Under Construction:
Delphi 6 Web Services: SOAP

by Bob Swart

When I wrote last month, Delphi 6 had just been announced but was not yet available. Delphi 6 has now been available for a short while, and I've been playing with my copy of Delphi 6 Enterprise, especially with the new SOAP support, which is most notably featured in the new functionality called Web Services. Another topic that's hot on my list to cover is WebSnap, the second enhancement to WebBroker (the first enhancement was InternetExpress, which seems to have been left on a side track this time). WebSnap is new, but not very well documented to be honest: it combines lots and lots of possibilities with a significant learning curve, so I want to do some more exploration of WebSnap and start my (multipart) coverage of this featureset next month in these pages. For now, let's talk Web Services and start to use SOAP.

BizSnap: Web Services

One of the hot new features of Delphi 6 is called BizSnap. As part of BizSnap, we now have SOAP support and we can make Web Services (one of the most recent 'hyped-up' topics). Before I can explain why we need SOAP in the first place, let's talk a bit about the end result: a Web Service. Why is this so hot, and what does Delphi 6 offer that other tools don't have yet)? A Web Service can be seen as the server side of a distributed application. Using SOAP (Simple Object Access Protocol), we make the methods of our remote objects available to the outside world. The actual description of the Web Service is written in WSDL, which stands for Web Service Description Language, a subset of XML. Don't worry: you do not really have to learn very much about SOAP, XML, WSDL or some of the other acronyms that I won't even cover (like UDDI) in order to use Delphi 6 to produce a working Web Service, as we will see in this step-by-step article.

Building Web Services

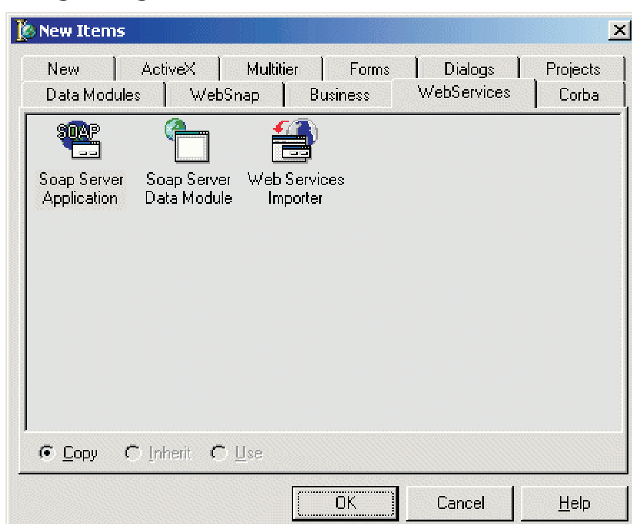
Delphi 6 BizSnap contains support for SOAP and XML in the form of Web Services. You can think of a Web Service as a 'remote object instance' (almost like a remote component) that any client application can use to execute or provide some service ('interface'). It is very similar to what CORBA and DCOM have been doing for years, but Web Services use SOAP/XML as the protocol to exchange messages. *[The advantage of SOAP over CORBA or DCOM is that it is text-based (using XML) and so penetrates firewalls easily. Ed]*

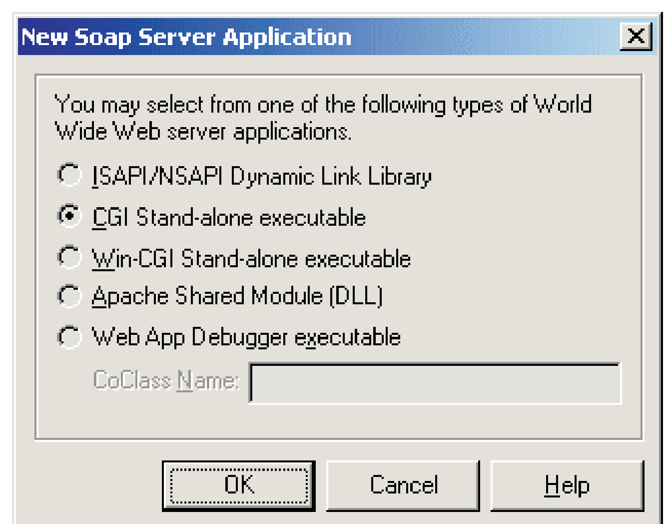
If this doesn't help to make it clear, yet, let's just go ahead and make a Web Service. In the end, it will turn out to be very simple, and yet powerful enough (SOAP is both platform- and language independent) to change the world of distributed computing.

In order to make a Web Service, we must go to the `Web Services` tab of the Object Repository, and select the `Soap Server Application` wizard (the other two Web Service wizards are `Soap Server Data Module` and the `Web Services Importer`: we will see the latter in the second half of this article).

Once you start the wizard you almost feel like you've started the wrong one (the web server application wizard) by mistake. We can make a choice for our type of web server application. Web Services made by Delphi 6 are in fact WebBroker applications that respond to certain input requests.

➤ *Below: Figure 1*
Right: Figure 2





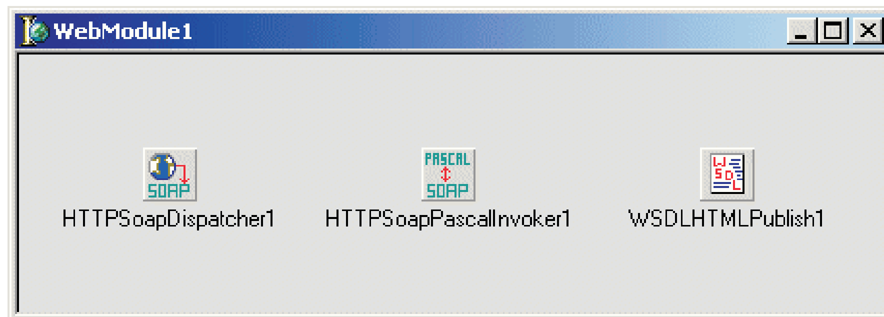
For a real-world Web Service, you may need to select an ISAPI/NSAPI or Apache Shared Module (DLL) application. For debugging purposes you can use the new Web App Debugger executable target, which is based on COM (but cannot be deployed, so you need to change your project wrapper to a different target once you're done debugging).

For now, select the CGI standalone executable (the easiest choice to deploy while writing this article) and click on the `OK` button. Save the resulting web module as SWebMod.pas (notice that the web module already has components on it: `HTTPSoapDispatcher1`, `HTTPSoapPascalInvoker1` and `WSDLHTMLPublish1`), and save the Web Service project as WebService42.dpr.

The `THHTPSoapDispatcher` component is the one that manages the calls to the web server executable, dispatching the incoming actions (like `/WSDL` and `/SOAP`). It provides customised support for SOAP applications by providing some standard actions that provide basic information about the Web Service itself.

The `HTTPSoapPascalInvoker` is the component that receives the SOAP requests and converts it to our own ObjectPascal implementation. This one uses an interface derived from `IInvokable` (that is, an invocable interface). *[And yes, 'invocable' is a real English word, but 'invokable' is not! Ed]*

The `TWSDLHTMLPublish` component publishes the WSDL interface for the Web Service, so that others can access it easily and use it.



➤ *Figure 3*

Without this, nobody would know what the Web Service could do, which isn't very useful of course.

The good news is that we don't have to do (or change) anything with these components. Just prepare the next step, which consists of defining the interface to the outside world.

Interface

In order to add a Web Service, you have to add an interface, derived from `IInvokable`, and provide an implementation for it. For example, let's define an interface called `THitchHiker` with two functions: `TheAnswer` and `TheQuestion`. First, we have to manually add a new unit to our project using `File | New | Unit`, save it as HitchHiker.pas and add the code in Listing 1. Note that we can get a fresh unique GUID by pressing `Ctrl+G` inside the Delphi 6 code editor.

Note that we don't need anything special to define the interface. But we do have to include the `InvokeRegistry` unit in order to be able to register our interface. Not that we can do anything with it, yet, since we haven't implemented it, but at least we are publishing the interface, its methods and arguments (if we have used any).

At this time, you can still change the interface, which will be a contract between the Web Service 'server' application and any Web Service 'client' that wants to make use of it. You must realise that once you've made your Web Service available worldwide (by deploying it on a web server), users will not appreciate it if you change the interface again! You

➤ *Listing 1: IHitchHiker Interface.*

```
unit HitchHiker;
interface
type
  IHitchHiker = interface(IInvokable)
    ['{06723E05-662D-11D5-81CE-00104BF89DAD}']
    function TheAnswer: Integer; stdcall;
    function TheQuestion: String; stdcall;
  end;
implementation
uses InvokeRegistry;
initialization
  InvRegistry.RegisterInterface(TypeInfo(IHitchHiker));
end.
```

➤ *Listing 2: IHitchHiker Implementation.*

```
unit HitchHiker;
interface
uses
  InvokeRegistry;
type
  IHitchHiker = interface(IInvokable)
    ['{06723E05-662D-11D5-81CE-00104BF89DAD}']
    function TheAnswer: Integer; stdcall;
    function TheQuestion: String; stdcall;
  end;
  THitchHiker = class(TInvokableClass, IHitchHiker)
  public
    function TheAnswer: Integer; stdcall;
    function TheQuestion: String; stdcall;
  end;
implementation
{ THitchHiker }
function THitchHiker.TheAnswer: Integer;
begin
  Result := 42
end;
function THitchHiker.TheQuestion: String;
begin
  Result := 'What do you get when you multiply six by nine?'
end;
initialization
  InvRegistry.RegisterInterface(TypeInfo(IHitchHiker));
  InvRegistry.RegisterInvokableClass(THitchHiker);
end.
```

may get away with adding new methods, but changing the signature of one or more methods is not recommended (and will quickly discourage people from using your Web Services at all).

It doesn't really matter that the documentation (or rather, the WSDL specification) of your Web Service is automatically made public and produced dynamically so it's never outdated (as we'll see in a moment): a Web Service client can still be made based on a previous interface, which means it can still be broken.

Implementation

Once the interface is complete (and fixed in stone), it's time to implement it (Listing 2). Note that the functions that I defined in the interface are all explicitly using the `stdcall` calling convention. This is mandatory, so be sure not to forget it or your Web Service will not work: been there, done that, had to change it back to `stdcall`.

Now we have a `WebService42` Web Service, with two exported methods: `TheAnswer` and `TheQuestion`. To test it, just move it to a Scripts directory, and execute it with the `/wsdl/IHitchHiker pathinfo` argument. If you didn't type with me, you can still see the output at

```
http://drbob42.tdmweb.com/
  cgi-bin/WebService42.exe
```

since I've also moved the 763,904 bytes Web Service application to the cgi-bin directory of my website (hosted by TDMWeb, which offers you the choice of a Linux web server or a Windows 2000 web server: my Kylix and Delphi web server applications are indeed in safe hands).

When using Netscape Navigator (at least my version 4.7), I'm prompted to open the resulting XML file (inside Internet Explorer) or save it to disk. The resulting XML file when saved to disk is 2,282 bytes in size and is included on the companion disk (it's not hard to read, but contains a number of long lines). It doesn't hurt to save this file, since it may come in handy at a later time (when we have to import the WSDL definition, either by using the full URL seen in Internet Explorer, or by specifying the XML file that contains it).

The XML consists of a number of sections, and contains the SOAP definition for our Web Service. It can be used by any environment, including the upcoming Visual Studio .NET (now in Beta 2). This means that we can write distributed multi-tier applications using Delphi and C#, provided that the interface is using data types that are native to both environments (I wouldn't want to try to pass a Midas data packet from Delphi to C# for example, but that's a story for another day).

➤ *Listing 3: IHitchHiker SOAP XML.*

```
<?xml version="1.0"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:xs="http://www.w3.org/2001/XMLSchema"
             name="IHitchHikerservice" targetNamespace="http://www.borland.com/soapServices/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
  <message name="TheAnswerRequest"/>
  <message name="TheAnswerResponse">
    <part name="return" type="xs:int"/>
  </message>
  <message name="TheQuestionRequest"/>
  <message name="TheQuestionResponse">
    <part name="return" type="xs:string"/>
  </message>
  <portType name="IHitchHiker">
    <operation name="TheAnswer">
      <input message="TheAnswerRequest"/>
      <output message="TheAnswerResponse"/>
    </operation>
    <operation name="TheQuestion">
      <input message="TheQuestionRequest"/>
      <output message="TheQuestionResponse"/>
    </operation>
  </portType>
  <binding name="IHitchHikerbinding" type="IHitchHiker">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="TheAnswer">
      <soap:operation soapAction="urn:HitchHiker-IHitchHiker#TheAnswer"/>
      <input>
        <soap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              namespace="urn:HitchHiker-IHitchHiker"/>
      </input>
      <output>
        <soap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              namespace="urn:HitchHiker-IHitchHiker"/>
      </output>
    </operation>
    <operation name="TheQuestion">
      <soap:operation soapAction="urn:HitchHiker-IHitchHiker#TheQuestion"/>
      <input>
        <soap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              namespace="urn:HitchHiker-IHitchHiker"/>
      </input>
      <output>
        <soap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
              namespace="urn:HitchHiker-IHitchHiker"/>
      </output>
    </operation>
  </binding>
  <service name="IHitchHikerservice">
    <port name="IHitchHikerPort" binding="IHitchHikerbinding">
      <soap:address location="http://drbob42.tdmweb.com/cgi-bin/WebService42.exe/soap/IHitchHiker"/>
    </port>
  </service>
</definitions>
```

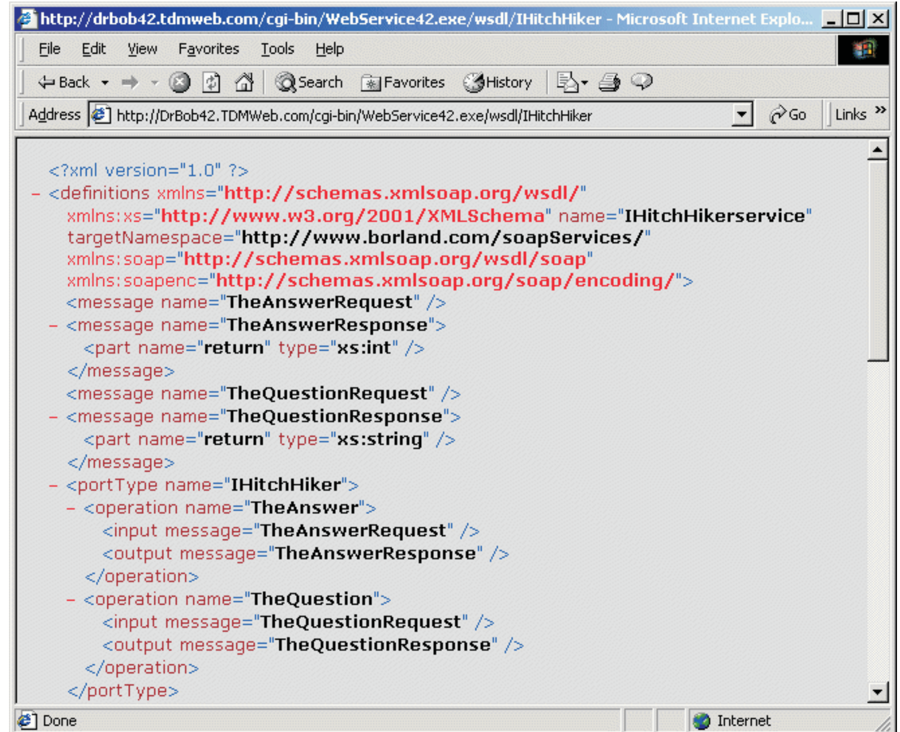
Using Web Services

Now, publishing a Web Service is one thing, but only half of the story. It really wouldn't be complete if we didn't try to use it again. And this time we'll be using the remote Web Service (the one on my web server at http://drbob42.tdmweb.com/cgi-bin/WebService42.exe). We need two things: the URL that returns the XML with the WSDL (WebService Description Language) and the Web Services Importer from the Delphi 6 Object Repository.

Start a new Delphi 6 project. It can be anything you like, since you can use a Web Service in any client (except for a CLX application at this time, but I'm sure that it won't be long until Kylix Enterprise Studio is available, which will also contain Web Service support).

To keep it simple, let's start a regular application, and save the form in MainForm.pas and the application itself in TDM72.dpr. Drop two buttons (`btnAnswer` and `btnQuestion`) on the form and two `LabeledEdit` controls (`lbeAnswer` and `lbeQuestion`).

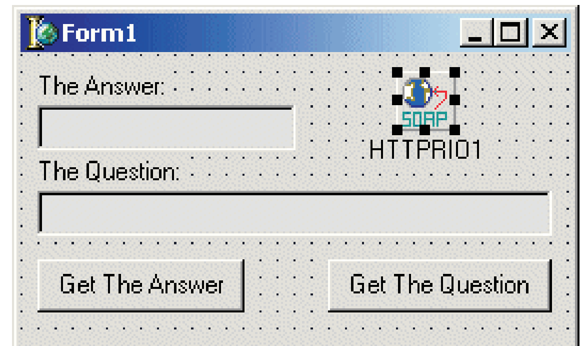
Just in case you didn't read the Delphi 6 review last month: we now have support for nested components as sub-properties. The `TLabeledEdit` component is one such example, where the `EditLabel` property is in fact a `TLabel` component, for which we need to set the `Caption`. The Delphi 6 Object Inspector allows us to edit sub-components as properties, including the properties of the sub-component. So without leaving the `LabeledEdit` component, we can now edit the `Caption` of



➤ *Figure 4*

the `EditLabel` sub-component using the Object Inspector.

Set the `EditLabel.Caption` sub-property of the first `LabeledEdit` to `The Answer`, and the second to `The Question`.



➤ *Figure 5*

HTTPRIO

There's one more component we need to use the Web Service application, and that's the `HTTPRIO` component from the `Web Services` tab of the Delphi 6 Component Palette. This component represents a remote invokable object over an HTTP connection (hence the name `HTTPRIO`). The resulting form should now look like Figure 5.

➤ *Listing 4: IHitchHiker Import Unit.*

```
Unit HitchHikerImport;
interface
uses
  Types, XSBuiltIns;
type
  IHitchHiker = interface(IInvokable)
    ['{E325004C-3203-482D-95F8-30D0286C68DC}']
    function TheAnswer: Integer;  stdcall;
    function TheQuestion: WideString;  stdcall;
  end;
implementation
uses
  InvokeRegistry;
initialization
  InvRegistry.RegisterInterface(TypeInfo(IHitchHiker),
   'urn:HitchHiker-IHitchHiker', '');
end.
```

Before we can actually use the `HTTPRIO` component, however, we need to generate a special import unit that represents the remote object. For this we need the Web Services Importer wizard from the Object Repository. On the first page, there is one editbox that needs the full URL for the Web Server Description Language file of our `WebService42` project. In our example (and also for your own experiments if you like), we can use the Web Service hosted on my website at

```
http://DrBob42.TDMWeb.com/
  cgi-bin/WebService42.exe/
  wsdl/IHitchHiker
```

As an alternative, you can load the resulting XML file (for example, when you use Netscape Navigator

```
unit MainForm;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, Rio, SoapHTTPClient, StdCtrls,
  ExtCtrls;
type
  TForm1 = class(TForm)
    bntAnswer: TButton;
    btnQuestion: TButton;
    lbeAnswer: TLabeledEdit;
    lbeQuestion: TLabeledEdit;
    HTTPRIO1: THTTPRIO;
    procedure bntAnswerClick(Sender: TObject);
    procedure btnQuestionClick(Sender: TObject);
  private
  public
  end;
var
  Form1: TForm1;
```

```
implementation
{$R *.dfm}
uses
  HitchHikerImport;
procedure TForm1.bntAnswerClick(Sender: TObject);
var
  HitchHiker: IHitchHiker;
begin
  HitchHiker := (HTTPRIO1 AS IHitchHiker);
  lbeAnswer.Text := IntToStr(HitchHiker.TheAnswer)
end;
procedure TForm1.btnQuestionClick(Sender: TObject);
var
  HitchHiker: IHitchHiker;
begin
  HitchHiker := (HTTPRIO1 AS IHitchHiker);
  lbeQuestion.Text := HitchHiker.TheQuestion
end;
end.
```

➤ *Listing 5: HitchHiker WebService Usage.*

to go to the above URL, you get the option to save the XML file, and you can use this file to create the import unit).

Never mind that the XML file is not connected to the actual remote Web Service, because you still need to tell the `HTTPRIO` component the (invocation) location of the remote Web Service, so a local file to generate the import units for the Web Service works just as well.

We can use the `Advanced` tab to set some special options. I'll actually leave this in peace, since the default options are OK for this example. The generated unit is automatically added to the current project, and I've saved it in the file HitchHikerImport.pas. The source code for this unit can be seen in Listing 4.

Now, where have we seen this before? You must admit that it looks very much like Listing 1 which we saw earlier. In fact, the only differences are the units included in the `uses` clause of the interface section (which can be removed, by the way) and the GUID, which now refers to the SOAP object on the remote web server. Now that we have this interface definition, we can use the `HTTPRIO` component to create an instance of the remote server (as remote invocable object) and start using the interface.

Using HTTPRIO

Now that we have the import unit and `HTTPRIO` component on our main form, we still need to add the import unit to the main form, so add `uses HitchHikerImport`, to the `implementation` section of the MainForm.pas unit.

Now, click on the `HTTPRIO` component, move to the Object Inspector and make sure to enter the full URL for our Web Service WSDL XML description in the `WSDLLocation` property:

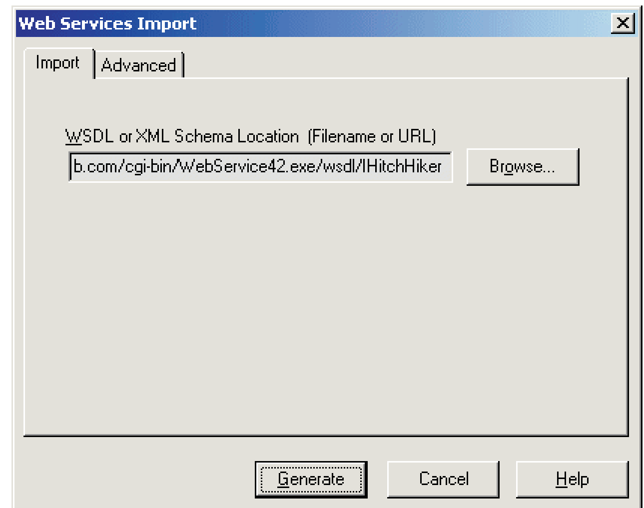
```
http://DrBob42.TDMWeb.com/
  cgi-bin/WebService42.exe/
  wsdl/IHitchHiker
```

I highly recommend the use of copy and paste in all cases like this that involve a WSDL URL, by the way: it could save you a lot of hassle later!

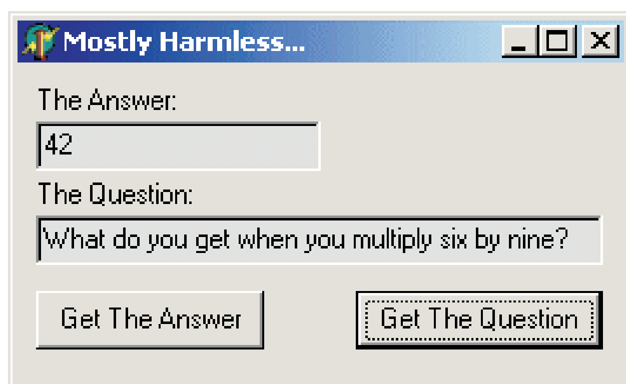
After we have specified the `WSDLLocation` property, we need to select a value for the `Service` property (just open up the drop-down combobox to select the only possible value: `IHitchHikerservice`) and a value for the `Port` property (after you've selected a service, open the drop-down combobox for the `Port` property and select the only possible value: `IHitchHikerPort`).

That's all there is to it to let the `HTTPRIO` be a connector to a remote invocable object over HTTP. In our client application, we can now treat the `HTTPRIO` component as a remote object (one that has implemented the `IHitchHiker` interface), and hence extract the `IHitchHiker` interface from it and use it directly.

Obviously, we need an internet connection (or rather an HTTP connection) to the Web Service server, but you can also deploy Web Service servers on your local intranet or even on your local machine, although for the solution that I've presented in this paper you need a web server as well to



➤ *Figure 6*



➤ *Figure 7*

host the WebBroker application, of course.

The full, but not very long, source code for the Web Service client can be seen in Listing 5. Note that I simply need to cast the `HTTPRIO` instance to my `IHitchHiker` interface type (the one from the `HitchHikerImport` unit) before I can use it.

The first time you click on any of the buttons the response will take a while: the connection has to be made, and so on. However, subsequent requests to the same Web Service will be much faster, as you will see when you try this example for yourself.

As long at the Web Service on my web server is running you can connect to it. However, for a more extensive list of Web Services (created in various different environments), see the www.xmethods.net website: I hope to be listed there as well by the time you read this!

Next Time

Apart from Web Services, Delphi 6 introduced a number of other additional new web server enhancements and functionality. One example is WebSnap, which is another extension or enhancement of the good old WebBroker Technology. For Delphi 5, WebBroker had already been extended with MIDAS support using `XMLBroker` and the `MidasPageProducer`, resulting in a toolset called InternetExpress. WebSnap can be compared with InternetExpress in that it is an extension of WebBroker, and it uses some of the same ideas. But it consists of many more components, wizards and other tools than I've ever seen in InternetExpress and there is a lot less documentation (although Borland is working hard on more documentation and examples at this time). Nevertheless, in the next two or three months I plan to cover WebSnap: explaining how to use it, how it works and also how to extend it with your own wishes. All this and more next time, so stay tuned!

Bob Swart (aka Dr.Bob, www.drbob42.com) is an @-Consultant, Delphi Clinic Trainer and co-founder of the Kylix/Delphi OplossingsCentrum (visit www.KDOC.nl) in The Netherlands.