4

Accessing SQL 2000 via HTTP

IN THE FIRST THREE CHAPTERS, WE COVERED THE XML specification, Extensible Stylesheet Language Transformations (XSLT), and the necessary setup steps for virtual roots on SQL Server. In this chapter, we'll first discuss client/server architecture to give you a feel for how the different system components, application servers, database servers, and so on interact with each other. Then we'll look at how to utilize the HTTP protocol in various ways to execute SQL statements against SQL Server. This includes the use of template files to generate XML data. Utilizing the HTTP protocol via URLs will simplify our tasks because most people in the computer industry are very familiar with this process.

This chapter will cover the following topics:

- General client/server architecture in two-, three-, and n-tiered configurations
- SQL Server 2000's HTTP capabilities
- Entities in XML and URLs
- Generating XML documents by querying SQL Server via HTTP
- Generating XML documents utilizing XML template files
- Generating XML documents utilizing stored procedures

I think it's about time to define this *template file* that we've been talking about. It's not some new language you'll have to learn, so you can relax. It also has nothing to do with the template elements of XSLT that we learned about in Chapter 2, "XSLT Stylesheets." Simply put, these templates are just XML files that contain one or more SQL statements. When these templates are applied to a database through mechanisms you'll learn about in this chapter, they help produce results in XML format.

Let's take one last look at the XML process diagram that we used in Chapter 1, "Database XML," and Chapter 2 (see Figure 4.1).

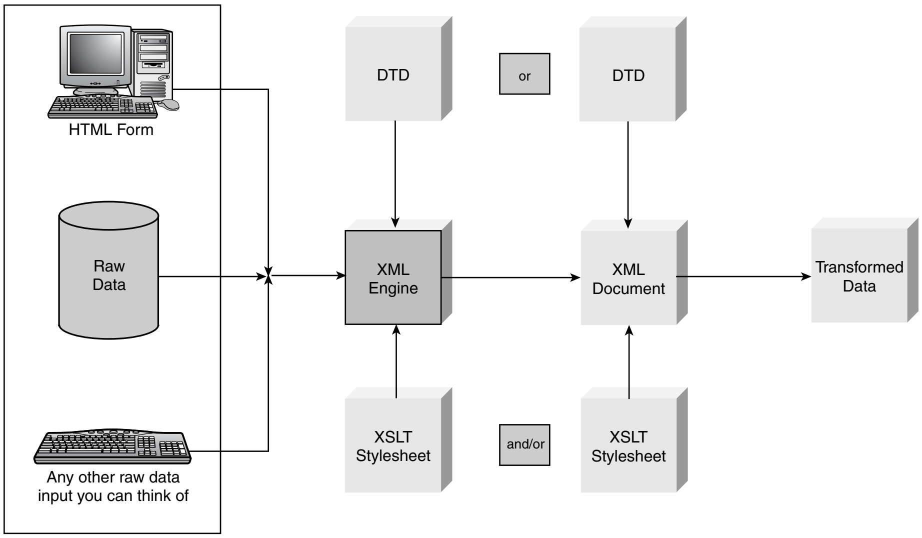


Figure 4.1 The XML process.

Yes, that's right; this is the last look. We've covered all the components in the diagram and will remain at the XML engine component for the rest of this book. The XML engine is SQL Server 2000.

Two-, Three-, and N-Tiered Architectures

To get a better understanding of how client/server components function together and therefore a better understanding of system extensibility and capability, we need to have a brief discussion of client/server architecture.

Each component of a client/server system performs one or more specific logical functions. We'll begin this discussion by defining these functions, and then we'll show you how they are distributed throughout a client/server system. These four functions are the basic building blocks of any application:

- **Data storage logic.** Most applications need to store data, whether it's a small memo file or a large database. This covers such topics as input/output (I/O) and data validation.
- **Data access logic.** This is the processing required to access stored data, which is usually in the form of SQL queries (whether they are issued from the command line of Query Analyzer or from a stored procedure).
- **Application logic.** This is the application itself, which can be simple or complex. Application logic is also referred to as a company's *business rules*.
- **Presentation logic.** This is the projection of information to the user and the acceptance of the user's input.

With these different *functional processes* now defined, we can look at how different client/server configurations split these functions up and what effect they have on extensibility. We'll begin with two-tier architecture, followed by three-tier and then n-tier architecture. Finally, we'll talk briefly about a sample SQL Server and IIS configuration.

Two-Tier Client/Server Architecture

In its simplest form, client/server architecture is a two-tier structure consisting of, believe it or not, a client component and a server component. A static Web site is a good two-tier example (see Figure 4.2).

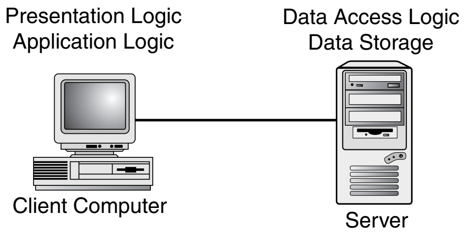


Figure 4.2 Two-tier client/server architecture.

In this configuration, the client accepts user requests and performs the application logic that produces database requests and transmits them to the server. The server accepts the requests, performs the data access logic, and transmits the results to the client. The client accepts the results and presents them to the end user.

Three-Tier Client/Server Architecture

The next step up is three-tier architecture. Take a look at Figure 4.3.

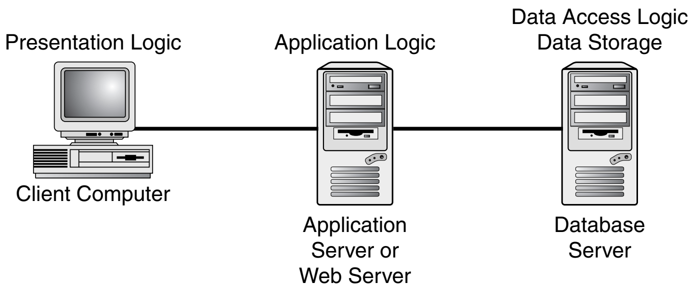


Figure 4.3 Three-tier client/server architecture.

This design utilizes three different focus points. In this case, the client is responsible for presentation logic, an application server is accountable for application logic, and a separate database server is responsible for data access logic and data storage. Many large-scale Web sites with database servers separate from Web servers are examples of this.

N-Tier Client/Server Architecture

Last but not least, there is the n-tier client/server architecture. This configuration is basically open ended (see Figure 4.4).

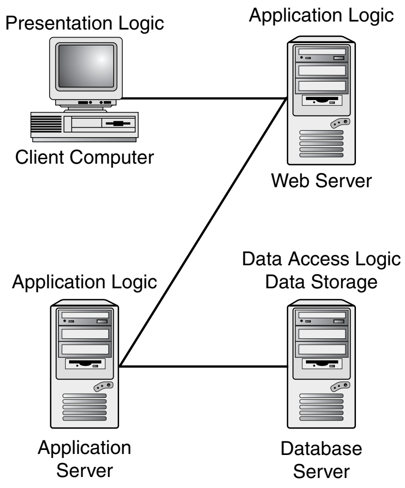


Figure 4.4 N-tier client/server architecture.

In this figure, there are more than three focus points. The client is responsible for presentation logic, a database server(s) is responsible for data access logic and data storage, and application logic is spread across two or more different sets of servers. In our example, one of the application servers is a Web server, and the other is a non-Web server. This isn't required for the architecture. Any combination of two or more types of application servers is all right.

The primary advantage of n-tiered client/server architecture, as compared to a two-tiered architecture (or a three-tiered to a two-tiered), is that it prepares an application for *load balancing*, distributing the processing among multiple servers. Also, a proper n-tier setup enables better integration with other components and ease of development, testing, and management.

Typical Microsoft Three-Tier Architecture for IIS and SQL 2000

Looking at Figure 4.5, you'll see that I have diagrammed it in a slightly different way to illustrate some important points. At first glance, you'll see that it represents three-tier client/server architecture. Nothing is new here; the client is responsible for presentation logic, a SQL Server 2000 server(s) is responsible for data access logic and data storage, and IIS contains the application logic. It's the internal workings of IIS in this instance that I want to explain a little more deeply.

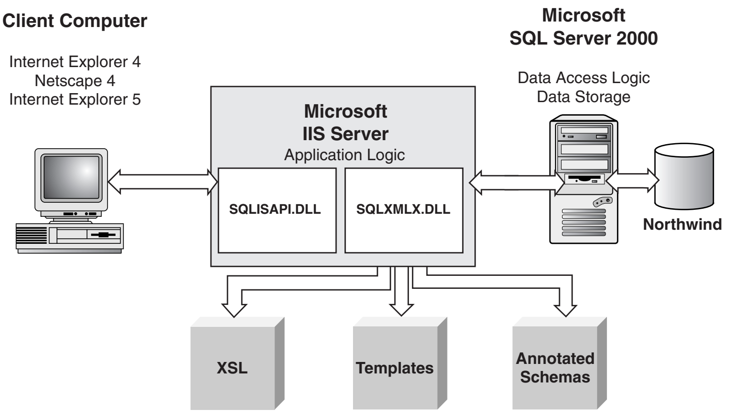


Figure 4.5 Typical Microsoft client/server architecture.

When a URL-type query is passed to IIS, it examines the virtual root contained in the URL and ensures that the SQLISAPI.DLL has been registered for this virtual root. This should have been done when the virtual root was configured via one of the two configuration methods covered in Chapter 3, "Internet Information Server and Virtual Directories."

SQLISAPI.DLL, in conjunction with other DLLs, establishes a connection with the SQL Server identified in the virtual root. After the connection is established and it is determined that the command in the URL is an XML command, the command is passed to SQLXMLX.DLL. SQLXMLX.DLL executes the command, and the results are returned. All XML functionality is contained in SQLXMLX.DLL.

If you take into account what we have just discussed and look back to Chapter 3's section "The Advanced Tab," you'll see why the configuration of the Advanced tab is so important. If the SQLISAPI.DLL file can't be found, nothing works.

As you can see in Figure 4.5, template files, schema files, and XSLT stylesheets all reside on the IIS server.

What Are Our Capabilities When Utilizing HTTP?

Now let's take a quick look at what we can accomplish by using the HTTP protocol. The rest of this chapter will then go into each of these functions in depth.

Placing a SQL Query Directly in a URL

Take a look at the following:

```
http://IISServer/Nwind?sql=SELECT+*+FROM+Employees+FOR+XML+AUTO&root=root
```

Placing a SQL query in a URL like this is simple enough, don't you think? After the URL, which points to the Nwind virtual directory, insert a question mark followed by `sql=` and then the SQL query itself. Separate all words in the query with a plus (+) sign. We'll explain the `&root` parameter in the upcoming section "Well-Formed Documents, Fragments, and `&root`."

FOR XML AUTO is a new extension to the SELECT statement making its appearance with SQL Server 2000. It will be covered in depth in Chapter 8, "OPENXML." For now, it's only necessary to know that it returns the results of the SQL statement as an XML document instead of as a standard rowset, as you're probably used to. You might also see it as FOR XML RAW or FOR XML EXPLICIT. Hang in there for now; we'll get to it. If you try leaving out the FOR XML statement with code similar to the following:

```
http://griffinj/Nwind/?sql=select+*+from+Employees&root=root
```

You'll see an error message in your browser that's very similar to what is shown in Listing 4.1.

Listing 4.1 **Error Generated by Missing FOR XML Statement**

```
<?xml version="1.0" encoding="utf-8" ?>
<root>
  <?MSSQLError HResult="0x80004005" Source="Microsoft XML Extensions to
SQL Server" Description="Streaming not supported over multiple column
result"?>
</root>
```

Specifying a Template Directly in a URL

Here's an example of specifying a template directly:

```
http://IISServer/Nwind?template=<ROOT+xmlns:sql="urn:schemas-
microsoft-com:xml-sql"><sql:query>SELECT+*+FROM+Employees+FOR+XML+AUTO
</sql:query></ROOT>
```

Now you see an example of a template file. It is in the form of an XML document and contains one or more SQL statements.

Templates allow the data to be returned as a well-formed XML document. As you'll see shortly, this isn't necessarily so when specifying a SQL statement directly in a URL. Also, some SQL query statements can become quite long. If they were in a template file, they would be easier to read than in a URL with all the additional markup needed (the plus [+] signs).

Declaring a Template File in a URL

Rather than writing a very long URL statement similar to the one in the preceding section, we could put the SQL query in a template file and refer to it in the URL like this:

```
http://IISServer/Nwind/TemplateVirtualName/template.xml
```

Remember that the `TemplateVirtualName` was specified with the Virtual Directory Management utility.

This also provides for better security by keeping the user away from the details of the database.

Specifying an XPath Query Against a Schema File in a URL

The following example shows how this would look:

```
http://IISServer/Nwind/SchemaVirtualName.schemafile.xml/
Employee[@EmployeeID=6]
```

Here the `SchemaVirtualName` was specified with the Virtual Directory Management Utility, and `Employee[@EmployeeID=6]` is the XPath query executed against schemafile.xml.

Specifying Database Objects Directly in a URL

Database objects such as tables and views can be specified in a URL, and then an XPath query can be issued against it to produce results as shown in the following example:

```
http://IISserver/Nwind/dbobjectVirtualName/XpathQuery
```

The XPath query is placed as the last entity in the URL, directly after the `VirtualDirectoryName`.

Entities

When we talked about XML documents in Chapter 1, we mentioned certain special characters that must be treated differently than other characters because they are interpreted differently depending on their location in documents. For more information, go to `http://www.landfield.com/rfcs/rfc1738.html`.

We must also be concerned now about special characters in URLs. Certain characters have a functionality all their own when used in URLs.

Entities in XML

The characters listed in Table 4.1 should not be used between tags in an XML document. These characters have special meaning to XML and will cause misinterpretation during parsing. The appropriate substitution entities that should be used in their place are provided in the table.

For a more thorough discussion of these characters and why they affect things the way they do, see RFC 2396. This RFC is freely available on the Internet at`http://www.landfield.com/rfcs/rfc2396.html`.

Table 4.1 **Entity Substitutions**

Character	Entity
& (ampersand)	Use `&amp;`
‘ (apostrophe)	Use `&apos;`
< (less than)	Use `&lt;`
> (greater than)	Use `&gt;`
“ (quote)	Use `&quot;`

Let's look at the sample template file in Listing 4.2. You'll see why these entities are necessary. Listing 4.3 shows the result.

Listing 4.2 **Entities in Template Files**

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
    <sql:query>
      SELECT CustomerID, OrderDate, Freight
      FROM   Orders
      WHERE  Freight &gt; 800            <!--&gt; substituted for '>' -->
      FOR XML AUTO
    </sql:query>
</ROOT>
```

Listing 4.3 **Results of Entity Substitution**

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
    <Orders CustomerID="QUEEN" OrderDate="1996-12-04T00:00:00"
     Freight="890.78" />
    <Orders CustomerID="QUICK" OrderDate="1997-05-19T00:00:00"
     Freight="1007.64" />
    <Orders CustomerID="QUICK" OrderDate="1997-10-03T00:00:00"
     Freight="810.05" />
    <Orders CustomerID="SAVEA" OrderDate="1998-04-17T00:00:00"
     Freight="830.75" />
</ROOT>
```

Entities in URLs

When executing a query in a URL, you must be careful when using the characters listed in Table 4.2. They are interpreted according to the description in the table. All these characters are required at one point or another in the interpretation of URLs. Again, RFC 2396 discusses these characters in greater detail.

Table 4.2 **Special Characters in URLs**

Character	Description	Hexadecimal Value
+	Indicates a space (spaces cannot be used in a URL).	%20
/	Separates directories and subdirectories.	%2F
?	Separates the URL from the parameters.	%3F
%	Specifies special characters.	%25
#	Indicates bookmark anchors.	%23
&	Separates parameters specified in the URL.	%26

Here's an example of a direct SQL query in a URL:

```
http://IISServer/Nwind?sql=SELECT+*+FROM+Employees+WHERE+LastName+LIKE+
'D%'FOR+XML+AUTO&root=root
```

Here we are trying to retrieve all information concerning employees whose last names start with `D`. Because the `%` character is one of the special characters for URLs, trying this query directly in a URL results in several errors. To fix the problem, you need to substitute the hexadecimal value of the special character in its place, like this:

```
http://IISServer/Nwind?sql=SELECT+*+FROM+Employees+WHERE+LastName+LIKE+
'D%25'+FOR+XML+AUTO&root=root
```

There's one more point to make here. There are instances when it might become necessary to use a combination of XML and URL special characters. Look at the following sample template that could be specified directly in a URL. See if you can spot the problem.

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql"><sql:query>SELECT+
CustomerID,OrderDate,Freight+FROM+Orders+WHERE+Freight+>+800+FOR+XML+AUTO
</sql:query></ROOT>
```

Hopefully, by now you know that the `>` character won't work here, so we'll change it to the required XML entity.

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql"><sql:query>SELECT+
CustomerID,OrderDate,Freight+FROM+Orders+WHERE+Freight+&gt;+800+FOR+XML+
AUTO</sql:query></ROOT>
```

Did you make it this far? If so, good; but if you stopped here, you didn't go quite far enough. The `&` character is a URL special character, and a substitution needs to be made here also. Replace the `&` with the hexadecimal value `%26`.

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql"><sql:query>SELECT+
CustomerID,OrderDate,Freight+FROM+Orders+WHERE+Freight+%26gt;+800+FOR+XML
+AUTO</sql:query></ROOT>
```

Executing SQL via HTTP

Up to this point, we've given a lot of examples to show how the HTTP protocol is used to request XML documents from SQL Server. We've never really given a formal definition of the syntax. Table 4.3 gives the formal definition. We'll also cover some more details of querying via a URL, such as stored procedures, templates in depth, utilizing XSLT stylesheets, and so on.

Here's the formal syntax for URL access accepted by the SQL ISAPI extension:

```
http://iisserver/virtualroot/virtualname[/pathinfo][/XPathExpression]
[?param=value[&param=value]...n]
```

or:

```
http://iisserver/virtualroot?{sql=SqlString | template=XMLTemplate}
[&param=value[&param=value]...n]
```

Table 4.3 **HTTP Syntax Explanation**

Keyword	Description	
`iisserver`	The IIS server to access.	
	For example: `www.newriders.com`.	
`virtualroot`	The virtual root configured with the Virtual Directory Management utility (graphically or programmatically).	
`virtualname`	A virtual name defined when the virtual root was configured. It will be one of three types: template, schema, or dbobject.	
`[/pathinfo]`	Path information to locate template or schema files in addition to the path information specified during `virtualname` configuration. Not needed for a dbobject type.	
`[/XPathExpression]`	Specified if necessary for schema files or dbobject types.	
`?sql`	Delimits an SQL query string.	
`SqlString`	An SQL query or stored procedure name. Usually contains the FOR XML extension unless the returned data is already in XML format. For example, a stored procedure that returns XML data.	
`?template`	Delimits a SQL query string formatted as an XML document.	
`param`	This is either a parameter name or one of the following:	
	`contenttype`	Specifies the content format of the returned document to allow a Web browser to pick the proper display method. Specified in two parts, the `contenttype` and `subtype`. It's sent in the HTTP header to become the MIME type of the document. text/XML, text/HTML, and image/gif designate an XML document, an HTML document, and a GIF image, respectively.
	`outputencoding`	The character set used to render the generated XML document. The default is UTF-8. Templates specified directly in a URL via `template=` are rendered in Unicode. Template files can specify the `outputencoding` themselves because they are XML documents.

continues

Table 4.3 **Continued**

Keyword	Description	
	`root`	When specified, the returned data is bracketed with the element name given to generate a well-formed XML document.
	`xsl`	The URL of an XSLT stylesheet used to process the returned data. By default, output documents are UTF-8 encoded unless overridden by an encoding instruction in the XSL file. If `outputencoding` is specified, it overrides the XSL specified encoding.

Character encodings are specified with the `encoding=` attribute in the XML declaration. The XML specification explicitly says XML uses ISO 10646, the international standard 31-bit character repertoire that covers most human languages. It is planned to be a superset of Unicode and can be found at `http://www.iso.ch`.

The spec says (2.2) "All XML processors must accept the UTF-8 and UTF-16 encodings of ISO 10646... ." UTF-8 is an encoding of Unicode into 8-bit characters: The first 128 are the same as ASCII; the rest are used to encode the rest of Unicode into sequences of between 2 and 6 bytes. UTF-8 in its single-octet form is therefore the same as ISO 646 IRV (ASCII), so you can continue to use ASCII for English or other unaccented languages using the Latin alphabet. Note that UTF-8 is incompatible with ISO 8859-1 (ISO Latin-1) after code point 126 decimal (the end of ASCII). UTF-16 is like UTF-8 but with a scheme to represent the next 16 planes of 64k characters as two 16-bit characters.

Regardless of the encoding used, any character in the ISO 10646 character set can be referred to by the decimal or hexadecimal equivalent of its bit string. So, no matter which character set you personally use, you can still refer to specific individual characters by using `&#dddd;` (decimal character code) or `&#xHHHH;` (hexadecimal character code in uppercase). The terminology can get confusing, as can the numbers: See the ISO 10646 Concept Dictionary at `http://www.cns-web.bu.edu/djohnson/web_files/i18n/ISO-10646.html`.

Well-Formed Documents, Fragments, and *&root*

As promised, the `&root=root` parameter used in some of the URLs and template files in the previous section requires a little explanation.

Think back to our discussion of well-formedness and XML documents. One of the conditions that must be met for an XML document to be well-formed is that it must contain a single top-level element. Remember our RESUMES shown in the following example:

```
<RESUMES xmlns='http://www.myorg.net/tags'>
  <PERSON PERSONID="p1">
    <NAME>
...
</RESUMES>
```

Our document contains the single top-level element `<RESUMES>`. Although there are other requirements for well-formed documents, if this one isn't met, the document fails the test.

How does this relate to the `&root` parameter in the URL? The `&root` parameter specifies the name of the document `ROOT` element. The result is an XML document with that all-important single top-level element. Let's look at some examples by reusing some of the code we have given previously:

```
http://IISServer/Nwind?sql=SELECT+*+FROM+Employees+FOR+XML+AUTO&root=root
```

Here we specify `&root=root`. This will generate a document that contains that single top-level element, and the element will be `<ROOT>`. In the following example, you would expect a document fragment to be returned because no root element is specified. You actually receive an error message stating, "Only one top-level element is allowed in an XML document." Because the root element is missing, all employee elements are assumed to be at the top level, which isn't allowed.

```
http://IISServer/Nwind?sql=SELECT+*+FROM+Employees+FOR+XML+AUTO
```

When template files are used, the same conditions hold true. Let's say we have the template file shown in Listing 4.4.

Listing 4.4 **Template File Without a Root Element**

```
<ROOT XMLNS:SQL="urn:schemas-microsoft-com:xml-sql">
  <sql:query>
    SELECT  *
    FROM    Employees
    FOR XML AUTO
  </sql:query>
</ROOT>
```

Let's say we employ this template file via this URL:

```
http://IISServer/Nwind/TemplateVirtualName/template.xml
```

The template file will provide the single top-level element via the `<ROOT>` element declaration.

Just to make sure we understand the root declaration, if I make the specification `root=EMPS` in the template file, the following fragment shows how the resulting document's root element has changed.

```
<EMPS xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <Employees EmployeeID="1" LastName="Davolio" ...
...
</EMPS>
```

Queries on Multiple Tables

Querying multiple tables has implications that you need to consider carefully if you want to generate your resulting documents with the proper element order. Here's the rule of thumb: "The order in which tables are specified in the SQL query determines the element nesting order." We'll take a look at the Orders and Employees tables in Northwind in a couple of ways (also refer to Appendix A, "Northwind Database Schema").

Here's the first query:

```
http://iisserver/Nwind?sql=SELECT+TOP+2+Orders.OrderID,+Employees.
LastName,+Orders.ShippedDate+FROM+Orders,+Employees+WHERE+Orders.
EmployeeID=Employees.EmployeeID+Order+by+Employees.EmployeeID,
OrderID+FOR+XML+AUTO&
root=ROOT
```

This returns the results in Listing 4.5.

Listing 4.5 **Results of Querying Multiple Tables**

```
<?xml version="1.0" encoding="utf-8" ?>
<ROOT>
  <Orders OrderID="10258" ShippedDate="1996-07-23T00:00:00">
    <Employees LastName="Davolio" />
  </Orders>
  <Orders OrderID="10270" ShippedDate="1996-08-02T00:00:00">
    <Employees LastName="Davolio" />
  </Orders>
</ROOT>
```

Let's do this again with three tables, this time adding the Order Details table.

```
http://iisserver/Nwind?sql=SELECT+TOP+2+Orders.OrderID,+Employees.
LastName,+Orders.ShippedDate,+[Order+Details].UnitPrice,+[Order+
Details].ProductID+FROM+Orders,+Employees,+[Order+Details]+WHERE+Orders.
EmployeeID=Employees.EmployeeID+AND+Orders.OrderID=[Order+Details].OrderID
+Order+by+Employees.EmployeeID,Orders.OrderID+FOR+XML+AUTO&root=ROOT
```

Listing 4.6 shows the results of this query.

Listing 4.6 **Results of Querying Three Tables**

```
<?xml version="1.0" encoding="utf-8" ?>
<ROOT>
  <Orders OrderID="10258" ShippedDate="1996-07-23T00:00:00">
    <Employees LastName="Davolio">
      <Order_x0020_Details UnitPrice="15.2" ProductID="2" />
      <Order_x0020_Details UnitPrice="17" ProductID="5" />
    </Employees>
  </Orders>
</ROOT>
```

Listing 4.7 shows the results obtained when we execute the same SQL expression but move the `Employees.LastName` element to the last element specified.

Listing 4.7 **Results of Moving *Employees.LastName* to the Last Element**

```
<?xml version="1.0" encoding="utf-8" ?>
<ROOT>
  <Orders OrderID="10258" ShippedDate="1996-07-23T00:00:00">
    <Order_x0020_Details UnitPrice="15.2" ProductID="2">
      <Employees LastName="Davolio" />
    </Order_x0020_Details>
    <Order_x0020_Details UnitPrice="17" ProductID="5">
      <Employees LastName="Davolio" />
    </Order_x0020_Details>
  </Orders>
</ROOT>
```

The key point I want you to grasp here is that the placement of elements in the result XML document depends on their placement in the SQL expression. This should be especially evident in the difference between Listings 4.6 and 4.7.

Passing Parameters

It is possible to pass parameters to SQL queries in URLs. This is known as *run-time* substitution as opposed to *design-time* substitution. In this case, we use a placeholder to specify the location where the parameter is to be substituted at execution time. The placeholder is a `?`, which must be specified as `%3F` in a URL. Here's an example:

```
http://iisserver/Nwind?sql=SELECT+TOP+4+OrderID+FROM+Orders+WHERE+
EmployeeID=%3F+FOR+XML+AUTO&EmployeeID=5&root=ROOT
```

Listing 4.8 is the resulting document.

Listing 4.8 **Results of Parameter Substitution in Our URL**

```
<?xml version="1.0" encoding="utf-8" ?>
<ROOT>
  <Orders OrderID="10248" />
  <Orders OrderID="10254" />
  <Orders OrderID="10269" />
  <Orders OrderID="10297" />
</ROOT>
```

Passing multiple parameters would just consist of more than one item with a question mark and a separate parameter for each. This URL passes two parameters:

```
http://iisserver/Nwind?sql=SELECT+TOP+4+OrderID+FROM+Orders+WHERE+
EmployeeID=%3F+AND+CustomerID=%3F+FOR+XML+AUTO&EmployeeID=5&CustomerID=
'VINET'&root=ROOT
```

This might not seem like a big deal here, and perhaps it's not to you, but wait until we start specifying template files in URLs that have parameters queries designed into them. We'll get to these shortly.

The XSL Keyword

Now we get to make use of what we learned in Chapter 2. Utilizing XSLT stylesheets gives us the much-needed flexibility to manipulate the XML output we generate. We can create HTML on-the-fly for immediate or later display, or we can change the returned XML document to a different one for further processing. The latter occurs more often than you would think, as with Electronic Data Interchange (EDI) related messages. Let's take the following SQL query and generate an HTML page:

```
http://iisserver/Nwind?sql=SELECT+TOP+4+OrderID,EmployeeID,Shipname+FROM+
Orders+WHERE+EmployeeID=5+FOR+XML+AUTO&xsl=order.xsl &root=ROOT
```

In this example, the XSL file is located in the virtual root directory. The resulting XML document is given in Listing 4.9, followed by the XSLT stylesheet in Listing 4.10.

Listing 4.9 **XML Document Containing Order Information**

```
<?xml version="1.0" encoding="utf-8" ?>
<ROOT>
  <Orders OrderID="10248" EmployeeID="5" Shipname="Vins et alcools
   Chevalier" />
  <Orders OrderID="10254" EmployeeID="5" Shipname="Chop-suey Chinese" />
  <Orders OrderID="10269" EmployeeID="5" Shipname="White Clover Markets"
/>
  <Orders OrderID="10297" EmployeeID="5" Shipname="Blondel père et fils"
/>
</ROOT>
```

Listing 4.10 **XSLT Stylesheet to Apply to Order Information**

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl='http://www.w3.org/XSL/Transform/1.0'>
<xsl:output media-type="text/html"/>

  <xsl:template match="/">
    <HTML>
    <BODY>
      <TABLE width='400' border='1'>
      <TR>
      <TD><B>Order ID</B></TD>
      <TD><B>Ship Name</B></TD>
      </TR>
      <xsl:apply-templates/>
      </TABLE>
    </BODY>
    </HTML>
  </xsl:template>

  <xsl:template match="Orders">
    <TR>
      <TD>
        <xsl:value-of select="@OrderID"/>
      </TD>
      <TD>
        <xsl:value-of select="@Shipname"/>
      </TD>
    </TR>
  </xsl:template>
</xsl:stylesheet>
```

This results in a simple table of Order ID versus Shipname, as shown in Table 4.4.

Table 4.4 **HTML Table of Results**

Order ID	**Shipname**
10248	Vins et alcools Chevalier
10254	Chop-suey Chinese
10269	White Clover Markets
10297	Blondel per`e et fils

Again, the result in this case might be simple, but the potential is enormous. Data can be retrieved from a database and presented in real-time to the viewer via a thin-client browser. Static pages can be generated and stored for viewing as needed by the client. In this case, the XSLT stylesheet functions as an HTML template. (Templates are much easier to maintain than rewriting HTML documents.) Business-to-business e-commerce documents can be generated from existing queries without having to modify the query itself. The XSLT stylesheet can manipulate the data in any way desired to create the required new XML document.

Next we change our focus to using template files to generate XML documents. We'll discuss them in a lot more detail than we have so far.

Executing Template Files via HTTP

If you look back at some of the queries we wrote in the previous section, you'll see that they can be pretty difficult to read sometimes. Take a look at the query that generated Listings 4.6 and 4.7 and tell me that you immediately know exactly what the query is doing. I doubt you can.

Template files have the same functionality as SQL queries written directly in URLs. Template files can do the following:

- Specify SQL queries or XPath queries
- Define parameters that can be passed to these queries
- Specify a top-level (root) element for the XML document
- Declare namespaces
- Specify an XSLT stylesheet to apply to the results

Template files have the added benefits of being easier to read and some say easier to write. In addition, they remove the database details from the general user for added security. Editing a file can be made impossible for the user, but because he can see a URL, he can change it or write his own and obtain information you might not want him to see or have. Also, there are fewer training requirements because the user only needs to know the filename and any parameters that might need to be passed.

Using XML Templates

Up to this point, when we've written a template file, we've used only the `<sql:query>` element to specify what the statement is to execute. In addition to this `<sql:query>` element, there are four other elements that can appear in a template file. Listing 4.11 shows the general format of a template file and is followed by an explanation of each of the elements in Table 4.5.

Listing 4.11 **XML Template Format**

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql"
      sql:xsl="XSL FileName" >
  <sql:header>
    <sql:param>..</sql:param>
    <sql:param>..</sql:param>...n
  </sql:header>
  <sql:query>
    sql statement(s)
  </sql:query>
  <sql:xpath-query mapping-schema="SchemaFileName.xml">
    XPath query
  </sql:xpath-query>
</ROOT>
```

Table 4.5 **XML Template Elements**

Element	Description
`<ROOT>`	This tag provides a single top-level element (also referred to as the *root tag*) for the resulting XML document. It can have any name.
`<sql:header>`	This tag is used to hold any header values. In the current implementation of SQL Server 2000, only the `<sql:param>` element can be specified in this tag. The `<sql:header>` tag acts as a containing tag, enabling you to define multiple parameters. This provides greater efficiency because all the parameter definitions are in one place. This is similar to declaring variables at the start of a T-SQL stored procedure.
`<sql:param>`	This element defines parameters that are passed to the queries inside the template. Each `<param>` element defines one parameter. Multiple `<param>` elements can be specified in the `<sql:header>` tag.
`<sql:query>`	This element specifies SQL queries. You can have multiple `<sql:query>` elements in a template. If there are multiple `<sql:query>` tags in the template and one fails, the others will proceed.
`<sql:xpath-query>`	This element specifies an XPath query. The schema filename must be specified using the `mapping-schema` attribute. If there are multiple `<sql:XPath-query>` tags in the template and one fails, the others will proceed.
`<sql:xsl>`	Specifies an XSLT stylesheet to be applied to the result document. A relative or absolute path can be given for the file. If a relative path is given, it is relative to the directory that was defined as the Template directory with the Virtual Directory Management utility.
`mapping-schema`	If you are executing an XPath query in a template, this attribute identifies the associated XDR schema. It can have a specified path identical to the path requirements of the `sql:xsl` element.

Here are some examples of using templates and template files in URLs. I'll reuse some of the earlier examples of URL SQL queries to illustrate the differences.

Here is a simple `SELECT` statement on a single table specified directly in a URL:

```
http://iisserver/Nwind?template=<ROOT+xmlns:sql="urn:schemas-microsoft-
com:xml-sql"><sql:query>SELECT+LastName,FirstName+FROM+Employees+
FOR+XML+AUTO</sql:query></ROOT>
```

Here is the result in Listing 4.12.

Listing 4.12 **Specifying a Template Directly in a URL**

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <Employees LastName="Davolio" FirstName="Nancy" />
  <Employees LastName="Fuller" FirstName="Andrew" />
  <Employees LastName="Leverling" FirstName="Janet" />
  <Employees LastName="Peacock" FirstName="Margaret" />
  <Employees LastName="Buchanan" FirstName="Steven" />
  <Employees LastName="Suyama" FirstName="Michael" />
  <Employees LastName="King" FirstName="Robert" />
  <Employees LastName="Callahan" FirstName="Laura" />
  <Employees LastName="Dodsworth" FirstName="Anne" />
</ROOT>
```

Taking the same template and making it a template file enables us to write it in a manner that is much easier to read (see Listing 4.13).

Listing 4.13 **SQL Query Rewritten into a Template Format**

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <sql:query>
    SELECT LastName, FirstName
    FROM Employees
    FOR XML AUTO
  </sql:query>
</ROOT>
```

Assuming that this template would be saved as the file template1.xml and saved to the directory with the virtual name templates, we would execute this template using the following URL:

```
http://iisserver/Nwind/templates/template1.xml
```

Let's look at one more example. When we queried a combination of three tables, we ended up with the following URL:

```
http://iisserver/Nwind?sql=SELECT+TOP+2+Orders.OrderID,+Employees.
LastName,+Orders.ShippedDate,+[Order+Details].UnitPrice,+[Order+
Details].ProductID+FROM+Orders,+Employees,+[Order+Details]+WHERE+Orders.
EmployeeID=Employees.EmployeeID+AND+Orders.OrderID=[Order+Details].
OrderID+Order+by+Employees.EmployeeID,Orders.OrderID+FOR+XML+AUTO&root=ROOT
```

Converting this to a template file gives us Listing 4.14.

Listing 4.14 **Long SQL Query Rewritten in a Template File**

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <sql:query>
    SELECT TOP 2
      Orders.OrderID,
      Employees.LastName,
      Orders.ShippedDate,
      [Order Details].UnitPrice,
      [Order Details].ProductID
    FROM
      Orders, Employees, [Order Details]
    WHERE
      Orders.EmployeeID=Employees.EmployeeID
    AND
      Orders.OrderID=[Order Details].OrderID
    ORDER BY
      Employees.EmployeeID,Orders.OrderID
    FOR XML AUTO
  </sql:query>
</ROOT>
```

This template file will produce the same results as those shown in Listing 4.6, but don't you think this is easier to read than the URL method?

Passing Template Parameters

Just as we passed parameters to SQL queries, we can also pass them to templates. The `<sql:header>` element is used to define the parameters, which also can be assigned default values. These default values are used for parameters at run-time if values are not explicitly specified.

Explicit Default Values and Parameter Passing

In this example, we want the CustomerID, OrderID, RequiredDate, and freight costs for a CustomerID we specify in the URL. Take a close look at the template file in Listing 4.15. We have our query stated in the `<sql:query>` element as we would expect. In addition, we have explicitly specified a default value of `VINET` for the CustomerID. The `<sql:param>` element accomplishes this. The `sql:header` element holds all parameters and their values.

The item in the query that is the parameterized quantity is specified by prepending the `@` symbol to the quantity name. In this case, it is CustomerID. Don't confuse this `@` symbol usage with the XML attribute usage of `@`. In this case, they are different entities altogether. This usage is specific to Microsoft parameterized expressions. If we execute this template file with the following URL, we will generate the result given in Listing 4.16.

```
http://iisserver/Nwind/templates/customer.xml
```

Listing 4.15 **Customer.xml**

```
<ROOT xmlns:sql='urn:schemas-microsoft-com:xml-sql'>
  <sql:header>
    <sql:param name='CustomerID'>VINET</sql:param>
  </sql:header>
  <sql:query>
    SELECT CustomerID,OrderID,RequiredDate,Freight
    FROM Orders
    WHERE CustomerID=@CustomerID
    FOR XML AUTO
  </sql:query>
</ROOT>
```

Listing 4.16 **Customer.xml Results with no CustomerID Passed**

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <Orders CustomerID="VINET" OrderID="10248" RequiredDate="1996-08-
   01T00:00:00" Freight="32.38" />
  <Orders CustomerID="VINET" OrderID="10274" RequiredDate="1996-09-
   03T00:00:00" Freight="6.01" />
  <Orders CustomerID="VINET" OrderID="10295" RequiredDate="1996-09-
   30T00:00:00" Freight="1.15" />
  <Orders CustomerID="VINET" OrderID="10737" RequiredDate="1997-12-
   09T00:00:00" Freight="7.79" />
  <Orders CustomerID="VINET" OrderID="10739" RequiredDate="1997-12-
   10T00:00:00" Freight="11.08" />
</ROOT>
```

Because no value for the parameter CustomerID was passed in the URL, the template file will use the default value `VINET`. If we pass a parameter value of `WELLI`, we obtain the results in Listing 4.17. Here's the URL:

```
http://iisserver/Nwind/templates/customer.xml?CustomerID=WELLI
```

Listing 4.17 shows the results.

Listing 4.17 **Partial Results with Parameter of *CustomerID=WELLI***

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <Orders CustomerID="WELLI" OrderID="10256" RequiredDate="1996-08-
   12T00:00:00" Freight="13.97" />
  <Orders CustomerID="WELLI" OrderID="10420" RequiredDate="1997-02-
   18T00:00:00" Freight="44.12" />
  <Orders CustomerID="WELLI" OrderID="10585" RequiredDate="1997-07-
   29T00:00:00" Freight="13.41" />
...
</ROOT>
```

Passing Multiple Parameters

You would think that multiple parameter passing would present no new problems, and you would be right. The parameters can just be individually listed in the `<sql:header>` element and be given default values. See Listing 4.18 and the result in Listing 4.19.

Listing 4.18 **Shipvia.xml—Multiple Parameters in a Template**

```
<ROOT xmlns:sql='urn:schemas-microsoft-com:xml-sql'>
  <sql:header>
    <sql:param name='ShipVia'>1</sql:param>
    <sql:param name='ShipCountry'>France</sql:param>
  </sql:header>
  <sql:query>
    SELECT TOP 4 CustomerID,OrderID,Freight
    FROM Orders
    WHERE ShipVia=@ShipVia
    AND ShipCountry=@ShipCountry
    ORDER BY OrderID
    FOR XML AUTO
  </sql:query>
</ROOT>
```

Listing 4.19 **Results of Listing 4.18**

```
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <Orders CustomerID="VICTE" OrderID="10251" Freight="41.34" />
  <Orders CustomerID="BLONP" OrderID="10265" Freight="55.28" />
  <Orders CustomerID="VINET" OrderID="10274" Freight="6.01" />
  <Orders CustomerID="BONAP" OrderID="10331" Freight="10.19" />
</ROOT>
```

Executing the URL `http://iisserver/Nwind/templates/shipvia.xml`, which calls the template in Listing 4.18, gives the results in Listing 4.19 because no parameters were passed and the default values of `1` for `ShipVia` and `France` for `ShipCountry` were used. You could also pass just one of the parameters. It would be substituted for the default value, and the other parameter would use the default value provided.

Specifying an XSL Stylesheet

There also is really nothing new when you want to apply an XSLT stylesheet to the results of a template file. Just specify the stylesheet name in the `sql:xsl` attribute of the `ROOT` element. Let's take the example from "The XSL Keyword" section earlier in this chapter that illustrated using an XSLT stylesheet.

Stylesheets are developed utilizing some third-party (relative to SQL server) application using a model of the document it is supposed to transform. If everything is tested during development, the only error conditions that usually happen are a blank HTML document, garbled HTML output, or an attempted transformation of an XML document that ends up blank. Because the XSLT stylesheet and query are properly tested in development, almost all these errors are traceable to a bad XML document or bad data.

Listing 4.20 gives the template file XSLDemo.xml.

Listing 4.20 **XSLDemo.xml**

```
<ROOT xmlns:sql='urn:schemas-microsoft-com:xml-sql'
sql:xsl='XSLDemo.xml'>
  <sql:query>
    SELECT TOP 4 OrderID,EmployeeID,Shipname
    FROM Orders
    WHERE EmployeeID=5
    FOR XML AUTO
  </sql:query>
</ROOT>
```

The stylesheet to apply is given in Listing 4.9, and the results are given in Listing 4.10. These result in Table 4.4.

Executing Stored Procedures

So far, everything we've done has been either SQL queries or templates in a URL or template files accessed via a URL. There has been one glaring omission here, however: *stored procedures* and how we execute them. In essence, we've been mimicking stored procedures through the use of template files. Now we'll use stored procedures as they should be used. (No doubt we'll be required to know how to do this in both URL queries and template files.)

Granting users the capability to write and execute stored procedures against a database is not the most secure way of doing business. Administrators should allow the user to read and execute stored procedures written by developers but not to write files to the TemplateVirtualDirectory. You would be leaving yourself open to all sorts of problems otherwise.

Listing 4.21 gives the stored procedure we'll use throughout this discussion.

Listing 4.21 **Example Stored Procedure**

```
IF EXISTS (SELECT name FROM sysobjects
   WHERE name = 'OrderInfo' AND type = 'P')
   DROP PROCEDURE OrderInfo
GO
```

```
CREATE PROCEDURE OrderInfo
AS
    SELECT OrderID, CustomerID
    FROM   Orders
    WHERE  CustomerID='CHOPS'
FOR XML AUTO
GO
```

This stored procedure can be executed using this URL:

```
http://IISServer/Nwind?sql=EXECUTE+OrderInfo&root=ROOT
```

Listing 4.22 gives the result file.

Listing 4.22 **Results of Calling the Example Stored Procedure**

```
<?xml version="1.0" encoding="utf-8" ?>
<ROOT>
  <Orders OrderID="10254" CustomerID="CHOPS" />
  <Orders OrderID="10370" CustomerID="CHOPS" />
  <Orders OrderID="10519" CustomerID="CHOPS" />
  <Orders OrderID="10731" CustomerID="CHOPS" />
  <Orders OrderID="10746" CustomerID="CHOPS" />
  <Orders OrderID="10966" CustomerID="CHOPS" />
  <Orders OrderID="11029" CustomerID="CHOPS" />
  <Orders OrderID="11041" CustomerID="CHOPS" />
</ROOT>
```

Passing parameters is accomplished by utilizing the `@` symbol again for the parameter expression in the stored procedure, as shown in Listing 4.23.

Listing 4.23 **Passing a Parameter to a Stored Procedure**

```
...
    SELECT OrderID,CustomerID
    FROM   Orders
    WHERE  CustomerID=@CustomerID
    FOR XML AUTO
...
```

The stored procedure can then be called via a URL in one of two ways. The first method is as follows:

```
http://iisserver/Nwind?sql=execute+OrderInfo+CHOPS
```

This method provides the value `CHOPS` by virtue of its position. If two parameters were being passed, you could just put them one right after the other, and they would be correctly passed.

The second method is as follows:

```
http://iisserver/Nwind?sql=execute+OrderInfo+@CustomerID=CHOPS
```

This method provides the value `CHOPS` by name, which is the method we are most used to.

Accessing Database Objects via HTTP

There is one entity left that we need to understand how to access. We've covered direct SQL queries and direct templates in URLs and the use of template files. Now we'll take a look at accessing dbobjects (tables, views, and so on).

Posting Templates via HTML Forms

As a final exercise for this chapter, we are going to show how to post a template file using an HTML form. In a Web environment, this would probably be the most common way of executing template files. Let's face it; users are comfortable and familiar with HTML forms. They see them just about every time they access the Internet and on their company intranet if there is one. Forms are easy to generate and provide a high level of interaction.

This template file has one parameter passed to the query. It would be a simple exercise to add multiple parameters. This template file would be saved as an HTML file and placed in a directory other than the virtual root tree. The SQLISAPI.DLL file is not expecting an HTML file and will not function correctly if placed in its directory tree.

Let's look at the HTML form itself, as shown in Listing 4.24.

Listing 4.24 **Posting a Template via an HTML Form**

```
<head>
<TITLE>Sample Form </TITLE>
</head>
<body>
For a given customer ID, order ID, order date and freight costs is
retrieved.
<form action="http://griffinjnt4s/nwind" method="POST">
<B>Customer ID</B>
<input type=text name=CustomerID value='VINET'>
<input type=hidden name=contenttype value=text/xml>
<input type=hidden name=template value='
<ROOT xmlns:sql="urn:schemas-microsoft-com:xml-sql" >
<sql:header>
    <sql:param name="CustomerID">VINET</sql:param>
</sql:header>

<sql:query>
  SELECT OrderID, OrderDate, Freight
  FROM    Orders
  WHERE    CustomerID=@CustomerID
  FOR XML AUTO
</sql:query>
</ROOT>
'>

<p><input type="submit">
</form>
</body>
```

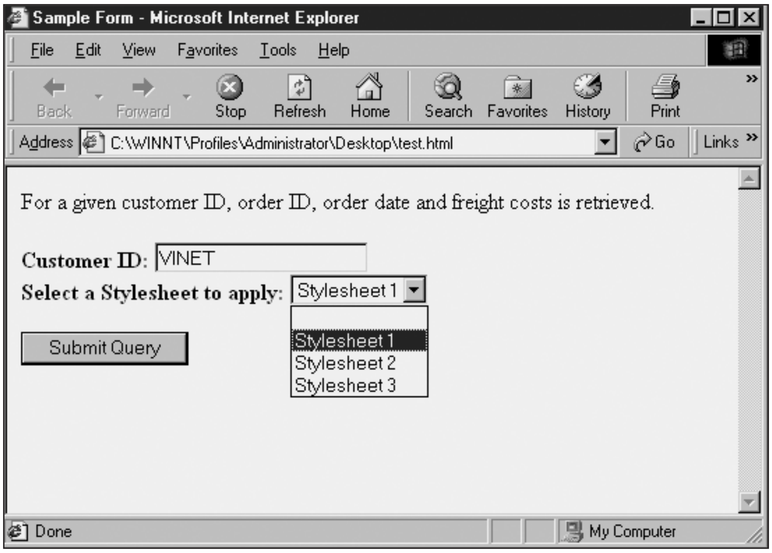
Here, the HTML text box supplies the value that is passed to the query. The query then returns the order ID, date ordered, and freight cost.

It is also possible to give the user a choice of what XSLT stylesheet to apply to the data. Add the following lines in Listing 4.25 to Listing 4.24.

Listing 4.25 **Additional HTML Code for the Drop-Down Box**

```
<br>
Select a Stylesheet to apply:
<select name='stylesheets' size='1'>
  <option value=''></option>
  <option value='ss1.xsl'>Stylesheet 1</option>
  <option value='ss2.xsl'>Stylesheet 2</option>
  <option value='ss3.xsl'>Stylesheet 3</option>
</select>
```

This adds a drop-down list of stylesheets for the user to choose from (see Figure 4.6).


Sample Form - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Print
Address C:\WINNT\Profiles\Administrator\Desktop\test.html
Go Links
Done
My Computer

For a given customer ID, order ID, order date and freight costs is retrieved.

Customer ID: VINET

Select a Stylesheet to apply: Stylesheet 1

Stylesheet 1
Stylesheet 2
Stylesheet 3

Submit Query

Figure 4.6 Adding an XSLT stylesheet selection drop-down box.

Also add an additional input field and parameter field like this:

```
<input type="text" name="stylesheet" value="ss1.xsl">

sql:param name="stylesheet">ss1</sql:param>
```

Finally, modify the `<root>` statement to this:

```
<root xmlns:sql="urn:schemas-microsoft-com:xml-sql"
sql:xsl="@stylesheet">
```

That finishes this chapter. One thing you should have noticed in this chapter is that the documents we generated are what are known as *attribute-centric documents*. All data is assigned to attributes of the elements of the document. The opposite of this is *element-centric documents.* All data is returned as a series of nested elements, which is probably the way you expected the data to be returned in the first place. SQL Server 2000 does provide ways to generate element-centric documents, and we will learn about them when we cover the FOR XML clause in detail in Chapter 8.

Recap of What Was Discussed in This Chapter

- Client/server is an open-ended architecture that provides for enhanced extensibility. It usually is implemented in two-tier, three-tier, or n-tier configurations and is based on the organization of data, application, and client logic.
- SQL Server 2000 provides several methods for generating XML documents directly from relational data. SQL queries can be placed directly in a URL, and they can be placed in template files. Templates themselves also can be placed directly in a URL.
- There are several special characters that, if placed in a URL, must be rewritten so they will be interpreted correctly.
- SQL queries in URLs provide much functionality. In addition to providing for document well-formedness by furnishing a `ROOT` element, they allow parameter passing and specifying included XSLT stylesheets.
- SQL template files provide the same functionality as direct SQL queries while at the same time making queries much more legible. They also provide a level of enhanced security by hiding the details of the database structure from the end user.
- The capability of executing SQL Server–stored procedures is very important to prevent possibly rewriting large amounts of code. This capability is provided for not only via SQL queries in URLs but also via template files.