Under Construction: Delphi 5 Active Server Objects

by Bob Swart

Contrary to my promise last month, I won't be talking about MIDAS 3 object pooling and connection brokering mechanisms in MIDAS 3 (I hit some problems). For the next two months we'll be finding out about Delphi 5's support for ASP Objects.

Active Server Pages

Before we start with Active Server Objects, let's examine the encapsulating technology called Active Server Pages (ASP). Before ASP, we could use certain scripting languages (such as JavaScript or VBScript) only on the client (ie, the web browser). For server-side processing, we could use CGI applicatons, ISAPI/NSAPI web server extension DLLs, or other scripting languages like iHTML (www.ihtml.com). Microsoft released ASP 1.0 in 1996, which was supported by Internet Information Server (IIS) 3.0. A year later ASP 2.0 took over, supported by IIS 4. A short while ago, ASP 3.0 was introduced with Windows 2000 and IIS 5. ASP 3.0 has some new features which we'll cover next time. For now, we'll mainly focus on the more widely used ASP 2.0 features.

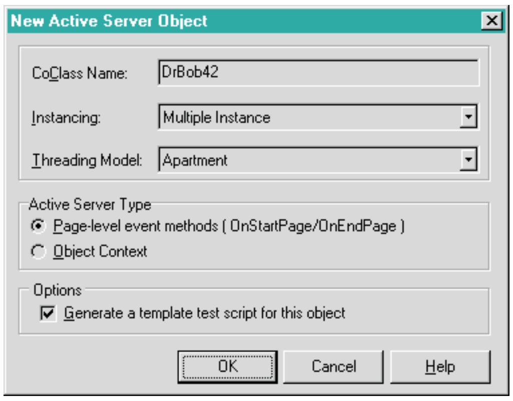
ASP is a server-side web solution. ASP comprises an optional compiled part (the Active Server Object) and a scripting part. ASP scripting is included inside `<%` and `%>` tags. For more background information on ASP 2.0, check out the book *ASP 2.0 Programmer's Reference* published by WROX, reviewed by me in the June issue of *Developers Review*.

In this article, we'll focus on the Active Server Objects that are created and used by these scripts.

Active Server Objects

Delphi 5 introduces a new wizard that enables us to create so-called Active Server (Page) Objects. These can be used in Active Server Pages to dynamically generate HTML code every time the server loads the page. To create a new Active Server Object, we must first create a new ActiveX Library (from the `ActiveX` tab of Delphi's Object Repository). Once we have an ActiveX Library, give it a sensible name (TDM58.dpr for example), then add an Active Server Object to it by double clicking on the Active Server Object icon, in the `ActiveX` tab of the Object Repository.

This last step produces the dialog in Figure 1, which needs some explanation if you're seeing it for the first time. The `CoClass Name` is the internal name of our COM Object. We can pick something like `DrBob42` here, or basically anything. The `Instancing` and `Threading Model` fields have good default values, as usual. `Multiple Instance` means that we have one ActiveX library which creates an instance of the `DrBob42` object for every client which wants to connect to it. `Single Instance` means that the ActiveX library can only create one instance of the `DrBob42` object; so, for every client which wants to connect to it, a new instance of the ActiveX library is loaded to create a single instance of the `DrBob42` object. The second option avoids multi-threaded issues between instances of `DrBob42` objects, but is much more resource intensive, as it requires a new executable (ActiveX library, DLL or OCX) to be loaded for every client connection.



CoClass Name: DrBob42
Instancing: Multiple Instance
Threading Model: Apartment
Active Server Type
(●) Page-level event methods (OnStartPage/OnEndPage)
( ) Object Context
Options
[✓] Generate a template test script for this object

➤ *Figure 1*

Check out the book *Delphi COM Programming* by Eric Harmon (reviewed on my website) for more in-depth information about these options and their effect.

What's more interesting are the Active Server Type options. These are in fact dependent on the version of IIS installed on your web server. For IIS 3 and 4, the page-level event methods using `OnStartPage` and `OnEndPage` are used, while IIS4 (again) and IIS5 can use the object context approach, using MTS to manage instance data of the Active Server Object. Most people I know who are working with ASP are using the `OnStartPage/OnEndPage` way, which is much easier than using MTS, so that's the option we're going to be working with. Next time I will show that the difference is not signifcant for an ASP developer anyway, as Delphi does a good job of encapsulating those issues for us.

The last option is used to generate a (very simple) HTML test script for this Active Server Object. If you don't know ASP or any of the scripting languages, this is a good way to learn. It's very basic, but the template already shows you how to call methods of your Active Server Object, so I always leave it checked.

This is a good moment to mention the fact that ASP developers have a number of ASP Objects to support them, the most important of which are `Request`, `Response` and `Session`. Each of these has a number of properties and methods that give us access to the input, output and session state of the Active Server Pages themselves. And the best thing is these

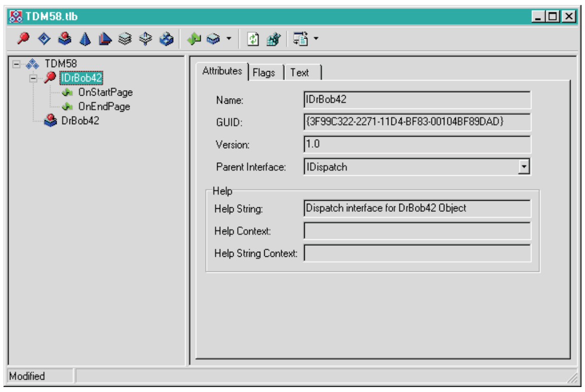
Objects are available both in the scripting language in the Active Server Pages (the HTML pages) and in the Active Server Objects, as for example encapsulated by Delphi 5. Why do I make this point? Well, mainly to state that the entire ASP scripting language can be ignored by us from now on, since just about everything we can do in it can be done in an Active Server Object as well. The goal in both locations is to dynamically generate HTML. And what better way to do so than inside our Active Server Object? So, in most cases, the generated HTML test script will only need one modification to suit our needs, and will then be left alone.

In summary, apart from the `CoClass Name`, we don't have to do anything with this dialog. So, just type the name of your CoClass (`TDM58`) and hit `OK` to continue. Now the Active Server Object is created, including a Type Library, and we end up in the Delphi 5 Type Library Editor for the Active Server Object.

You may have noticed from the Figure 2 screenshot that we already see the `OnStartPage` and `OnEndPage` methods for our `ITDM58` interface. As usual with Type Library interface methods, we can see their implementation too in the unit that contains the source code for our Active Server Object: see Listing 1.

Note from this code that if you want to change from multiple instance to single instance later, you can always change `ciMultiInstance` to `ciSingleInstance` inside the `initialization` section of this unit.

Apart from the Type Library and source code, you may also have noticed that a file called



Name: IDrBob42
GUID: {3F99C322-2271-11D4-BF83-00104BF89DAD}
Version: 1.0
Parent Interface: IDispatch
Help
Help String: Dispatch interface for DrBob42 Object
Help Context:
Help String Context:

➤ *Figure 2*

DrBob42.asp is created (and even opened inside the Code Editor). This file has the content shown in Listing 2.

Like I said before: ASP tags contain `%` to distinguish them from regular HTML tags. And in the single ASP tag, we see a two-line script. The first line creates an instance of our `DrBob42` object inside the TDM58 ActiveX library, while the second line calls a yet-unnamed method from it, which we must create first.

Custom Methods

Apart from the `OnEndPage` and `OnStartPage` methods, we can now also specify our custom methods. For example, using the Type Library we can add an `ASProduce` method to the `IDrBob42` interface, which can be used to produce dynamic HTML output (such as a welcome message). After we've added the method and refreshed the implementation, we can write the code for the `TDrBob42.ASProduce` method. As I mentioned earlier, ASP offers access to the `Request` and `Response` objects.

`Request` contains three important properties: `Form`, `QueryString` and `Cookies`. Each of these has an `Item` sub-property which can be used to extract specific values for specified items. We'll see some examples next time, when we connect Active Server Pages to HTML input forms.

This time, we only want to produce some dynamic HTML output, in which case we need to call the `Write` method with a string argument (note that we need to add the `SysUtils` unit to the `uses` clause of the `implementation` section to be able to use the `TimeToStr` function). See Listing 3.

The DrBob42.asp file only needs a single change inside the ASP tags, so I'm only showing the new ASP tags in Listing 4.

Deploying Active Server Objects

This is all it takes to prepare ourselves for the first operational test of the Active Server Object inside

```
unit Unit1;
interface
uses
  ComObj, ActiveX, AspTlb, TDM58_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.
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
<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("TDM58.DrBob42")
   DelphiASPObj.{Insert Method name here}
%>
<HR>
</BODY>
</HTML>
```

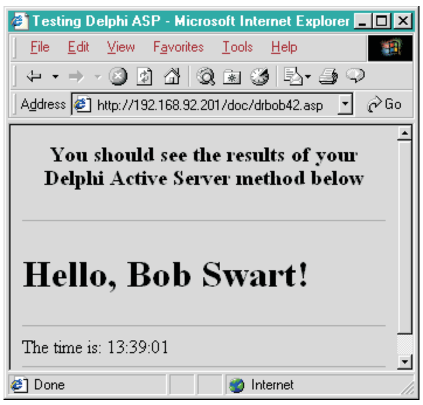
an Active Server Page. All we need to do now is register the TDM58.ocx Active Server Object and place DrBob42.asp in the right directory on the web server (with ASP scripting rights).

We can register Active Server Objects in two ways: as in-process or out-of-process servers. The former are more common and more secure, so we'll be using in-process servers only. To register TDM58.ocx as an in-process server, we need to execute `Run | Register ActiveX Server` from the Delphi IDE. To unregister the same server (for example when you want to remove it entirely from your computer), we need to perform `Run | Unregister ActiveX Server`.

After we've registered the ActiveX server (which includes our Active Server Object), we must deploy our Active Server Page. This requires a working (and running) version of Internet Information Server (IIS) version 4.0 in this case: part of Windows NT 4 Server with the NT4 Option Pack. Alternatively, on NT4 Workstation or Win95/98, it's possible to use the Personal Web Server (PWS), but I must admit I have no experience with PWS myself (according to the documentation, it should work exactly the same).

Having a web server available, we must make sure that the DrBob42.asp page is uploaded to a directory which has the scripting rights set. In IIS4 this is managed by the Internet Service Manager (also part of NT4 Option Pack). For each directory which contains ASP web pages, we must specify the `Script` attribute (which is 'less' than the `Execute` attribute required for CGI executables and ISAPI DLLs). The ActiveX Library that contains the Active Server Object can be on any location of the machine, as long as it's properly registered (so the Active Server Page knows where to find it to create it).

➤ *Figure 3*



Testing Delphi ASP - Microsoft Internet Explorer

http://192.168.92.201/doc/drbob42.asp

You should see the results of your Delphi Active Server method below

Hello, Bob Swart!

The time is: 13:39:01

Done

Internet

```
procedure TDrBob42.ASProduce;
begin
  Response.Write('<H1>Hello, world!</H1>');
  Response.Write('<HR>');
  Response.Write('The time is: '+TimeToStr(Now));
end;
```

➤ *Above: Listing 3*

➤ *Below: Listing 4*

```
<% Set DelphiASPObj = Server.CreateObject("TDM58.DrBob42")
   DelphiASPObj.ASProduce
%>
```

Once all this is done, we can open a web browser and enter the full URL of the Active Server Page (the .asp file) that we just created. Note that you must address this file using an http://... URL, and not via a file:///... URL (so opening it from Windows Explorer won't work), as a web server must be 'triggered' to actually interpret the ASP scripting code, which isn't done if you load a file via the file:///... protocol.

The IP address of my machine (also used by IIS4 on my machine) is 192.168.92.201, and the DrBob42.asp page was uploaded to the DOC virtual directory, so the URL I need is

```
http://192.168.92.201/
  doc/drbob42.asp
```

which results in the screenshot shown in Figure 3.

ASP Session Information

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: the fact that an Active Server Object can access (persistent) session and application information without any further effort on our part. The most useful of these is the `Session` object, which can be used to maintain (session-specific) state information. Our `TDrBob42` object is derived from the `TASPObject` which has all the above properties, so we can use them directly, for example, to store the name of the visitor of our website. In ASP script, this could be done as in Listing 5 (note the second line).

It's easy to use Delphi code to obtain this persistent value (persistent among other Active Server Pages that are visited by the same user in the same session), since we can use the `Session` property. `Session` has a `Values` property which is the default property, so we can call `Session.Values['UserName']` or `Session['UserName']` to obtain the value of the variable with the name `'UserName'` that was set in the current session (Listing 6).

Great! We no longer have to rely on fat URLs, hidden fields or cookies to store session-specific persistent information, we can use the `Session` object. But how does the ASP `Session` object maintain this state, you ask? Good question! Err, actually, it's using cookies behind the scenes...

Testing And Debugging ASP

Active Server Objects appear to be like ISAPI DLLs: once loaded, you need to bring down the web server to unload them (this is because Active Server Objects are loaded by the ASP.DLL which is an ISAPI DLL in itself). However, the advantage of ASP is that for the Active Server Pages themselves, you can update the scripts as much as you want, without having to change, unload or re-upload the Active Server Objects. As long as the functionality inside the Active Server Object doesn't change, you only need to update the scripts. Of course, making sure the Active Server Objects work correctly is another task, which at times

```
<% Set DelphiASPObj = Server.CreateObject("TDM58.DrBob42")
   Session.Value("UserName") = "Bob Swart"
   DelphiASPObj.ASProduce
%>
```

➤ *Above: Listing 5*

➤ *Below: Listing 6*

```
procedure TDrBob42.ASProduce;
var
  UserName: String;
begin
  UserName := Session['UserName'];
  Response.Write('<H1>Hello, '+UserName+'!</H1>');
  Response.Write('<HR>');
  Response.Write('The time is: '+TimeToStr(Now));
end;
```

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

➤ *Listing 7*

requires the ability to debug Active Server Objects.

The big problem of debugging Active Server Objects is the fact that they are loaded by the ASP.DLL (which is an ISAPI DLL loaded by the web server), and they remain loaded in memory as long as the IIS Admin is loaded. This means that stopping your web server is *not enough* to bring your Active Server Objects down! And without them being unloaded, you cannot even recompile them (not once they're loaded). And this is very painful if you want to make changes to Active Server Objects.

As soon as you want to update the Active Server Objects themselves, you are in a *worse* situation that when using ISAPI DLLs. At least ISAPI DLLs are unloaded when you stop the World Wide Web Publishing Server service. But for Active Server Objects you even have to unload the IIS Admin Service (which means no WWW, no FTP, no SMTP, no nothing, everything goes down until you bring it up again). Because it takes a while to do all this using the Services dialog in the control panel, I've written a little batch file (restart.bat), see Listing 7, that can do the job for me.

Note that starting the World Wide Web Publishing Service automatically starts the IIS Admin Service too. Any other services depending on the IIS Admin Service (such as FTP) will remain down. Using restart.bat, at least you can unload the Active Server Object from memory and recompile it again. You don't have to re-register it (unless you move it to another location on your machine).

[It's worth saying, as web hosters ourselves, that few commercial web hosters will be remotely interested in letting you do this kind of thing. In practical terms, if you want to use your own Active Server Objects or ISAPI DLLs, you either need to be running an intranet or have your own physical web server, either co-hosted or on your own dedicated internet connection. For the web, these technologies are really for high-throughput sites only, which can justify a dedicated server. Ed.]

Chapter 49 of the *Delphi 5 Developer's Guide* (which ships with Delphi 5) contains a short description of how to debug Active Server Objects. I don't know who wrote this text or why, but it's plain wrong and doesn't work on any of the machines that I tried it on. Next time, one of the things I'll show you is how to add some clever debug messages to your Active Server Objects (although actually debugging them seems to remain next to impossible).

Next Time

So far, we've seen an indication that Active Server Pages and Active Server Objects are an alternative compared to old-fashioned CGI or ISAPI web server applications. Next time we'll finish our Active Server Objects coverage as we dig into the `Request` object in some more detail, and I will show that we can re-use the `PageProducer` and `TableProducer` components from WebBroker (and even the `MidasPageProducer` and hence the InternetExpress XML stuff), which gives us enough support to make it worthwhile considering ASP support as a serious feature of Delphi 5.

See you next time with more Active Server Objects, *so stay tuned...*

Bob Swart (aka Dr.Bob, www.drbob42.com) is an @-Consultant, Delphi 'Clinic' Trainer and co-founder of the Delphi OplossingsCentrum of TAS Advanced Technologies (www.tas-at.com) in The Netherlands.