Chapter 15 Working with Web Services

The next generation of Web applications involves the use of Web Services. Visual FoxPro 7's new built-in XML capabilities and Web Services extensions make it easy to create Web Services that can be accessed from anywhere on the Internet.

Before jumping into the topic of Web Services, it's best to first understand the basics of XML, how it has been integrated into VFP 7, and where you can use it in your applications.

Introducing XML

In a nutshell, XML is a (relatively) new text-based data format. So what's all the hubbub about? Primarily it's because XML has been universally accepted by all major platform vendors, paving the way for a universal cross-platform standard for sharing data.

XML bears a strong resemblance to HTML, and with good reason—they are both related to a parent language definition standard called Standard Generalized Markup Language (SGML). XML contains tags (words in angled brackets) surrounding blocks of text. However, unlike HTML where each tag has a specific meaning, XML is extensible—its tags have no predefined meaning. You can assign your own meaning to the tags.

For example, in HTML, the <TABLE> tag has a specific meaning. Based on this, any browser that understands the HTML specification can render tables correctly. In contrast, the <TABLE> tag can mean anything in XML. For example, in the following XML fragment, the <TABLE> tag refers to different kinds of furniture—a dining room, coffee, or card table:

```
<FURNITURE>
  <TABLE>"Dining Room"</TABLE>
  <TABLE>"Coffee"</TABLE>
  <TABLE>"Card"</TABLE>
</FURNITURE>
```

For a more detailed explanation of XML, see the MSDN topic "XML Tutorial."

The structure of an XML document

Here is an example of an XML document:

```
<?xml version = "1.0" encoding="Windows-1252" standalone="yes"?>
<VFPData>
  <customer>
    <iid>1</iid>
    <cacctno>001000</cacctno>
    <cname>Journey Communications</cname>
    <caddress1>101 Main St.</caddress1>
    <ccity>Richmond</ccity>
    <cstate region="1">VA</cstate>
    <czip>22901</czip>
    <ctype/>
  </customer>
  <customer>
    <iid>4</iid>
    <cacctno>001003</cacctno>
    <cname>Sergio Vargas, Attorney at Law</cname>
    <caddress1>115 Pacific Coast Highway</caddress1>
    <ccity>Malibu</ccity>
    <cstate region="5">CA</cstate>
    <czip>80766</czip>
    <ctype/>
  </customer>
</VFPData>
```

The first line in the document is the *XML declaration*:

```
<?xml version = "1.0" encoding="Windows-1252" standalone="yes"?>
```

This declaration is not required, but it's recommended that you place it at the top of all of your XML documents. It identifies the document as an XML document and specifies the version of XML used to create it. Optionally, it can contain the character encoding used. The "standalone" instruction specifies whether the XML document stands alone or needs additional definitions declared in other files. Note that the word "xml" must be in lowercase. XML tags (unlike HTML) are case-sensitive.

The next line in the document is the start tag of the root element:

```
<VFPData>
```

The XML specification requires that all XML documents contain a single root element that encompasses all other elements in the document.

Within the root element are two <customer> elements, each representing an individual customer record. Within the customer element there are nine other elements, each representing a field in a record.

Elements

Elements are the most common form of markup found in an XML document. They consist of an opening and closing tag, content between the tags (also referred to as "data"), and optionally attributes with associated values (see the next section). For example:

```
<cname>Sergio Vargas, Attorney at Law</cname>
```

If a particular element does not have any content, you can use a single tag element to represent it. For example, the ctype tag in the previous section contains no data and is represented as:

```
<ctype/>
```

This contains both the start and end tag information in a single tag element.

Attributes

An attribute is a name-value pair enclosed within the opening tag of an element. For example:

```
<cstate region="5">CA</cstate>
```

In this element, region="5" is the name-value pair that comprises the attribute. Attributes can be used to provide additional information regarding a particular element.

Comments

In an XML document, comments take the form:

```
<!-- This is a comment -->
```

Comments are ignored by XML parsers.

Well-formed XML

XML documents are *well-formed*, if they adhere to the basic rules of XML. For example, the XML specification dictates that XML documents must contain a unique root node. Another rule of well-formed XML is that start and end tags match—including their capitalization. For example, these are valid XML tags:

```
<cacctno>001003</cacctno>
```

But these are not, since the case doesn't match:

```
<cacctno>001003</cAcctNo>
```

With HTML, you can be somewhat lax about ending tags. Browsers rendering HTML can be very forgiving—if you leave off an ending tag, the browser can compensate and render the HTML in a presentable fashion. In contrast, XML parsers are not at all forgiving. For each start tag, there must be a corresponding end tag.

Reserved characters

There are special reserved characters in XML that cannot be used "as is" in the data portion of an element because they have an alternate meaning in the XML specification. For example, the left angle bracket < is used in XML to mark the beginning of a tag; so if you want to use this character in the data portion of an XML element, you need to replace it with: &lt;.

The reserved characters and their replacement character sequence (entity encoding) are shown in **Table 1**.

***Table 1**. This table shows commonly used special reserved characters that must be replaced by corresponding entity encoding.*

Reserved character	Entity encoding replacement
& (ampersand)	&amp;
< (less than)	&lt;
> (greater than)	&gt;
" (double quote)	&quot;
' (apostrophe)	&apos;

For example, in XML, the text "Liederbach & Associates" must be changed to:

```
<cname>Liederbach &amp; Associates</cname>
```

Element-centric and attribute-centric XML

Visual FoxPro 7 can produce XML data in three different formats:

- Element-centric
- Attribute-centric
- Raw

The following sections show how a Customer table with the structure shown in **Table 2** is represented by VFP 7 in each of these formats.

***Table 2**. This table shows the structure of a Customer table used in the XML examples for this chapter.*

Name	Type	Width
iID	Integer	4
cAcctNo	Character	6
cName	Character	50
cAddress1	Character	40
cCity	Character	25
cState	Character	2
cZip	Character	10

Element-centric XML

Records formatted in element-centric XML look like this:

```
<?xml version = "1.0" encoding="Windows-1252" standalone="yes"?>
<VFPData>
  <customer>
    <iid>1</iid>
    <cacctno>001000</cacctno>
    <cname>Journey Communications</cname>
    <caddress1>101 Main St.</caddress1>
    <ccity>Richmond</ccity>
    <cstate>VA</cstate>
    <czip>22901</czip>
  </customer>
  <customer>
    <iid>4</iid>
    <cacctno>001003</cacctno>
    <cname>Sergio Vargas, Attorney at Law</cname>
    <caddress1>115 Pacific Coast Hwy</caddress1>
    <ccity>Malibu</ccity>
    <cstate>CA</cstate>
    <czip>80766</czip>
  </customer>
</VFPData>
```

In this format, each record is represented as an individual element. Each field is also represented as an element, but nested within a record element.

Attribute-centric XML

Records formatted in attribute-centric XML look like this:

```
<?xml version = "1.0" encoding="Windows-1252" standalone="yes"?>
<VFPData>
  <customer iid="1" cacctno="001000" cname="Journey Communications"
    caddress1="101 Main St." ccity="Richmond" cstate="VA" czip="22901"/>
  <customer iid="4" cacctno="001003" cname="Sergio Vargas, Attorney at Law"
    caddress1="115 Pacific Coast Hwy" ccity="Malibu" cstate="CA" czip="80766"/>
</VFPData>
```

In this format, each record is represented as a single-tag element (each element is given the same name as the associated cursor). All field values are represented as an attribute of a record element. All field values are represented as strings regardless of their original type. As you can see, attribute-centric XML is more concise than element-centric.

Raw XML

Records formatted in raw XML look like this:

```
<?xml version = "1.0" encoding="Windows-1252" standalone="yes"?>
<VFPData>
  <row iid="1" cacctno="001000" cname="Journey Communications" caddress1="101
    Main St." ccity="Richmond" cstate="VA" czip="22901"/>
  <row iid="4" cacctno="001003" cname="Sergio Vargas, Attorney at Law"
    caddress1="115 Pacific Coast Hwy" ccity="Malibu" cstate="CA" czip="80766"/>
</VFPData>
```

This format is the same as attribute-centric, except each record element is named “row.”

Schemas and valid XML

As described in the section “Well-formed XML,” in order for an XML document to be interpreted properly by a parser, it must be well-formed. In addition to being well-formed, an XML document can also be *valid*—if it adheres to a specified *schema*.

A schema defines the structure and meaning of an XML document including element names and data types and attributes. You can use a schema to specify the format you expect for an XML document.

Schemas can be stored in a separate file, or included in an XML document. Here is an example of an in-line schema:

```
<?xml version = "1.0" encoding="Windows-1252" standalone="yes"?>
<VFPData>
   <xsd:schema id="VFPSchema" xmlns:xsd=http://www.w3.org/2000/10/XMLSchema
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
      <xsd:element name="customer" minOccurs="1" maxOccurs="unbounded">
         <xsd:complexType>
            <xsd:attribute name="iid" use="required" type="xsd:int"/>
            <xsd:attribute name="cacctno" use="required">
               <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                     <xsd:maxLength value="6"/>
                  </xsd:restriction>
               </xsd:simpleType>
            </xsd:attribute>
            <xsd:attribute name="cname" use="required">
               <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                     <xsd:maxLength value="50"/>
                  </xsd:restriction>
               </xsd:simpleType>
            </xsd:attribute>
            <xsd:attribute name="caddress1" use="required">
               <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                     <xsd:maxLength value="40"/>
                 </xsd:restriction>
               </xsd:simpleType>
            </xsd:attribute>
            <xsd:attribute name="ccity" use="required">
               <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                     <xsd:maxLength value="25"/>
                  </xsd:restriction>
               </xsd:simpleType>
            </xsd:attribute>
            <xsd:attribute name="cstate" use="required">
               <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                     <xsd:maxLength value="2"/>
                  </xsd:restriction>
               </xsd:simpleType>
            </xsd:attribute>
            <xsd:attribute name="czip" use="required">
               <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                     <xsd:maxLength value="10"/>
                  </xsd:restriction>
               </xsd:simpleType>
            </xsd:attribute>
         </xsd:complexType>
      </xsd:element>
      <xsd:element name="VFPData" msdata:lsDataSet="true">
         <xsd:complexType>
            <xsd:choice maxOccurs="unbounded">
               <xsd:element ref="customer"/>
            </xsd:choice>
         </xsd:complexType>
      </xsd:element>
   </xsd:schema>
   <customer iid="1" cacctno="001000" cname="Journey Communications"
    caddress1="101 Main St." ccity="Richmond" cstate="VA" czip="22901"/>
   <customer cacctno="001003" cname="Sergio Vargas, Attorney at Law"
    caddress1="115 Pacific Coast Hwy" ccity="Malibu" cstate="CA" czip="80766"/>
</VFPData>
```

If you look closely at the in-line schema, you can see it specifies the name of elements, whether they are required, their length, and their type. Visual FoxPro 7 supports the W3C XML Schema standard (XSD). For details, see http://www.w3.org/TR/2000/CR-xmlschema-0-20001024/.

For a table that shows you how Visual FoxPro data types map to XSD types, see the VFP 7 Help file topic “XML Functions.”

XML namespaces

Namespaces are a way to uniquely identify elements in an XML document. This is important because the author of an XML document defines its XML tags, and if you merge documents from multiple authors, you run the chance of a collision occurring between tag names.

You can define one or more namespaces for a document with the xmlns attribute in an opening XML element tag. The namespace applies to all the contents of an element. Since a namespace is used to uniquely identify tags, the namespace itself must be unique. Often, XML document authors choose the URL of their company as a namespace since it’s guaranteed to be unique. For example:

```
<VFPData xmlns="www.oakleafsd.com">
  <customer iid="1" cacctno="001000" cname="Journey Communications"
   caddress1="101 Main St." ccity="Richmond" cstate="VA" czip="22901"/>
  <customer iid="4" cacctno="001003" cname="Sergio Vargas, Attorney at Law"
   caddress1="115 Pacific Coast Hwy" ccity="Malibu" cstate="CA" czip="80766"/>
</VFPData>
```

XML functions in Visual FoxPro 7

Although you could write your own custom functions that converted a cursor to XML and vice versa, VFP 6 knew nothing about XML. Visual FoxPro 7 changes that with the introduction of three new functions: CursorToXML(), XMLToCursor(), and XMLUpdateGram().

CursorToXML()

The new CursorToXML() function does just what its name says—converts a Visual FoxPro cursor to XML. This command can be used by business objects to convert internal cursors to XML that can be consumed by clients.

The syntax of CursorToXML() is:

```
CursorToXML(nWorkArea | cTableAlias, cOutput [, nOutputFormat [, nFlags [,
nRecords [, cSchemaName [, cSchemaLocation [, cNameSpace ]]]]]])
```

The first parameter allows you to specify either the work area or the alias of the cursor to be converted. The cOutput parameter specifies the name of a memory variable or file to which the resulting XML is stored. By default, cOutput specifies a variable name. To specify a file name, you must set the corresponding bit in the nFlags parameter (see the VFP 7 Help topic "CURSORTOXML() Function" for details).

The nOutputFormat parameter specifies the output format of the XML (1=Element-centric, 2=Attribute-centric, and 3=Raw).

The cSchemaName, cSchemaLocation, and cNameSpace parameters allow you to specify information regarding schemas and namespaces. The nFlags parameter allows you to specify additional output options such as preserving white space, output encoding, and how to format empty elements.

The following command converts the contents of the Customer table to element-centric XML and stores the result in a memory variable named lcXML:

```
CursorToXML("Customer", "lcXML")
```

This next command converts the contents of the Customer table to attribute-centric XML, stores the results in a file named Results.XML, and generates an inline schema with a "www.microsoft.com" namespace:

```
CURSORTOXML("customer","Results.xml", 2, 512, 0,"1", "", "www.microsoft.com")
```

XMLToCursor()

The new XMLToCursor() function converts an XML string to a Visual FoxPro cursor. This function can be used by business objects to convert XML input received by clients into an internal cursor format.

The syntax of XMLToCursor() is:

```
XMLToCursor(cXMLExpression | cXMLFile [, cCursorName [, nFlags]])
```

The cXMLExpression parameter specifies either XML text or an expression that evaluates to XML data. This can be any one of the following:

- A memory variable.
- A memo field.
- The return from an HTTP request.
- The return from a SOAP method call (see the section "Simple Object Access Protocol (SOAP)" later in this chapter).

- XML from the XML DOM (Document Object Model). The XML DOM is a programming interface for XML documents that allows you to access an XML document as a tree view with nodes.
- An ADO stream.

Alternately, you can specify the name of an XML file in the cXMLFile parameter. The nFlags parameter specifies how to handle the first parameter. For details, see the table in the VFP 7 Help topic "XMLToCursor()."

The cCursorName parameter specifies the name of the cursor to store the results. If this cursor is in use, VFP generates an error. XMLToCursor() can convert XML with or without a schema.

The following command converts the Results.xml file to a cursor named "MyCursor":

```
XMLTOCURSOR("Results.xml", "MyCursor", 512)
```

XMLUpdateGram()

The new XMLUpdateGram() function returns an XML string listing all the changes in a buffered table or cursor. You must enable table buffering and SET MULTILOCKS ON before running XMLUpdateGram().

The syntax of XMLUpdateGram() is:

```
XMLUpdateGram([cAliasList [, nFlags]])
```

The cAliasList parameter specifies a comma-delimited list of open tables or cursors to be included in the updategram. If cAliasList is empty, XMLUpdateGram() includes all open tables and cursors in the current data session.

The nFlags parameter contains a list of additive flags that specify how the XML output is formatted (if at all). See the VFP 7 Help file topic "XMLUPDATEGRAM() Function" for a table describing these flags.

Updating records

To demonstrate how XMLUpdateGram() works when updating records, launch Visual FoxPro, go to this chapter's ProjectX\Data directory, and enter the following in the Command Window:

```
USE Customer
SET MULTILOCKS ON
CURSORSETPROP("Buffering", 5, "Customer")   && Enable table buffering
REPLACE caddress1 with "500 Water St." IN Customer   && Make a change
lcUpdateXML = XMLUpdateGram("Customer")   && Generate the Update Gram
STRTOFILE(lcUpdateXML, "UpdateGram.xml")
MODIFY FILE UpdateGram.xml
```

The Developer Download files at www.hentzenwerke.com include the Customer table referenced in this chapter.

The UpdateGram.xml file should contain the following:

```
<?xml version = "1.0" encoding="Windows-1252" standalone="yes"?>
<root xmlns:updg="urn:schemas-microsoft-com:xml-updategram">
   <updg:sync>
      <updg:before>
         <customer>
            <iid>1</iid>
            <cacctno>001000</cacctno>
            <cname>Journey Communications</cname>
            <caddress1>101 Main St.</caddress1>
            <ccity>Richmond</ccity>
            <cstate>VA</cstate>
            <czip>22901</czip>
         </customer>
      </updg:before>
      <updg:after>
         <customer>
            <iid>1</iid>
            <cacctno>001000</cacctno>
            <cname>Journey Communications</cname>
            <caddress1>500 Water St.</caddress1>
            <ccity>Richmond</ccity>
            <cstate>VA</cstate>
            <czip>22901</czip>
         </customer>
      </updg:after>
   </updg:sync>
</root>
```

The updategram has a single "root" node containing a single <updg:sync> element. Nested within this element are the <updg:before> and <updg:after> elements. The <updg:before> element contains a list of the record's original values for *all* fields. The <updg:after> element contains a list of all current values. Notice the <caddress1> element contains the new "500 Water St." value.

If you use CURSORSETPROP() to set the key field list for the table before running XMLUpdateGram(), the resulting updategram contains only the key and changed fields. You can try this by entering the following in the Command Window:

```
TABLEREVERT(.T., "Customer")  && Revert the changes to the table
CURSORSETPROP("KeyFieldList", "iid", "Customer")   && Set the key field list
REPLACE caddress1 with "500 Water St." IN Customer   && Make a change
lcUpdateXML = XMLUpdateGram("Customer")   && Generate the Update Gram
STRTOFILE(lcUpdateXML, "UpdateGram1.xml")
MODIFY FILE UpdateGram1.xml
```

The UpdateGram1.xml file should contain the following:

```
<?xml version = "1.0" encoding="Windows-1252" standalone="yes"?>
<root xmlns:updg="urn:schemas-microsoft-com:xml-updategram">
```

```
   <updg:sync>
      <updg:before>
         <customer>
            <iid>1</iid>
            <caddress1>101 Main St.</caddress1>
         </customer>
      </updg:before>
      <updg:after>
         <customer>
            <iid>1</iid>
            <caddress1>500 Water St.</caddress1>
         </customer>
      </updg:after>
   </updg:sync>
</root>
```

Notice the updategram contains only information regarding two fields—the specified key field (iid) and the changed field (caddress1).

Adding records

To demonstrate how XMLUpdateGram() works when adding records, enter the following in the Command Window:

```
TABLEREVERT(.T., "Customer")  && Revert the changes to the table
INSERT INTO Customer (cacctno, cname, caddress1, ccity, cstate, czip);
  VALUES ("001004", "The Fox", "952 Market St.", "Reston", "VA", "22903")
lcUpdateXML = XMLUpdateGram("Customer")   && Generate the Update Gram
STRTOFILE(lcUpdateXML, "UpdateGram2.xml")
MODIFY FILE UpdateGram2.xml
```

The UpdateGram2.xml file should contain the following:

```
<?xml version = "1.0" encoding="Windows-1252" standalone="yes"?>
<root xmlns:updg="urn:schemas-microsoft-com:xml-updategram">
   <updg:sync>
      <updg:before/>
      <updg:after>
         <customer>
            <iid>6</iid>
            <cacctno>001004</cacctno>
            <cname>The Fox</cname>
            <caddress1>952 Market St.</caddress1>
            <ccity>Reston</ccity>
            <cstate>VA</cstate>
            <czip>22903</czip>
         </customer>
      </updg:after>
   </updg:sync>
</root>
```

Notice the <updg:before/> element is an empty single-tag element—since this is a new record, there is no “before” data. The <updg:after> element contains all the values for the new record.

Deleting records

To demonstrate how XMLUpdateGram() works when deleting records, enter the following in the Command Window:

```
TABLEREVERT(.T., "Customer")  && Revert the changes to the table
LOCATE   && Move the record pointer to the first record
DELETE   && Delete the first record
lcUpdateXML = XMLUpdateGram("Customer")   && Generate the Update Gram
STRTOFILE(lcUpdateXML, "UpdateGram3.xml")
MODIFY FILE UpdateGram3.xml
```

The UpdateGram3.xml file should contain the following:

```
<?xml version = "1.0" encoding="Windows-1252" standalone="yes"?>
<root xmlns:updg="urn:schemas-microsoft-com:xml-updategram">
   <updg:sync>
      <updg:before>
         <customer>
            <iid>1</iid>
            <cacctno>001000</cacctno>
            <cname>Journey Communications</cname>
            <caddress1>101 Main St.</caddress1>
            <ccity>Richmond</ccity>
            <cstate>VA</cstate>
            <czip>22901</czip>
         </customer>
      </updg:before>
      <updg:after/>
   </updg:sync>
</root>
```

Notice the <updg:before> element contains the original values of the deleted record and <updg:after/> is an empty single-tag element.

Multi-user contention checking

Why does the XMLUpdateGram() function include original values when generating an updategram? This allows you to verify that no other user has changed a record that is being updated or deleted.

Using updategrams in SQL Server 2000

Where can you use updategrams? First of all, updategrams are a new feature of SQL Server 2000 that allows SQL Server database updates to be defined as XML. Notice in the root node of the generated updategrams, the namespace is specified as:

```
<root xmlns:updg="urn:schemas-microsoft-com:xml-updategram">
```

The format used for VFP 7 updategrams is the standard format used by SQL Server 2000.

Using updategrams in distributed applications

Even if you're not using SQL Server 2000 as your back-end database, updategrams can be very useful in n-tier architectures. The Windows Distributed Internet Architecture (Windows DNA) specifies that business objects should be composed of two pieces—an emissary and an executant. In a distributed desktop system, the emissary portion of the business object resides on the workstation, and the executant portion resides on an application server elsewhere on the network.

When the workstation needs data, it sends a request to the emissary portion of the business object. Rather than retrieving the data directly, the business object passes the request to an executant object. The executant retrieves the data from the back end and passes it to the emissary. What transport mechanism can the executant use to send the data to the emissary? The most common choices are ADO and XML, but given VFP 7's new XML capabilities coupled with VFP 7's lightning fast string-manipulation abilities, there are compelling reasons to favor XML.

When the business object on the workstation receives the requested data as an XML string, it can convert it to a VFP cursor to be consumed by the desktop application. The application can manipulate the cursor—editing, adding, and deleting records. When the user saves changes, the emissary can run XMLUpdateGram() against the VFP cursor and pass the resulting updategram to the executant, which then applies the changes to the back-end database.

VFPXMLProgID

The new _VFP.VFPXMLProgID property allows you to specify a COM component that can be used instead of the built-in VFP 7 XML functions (CursorToXML(), XMLToCursor(), XMLUpdateGram()).

This property specifies the programmatic identifier (ProgID) of the surrogate COM component. The COM component you specify must implement VFP 7's IVFPXML interface. The VFP 7 Help topic "VFPXMLProgID Property" gives an example creating a COM object that implements this interface.

Introducing Web Services

Web Services are the next generation of Web applications. They are modular, self-describing applications that can be published and invoked across the Web. Web Services can range from simple functions to complex business processes.

A simple way to understand what Web Services are is to look at the way current generation Web applications work. At present, if you book a hotel room online, you usually need to go to a different Web site to determine what the weather will be during your stay (as long as the trip is not too far into the future). Wouldn't it be better if the hotel's Web site could provide this information for you? This requires the Web site to be able to communicate with a weather Web site—to pass it information regarding the weather based on the location of the hotel and your arrival date.

In actuality, this type of functionality has been available for some time. There are hotel Web sites that display the current weather; however, they're typically not using Web Services to accomplish this—they're using more of a "hard-coded" model. Usually, a weather service Web site posts a GIF file on their Web server, and hotel (or other) Web sites simply contain a pointer to the location of the GIF file. However, this approach is inflexible because it forces the weather information to be shown in a particular display format (size, shape, color, and so on). Using Web Services to retrieve the current weather allows consumers of this service to display the weather information in any format they choose.

Real, live Web Services!

How can you know what kind of Web Services are available on the Web? There are a number of Web Service resources that you can search. One of these is the Web site http://www.xmethods.net/, which provides a list of available Web Services (see **Table 3**).

Just looking through this list can help you visualize how Web Services can revolutionize Web applications. Web Services does for Web applications what ActiveX controls did for Win32 applications. Rather than reinventing the wheel, you can use services of existing Web sites to enhance your Web applications.

***Table 3**. Some of the more interesting Web Services available on the Internet.*

Name	Description	Schema location
FedEx Tracker	Access to FedEx tracking information	http://www.xmethods.net/sd/FedExTrackerService.wsdl
Babel Fish Service	Babel Fish Language Translation	http://www.xmethods.net/sd/BabelFishService.wsdl
Headline News	Latest news coverage from various sources	http://www.SoapClient.com/xml/SQLDataSoap.wsdl
Who Is	A SOAP version of the standard "who is" service that returns info on a specified Internet domain	http://soap.4s4c.com/whois/soap.asp?WSDL
Soap Web Search	SOAP interface to major search engines	http://www.soapclient.com/xml/SQLDataSoap.WSDL
SMS Messaging	Send a text message to a mobile phone	http://sal006.salnetwork.com:83/lucin/smsmessaging/process.xml
Currency Converter	Converts values from one international currency to another	http://sal006.salnetwork.com:83/lucin/currencyconvertor/ccurrencyc.xml

In addition to using the Web Services provided by other sites, you can also publish your own VFP 7 COM servers as Web Services that others can access.

Simple Object Access Protocol (SOAP)

Thinking about the example of hotel Web site to weather Web site communication raises some important questions—how does the hotel site know the capabilities of the weather Web site? How does it know the format of the request to be sent and the response it receives?

The answer to these questions is SOAP—Simple Object Access Protocol. SOAP leverages the existing HTTP and XML standards to provide a standard way to pass commands and parameters between clients and servers. Clients send requests to a server in XML format and the server sends the results back as XML. These messages are encapsulated in HTTP protocol messages.

For more information on SOAP-related topics, check Microsoft's SOAP Developer Resources Web site (http://msdn.microsoft.com/soap/).

Using the SOAP Toolkit 2.0

To make it easier to publish Web Services and create applications that invoke Web Services using SOAP, Microsoft has created the SOAP Toolkit (version 2.0 at the time of this writing).

The SOAP Toolkit 2.0 comes with an excellent Help file (Soap.CHM) located in the MSSOAP/Documentation folder. Rather than duplicate that information, this chapter touches on the highlights of the Toolkit. Refer to the Help file and samples for details.

Installing the SOAP Toolkit

You can download the SOAP Toolkit from Microsoft's SOAP Developer Resources Web site (http://msdn.microsoft.com/soap/). The download consists of two files: the SOAP Toolkit 2.0 Gold Release (SoapToolkit20.EXE) and Gold Release samples (SoapToolkit20Samples.EXE).

After downloading the Toolkit, double-click the EXE files and follow the instructions in the Install Wizard. By default, the SOAP Toolkit files are installed in the Program Files\MSSOAP directory.

The Toolkit contains the following components and tools:

- A component that makes it easier for clients to call a Web Service.
- A component that resides on the Web server and maps Web Service operations to COM object method calls.
- Additional components that construct, transmit, read, and process SOAP messages.
- A tool that generates Web Services Description Language (WSDL) documents and Web Services Meta Language (WSML) files (see the corresponding sections that follow).
- The SOAP Messaging Object (SMO) Framework, which provides an alternative to the XML DOM for processing XML in SOAP messages.

Web Services Description Language (WSDL)

The Web Services Description Language (WSDL) file can be likened to an XML type library for Web Services. This file identifies the services and sets of operations provided by a server. The WSDL file is placed on a Web server. Any client that wants to invoke the services of that server must first download a copy of the WSDL file to determine how to format a SOAP request for the provider.

Document-oriented and RPC-oriented operations

There are two main types of operations defined in a WSDL file—*document-oriented operations* and *RPC-oriented operations* (Remote Procedure Call). With document-oriented operations, request and response messages contain XML documents. The SOAP Messaging Object (SMO) Framework makes it easy to process the XML documents in a SOAP message.

RPC-oriented operations expect input parameters for input messages, and output messages contain return values.

Web Services Meta Language (WSML)

A Web Services Meta Language (WSML) file contains information that maps the operations of a service described in the WSDL file to a method in a COM object.

SOAP Listeners

A Listener is the entity that handles incoming SOAP messages on the server side. There are two types of Listeners you can choose from:

- An Internet Server API (ISAPI) Listener
- An Active Server Pages (ASP) Listener

From a sheer performance perspective, an ISAPI Listener is the best choice. Some developers have reported speeds of up to four times faster for an ISAPI Listener. However, if you want to do some processing between the request received from the client and the call to a method on a COM object, you may want to implement an ASP Listener instead. You can place code in an ASP page surrounding the call to the object. See the SOAP Toolkit 2.0 Help topics "Specifying an ASP Listener" and "Specifying an ISAPI Listener" for details.

Visual FoxPro 7.0 Web Services extensions

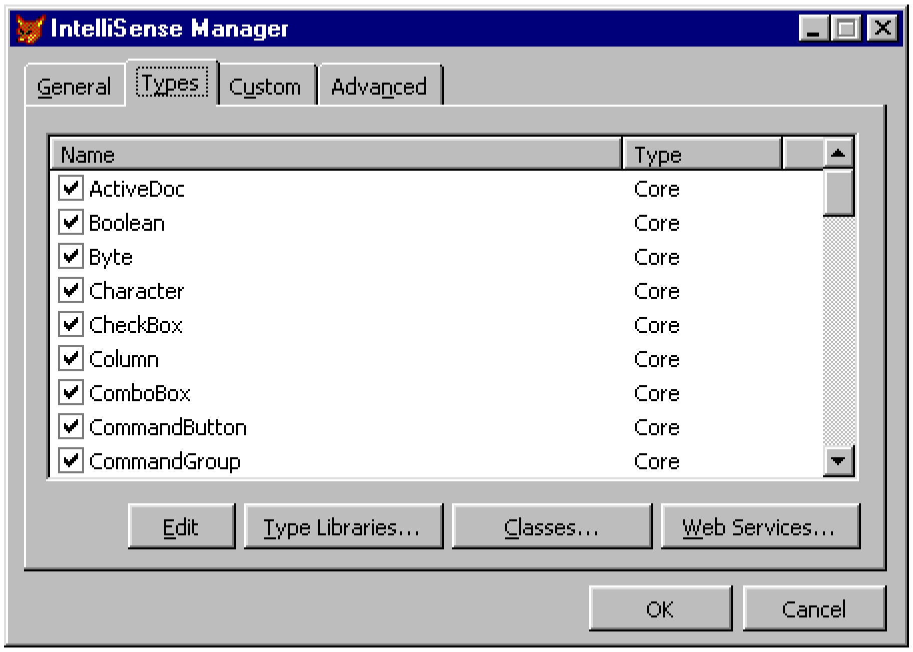
Although you can use the Microsoft SOAP Toolkit 2.0 directly, Visual FoxPro 7 provides a set of extensions to the Toolkit that make it easy to register a Web Service for use by a client (if you need finer control over publishing and consuming Web Services, you should use the SOAP Toolkit directly). The extensions also make it easy to publish Web Services, and can produce WSDL and WSML files for your COM servers.

For details on using the SOAP Toolkit with Visual FoxPro, check out Rick Strahl's white paper "Using Microsoft SOAP Toolkit for remote object access" at http://www.west-wind.com/presentations/soap/.

Registering a Web Service

To register a Web Service for use by a client:

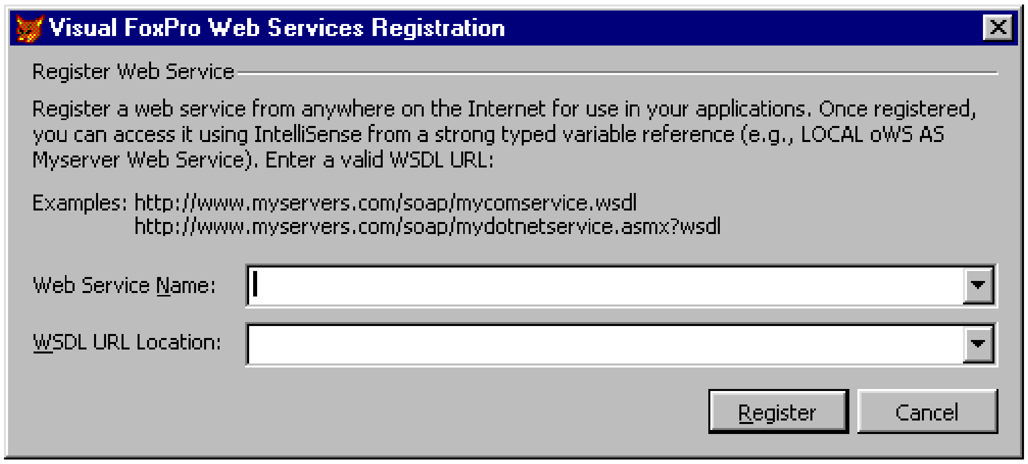
1. From the Tools menu, select IntelliSense Manager.
2. Select the Types tab and click the Web Services button (see **Figure 1**).



Name	Type
✓ ActiveDoc	Core
✓ Boolean	Core
✓ Byte	Core
✓ Character	Core
✓ CheckBox	Core
✓ Column	Core
✓ ComboBox	Core
✓ CommandButton	Core
✓ CommandGroup	Core

***Figure 1**. The IntelliSense Manager's* Web Services *button launches the Web Services Registration dialog.*

3. In the Web Services Registration dialog's Web Service Name box, specify a friendly name for the Web Service (see **Figure 2**). The name can include spaces (for example, "Language Translation"). A history of all Web Service names you've entered is stored and displayed in the Web Service Name combo box.



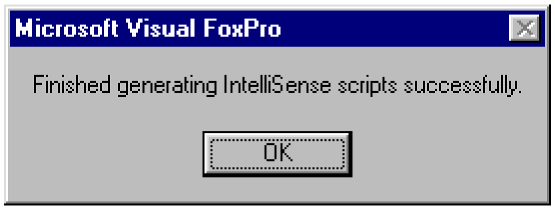
Register a web service from anywhere on the Internet for use in your applications. Once registered, you can access it using IntelliSense from a strong typed variable reference (e.g., LOCAL oWS AS Myserver Web Service). Enter a valid WSDL URL:

Examples: http://www.myservers.com/soap/mycomservice.wsdl
http://www.myservers.com/soap/mydotnetservice.asmx?wsdl

***Figure 2**. The Web Services Registration dialog allows you to register a Web Service located anywhere on the Internet for use in your applications.*

4. In the WSDL URL Location box, enter the WSDL URL of the Web Service you are registering. For example: http://www.xmethods.net/sd/BabelFishService.wsdl.
5. Click the Register button.

At this point, the dialog generates IntelliSense scripts based on the information you have entered. Note that this may take several seconds with no visual cue that any processing is going on! When the process is complete, you will see the dialog shown in **Figure 3**.


Microsoft Visual FoxPro

Finished generating IntelliSense scripts successfully.

***Figure 3**. This dialog displays when your Web Service registration has successfully completed. Note that it may take quite a while for this dialog to appear.*

Click the OK button to close this dialog. Your new Web Service is added to the list in the Types tab of the IntelliSense Manager. The next section discusses how you can access the new Web Service.

It is not required that you register a Web Service using the IntelliSense Manager. However, if you do, VFP provides IntelliSense for Web Service methods and parameters.

Calling Web Services

To programmatically call a Web Service that you have set up, do the following:

1. In code, declare a strong-typed variable. For example:

```
LOCAL loTranslate AS
```

2. In the dropdown list of IntelliSense types, select a Web Service. For example:

```
LOCAL loTranslate AS Language Translation
```

3. When you press the Enter key, Web Service proxy code is automatically added to your source file. For example:

```
LOCAL loTranslate AS Language Translation
LOCAL loWS
loWS = NEWOBJECT("Wsclient",HOME()+"ffc\_webservices.vcx")
```

```
loTranslate =
loWS.SetupClient("http://www.xmethods.net/sd/BabelFishService.wsdl",
"BabelFish", "BabelFishPort")
```

This code instantiates the VFP 7 WSClient base class (loWS) and then creates a SOAP client object (loTranslate).

4. To use this service, add the following code:

```
?loWS.BabelFish("en_es","fox")
```

This code tells the Babel Fish Web Service to translate the word "fox" from English to Spanish ("en_es") and display it on the VFP desktop.

Publishing Web Services

In addition to calling Web Services, the VFP 7 Web Services extensions allow you to publish VFP COM servers as Web Services.

In order to publish Web Services, you must have Internet Information Server installed.

To publish a Web Service:

1. Create a COM server project, containing at least one OLE Public class, and then build an in-process VFP COM server DLL (usually a multi-threaded COM server DLL) from the project.

2. Right-click your VFP COM server project and select Builder from the context menu (see **Figure 4**). This launches the Wizard Selection dialog. In the Wizard Selection dialog, select Web Services Publisher and click OK.

3. If this is the first time you have published a Web Service, the dialog in **Figure 5** is displayed recommending that you select a default URL for your Web Service support files such as ASP Listeners and WSDL and WSML files. Specifying a default directory is not mandatory, but if you don't specify one, you are prompted for a location each time you publish a Web Service. Click OK to launch the Web Service Location dialog.

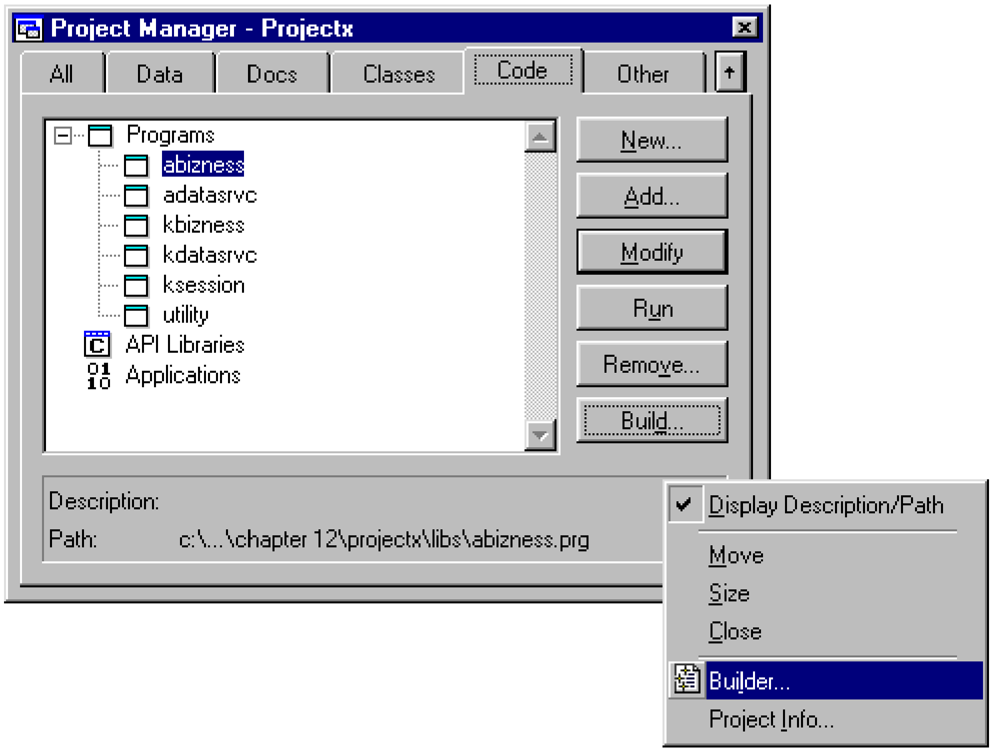
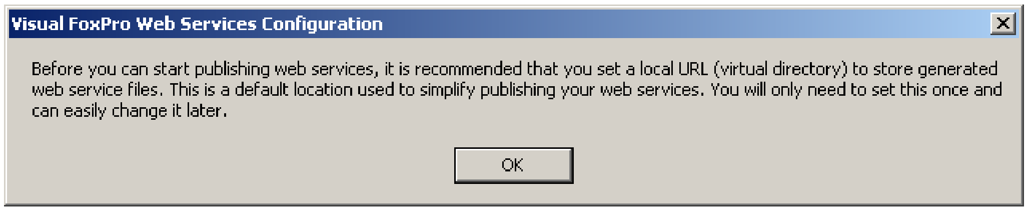


***Figure 4**. You can launch the Wizard Selection dialog from the Project Manager context menu. You can also launch it by selecting Tools | Wizards | All Wizards from the VFP 7 menu, and selecting Web Services Publisher in the Wizard Selection dialog.*


Visual FoxPro Web Services Configuration

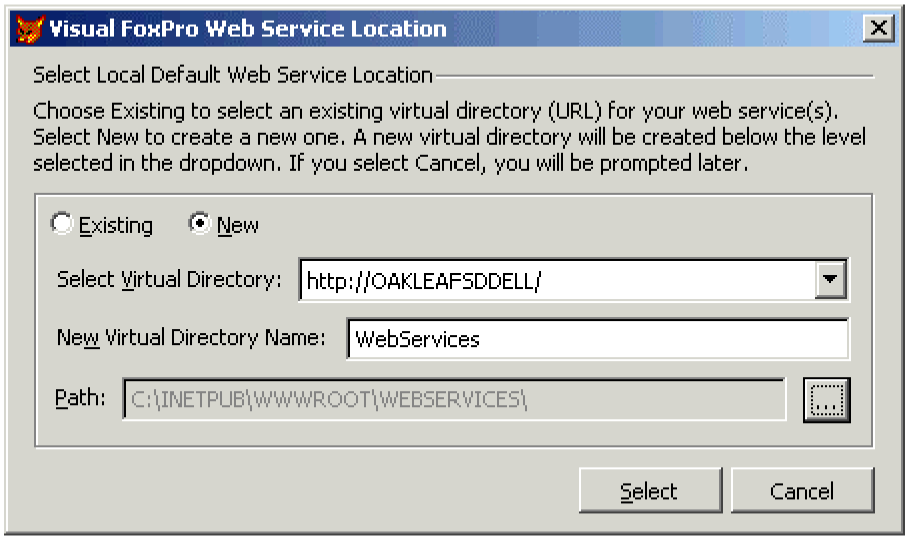
Before you can start publishing web services, it is recommended that you set a local URL (virtual directory) to store generated web service files. This is a default location used to simplify publishing your web services. You will only need to set this once and can easily change it later.

***Figure 5**. The first time you publish a Web Service, the Web Services Configuration dialog displays recommending that you select a default URL for your Web Service files.*

4. In the Web Service Location dialog, select an existing virtual directory for your Web Services support files or create a new one (see **Figure 6**).

   You can select an existing virtual directory by clicking the Existing option and selecting a directory from the Select Virtual Directory combo box.

You can select a new directory by clicking the New option and entering a directory name in the New Virtual Directory Name box. Next, click the Path command button and specify the physical path for the new virtual directory. Typically, you should create a new directory under your c:\inetpub\wwwroot directory.


Visual FoxPro Web Service Location

Select Local Default Web Service Location

Choose Existing to select an existing virtual directory (URL) for your web service(s). Select New to create a new one. A new virtual directory will be created below the level selected in the dropdown. If you select Cancel, you will be prompted later.

Existing [✓ New]

Select Virtual Directory: http://OAKLEAFSDDELL/

New Virtual Directory Name: WebServices

Path: C:\INETPUB\WWWROOT\WEBSERVICES\

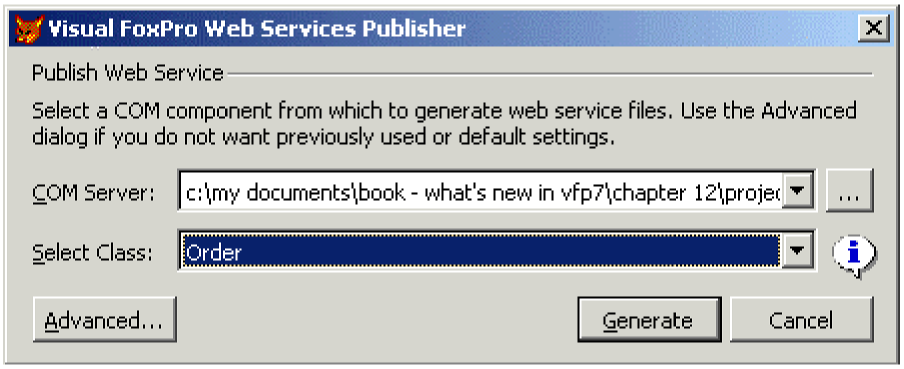
Figure 6. *The Web Service Location dialog allows you to specify a default virtual directory for your Web Services support files.*

The information entered in this dialog gets saved to a table named FoxWS.DBF located in your Visual FoxPro 7 program directory.

5. Click the Select button to continue. This launches the Web Services Publisher dialog (see **Figure** 7). By default, the COM Server combo box contains a list of all COM servers associated with open projects as well as previously published COM servers. Select a COM server from the combo box or click the ellipsis button to select a COM server that is not in the list.

6. The Select Class combo box displays a list of OLE public classes in the selected COM server. Select the class you want to publish. If the COM server you choose has only one class, the combo box is disabled.

   If the class you choose contains Currency or Variant parameter or return value types, an “information” image is displayed next to the Select Class combo box (see Figure 7). If you click on this image, a dialog displays telling you to specify strong typing for these; otherwise, an invalid WSDL file may be generated.


Visual FoxPro Web Services Publisher

Publish Web Service

Select a COM component from which to generate web service files. Use the Advanced dialog if you do not want previously used or default settings.

COM Server: c:\my documents\book - what's new in vfp7\chapter 12\projec

Select Class: Order

Figure 7. *The Web Services Publisher dialog allows you to generate Web Service files for a VFP 7 COM component.*

One of the downsides of using the VFP 7 Web Services extensions is that you cannot pick and choose which methods in a COM server are published—it's an all or nothing proposition. If you need finer control, you can use the SOAP Toolkit directly.

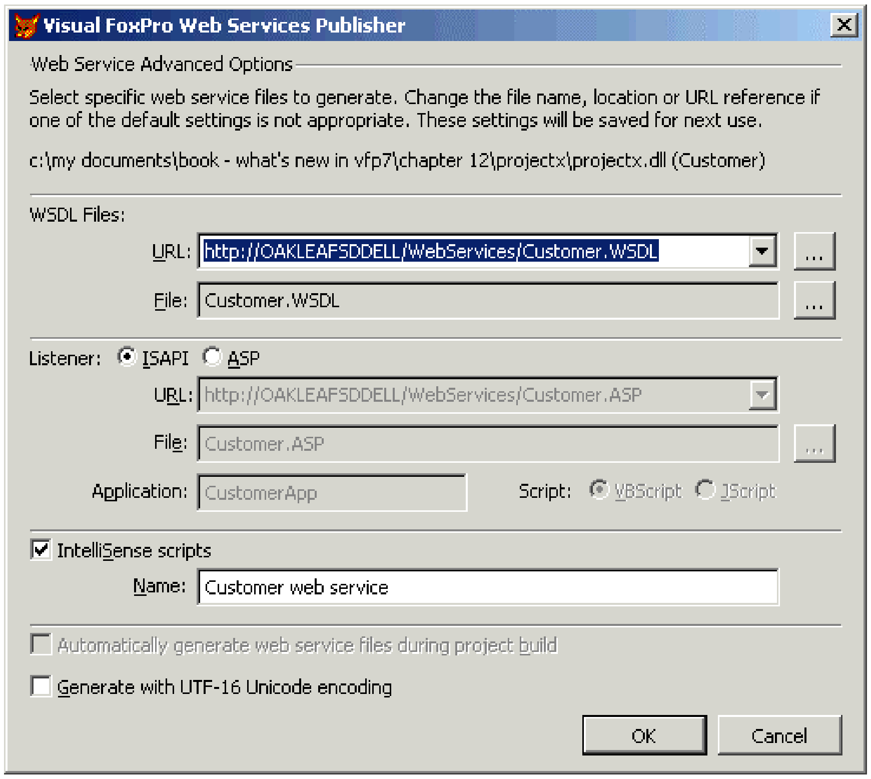
7. If you click the Advanced button, it launches the Web Services Publisher dialog shown in **Figure 8**. This dialog allows you to specify additional information such as:

   - The URL and name of the WSDL files
   - The type of Listener (ISAPI or ASP)
   - If using an ASP Listener, information regarding the ASP files
   - IntelliSense scripts

   For details on these settings, see the VFP 7 Help topic "Web Service Publishing."

8. Click the Generate button to generate the Web Service files.

If the Web Services Publisher is successful, it displays a dialog similar to the one shown in **Figure 9**.


Visual FoxPro Web Services Publisher

Web Service Advanced Options

Select specific web service files to generate. Change the file name, location or URL reference if one of the default settings is not appropriate. These settings will be saved for next use.

c:\my documents\book - what's new in vfp7\chapter 12\projectx\projectx.dll (Customer)

WSDL Files:

URL: http://OAKLEAFSDDELL/WebServices/Customer.WSDL

File: Customer.WSDL

Listener: [✓ ISAPI] ASP

URL: http://OAKLEAFSDDELL/WebServices/Customer.ASP

File: Customer.ASP

Application: CustomerApp Script: [✓ VBScript] JScript

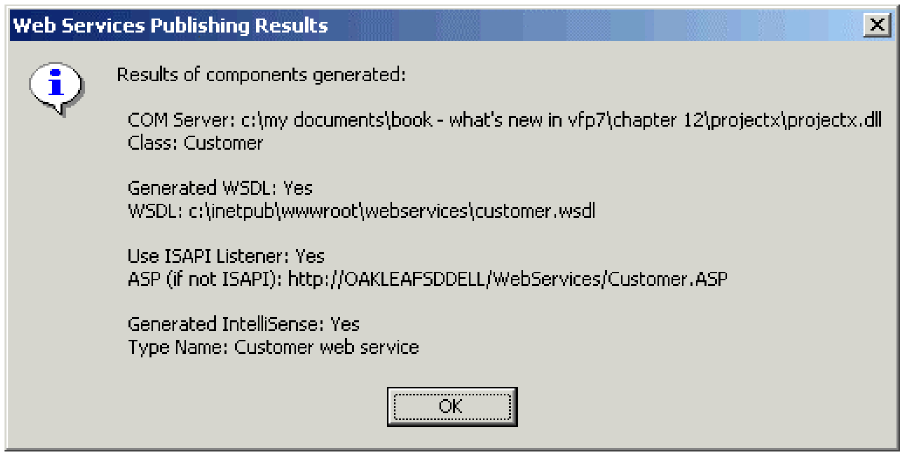
[✓] IntelliSense scripts

Name: Customer web service

[ ] Automatically generate web service files during project build

[ ] Generate with UTF-16 Unicode encoding

***Figure 8**. The Advanced Web Services Publisher dialog allows you to specify additional information regarding the generated Web Service files.*


Web Services Publishing Results

Results of components generated:

COM Server: c:\my documents\book - what's new in vfp7\chapter 12\projectx\projectx.dll
Class: Customer

Generated WSDL: Yes
WSDL: c:\inetpub\wwwroot\webservices\customer.wsdl

Use ISAPI Listener: Yes
ASP (if not ISAPI): http://OAKLEAFSDDELL/WebServices/Customer.ASP

Generated IntelliSense: Yes
Type Name: Customer web service

***Figure 9**. The Web Services Publisher displays the Results dialog if it successfully publishes Web Services for your COM component.*

The primary by-products of the publishing process are the WSDL and WSML files. These files are stored in the directory you specified in the Web Services Publisher dialog. Here is a partial listing of a WSDL file generated from the example ProjectX Customer class (all methods have been protected except GetCustomerByAcctNo):

```
<?xml version='1.0' encoding='UTF-8' ?>
 <!-- Generated 04/27/01 by Microsoft SOAP Toolkit WSDL File Generator, Version
1.00.623.0 -->
<definitions  name ='customer'   targetNamespace = 'http://tempuri.org/wsdl/'
     ...
  <message name='Customer.GetCustomerByAcctNo'>
    <part name='AcctNo' type='xsd:string'/>
  </message>
  <message name='Customer.GetCustomerByAcctNoResponse'>
    <part name='Result' type='xsd:string'/>
  </message>
  <portType name='CustomerSoapPort'>
    <operation name='GetCustomerByAcctNo' parameterOrder='AcctNo'>
      <input message='wsdlns:Customer.GetCustomerByAcctNo' />
      <output message='wsdlns:Customer.GetCustomerByAcctNoResponse' />
    </operation>
  </portType>
  <binding name='CustomerSoapBinding' type='wsdlns:CustomerSoapPort' >
    <stk:binding preferredEncoding='UTF-8'/>
    <soap:binding style='rpc' transport='http://schemas.xmlsoap.org/soap/http'
/>
    <operation name='GetCustomerByAcctNo' >
      <soap:operation
soapAction='http://tempuri.org/action/Customer.GetCustomerByAcctNo' />
      <input>
        <soap:body use='encoded' namespace='http://tempuri.org/message/'
        encodingStyle='http://schemas.xmlsoap.org/soap/encoding/' />
      </input>
      <output>
        <soap:body use='encoded' namespace='http://tempuri.org/message/'
        encodingStyle='http://schemas.xmlsoap.org/soap/encoding/' />
      </output>
    </operation>
  </binding>
  <service name='customer' >
    <port name='CustomerSoapPort' binding='wsdlns:CustomerSoapBinding' >
      <soap:address location='http://OAKLEAFSDDELL/WebServices/customer.wsdl'
/>
    </port>
  </service>
</definitions>
```

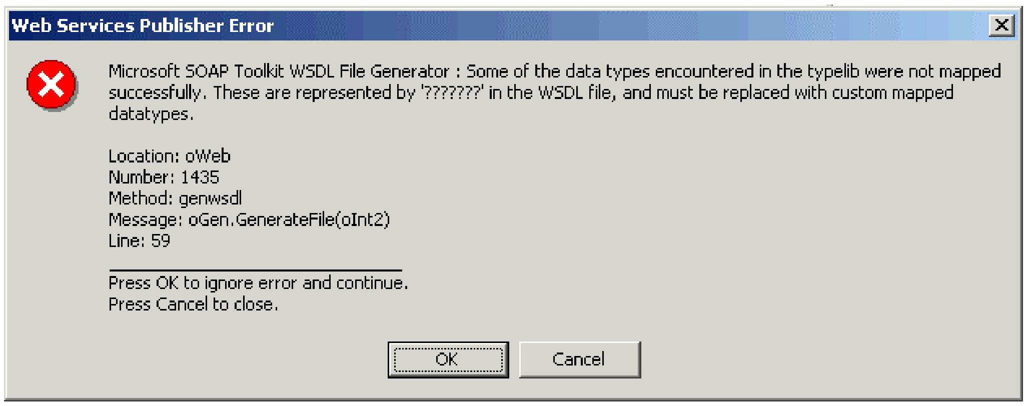
The Developer Download files at www.hentzenwerke.com include the Customer class referenced in this example.

Here is the associated WSML file generated by the Web Services Publisher:

```
<?xml version='1.0' encoding='UTF-8' ?>
 <!-- Generated 04/27/01 by Microsoft SOAP Toolkit WSDL File Generator, Version
1.00.623.0 -->
<servicemapping name='customer'>
  <service name='customer'>
    <using PROGID='projectx.Customer' cachable='0' ID='CustomerObject' />
    <port name='CustomerSoapPort'>
      <operation name='GetCustomerByAcctNo'>
        <execute uses='CustomerObject' method='GetCustomerByAcctNo' dispID='0'>
          <parameter callIndex='1' name='AcctNo' elementName='AcctNo' />
          <parameter callIndex='-1' name='retval' elementName='Result' />
        </execute>
      </operation>
    </port>
  </service>
</servicemapping>
```

For an explanation of the contents of these files, check the SOAP Toolkit 2.0 Help files.

If the Web Services Publisher encounters errors while mapping data types during generation of the Web Services files, you will see the dialog shown in **Figure 10**. You can examine the resulting WSDL file to determine the parameters and/or return values that could not be mapped.


Web Services Publisher Error

Microsoft SOAP Toolkit WSDL File Generator : Some of the data types encountered in the typelib were not mapped successfully. These are represented by '???????' in the WSDL file, and must be replaced with custom mapped datatypes.

Location: oWeb
Number: 1435
Method: genwsdl
Message: oGen.GenerateFile(oInt2)
Line: 59

Press OK to ignore error and continue.
Press Cancel to close.

***Figure 10**. The Web Services Publisher lets you know if errors are encountered when mapping data types of your COM component's methods.*

The Web Services project hook

The Web Services Publisher Advanced dialog (Figure 8) has a check box that allows you to automatically generate Web Service files during a project build. If you select this check box, it sets up a project hook for your project. This hook calls the Web Service engine to rebuild the support files whenever you build your project.

In addition, the Web Service project hook automatically terminates any IIS process that has a lock on your COM server. If you do not use the project hook, you must manually terminate the process using the Component Services Explorer.

Summary

Once again, the Fox team has done a great job of keeping Visual FoxPro in line with the future of software development. The enhancements to VFP 7 for XML and Web Services make it even easier to build distributed, Web-based, "next generation" applications.