

ASP Development

by Bob Swart

CHAPTER

22

IN THIS CHAPTER

- Understanding Active Server Objects 1050
- The Active Server Object Wizard 1052
- ASP Session, Server, and Application Objects 1065
- Active Server Objects and Databases 1066
- ASP Objects and NetCLX Support 1069
- Debugging Active Server Objects 1071

In this chapter, you will learn what Active Server Pages and Active Server Objects are, and how Delphi 6 can support you when creating and deploying Active Server Objects.

Understanding Active Server Objects

Like *CGI (common gateway interface)* and *ISAPI/NSAPI (Internet Server API/Netscape Server API)* extensions, as supported by WebBroker, *ASP (Active Server Pages)* is a server-side Web application solution. This means that you can put Active Server Pages and Active Server Objects on a Web server to let clients connect to the Web server and load the pages and objects. This chapter focuses mainly on Active Server Objects written in Delphi 6 but created and used within Active Server Pages.

Delphi 5 introduced a new wizard that enables you to create Active Server Objects. These Active Server Objects can be used in ASP to dynamically generate HTML code every time the server loads the page. This chapter explains what Active Server Page Objects are, how they are related to CGI, ISAPI, and COM, and how they can be used in the context of Active Server Pages. Further, we will focus on different aspects that are important when creating Active Server Objects. Active Server Objects are server-side components, and differences between operating systems (like Windows NT version 4 and Windows 2000) as well as differences between Internet Information (Web) Servers (IIS 3 and 4 compared to IIS 5) will also affect the way we deal with Active Server Objects.

As an example, we will generate a simple Active Server Object and a template script and then adjust the object and the script for our own needs by adding some methods. Then we will install and register the object on the Web server. Finally, we will examine how to deploy new versions of Active Server Objects and how to test and debug them.

Active Server Pages

Before we start creating our own Active Server Objects, I want to give you an introduction into the technology and syntax of Active Server Pages, which will be the operating environment for our Active Server Objects. Active Server Pages enable you to use a scripting language that is interpreted by the Web server—and not the Web browser. This means that you must have a Web server installed to be able to test the source code listings and examples in this chapter. We've used Microsoft *Internet Information Server (IIS)* version 4 on Windows NT 4 as well as IIS 5 on Windows 2000, but *Personal Web Server (PWS)* on Windows 95 or 98 works just as well. Whereas normal HTML pages have the .htm or .html extension, ASP pages have an .asp extension. In order for the Web server to execute ASP pages, you must place them in a directory that has scripting rights enabled. In your default installation of any of the Microsoft Web servers, you'll have a Scripts directory. But even if you don't have Scripts, it's easy to create a new virtual directory with scripting rights. On Windows NT, start the Internet Service Manager

(Microsoft Management Console), go to your Web service, add a new virtual directory—for example a directory called Scripts or cgi-bin—and make sure that the Scripting option is enabled.

You can change the server-side scripts without having to compile them or restart the Web server. The scripting statements are written between `<%` and `%>` tags, and the Active Scripting language is based on JavaScript and VBScript, which isn't hard to learn or understand.

As special support, Active Server Pages get a number of built-in objects that they can use to communicate with the browser and server environment.

The two most useful objects are as follows:

- **Request**—Implemented for user input. The Request object can access form input variables and check their values.
- **Response**—Used to generate user output. The Response object has a write method that can be used to generate HTML output.

As a little ASP scripting example, the following script will check the HTML input variable Name, and if the entered value is Bob, Response will write "Hello, Bob!"; otherwise Response will simply write "Hello, User!":

```
<%  
  if Request("Name") = "Bob" then  
    Response.Write("Hello, Bob!")  
  else  
    Response.Write("Hello, User!")  
  end if  
>%
```

If this ASP code is contained in a page called `test.asp`, the following HTML form can be used to trigger it:

```
<FORM ACTION="test.asp" METHOD=POST>  
Name: <INPUT TYPE=text NAME=Name>  
<P>  
<INPUT TYPE=submit>  
</FORM>
```

NOTE

The input variable called Name can be queried using the ASP Request variable.

Remember that Active Server Pages can only be executed (interpreted) by the Web server if they are actually served by the Web server. This means that the URL used to view them must activate the Web server. So a file `test.asp` in the `\cgi-bin` directory shouldn't be activated as `file:///d:/www/cgi-bin/test.asp` because that won't involve the Web server and will just show the file itself with the ASP source intact. However, the URL `http://localhost/cgi-bin/test.asp` will activate the Web server (for a local machine), and the result will be the executed output from the Active Server Page.

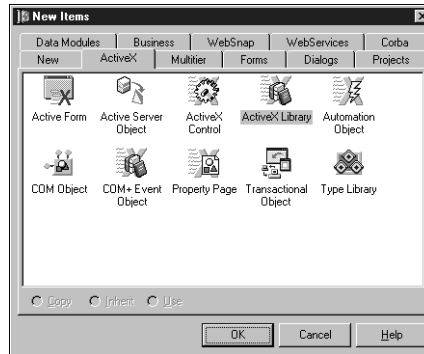
On the surface, this might seem simple and easy to use. As a Delphi developer however, you wouldn't want to write your entire Internet Web application using server-side ASP scripting. Consider the performance issue when it comes to interpreting ASP scripts that aren't compiled. ASP counts among its advantages the capacity to change its scripts on-the-fly as opposed to recompiling and redeploying them. However, because sites have grown larger and more complex, this advantage is far outweighed by the performance deficits introduced by the use of an interpreter. Fortunately, using the ASP scripting language, you can create and use special Active Server COM Objects that reside on the server. These objects are compiled binaries; therefore, they are faster and more efficient. This is where Delphi comes in, of course because we can make these special Active Server Objects using Delphi 6 Enterprise.

The Active Server Object Wizard

Delphi 6 Enterprise contains wizards that accelerate the creation of Active Server Objects. You can still write Active Server Objects using Delphi 6 Professional, but you'll have to do a lot of the work manually—users would be well-advised to consider moving to the Enterprise edition if development time is at a premium.

The Object Repository of Delphi 6 Enterprise contains a wizard to create new Active Server Objects on the ActiveX tab. To create a new Active Server Object (referred to as ASP Object from now on), you must close all projects (if any are open), and start a new ActiveX Library to contain our ASP Object. This can be done using the following steps:

1. Start Delphi 6 and close the default project.
2. From the menu, choose File, New, Other and select the ActiveX Library icon from the ActiveX tab in the Delphi 6 Object Repository (see Figure 22.1).
3. Save your ActiveX Library project as **D6ASP.dpr**.

**FIGURE 22.1**

The ActiveX Library icon on the ActiveX tab.

TIP

If, like me, you grow tired of having to close down the default project every time you start Delphi, you will be happy to learn that there is an easy way to have Delphi start up with no project loaded, using a `-np` command line option. You have to go to the program group that contains your Delphi 6 shortcut and change the way Delphi 6 is started.

To do so, right-click on the taskbar and select the Properties pop-up menu. Go to Start Menu Programs, and click on the Advanced button. You are now exploring the Start Menu items. Go to the Program group of All Users, which will contain the Borland Delphi 6 group. Select the Delphi 6 item, and right-click to get a pop-up menu in which you can select the Properties option. Click on the tab that says Shortcut, and add the `-np` text right after the current value specified in the Target editbox. For example, a default would look like the following:

```
"C:\Program Files\Borland\Delphi6\Bin\delphi32.exe " -np
```

This is also a good place to consider the Start In editbox because you might want Delphi 6 to start in a specific default directory.

When you have saved the ActiveX Library you've just created (as `D6ASP.dpr`), you can add an Active Server Object to it by selecting the Active Server Object icon from the ActiveX tab of the Delphi 6 Object Repository that you also see in Figure 22.1.

This will produce the Delphi 6 New Active Server Object dialog box, shown in Figure 22.2, which needs some explanation if you're seeing it for the first time (especially if you have no or little previous experience with COM or ASP Objects).

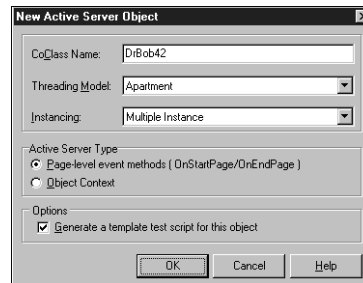


FIGURE 22.2

The New Active Server Object dialog box.

The CoClass Name is the internal name of your COM Object. Normally, you can enter anything here. For the example in this chapter, use `DrBob42` as the CoClass Name. This will result in the classname `TDrBob42`, which derives from `TASPObject` and implements the `IDrBob42` interface. The Threading Model is set to `Apartment` by default, and Instancing is set to `Multiple Instance`. These are fine for most purposes because you shouldn't have to change these settings generally.

The five Threading Model choices are as follows:

- **Single**—All client requests are handled in a single thread. This isn't a good idea because others have to wait until the first client is finished.
- **Apartment**—Every client request runs in its own thread, separated from the others. (No thread can access the state of another.) Class instance data is safe, but we must guard against threading issues when using global variables and the like. This is the preferred threading model that I always use.
- **Free**—A class instance can be accessed by multiple threads at the same time. Class instance data is no longer thread-safe, so you must take care to avoid multiuser issues here.
- **Both**—A combination of Apartment and Free because it follows the Free threading model with the exception that callbacks are executed in the same thread. (So parameters in callback functions are safe from multithreading problems.)

- Neutral—COM+ specific, and it defaults to Apartment for COM. Client requests can access object instances on different threads, but COM ensures that the calls won't conflict. Still, you'll have to watch threading issues (see the chapter on multi-threading) with global variables as well as instance data in between method calls.

The Instancing option offers three choices. Note that it doesn't matter what you select, if you're registering the Active Server Object as an in-process server (we'll cover in-process and out-of-process later), but it's good to know what the choices are:

- Internal Instance—This COM object is only instantiated within its own DLL.
- Single Instance—The application can have one client instance.
- Multiple Instance—A single application (ActiveX Library) can instantiate more than one instance of the COM object.

Also found in this dialog box are the Active Server Type options. These are dependent on the version of the IIS installed on your machine. For IIS 3 and IIS 4, the page-level event methods with `OnStartPage` and `OnEndPage` are used, whereas IIS 4 and IIS 5 can also use the Object Context method; that is, using *Microsoft Transaction Server (MTS)* or COM+ to manage instance data of the Active Server Object.

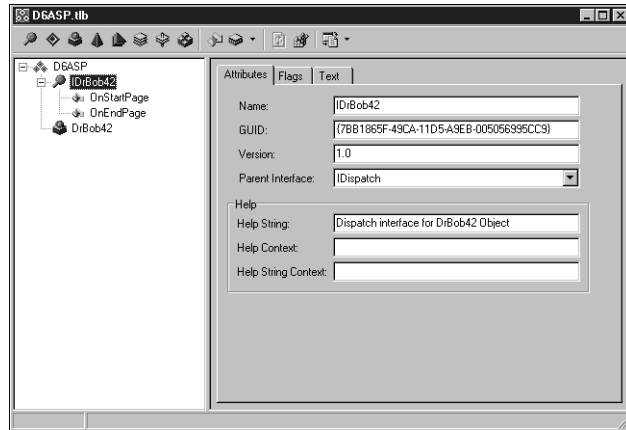
Delphi 6 will encapsulate most of the differences, so for this example, select the default Page-Level Event Methods. You can do the same with Active Server Objects if you select the Object Context option. Remember that you need to select an option that's right for (or at least supported by) your Web server.

The last option on the New Active Server Object dialog box is used to generate a very simple HTML test script for this Active Server Object. If you don't know ASP or the ASP scripting language, this is a good way to begin learning. It consists of only two lines, but the template shows you how to call methods of your Active Server Object using script.

Apart from assigning the CoClass Name, you generally don't have to do anything with this dialog box.

Type Library Editor

The Active Server Object has been created, including a type library for it. You end up in the Delphi 6 Type Library Editor for the DrBob42 Active Server Object shown in Figure 22.3.

**FIGURE 22.3**

The Type Library Editor for IDrBob42.

Save the project files again (File, Save All), which first prompts you for a name for Unit1 (the unit containing the Active Server Object itself). I've named this unit DrBob42ASP.pas. Next, you're asked to save the file DrBob42.asp, which contains the ASP HTML template file. This file initially has the following content:

```
<HTML>
<BODY>
<TITLE> Testing Delphi ASP </TITLE>
<CENTER>
<H3> You should see the results of your Delphi Active Server method below </H3>
</CENTER>
<HR>
<% Set DelphiASPObj = Server.CreateObject("D6ASP.DrBob42")
   DelphiASPObj.{Insert Method name here}
%>
<HR>
</BODY>
</HTML>
```

As we've seen earlier in this chapter, ASP tags use % to distinguish themselves from regular HTML tags. In the single ASP tag, you'll see a two-line script. The first line creates an instance of the DrBob42 object from the D6ASP ActiveX Library, and the second line calls an unnamed method.

The second thing that you might have noticed in Figure 22.3 is that you already see the OnStartPage and OnEndPage methods for your IDrBob42 interface. This is a consequence of selecting the Page-Level Event Methods option in the New Active Server Object dialog box.

(You wouldn't have seen them when selecting the Object Context, as you can see in Listing 22.2.) You can see their implementation in the generated DrBob42ASP unit that contains the source code, shown in Listing 22.1, for your Active Server Object.

LISTING 22.1 DrBob42ASP—Active Server Object Source Code

```
unit DrBob42ASP;

{$WARN SYMBOL_PLATFORM OFF}

interface
uses
  ComObj, ActiveX, AspTlb, D6ASP_TLB, StdVcl;

type
  TDrBob42 = class(TASPObject, IDrBob42)
  protected
    procedure OnEndPage; safecall;
    procedure OnStartPage(const AScriptingContext: IUnknown); safecall;
  end;

implementation

uses ComServ;

procedure TDrBob42.OnEndPage;
begin
  inherited OnEndPage;
end;

procedure TDrBob42.OnStartPage(const AScriptingContext: IUnknown);
begin
  inherited OnStartPage(AScriptingContext);
end;

initialization
  TAutoObjectFactory.Create(ComServer, TDrBob42, Class_DrBob42,
    ciMultiInstance, tmApartment);
end.
```

Before you start adding more methods, let's see what an Active Server Object generated with the Object Context would look like. Fortunately, you can add more than one Active Server Object to a single ActiveX Library, so start the New Active Server Object dialog box again—this time specifying Micha42 as CoClass Name and selecting the Object Context option. Save the resulting source code in Micha42ASP.pas and the corresponding ASP file in Micha42.asp.

The source code listing can be seen in Listing 22.2. The differences are minor. (You only lack the `OnEndPage` and `OnStartPage` events. More importantly, `TDrBob42` is derived from `TASPObject` whereas `TMicha42` is derived from `TASPMTSObject`.) This is one of the major benefits of creating ASP objects with Delphi: the implementation details, while available, aren't necessary for the creation of robust, fast objects. From now on, you can add the same functionality to `DrBob42ASP` or `Micha42ASP`, and they will behave identically while using very different technology behind the scenes.

LISTING 22.2 Micha42ASP—Active Server Object Source Code

```
unit Micha42ASP;  
  
{ $WARN SYMBOL_PLATFORM OFF }  
  
interface  
uses  
    ComObj, ActiveX, AspTlb, D6ASP_TLB, StdVcl;  
  
type  
    TMicha42 = class(TASPMTSObject, IMicha42)  
    end;  
  
implementation  
  
uses ComServ;  
  
initialization  
    TAutoObjectFactory.Create(ComServer, TMicha42, Class_Micha42,  
        ciMultiInstance, tmApartment);  
end.
```

New Methods

It's now time to add a new method to the `IDrBob42` (or `IMicha42`) interface that can be invoked by the outside world (typically from the `.asp` Web page).

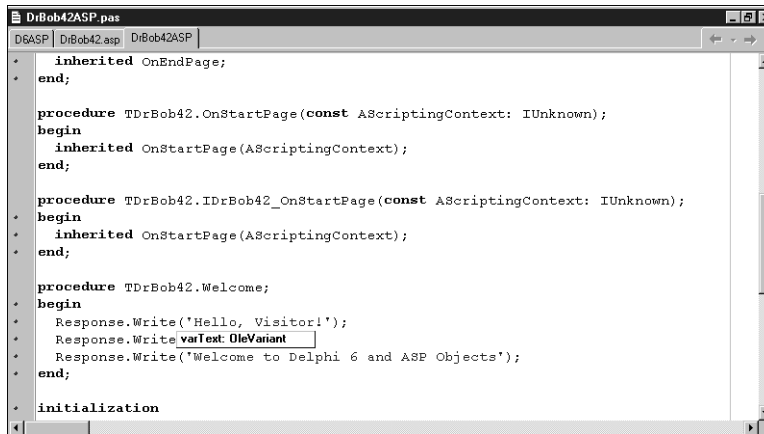
Apart from the `OnEndPage` and `OnStartPage` methods (in the `TDrBob42` object), you can also specify one or more custom methods. For example, using the type library, you can add a method called `Welcome` to the `IDrBob42` interface. (Right-click on the `IDrBob42` node and select `New, Method` from the pop-up menu.)

This method can be used to display a dynamic welcome message. After you've added the method and refreshed the implementation, you can write the code for the `TDrBob42>Welcome`

method. To do this, you should know a little bit about the ASP internal objects and functionality made available by Delphi 6. Like ASP scripting, Delphi ASP Objects have access to special Request and Response objects.

ASP Response Object

The ASP Response object is an internal object that is available within methods of your Active Server Object. You should use Response whenever you want to generate dynamic output. Response has a number of properties and methods to set the content of the response. The most important one by far is the Write method. This method takes an OleVariant as argument (as you can see from the Code Insight hint in figure 22.4) and makes sure that the argument is written to the dynamic output at the exact location in the ASP script where the call appeared inside the <% and %> tags.



```
DrBob42ASP.pas
D6ASP | DrBob42.asp | DrBob42ASP
+
+ inherited OnEndPage;
+ end;
+
+ procedure TDrBob42.OnStartPage(const AScriptingContext: IUnknown);
+ begin
+   inherited OnStartPage(AScriptingContext);
+ end;
+
+ procedure TDrBob42.IDrBob42_OnStartPage(const AScriptingContext: IUnknown);
+ begin
+   inherited OnStartPage(AScriptingContext);
+ end;
+
+ procedure TDrBob42.Welcome;
+ begin
+   Response.Write('Hello, Visitor!');
+   Response.Write varText: OleVariant
+   Response.Write('Welcome to Delphi 6 and ASP Objects');
+ end;
+
+ initialization
```

FIGURE 22.4
The Code Editor.

To write a welcome message, insert the code in Listing 22.3 within the TDrBob42.Welcome method.

LISTING 22.3 Implementation of the Welcome Method

```
procedure TDrBob42.Welcome;
begin
  Response.Write('Hello, Visitor!');
  Response.Write('<P>');
  Response.Write('Welcome to Delphi 6 and ASP Objects');
end;
```

The DrBob42.ASP file needs only a single change inside the ASP tags (the method now has a name: Welcome). The new ASP tags are as follows:

```
<% Set DelphiASPObj = Server.CreateObject("D6ASP.DrBob42")
    DelphiASPObj.Welcome
%>
```

Note that the ASP script doesn't need to destroy or free the DelphiASPObj variable: this will automatically be taken care of when the object gets out-of-scope. Apart from the call to Welcome, you can add more methods to the IDrBob42 interface and call these additional methods also from the ASP script as listed earlier. But let's first take the Active Server Object for a test run and worry about extending it later.

First Run

This is all it takes to prepare for the first operational test of the Active Server Object inside an Active Server Page. All you need to do now is register the D6ASP.d11 Active Server Object and place DrBob42.asp in the correct directory (with ASP scripting rights).

We mentioned earlier in-process and out-of-process options for running the ASP objects. *In-process* means that your ASP object will be loaded and run alongside your Web server, and only unloaded when the Web server shuts down. *Out-of-process* means that your ASP object will be loaded and unloaded as clients request it from the server. In-process objects generally perform better, whereas out-of-process objects are more easily debugged. You can register Active Server Objects in two ways: either as in-process or as out-of-process servers.

To register D6ASP.d11 as an in-process server containing the Active Server Object(s), choose Run, Register ActiveX Server from the Delphi 6 menu.

To unregister the same server, choose Run, Unregister ActiveX Server.

Figure 22.5 shows the confirmation message after you've registered the D6ASP ActiveX Server.

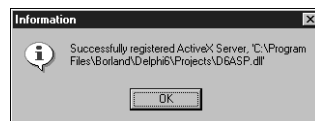


FIGURE 22.5

Registered ActiveX Server.

We'll get back to registering the ActiveX Server as an out-of-process server later in this chapter. But first, let's finish the test run.

After the D6ASP.dll ActiveX Server has been registered on your development machine, you have to move the DrBob42.asp file to a location that has ASP Scripting rights, such as the WWWRoot/Scripts directory.

The requesting URL will be `http://localhost/scripts/DrBob42.asp`; the result is shown in Figure 22.6.

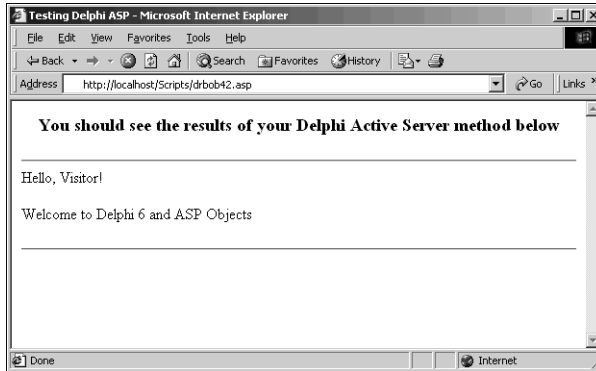


FIGURE 22.6

Running Active Server Object.

Using the `Response.Write` method, you can put any dynamic text at the location where the `Welcome` method was called inside the `DrBob42.asp` Web page.

ASP Request Object

Before we continue, let's consider another very important internal ASP object: `Request`. Like `Response`, `Request` is available within your Active Server Object (interface) methods. `Request` can be used to obtain all input. There are three different ways in which input can be given: by form variables using the `POST` method, by "fat" URL variables using the `GET` method, and using cookies. Each of these possesses a property called `Items`, which is a stringlist for holding the content of the `Response` or `Request`.

A modified `Welcome` method that obtains the `Name` value from the input form used to start the ASP script is shown in Listing 22.4.

LISTING 22.4 Definition of the Modified Welcome Method

```
procedure TDrBob42>Welcome;  
var  
    Str: String;  
begin
```

LISTING 22.4 Continued

```
Str := Request.Form.Item['Name'];
Response.Write('Hello, '+Str+'!');
Response.Write('<P>');
Response.Write('Welcome to Delphi 6 and ASP Objects');
end;
```

The same technique can be used for the `QueryString` and `Cookies` objects.

Recompiling Active Server Objects

If you went on and tried to recompile the D6ASP project again, you probably received this Delphi error message: `Could not create output file D6ASP.dll..` You received the error because the Active Server Object `DrBob42` inside `D6ASP.dll` has been used and is still cached by the Web server. So, when you try to recompile your Active Server Object, the linker will give an error message: `The file is still in use..` You can try to shut down IIS, but that won't help. Shutting down the World Wide Web Publishing Service won't help either. You actually have to shut down the entire IIS Admin Service before the `ASP.DLL` and all Active Server Objects are released from memory so that you can recompile any of them. Note that shutting down IIS Admin Service from the dialog box shown in Figure 22.7 means that all dependent services (WWW, FTP, and so on) will shut down as well. This isn't something you want to do on a live Web server, of course.

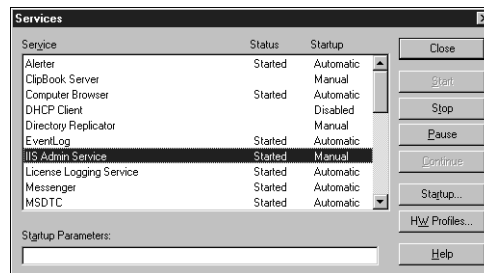
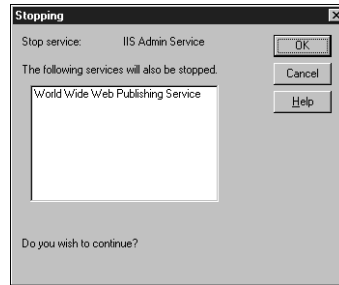


FIGURE 22.7

IIS Admin Service.

If you try to shut down the IIS Admin Service, you will see the Stopping dialog box shown in Figure 22.8, telling you which sub-services depend on the IIS Admin Service and will also have to shut down first.

**FIGURE 22.8**

Stopping World Wide Web Publishing Service.

To simplify the process of shutting down and starting up the required NT services, the following simple batch file, `RESTART.BAT`, can be used when you want to recompile and redeploy an Active Server Object:

```
net stop "World Wide Web Publishing Service"  
net stop "IIS Admin Service"  
net start "World Wide Web Publishing Service"
```

TIP

On a machine that will be used for development only, you can specify that your Active Server Object should explicitly not be cached by your web server. Obviously, this should never be used on production machines because it effectively turns Active Server Objects into even slower CGI applications: loading them for each client request. This is equivalent to having an out-of-process Active Server Object that is generally never used, except in this particular development situation.

Running Active Server Pages Again

Although you can load an Active Server Page by itself, it's often more effective if you run it in response to an HTML input form. For this example, you can use the small HTML page that follows:

```
<HTML>  
<HEAD>  
<TITLE>Dr.Bob's ASP Example</TITLE>  
</HEAD>
```

```
<BODY BGCOLOR=FFFFCC>
<FONT FACE="Verdana" SIZE=2>
<FORM ACTION="drbob42.asp" METHOD=POST>
Name: <INPUT TYPE=text NAME=Name>
<P>
<INPUT TYPE=submit>
</FORM>
</BODY>
</HTML>
```

Loaded inside Internet Explorer as page `http://localhost/cgi-bin/drbob42.htm`, this gives the output shown in Figure 22.9. I've already typed a name in the edit box, and am now ready to click the Submit Query button.

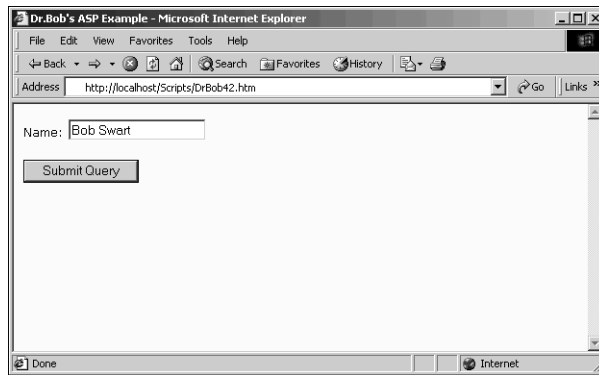
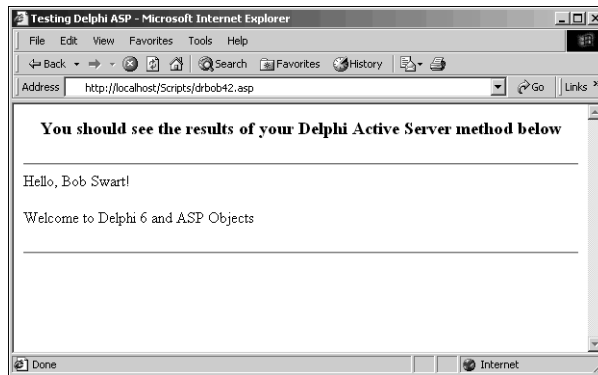


FIGURE 22.9

Internet Explorer with DrBob42.htm.

After you fill your name in the edit box and click Submit Query, the Active Server Page is loaded. It will create an instance of the DrBob42 Active Server Object and call the `Welcome` method as specified in the ASP script. This will result in the dynamic output shown in Figure 22.10.

This was yet another simple ASP example, using only `Request` and `Response`, but you get the idea. We'll now continue with some more ASP internal objects such as `Session`, `Server`, and `Application`, and then we will return to Delphi specific Web server application support—such as `WebBroker` components—most of which also can be used in combination with Active Server Objects.

**FIGURE 22.10**

Internet Explorer with DrBob42.asp output.

ASP Session, Server, and Application Objects

Apart from the Request and Response objects, ASP also has access to Session, Server, and Application objects. This is actually one of the benefits of ASP over CGI and ISAPI: An Active Server Object can access session and application information without any further effort on your part (by using cookies, hidden fields, or fat URLs).

Recall from the previous example that your TDrBob42 class was derived from the TASPObject class. In addition to the Request and Response properties, this class introduces the TASPObject.Session, Server, and Application properties. These properties provide direct access to the underlying ASP Session, Server, and Application objects, respectively. You can also use these properties from within your TDrBob42 methods to store the names of each visitor to your Web site, for example. In ASP HTML, this could be done as follows (note the second line):

```
<% Set DelphiASPObj = Server.CreateObject("D6ASP.DrBob42")
Session.Value("Name") = "Bob Swart"
DelphiASPObj.Welcome
%>
```

To obtain this persistent value (persistent among other Active Server Pages that are visited by the same user in the same session), you can use the Session object in your Active Server Object, just like using the Form, QueryString, or Cookies object (see Listing 22.5).

LISTING 22.5 DrBob42ASP—Active Server Object Source Code

```
procedure TDrBob42.Welcome;
var
  Str: String;
begin
  Str := Session.Value['Name'];
  Response.Write('Hello, '+Str+'!');
  Response.Write('<P>');
  Response.Write('Welcome back to Delphi 6 and ASP Objects');
end;
```

The `Session` object maintains its state using cookies, so make sure that your Web server has cookies enabled.

Active Server Objects and Databases

Now, let's make the example a bit more useful and introduce a database or table in it, so we can demonstrate how to perform a query or open a table on the server and show the results inside an Active Server Page. In order to add this functionality, you should first add a new data module, using File, New, Data Module.

Give the `Name` property of the data module a new value, say `DataModuleASP`, and save this new unit in file `DataMod.pas`. Also choose File, Save All to save the entire project so that your main project file `D6ASP.dpr` will contain the data module in its `uses` clause. Inside this data module, you will use a `TClientDataSet` component from the Data Access tab because this will be the simplest way of providing a dataset, and also the most flexible way. You can replace it with another dataset component later if you want to extend this example.

Drop a `TClientDataSet` component on the data module. In order to supply it with data, click on the ellipsis button next to the `FileName` property. Go to the `C:\Program Files\Common Files\Borland Shared\Data` directory and you'll see all the well-known tables from `DBDEMOS` in `MyBase XML` as well as binary `ClientDataSet` format. Select the `bio1ife.xml` file for this example.

Having the data module, you should still worry about sharing it in a multithreading environment! The best way is to create the data module inside your Active Server Object when you need it, either in the `BeginPage` and `EndPage` events, or—even more clearly—inside the `Welcome` method itself.

But you need to add the `DataMod` unit to the `uses` clause of the `DrBob42ASP` unit, so you can actually use it. Then, write the code from Listing 22.6 inside the `Welcome` method to create, use, and safely destroy the data module.

LISTING 22.6 DrBob42ASP—Active Server Object Source Code

```
procedure TDrBob42.Welcome;
var
  Str: String;
  DM: TDataModuleASP;
begin
  Str := Request.Form.Item['Name'];
  Response.Write('Hello, '+Str+'!');
  Response.Write('<P>');
  Response.Write('Welcome to Delphi 6 and ASP Objects');
  try
    DM := TDataModuleASP.Create(nil);
    // use DM...
  finally
    DM.Free;
  end
end;
```

In order to present the information from the dataset to the browser, let's walk through the data inside the `ClientDataSet` and produce a grid-like HTML table that shows the common names, `Common_Name`, and description, `Notes`, of the fish listed in the `biolife` dataset. This only takes a few lines of additional code producing dynamic HTML (see Listing 22.7).

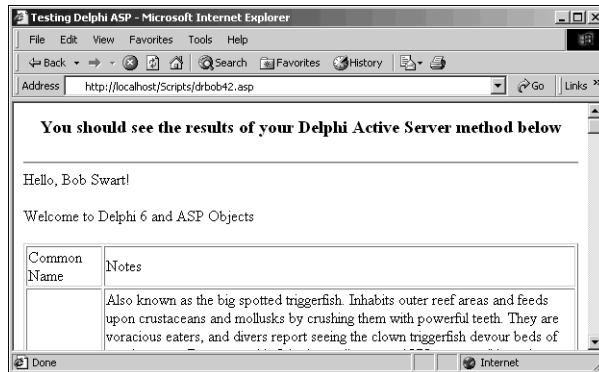
LISTING 22.7 DrBob42ASP—Active Server Object Source Code

```
procedure TDrBob42.Welcome;
var
  Str: String;
  DM: TDataModuleASP;
begin
  Str := Request.Form.Item['Name'];
  Response.Write('Hello, '+Str+'!');
  Response.Write('<P>');
  Response.Write('Welcome to Delphi 6 and ASP Objects');
  try
    Response.Write('<P>');
    DM := TDataModuleASP.Create(nil);
    with DM.ClientDataSet1 do
      try
        Open;
        First;
        Response.Write('<TABLE BORDER=1><TR><TD>Common_Name</TD>');
        Response.Write('<TD>Notes</TD></TR>');
```

LISTING 22.7 Continued

```
while not Eof do
begin
  Response.Write('<TR><TD>');
  Response.Write(FieldByName('Common_Name').AsString);
  Response.Write('</TD><TD>');
  Response.Write(FieldByName('Notes').AsString);
  Response.Write('</TD></TR>');
  Next
end;
Close;
finally
  Response.Write('</TABLE>')
end;
finally
  DM.Free
end
end;
```

The output can be seen in Figure 22.11.

**FIGURE 22.11**

Dynamic HTML output from Active Server Object.

Delphi already contains a lot of helpful components and techniques to produce dynamic and well-formatted HTML in the NetCLX HTML-producing components called *PageProducers*. Rather than spend a lot of time learning HTML, using PageProducers will generate the HTML needed dynamically.

Active Server Objects and NetCLX Support

If you compare Active Server Objects with the NetCLX architecture, you should notice a lot of similarities. Both use Request and Response objects as the primary means to communicate with the client. From a developer point of view, however, there is much more support for NetCLX with the PageProducer and TableProducer components that were especially written for use inside Web modules. Fortunately, these HTML-producing components aren't limited to use inside Web modules; they can be used anywhere, and you're free to dynamically create a TDataSetTableProducer, assign it to a dataset, and write the resulting HTML back using the Response.Write method. You can drop a TDataSetTableProducer component on the data module you just created and even customize it at design-time!

In fact, it isn't very hard to use the same HTML-producing components, originally written for NetCLX, inside your Active Server Object. The only exceptions are the TQueryTableProducer and TSQLQueryTableProducer components, which rely on input passed on by the NetCLX Request object, not the ASP Request object. All other PageProducers can be used as they are, as the next example will demonstrate.

Drop a TDataSetTableProducer component on the data module, and assign its DataSet property to the ClientDataSet you used in the previous example. In order to customize the settings of the DataSetTableProducer, make sure that the ClientDataSet actually contains data. So, temporarily set the Active property of the ClientDataSet component to True (set it back to False afterward), and then click on the ellipsis next to the Columns property of the DataSetTableProducer (or right-click on the DataSetTableProducer component and select Response Editor). This will give you the Columns property editor shown in Figure 22.12.

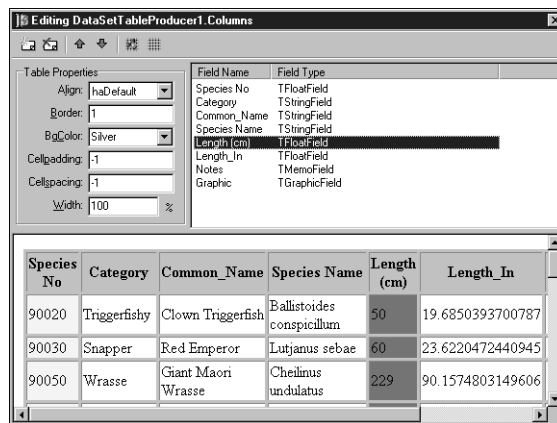


FIGURE 22.12

DataSetTableProducer Response Editor.

Back in the `Welcome` method, call the `DataSetTableProducer.Content` method as shown in Listing 22.8 to get the dynamic produced HTML output you need (a lot smaller than you had to write yourself, and much easier to customize as well).

LISTING 22.8 DrBob42ASP—Active Server Object Source Code

```

procedure TDrBob42>Welcome;
var
  Str: String;
  DM: TDataModuleASP;
begin
  Str := Request.Form.Item['Name'];
  Response.Write('Hello, '+Str+'!');
  Response.Write('<P>');
  Response.Write('Welcome to Delphi 6 and ASP Objects');
  try
    Response.Write('<P>');
    DM := TDataModuleASP.Create(nil);
    Response.Write(DM.DataSetTableProducer1.Content);
  finally
    DM.Free;
  end
end;

```

After you've recompiled your Active Server Object, you can reload the `DrBob42.htm` file to start the Active Server Page, resulting in the output shown in Figure 22.13.

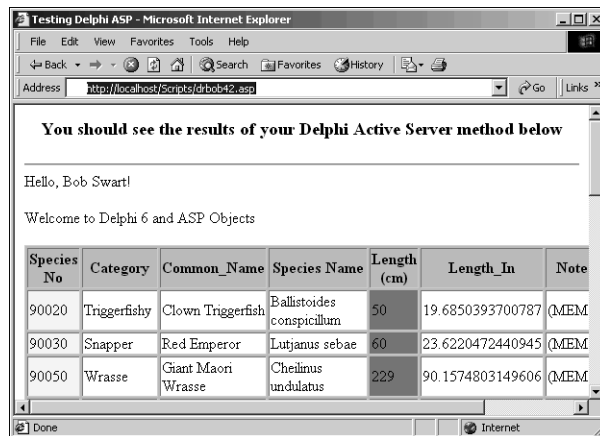


FIGURE 22.13

Internet Explorer with DrBob42.asp NetCLX output.

Debugging Active Server Objects

As you saw in the previous section, Active Server Objects are like ISAPI DLLs: After they are loaded, you need to bring down the entire Web server to unload them because Active Server Objects are loaded by the ASP.DLL, which is an ISAPI DLL in itself. However, the advantage of ASP is the fact that for the Active Server Pages themselves, you can update the scripts as much as you want, without having to change, unload, or reload the Active Server Objects themselves. As long as the functionality inside the Active Server Object doesn't change, you need only to update the scripts. Of course, making sure that the Active Server Objects work correctly is another task, which at times requires the capability to debug Active Server Objects.

When it comes to debugging Active Server Objects, a few things can be done right away, such as showing a simple message box or using a debug window to show strings sent from the Active Server Object. In order to get any of these messages, however, you must first specify that the owner of the Active Server Object is indeed qualified to interact with the desktop. Specifically, for the IIS Admin Service, set the Interact with Desktop option in the Services applet (dialog box) of the Control Panel Services dialog box, as shown in Figure 22.14.

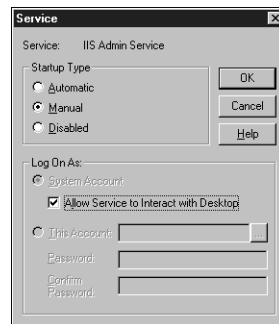


FIGURE 22.14

Allow Service to interact with Desktop.

After you've specified that option, you can use almost any means to have your Active Server Objects interact with the desktop. This is still a bit crude, but it can be effective enough at times.

Debugging Active Server Objects with MTS

There is an easier way to manage your Active Server Objects, which also greatly improves your abilities to actually debug Active Server Objects written in Delphi (or C++Builder for that matter). The solution, as the title of this section indicates, involves MTS as host for your Active Server Object.

The first step involves unregistering the Active Server Object you've written in this chapter. This can be done by choosing Run, Unregister ActiveX Server from the Delphi 6 menu.

After the Active Server Object has been successfully unregistered, you can register it again, but this time as an MTS Object. To do this, choose Run, Install MTS Objects. (On Windows 2000, this menu option will be called Install COM+ Object.) The resulting dialog box appears in Figure 22.15.

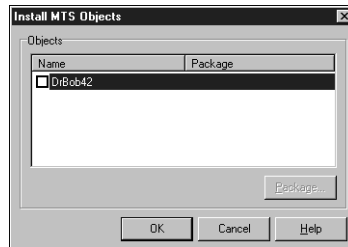


FIGURE 22.15

Install MTS (or COM+) Objects.

In this Install MTS Objects dialog box, select your DrBob42 object by clicking on the check box. This will result in the pop-up dialog box shown in Figure 22.16, which asks for a package name to install the DrBob42 object into. You can either select an existing package, or specify a new package such as DelphiDebugPackage: The COM+ dialog box works in a similar way on Windows 2000.

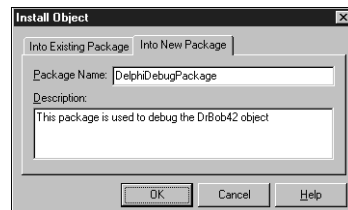


FIGURE 22.16

Install Object DelphiDebugPackage.

Now, click OK in the Install Object dialog box and click OK again in the Install MTS Objects dialog box.

After having installed the DrBob42 object as an MTS object, you can debug the Active Server Object from within the Delphi IDE itself. For this, you need a host application in order to load the Active Server Object. The steps you must take from this point on differ in Windows NT

and Windows 2000. First, I'll show you the steps for Windows NT, followed by the steps for Windows 2000.

Debugging Using Windows NT 4

For Windows NT with the Option Pack installed, you need to specify MTS as the Host Application. MTS will already be running, so you must shut it down first.

CAUTION

As a consequence, you should never try to do this on a real production machine. Only use a development machine in which you can afford to shut down MTS from time to time (when debugging your Active Server Objects that are hosted inside MTS).

22

ASP
DEVELOPMENT

In order to shut down MTS, you must start the Internet Service Manager application, which is part of the Windows NT 4 Option Pack. Open the Microsoft Transaction Server node in the treeview until you see Packages Installed under My Computer (which includes the DelphiDebugPackage you just installed).

If you right-click on the My Computer icon, you can shut down all server processes (see Figure 22.17). This won't give you any feedback. You can verify the shutdown by looking at the Processes list of the Windows NT Task Manager. It shouldn't list `mtx.exe` anymore.

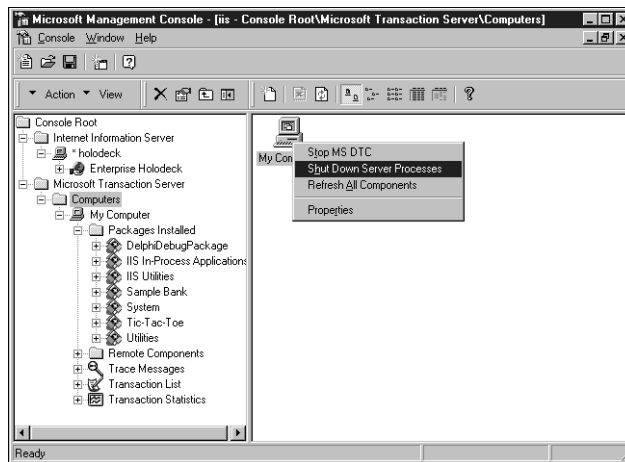


FIGURE 22.17

Shutdown server processes.

Now all you have to do is specify MTS as the Host Application inside the Delphi 6 Run Parameters dialog box, so you can use MTS as host for your DrBob42 Active Server Object. On my NT 4 machine, that's `c:\winnt\system32\mtx.exe`. You must also specify the package that contains your DrBob42 object using the `/p:"DelphiDebugPackage"` parameter. In this case, the DrBob42 object is `DelphiDebugPackage`. Specifying the package brings up a dialog box similar to that shown in Figure 22.18.

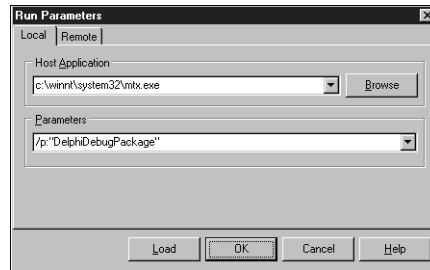


FIGURE 22.18

Run Parameters dialog box.

Now, set a breakpoint in your code by clicking in the left-side gutter or pressing F5 while your cursor is on a line of code and press F9 to run and debug your DrBob42 Active Server Object as hosted inside MTS. Nothing will happen because MTS is running, but your Active Server Object has not been invoked by a browser call yet. You must now restart Internet Explorer (or another browser) and load the `DrBob42.asp` Web page that will load the Active Server Object.

This will trigger the breakpoint: At which time, you're able to use the Delphi integrated debugger on your Active Server Object.

In order to end the Delphi debug session, you have to shut down MTS again (just as you did when you started to debug using MTS).

Debugging Using Windows 2000

Using Windows 2000, you can no longer use `mtx.exe` simply because under Windows 2000, MTS is integrated into the operating system. However, you can use `dllhost.exe` to load the ProcessID of your Active Server Object. This technique will also work on Windows NT. but, it is slightly more complex, which is why I first showed you how to debug using MTS as the Host Application under Windows NT.

You should now use `dllhost.exe` as the Host Application, which can be specified in the Run, Parameters dialog box. The parameter should be the ProcessID of the `DelphiDebugPackage` containing your DrBob42 Active Server Object. You can obtain this information using the

Internet Service Manager (Microsoft Management Console) on Windows NT or the Component Services on Windows 2000.

The Package ID of the `DelphiDebugPackage` in this example is `{50AE66A2-349B-11D5-A9F0-005056995CC9}`. This ID can be copied from the text label of the Run Properties dialog box shown in Figure 22.19. This is the most convenient way to copy it because you probably don't want to type it in yourself. Figure 22.20 shows the Run Parameters dialog box with the ID pasted into the Parameters text box.

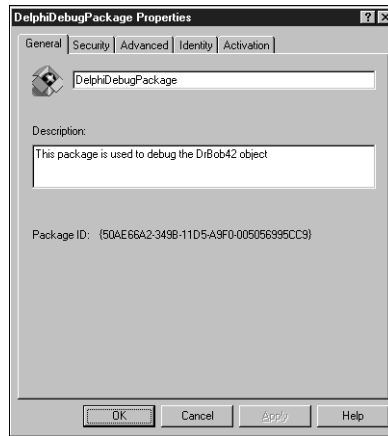


FIGURE 22.19

DelphiDebugPackage Package ID.

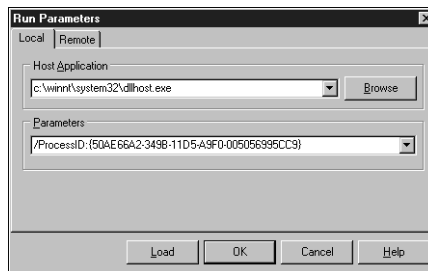


FIGURE 22.20

Run Parameters with ID inserted.

Now, make sure to set a breakpoint and press F9 to run (and debug) your DrBob42 Active Server Object. Similar to the previous example, nothing will happen until the ASP object is invoked through a browser. You must restart Internet Explorer (or another browser) and load

the `DrBob42.asp` Web page that will in turn load the Active Server Object. This will trigger the breakpoint; at which time, you're able to use the Delphi integrated debugger on your Active Server Object.

Note that in order to end the Delphi debug session, you have to shut down `DelphiDebug` Package inside MTS again (just as you did when you started to debug using MTS).

Summary

In this chapter, you have learned what Active Server Pages are, what role Active Server Objects play in them, and how Delphi 6 can be used to write these Active Server Objects. You've also seen how you can use internal objects (like `Request` and `Response`), how you can add database processing to your Active Server Objects, how you can combine Active Server Objects and `NetCLX` components, and finally how to debug Active Server Objects with Delphi 6 under Windows NT or Windows 2000.