C++Builder WebBroker

This paper is designed to get you up to speed using the basic features of the C++Builder 4.0 WebBroker. The WebBroker consists of various Wizards and components that make it easy for you to create CGI, ISAPI and NSAPI web servers. Servers of this type provide a simple, very reliable way to create distributed applications that can be viewed in a web browser.

Because C++Builder is a Windows compiler, the executables or DLLs you learn how to create in this paper wil work only on NT or Windows 95/98 web servers. The code I'm showing you is not portable to UNIX. You can, of course, use a browser to view the HTML from your servers on any type of machine you want, including Apple, Linux or Unix clients. Only the server itself must be Windows based.

The technology shown in the paper is not particularly difficult., Nonetheless I assume the audience for this article has a basic knowledge of C++Builder and Web server technology.

The Web Broker ships with C++Builder 4.0 Enterprise edition.

As long as your server is Windows based, the actual brand name of the server you use should not be important. I use the Microsoft Internet Information Server that ships with Windows NT, but you can also use the Personal Web Server (PWS) that ships on the Windows 98 CD. PWS can also be downloaded from the Microsoft Web site. These servers support both CGI and ISAPI. The Netscape server supports both NSAPI and CGI. In the future it will also support ISAPI. Other Windows based Web servers should at least minimally support CGI, though a few also support ISAPI/NSAPI. A very simple server that can aid in debugging ISAPI applications is available at http://www.drbob42.com/.

Index

A Simple Database Example

Perhaps the best way to get started is to create a CGI application that will broadcast a table across the web. Once you have seen how easy it is to create a database application on the web, I will then go back to the basics and give you a tour of the important tools you can use when building web applications.

To get started, launch C++Builder and choose File | Close All. Select File | New | Web Server Application from the menu. You will be presented with the New Web Server Application dialog, as shown in Figure 1. From the options available, choose CGI stand alone executable.

Figure 1: The New Web Server Application dialog let's you specify whether you want to create a CGI application or an ISAPI/NSAPI application.

An ISAPI or NSAPI application is a DLL that will reside in the address space of your web server. ISAPI is a Microsoft standard; NSAPI is a Netscape standard. Delphi will automatically create a single binary file compatible with either standard.

A CGI application is a stand alone executable that will reside in its own address space. Win-CGI applications were developed to support tools such as Visual Basic. C++Builder developers will rarely have reason to create tools based on the Win-CGI standard.

In this paragraph I say a few words about the relative merits of ISAPI/NSAPI vs CGI. This is a controversial topic, which typically gives rise to very strong opinions. ISAPI and NSAPI DLLs need only be loaded once, and will then stay in the address space of your server. This means that access to them, after the first time they are loaded, is reasonably fast. Executables need to be loaded each time they run. This can result in significant overhead that you do not get from DLLs. On the other hand, DLLs can be difficult to debug, as they tend to stay in memory, making it difficult to replace a previous version of your DLL with a new one. This latter problem is why I suggest you start out by building CGI applications. One further advantage of CGI applications is that you can start them at the command prompt and pipe their output to a text file. This can provide a handy means of debugging simple CGI applications of the kind created by newcomers to the technology. If you want to step through the code of your server, then sometimes it is easier to use an ISAPI dll. Tips on debugging both CGI and ISAPI applications are available in the CBuilder online help.

Note

You will find it very easy to move from a CGI application to an ISAPI/NSAPI application. As a result, there is little or no penalty for using CGI during your debugging phase and ISAPI DLLs once you have a product you want to release.

After clicking the OK button on the New Web Server Application dialog CBuilder will automatically create a web module, its source, and a project source file. A web module is a TDataModule with a few new properties and methods added to it. These new methods and properties allow you to create database applications that can be shown in a web browser.

Note

You can help port old applications to the web by creating your own TWebModule that consists of a TDataModule with the TWebDispatcher component from the Internet page. A TDataModule with a TWebDispatcher component on it is roughly equivalent to a TWebModule.

Drop down a TTable object on the Web module, just as you would on a TDataModule. Set the DatabaseName property of the TTable object to BCDEMOS, and the TableName to the Country table. (The choice of database and table that you use is entirely arbitrary. I am simply selecting a table and Alias that should be widely

available to all users of C++Builder.)

Drop down a TDataSetTableProducer from the Internet page in the Component Palette and set its DataSet field to Table1. Double click on the TDataSetTableProducer to bring up the Response Editor dialog. You can also reach this dialog by right clicking on the TDataSetTableProducer component. Use the dialog to set the table's Border property to 1, as shown in Figure 2.

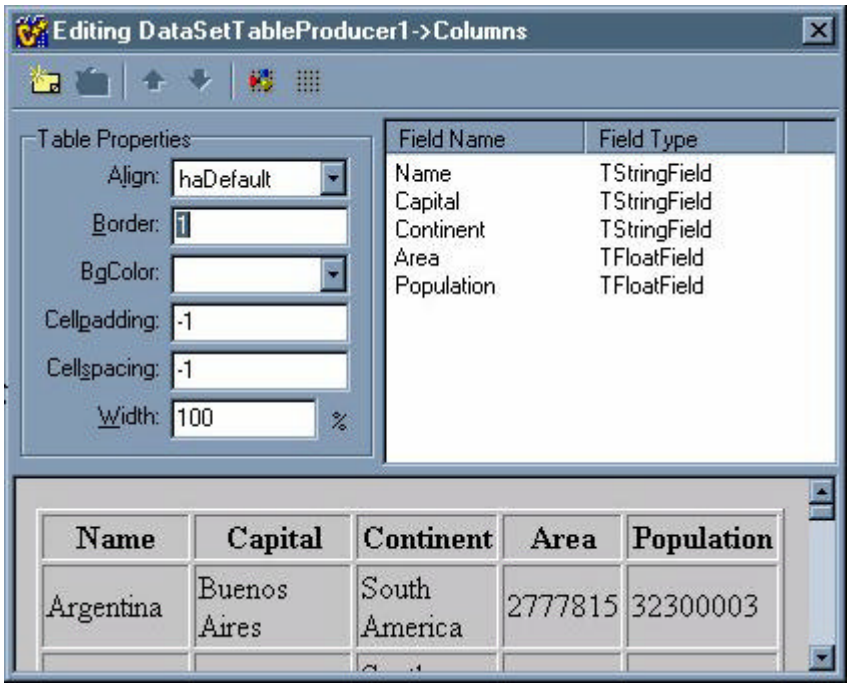


Figure 2: Using the Response Editor to set the border used when displaying your table on the web.

Right click on the TWebModule itself to bring up the Action Editor. Right click on the Action Editor and choose Add from the popup menu to create a WebAction Item, as shown in Figure 3. Set the Default property of the action you created to True.

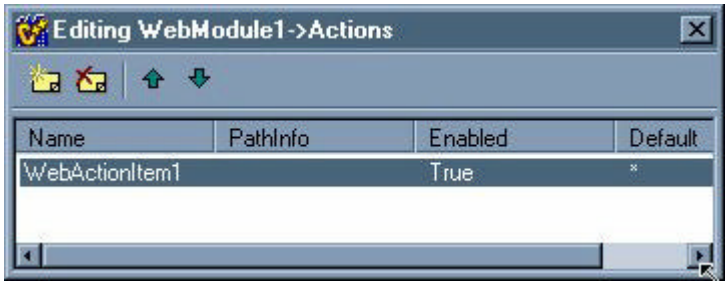


Figure 3: Working with a web action.

Click on the events page for you action, and create an OnAction event. Fill it in as follows:

```
void __fastcall TWebModule1::WebModule1WebActionItem1Action(
  TObject *Sender, TWebRequest *Request, TWebResponse *Response,
  bool &Handled)

{
  Response->Content = DataSetTableProducer1->Content();
}
```

At this point you can save your work. When I saved my example program I called the project file SimpleData.bpr and the Web module Main.cpp. In other words, I saved Unit1 as Main.cpp and Project1.bpr as SimpleData.bpr.

Many developers might find it helpful to choose Project | Options and turn to the Directories/Conditional page and fill in the Final Output field so that your application will end up somewhere that it can easily be reached by your web server. For instance, in Figure 4, I have set my program up so the executable will be sent to the c:\Inetpub\Scripts directory.

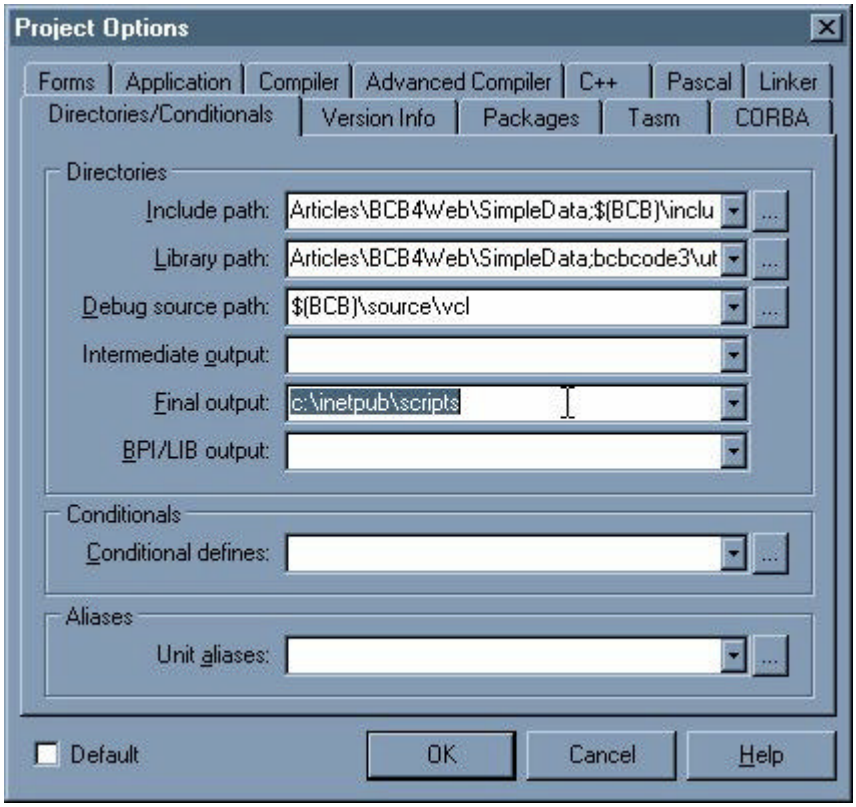


Figure 4: Configuring Delphi to store the server in the Scripts directory.

Make your project so that it compiles and links. It is probably wise to also go to the out put directory for your project and make sure that the executable is there. If you want, you can run it once from the command prompt. It should spit out a screenful or so of messy looking HTML. If you want, you can redirect this HTML into a tex file, so that you can study the output from your program. To do this, enter the following command at the DOS prompt: SimpleData.exe > Temp.txt.

Of course, you don't really want to see a messy text file full of HTML. Instead, you want to see the output of your program on the World Wide Web.

One simple way to get quick gratification is to start your web browser, and to enter the following URL, where you write not the name of my machine, which is EastFarthing, but the name of your own machine. In this example I assume you have put your executable in the scripts directory:

http://eastfarthing/Scripts/SimpleData.exe

When you are done, you should see something like the output shown in Figure 5. The output I show here is squashed down into the smallest possible space, so that the bitmap I am using in this article will download quickly to your site. Needless to say, the results are more aesthetically pleasing if you expand your browser to 1024 X 768.

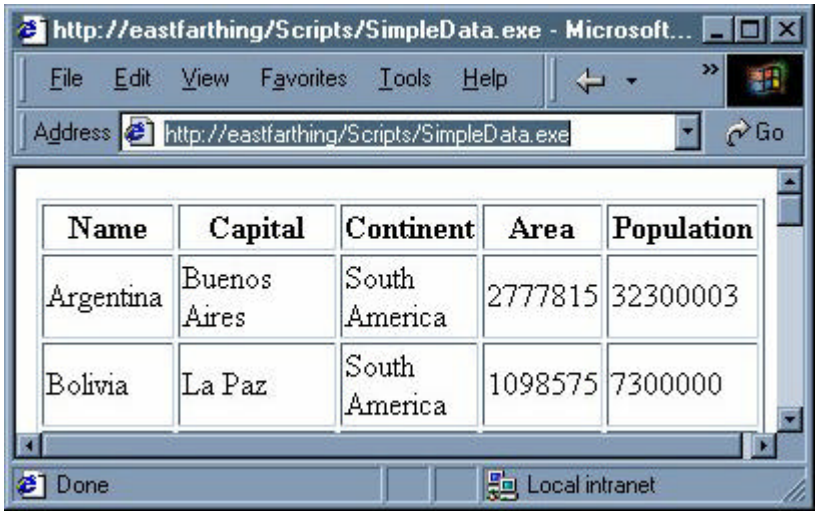


Figure 5: The output from the SimpleData program.

Changing Your CGI Application to an ISAPI DLL

In this section you will learn how to convert your CGI application to an ISAPI/NSAPI DLL.

All the code that makes your program into either an ISAPI DLL or CGI application is located in the project source. If you want, you can create a single project source file, and use IFDEFs to decide whether the application will be an ISAPI DLL or CGI application. In this case, however, I will simple create two project source files, and link them both to the same Web Module.

Select File | Close All to close your current project. From the menu, choose File | New | Web Server Application. This time, elect to build an ISAPI/NSAPI project. Once you have completed the wizard, choose Project | Remove from the Project menu and remove Unit1 from the project. Save your work into the same directory where you created the CGI application from the previous section of this paper. In this case, I would suggest saving the BPR file under the name SimpleDataISAPI.bpr.

Now choose Project | Add to Project from the BCB menu. Select the main file from the SimpleData project. You now have a new project source file which is attached to the Web module you created in the previous section of this paper.

If you want, choose, Project | Options and set the Final Output directory for this project to c:\Inetpub\Scripts, or some other location you find appropriate. When you are done, build your project and make sure the DLL is indeed in the directory you suppose it to be.

Now launch your browser, and enter the following URL, adjusting any of the details to the naming and directory conventions active on your system:

```
http://eastfarthing/Scripts/SimpleDataISAPI.dll
```

When you are done, the SimpleDataISAPI.dll should output the same information shown in Figure 5.

As you can see, C++Builder makes it easy for you to build either ISAPI/NSAPI or CGI Web server applications. In this case, I first built a CGI application, and then converted it into an ISAPI/NSAPI DLL. You can, of course, go directly to an ISAPI implementation, or you can stick with a CGI implementation. The choice is yours.

Looking Ahead

At this stage, you know how to create a simple CGI application that can display the contents of a table over the

web. If you have an NT server connected to the Web, then you can view this content from your current computer, from a computer down the hall, or from a computer on the other side of the world. The output from your program is sent to your clients using HTTP, so anyone who can get at your server can view this data, regardless of the type of machine or the type of operating system they are using.

While creating this application, you learned about TWebModules, TDataSetTableProducers, response editors, action editors, web action items, as well as ISAPI/NSAPI and CGI technologies. Now that you understand the basics of the WebBroker, the next step is to dig into this technology a bit deeper. Ironically, this will involve taking several steps back and seeing some of the fundamental principles of the WebBroker technology.

Learning the Basics

The next program I want to build will simply output a text string into your browser. This is the simplest possible form of CGI application, at least in terms of its functionality.

Choose File | Close All to close any open files, then select File | New | Web Server Application. Choose to build either an ISAPI/NSAPI or CGI application, depending on your wishes, though I suggest using CGI for the first builds of your server.

In this project, you can leave your TWebModule blank. Instead, right click on it and select the Action Editor. Create a single action called WebAction1, and use the Object Inspector to help you create the following OnAction event handler:

```
void __fastcall TWebModule1::WebModule1WebActionItem1Action(
      TObject *Sender, TWebRequest *Request, TWebResponse *Response,
      bool &Handled)

{
  Response->Content = "<P>This <B>information</B>.</P>";
}
```

Save your project as RawOutput.bpr, and save the Web module as Main.cpp. Compile your work. When you test your program, you should see a simple string in your browser, as shown in Figure 6.

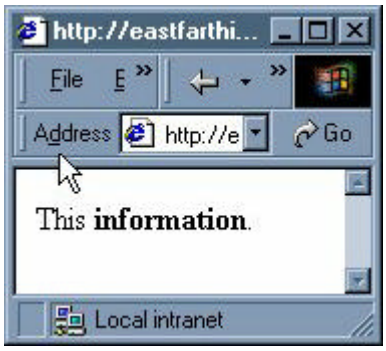


Figure 6: The simple string you see when accessing the RawOutput server.

Now that you see how the system works, it should be obvious to you that you can use this technology to output a wide variety of information across the World Wide Web. The example shown here outputs a very simple text string, but obviously you could create or compose much more complex strings, while embedding as much or as little HTML into the string as you desire.

Easy Access to Your Servers

By this time you have created three servers, and you probably want a means for accessing them as simply and easily as possible. To do this, you can create the following HTML file:

```
<HTML>
<HEAD><TITLE>MyScripts</TITLE></HEAD>
<BODY>

<TABLE BORDER = 1>
<TR>
<TD>
<A HREF="/Scripts/SimpleData.exe">Simple Data CGI Version</A>
</TD>
<TD>
<A HREF="/Scripts/SimpleDataISAPI.dll">Simple Data ISAPI Version</A>
</TD>
<TD>
<A HREF="/Scripts/RawOutput.exe">RawOutput Sample</A>
</TD>
</TR>
</TABLE>

</BODY>
</HTML>
```

If you link to this file from your home page, then you can have a relatively easy means of loading your servers. Needless to say, the paths I've hard coded into this file may well differ on your system.

Using the TPageProducer Component

The previous example gives you a sense of the flexibility of the WebBroker technology. However, the WebBroker contains a number of tools that make it easy for you to create more complex web servers than the one's I've shown you so far. In this section, you will learn about the TPageProducer component, which can greatly ease your burden when creating more complex examples of the type of non-database server shown in the RawOutput server example.

Choose File | New | Web Server Application and build either an ISAPI./NSAPI or a CGI project, depending on your tastes, predilections, etc. Turn to the Internet page in the Component Palette, and drop down a TPageProducer component on your Web module.

A TPageProducer component has two key properties called HTMLDoc and HTMLFile. HTMLDoc can contain while HTMLFile can reference, an HTML file that you want to display over the web. For instance, if you click on the HTMLDoc property editor, you could enter the following simple bit of HTML:

```
<HTML>
<HEAD><TITLE>Sample Page</TITLE></HEAD>
<BODY>

<P>Here is some text that I am displaying on the web.</P>

</BODY>
</HTML>
```

You could also place this text in a file, and point the HTMLFile property at the text.

Right click on the Web module to bring up the Action Editor. Create a default action and associate the following code with it:

```
void __fastcall TWebModule1::WebModule1WebActionItem1Action(
      TObject *Sender, TWebRequest *Request, TWebResponse *Response,
      bool &Handled)

{
  Response->Content = PageProducer1->Content();
}
```

Now compile and test your server. When you view the server in your browser, the string from your HTML file

should be displayed for the user's perusal.

When used in this manner, the TPageProducer component does not provide much more functionality than what you saw in the RawOutput example. To see the specific advantages of the TPageProducer component, you neec to learn about tags.

TPageProducer

It's finally time to start building a somewhat more complex example. Create a new example called WebOracle, and add a PageProducer to it, as you did in the previous example. This time, the HTML you add to the project should look like this:

```
<HTML>
<HEAD><TITLE>Knowledge Repository</TITLE></HEAD>
<BODY>

<H1>Knowledge Repository</H1>

<P>Welcome to the Knowlege Repository.  Our servers contain all the
known knowledge in the world, plus a large fund of additional "unknown"
information about both the past and the future.  Our database is free to
the general public.</P>

<P> If you have a question you would like to ask, type it into the
control shown below, and then push the button.</P>

<FORM ACTION="/Scripts/WebOracle.exe", METHOD=POST>
<INPUT TYPE=EDIT NAME=UserQuery>
<INPUT TYPE=SUBMIT>

</BODY>
</HTML>
```

Readers who know HTML should be able to picture the output generated by this code, but if you need help visualizing it, you can peek ahead to Figure 7, ignoring the last bit of content at the bottom of the screen shot.

The response method for your object should look like this:

```
void __fastcall TWebModule1::WebModule1WebActionItem1Action(
      TObject *Sender, TWebRequest *Request, TWebResponse *Response,
      bool &Handled)

{
  Response->Content = PageProducer1->Content();

  if (Request->Method == "POST")
    Response->Content = Response->Content + "<BR>" + Request->Content;
}
```

When you run this program, you presented with some text, and then at the bottom of the file there is an Edit control and a button. If you type something in the edit control and press the button, then you will see the text, edit control and button again, plus the text that you entered. This is shown in Figure 7.

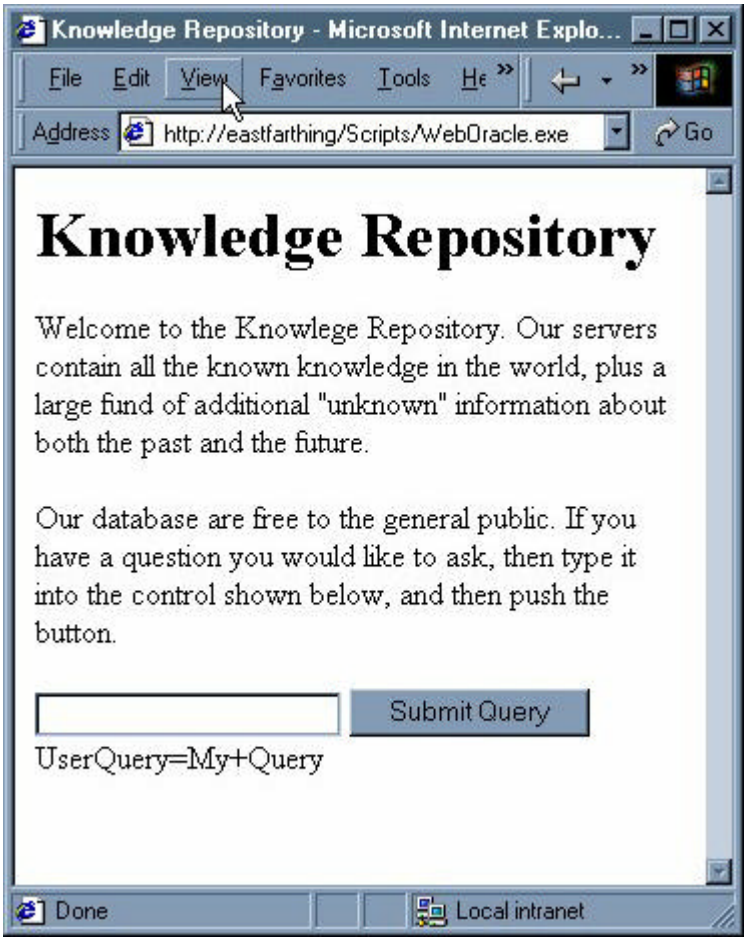


Figure 7: The text at the bottom of the picture was entered by the user, and then echoed back to him by the WebOracle server.

If the user presses the button on the HTML form, the WebModule1WebActionItem1Action method is called. In that case, the Request object passed to the method will contain the content of the information passed in to the application when you pressed the button.

Though it is still in a very nascent form, the WebOracle program already has some interesting features. In particular, it lets a client carry on a dialog, albeit a very simple one, with the server. Because of the nature of the web, this means you can communicate between computers located in two locations. In particular, one computer in England could be talking to a second computer in San Francisco, or one computer in Australia could be talking to a second computer in New York. The power of this technology is quite significant.

Responding to a Button Click

The next thing we can do with this program is have it respond in some kind of useful fashion to the button press on the HTML form. In this case, I just want to display the text the user has sent in a custom made HTML file. This means I have two HTML files I want to work with. The first file you have already seen, and it is attached to PageProducer1. The second file I can attach to a second TPageProducer that I have called FormProducer:

```
<HTML>
<BODY>

<P>You asked: <#UserQuery></P>

<P>This question is posed in an incorrect format. <B>Query
Aborted!</B></P>

</BODY>
</HTML>
```

Our goal is to write code that will properly respond to a user's questions. For instance, if the user asks, "What is the secret of the universe?," then we want to respond by generating an HTML page that looks like this:

```
You asked: "What is the secret of the universe?"
This question is posed in an incorrect format. Query Aborted.
```

It's easy enough to see how to generate the second of these two sentences, but it is going to take me a few paragraphs to explain how to generate the first sentence. This explanation will contain descriptions of how to work with the PathInfo property and how to work with tags. A tag is the little bit of HTML that looks like this: <#UserQuery>. But I'm going to ask you to push the subject of tags onto your stack for a bit, and to focus first on the PathInfo property.

To understand the PathInfo property, you need to recall that the button and edit control on our original HTML form were declared with this code:

```
<FORM ACTION="/Scripts/WebOracle.exe/FormInfo", METHOD=POST>
<INPUT TYPE=EDIT NAME=UserQuery>
<INPUT TYPE=SUBMIT>
```

Here is the way the key line in this example used to look:

```
<FORM ACTION="/Scripts/WebOracle.exe", METHOD=POST>
```

Here is the way this line looks now that we have edited it to include the /FormInfo URI:

```
<FORM ACTION="/Scripts/WebOracle.exe/FormInfo", METHOD=POST>
```

When the user clicks the HTML button, this code will cause the WebOracle program to be called with a URI, or PathInfo, of /FormInfo. You can explicitly respond to this URI by setting up a new WebActionItem which I decided to call FormInfo that has its PathInfo set to /FormInfo. To get started, right click on the TWebModule, and create a new WebActionItem. When you finish editing it, the Object Inspector for your new Action Item should look something like the one shown in Figure 8.

Figure 8: This WebActionItem has its PathInfo set to /FormInfo.

Now create an OnAction event for this WebActionItem. Here is how to fill out the FormInfo OnAction event handler:

```
void __fastcall TMod1::Mod1FormInfoAction(TObject *Sender,
      TWebRequest *Request, TWebResponse *Response, bool &Handled)

{
  Response->Content = FormProducer->Content();
}
```

As you can see there is nothing unusual about this code. In short, it does nothing to explain how I can generate a sentence which will mirror back the user's question to him.

To solve this problem, you need to understand that each page producer has a property called OnHTMLTag. When this method gets called, you can substitute text that you want to use in place of the tag in your HTML. In particular, I can substitute the user's question for the tag in the HTML that looks like this: <#UserQuery>. Here is the OnHTMLTag handler that makes the substitution for us:

```
void __fastcall TMod1::FormProducerHTMLTag(TObject *Sender,
      TTag Tag, const AnsiString TagString, TStrings *TagParams,
      AnsiString &ReplaceText)

{
  AnsiString S;

  S = Request->ContentFields->Values[Request->ContentFields->Names[0]];
  ReplaceText = "<B>" + S + "</B>";
}
```

In this code, I set the ReplaceText property to the value of a new string. This value will automatically be pasted into my HTML in place of the Tag.

The Request object, which plays a key role in the FormPropucerHTMLTag method, is global to the TWebModule. It has a property called Content, which contains the request being passed to us. Given the example currently being described, the Content field will look something like this: "UserQuery=What+is+the+secret+of+the+universe%3F." Clearly this is close to what we want, but we would have to parse it some before displaying it to the user.

Note

> *The Content field looks as it does because PageProducer 1 contains a line of HTML that sets the EDIT NAME to UserQuery. If the user types "What is the secret of universe?" into the HTML edit control, then he is, in effect, setting UserQuery equal to the string "What is the secret of the Universe?" In other words, the edit control named "UserQuery" is now equal to the string.*

Fortunately for the beleagured programmers of the world, the Request property has a property called ContentFields. The ContentFields property contains the parsed version of your text. In short, it converts the string "What+is+the+secret+of+the+universe%3F" into the string "What is the secret of the universe?" You can access the value the user typed in by writing the following code:

```
S = Request->ContentFields->Values["UserQuery"];
```

After making this call, S will contain the string the user typed into the HTML edit control. In the case we are currently describing, that string looks like this: "What is the secret of the universe?" Hallelujah!

The code I show here can be improved slightly by taking advantage of a property of the ContentFields object called the Names array. This is a string array with the name of each parameter passed to your application embedded in it. In this case, Names[0] will contain the string "UserQuery". This allows us to write the following, very reusable, line:

```
S = Request->ContentFields->Values[Request->ContentFields->Names[0]];
```

This code is better than the previous example, because it does not have the word "UserQuery" hard coded into it.

As you recall, PageProducer1 had a line of HTML in it that looks like this:

```
<INPUT TYPE=EDIT NAME=UserQuery>
```

You might change it to look like this:

```
<INPUT TYPE=EDIT NAME=UserQuestion>
```

Making this change would break code that looks like this:

```
S = Request->ContentFields->Values["UserQuery"];
```

The change would not break the following code, because the Names field would track the changes to your HTML:

```
S = Request->ContentFields->Values[Request->ContentFields->Names[0]];
```

In other words, Names[0] would now contain the string UserQuestion rather than UserQuery.

> *When contemplating these issues, take care not to confuse the PathInfo, which is set to /FormInfo, with the content string "UserQuery." Furthermore, note that I (perhaps somewhat confusingly), set the tag name to <#UserQuery>. So far, the code I have written would work just as well if the tag looked like this: <#UserFoo>. In other words, I haven't yet made any reference to the string in the tag itself. I will show you how to do that just a little later in this paper.*

Taking Stock

At this stage you are beginning to understand a good deal about the powerful WebBroker technology. However I readily concede that you need to assimulate quite a bit of information before you are ready to take advantage of this tool. To help increase your understanding, I want to now take a side trip into two subjects:

- The first will be a more detailed look at tags.
- The second will be an examination of HTML queries.

To aid in the exploration of these subjects, I have modified the HTML associated with PageProducer1 so that it looks like this:

```
<HTML>
<HEAD><TITLE>Knowledge Repository</TITLE></HEAD>
<BODY>

<H1>Knowledge Repository</H1>

<P>Welcome to the Knowlege Repository. Our servers contain all the known
knowledge in the world, plus a large fund of additional "unknown"
information about both the past and the future.</P>

<P>Our database are free to the general public. If you have a question
you would like to ask, then type it into the control shown below, and
then push the button.</P>

<FORM ACTION="/Scripts/WebOracle.exe/FormInfo", METHOD=POST>
<INPUT TYPE=EDIT NAME=UserQuery>
<INPUT TYPE=SUBMIT>

<HR>

<P>Here are some links that test various features of this Web Server</P>

<A HREF="/Scripts/WebOracle.exe/TagInfo">Tag Test</A><BR>
<A HREF="/Scripts/WebOracle.exe/UserInfo?Sammy=3&Frank=5">Query Test</A>

</BODY>
</HTML>
```

This code differs from the original version in that it has two hyperlinks at the very bottom that will call your

form with a URI of TagInfo and UserInfo. As I'm sure you can imagine, my next step is to set up WebAction events for each of these URIs. The first example will demonstrate how to handle tags, and the second will show how to handle a query.

Understanding Tags

In this section, you will get an in depth look at working with Tags.

To show you how to work with tags, I have added a TPageProducer which I called TagProducer to the Web module of the WebOracle program. I have set its HTMLDoc property to the following bit of HTML:

```
<HTML>
<HEAD><TITLE>Visual Slick Edit</TITLE></HEAD>
<BODY>

<P>This is a link: <#LINK NAME=MyLink></P>
<P>This is an image: <#IMAGE NAME=MyImage></P>
<P>This is custom: <#MyCustom NAME=Sunny, TYPE=FoolishLove,
  WAY=Crooked></P>

</BODY>
</HTML>
```

Note that this simple HTML file has three tags in it. The first is a LINK tag, the second an IMAGE tag and the third a CUSTOM tag. It happens that Delphi allows you to pass zero or more parameters to each tag, and you can name each parameter as you choose. In most cases, you will pass in zero parameters. But in this case I have passed in one parameter to the first two tags, and three parameters (NAME, TYPE and WAY) to the third tag.

To work with the TagProducer and its HTML, you need to set up a WebAction that has its PathInfo set to /TagInfo. Its OnAction event should look like this:

```
void __fastcall TMod1::Mod1TagInfoAction(TObject *Sender,
  TWebRequest *Request, TWebResponse *Response, bool &Handled)

{
  AnsiString S = "This example shows how to respond to tags. "
    "For more information, view the TagProducer on the WebModule.";


  Response->Content = S + "<HR>" + TagProducer->Content();
}
```

This code will mirror back the HTML in TagProducer. However, you know that you must respond to the OnHTMLTag event if you want to properly handle the tags embedded in the HTML. When you first create the event by clicking on the OnHTMLTag property of TagProducer, the code generated will look like this:

```
void __fastcall TMod1::TagProducerHTMLTag(TObject *Sender, TTag Tag,
      const AnsiString TagString, TStrings *TagParams,
      AnsiString &ReplaceText)

{
}
```

I want to spend some time looking at the parameters passed to the OnHTMLTag event. The first is of type TTag, and it designates the kind of tag that you are currently being asked to replace. Here are the possible kinds:

```
enum TTag { tgCustom, tgLink, tgImage, tgTable,
  tgImageMap, tgObject, tgEmbed };
```

In my experience, most of the tags you create will be of type tgCustom. For instance, the <#UserQuery> tag from the previous example is a custom tag. If you want to create a different kind of tag, you can use this format <#Link Name=MyTag> This tag is of type Link and has a parameter called Name set to the value MyTag. Here is a simple Image tag with no parameters: <#IMAGE>.

Here is an example of how to work with tags:

```
AnsiString __fastcall GetParamInfo(TStrings *TagParams)
{
  AnsiString S;
  for (int i = 0; i < TagParams->Count; i++)
  {
    S = S + "Param: " + TagParams->Strings[i] + "<BR>";
    S = S + "Name: " + TagParams->Names[i] + "<BR>";
    S = S + "Value: " + TagParams->Values[TagParams->Names[i]] +
      "<BR>";
  }
  return S;
}

void __fastcall TWebModule1::TagProducerHTMLTag(TObject *Sender,
  TTag Tag, const AnsiString TagString, TStrings *TagParams,
  AnsiString &ReplaceText)

{
  switch (Tag)
  {
    case tgLink:
      ReplaceText = TagString + "<BR>" + GetParamInfo(TagParams);
      break;

    case tgImage:
      ReplaceText = TagString + "<BR>" + GetParamInfo(TagParams);
      break;

    case tgCustom:
      ReplaceText = TagString + "<BR>" + GetParamInfo(TagParams);
    default:
      ReplaceText = "Unknown Tag";
  }
}
```

If you are passing in zero parameters to a tag, then you can always get the tag name by checking the value of the TagString. For instance, if you created this tag: <#MyTag>, then the TagString parameter to the OnHTMLTag event will be set to *MyTag*.

If you are passing in parameters to a tag, then you can check the TagParams property to get the values of the parameters that you passed in. The TagParams->Strings[n] property gives you the entire parameter. For instance, if you create this tag: <#LINK NAME=MyLink>, then TagParams->Strings[0] is equal to NAME=MyLink. To parse this parameter, use the TagParams->Names[n] and TagParams->Values[n] properties.

The following code will display all the parameters passed in the TagParams parameter:

```
  AnsiString S;
  for (int i = 0; i < TagParams->Count; i++)
  {
    S = S + "Param: " + TagParams->Strings[i] + "<BR>";
    S = S + "Name: " + TagParams->Names[i] + "<BR>";
    S = S + "Value: " + TagParams->Values[TagParams->Names[i]] + "<BR>";
  }
```

For the following tag: <#LINK NAME=MyLink>, the output from this code would look something like this:

```
Param: NAME=MyLink
Name: NAME
Value: MyLink
```

Rather than trying to explain how this output is produced, I will ask you to compare the code and the output. A quick perusal of these two elements should tell you everything you need to know.

You can test which kind of link is being passed in to an OnHTMLTag event, but using the TTag enumeration:

```
  switch (Tag)
  {
    case tgLink:
      ReplaceText = TagString + "<BR>" + GetParamInfo(TagParams);
      break;

    case tgImage:
      ReplaceText = TagString + "<BR>" + GetParamInfo(TagParams);
      break;

    case tgCustom:
      ReplaceText = TagString + "<BR>" + GetParamInfo(TagParams);
    default:
      ReplaceText = "Unknown Tag";
  }
```

Given the following HTML:

```
<P>This is a link: <#LINK NAME=MyLink></P>
<P>This is an image: <#IMAGE NAME=MyImage></P>
<P>This is custom: <#MyCustom NAME=Sunny, TYPE=FoolishLove,
  WAY=Crooked></P>
```

The LINK tag would be caught by the tgLink part of the switch statement, the IMAGE tag by tgImage, and the last option by tgCustom. Obviously it would make sense to use <A HREF> code with tgLink and <IMG MYFILE.JPG> with tgImage, etc. The purpose the TTag property is clear, but there is nothing that forces you to use a particular tag with a particule HTML tag. Associating HTML image tags with the tgImage identifier is just a convention.

In my code, I show you how the tag technology works, but in your code you might want to pay closer attention to the implied conventions. Assuming you followed the conventions, in big HTML files, you might have many tgImage or tgLink tags. To distinguish one for the other, you might want to call your first paramter NAME, and assign a unique name to each tgLink tag that you create. For instance, if you had many images with pictures of people in it, you could write code like this:

```
<#IMAGE Name="Suzy">
<#IMAGE Name="Fred">
<#IMAGE Name="Lisa">
```

Now that you understand how tags work, I can show you the output from the TagInfo URI code used in the WebOracle program:

```
This is a link: LINK
Param: NAME=MyLink
Name: NAME
Value: MyLink

This is an image: IMAGE
Param: NAME=MyImage
Name: NAME
Value: MyImage

This is custom: MyCustom
Param: NAME=Sunny,
Name: NAME
Value: Sunny,
Param: TYPE=FoolishLove,
Name: TYPE
Value: FoolishLove,
Param: WAY=Crooked
Name: WAY
Value: Crooked
```

Compare this output with the original HTML file in TagProducer. If you study the output, the original HTML, and the code in the OnHTMLTag event, then you should be able to garner enough information to thoroughly understand WebBroker Tags.

Working with Queries

Consider the following HTML:

```
<A HREF="/Scripts/WebOracle.exe/UserInfo?Sammy=3&Frank=5">Query Test</A>
```

This HTML link has a PathInfo of /UserInfo. If you create a WebActionItem with its PathInfo set to /UserInfo, then it will be called when the user clicks on this link.

As you can see, in this HTML the URI has a question mark followed by the following code: Sammy=3&Frank=5. These are queries appended to the PathInfo.

What I am generating here is something called a Fat URL, which is usually whimsically pronounced "Fat Earl." A Fat URL can contain all kinds of information. It can be used to maintain state between calls from a client, or it can be used to contain information the user filled out in a form.

Here is how to parse the queries appended onto the PathInfo:

```
void __fastcall TMod1::Mod1UserInfoAction(TObject *Sender,
  TWebRequest *Request, TWebResponse *Response, bool &Handled)

{
  AnsiString S;

  S = "Method: " + Request->Method;
  S = S + "<BR>Query: " + Request->Query;
  for (int i = 0; i < Request->QueryFields->Count; i++)
  {
    S = S + "<BR>Field: " + Request->QueryFields->Names[i];
    S = S + "<BR>Value: " +
      Request->QueryFields->Values[Request->QueryFields->Names[i]];
  }

  Response->Content = S;
}
```

As you recall, the Request object has ContentFields property which you used to return the question the user typed into an HTML edit control. The Request object also has a QueryFields property, and it can be used to

parse the information in a Query.

If you want to see the query the user passed in, then use the Request->Query property. In this case, the Query field would contain the following string: "Sammy=3&Frank=5."

This is fine as far as it goes, but you obviously want to parse this information so that it is more readily comprehensible. To parse this query, use the QueryFields property:

```
S =  Request->QueryFields->Values[Request->QueryFields->Names[i]];
```

This code will retrieve the value of each query. For instance, QueryFields->Values["Sammy"] will return 3, and QueryFields->Values["Frank"] will return 5. Needless to say, QueryFields->Names[0] will return the string Sammy, and QueryFields->Names[1] will return "Frank". Please note that QueryFields->Values[0] is code that doesn't get you anything of value! (I say this to you here, but nevertheless, if you are as human as I am, then you will still end up trying to write that code and wondering why it doesn't work. It's an easy mistake to make. Drill it in to your head: don't write QueryFields->Values[0], instead, write QueryFields->Values["Some Valid String"].

This is the end of the section on tags and queries. Indeed, this is the end of the section on PageProducers. In the next section, I will start bring the article around full circle to where we began by returning to the subject of databases.

More on Working with DataBases

In this section I'll show you how to display and search for a single record from a TQuery object, as shown in Figure 9.

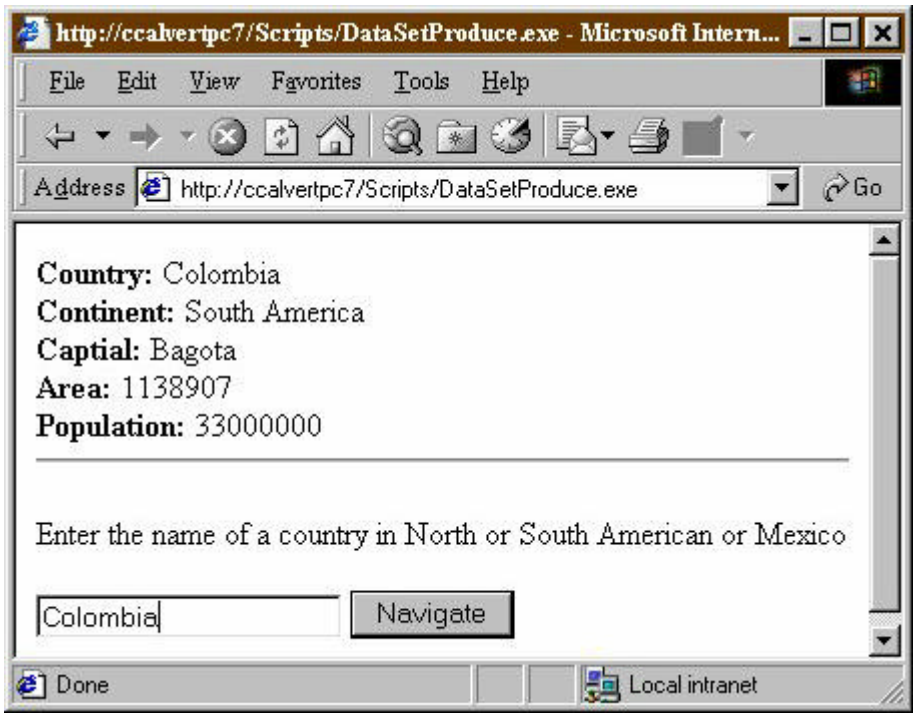


Figure 9: Displaying and searching for a single record from a dataset.

To get started, drop down a TDataSetPageProducer on a TWebModule. Don't confuse the TDataSetPageProducer with the TPageProducer, TQueryTableProducer or TDataSetTableProducer.

A TDataSetPageProducer is a lot like a TPageProducer, except its tags are replaced with fields from a TTable or TQuery object. Consider the Country table from the BCDEMOS alias which has a field called Name. If you write the following HTML, then the TDataSetPageProducer will place the current value of the Name field in the <#Name> tag:

```
<B>Country:</B> <#Name><BR>
```

Your goal as a programmer is to create tags that have the same name as a field of the table. Then, if you write the following simple OnAction handler, the tag will be automatically replaced with the current value of the field:

```
void __fastcall TWebModule1::WebModule1WebActionItem1Action(
  TObject *Sender, TWebRequest *Request, TWebResponse *Response,
  bool &Handled)

{
  Response->Content = DataSetPageProducer1->Content();
}
```

Obviously this is a very easy technology to use.

Once you understand the basics of using this technology, you might want to create an example that will allow the user to type in a value, and then have the Web server display the record associated with that value. For instance, you could ask the user to type in the name of a movie, and then show him a review of the movie, or the name of a person, and then show that person's address.

Consider the following HTML, which is designed to support a program like the one described in the previous paragraph:

```
<HTML>
<BODY>

<B>Country:</B> <#Name><BR>
<B>Continent:</B> <#Continent><BR>
<B>Captial:</B> <#Capital><BR>
<B>Area:</B> <#Area><BR>
<B>Population:</B> <#Population><BR>
<HR>

<FORM ACTION="/Scripts/DataSetProduce.exe" METHOD=POST>
<P>Enter the name of a country in North or South America.
  (Mexico also supported.)</P>
<INPUT TYPE=EDIT NAME=EDIT1>
<INPUT TYPE=SUBMIT NAME=NAVIGATOR VALUE="Navigate">

</BODY>
</HTML>
```

This HTML file works with the Country table from the BCDEMOS alias, and appears to the user like the screen shot shown in Figure 9. Notice that I create tags for each field of the Country table. At the bottom of the file a form allows the user to enter the name of a country. If the user presses the button on the form, then the code attempts to display data about the country the user entered. The program includes no error handling for cases when the user types in an invalid name. You can, however, type in only one or more letters of the name o a country, and then see if that value is unique enough to help you view your country.

Here is the default OnAction event for handling the button click:

```
void __fastcall TWebModule1::WebModule1WebActionItem1Action(
  TObject *Sender, TWebRequest *Request, TWebResponse *Response,
  bool &Handled)

{
  if (Request->ContentFields->Count > 0)
  {
    AnsiString S = Request->ContentFields->Values["Navigator"];
    if (S == "Navigate")
    {
      S = Request->ContentFields->Values["EDIT1"];
      char SQL[250];
      sprintf(SQL, "Select * from Country where Name like \"%s\"", S.c_str());
      CountryQuery->Close();
      CountryQuery->SQL->Clear();
      CountryQuery->SQL->Add(AnsiString(SQL));
      CountryQuery->Open();
    }
  }
  Response->Content = DataSetPageProducer1->Content();
}
```

This code uses the ContentFields property to make sure that we are responding to a click on the correct button. In this case, the code is overkill, but I think it is interesting to see that you can write code that distinguishes one button on a form from another. This could be useful if you are creating forms that have, for instance, Ok, Cancel, and Abort buttons on them.

Assuming that the user has clicked the Navigate button, the code shown here retrieves the string entered by the user. It then uses the string to create a valid SQL statement of the form Select * from X where Y = Z. The code then closes the table, and opens it again on the new SQL statement. At this point, all that remains to be done is return the values of the new SQL statement.

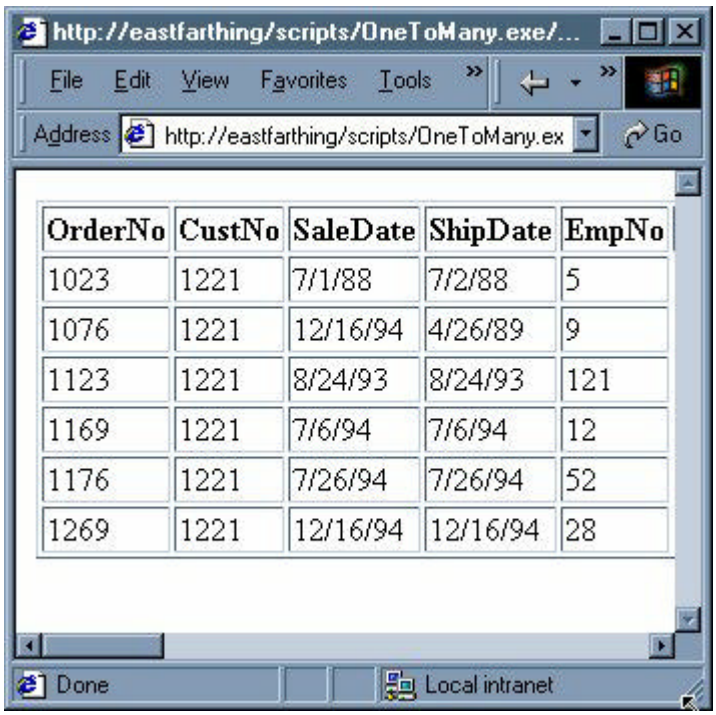
If the SQL statement returned multiple rows, only the first row will be shown. If no rows are selected, then I simply return a form with captions but no data in it. A more complex program might attempt to handle these problems with more aplomb.

TQueryTableProducer: A One To Many

In this example, you will see how to create a one to many that can be displayed over the web. The name of the server you create will be called OneToMany.exe or OneToMany.dll, depending on whether you create an ISAPI or CGI application. As usual, I suggest you start by working with a CGI application.

When the user first accesses the program, he will see a list of names of companies, as shown in Figure 10. If he clicks on any of the names, he will see details about transactions undertaken by the company, as shown in Figure 11. Needless to say, the transactions are kept in a second table that is connected to the first table in a master/detail relationship.

Figure 10: The default WebActionItem for the OneToMany program allows the user to click on the name of company to see details about its transactions.



OrderNo	CustNo	SaleDate	ShipDate	EmpNo
1023	1221	7/1/88	7/2/88	5
1076	1221	12/16/94	4/26/89	9
1123	1221	8/24/93	8/24/93	121
1169	1221	7/6/94	7/6/94	12
1176	1221	7/26/94	7/26/94	52
1269	1221	12/16/94	12/16/94	28

Figure 11: Using the OneToMany server to view the transactions for a particular company.

In figure 11, you see that company number 1221 had six transactions. You can view the dates of the transactions in this screen shot. If you blow the screen up to 1024X768, then you can find out addional information, such as the sums of money involved in each transaction. I don't show you that big screen shot, however, because I want you to be able to download my bitmaps as quickly as possible.

To get started creating the program, drop down a TTable and TQuery. Name the TTable CustomerTable and the TQuery OrdersQuery. Set the table to access the Customer table from the BCDEMOS database. Set the OrdersQuery up to access the Orders table from BCDEMOS by writing the following simple SQL statement: Select * from Orders where CustNo = :CustNo.

To test your work, first pull up the OrdersQuery and fill in its Params property so that the DataType is ftInteger

and the ParamType is ptInput. You can now set the query to Active, thereby confirming that your SQL is valid.

To get started working with the CustomerTable, drop down a TPageProducer and name it CustomerProducer. Add the following code to the HTMLDoc property of the page producer:

```
<HTML>
<!------------------------------------------------>
<!-- Copyright 1999 by Charlie Calvert -->
<!------------------------------------------------>
<HEAD>
<TITLE>Companies</TITLE>
</HEAD>
<BODY>
<H2>Company</H2>
<HR>
Click highlight words to view Company Info.<P>
<#CompanyType><P>
</BODY>
</HTML>
```

As you can see, this code has a tag in it called <#CompanyType>. You are going to need to add a WebActionItem to the Web module that looks like this:

```
void __fastcall TWebModule1::WebModule1ShowCustomersAction(TObject *Sender,
      TWebRequest *Request, TWebResponse *Response, bool &Handled)

{
 Response->Content = CustomerProducer->Content();
}
```

As you know by now, the real work of filling in a tag is usually done not in the ActionItem itself, but in the OnHTMLTag event for the producer you are currently using, which in this case is the CustomerProducer. Here is the relevant code:

```
void __fastcall TWebModule1::CustomerProducerHTMLTag(TObject *Sender,
  TTag Tag, const AnsiString TagString, TStrings *TagParams,
  AnsiString &ReplaceText)

{
  AnsiString S, Temp, CustNo, FormatString;

  CustomerTable->Open();
  CustomerTable->First();
  while (!CustomerTable->Eof)
  {
    Temp = CustomerTable->FieldByName("Company")->AsString;
    CustNo = CustomerTable->FieldByName("CustNo")->AsString;
    FormatString =
      "<a href=\"/scripts/OneToMany.exe/Orders?CustNo=%s\">%s</a><BR>";
    S = S + Format(FormatString, OPENARRAY(TVarRec, (CustNo, Temp)));
    CustomerTable->Next();
  }
  CustomerTable->Close();
  ReplaceText = S;
}
```

This method retrieves the Company and CustNo fields from each record in the the Customer table, and inserts them in series of links that look like this:

```
<a href="/scripts/OneToMany.exe/Orders?CustNo=1221">Kauai Dive
  Shoppe</a>
<a href="/scripts/OneToMany.exe/Orders?CustNo=1231">Unisco</a>
<a href="/scripts/OneToMany.exe/Orders?CustNo=1351">Sight
  Diver</a>
```

If the user clicks on the first of these links, then the server is called with the following URL:

```
http://eastfarthing/scripts/OneToMany.exe/Orders?CustNo=1221
```

As you can see, the server is being called with a URI (PathInfo) of /Orders. You therefore need to set up a new WebActionItem that has /Orders as its PathInfo.

A query is being passed to this OnAction event that looks like this: CustNo=1221. From what you read earlier, you might think that you would use the QueryFields property of the Request object to parse this information, and then use the resulting data to compose a SQL statement that would return the data requested by the user. Indeed, you could take that approach if you wanted. It happens, however, that the VCL has a component, called TQueryTableProducer, that will parse the query for you, and that will automatically compose the requisite SQL to retrieve the correct rows from the Orders table.

To make all this work, first drop down a TQueryTableProducer on the TWebModule. Set its Query property to the OrdersQuery. Now fill in the WebActionItem with a PathInfo of /Orders so that it looks like this:

```
void __fastcall TWebModule1::WebModule1RunQueryAction(TObject *Sender,
      TWebRequest *Request, TWebResponse *Response, bool &Handled)

{
  Response->Content = QueryTableProducer1->Content();
}
```

That's all you need to do! Now when you call your server with the following URL, you will automatically get the correct rows in your detail table, as shown in Figure 11:

```
/scripts/OneToMany.exe/Orders?CustNo=1231
```

What's important here is that you have composed a Query called CustNo that is set to a valid customer number. The Query must have the right name, (ie CustNo), or this won't work. You can string together multiple fields to compose more complex queries, using the & syntax. The following examples shows how you could pass in multiple fields if you had a complex query with multiple parameters:

```
/scripts/OneToMany.exe/Orders?CustNo=1231&LastName="Smith"&FirstName="Paul"
```

If the query had parameters called CustNo, FirstName and LastName, then this kind of URL would be appropriate.

That's all I'm going to say about databases and queries. Hopefully the examples provided here gives you enough information to start creating your own database applications.

Summary

In this paper you have had a tour of the Borland WebBroker technology. The WebBroker is designed to supply you with components that automatically perform common tasks encountered in this kind of development.

The examples shown in this paper demonstrate how to easily access one or more tables from a database, how to parse queries passed in a URL, how to respond to button clicks on HTML forms, and how to carry on a dialog between a web client and your server.

Back To Top
Home Page