

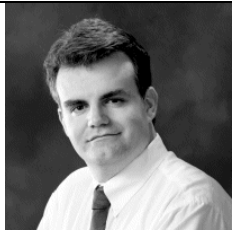
## SQL Server 2000 XML Integration

By Rob Vieira

### Introduction

The Extensible Markup Language (XML) is a relatively new and powerful markup based information definition language that is taking the world by storm. The self-describing nature of XML in a text-based format provides a level of flexibility in information exchange that was previously only found in our dreams. Through XML, formats can easily be defined to facilitate information to be exported into one common format, but be easily transformed into a variety of other forms.

Microsoft SQL Server™ is one of the leading Relational Database Management Systems (RDBMS) for the NT and Windows 2000 Server platforms. SQL Server 2000 is the latest release of this Enterprise class product, and provides all new functionality to support XML data operations and exchanges. This document strives to present an overview of both XML and the new XML integration features that can be found in SQL Server 2000.



#### **Rob Vieira**

The author of two Professional SQL Server Programming books, Rob Vieira has been developing since 1980, and has been involved in databases since 1985. Currently, a Principal Engineer with STEP Technology's Technology and Innovation group, Rob focuses on system architecture - primarily for data related technologies. Rob currently resides in Vancouver, Washington with his wife Nancy, 12 year old daughter Ashley, and new baby Adrianna.

Contact Rob at: [robv@ProfessionalSQL.com](mailto:robv@ProfessionalSQL.com)

---

## An XML Primer

XML has actually been around for a few years now, but, while the talk was big, its actual usage was not what it could have been. At issue, was the fact that there was very little in the way of ratified standards to deal with XML – no sooner would a vendor provide an XML feature of some kind, then it would suddenly be out of date versus the latest change in the proposed standards. Now that well-defined standards exist, the 3<sup>rd</sup> party support (XML parsers, integration into mainstream products, standardized XML implementations within major industries, etc. etc.) is exploding. In turn, XML solutions are becoming increasingly more workable and more desirable as an information solution.

XML is, from a raw performance standpoint, not a very good place to *store* data. It is, however, a positively spectacular way of making data *useful* – As such, the ways of utilizing XML will likely continue to grow, and grow, and grow.

So what, then, is XML? For a very large percentage of the developer and general IT community, the response would probably be that it is a new web technology that is going to replace HTML – this couldn't be any less true. XML is first and foremost an *information* technology. It is *not* a web specific technology at all. Instead, it just tends to be thought of that way for several reasons – such as:

- XML is a markup language, and looks very similar to HTML to the untrained eye.
- XML is often easily transformed into HTML. As such, it has become a popular way to keep the information part of a page, with a final transformation into HTML only on request – a separate transformation can take place based on criteria (such as what browser is asking for the information).
- One of the first widely used products to support XML was Microsoft's Internet Explorer.
- The Internet is quite often used as a way to exchange information, and that's something that XML is ideally suited for.

Like HTML, XML is a text based markup language. Indeed, they are both derived from the same original language, called Standard Generalized markup Language – or SGML. SGML has been around for much longer than the Internet (at least as we think of the internet today), and is most often used in the printing industry or in government related documentation. Simply put, the "S" in SGML certainly doesn't stand for simple – SGML is anything but intuitive and is downright difficult to learn (I can only read about 35% of SGML documents that I've seen. I have, however, been able to achieve a full 100% nausea rate when reading any SGML.). XML, on the other hand, tends to be reasonably easy to decipher.

So, this might have one asking the question: "Great – Where can I get a listing of XML tags?" Well, you can't – at least, not in the sense that one thinks of when they ask the question. XML has very few tags that are actually part of the language. Instead, it provides for ways of defining your own tags and for utilizing tags as defined by others. XML is largely about flexibility – which includes the ability for you to set your own rules for your XML through the use of either what's called a DTD (Document Type Definition) or, as an alternative, the use of a XML schema.

*As of this writing, the W3C had not yet made a final recommendation for XML Schemas. Microsoft has, for the time being, implemented their own brand of XML Schema, in the form of what they call XML Data Reduced documents– or XDR's. With this in mind, you'll probably want to stick with the tried and true – DTD's until the W3C makes it's recommendation and we know whether the Microsoft implementation is compatible or not.*

This might bring up the next question: "What *are* the rules?" Well, the most basic rules of XML are fairly simple. They merely state that the XML must be "well formed" (we'll get to those rules in a moment) and that it must meet any rules as defined by the DTD or XML Schema if there is one.

## ***XML Must Be "Well Formed"***

If you're used to HTML at all, then you've seen some pretty sloppy stuff as far as a tag based language goes. XML has much more strict rules about what is and isn't OK. The short rendition looks like this:

- Every XML document *must* have a unique "root" node
- Every tag must have a matching closing tag
- Tags denote an "element" – elements may have one or more attributes
- Tags cannot straddle other tags

The following is an example of a document that is "well formed":

```
<?xml version="1.0" encoding="UTF-8"?>
<ThisCouldBeCalledAnything>
  <AnElement>
    <AnotherElement AnAttribute="Some Value">
      <ASelfClosingElement/>
    </AnotherElement>
  </AnElement>
</ThisCouldBeCalledAnything>
```

Note that every element tag has a matching closing tag. There are only two exceptions to this:

- The `<?xml version="1.0" encoding="UTF-8"?>` tag. This is one of those few tags that is defined by the XML standard. It is entirely optional – you can add it (in which case it must be first) or leave it out. There are a few parts to it, but for the purposes of this paper, suffice to say that it defines the document as an XML document and also provides for the version of XML as well as the encoding scheme used.
- The `<ASelfClosingElement/>` tag. This one, just as it says, is actually properly closed. The `/` slipped in at the end indicates that the tag both opens and closes itself (`<BR/>`, for example).

In addition, elements can have "attributes" – things that further describe the element. These, like the element name, must have no embedded spaces. Their value must be enclosed in double quotes.

This is an extremely abbreviated version of what's required for your XML document to be considered to be well formed, but it pretty much covers the basics for the limited scope of this document.

## ***DTD's And XML Schemas***

A Document Type Definition, or DTD, is one method of defining a set of rules for an XML document. DTD's also exist in SGML and there is even a DTD that says specifically what HTML is *supposed* to look like (most browsers don't really enforce it). Full discussion of DTD's is outside the scope of this document, but the basic thing to understand is that DTD's provide a *user definable* set of rules for what an XML document needs to look like.

XML Schemas are an alternate approach to DTD's (the XML standard allows for either approach). Essentially, this is XML defining other XML. As of this writing, the standard for XML schemas is not finalized. Further discussion of XML Schemas is outside the scope of this document.

## SQL Server XML Overview

SQL Server 2000 provides two main approaches to providing XML integration: The FOR XML clause and XML over HTTP. Each has variations within it. In this paper, we will be briefly exploring each including how to implement each approach and some basic pros and cons that each carries with it.

### *FOR XML*

This is certainly the most basic of the two major integration models. It is essentially just an option added onto the end of the existing T-SQL SELECT statement, and provides three different options for how you want your XML formatted in the results:

- RAW: This sends each row of data in your resultset back as a single data element with the element name of "row" and with each column listed as an attribute of the "row" element. Even if you join multiple tables, RAW outputs the results with the same number of elements as you would have rows in a standard SQL query.
- AUTO: This option labels each element with either the table name or table name alias the data is sourced from. If there is data output from more than one table in the query, the data from each table is separated into separate, nested elements. If this option is used, then an additional option, ELEMENTS, is also supported if you would like column data presented as elements rather than attributes.
- EXPLICIT: This one is certainly the most complex to format your query with, but the end result is that you have a high degree of control of what the XML looks like in the end. With this option, you define something of a hierarchy to the data that's being returned, and then format your query such that each piece of data belongs to a specific hierarchy level (and gets assigned a tag accordingly) as desired.

Note that none of these options provide the required root element. If you want the XML document to be considered to be "well formed", then you will need to wrap the results with a proper opening and closing tag for your root element.

Below is a relatively simple two table query using FOR XML with the RAW option:

```
SELECT Customers.CustomerID
       Customers.CompanyName,
       Orders.OrderID,
       Orders.OrderDate
FROM Customers
JOIN Orders
  ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
FOR XML RAW
```

The output is a one-to-one match versus what we would have seen in the resultset had we ran just a standard SQL query:

```
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taquería"
OrderID="10365" OrderDate="1996-11-27T00:00:00" />
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taquería"
OrderID="10507" OrderDate="1997-04-15T00:00:00" />
```

```

<row CustomerID="ANTON" CompanyName="Antonio Moreno Taquería"
OrderID="10535" OrderDate="1997-05-13T00:00:00" />
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taquería"
OrderID="10573" OrderDate="1997-06-19T00:00:00" />
<row CustomerID="ALFKI" CompanyName="Alfreds Futterkiste" OrderID="10643"
OrderDate="1997-08-25T00:00:00" />
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taquería"
OrderID="10677" OrderDate="1997-09-22T00:00:00" />
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taquería"
OrderID="10682" OrderDate="1997-09-25T00:00:00" />
<row CustomerID="ALFKI" CompanyName="Alfreds Futterkiste" OrderID="10692"
OrderDate="1997-10-03T00:00:00" />
<row CustomerID="ALFKI" CompanyName="Alfreds Futterkiste" OrderID="10702"
OrderDate="1997-10-13T00:00:00" />
<row CustomerID="ALFKI" CompanyName="Alfreds Futterkiste" OrderID="10835"
OrderDate="1998-01-15T00:00:00" />
<row CustomerID="ANTON" CompanyName="Antonio Moreno Taquería"
OrderID="10856" OrderDate="1998-01-28T00:00:00" />
<row CustomerID="ALFKI" CompanyName="Alfreds Futterkiste" OrderID="10952"
OrderDate="1998-03-16T00:00:00" />
<row CustomerID="ALFKI" CompanyName="Alfreds Futterkiste" OrderID="11011"
OrderDate="1998-04-09T00:00:00" />

```

Note that I've cleaned up the formatting a bit versus what you would see in the Query Analyzer, but it is functionally the same. We have one element in XML for each row of data our query produced. All column information, regardless of what table was the source of the data, is represented as an attribute of the "row" element. The downside of this is that we haven't represented the true hierarchical nature of our data – orders are only placed by customers. The upside, however, is that the XML DOM – if that's the model you're using – is going to be much less deep, and, hence, will have a slightly smaller footprint in memory and will, depending on what you're doing, perform better.

Now let's contrast this with what we would have seen if we had used the AUTO option. We only need to make a minimal change to our code:

```

SELECT Customers.CustomerID,
Customers.CompanyName,
    Orders.OrderID,
    Orders.OrderDate
FROM Customers
JOIN Orders
    ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
FOR XML AUTO

```

At first, the output looks pretty similar to our RAW query. The first apparent difference is that the element name has changed to be that of the name or alias of the table that is the source of the data, but another, even more significant difference appears when we look at the XML more thoroughly:

```

<Customers CustomerID="ANTON" CompanyName="Antonio Moreno Taquería">
  <Orders OrderID="10365" OrderDate="1996-11-27T00:00:00" />
  <Orders OrderID="10507" OrderDate="1997-04-15T00:00:00" />
  <Orders OrderID="10535" OrderDate="1997-05-13T00:00:00" />
  <Orders OrderID="10573" OrderDate="1997-06-19T00:00:00" />
</Customers>
<Customers CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
  <Orders OrderID="10643" OrderDate="1997-08-25T00:00:00" />

```

```

</Customers>
<Customers CustomerID="ANTON" CompanyName="Antonio Moreno Taquería">
  <Orders OrderID="10677" OrderDate="1997-09-22T00:00:00" />
  <Orders OrderID="10682" OrderDate="1997-09-25T00:00:00" />
</Customers>
<Customers CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
  <Orders OrderID="10692" OrderDate="1997-10-03T00:00:00" />
  <Orders OrderID="10702" OrderDate="1997-10-13T00:00:00" />
  <Orders OrderID="10835" OrderDate="1998-01-15T00:00:00" />
</Customers>
<Customers CustomerID="ANTON" CompanyName="Antonio Moreno Taquería">
  <Orders OrderID="10856" OrderDate="1998-01-28T00:00:00" />
</Customers>
<Customers CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
  <Orders OrderID="10952" OrderDate="1998-03-16T00:00:00" />
  <Orders OrderID="11011" OrderDate="1998-04-09T00:00:00" />
</Customers>

```

Data that is sourced from our second table (as determined by the select list) is nested inside the data sourced from the first table. In this case, our "Orders" elements are nested inside our "Customers" elements. If a column from the Orders table were listed first in our select list, then Customers would be nested inside of Orders. If we had used the ELEMENTS option, then each element would have no attributes – instead, all attributes would be converted to an element and would be nested inside the appropriate parent tag.

The downside to using AUTO is that the model winds up being slightly more complex. The upside is that the data is more explicitly broken up into a hierarchical model. This makes life easier for situations where the elements are more significant breaking points – such as where you have a doubly sorted report (e.g. Orders sorted within Customers).

The EXPLICIT options takes much more effort to prepare, but it also rewards that effort with very fine granularity of control over what's an element and what's an attribute as well as what elements are nested in what other elements.

```

SELECT 1
        NULL
        Customers.CustomerID
        Customers.CompanyName
        NULL
        NULL
        as Tag,
        as Parent,
        as [Customer!1!CustomerID],
        as [Customer!1!CompanyName],
        as [Order!2!OrderID],
        as [Order!2!OrderDate]

FROM Customers
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'

UNION ALL

SELECT 2,
        1,
        Customers.CustomerID,
        Customers.CompanyName,
        Orders.OrderID,
        Orders.OrderDate
FROM Customers
JOIN Orders
ON Customers.CustomerID = Orders.CustomerID
WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID = 'ANTON'
ORDER BY [Customer!1!CustomerID], [Order!2!OrderID]
FOR XML EXPLICIT

```

This particular piece of code yields us exactly what we got in our AUTO example, but with just a few changes, I could change my XML into forms that AUTO wouldn't give me – for example, imagine that we wanted, for whatever reason, for our Order Date to be in a separate element, we could do that with only a minimal change. All we need to do is add the "element" keyword to the description of our OrderDate field:

```
NULL as [Order!2!OrderDate!element]
```

And suddenly we have an extra element instead of an attribute:

```
<Customer CustomerID="ALFKI" CompanyName="Alfreds Futterkiste">
  <Order OrderID="10643">
    <OrderDate>1997-08-25T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10692">
    <OrderDate>1997-10-03T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10702">
    <OrderDate>1997-10-13T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10835">
    <OrderDate>1998-01-15T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10952">
    <OrderDate>1998-03-16T00:00:00</OrderDate>
  </Order>
  <Order OrderID="11011">
    <OrderDate>1998-04-09T00:00:00</OrderDate>
  </Order>
</Customer>
<Customer CustomerID="ANTON" CompanyName="Antonio Moreno Taquería">
  <Order OrderID="10365">
    <OrderDate>1996-11-27T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10507">
    <OrderDate>1997-04-15T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10535">
    <OrderDate>1997-05-13T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10573">
    <OrderDate>1997-06-19T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10677">
    <OrderDate>1997-09-22T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10682">
    <OrderDate>1997-09-25T00:00:00</OrderDate>
  </Order>
  <Order OrderID="10856">
    <OrderDate>1998-01-28T00:00:00</OrderDate>
  </Order>
</Customer>
```

Using newer versions of ADO, the results can now be streamed to a client without having to build all the metadata that goes with a full resultset. Once at a client or a webserver, the XML

can then be transformed into about anything you would like – for example, and XHTML web page or a data file for pickup/transmission to a customer or vendor.

## **OPENXML**

SQL Server now also allows you to open a string of XML and represent it in the tabular format that is expected in SQL. This means that you can join to and XML document, or even use it as the source of input data by using an `INSERT..SELECT` or a `SELECT INTO`. `OPENXML` is a rowset function that opens your string much as other rowset functions (such as `OPENQUERY` and `OPENROWSET`) work. The major difference is that it requires that you use a couple of system-stored procedures to prepare your document and clear the memory after you're done using it.

To set up your document, you use `sp_xml_preparedocument`. This moves the string into memory and pre-parses it for optimal query performance. In calling this system sproc, you get back a handle that you will need to use every time you make a call to `OPENXML` related to this XML string. When you're all done with your XML, you need to call `sp_xml_removedocument` to clear up memory.

As an example, imagine that you are merging with another company and need to import some of their data into your system. For this example, we'll say that we're working on importing a few shippers that they have and our company (Northwind) doesn't. A sample of what our script might look like to import these from an XML document might be:

```
USE Northwind

DECLARE @idoc      int
DECLARE @xmldoc   nvarchar(4000)

-- define the XML document
SET @xmldoc = '
<ROOT>
<Shipper ShipperID="100" CompanyName="Billy Bob&apos;s Pretty Good
Shipping"/>
<Shipper ShipperID="101" CompanyName="Fred&apos;s Freight"/>
</ROOT>
'

--Load and parse the XML document in memory
EXEC sp_xml_preparedocument @idoc OUTPUT, @xmldoc

--List out what our shippers table looks like before the insert
SELECT * FROM Shippers

-- ShipperID is an IDENTITY column, so we need to allow direct updates
SET IDENTITY_INSERT Shippers ON

--See our XML data in a tabular format
SELECT * FROM OPENXML (@idoc, '/ROOT/Shipper', 0) WITH (
    ShipperID      int,
    CompanyName    nvarchar(40))

--Perform and insert based on that data
INSERT INTO Shippers
(ShipperID, CompanyName)
SELECT * FROM OPENXML (@idoc, '/ROOT/Shipper', 0) WITH (
    ShipperID      int,
    CompanyName    nvarchar(40))
```

```

--Set things back to normal
SET IDENTITY_INSERT Shippers OFF

--Now look at the Shippers table after our insert
SELECT * FROM Shippers

--Now clear the XML document from memory
EXEC sp_xml_removedocument @idoc

```

The final resultset from this looks just like what we wanted:

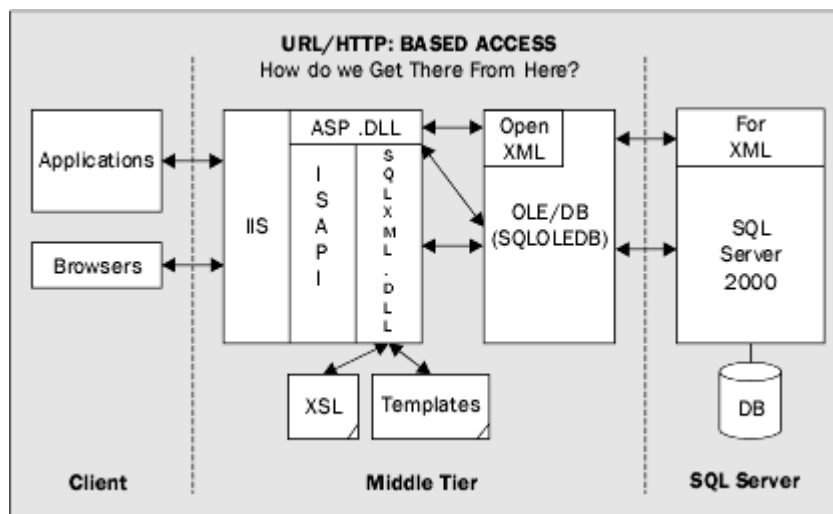
ShipperID	CompanyName	Phone
1	Speedy Express	(503) 555-9831
2	United Package	(503) 555-3199
3	Federal Shipping	(503) 555-9931
4	Speedy Shippers, Inc.	(503) 555-5566
100	Billy Bob's Pretty Good Shipping	NULL
101	Fred's Freight	NULL

OPENXML will actually allow for much more complex queries against your XML data. Our example here was fairly simple for the sake of brevity, but, with a little practice, you'll find that you can retrieve almost any kind of information from your XML data.

## HTTP Access

Access to XML formatted results is also available semi-directly through HTTP with the use of URL based access. This access requires that you also run IIS (though not necessarily on the same server). SQL Server provides an excellent management tool to facilitate the setup of HTTP: based access. Full instructions on how to set up the Northwind sample database for same-server HTTP access can be found in the SQL Server books online.

The architecture of the URL/HTTP based access is diagrammed below:



As you can see, a special ISAPI DLL is required for HTTP access. There are plans to fully integrate this functionality into SQL Server at some later date, but, for now, the ISAPI solution is effective and allowed this functionality to ship much sooner that might have otherwise been possible.

## URL based queries

URL based queries are optional with HTTP access. That is, you can allow queries to be entered right into the URL request from the browser. This clearly has security concerns that should be carefully considered prior to allowing such queries.

A sample query to provide the same results we saw in the previous section would look as follows:

```
http://ServerName/Northwind?SQL=SELECT+Customers.CustomerID,+Customers.CompanyName,+Orders.OrderID,+Orders.OrderDate+FROM+Customers+JOIN+Orders+ON+Customers.CustomerID+=+Orders.CustomerID+WHERE+Customers.CustomerID+=+'ALFKI'+OR+Customers.CustomerID+=+'ANTON'+FOR+XML+RAW&root=ROOT
```

Several things about this query are worth noting:

- The "&root=ROOT" at the end explicitly defined a root element called "ROOT". It doesn't need to be called "ROOT" – it could have been called "BillClinton" if that's what we wanted it to be – it just needs to be unique.
- The "+" signs take the place of spaces. The "%20" that is somewhat more common in URL encoding is also valid.
- The "SQL=" parameter marks the beginning of our query
- It produces exactly the same output as our original query from the previous section with the exception of this query providing the root element according to our instructions.

We could also simulate our AUTO and EXPLICIT queries from the last section by substituting the appropriate SQL.

## Using Templates

Using templates allows us to encapsulate much of the query logic and information for both ease of maintenance and for security purposes. Creating a template is as simple as defining an XML document that lists out the basic structure of your XML document and also provides the one or more queries you want to make up the results.

To stick with our example that has formed the basis of most of our queries thus far, here is a template that would achieve the same results:

```
<Root xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <sql:query>
    SELECT Customers.CustomerID,
           Customers.CompanyName,
           Orders.OrderID,
           Orders.OrderDate
    FROM Customers
    JOIN Orders
      ON Customers.CustomerID = Orders.CustomerID
    WHERE Customers.CustomerID = 'ALFKI' OR Customers.CustomerID =
'ANTON'
    ORDER BY Customers.CustomerID
    FOR XML AUTO
  </sql:query>
</Root>
```

The "xmlns" attribute in the Root element is defining what is called a "name space". Its purpose is just to make sure that when we use the word "query" later in the document, we do not create a conflict with some other name space that might also use the word "query" to mean something. We can have multiple namespaces represented in a single XML document. Each namespace has a definition indicating where to find the reference document that sets the rules for that namespace.

If we were to add the above template code into a template called CustomerOrders.xml, then we could call it by the URL reference:

<http://<ServerName>/Northwind/templates/CustomerOrders.xml>

The possibilities do not end there though – we can parameterize our templates if we need to. For example, we could alter our query to accept a parameter to look for a specific customer:

```
<Root xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <sql:header>
    <sql:param name="CustomerID">ALFKI</sql:param>
  </sql:header>
  <sql:query>
    SELECT Customers.CustomerID,
           Customers.CompanyName,
           Orders.OrderID,
           Orders.OrderDate
    FROM Customers
    JOIN Orders
      ON Customers.CustomerID = Orders.CustomerID
    WHERE Customers.CustomerID = @CustomerID
    ORDER BY Customers.CustomerID
    FOR XML AUTO
  </sql:query>
</Root>
```

Adding the parameter to our URL works much as it does for parameters pass in to a HTTP GET through the URL:

<http://<ServerName>/Northwind/templates/CustomerOrdersParam.xml?CustomerID=ANTON>

Now we receive only one CustomerID at a time, but we can change it to any customer we want:

```
<Root xmlns:sql="urn:schemas-microsoft-com:xml-sql">
<Customers CustomerID="ANTON" CompanyName="Antonio Moreno Taquería">
  <Orders OrderID="10365" OrderDate="1996-11-27T00:00:00" />
  <Orders OrderID="10507" OrderDate="1997-04-15T00:00:00" />
  <Orders OrderID="10535" OrderDate="1997-05-13T00:00:00" />
  <Orders OrderID="10573" OrderDate="1997-06-19T00:00:00" />
  <Orders OrderID="10677" OrderDate="1997-09-22T00:00:00" />
  <Orders OrderID="10682" OrderDate="1997-09-25T00:00:00" />
  <Orders OrderID="10856" OrderDate="1998-01-28T00:00:00" />
</Customers>
</Root>
```

Since we supplied a default (in this case, ALFKI), we can even leave the parameter out and still receive the information on our default customer. It's also worth noting that you are not limited to just queries in your templates – you can also use stored procedures or user-defined functions.

## ***POST***

You can also "post" templates. There are two pieces of functionality that are enabled when you allow posts in the virtual directory setup:

- Treatment of HTTP POST form variables the same as HTTP GET (the same as URL based template use)
- The ability to post templates from your HTTP forms.

While the first option above is *highly* desirable, the second one is a major security problem in situations where you aren't able to perform some form of NT security. It is extremely unfortunate that you can't do one without the other.

The main thing to keep in mind if you activate POST queries is that anyone who can send an HTTP stream to your server will effectively have all the rights of the user that the virtual directory is aliased to. The only limitations are restrictions you placed on the virtual directory user and a length limit on the size of the accepted post.

If you elect to enable this option, it is highly recommended that you set the accepted POST length to the smallest size your application can function with. This will at least limit the amount of damage that can be created by rogue queries.

Due to the risky nature of this kind of query, further discussion of this option is deemed out of scope for this document.

## ***XPath***

SQL Server 2000 also now supports the widely recognized XPath query language. XPath can be particularly useful in dealing with complex XML queries. XPath does not exist as a W3C recommendation in its own right. Instead, for W3C purposes, the elements of XPath have been broken out for use in both XSLT (which is now a formal recommendation) and Xpointer (still only a working draft as of this writing). XPath can allow many (but not all) of the same flexibility options that FOR XML EXPLICIT offers, but in a somewhat easier query format.

The concept of using XPath with SQL Server is essential similar to that of a view. A "map" is needed to relate the relational structures of a table or query to that of a XML Data Reduced - or "XDR" - Schema.

XPath and XDR are both full topics unto themselves. For purposes of this document, the goal is to make the reader aware that SQL Server 2000 does support a partial implementation of XPath, and provides the means to map the relational data stored in SQL Server to an XPath query.

## **What Else Do I Need To Know?**

To make the most out of SQL Server 2000's new XML features, both the client and server sides of the connection should be utilizing MDAC 2.6 or higher. MDAC 2.6 is installed automatically with SQL Server 2000, but client side installation is the responsibility of the developer. Keep in mind that changes in MDAC versions can cause problems with legacy software.

Features added in 2.6 that provide additional XML support include special features to facilitate the streaming of XML data. This can be highly efficient when utilized with a FOR XML query. Before completing a system design that seeks to utilize XML, a thorough consideration of the role of MDAC 2.6 and beyond should be completed.

In addition, one should visit the SQL Server home page on a regular basis to look for updates to the web based technologies (URL queries). Updates to this portion of the product have gone to a "web release" cycle in order to facilitate keeping SQL Server's web interaction technology at the leading edge of the industry.

## Summary

The XML features added in SQL Server 2000 greatly enhance the power and flexibility of SQL Server installations. The developer now has several options for taking high performance relational data and providing the stored information in a format that is both highly adaptable and easily understood.

While the functionality added in this release does not solve all of the problems of working with XML, SQL Server 2000 provides a major step forward in XML integration with RDBMS systems.

## Resources

<http://msdn.microsoft.com/sqlserver/http://www.microsoft.com/data/http://msdn.microsoft.com/xml/http://www.w3c.org>