Dr.Bob's Prescriptions N°5: GoogleSearch using C++Builder 6 and Web Services

by Bob Swart

C++Builder 6 has been available for a number of months now, but has gone relatively unnoticed. Most people know by now that SOAP and Web Services are among the hot new features of Delphi 6, but the same (and sometimes better) support can be found in C++Builder 6 as well. In fact, apart from the C++ source code that we have to write at the end of the article, you would hardly know the difference between using C++Builder 6 and Delphi 6.02

C++Builder 6

This article explains how to consume existing Web Services using C++Builder 6 (either Professional or Enterprise). I use a Web Service to search the web using the Google search engine. Not the official Google search engine (which is a bit more complex, and for which you need to register yourself first), but a simple one implemented by Santra Technology in MS Visual Studio .NET. The main reason why I've decided to use C++Builder 6 to consume a Web Service written in another environment is to illustrate the cross-language nature of Web Services right from the start.

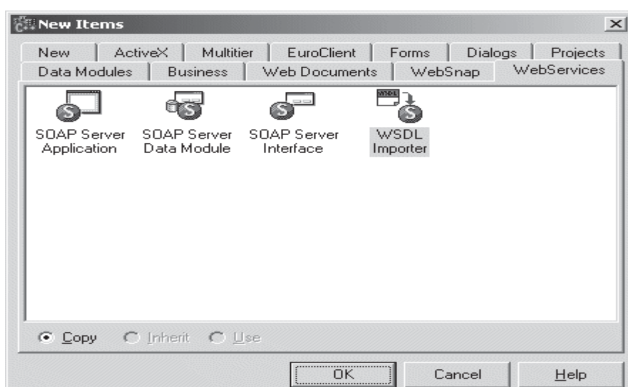
Google

The example Web Service that we will be using in this article is the GoogleSearch, available on the internet as http://rob.santra.com/webservices/public/google/ with the formal WSDL definition at http://rob.santra.com/webservices/public/google/index.asmx?WSDL. Using a browser such as Internet Explorer, we can view the WSDL (Web Services Description Language) of this web service. More useful, however, is to take this WSDL URL and feed it to C++Builder 6 in order to generate an import unit that we can use in our own applications.

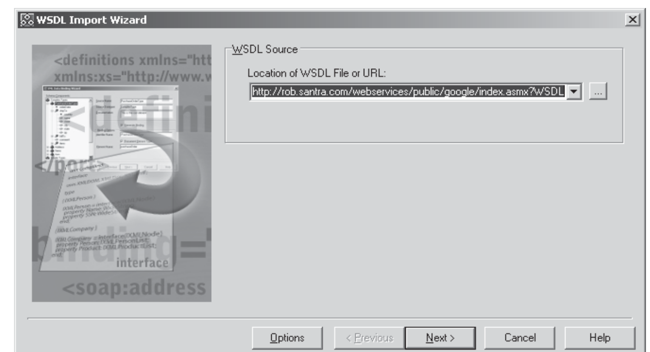
C++Builder 6

Start C++Builder 6 (Professional or Enterprise), and save the new default project (or create a new one). Save the main form in file SearchForm.cpp and the project itself in Google42.bpr. Like I said, before we can use the GoogleSearch web service, we first need to create a C++ import unit based on the WSDL definition. This can be done using the WSDL Importer.

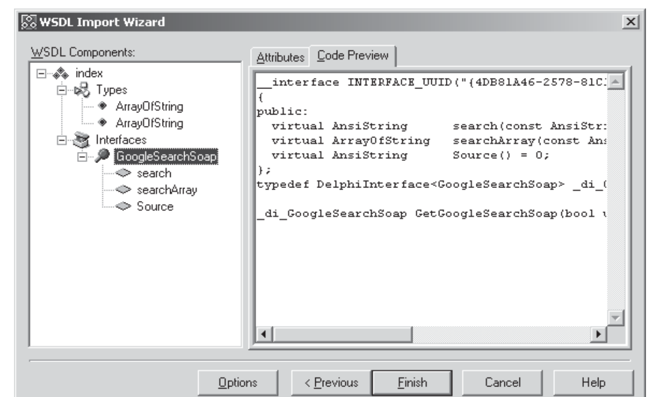
Do *File | New - Other*, and go to the WebServices tab of the Object Repository. Here, you'll find four SOAP specific wizards (C++Builder 6 professional users will only find the WSDL Importer wizard, but can still work along with this article, because that's the only wizard we're using this time).



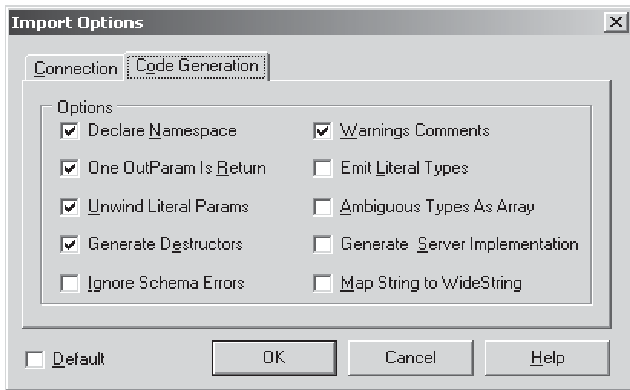
When you start the WSDL Importer, you'll get a wizard that starts by asking the location of the WSDL (the Web Service Description Language). Note that this can either be a local file that you saved (with the WSDL contents, according to the previous section), or a live URL to the web service on the website, which in our case is http://rob.santra.com/webservices/public/google/index.asmx?WSDL (if you do not want to be connected to the internet while developing the Google42 application, you can save the WSDL in a local file and use that one instead).



When you close down the Import Options dialog, and click on the Next button, you get to the second (and last) page of the WSDL Import Wizard. In this page, you are presented with an overview of the WSDL components, including the types and interfaces (in this case only GoogleSearchSoap) and the methods search, searchArray and Source that are available, including all attributes and source code that will be generated, as you can see in the screenshot.



The Options button is available on all pages, and will enable you to specify a number of Connection (through a Proxy) and Code Generation options that you may want to set (I personally use the default settings, but you may want to check out what's possible just in case you ever need it).



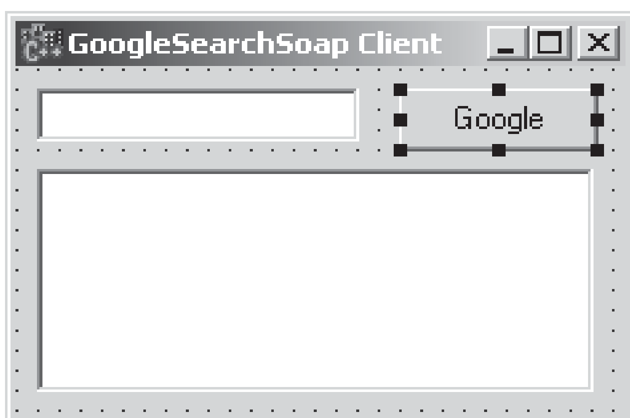
For closer examination of the generated source code, just click on the Finish button, which will generate the new import unit index.cpp for the WSDL file or location specified. The header file index.h contains as a most important part the definition of the GoogleSearchSoap interface, derived from IInvokable, and containing the three abstract virtual methods search(), searchArray() and Source(). Search() can be used to search Google, returning the result in a single AnsiString, while searchArray() will return an array of AnsiStrings (one for each result). Source() will return the source code for the Web Service itself, which turns out to be written in C#.

The corresponding implementation inside file index.cpp contains a function called GetGoogleSearchSoap(), which makes the usage of the generated GoogleSearchSoap interface inside the generated files very simple. We only need to call the GetGoogleSearchSoap() function (or in general any Get followed by the name of a SOAP interface) to get an interface back to the SOAP server that we want to use.

In our case, the call to GetGoogleSearchSoap() returns the GoogleSearchSoap interface - implemented by a component behind the scenes - ready for us to use. Since the GoogleSearchSoap interface contains only three methods - two of which implement the actual Google search - we can simply select one, for example the simple search() method which takes a search query of type AnsiString as argument and returns an AnsiString with the results.

GetGoogleSearchSoap()

Now, return to the SearchForm unit, and add the generated index.h header file to this unit (with *File | Include Unit Hdr*), so we can access the GetGoogleSearchSoap() to return the GoogleSearchSoap interface and its methods. But before we can call this function, let's first finish the Google Search Form itself. For this, we need a TEdit, a TMemo and a TButton components. Call them edtSearch, memResult and btnGoogle. Clear the contents of edtSearch and edtMemo, and give it a nice caption, as can be seen in the following screenshot:



The OnClick event handler of the btnGoogle will call the search() method of the GoogleSearchSoap interface, passing the content of edtSearch as argument, and placing the result in memResult. All this is all done in the following one-liner:

```
void __fastcall TForm1::btnGoogleClick(TObject
                                    *Sender)
{
  memResult->Lines->Text =
 GetGoogleSearchSoap()->search(edtSearch->Text);
}
```

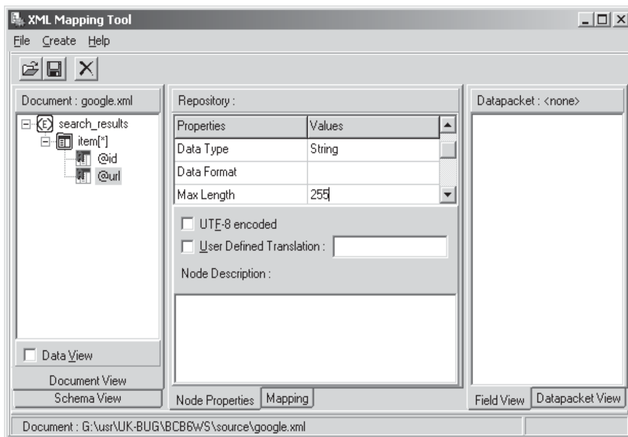
Using this event handler, we can search for the words Borland, User and Group. This combination results in www.richplum.co.uk as the first entry (and the second one, by the way). This result that satisfies me, as the new webmaster of the UK-BUG website! The result comes in rather a new way, however: as an XML document, formatted as follows:

```
<search_results>
<item id="1" url="http://www.richplum.co.uk/" />
<item id="2" url="http://www.richplum.co.uk/
                html/books.asp" />
<item id="3" url="http://www.borland.com/
                programs/usergroups/" />
<item id="4" url="http://www.borland.com/
          programs/usergroups/uglist.html" />
<item id="5" url="http://homepages.borland.com/
                            oregondug/" />
<item id="6" url="http://linux.org.mt/" />
<item id="7" url="http://www.linux.org.tr/" />
<item id="8" url="http://www.compinfo-
                      center.com/ugrp.htm" />
<item id="9" url="http://www.ugvendors.com/
                      companies/borland.htm" />
<item id="10" url="http://e-web.ecommsite.com/
                            philbug/cms/" />
</search_results>
```

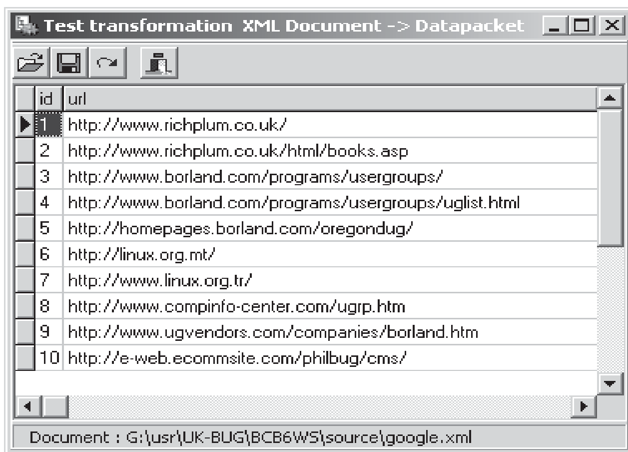
Darn. Does this mean we have to parse the entire result in order to list the URLs? Well, it would, if we didn't know or remember that C++Builder 6 Enterprise comes with a nice tool called the XML Mapper, which is able to map any XML document to a dataset that can be loaded by a ClientDataSet. So, let's save the above example in file google.xml and start the XML Mapper to see how hard it can be to transform this into something more useful.

XML Mapper

The XML Mapper can be found in the Tools menu of C++Builder, but also as a separate icon in the C++Builder group. Both point to the same executable, obviously. Start the XML Mapper, and load the google.xml file, which displays the structure of this rather flat XML file in the left pane of the XML Mapper. The search_results consists of one node that has two attributes: id and url. One important thing to note is the Max Length value of the url attribute, which is set to 54 (based on the sample data in the XML document). This may be far too small to contain some of the longer URLs, so let's change this to 255 just in case:



Now, right-click on the search_results node and do "Select All" in order to select all nodes in the XML document (as well as all attributes). Right-click again and this time do "Create Datapacket from XML", which shows the generated and corresponding datapacket in the right pane of the XML Mapper. We now need a final step: a click on the "Create and Test Transformation" button in order to actually create (and see) the transformation results.



After you've verified that the transformation results look like mine, close the test transformation dialog again, and save the transformation information using *File | Save - Transformation*. Use the suggested filename ToDp.xtr and then close the XML Mapper, since we were only interested in the transformation information anyway.

XML Transformation

Now, get back to the SearchForm, and remove the memResult. Instead, drop in a XMLTransformProvider component as well as a ClientDataSet, DataSource and DBGrid. The XMLTransformProvider will transform the generated XML document into a data packet, which can be used by the ClientDataSet to display inside the DBGrid component. Make sure the DataSource property of the DBGrid component points to the DataSource component, and the DataSet property of the DataSource component points to the ClientDataSet component. The ClientDataSet component should set its ProviderName property to the XMLTransformProvider (you can select this one from the drop-down combobox in the Object Inspector).

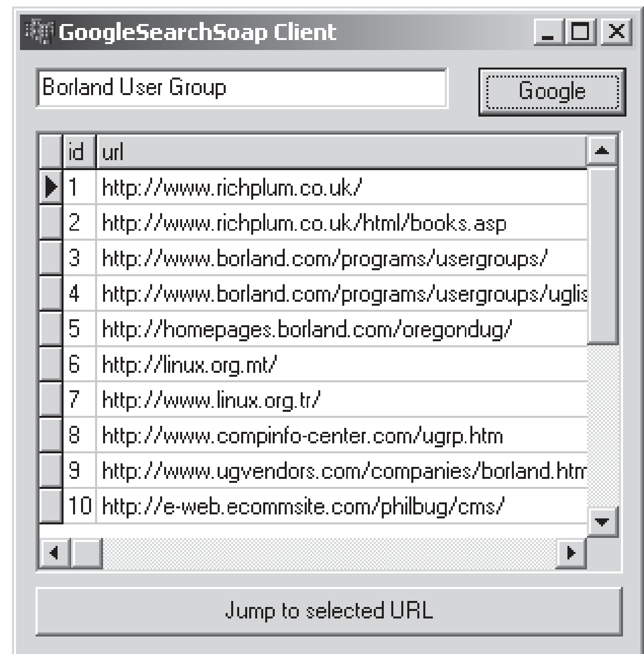
Finally, the XMLTransformProvider should get two properties assigned. The first one is the TransformationFile sub-property of the TransformRead property, which should point to the ToDp.xrt file that we generated using the XML Mapper. The second property that we must give a value is the XMLDataFile property. This one must point to the XML file that was received by the call to GoogleSearchSoap. In our example case, we can point it to google.xml.

However, this also means that we must change the call to search() and in fact make sure we save the resulting XML document in the google.xml document. Once we re-open the ClientDataSet, the XML document will be transformed again, and provided by the XML TransformProvider component. In code, the new OnClick event handler is as follows:

```
void __fastcall TForm1::btnGoogleClick(TObject
                            *Sender)
{
  ClientDataSet1->Close();
  XMLTransformProvider1->CacheData = false;
  TStringList* S = new TStringList();
  try
  {
    S->Add(GetGoogleSearchSoap()-
            >search(edtSearch->Text));
    S->SaveToFile("google.xml");
  }
  __finally
  {
    S->Free();
  }
  ClientDataSet1->Open();
}
```

Note that I'm using a TStringList as a convenient way to load the resulting XML document and save it to disk. Using this StringList, we can also display the original XML document in a Memo or use the contents for debugging purposes.

The final application can be seen in figure 8, which also contains a button to Jump to the selected URL:



The OnClick event handler of the btnJump is as follows:

```
void __fastcall TForm1::btnJumpClick(TObject
                            *Sender)
{
  ShellExecute(0,"open",ClientDataSet1-
  >FieldByName("url")->AsString.c_str(),0,0,0);
}
```

The result is a small Windows application that can be used to enter a number of search words, return the top 10 URLs, and jump directly to one of these pages when needed. And you can integrate this feature in your own applications as well, of course. As long as an internet connection is available to talk to the GoogleSearchSoap web service (which in turn probably needs to talk to Google itself).

Conclusion

In this article, I've tried to explain how to consume Web Services with C++Builder 6. We've seen how to generate the import files based on a WSDL definition of the Web Service, and how to actually call the methods from the Web Service. We've even seen how to use the XML Mapper, required to transform the resulting XML document into a data packet.

For more information about SOAP and Web Services in C++Builder, Delphi, Kylix or JBuilder, check out my SOAP Bubbles website at http://www.drbob42.com/SOAP. Finally, it's also useful to watch Borland's progress in the area of interoperability of Web Services at http://soap-server.borland.com

---

Bob Swart (aka Dr.Bob - www.drbob42.com) is an author,

trainer and consultant who just started his own one-man company called "eBob42" in Helmond, The Netherlands. Bob, who writes his own "Delphi Clinic" training material, has spoken at Delphi and Borland Developer Conferences since 1993.

Bob is co-author of the Revolutionary Guide to Delphi 2, Delphi 4 Unleashed, C++Builder 4 Unleashed, C++Builder 5 Developer's Guide, Kylix Developer's Guide, Delphi 6 Developer's Guide and the upcoming C++Builder 6 Developer's Guide.