC displays a login dialog. Although you might think it's handy to let ODBC worry about user names and passwords, you don't have any control over the appearance of the dialog or what happens if the user enters improper values. Instead, you're better off asking the user for the appropriate information in your own VFP dialog and then passing the information to ODBC. As described in the VFP help for the SQLCONNECT() function, you should use SQLSETPROP() to set DispLogin to 3.

Rather than having each component that requires access to remote data manage their own ODBC connections, use an object whose sole responsibility is managing the connection and accessing the data. SFConnectionMgr, which is based on Custom, is an example. It has several custom properties, some of which you must set before you can use it to connect to a remote data source.

Property	**Description**
cDatabase	The database to connect to; can leave blank if cDSN filled in.
cDriver	The ODBC driver or OLE DB provider to use; can leave blank if cDSN filled in.
cDSN	The ODBC DSN to connect to; leave blank to use a DSNless connection.
cErrorMessage	The message of any error that occurs.
cPassword	The password for the data source; can leave blank if a trusted connection is used.
cServer	The server the database is on; can leave blank if cDSN filled in.
cUserName	The user name for the data source; can leave blank if a trusted connection is used.
lConnected	.T. if we are connected to the data source.

It has several custom methods:

Method	**Description**
Connect	Connects to the data source.

Disconnect	Disconnects from the data source (called from Destroy but can also call manually).
Execute	Executes a statement against the data source.
GetConnection	Returns the connection handle or ADO Connection object.
GetConnectionString	Returns the connection string from the various connection properties (protected).
HandleError	Sets the cErrorMessage property when an error occurs (protected).

SFConnectionMgr isn't intended to be used directly, but is subclassed into SFConnectionMgrODBC and SFConnectionMgrADO, which are specific for ODBC and ADO, respectively. Those subclasses override most of the methods to provide the specific behavior needed. Let's look at SFConnectionMgrODBC.

The Init method saves the current DispLogin setting to a custom nDispLogin property and sets it to 3 so the login dialog is never displayed.

```
This.nDispLogin = sqlgetprop(0, 'DispLogin')
sqlsetprop(0, 'DispLogin', 3)
dodefault()
```

The Connect method checks to see if we're already connected, then either uses SQLCONNECT() if a DSN was specified in the cDSN property or builds a connection string and uses the SQLSTRINGCONNECT() function. Either way, if the connection succeeded, the nHandle property contains the connection handle and lConnected is .T. If it failed, nHandle is 0, lConnected is .F, and cErrorMessage contains information about what went wrong (set in the HandleError method, which we won't look at).

```
* If we're not already connected, connect to either the specified DSN or to
* the data source using a DSNless connection.

local lcConnString
with This
  if not .lConnected
    if empty(.cDSN)
      lcConnString = .GetConnectionString()
      .nHandle = sqlstringconnect(lcConnString)
    else
      .nHandle = sqlconnect(.cDSN, .cUserName, .cPassword)
    endif empty(.cDSN)

* Set the lConnected flag if we succeeded in connecting, and if not, get the
* error information.

    .lConnected = .nHandle > 0
```

```
    if not .lConnected
      .nHandle = 0
      .HandleError()
    endif not .lConnected
  endif not .lConnected
endwith
return This.lConnected
```

GetConnectionString returns a connection string from the values of the connection properties.

```
local lcSpecifier, ;
  lcConnString
with This
  lcSpecifier  = iif(upper(.cDriver) = 'SQL SERVER', 'database=', ;
    'dbq=')
  lcConnString = 'driver=' + .cDriver + ';' + ;
    iif(empty(.cServer),   '', 'server='   + .cServer   + ';') + ;
    iif(empty(.cUserName), '', 'uid='      + .cUserName + ';') + ;
    iif(empty(.cPassword), '', 'pwd='      + .cPassword + ';') + ;
    iif(empty(.cDatabase), '', lcSpecifier + .cDatabase + ';') + ;
    'trusted connection=' + iif(empty(.cUserName), 'yes', 'no')
endwith
return lcConnString
```

The Execute method executes a statement (such as a SQL SELECT command) against the data source. It first calls Connect to ensure we have a connection, then uses the SQLEXEC() function to pass the statement to the data source.

```
lparameters tcStatement, ;
  tcCursor
local lcCursor, ;
  llReturn
with This
  .Connect()
  if .lConnected
    lcCursor = iif(vartype(tcCursor) = 'C' and not empty(tcCursor), ;
      tcCursor, sys(2015))
    llReturn = sqlexec(.nHandle, tcStatement, lcCursor) >= 0
    if not llReturn
      .HandleError()
    endif not llReturn
  endif .lConnected
endwith
return llReturn
```

The Disconnect method disconnects from the data source and sets nHandle to 0 and lConnected to .F.

```
with This
  if .lConnected
    sqldisconnect(.nHandle)
    .nHandle    = 0
    .lConnected = .F.
  endif .lConnected
endwith
```

Destroy simply disconnects and restores the saved DispLogin setting.

```
This.Disconnect()
sqlsetprop(0, 'DispLogin', This.nDispLogin)
dodefault()
```

Here's an example (taken from TestConnMgr.prg) that shows how SFConnectionMgrODBC can be used. It first connects to the SQL Server Northwind database and grabs all the customer records, and then it connects to the Access Northwind database and again retrieves all the customer records. Of course, this example uses hard-coded connection information; a real application would likely store this information in a local table, an INI file, or the Windows Registry to make it more flexible.

```
loConnMgr = newobject('SFConnectionMgrODBC', 'SFRemote')
with loConnMgr

* Connect to the SQL Server Northwind database and get customer records.

  .cDriver   = 'SQL Server'
  .cServer   = '(local)'
  .cDatabase = 'Northwind'
  .cUserName = 'sa'
  .cPassword = ''
  do case
    case not .Connect()
      messagebox(.cErrorMessage)
    case .Execute('select * from customers')
      browse
      use
    otherwise
      messagebox(.cErrorMessage)
  endcase

* Now connect to the Access Northwind database and get customer records.

  .Disconnect()
  .cDriver   = 'Microsoft Access Driver (*.mdb)'
  .cServer   = ''
  .cDatabase = 'd:\Program Files\Microsoft Visual Studio\VB98\Nwind.mdb'
  .cUserName = ''
  .cPassword = ''
  do case
    case not .Connect()
      messagebox(.cErrorMessage)
    case .Execute('select * from customers')
      browse
      use
    otherwise
      messagebox(.cErrorMessage)
  endcase
endwith
```

Advantages

The advantages of using SPT are:

- You have a lot more flexibility in data access than with remote views, such as calling stored procedures using the SQLEXEC() function.

- You can change the connection information on the fly as needed. For example, you can store the user name and password as encrypted values and only decrypt them just before using them in the SQLCONNECT() or SQLSTRINGCONNECT() functions. You can also change servers or even backend database engines (for example, you could switch between SQL Server and Access database by simply changing the ODBC driver specified in the SQLSTRINGCONNECT() call). As I mentioned earlier, this isn't nearly the advantage over remote views that it used to be, now that VFP 7 allows you to specify the connection string on the USE command.

- You can change the SQL SELECT statement as needed. For example, you can easily vary the list of the fields, the WHERE clause (such as changing which fields are involved or eliminating it altogether), the tables, and so on.

- You don't need a DBC to use SPT, so there's nothing to maintain or install, lock contention isn't an issue, and you don't have to worry about a SELECT * statement being made invalid when the structure of the backend tables change.

- As with remote views, the result set of a SPT call is a VFP cursor, which can be used anywhere in VFP without the conversion issues that ADO and XML have.

- Although you have to code for it yourself (this is discussed in more detail under Disadvantages), you have greater control over how updates are done. For example, you might use a SQL SELECT statement to create the cursor but call a stored procedure to update the backend tables.

- You can manage your own connections. For example, you might want to use a connection manager similar to the one discussed earlier to manage all the connections used by your application in one place.

Disadvantages

The disadvantages of using SPT are:

- It's more work, since you have to code everything: creating and closing the connection, the SQL SELECT statements to execute, and so on. You don't have a nice visual tool like the View Designer to show you which fields exist in which tables on the backend.

- You can't visually add a cursor created by SPT to the DataEnvironment of a form or report. Instead, you have to code the opening of the cursors (for example, in the BeforeOpenTables method), you have to manually create the controls, and you have to fill in the binding

properties (such as ControlSource) by typing them yourself (don't make a typo when you enter the alias and field names or the form won't work).

- They're harder to use than remote views in a development environment: instead of just issuing a USE command, you have to create a connection, then use a SQLEXEC() call to get the data you want to look at. You can make things easier on yourself if you create a set of PRGs to do the work for you (such as UseCustomers.prg, which opens and displays the contents of the Customers table). You can also use the SQL Server Enterprise Manager (or similar tool) to examine the structures and contents of the tables. You could even create a DBC and set of remote views used only in the development environment as a quick way to look at the data.

- As with remote views and other types of VFP cursors, you can't pass the result set outside the current data session. Instead, you may have to pass the information used to create the cursor (the SQL SELECT statement or stored procedure call, and even possibly the connection information) and let the other object do the work itself.

- Cursors created with SPT can be updatable, but you have to make them so yourself using a series of CURSORSETPROP() calls to the SendUpdates, Tables, KeyFieldList, UpdatableFieldList, and UpdateNameList properties. Also, you have to manage transaction processing and update conflict detection yourself.

- Since SPT cursors aren't defined like remote views, you can't easily switch between local and remote data using SPT as you can with remote views (by simply changing which view you open in a form or report).

- Like remote views, SPT is based on ODBC only, so you can't take advantage of the benefits of ADO or XML.

When to Use

As with remote views, SPT is best suited to client-server, 2-tier applications where you have a direct connection to the data. Since it's more work to convert an existing application to SPT than remote view or CursorAdapters (which we'll see later), SPT is best suited to utilities, simple applications, or narrow-focus cases.

ADO

OLE DB and ADO are part of Microsoft's Universal Data Access strategy, in which data of any type stored anywhere in any format, not just in relational databases on a local server, can be made available to any application requiring it. OLE DB providers are similar to ODBC drivers: they provide a standard, consistent way to access data sources. A variety of OLE DB providers are available for specific DBMS (SQL Server, Oracle, Access/Jet, etc.), and Microsoft provides an OLE DB provider for ODBC data sources.

Because OLE DB is a set of low-level COM interfaces, it's not easy to work with in languages like VFP. To overcome this, Microsoft created ActiveX Data Objects, or ADO, a set of COM objects that provide an object-oriented front-end to OLE DB.

ADO consists of several objects, including:

- Connection: This is the object responsible for communicating with the data source.
- RecordSet: This is the equivalent of a VFP cursor: it has a defined structure, contains the data in the data set, and provides properties and methods to add, remove, or update records, move from one to another, filter or sort the data, and update the data source.
- Command: This object provides the means of doing more advanced queries than a simple SELECT statement, such as parameterized queries and calling stored procedures.

Here's an example (ADOExample.prg) that gets all Brazilian customers from the Northwind database and displays the customer ID and company name. Notice that the Connection object handles the connection (the RecordSet's ActiveConnection property is set to the Connection object), while the RecordSet handles the data.

```
local loConn as ADODB.Connection, ;
  loRS as ADODB.Recordset
loConn = createobject('ADODB.Connection')
loConn.ConnectionString = 'provider=SQLOLEDB.1;data source=(local);' + ;
  'initial catalog=Northwind;uid=sa;pwd='
loConn.Open()
loRS = createobject('ADODB.Recordset')
loRS.ActiveConnection = loConn
loRS.LockType         = 3  && adLockOptimistic
loRS.CursorLocation   = 3  && adUseClient
loRS.CursorType       = 3  && adOpenStatic
loRS.Open("select * from customers where country='Brazil'")
lcCustomers = ''
do while not loRS.EOF
  lcCustomers = lcCustomers + loRS.Fields('customerid').Value + chr(9) + ;
    loRS.Fields('companyname').Value + chr(13)
  loRS.MoveNext()
enddo while not loRS.EOF
messagebox(lcCustomers)
loRS.Close()
loConn.Close()
```

Because of its object-oriented nature and its capabilities, ADO has been the data access mechanism of choice for n-tier development (although this is quickly changing as XML becomes more popular). Unlike ODBC, you don't have to have a direct connection to the data source.

For details on ADO and using it in VFP, see John Petersen's "ADO Jumpstart for Visual FoxPro Developers" white paper, available from the VFP home page (http://msdn.microsoft.com/vfoxpro; follow the "Technical Resources", "Technical Articles", and "COM and ActiveX Development" links to get to the document).

Advantages

The advantages of using ADO are:

- As with SPT, you have more flexibility in data access than with remote views, such as calling stored procedures.
- You can change the connection information on the fly as needed, such as encrypting the user name and password, changing servers, or even backend database engines.
- You can change the SQL SELECT statement as needed.
- There's no DBC involved.
- Although performance differences aren't significant in simple scenarios (in fact, in my testing, ODBC is faster than ADO), ADO is more scalable in heavily-used applications such as Web servers.
- Unlike VFP cursors, ADO objects can be passed outside the current data session, whether to another component in the application, to another application or COM object (such as Excel or a middle-tier component), or to another computer altogether. You can even send them over HTTP using Remote Data Services (RDS; see John Petersen's white paper for more information), although this is a problem when firewalls are involved.
- ADO is object-oriented, so you can deal with the data like objects.
- Depending on how it's set up, ADO RecordSets are automatically updateable without any additional work (other than calling the Update or UpdateBatch methods). Transaction processing and update conflict detection are built-in.
- You can manage your own connections.
- You can easily persist a RecordSet to a local file, then later reload it and carry on working, and finally update the backend data source. This makes it a much better choice for "road warrior" applications than remote views or SPT.

Disadvantages

The disadvantages of ADO are:

- It's more work, since you have to code everything: creating and closing the connection, the SQL SELECT statements to execute, and so on. You don't have a nice visual tool like the View Designer to show you which fields exist in which tables on the backend.
- An ADO RecordSet is not a VFP cursor, so you can't use it in places that require a cursor, such as grids and reports. There are functions in the VFPCOM utility (available for download from the VFP home page, http://msdn.microsoft.com/vfoxpro) that can convert a RecordSet

to a cursor and vice versa, but using them can impact performance, especially with large data sets, and they have known issues with certain data types.

- There's no visual support for ADO RecordSets, so have to code their creation and opening, you have to manually create the controls, and you have to fill in the binding properties (such as ControlSource) by typing them yourself. This is even more work than for SPT, because the syntax isn't just CURSOR.FIELD—it's RecordSet.Fields('FieldName').Value.

- They're the hardest of the technologies to use in a development environment, since you have to code everything: making a connection, retrieving the data, moving back and forth between records. You can't even BROWSE to see visually what the result set looks like (unless you use VFPCOM or another means to convert the RecordSet to a cursor).

- There's a bigger learning curve involved with ADO than using the cursors created by ODBC.

- You'll likely have to ensure the client has the latest version of Microsoft Data Access Components (MDAC) installed to ensure their version of OLEDB and ADO match what your application requires.

- ADO is a Windows-only technology.

When to Use

ADO is easily used when passing data back and forth between other components. For example, a method of a VFP COM object can easily return an ADO RecordSet to Excel VBA code, which can then process and display the results.

If you're designing an application using an n-tier architecture, ADO may be a good choice if you're already familiar with it or already have infrastructure in place for it. However, XML is fast becoming the mechanism of choice for n-tier applications, so I expect ADO to be used less and less in this area.

XML

XML (Extendible Markup Language) isn't really a data access mechanism; it's really a transport technology. The data is packaged up as text with a structured format and then shipped off somewhere. However, because it's simply text, XML has a lot of advantages over other technologies (as we'll discuss later).

A few years ago, Microsoft "discovered" XML, and has been implementing it pretty much everywhere since then. The data access technology built into the .NET framework, ADO.NET, has XML as its basis (in fact, one simplistic view is that ADO.NET is really just a set of wrapper classes that expose XML data via an OOP interface). Web Services, based on SOAP (Simple Object Access Protocol), use XML as the basis for communications and data transfer. XML is even quickly becoming the data transport mechanism of choice for n-tier applications, which for a long time favored ADO in that role.

VFP 7 added several functions that work with XML: XMLTOCURSOR(), which converts XML to a cursor, CURSORTOXML(), which does the opposite, and XMLUPDATEGRAM(), which generates an updategram (XML in a specific format for indicating changes to data) from changes to a cursor. Here's some VFP code (taken from XMLExample1.prg) that shows how VFP can deal with XML:

```
* Get the information for the ALFKI customer and display the raw XML.

close databases all
lcXML = GetCustomerByID('ALFKI')
strtofile(lcXML, 'ALFKI.XML')
modify file ALFKI.XML
erase ALFKI.XML

* Put it into a cursor, browse and make changes, then view the updategram.

xmltocursor(lcXML, 'CUSTOMERS')
set multilocks on
cursorsetprop('Buffering', 5)
browse
lcUpdate = xmlupdategram()
strtofile(lcUpdate, 'UPDATE.XML')
modify file UPDATE.XML

* By setting the KeyFieldList property, the updategram will only contain key
* and changed fields.

cursorsetprop('KeyFieldList', 'cust id')
lcUpdate = xmlupdategram()
strtofile(lcUpdate, 'UPDATE.XML')
modify file UPDATE.XML
erase UPDATE.XML
close databases all

* This function returns the specified customer record as XML.

function GetCustomerByID(tcCustomerID)
local lcXML
open database (_samples + 'data\testdata')
select * from customer where cust id = tcCustomerID into cursor Temp
cursortoxml('Temp', 'lcXML', 1, 8, 0, '1')
use in Temp
use in Customer
return lcXML
```

Note that until version 8, while VFP could create an XML updategram, it didn't have a simple way to consume one (that is, to update VFP tables). Visual FoxPro MVP Alex Feldstein wrote a routine to do this (http://fox.wikis.com/wc.dll?Wiki~XMLUpdateGramParse), but in VFP 8, you can use an instance of the new XMLAdapter class to do this (we'll take a look at that class in the "Data Strategies in VFP: Advanced" document.

Here's an example (XMLExample2.prg) that uses SQLXML to get a customer's record from SQL Server via a Web Server (we'll discuss SQLXML in more detail in the Advanced document), then send changes back.

```
* Get the information for the ALFKI customer and display the raw XML.

close databases all
lcXML = GetNWCustomerByID('ALFKI')
strtofile(lcXML, 'ALFKI.XML')
modify file ALFKI.XML
erase ALFKI.XML

* Put it into a cursor, browse, and make changes.

xmltocursor(lcXML, 'CUSTOMERS')
cursorsetprop('KeyFieldList', 'customerid')
set multilocks on
cursorsetprop('Buffering', 5)
browse

* Get the updategram and save the changes to SQL Server.

lcUpdate = xmlupdategram()
SaveNWCustomers(lcUpdate)
use

* This function uses SQLXML to get the specified customer record as XML.

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

* This function uses SQLXML to get the specified customer record as XML.

function SaveNWCustomers(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)
```

Advantages

There are a lot of benefits to using XML:

- You have the same benefits over remote views, such as no DBC and an easily-changed SQL SELECT statement, as SPT and ADO have.

- Since XML isn't really a data access mechanism, you have the most flexibility in data access with XML. For example, you can use the VFP CURSORTOXML() function, Web Services,

middle tier components, ADO.NET DataSets, XMLHTTP, and many other mechanisms to get data from one place to another.

- Since it's just text, XML can be passed anywhere, even through firewalls (which is a problem for ADO).
- The XML DOM object provides an object-oriented interface to the XML data.
- XML is easily persisted anywhere: to a file, a memo field in a table, etc.
- XML is completely platform/operating system-independent.
- XML is human-readable, unlike the binary formats of other mechanisms.

Disadvantages

There are also some downsides to using XML:

- Belying its simple format, there's a bigger learning curve with XML than there is with the other mechanisms. The XML DOM object has its own object model, there's all kinds of new technology (and acronyms!) to learn like schemas, XSLT, XPath, XDR, and XQueries.
- The same set of data can be quite a bit larger in XML than in the other mechanisms because every element has beginning and ending tags. For example, the VFP CUSTOMER sample table goes from 26,257 bytes as a DBF file to 40,586 as XML.
- While CURSORTOXML() is fast, XMLTOCURSOR(), which uses the XML DOM object to do the work, can be quite slow, especially with large amounts of data.
- XML standards are still evolving.

When to Use

XML is great for lots of things, including storing configuration settings, passing small amounts of data between application components, storing structured data in memo fields, etc. However, XML is ideally suited to n-tier applications because it's easily transported (either between components or across a wire) and converted to and from data sets such as VFP cursors. Using XML updategrams (and the newer diffgrams), you can limit the amount of data being transported. If you're starting new n-tier projects, this is clearly the data access mechanism to use.

CursorAdapter

One of the things you've likely noted is that each of the mechanisms we've looked at is totally different from the others. That means you have a new learning curve with each one, and converting an existing application from one mechanism to another is a non-trivial task.

In my opinion, CursorAdapter is one of the biggest new features in VFP 8. The reasons I think they're so cool are:

- They make it easy to use ODBC, ADO, or XML, even if you're not very familiar with these technologies.
- They provide a consistent interface to remote data regardless of the mechanism you choose.
- They make it easy to switch from one mechanism to another.

Here's an example of the last point. Suppose you have an application that uses ODBC with CursorAdapters to access SQL Server data, and for some reason you want to change to use ADO instead. All you need to do is change the DataSourceType of the CursorAdapters and change the connection to the backend database, and you're done. The rest of the components in the application neither know nor care about this; they still see the same cursor regardless of the mechanism used to access the data.

We'll take a close look at CursorAdapter in the Advanced document. However, in the meantime, here's an example (CursorAdapterExample.prg) that gets certain fields for Brazilian customers from the Customers table in the Northwind database. The cursor is updateable, so if you make changes in the cursor, close it, then run the program again, you'll see that your changes were saved to the backend.

```
local loCursor as CursorAdapter, ;
  laErrors[1]
loCursor = createobject('CursorAdapter')
with loCursor
  .Alias              = 'Customers'
  .DataSourceType     = 'ODBC'
  .DataSource         = sqlstringconnect('driver=SQL Server;' + ;
    'server=(local);database=Northwind;uid=sa;pwd=;trusted_connection=no')
  .SelectCmd          = "select CUSTOMERID, COMPANYNAME, CONTACTNAME " + ;
    "from CUSTOMERS where COUNTRY = 'Brazil'"
  .KeyFieldList       = 'CUSTOMERID'
  .Tables             = 'CUSTOMERS'
  .UpdatableFieldList = 'CUSTOMERID, COMPANYNAME, CONTACTNAME'
  .UpdateNameList     = 'CUSTOMERID CUSTOMERS.CUSTOMERID, ' + ;
    'COMPANYNAME CUSTOMERS.COMPANYNAME, CONTACTNAME CUSTOMERS.CONTACTNAME'
  if .CursorFill()
    browse
  else
    aerror(laErrors)
    messagebox(laErrors[2])
  endif .CursorFill()
endwith
```

Advantages

The advantages of CursorAdapters are essentially the combination of those of all of the other technologies.

- Depending on how it's set up (if it's completely self-contained, for example), opening a cursor from a CursorAdapter subclass can almost be as easy as opening a remote view: you simply instantiate the subclass and call the CursorFill method (you could even call that from Init to make it a single-step operation).

- It's easier to convert an existing application to use CursorAdapters than to use cursors created with SPT.

- Like remote views, you can add a CursorAdapter to the DataEnvironment of a form or report and take advantage of the visual support the DE provides: dragging and dropping fields to automatically create controls, easily binding a control to a field by selecting it from a combobox in the Properties Window, and so on.

- It's easy to update the backend with changes: assuming the properties of the view have been set up properly, you simply call TABLEUPDATE().

- Because the result set created by a CursorAdapter is a VFP cursor, they can be used anywhere in VFP: in a grid, a report, processed in a SCAN loop, and so forth. This is true even if the data source comes from ADO and XML, because the CursorAdapter automatically takes care of conversion to and from a cursor for you.

- You have a lot of flexibility in data access, such as calling stored procedures or middle-tier objects.

- You can change the connection information on the fly as needed.

- You can change the SQL SELECT statement as needed.

- You don't need a DBC.

- They can work with ODBC, ADO, XML, or native tables, allowing you to take advantage of the benefits of any of these technologies or even switch technologies as required.

- Although you have to code for it yourself (this is discussed in more detail under Disadvantages), you have greater control over how updates are done. For example, you might use a SQL SELECT statement to create the cursor but call a stored procedure to update the backend tables.

- You can manage your own connections.

Disadvantages

There aren't a lot of disadvantages for CursorAdapters:

- You can't use a nice visual tool like the View Designer to create CursorAdapters, although the CursorAdapter Builder is a close second.

- Like other types of VFP cursors, you can't pass the result set outside the current data session. However, since CursorAdapters are really for the UI layer, that isn't much of an issue.
- They're awkward to use in reports; we'll discuss this in more detail in the Advanced document.
- Like all new technologies, there's a learning curve that must be mastered.

When to Use

Because CursorAdapters create VFP cursors, you won't likely do much with them in the middle tier of an n-tier application. However, in the UI layer, I see CursorAdapters replacing other techniques for all remote data access and even replacing Cursors in new applications that might one day be upsized.

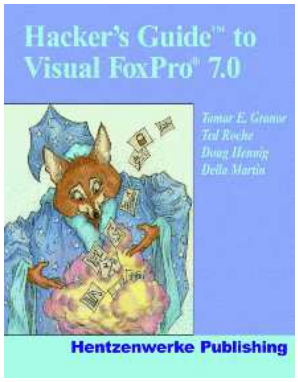
Summary

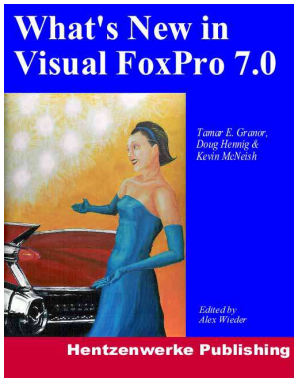
This document discussed the advantages and disadvantages of ODBC (whether remote views or SQL passthrough), ADO, and XML as means of accessing non-VFP data such as SQL Server or Oracle. As usual, which mechanism you should choose for a particular application depends on many factors. However, using the new CursorAdapter technology in VFP 8 makes the transition to remote data access easier, and makes it easier to switch between mechanisms should that need arise.

In the "Data Strategies in VFP: Advanced" document, we'll discuss the CursorAdapter class in detail, looking at specifics of accessing native data or non-VFP data using ODBC, ADO, and XML. We'll also look at creating reusable data classes, and discuss how to use CursorAdapters in reports.

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and co-author of the award-winning Stonefield Query. He is co-author (along with Tamar Granor, Ted Roche, and Della Martin) of "The Hacker's Guide to Visual FoxPro 7.0" and co-author (along with Tamar Granor and Kevin McNeish) of "What's New in 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 writes the monthly "Reusable Tools" column in FoxTalk. 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).





Doug Hennig
Partner
Stonefield Systems Group Inc.
1112 Winnipeg Street, Suite 200
Regina, SK Canada S4R 1J6
Phone: (306) 586-3341 Fax: (306) 586-5080
Email: dhennig@stonefield.com
Web: www.stonefield.com

Data Strategies in VFP: Advanced

By Doug Hennig

Introduction

In the "Data Strategies in VFP: Introduction" document, we examined the different mechanisms for accessing non-VFP data (such as SQL Server) in VFP applications: remote views, SQL passthrough, ADO, XML, and the CursorAdapter class added in VFP 8. In this document, we'll look at CursorAdapter in a lot more detail and discuss the concept of reusable data classes. In addition, we'll take a brief look at the new XMLAdapter base class and see how it can help exchange data with other sources, such as ADO.NET.

CursorAdapter

In my opinion, CursorAdapter is one of the biggest new features in VFP 8. The reasons I think they're so cool are:

- They make it easy to use ODBC, ADO, or XML, even if you're not very familiar with these technologies.
- They provide a consistent interface to remote data regardless of the mechanism you choose.
- They make it easy to switch from one mechanism to another.

Here's an example of the last point. Suppose you have an application that uses ODBC with CursorAdapters to access SQL Server data, and for some reason you want to change to use ADO instead. All you need to do is change the DataSourceType of the CursorAdapters and change the connection to the backend database, and you're done. The rest of the components in the application neither know nor care about this; they still see the same cursor regardless of the mechanism used to access the data.

Let's start examining CursorAdapters by looking at their properties, events, and methods (PEMs).

PEMS

We won't discuss all of the properties, events, and methods of the CursorAdapter class here, just the more important ones. See the VFP documentation for the complete list.

DataSourceType

This property is a biggie: it determines the behavior of the class and what kinds of values to put into some of the other properties. The valid choices are "Native", which indicates that you're using native tables, or "ODBC", "ADO", or "XML", which means you're using the appropriate mechanism to access the data. You likely won't use "Native", since you would probably use a

Cursor object rather than CursorAdapter, but this setting would make it easier to later upsize an application.

DataSource

This is the means to access the data. VFP ignores this property when DataSourceType is set to "Native" or "XML". For ODBC, set DataSource to a valid ODBC connection handle (meaning you have to manage the connection yourself). In the case of ADO, DataSource must be an ADO RecordSet that has its ActiveConnection object set to an open ADO Connection object (again, you have to manage this yourself).

UseDEDataSource

If this property is set to .T. (the default is .F.), you can leave the DataSourceType and DataSource properties alone since the CursorAdapter will use the DataEnvironment's properties instead (VFP 8 adds DataSourceType and DataSource to the DataEnvironment class as well). An example of when you'd set this to .T. is when you want all the CursorAdapters in a DataEnvironment to use the same ODBC connection.

SelectCmd

In the case of everything but XML, this is the SQL SELECT command used to retrieve the data. In the case of XML, this can either be a valid XML string that can be converted into a cursor (using an internal XMLTOCURSOR() call) or an expression (such as a UDF) that returns a valid XML string.

CursorSchema

This property holds the structure of the cursor in the same format as you'd use in a CREATE CURSOR command (everything between the parentheses in such a command). Here's an example: CUST_ID C(6), COMPANY C(30), CONTACT C(30), CITY C(25). Although it's possible to leave this blank and tell the CursorAdapter to determine the structure when it creates the cursor, it works better if you fill CursorSchema in. For one thing, if CursorSchema is blank or incorrect, you'll either get errors when you open the DataEnvironment of a form or you won't be able to drag and drop fields from the CursorAdapter to the form to create controls. Fortunately, the CursorAdapter Builder that comes with VFP can automatically fill this in for you.

AllowDelete, AllowInsert, AllowUpdate, and SendUpdates

These properties, which default to .T., determine if deletes, inserts, and updates can be done and if changes are sent to the data source.

KeyFieldList, Tables, UpdatableFieldList, and UpdateNameList

These properties, which serve the same purpose as the same-named CURSORSETPROP() properties for views, are required if you want VFP to automatically update the data source with changes made in the cursor. KeyFieldList is a comma-delimited list of fields (without aliases)

that make up the primary key for the cursor. Tables is a comma-delimited list of tables. UpdatableFieldList is a comma-delimited list of fields (without aliases) that can be updated. UpdateNameList is a comma-delimited list that matches field names in the cursor to field names in the table. The format for UpdateNameList is as follows: CURSORFIELDNAME1 TABLE.FIELDNAME1, CURSORFIELDNAME2 TABLE.FIELDNAME2, ... Note that even if UpdatableFieldList doesn't contain the name of the primary key of the table (because you don't want that field updated), it must still exist in UpdateNameList or updates won't work.

*Cmd, *CmdDataSource, *CmdDataSourceType

If you want to specifically control how VFP deletes, inserts, or updates records in the data source, you can assign the appropriate values to these sets of properties (replace the * above with Delete, Insert, and Update).

CursorFill(UseCursorSchema, NoData, Options, Source)

This method creates the cursor and fills it with data from the data source (although you can pass .T. for the NoData parameter to create an empty cursor). Pass .T. for the first parameter to use the schema defined in CursorSchema or .F. to create an appropriate structure from the data source (in my opinion, this behavior is reversed). MULTILOCKS must be set on or this method will fail. If CursorFill fails for any reason, it returns .F. rather than raising an error; use AERROR() to determine what went wrong (although be prepared for some digging, since the error messages you get often aren't specific enough to tell you exactly what the problem is).

CursorRefresh()

This method is similar to the REQUERY() function: it refreshes the cursor's contents.

Before*() and After*()

Pretty much every method and event has before and after "hook" events that allow you to customize the behavior of the CursorAdapter. For example, in AfterCursorFill, you can create indexes for the cursor so they're always available. In the case of the Before events, you can return .F. to prevent the action that trigger it from occurring (this is similar to database events).

Here's an example (CursorAdapterExample.prg) that gets certain fields for Brazilian customers from the Customers table in the Northwind database that comes with SQL Server. The cursor is updateable, so if you make changes in the cursor, close it, and then run the program again, you'll see that your changes were saved to the backend.

```
local loCursor as CursorAdapter, ;
  laErrors[1]
loCursor = createobject('CursorAdapter')
with loCursor
  .Alias              = 'Customers'
  .DataSourceType     = 'ODBC'
  .DataSource         = sqlstringconnect('driver=SQL Server;' + ;
    'server=(local);database=Northwind;uid=sa;pwd=;trusted_connection=no')
  .SelectCmd          = "select CUSTOMERID, COMPANYNAME, CONTACTNAME " + ;
```

```
    "from CUSTOMERS where COUNTRY = 'Brazil'"
  .KeyFieldList       = 'CUSTOMERID'
  .Tables             = 'CUSTOMERS'
  .UpdatableFieldList = 'CUSTOMERID, COMPANYNAME, CONTACTNAME'
  .UpdateNameList     = 'CUSTOMERID CUSTOMERS.CUSTOMERID, ' + ;
    'COMPANYNAME CUSTOMERS.COMPANYNAME, CONTACTNAME CUSTOMERS.CONTACTNAME'
  if .CursorFill()
    browse
  else
    aerror(laErrors)
    messagebox(laErrors[2])
  endif .CursorFill()
endwith
```

DataEnvironment and Form Changes

To support the new CursorAdapter class, several changes have been made to the DataEnvironment and Form classes and their designers.

First, as I mentioned earlier, the DataEnvironment class now has DataSource and DataSourceType properties. It doesn't use these properties itself but they're used by any CursorAdapter member that has UseDEDataSource set to .T. Second, you can now create DataEnvironment subclasses visually using the Class Designer (woo hoo!).

As for forms, you can now specify a DataEnvironment subclass to use by setting the new DEClass and DEClassLibrary properties. If you do this, anything you've done with the existing DataEnvironment (cursors, code, etc.) will be lost, but at least you're warned first. A cool new feature of forms that's somewhat related is the BindControls property; setting this to .F. in the Property Window means that VFP won't try to data bind the controls at init time, only when you set BindControls to .T. What's this good for? Well, how many times have you cursed that parameters are passed to Init, which fires after all the controls have been initialized and bound to their ControlSources? What if you want to pass a parameter to a form that tells it which table to open or other things that affect the ControlSources? This new property makes this issue a snap.

Other Changes

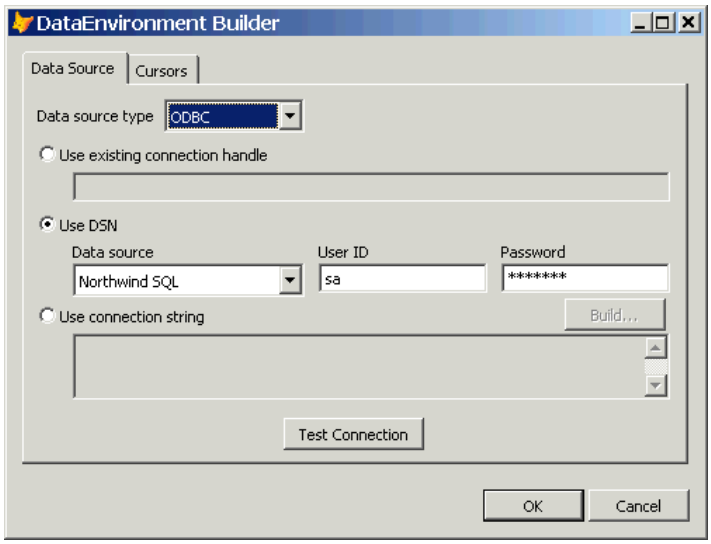
CURSORGETPROP('SourceType') returns a new range of values: if the cursor was created with CursorFill, the value is 100 + the old value (for example, 102 for remote data). If the cursor was created with CursorAttach (which allows you to attach an existing cursor to a CursorAdapter object), the value is 200 + the old value. If the data source is an ADO RecordSet, the value is 104 (CursorFill) or 204 (CursorAttach).

Builders

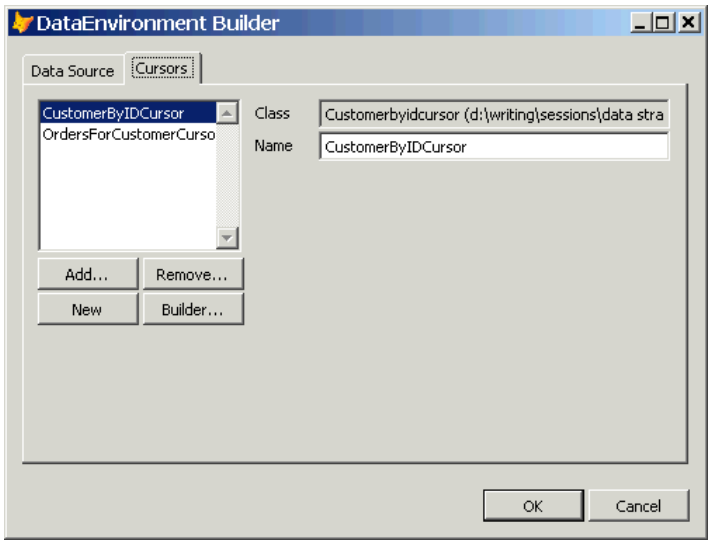
VFP includes DataEnvironment and CursorAdapter builders that make it easier to work with these classes.

The DataEnvironment Builder is brought up in the usual way: by right-clicking on the DataEnvironment of a form or on a DataEnvironment subclass in the Class Designer and

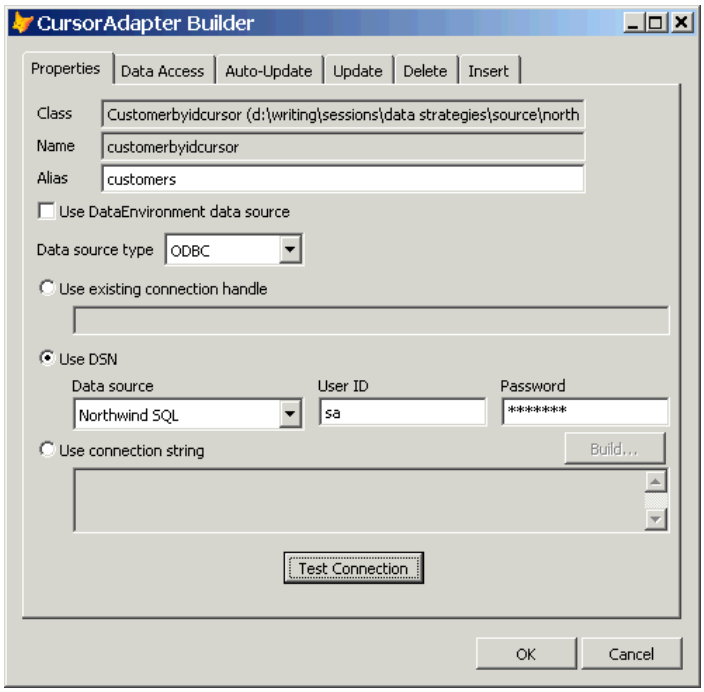
choosing Builder. The “Data Source” page of the DataEnvironment Builder is where you set data source information. Choose the desired data source type and where the data source comes from. If you choose “Use existing connection handle” (ODBC) or “Use existing ADO RecordSet” (ADO), specify the expression containing the data source (such as “goConnectionMgr.nHandle”). You can also choose to use one of the DSNs on your system or a connection string. The Build button, which is only enabled if you choose “Use connection string” for ADO, displays the Data Link Properties dialog, which you can use to build the connection string visually. If you select either “Use DSN” or “Use connection string”, the builder will generate code in the BeforeOpenTables method of the DataEnvironment to create the desired connection. If you choose “Native”, you can select a VFP database container as a data source; in that case, the generated code will ensure the database is open (you can also use free tables as the data source).



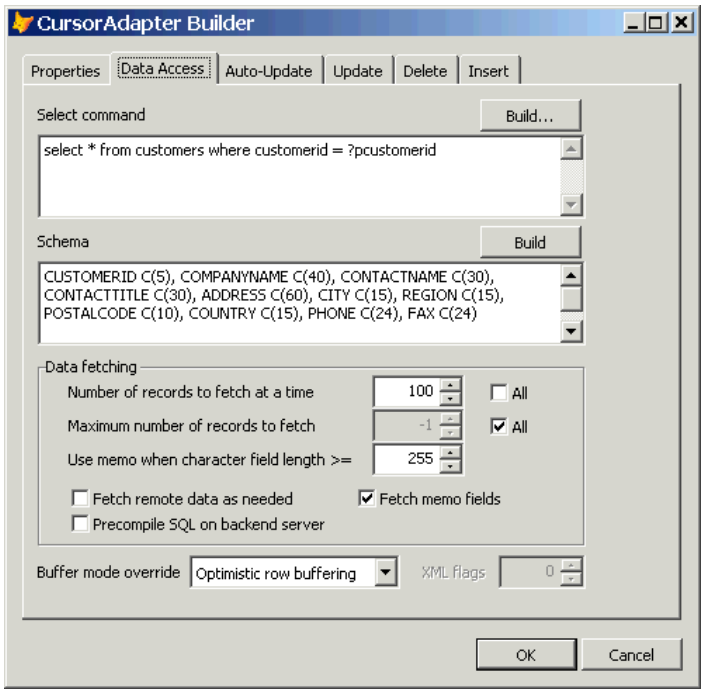
The “Cursors” page allows you to maintain the CursorAdapter members of the DataEnvironment (Cursor objects don’t show up, nor can they be added, in the builder). The Add button allows you to add a CursorAdapter subclass to the DataEnvironment, while New create a new base class CursorAdapter. Remove deletes the select CursorAdapter and Builder invokes the CursorAdapter Builder for the selected CursorAdapter. You can change the name of the CursorAdapter object, but you’ll need the CursorAdapter Builder for any other properties.



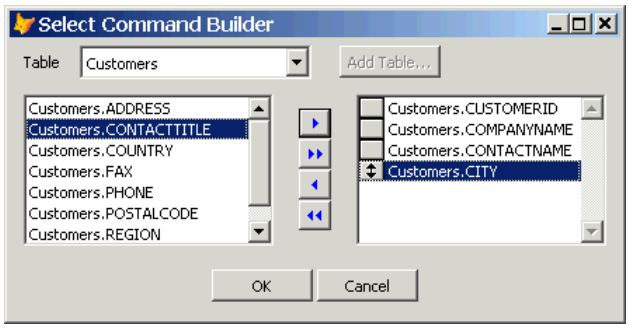
The CursorAdapter Builder is also invoked by choosing Builder from the shortcut menu. The "Properties" page shows the class and name of the object (Name can only be changed if the builder is brought up from a DataEnvironment, since it's read-only for a CursorAdapter subclass), the alias of the cursor it'll create, whether the DataEnvironment's data source should be used or not, and connection information if not. As with the DataEnvironment Builder, the CursorAdapter Builder will generate code to create the desired connection (in the CursorFill method in this case) if you select either "Use DSN" or "Use connection string".



The “Data Access” page allows you to specify the SelectCmd, CursorSchema, and other properties. If you have specified connection information, you can click on the Build button for SelectCmd to display the Select Command Builder, which makes it easy to create the SelectCmd.

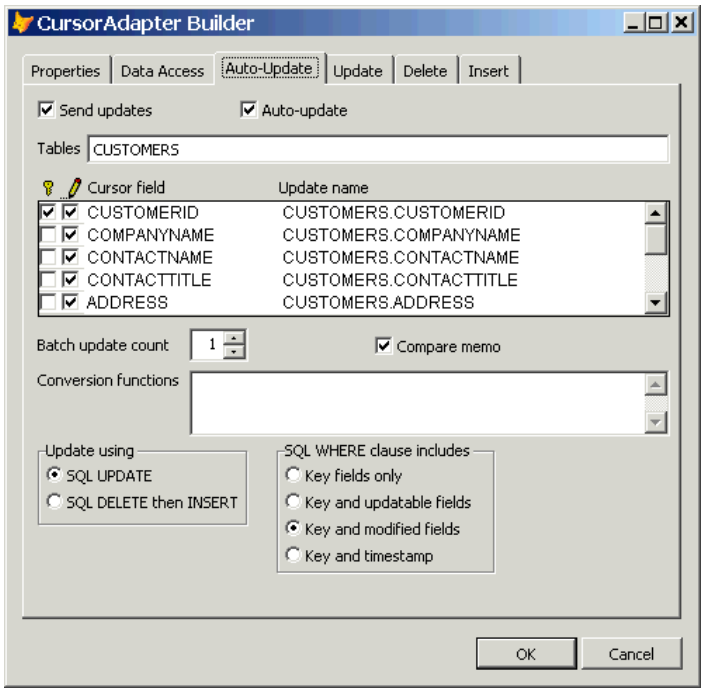


The Select Command Builder makes short work of building a simple SELECT statement. Choose the desired table from the table dropdown, then move the appropriate fields to the selected side. In the case of a native data source, you can add tables to the Table combobox (such as if you want use free tables). When you choose OK, the SelectCmd will be filled with the appropriate SQL SELECT statement.

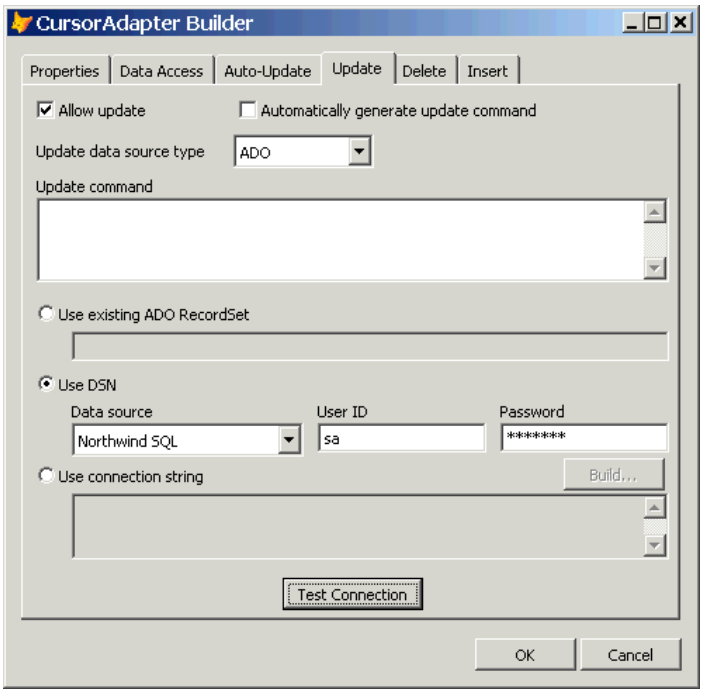


Click on the Build button for the CursorSchema to have this property filled in for you automatically. In order for this to work, the builder actually creates a new CursorAdapter object, sets the properties appropriately, and calls CursorFill to create the cursor. If you don’t have a live connection to the data source or CursorFill fails for some reason (such as an invalid SelectCmd), this obviously won’t work.

Use the Auto-Update page to set the properties necessary for VFP to automatically generate update statements for the data source. The Tables property is automatically filled in from the tables specified in SelectCmd, and the fields grid is filled in from the fields in CursorSchema. Like the View Designer, you select which are the key fields and which fields are updatable by checking the appropriate column in the grid. You can also set other properties, such as functions to convert the data in certain fields of the cursor before sending it to the data source.



The Update, Insert, and Delete pages have a nearly identical appearance. They allow you to specify values for the sets of Update, Delete, and Insert properties. This is especially important for XML, for which VFP can't automatically generate update statements.



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 DataSourceType 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 DBMS such as SQL Server.

When DataSourceType is set to "Native", VFP ignores DataSource. 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) 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.

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

```
local loCursor as CursorAdapter, ;
  laErrors[1]
loCursor = createobject('CursorAdapter')
with loCursor
  .Alias          = 'CustomerHistory'
  .DataSourceType = 'ODBC'
  .DataSource     = sqlstringconnect('driver=SQL Server;server=(local);' + ;
    'database=Northwind;uid=sa;pwd=;trusted_connection=no')
  .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'
  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 UpdateGram 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.UpdateGram" 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).

```
local loWS as dataserver 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 name of a SQL Server 2000 XML template and the customer ID to retrieve is passed to a UDF called GetNWXML, which we'll look at in a moment. For UpdateCmd, VFP passed the UpdateGram 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              = 'GetNWXML([customersbyid.xml?customerid=ALFKI])'
  .Tables                 = 'CUSTOMERS'
  .UpdatableFieldList     = 'CUSTOMERID, COMPANYNAME, CONTACTNAME, ' + ;
    'CONTACTTITLE, ADDRESS, CITY, REGION, POSTALCODE, COUNTRY, PHONE, FAX'
  .UpdateCmdDataSourceType = 'XML'
  .UpdateCmd              = 'SendNWXML(This.UpdateGram)'
  .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
```

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 appendix for details on configuring IIS to work with SQL Server).

Here's the code for GetNWXML. It uses an MSXML2.XMLHTTP object to access a SQL Server 2000 XML template on a Web server and returns the results. The name of the template (and optionally any query parameters) is passed as a parameter to this code.

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

SendNWXML looks similar, 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.

CursorAdapter and DataEnvironment Subclasses

As is normally the case in VFP, I've created subclasses of CursorAdapter and DataEnvironment that I'll use rather than the base classes.

SFCursorAdapter

SFCursorAdapter (in SFDataClasses.vcx) is a subclass of CursorAdapter that has some additional functionality added:

- It can automatically handle parameterized queries; you can define the parameter values as static (a constant value) or dynamic (an expression, such as "=Thisform.txtName.Value", that's evaluated when the cursor is opened or refreshed).
- It can automatically create indexes on the cursor after it's opened.
- It does some special things for ADO, such as setting DataSource to an ADO RecordSet, setting the ActiveConnection property of the RecordSet to an ADO Connection object, and creating and passing an ADO Command object to CursorFill when a parameterized query is used.
- It provides some simple error handling (the cErrorMessage property is filled with the error message).
- It has Update and Release methods that are missing in CursorAdapter.

Let's take a look at this class.

The Init method creates two collections (using the new Collection base class, which maintains collections of things), one for parameters that may be needed for the SelectCmd property, and one for tags that should be created automatically after the cursor is opened. It also sets MULTILOCKS on, since that's required for CursorAdapter cursors.

```
with This

* Create parameters and tags collections.

  .oParameters = createobject('Collection')
  .oTags       = createobject('Collection')

* Ensure MULTILOCKS is set on.
```

```
  set multilocks on
endwith
```

The AddParameter method adds a parameter to the parameters collection. Pass this method the name of the parameter (this should match the name as it appears in the SelectCmd property) and optionally the value of the parameter (if you don't pass it now, you can set it later by using the GetParameter method). This code shows a couple of new features in VFP 8: the new Empty base class, which has no PEMs, making it ideal for lightweight objects, and the ADDPROPERTY() function, which acts like the AddProperty method for those objects that don't have that method.

```
lparameters tcName, ;
  tuValue
local loParameter
loParameter = createobject('Empty')
addproperty(loParameter, 'Name',  tcName)
addproperty(loParameter, 'Value', tuValue)
This.oParameters.Add(loParameter, tcName)
```

Use the GetParameter method to return a specific parameter object; this is normally used when you want to set the value to use for the parameter.

```
lparameters tcName
local loParameter
loParameter = This.oParameters.Item(tcName)
return loParameter
```

The SetConnection method is used to set the DataSource property to the desired connection. If DataSourceType is "ODBC", pass the connection handle. If it's "ADO", DataSource needs to be an ADO RecordSet with its ActiveConnection property set to an open ADO Connection object, so pass the Connection object and SetConnection will create the RecordSet and set its ActiveConnection to the passed object.

```
lparameters tuConnection
with This
  do case
    case .DataSourceType = 'ODBC'
      .DataSource = tuConnection
    case .DataSourceType = 'ADO'
      .DataSource = createobject('ADODB.RecordSet')
      .DataSource.ActiveConnection = tuConnection
  endcase
endwith
```

To create the cursor, call the GetData method rather than CursorFill, since it handles parameters and errors automatically. Pass .T. to GetData if you want the cursor created but not filled with data. The first thing this method does is create privately-scoped variables with the same names and values as the parameters defined in the parameters collection (the GetParameterValue method called from here returns either the Value of the parameter object or the evaluation of the Value if it starts with an "="). Next, if we're using ADO and there are any parameters, the code creates an ADO Command object and sets its ActiveConnection to the Connection object, then passes the Command object to the CursorFill method; the CursorAdapter requires this for

parameterized ADO queries. If we're not using ADO or we don't have any parameters, the code simply calls CursorFill to fill the cursor. Note that .T. is passed to CursorFill, telling it to use CursorSchema, if CursorSchema is filled in (this is the behavior I wish the base class had). If the cursor was created, the code calls the CreateTags method to create the desired indexes for the cursor; if not, it calls the HandleError method to handle any errors that occurred.

```
lparameters tlNoData
local loParameter, ;
  lcName, ;
  luValue, ;
  llUseSchema, ;
  loCommand, ;
  llReturn
with This

* If we're supposed to fill the cursor (as opposed to creating an empty one),
* create variables to hold any parameters. We have to do it here rather than
* in a method since we want them to be scoped as private.

  if not tlNoData
    for each loParameter in .oParameters
      lcName  = loParameter.Name
      luValue = .GetParameterValue(loParameter)
      store luValue to (lcName)
    next loParameter
  endif not tlNoData

* If we're using ADO and there are any parameters, we need a Command object
* to handle this.

  llUseSchema = not empty(.CursorSchema)
  if '?' $ .SelectCmd and (.DataSourceType = 'ADO' or (.UseDEDataSource and ;
    .Parent.DataSourceType = 'ADO'))
    loCommand = createobject('ADODB.Command')
    loCommand.ActiveConnection = iif(.UseDEDataSource, ;
      .Parent.DataSource.ActiveConnection, .DataSource.ActiveConnection)
    llReturn = .CursorFill(llUseSchema, tlNoData, .nOptions, loCommand)
  else

* Try to fill the cursor.

    llReturn = .CursorFill(llUseSchema, tlNoData, .nOptions)
  endif '?' $ .SelectCmd ...

* If we created the cursor, create any tags defined for it. If not, handle
* the error.

  if llReturn
    .CreateTags()
  else
    .HandleError()
  endif llReturn
endwith
return llReturn
```

The Requery method is similar to GetData, except that it refreshes the data in the cursor. Call this method rather than CursorRefresh because, as with GetData, it handles parameters and errors.

```
local loParameter, ;
  lcName, ;
  luValue, ;
  llReturn
with This

* Create variables to hold any parameters. We have to do it here rather than
* in a method since we want them to be scoped as private.

  for each loParameter in .oParameters
    lcName  = loParameter.Name
    luValue = .GetParameterValue(loParameter)
    store luValue to (lcName)
  next loParameter

* Refresh the cursor; handle any error if it failed.

  llReturn = .CursorRefresh()
  if not llReturn
    .HandleError()
  endif not llReturn
endwith
return llReturn
```

The Update method is simple: it simply calls TABLEUPDATE() to attempt to update the original data source and calls HandleError if it failed.

```
local llReturn
llReturn = tableupdate(1, .F., This.Alias)
if not llReturn
  This.HandleError()
endif not llReturn
return llReturn
```

There are a few methods we won't look at here; feel free to examine them yourself. AddTag adds information about an index you want created after the cursor is created to the tags collection, while CreateTags, which is called from GetData, uses the information in that collection in INDEX ON statements. HandleError uses AERROR() to determine what went wrong and puts the second element of the error array into the cErrorMessage property.

Let's look at a couple of examples of using this class. The first one (taken from TestCursorAdapter.prg) gets all records from the Customers table in the Northwind database. This code isn't all that different from what you'd use for a base class CursorAdapter (you'd have to pass .F. for the first parameter to CursorFill since the CursorSchema isn't filled in).

```
loCursor = newobject('SFCursorAdapter', 'SFDataClasses')
with loCursor

* Connect to the SQL Server Northwind database and get customers records.
```

```
  .DataSourceType = 'ODBC'
  .DataSource     = sqlstringconnect('driver=SQL Server;server=(local);' + ;
    'database=Northwind;uid=sa;pwd=;trusted_connection=no')
  .Alias          = 'Customers'
  .SelectCmd      = 'select * from customers'
  if .GetData()
    browse
  else
    messagebox('Could not get the data. The error message was:' + ;
      chr(13) + chr(13) + .cErrorMessage)
  endif .GetData()
endwith
```

The next example (also taken from TestCursorAdapter.prg) uses the ODBC version of the SFConnectionMgr, which we looked at in the "Data Strategies in VFP: Introduction" document, to manage the connection. It also uses a parameterized statement for SelectCmd, shows how the AddParameter method allows you to handle parameters, and demonstrates how you can automatically create tags for the cursor using the AddTag method.

```
loConnMgr = newobject('SFConnectionMgrODBC', 'SFRemote')
with loConnMgr
  .cDriver   = 'SQL Server'
  .cServer   = '(local)'
  .cDatabase = 'Northwind'
  .cUserName = 'sa'
  .cPassword = ''
endwith
if loConnMgr.Connect()
  loCursor = newobject('SFCursorAdapter', 'SFDataClasses')
  with loCursor
    .DataSourceType = 'ODBC'
    .SetConnection(loConnMgr.GetConnection())
    .Alias     = 'Customers'
    .SelectCmd = 'select * from customers where country = ?pcountry'
    .AddParameter('pcountry', 'Brazil')
    .AddTag('CustomerID', 'CustomerID')
    .AddTag('Company',    'upper(CompanyName)')
    .AddTag('Contact',    'upper(ContactName)')
    if .GetData()
      messagebox('Brazilian customers in CustomerID order')
      set order to CustomerID
      go top
      browse
      messagebox('Brazilian customers in Contact order')
      set order to Contact
      go top
      browse
      messagebox('Canadian customers')
      loParameter = .GetParameter('pcountry')
      loParameter.Value = 'Canada'
      .Requery()
      browse
    else
      messagebox('Could not get the data. The error message was:' + ;
```

```
        chr(13) + chr(13) + .cErrorMessage)
    endif .GetData()
  endwith
else
  messagebox(loConnMgr.cErrorMessage)
endif loConnMgr.Connect()
```

SFDataEnvironment

SFDataEnvironment (also in SFDataClasses.vcx) is much simpler than SFCursorAdapter, but has some useful functionality added:

- The GetData method calls the GetData method of all SFCursorAdapter members so you don't have to call them individually.
- Similarly, the Requery and Update methods call the Requery and Update methods of every SFCursorAdapter member.
- Like SFCursorAdapter, the SetConnection method sets DataSource to an ADO RecordSet and sets the ActiveConnection property of the RecordSet to an ADO Connection object. However, it also calls the SetConnection method of any SFCursorAdapter member that has UseDEDataSource set to .F.
- It provides some simple error handling (a cErrorMessage property is filled with the error message).
- It has a Release method.

GetData is very simple: it just calls the GetData method of any member object that has that method.

```
lparameters tlNoData
local loCursor, ;
  llReturn
for each loCursor in This.Objects
  if pemstatus(loCursor, 'GetData', 5)
    llReturn = loCursor.GetData(tlNoData)
    if not llReturn
      This.cErrorMessage = loCursor.cErrorMessage
      exit
    endif not llReturn
  endif pemstatus(loCursor, 'GetData', 5)
next loCursor
return llReturn
```

SetConnection is a little more complicated: it calls the SetConnection method of any member object that has that method and that has UseDEDataSource set to .F., then uses code similar to that in SFCursorAdapter to set its own DataSource if any of the CursorAdapters have UseDEDataSource set to .T.

```
lparameters tuConnection
local llSetOurs, ;
  loCursor, ;
  llReturn
with This

* Call the SetConnection method of any CursorAdapter that isn't using our
* DataSource.

  llSetOurs = .F.
  for each loCursor in .Objects
    do case
      case upper(loCursor.BaseClass) <> 'CURSORADAPTER'
      case loCursor.UseDEDataSource
        llSetOurs = .T.
      case pemstatus(loCursor, 'SetConnection', 5)
        loCursor.SetConnection(tuConnection)
    endcase
  next loCursor

* If we found any CursorAdapters that are using our DataSource, we'll need to
* set our own.

  if llSetOurs
    do case
      case .DataSourceType = 'ODBC'
        .DataSource = tuConnection
      case .DataSourceType = 'ADO'
        .DataSource = createobject('ADODB.RecordSet')
        .DataSource.ActiveConnection = tuConnection
    endcase
  endif llSetOurs
endwith
```

Requery and Update are almost identical to GetData, so we won't bother looking at them.

TestDE.prg shows the use of SFDataEnvironment as a container for a couple of SFCursorAdapter classes. Since this example uses ADO, each SFCursorAdapter needs its own DataSource, so UseDEDataSource is set to .F. Notice that a single call to the DataEnvironment SetConnection method takes care of setting the DataSource property for each CursorAdapter.

```
loConnMgr = newobject('SFConnectionMgrADO', 'SFRemote')
with loConnMgr
  .cDriver   = 'SQLOLEDB.1'
  .cServer   = '(local)'
  .cDatabase = 'Northwind'
  .cUserName = 'sa'
  .cPassword = ''
endwith
if loConnMgr.Connect()
  loDE = newobject('SFDataEnvironment', 'SFDataClasses')
  with loDE
    .NewObject('CustomersCursor', 'SFCursorAdapter', 'SFDataClasses')
    with .CustomersCursor
      .Alias          = 'Customers'
```

```
      .SelectCmd      = 'select * from customers'
      .DataSourceType = 'ADO'
    endwith
    .NewObject('OrdersCursor', 'SFCursorAdapter', 'SFDataClasses')
    with .OrdersCursor
      .Alias          = 'Orders'
      .SelectCmd      = 'select * from orders'
      .DataSourceType = 'ADO'
    endwith
    .SetConnection(loConnMgr.GetConnection())
    if .GetData()
      select Customers
      browse nowait
      select Orders
      browse
    else
      messagebox('Could not get the data. The error message was:' + ;
        chr(13) + chr(13) + .cErrorMessage)
    endif .GetData()
  endwith
else
  messagebox(loConnMgr.cErrorMessage)
endif loConnMgr.Connect()
```

Reusable Data Classes

Now that we have CursorAdapter and DataEnvironment subclasses to work with, let's talk about reusable data classes.

One thing VFP developers have asked Microsoft to add to VFP for a long time is reusable data environments. For example, you may have a form and a report that have exactly the same data setup, but you have to manually fill in the DataEnvironment for each one because DataEnvironments aren't reusable. Some developers (and almost all frameworks vendors) made it easier to create reusable DataEnvironments by creating DataEnvironments in code (they couldn't be subclassed visually) and using a "loader" object on the form to instantiate the DataEnvironment subclass. However, this was kind of a kludge and didn't help with reports.

Now, in VFP 8, we have the ability to create both reusable data classes, which can provide cursors from any data source to anything that needs them, and reusable DataEnvironments, which can host the data classes. As of this writing, you can't use CursorAdapter or DataEnvironment subclasses in a report, but you can programmatically add CursorAdapter subclasses (such as in the Init method of the DataEnvironment) to take advantage of reusability there.

Let's create data classes for the Northwind Customers and Orders tables. First, create a subclass of SFCursorAdapter called CustomersCursor and set the properties as shown below.

Property	**Value**

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)
KeyFieldList	CUSTOMERID
SelectCmd	select * from customers
Tables	CUSTOMERS
UpdatableFieldList	CUSTOMERID, COMPANYNAME, CONTACTNAME, CONTACTTITLE, ADDRESS, CITY, REGION, POSTALCODE, COUNTRY, PHONE, FAX
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

Note: you can use the CursorAdapter Builder to do most of the work, especially setting the CursorSchema and update properties. The trick is to turn on the “use connection settings in builder only” option, fill in the connection information so you have a live connection, then fill in the SelectCmd and use the builder to build the rest of the properties for you.

Now, anytime you need records from the Northwind Customers table, you simply use the CustomersCursor class. Of course, we haven’t defined any connection information, but that’s actually a good thing, since this class shouldn’t have to worry about things like how to get the data (ODBC, ADO, or XML) or even what database engine to use (there are Northwind databases for SQL Server, Access, and, new in version 8, VFP).

However, notice that this cursor deals with all records in the Customers table. Sometimes, you only want a specific customer. So, let's create a subclass of CustomersCursor called CustomerByIDCursor. Change SelectCmd to "select * from customers where customerid = ?pcustomerid" and put the following code into Init:

```
lparameters tcCustomerID
dodefault()
This.AddParameter('pCustomerID', tcCustomerID)
```

This creates a parameter called pCustomerID (which is the same name as the one specified in SelectCmd) and sets it to any value passed. If no value is passed, use GetParameter to return an object for this parameter and set its Value property before calling GetData.

Create an OrdersCursor class similar to CustomersCursor except that it retrieves all records from the Orders table. Then create an OrdersForCustomerCursor subclass that retrieves only those orders for a specific customer. Set SelectCmd to "select * from orders where customerid = ?pcustomerid" and put the same code into Init as CustomerByIDCursor has (since it's the same parameter).

To test how this works, run TestCustomersCursor.prg.

Example: Form

Now that we have some reusable data classes, let's put them to use. First, let's create a subclass of SFDataEnvironment called CustomersAndOrdersDataEnvironment that contains CustomerByIDCursor and OrdersForCustomerCursor classes. Set AutoOpenTables to .F. (because we need to set connection information before the tables can be opened) and UseDEDataSource for both CursorAdapters to .T. This DataEnvironment can now be used in a form to show information about a specific customer, including its orders.

Let's create such a form. Create a form called CustomerOrders.scx (it's included with the sample files accompanying this document), set DEClass and DEClassLibrary to CustomersAndOrdersDataEnvironment so we use our reusable DataEnvironment. Put the following code into the Load method:

```
#define ccDATASOURCETYPE 'ADO'
with This.CustomersAndOrdersDataEnvironment

* Set the DataEnvironment data source.

  .DataSourceType = ccDATASOURCETYPE

* If we're using ODBC or ADO, create a connection manager and open a
* connection to the Northwind database.

  if .DataSourceType $ 'ADO,ODBC'
    This.AddProperty('oConnMgr')
    This.oConnMgr = newobject('SFConnectionMgr' + ccDATASOURCETYPE, ;
      'SFRemote')
    with This.oConnMgr
      .cDriver   = iif(ccDATASOURCETYPE = 'ADO', 'SQLOLEDB.1', ;
```

```
        'SQL Server')
      .cServer   = '(local)'
      .cDatabase = 'Northwind'
      .cUserName = 'sa'
      .cPassword = ''
    endwith
    if not This.oConnMgr.Connect()
      messagebox(oConnMgr.cErrorMessage)
      return .F.
    endif not This.oConnMgr.Connect()

* If we're using ADO, each cursor must have its own datasource.

    if .DataSourceType = 'ADO'
      .CustomerByIDCursor.UseDEDataSource      = .F.
      .CustomerByIDCursor.DataSourceType       = 'ADO'
      .OrdersForCustomerCursor.UseDEDataSource = .F.
      .OrdersForCustomerCursor.DataSourceType  = 'ADO'
    endif .DataSourceType = 'ADO'

* Set the DataSource to the connection.

    .SetConnection(This.oConnMgr.GetConnection())

* If we're using XML, change the SelectCmd to call the GetNWXML function
* instead.

  else
    .CustomerByIDCursor.SelectCmd = 'GetNWXML([customersbyid.xml?' + ;
      'customerid=] + pCustomerID)'
    .CustomerByIDCursor.UpdateCmdDataSourceType = 'XML'
    .CustomerByIDCursor.UpdateCmd = 'SendNWXML(This.UpdateGram)'
    .OrdersForCustomerCursor.SelectCmd = 'GetNWXML([ordersforcustomer.' + ;
      'xml?customerid=] + pCustomerID)'
    .OrdersForCustomerCursor.UpdateCmdDataSourceType = 'XML'
    .OrdersForCustomerCursor.UpdateCmd = 'SendNWXML(This.UpdateGram)'
  endif .DataSourceType $ 'ADO,ODBC'

* Specify that the value for the cursor parameters will be filled from the
* CustomerID textbox.

  loParameter       = .CustomerByIDCursor.GetParameter('pCustomerID')
  loParameter.Value = '=Thisform.txtCustomerID.Value'
  loParameter       = .OrdersForCustomerCursor.GetParameter('pCustomerID')
  loParameter.Value = '=Thisform.txtCustomerID.Value'

* Create empty cursors and display an error message if we failed.

  if not .GetData(.T.)
    messagebox(.cErrorMessage)
    return .F.
  endif not .GetData(.T.)
endwith
```

This looks like a lot of code, but most of it is there for demo purposes to allow switching to different data access mechanisms.

This code creates a connection manager to handle the connection, either ADO, ODBC, or XML, depending on the ccDATASOURCETYPE constant, which you can change to try each of the mechanisms. In the case of ADO, since each CursorAdapter must have its own DataSource, the UseDEDataSource and DataSourceType properties are set for each one. The code then calls the SetConnection method to set up the connection information. In the case of XML, the SelectCmd, UpdateCmdDataSourceType, and UpdateCmd properties must be changed as discussed earlier. Next, the code uses the GetParameter method of both CursorAdapter objects to set the Value of the pCustomerID parameter to the contents of a textbox in the form. Note the use of the "=" in the value; this means the Value property will be evaluated every time it's needed, so we essentially have a dynamic parameter (saving the need to constantly change the parameter to the current value as the user types in the textbox). Finally, the GetData method is called to create empty cursors so the data binding of the controls will work.

Drop a textbox on the form and name it txtCustomerID. Put the following code into its Valid method:

```
with Thisform
  .CustomersAndOrdersDataEnvironment.Requery()
  .Refresh()
endwith
```

This causes the cursors to be requeried and the controls to be refreshed when a customer ID is entered.

Drop a label on the form, put it beside the textbox, and set its Caption to "Customer ID".

Drag the CompanyName, ContactName, Address, City, Region, PostalCode, and Country fields from the Customers cursor in the DataEnvironment to the form to create controls for these fields. Then select the OrderID, EmployeeID, OrderDate, RequiredDate, ShippedDate, ShipVia, and Freight fields in the Orders cursor and drag them to the form to create a grid.

That's it. Run the form and enter "ALFKI" for the customer ID. When you tab out of the textbox, you should see the customer address information and orders appear. Try changing something about the customer or order, then closing the form, running it again, and entering "ALFKI" again. You should see that the changes you made were written to the backend database without any effort on your part.

Cool, huh? That wasn't a lot more work than creating a form based on local tables or views. Even better, try changing the ccDATASOURCETYPE constant to "ADO" or "XML" and notice that the form looks and works exactly the same. That's the whole point of CursorAdapters!

Example: Report

Let's try a report. The example discussed here was taken from CustomerOrders.frx that accompanies this document. The biggest issue here is that, unlike a form, we can't tell a report to use a DataEnvironment subclass, nor can we drop CursorAdapter subclasses in the DataEnvironment. So, we'll have to put some code into the report to add CursorAdapter subclasses to the DataEnvironment. Although it might seem logical to put this code into the

BeforeOpenTables event of the report's DataEnvironment, that won't actually work because for reasons I don't understand, BeforeOpenTables fires on every page when you preview the report. So, we'll put the code into the Init method.

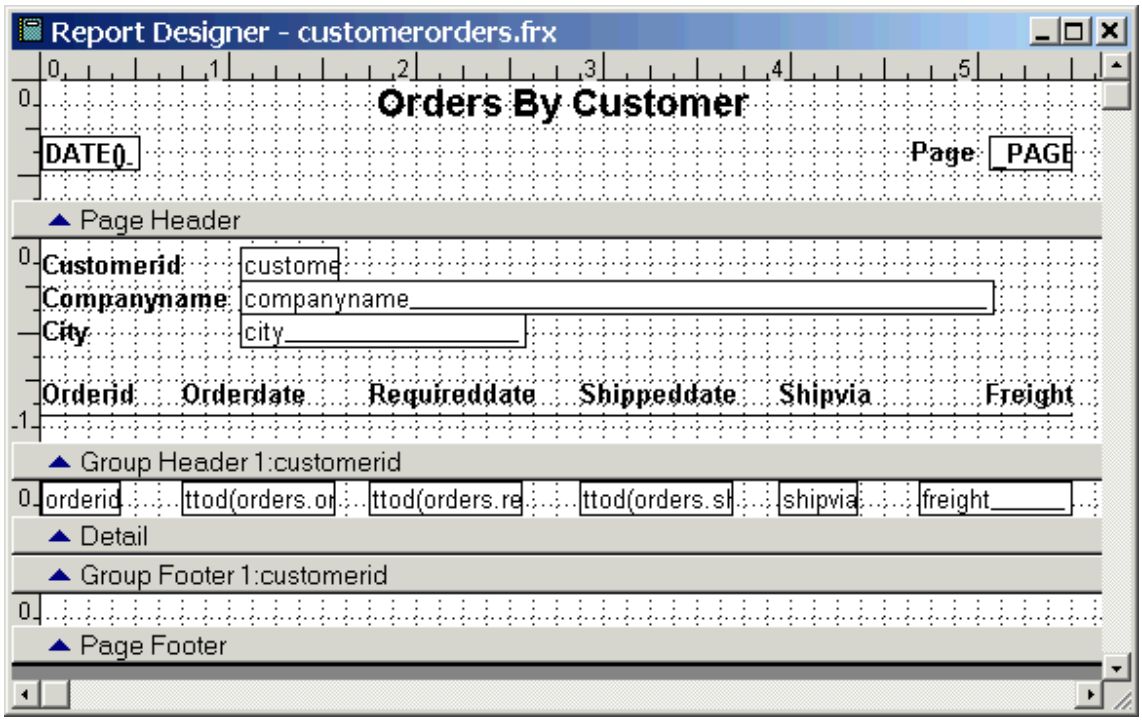
```
#define ccDATASOURCETYPE 'ODBC'
with This
  set safety off

* Set the DataEnvironment data source.

  .DataSourceType = ccDATASOURCETYPE

* Create CursorAdapter objects for Customers and Orders.

  .NewObject('CustomersCursor', 'CustomersCursor', 'NorthwindDataClasses')
  .CustomersCursor.AddTag('CustomerID', 'CustomerID')
  .NewObject('OrdersCursor', 'OrdersCursor', 'NorthwindDataClasses')
  .OrdersCursor.AddTag('CustomerID', 'CustomerID')

* If we're using ODBC or ADO, create a connection manager and open a
* connection to the Northwind database.

  if .DataSourceType $ 'ADO,ODBC'
    .AddProperty('oConnMgr')
    .oConnMgr = newobject('SFConnectionMgr' + ccDATASOURCETYPE, ;
      'SFRemote')
    with .oConnMgr
      .cDriver   = iif(ccDATASOURCETYPE = 'ADO', 'SQLOLEDB.1', ;
        'SQL Server')
      .cServer   = '(local)'
      .cDatabase = 'Northwind'
      .cUserName = 'sa'
      .cPassword = ''
    endwith
    if not .oConnMgr.Connect()
      messagebox(.oConnMgr.cErrorMessage)
      return .F.
    endif not .oConnMgr.Connect()

* If we're using ADO, each cursor must have its own datasource.

    if .DataSourceType = 'ADO'
      .CustomersCursor.UseDEDataSource = .F.
      .CustomersCursor.DataSourceType  = 'ADO'
      .CustomersCursor.SetConnection(.oConnMgr.GetConnection())
      .OrdersCursor.UseDEDataSource = .F.
      .OrdersCursor.DataSourceType  = 'ADO'
      .OrdersCursor.SetConnection(.oConnMgr.GetConnection())
    else
      .CustomersCursor.UseDEDataSource = .T.
      .OrdersCursor.UseDEDataSource    = .T.
      .DataSource = .oConnMgr.GetConnection()
    endif .DataSourceType = 'ADO'
    .CustomersCursor.SetConnection(.oConnMgr.GetConnection())
    .OrdersCursor.SetConnection(.oConnMgr.GetConnection())
```

```
* If we're using XML, change the SelectCmd to call the GetNWXML function
* instead.

  else
    .CustomersCursor.SelectCmd      = 'GetNWXML([getallcustomers.xml])'
    .CustomersCursor.DataSourceType = 'XML'
    .OrdersCursor.SelectCmd         = 'GetNWXML([getallorders.xml])'
    .OrdersCursor.DataSourceType    = 'XML'
  endif .DataSourceType $ 'ADO,ODBC'

* Get the data and display an error message if we failed.

  if not .CustomersCursor.GetData()
    messagebox(.CustomersCursor.cErrorMessage)
    return .F.
  endif not .CustomersCursor.GetData()
  if not .OrdersCursor.GetData()
    messagebox(.OrdersCursor.cErrorMessage)
    return .F.
  endif not .OrdersCursor.GetData()

* Set a relation from Customers to Orders.

  set relation to CustomerID into Customers
endwith
```

This code looks similar to that of the form. Again, the majority of the code is to handle different data access mechanisms. However, there is some additional code because we can't use a DataEnvironment subclass and have to code the behavior ourselves.

Now, how do we conveniently put the fields on the report? Since the CursorAdapter don't exist in the DataEnvironment at design time, we can't just drag fields from them to the report. Here's a tip: create a PRG that creates the cursors and leaves them in scope (either by suspending or making the CursorAdapter objects public), then use the Quick Report function to put fields with the proper sizes on the report.

Create a group on CUSTOMERS.CUSTOMERID and check "Start each group on a new page". Then lay out the report to look similar to the following:



XMLAdapter

In addition to CursorAdapter, VFP 8 has three new base classes to improve VFP's support for XML: XMLAdapter, XMLTable, and XMLField. XMLAdapter provides a means of converting data between XML and VFP cursors. It has a lot more capabilities than the CURSORTOXML() and XMLTOCURSOR() functions, including support for hierarchical XML and the ability to consume types of XML those function don't support such as ADO.NET DataSets. XMLTable and XMLField are child objects that provide the ability to fine-tune the schema for the XML data. In addition, XMLTable has an ApplyDiffgram method intended to allow VFP to consume updategrams and diffgrams, something that was missing from VFP 7.

To give you an idea of its capabilities, I created an ASP.NET Web Service that returns an ADO.NET DataSet, and then used an XMLAdapter object in VFP to consume that DataSet. Here's now I did this.

First, in Visual Studio.NET, I dragged the Northwind Customers table from the Server Explorer to a new ASP.NET Web Service project I called NWWebService. This automatically created two objects, SQLConnection1 and SQLDataAdapter1. I then added the following code to the existing generated code:

```
<WebMethod()> Public Function GetAllCustomers() As DataSet
    Dim loDataSet As New DataSet()
    Me.SqlConnection1.Open()
    Me.SqlDataAdapter1.Fill(loDataSet)
    Return loDataSet
End Function
```

I built the project to generate the appropriate Web Service files in the NWWebService virtual directory (which VS.NET automatically created for me).

To consume this Web Service in VFP, I used the IntelliSense Manager to register a Web Service called "Northwind.NET", pointing it at "http://localhost/NWWebService/NWWebService.asmx?WSDL" as the location of the WSDL file. I then created the following code (in XMLAdapterWebService.prg) to call the Web Service and convert the ADO.NET DataSet into a VFP cursor.

```
local loWS as Northwind.NET, ;
  loXMLAdapter as XMLAdapter, ;
  loTable as XMLTable

* Get the .NET DataSet from a .NET Web Service.

loWS = NEWOBJECT("Wsclient",HOME()+"ffc\_webservices.vcx")
loWS.cWSName = "Northwind.NET"
loWS = loWS.SetupClient("http://localhost/NWWebService/NWWebService.asmx" + ;
  "?WSDL", "NWWebService", "NWWebServiceSoap")
loXML = loWS.GetAllCustomers()

* Create an XMLAdapter and load the data.

loXMLAdapter = createobject('XMLAdapter')
loXMLAdapter.XMLSchemaLocation = '1'
loXMLAdapter.LoadXML(loXML.Item(0).parentnode.xml)

* If we succeeded in loading the XML, create and browse a cursor from each
* table object.

if loXMLAdapter.IsLoaded
  for each loTable in loXMLAdapter.Tables
    loTable.ToCursor()
    browse
    use
  next loTable
endif loXMLAdapter.IsLoaded
```

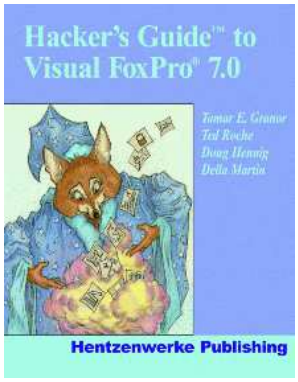
Note that in order to use XMLAdapter, you need MSXML 4.0 Service Pack 1 or later installed on your system. You can download this from the MSDN Web site (go to http://msdn.microsoft.com and search for MSXML).

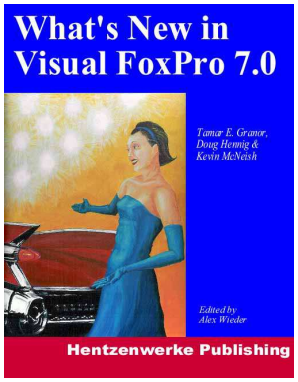
Summary

I think CursorAdapter is one of the biggest and most exciting enhancements in VFP 8 because it provides a consistent and easy-to-use interface to remote data, plus it allows us to create reusable data classes. I'm sure once you work with them, you'll find them as exciting as I do.

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and co-author of the award-winning Stonefield Query. He is co-author (along with Tamar Granor, Ted Roche, and Della Martin) of "The Hacker's Guide to Visual FoxPro 7.0" and co-author (along with Tamar Granor and Kevin McNeish) of "What's New in 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 writes the monthly "Reusable Tools" column in FoxTalk. 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).



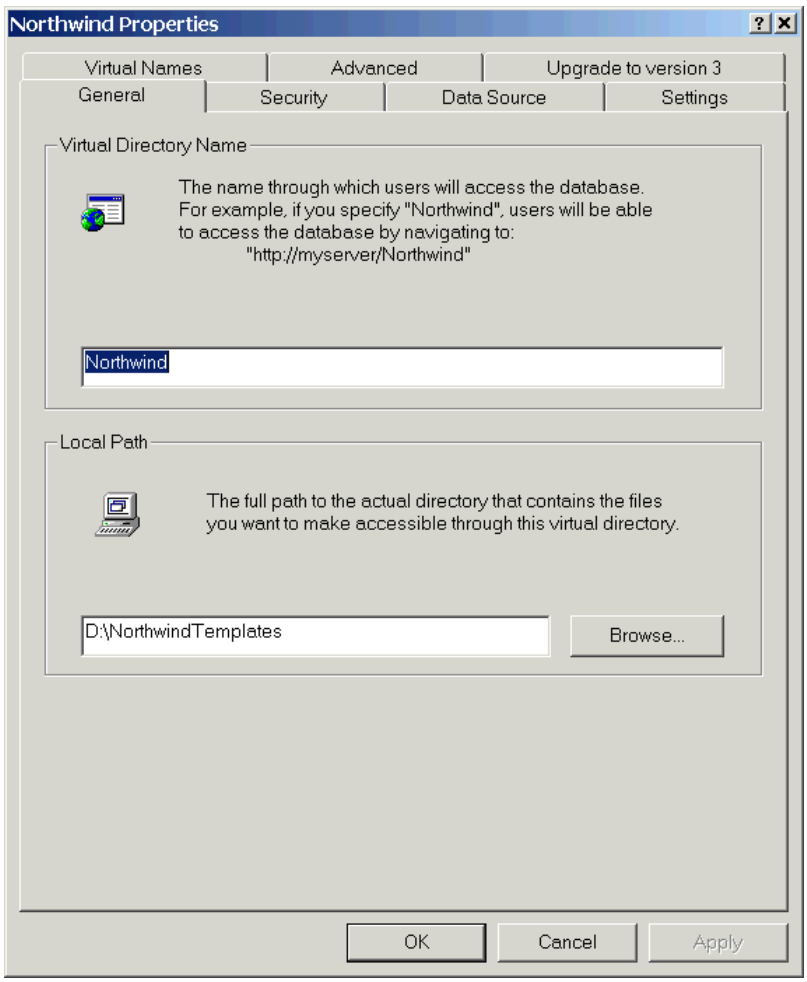


Doug Hennig
Partner
Stonefield Systems Group Inc.
1112 Winnipeg Street, Suite 200
Regina, SK Canada S4R 1J6
Phone: (306) 586-3341 Fax: (306) 586-5080
Email: dhennig@stonefield.com
Web: www.stonefield.com

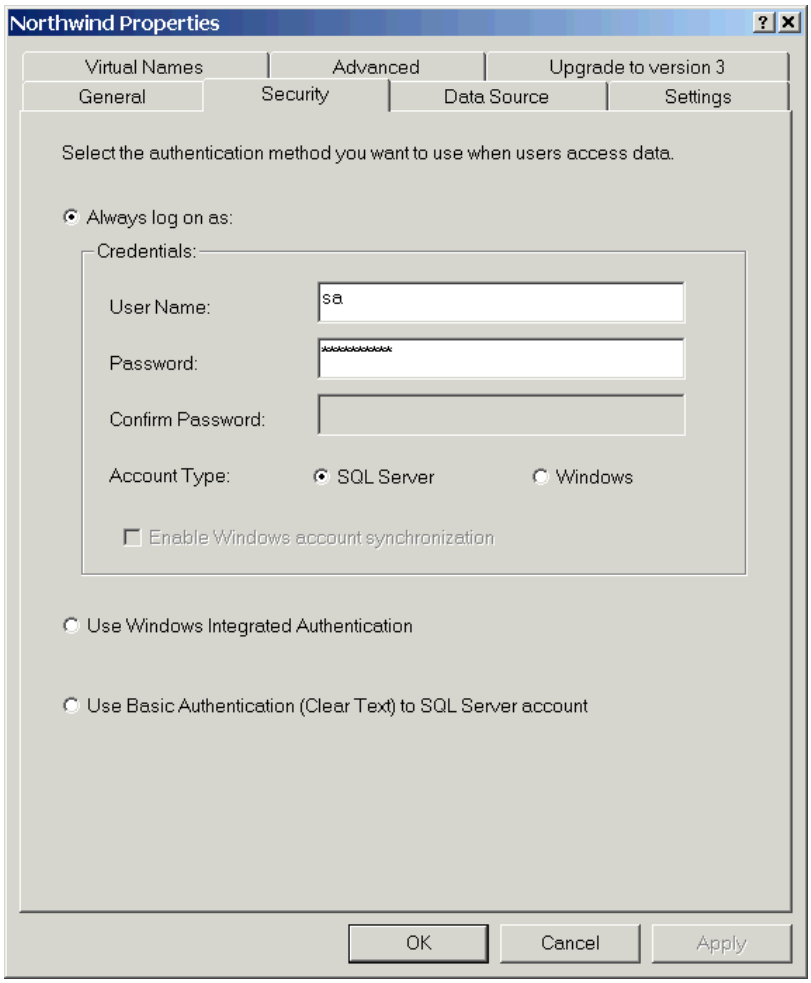
Appendix: Setting Up SQL Server 2000 XML Access

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).

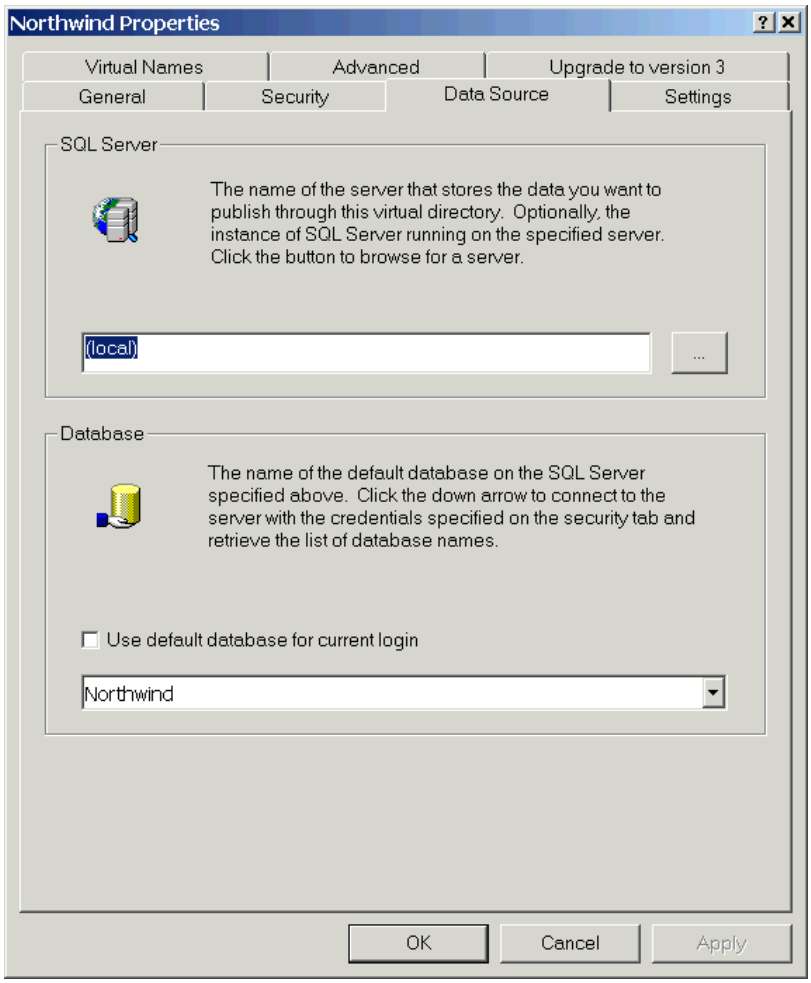
Next, you have to set up an IIS virtual directory. To do this, choose the Configure IIS Support shortcut in the SQLXML 3.0 folder under Start, Programs in your Windows Taskbar. Expand the node for your server, choose the Web site to work with, and then right-click and choose New, Virtual Directory. In the General tab of the dialog that appears, enter the name of the virtual directory and its physical path. For the samples in this document, 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 templates files included with this article to the Template subdirectory.



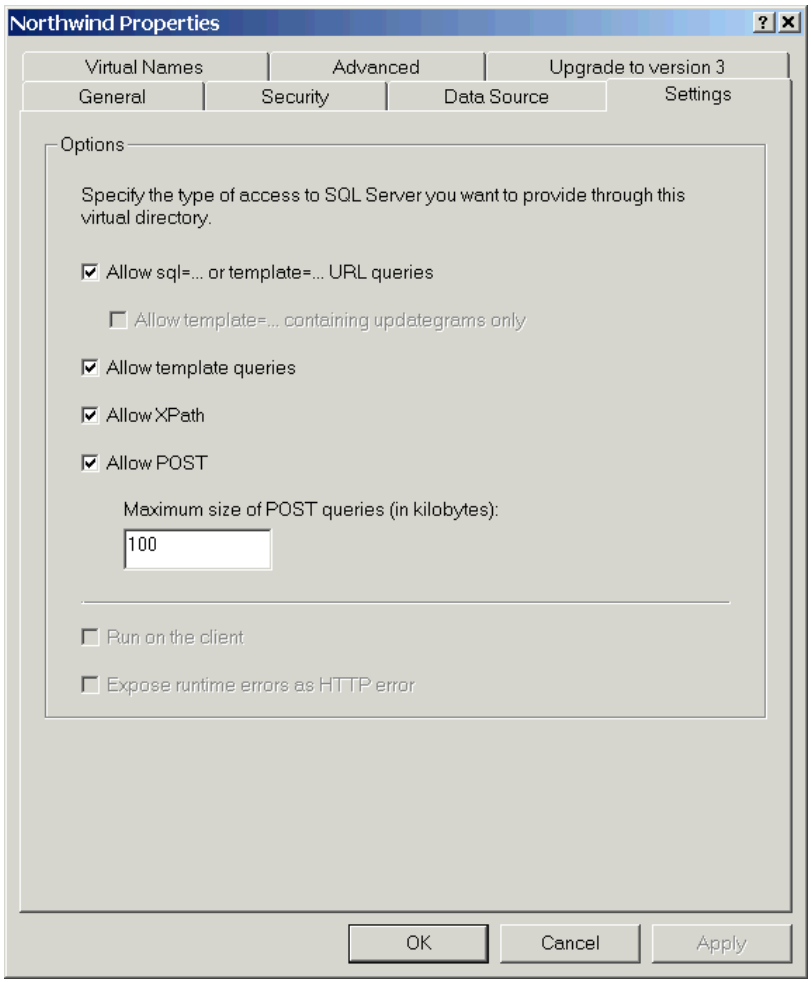
In the Security tab, enter the appropriate information to access SQL Server, such as the user name and password or the specific authentication mechanism you want to use.



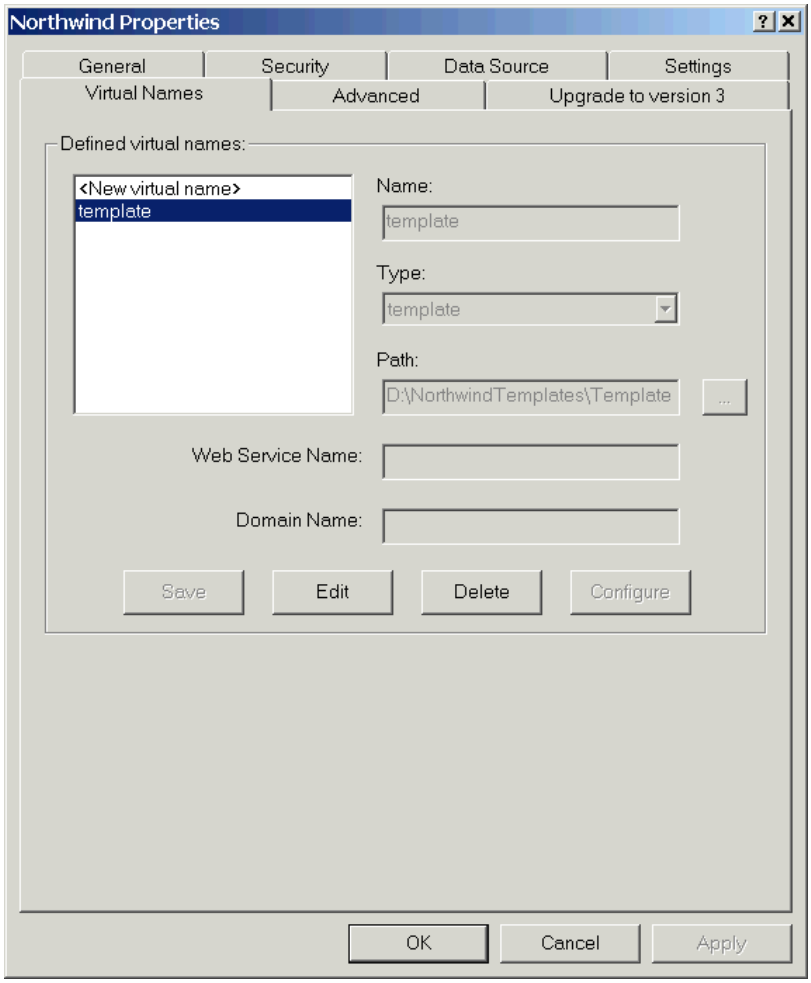
In the Data Source tab, choose the server and, if desired, the database to use. For the examples for this document, choose the Northwind database.



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 document, use “template”) and physical path (which should be a subdirectory of the virtual directory; for this document, use “Template”) to use for templates. Click on OK



To test this, bring up your browser and type the following URL: http://localhost/northwind/template/getallcustomers.xml. You should see something like the following: