CHAPTER

20

BizSnap Development: Writing SOAP-Based Web Services

IN THIS CHAPTER

Developing eBusiness solutions *rapidly* is key to the success of many organizations. Fortunately, Borland has made this rapid development possible through the use of a new Delphi 6 feature called BizSnap. BizSnap is a technology that integrates XML and Web Services using the SOAP protocol into Delphi 6.

What Are Web Services?

Borland describes Web Services as follows:

> Using the Internet and Web infrastructure as the platform, Web Services seamlessly connect applications, business processes, customers, and suppliers—anywhere in the world—with standardized language and machine-independent Internet protocols.

Distributed applications generally consist of servers and clients—servers that provide some functionality to the clients. Any distributed application might contain many servers, and those servers might themselves be clients. Web Services are a new type of server component for applications with a distributed architecture. Web Services are applications that use common Internet protocols to deliver their functionality.

Because Web Services communicate using open standards, they offer the opportunity for many different platforms to interoperate. For instance, from the perspective of a client application, a Web service deployed on a Sun Solaris machine will look (for all intents and purposes) identical to the same service deployed on a Windows NT machine. Prior to Web Services, this type of integration was extremely time-consuming, expensive, and generally proprietary.

This open nature and the ability to use existing network hardware and software position Web Services to be powerful tools for both internal and business-to-business transactions.

What Is SOAP?

SOAP is the acronym for *Simple Object Access Protocol*. SOAP is a lightweight protocol used for exchanging data in a distributed environment, similar to portions of CORBA and DCOM, but with less functionality and resulting overhead. SOAP exchanges data using XML documents, using HTTP (or HTTPS) for its communications. A specification on SOAP is available for reference on the Internet at `http://www.w3.org/TR/SOAP/`.

Web Services also use a form of XML to instruct users about themselves, called WSDL. WSDL is short for *Web Services Description Language*. WSDL is used by client applications to identify what a Web service can do, where it can be found, and how to call it.

The wonderful thing about BizSnap is that you don't have to learn all the specifics of SOAP, XML, or WSDL in order to create Web Service applications.

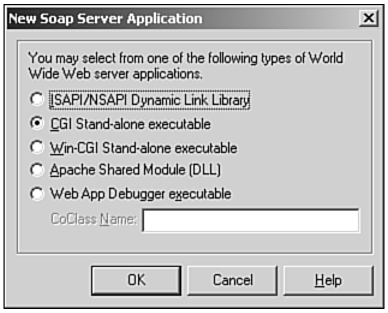
In this chapter, we will show you how simple it is to create a Web Service, and then we'll show you how to access this service from a client application.

Writing a Web Service

To demonstrate how to create a Web Service, we'll show you how to create the ever-popular Fahrenheit Celsius converter as a Web Service.

A Web service written in Delphi consists of three main things. The first is a WebModule with a few SOAP components (described in a moment). The module is automatically created for you when you execute the SOAP Server Wizard. The second two components you must build yourself. One of those is a class implementation, which is simply the code that describes what your Web service will actually do. The second thing to create is an interface to that class. The interface will expose only those pieces of the class that you want to offer to the rest of the world through your Web service.

Delphi provides a Web Service Wizard in the WebServices tab of the Object Repository. You will see three items in this tab. At this point, we'll concern ourselves with only the Soap Server Application Wizard. When you click this, you'll be shown the New Soap Server Application dialog box (see Figure 20.1). This dialog box should look familiar if you've done any Web Server development. In fact, Web Services are really Web Servers that handle the specific SOAP response.


New Soap Server Application

You may select from one of the following types of World Wide Web server applications.

( ) ISAPI/NSAPI Dynamic Link Library
[✓ CGI Stand-alone executable]
( ) Win-CGI Stand-alone executable
( ) Apache Shared Module (DLL)
( ) Web App Debugger executable

CoClass Name:

FIGURE 20.1
The New Soap Server Application dialog box.

In our example, we chose a CGI Stand-alone Executable. Click OK, and the wizard will generate a `TWebModule` as shown in Figure 20.2.

A Look at the `TWebModule`

Three components exist on the `TWebModule`. The purpose of these components is as follows:

- `THTTPSoapDispatcher` receives SOAP messages and dispatches them to the appropriate *Invoker* as specified by its `Dispatcher` property.

- `THTTPSoapPascalInvoker` is the component referred to by the `THTTPSoap Dispatcher.Dispatcher` property. This component receives the SOAP message, interprets it, and then invokes the invokable interface being called by the message.
- `TWSDLHTMLPublish` is used to publish the list of WSDL documents that contain the information on available invokable interfaces. This allows clients other than Delphi to identify and use the methods made available through a given Web Service.

There is nothing that you have to do with the Web Module at this point. However, you must define and implement an invokable interface.

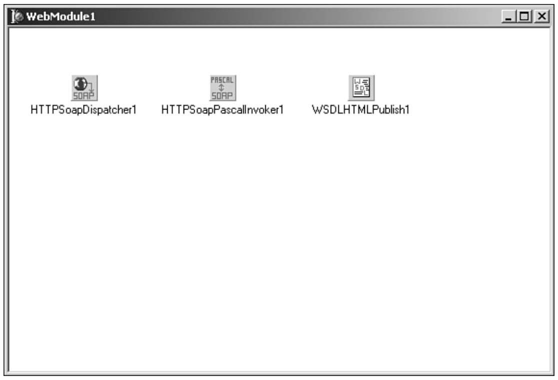


FIGURE 20.2
The Web Module generated from the wizard.

Defining an Invokable Interface

You must create a new unit in which you'll place your interface definition. Listing 20.1 shows the source code for the unit that we've created for our demonstration application, which you'll find on the CD. We've named this unit `TempConverterIntf.pas`.

LISTING 20.1 `TempConverter.pas—Invokable` Interface Definition

```
unit TempConverterIntf;

interface

type
  ITempConverter = Interface(IInvokable)
    ['{6D239CB5-6E74-445B-B101-F76F5C0F6E42}']
    function FahrenheitToCelsius(AFValue: double): double; stdcall;
    function CelsiusToFahrenheit(ACValue: double): double; stdcall;
```

LISTING 20.1 Continued

```
    function Purpose: String; stdcall;
  end;

implementation
uses InvokeRegistry;
initialization
  InvRegistry.RegisterInterface(TypeInfo(ITempConverter));

end.
```

This small unit contains only an interface that defines the methods that we intend to publish as part of our Web Service. You'll note that our interface descends from `IInvokable`. `IInvokable` is a simple interface compiled with the `{M+}` compiler option to ensure that RTTI is compiled into all of its descendants. This is necessary to allow the Web Services and Clients to translate code and symbolic information passed to each other.

In our example, we've defined two methods for converting temperatures and a `Purpose()` method that returns a string. Also, note that we provided a GUID for this interface to give it unique identification (to create a GUID in your own code, simply press Ctrl+Shift+G in the editor).

CAUTION

Note that each method defined in the invokable interface is defined using the `stdcall` calling convention. This convention must be used, or the invokable interface will not work.

Finally, the last items to note are the user of the `InvokeRegistry` unit and the call to `InvRegistry.RegisterInterface()`. The `THTTPSoapPascalInvoker` component must be able to identify the invokable interface when it is passed a SOAP message. The `RegisterInterface()` method call registers the interface with the invocation registry. When we discuss the client code later, you'll see that the `RegisterInterface()` call is also made on the client. The server requires the registration so that it can identify the interface implementation to execute on an interface call. On the client, the method is used to allow components to look up information on invokable interfaces and how to call them. By placing the `RegisterInterface()` call in the initialization block, we ensure that the method is called when the service is run.

Implementing an Invokable Interface

Implementing an invokable interface is no different from implementing any interface. Listing 20.2 shows the source for our temperature conversion interface.

LISTING 20.2 `TempConverterImpl.pas`—Invokable Interface Implementation

```
unit TempConverterImpl;

interface
uses InvokeRegistry, TempConverterIntf;
type

  TTempConverter = class(TInvokableClass, ITempConverter)
  public
    function FahrenheitToCelsius(AFValue: double): double; stdcall;
    function CelsiusToFahrenheit(ACValue: double): double; stdcall;
    function Purpose: String; stdcall;
  end;

implementation

{ TTempConverter }

function TTempConverter.CelsiusToFahrenheit(ACValue: double): double;
begin
// Tf = (9/5)*Tc+32
  Result := (9/5)*ACValue+32;
end;

function TTempConverter.FahrenheitToCelsius(AFValue: double): double;
begin
// Tc = (5/9)*(Tf-32)

  Result := (5/9)*(AFValue-32);
end;

function TTempConverter.Purpose: String;
begin
  Result := 'Temperature converstions';
end;

initialization
  InvRegistry.RegisterInvokableClass(TTempConverter);
end.
```

First, note that our interface implementation is a descendant of the `TInvokableClass` object. There are two primary reasons for doing this. The following reasons are take from the Delphi 6 online help:

- The invocation registry (`InvRegistry`) knows how to create instances of `TInvokableClass` and (because it has a virtual constructor) its descendants. This allows the registry to supply an invoker in a Web Service application with an instance of the invokable class that can handle an incoming request.
- `TInvokableClass` is an interfaced object that frees itself when the reference count on its interface drops to zero. Invoker components do not know when to free the implementation classes of the interfaces they call. Because `TInvokableClass` knows when to free itself, you do not need to supply your own lifetime management for this object.

Additionally, you'll see that our `TTempConverter` class implements the `ITempConverter` interface. The implementation methods for performing temperature conversions are self-explanatory.

In the initialization section, the call to `RegisterInvokableClass()` registers the `TTempConverter` class with the invocation registry. This is required only on the server so that the Web Service will be able to invoke the appropriate interface implementation.

That is really all there is to creating a simple Web Service. At this point, you can compile the Web Service and place it into an executable directory of a Web Server such as IIS or Apache. Typically, this would be a `\Scripts` or `\cgi-bin` directory.

Testing the Web Service

The URL `http://127.0.0.1/cgi-bin/TempConvWS.exe/wsdl/ITempConverter` was used to view the WSDL document generated from our Web Service. This service is hosted on an Apache server. To get a list of all the service interfaces available from a Delphi-generated Web service, the URL can be ended at `wsdl`. To see the specific WSDL document for a service, append the interface name desired—in this case `ItempConverter`. The resulting WSDL document is shown in Listing 20.3.

LISTING 20.3 Resulting WSDL Document from Web Service

```
<?xml version="1.0" ?>
- <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
➥xmlns:xs="http://www.w3.org/2001/XMLSchema" name="ITempConverterservice"
targetNamespace="http://www.borland.com/soapServices/"
➥xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
- <message name="FahrenheitToCelsiusRequest">
  <part name="AFValue" type="xs:double" />
  </message>
- <message name="FahrenheitToCelsiusResponse">
  <part name="return" type="xs:double" />
  </message>
```

LISTING 20.3 Continued

```
- <message name="CelsiusToFahrenheitRequest">
  <part name="ACValue" type="xs:double" />
  </message>
- <message name="CelsiusToFahrenheitResponse">
  <part name="return" type="xs:double" />
  </message>
  <message name="PurposeRequest" />
- <message name="PurposeResponse">
  <part name="return" type="xs:string" />
  </message>
- <portType name="ITempConverter">
- <operation name="FahrenheitToCelsius">
  <input message="FahrenheitToCelsiusRequest" />
  <output message="FahrenheitToCelsiusResponse" />
  </operation>
- <operation name="CelsiusToFahrenheit">
  <input message="CelsiusToFahrenheitRequest" />
  <output message="CelsiusToFahrenheitResponse" />
  </operation>
- <operation name="Purpose">
  <input message="PurposeRequest" />
  <output message="PurposeResponse" />
  </operation>
  </portType>
- <binding name="ITempConverterbinding" type="ITempConverter">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
- <operation name="FahrenheitToCelsius">
  <soap:operation soapAction="urn:TempConverterIntf-
➥ITempConverter#FahrenheitToCelsius" />
- <input>
  <soap:body use="encoded" encodingStyle="http:
➥//schemas.xmlsoap.org/soap/encoding/"
namespace="urn:TempConverterIntf-ITempConverter" />
  </input>
- <output>
  <soap:body use="encoded" encodingStyle=
➥"http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:TempConverterIntf-ITempConverter" />
  </output>
  </operation>
- <operation name="CelsiusToFahrenheit">
  <soap:operation soapAction="urn:TempConverterIntf-
➥ITempConverter#CelsiusToFahrenheit" />
- <input>
```

LISTING 20.3 Continued

```
  <soap:body use="encoded" encodingStyle="http:
➥//schemas.xmlsoap.org/soap/encoding/"
namespace="urn:TempConverterIntf-ITempConverter" />
  </input>
- <output>
  <soap:body use="encoded" encodingStyle=
➥"http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:TempConverterIntf-ITempConverter" />
  </output>
  </operation>
- <operation name="Purpose">
  <soap:operation soapAction="urn:TempConverterIntf-ITempConverter#Purpose" />
- <input>
  <soap:body use="encoded" encodingStyle=
➥"http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:TempConverterIntf-ITempConverter" />
  </input>
- <output>
  <soap:body use="encoded" encodingStyle=
➥"http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:TempConverterIntf-ITempConverter" />
  </output>
  </operation>
  </binding>
- <service name="ITempConverterservice">
- <port name="ITempConverterPort" binding="ITempConverterbinding">
  <soap:address location="http://127.0.0.1/cgi-bin/TempConvWS.exe
➥/soap/ITempConverter" />
  </port>
  </service>
  </definitions>
```

Now we'll show you how simple it is to invoke a Web Service.

Invoking a Web Service from a Client

To invoke the Web Service, you must know the URL used to retrieve the WSDL document. This is the same URL we used earlier.

To demonstrate this, we used a simple application with single, main form (see Figure 20.3).

This application is straightforward: The user enters a temperature in the edit control, presses the desired conversion button, and the converted value is displayed in the Temperature label. The source for this application is shown in Listing 20.4.

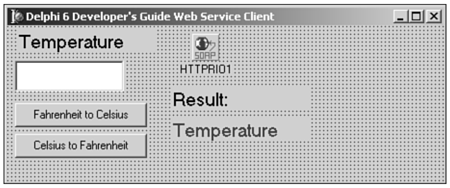


FIGURE 20.3
The Main Form to the Web Service Client Application.

LISTING 20.4 Web Service Client

```
unit MainFrm;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Rio, SoapHTTPClient;

type
  TMainForm = class(TForm)
    btnFah2Cel: TButton;
    btnCel2Fah: TButton;
    edtArguement: TEdit;
    lblTemperature: TLabel;
    lblResultValue: TLabel;
    lblResult: TLabel;
    HTTPRIO1: THTTPRIO;
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  MainForm: TMainForm;

implementation

uses TempConvImport;

{$R *.dfm}

end.
```

On the main form, we've placed a THTTPRIO component. A THTTPRIO represents a remotely invokable object, and acts as a local proxy for a Web service that very likely resides on a remote machine somewhere. The two TButton event handlers perform the code to invoke the remove object from our Web Service. Note that we must cast the THTTPRIO component as ITempConverter to refer to it. Then, we are able to invoke its method call.

Before any of this code will run, we must prepare the THTTPRIO component, which requires a few steps.

Generating an Import Unit for the Remote Invokable Object

Before we are able to use the THTTPRIO component, we need to create an import unit for our invokable object. Fortunately, Borland made this easy by providing a wizard to handle this. This wizard is available on the WebServices page of the Object Repository. When launched, you'll see the dialog box shown in Figure 20.4.

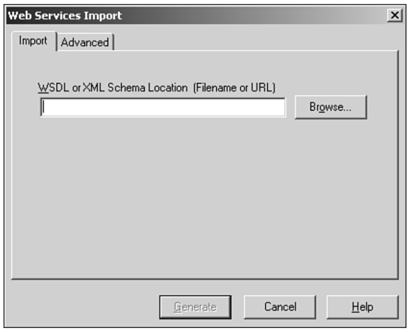


FIGURE 20.4
The Web Services Import Wizard.

In order to import a Web service into a client application, you put the WSDL path (the URL specified earlier) in the Schema Location and then press the Generate button to create the import unit. The import unit for our Web Service is shown in Listing 20.5 and looks almost exactly like our original interface definition unit.

LISTING 20.5 Web Service Import Unit

```
Unit TempConvImport;

interface

uses Types, XSBuiltIns;
```

LISTING 20.5 Continued

```
type

  ITempConverter = interface(IInvokable)
    ['{684379FC-7D4B-4037-8784-B58C63A0280D}']
    function FahrenheitToCelsius(const AFValue: Double): Double;  stdcall;
    function CelsiusToFahrenheit(const ACValue: Double): Double;  stdcall;
    function Purpose: WideString;  stdcall;
  end;


implementation

uses InvokeRegistry;

initialization
  InvRegistry.RegisterInterface(TypeInfo(ITempConverter),
➥ 'urn:TempConverterIntf-ITempConverter', '');

end.
```

Once it has been generated, return to the main form of the client application and use the newly-generated import unit. This will make the main form aware of the new interface.

Using the `THTTPRIO` Component

Three properties must be set for the `THTTPRIO` component. The first, `WSDLLocation`, needs to contain, once again, the path to the WSDL document. Once set, you can drop down the `Service` property to select the only available option. Then, do the same for the `Port` property. At this point, you will be able to run the client.

Putting the Web Service to Work

Now that all the pieces are in place, create an event handler for the button's `OnClick` event by double-clicking on it. The event should look like Listing 20.6.

LISTING 20.6 `OnClick` Event Handler

```
procedure TMainForm.btnFah2CelClick(Sender: TObject);
var
  TempConverter: ITempConverter;
  FloatVal: Double;
begin
  TempConverter := HTTPRIO1 as ITempConverter;
  FloatVal := TempConverter.FahrenheitToCelsius(StrToFloat(edtArguement.Text));
```

LISTING 20.6 Continued

```
  lblResultValue.Caption := FloatToStr(FloatVal);
end;

procedure TMainForm.btnCel2FahClick(Sender: TObject);
var
  TempConverter: ITempConverter;
  FloatVal: Double;
begin
  TempConverter := HTTPRIO1 as ITempConverter;
  FloatVal := TempConverter.CelsiusToFahrenheit(StrToFloat(edtArguement.Text));
  lblResultValue.Caption := FloatToStr(FloatVal);
end;
```

While entering this code, notice that Delphi's CodeInsight is available for the Web service itself. This is because Delphi has adapted the Web service into your application as a native object. The implications here are very broad-ranging: any Web service brought into a Delphi application, regardless of whether that service is deployed on Solaris, Windows, Linux, a mainframe, and independent of what language the service is written in, will benefit from this. In addition to CodeInsight, an application written to use a Web service will also gain compiler typechecking and other debugging features because of this tight integration.

Summary

Web Services are a powerful new tool in distributed computing, using open standards and existing infrastructure to enable interoperation within and between different platforms.

In this chapter, we showed you how to create a simple Web Service and the Client to use this service. We demonstrated the steps required to deploy this server and to set up the client's `THTTPRIO` component properly. At this point, you should be familiar enough with developing Web Services in greater complexity. You can examine many more examples of Web Services on Borland's community site. One that we highly recommend is "Managing Sessions with Delphi 6 Web Services." This article was written by Daniel Polistchuck (Article ID: 27575) and can be read at `http://community.borland.com/article/0,1410,27575,00.html`.