Under Construction:
Delphi 5 Active Server Objects, 2

by Bob Swart

This time, we'll continue from last month and finish our coverage of Active Server Pages and Objects with Delphi 5. We'll examine the way in which WebBroker and InternetExpress components can be used by an Active Server Object, and finally see some semi-debugging techniques.

Request

Last month, we used the Active Server Object wizard to create an ASP Object. We then used the Type Library Editor to add a method named `ASProduce` to it, which produced some dynamic HTML. We then saw how to deploy and test Active Server Objects inside Active Server Pages. What we didn't examine was how Active Server Pages are most often called: as the results of HTML input forms (also sometimes called CGI forms), handling input as well as producing output.

We did explore the internal ASP objects `Response` and `Session`. This time, we start with the `Request` object, which is also an internal ASP object, and available to our Active Server Object as a property. `Request` can get input in three different ways: using the CGI input form (the form variables), using the 'fat' URL (the query string variables) and finally using cookies. For each of these, there is a collection of items (containing name-value pairs) that we can use, namely `Form`, `QueryString` and `Cookies`. Each of these three properties is of type `IRequestDictionary`: an interface which in turn contains a (default) `Items` array property that we can supply with an `OleVariant` name to get an `OleVariant` value back. In code, to get the value of the form variable `Name`, we'd have to look at `Request.Form.Items['Name']` or (since it is the default property) simply `Request.Form['Name']`. Similarly, to get the value of the cookie with the name `Chocolate`, we'd have to look at `Request.Cookies['Chocolate']`. As a reminder: these string names that we pass as the 'index' for the array parameter are case insensitive, so `Request.Form['Name']` returns the same as `Request.Form['name']`.

Example DrBob42.HTM

Let's make an example HTML input form, so we can experiment with the different ways of getting input. For this example, I'll use a simple HTML form with user name and password fields. The HTML syntax is shown in Listing 1. The result can be seen in Figure 1.

Note the password edit box, which shows only asterisks. This input type is very useful, especially in situations where people have to enter sensitive information like passwords, of course. The `ACTION` part of our HTML form calls the drbob42.asp file that we made last time, we only have to modify the `ASProduce` method to obtain the input data using the ASP `Request` object: see Listing 2.

Since we're using the `POST` method, we need to use `Request.Form` to obtain the values of the `UserName` and `Password` variables. We can then present them back in the browser using the `Response.Write` method.

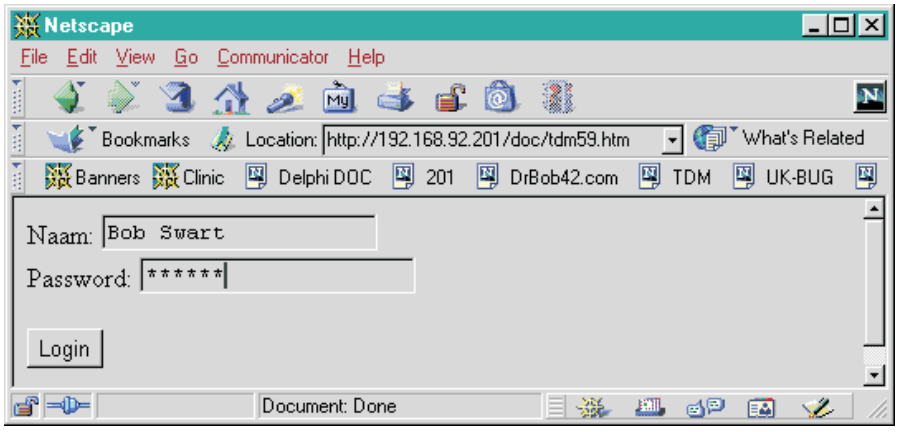
This concludes the built-in support of internal ASP objects that Delphi 5 offers. Using the `Request` we can get any input, and using `Response.Write` we can produce any output.

A more interesting question now is how we can extend this simple `Write` method by looking at the several HTML-producing components that already exist in Delphi. Yes, I'm talking about the WebBroker and InternetExpress components here. Let's see how these can be combined with an Active Server Object.

➤ *Listing 1*

```
<HTML>
<BODY>
<FORM ACTION="http://192.168.92.201/doc/drbob42.asp" METHOD=POST>
Naam: <INPUT TYPE=edit NAME="UserName">
<BR>
Password: <INPUT TYPE=password NAME="Password">
<P>
<INPUT TYPE=submit Value="Login">
</FORM>
</BODY>
</HTML>
```

➤ *Figure 1*



Producing HTML

First of all, there are at least two ways in which we can use WebBroker components (such as a `TPageProducer`). We could create and use them dynamically, or we could use a web module, drop them on it and configure everything at design-time. While the former would probably work fine for a simple `TPageProducer`, it gets more complex once we start talking about a `TDataSetTableProducer` (which needs an additional `TDataSet` component, as well as numerous settings that are really best applied at design-time). So, let's skip the dynamic component creation.

The alternative, on the other hand, isn't really directly applicable either. We can only get a web module as part of a WebBroker project, CGI or ISAPI. Of course, you can 'fake' this by creating a WebBroker project (CGI or ISAPI) and then start an ActiveX library and Active Server Object and share the web module between the WebBroker and ActiveX projects. This is actually quite useful, as we've already seen that it is hard to debug Active Server Objects. And once you share a web module between an ISAPI DLL and an ActiveX (ASO) project, you can at least debug the ISAPI DLL using good old IntraBob, for example.

Multi-Threading

Say we have a unit with a web module that we share between an ISAPI DLL and the ASP Object. There's one thing that the WebBroker takes care of for us behind the scenes, and that's management of requests and web module instances in a queue. When writing a WebBroker application, we don't have to concern ourselves with the possibility of multiple requests working with the same web module. In fact, we just get a number of web module instances and, for every incoming request, one of these instances is 'activated' with that particular request.

With Active Server Objects, there is no such queue. This means we have to create our own web module (or data module, as we'll see in a moment). But if we add one global web module to the project, we would be in trouble, as multiple instances of the Active Server Object would all access the same web module instance, which would guarantee some nice multi-threading clashes (if not database session conflicts first). Apart from those potential problems, you'll find that it isn't easy to add a web module to an existing project (just about the only way is to 'borrow' a web module unit from an existing CGI or ISAPI WebBroker project, which is indeed what I often do).

A more sensible way to start, if you don't want to include other WebBroker targets, is to do `File | New` and add a data module to your project. Normally, you would also have to make sure you have unset the `Auto create forms` option in the `Tools | Environment Options | Preferences` dialog, but it doesn't seem to matter inside ActiveX libraries (no forms or data modules are autocreated anyway). As a consequence, we have to create and destroy our own instance of the data module. When? Well, I think there's no better moment to make use of the `OnStartPage` and `OnEndPage` methods, as shown in Listing 3.

Note that we also needed to add a private member field `DataMod` of type `TDataModule2` (or whatever the type is of your data module). This technique will ensure that every Active Server Page will be served by a unique instance of the data module. Of course, when using BDE databases and tables on this data module, we still need to add a `TSession` component and set `AutoSessionName` to `True` to avoid session conflicts, but at least we can access the components on the data module in a thread-safe way.

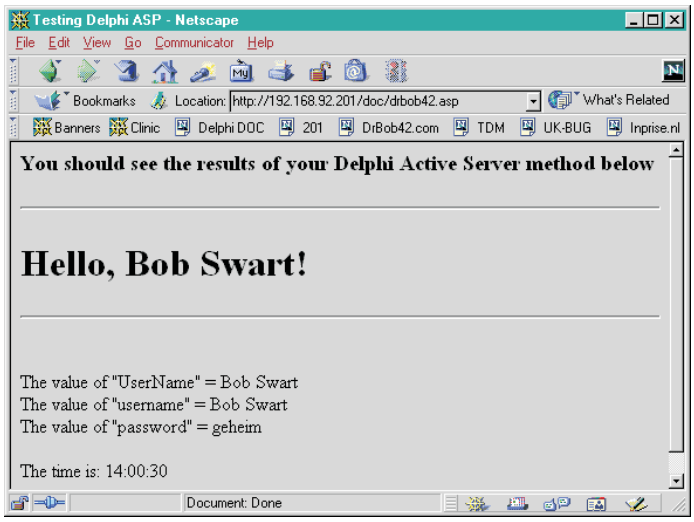
```
procedure TDrBob42.ASProduce;
var
  Name,Pass: String;
begin
  Name := Request.Form['UserName'];
  Response.Write('<H1>Hello, '+Name+'!</H1>');
  Response.Write('<HR>');
  Response.Write('<BR>The value of "UserName" = '+Name);
  Name := Request.Form['username'];
  Response.Write('<BR>The value of "username" = '+Name);
  Pass := Request.Form['password'];
  Response.Write('<BR>The value of "password" = '+Pass);
  Response.Write('<P>');
  Response.Write('The time is: '+TimeToStr(Now));
end;
```

➤ *Above: Listing 2*

➤ *Below: Listing 3*

```
procedure TDrBob42.OnStartPage(const AScriptingContext: IUnknown);
{ DataMod is a private memberfield of type TDataModule2 }
begin
  inherited OnStartPage(AScriptingContext);
  DataMod := TDataModule2.Create(nil);
end;
procedure TDrBob42.OnEndPage;
begin
  FreeAndNil(DataMod);
  inherited OnEndPage;
end;
```

➤ *Figure 2*



ASP PageProducing

Once we have this data module, we can add all kinds of HTML producing components to it. For example the `TPageProducer` or `TDataSetPageProducer` (to keep it simple at first).

Just drop a `TPageProducer` component (from the `Internet` tab) on the new data module you've just created. As usual, we can enter some HTML content at design-time to the `HTMLDoc` property (of type `Strings`) or we can point the `HTMLFile` property to an external file. For the purpose of this demo, I always use the `HTMLDoc` property, but in real life the advantage of using the `HTMLFile` property is that it allows you to potentially change the entire look and feel of that particular HTML fragment without having to recompile (and also redeploy!) your web server application. And if it's one thing we learned last month, then it's that Active Server Objects are hard to re-deploy, as it means shutting down your IIS Admin Manager (or the particular virtual directory, as I found out later, but that's no easy task either). Anyway, for this demo we just keep using the `HTMLDoc` property and enter the following lines to it:

```
<FONT FACE="Verdana"SIZE=2>
Hello, ASP
<P>
Today is:
  <#DATE FORMAT=YYYY/MM/DD>
<P>
<HR>
```

There is one so-called #-tag, `DATE`, which even has a parameter `FORMAT`. Currently, `FORMAT` is set to `YYYY/MM/DD` which leads to a useful date representation (the only one that you can sort easily, to mention the most obvious advantage).

The `OnHTMLTag` event handler for the `PageProducer` component has to work correctly with or without a `FORMAT` parameter present (in `TagParams`): see Listing 4.

➤ *Listing 4*

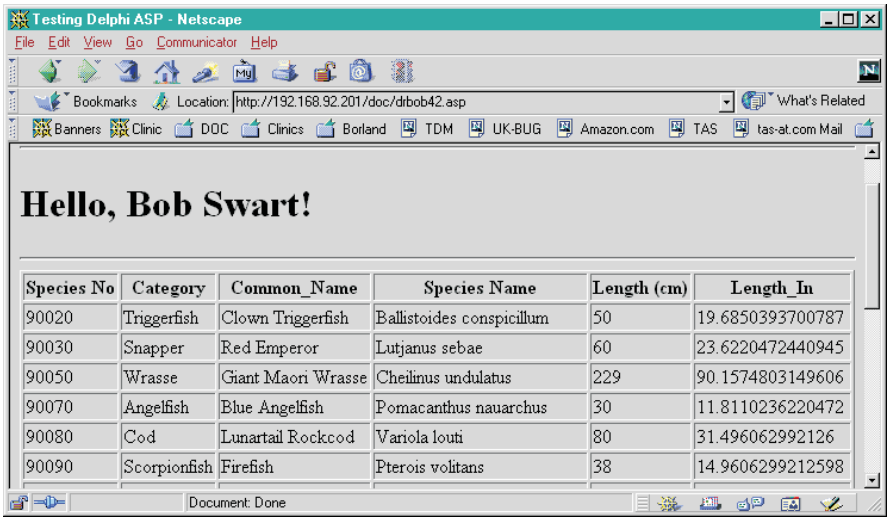
```
procedure TDataModule2.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
  const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
  if TagString = 'DATE' then
    if TagParams.Values['FORMAT'] <> '' then
      ReplaceText := FormatDateTime(TagParams.Values['FORMAT'],Date)
    else
      ReplaceText := DateToStr(Date)
end;
```

```
procedure TDrBob42.ASProduce;
var
  Name: String;
begin
  Name := Request.Form['UserName'];
  Response.Write('<H1>Hello, '+Name+'!</H1>');
  Response.Write('<HR>');
  Response.Write(DataMod.PageProducer1.Content); // WebBroker!
  Response.Write('<P>The time is: '+TimeToStr(Now));
end;
```

➤ *Above: Listing 5*

➤ *Below: Listing 6*

```
procedure TDrBob42.ASProduce;
var Name: String;
begin
  Name := Request.Form['UserName'];
  Response.Write('<H1>Hello, '+Name+'!</H1>');
  Response.Write('<HR>');
  Response.Write(DataMod.DataSetTableProducer1.Content); // WebBroker!
end;
```



➤ *Figure 3*

Only one question remains: how is the `PageProducer` used by our Active Server Object? Well, this is done in the same `ASProduce` method that we've used before. Only this time, we need to 'call' the `Content` property of the `PageProducer1` from the `DataMod`, and pass the `String` result to the `Response.Write` method. The code for this is is shown in Listing 5.

The result is as can be expected, and merely proves that we can call the HTML page producing components' `Content` property from anywhere.

TDataSetTableProducer

A similar result can be obtained using a `TDataSetTableProducer` component. And, in fact, since we often need to visually manipulate the `Columns` property of the `TDataSetTableProducer` at design-time, it's important not to have to create it dynamically!

Just to demonstrate how it all works, drop a `TQuery` on the data module, set the `DatabaseName` to `DBDEMOS` and write the following query in the `SQL` property:

```
SELECT * FROM BIOLIFE.DB
```

Double click on the `Active` property to open the query (if we made no mistakes). Now we can click on the ellipsis next to the `Columns` property in order to work on the output. Make sure to add all fields you want to see. Finally, the `ASProduce` method needs only

```
unit DMQueryTableProducer;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, DBWeb, DB, Asptlb;
type
  TDMQueryTableProducer = class(TQueryTableProducer)
  private
    FRequest: IRequest;
  public
    constructor Create(AOwner: TComponent); override;
    function Content: String; override;
  published
    property Request: IRequest read FRequest write FRequest;
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents('DrBob42', [TDMQueryTableProducer]);
end;
{ TDMQueryTableProducer }
constructor TDMQueryTableProducer.Create(AOwner: TComponent);
begin
  inherited;
  FRequest := nil;
end;
function TDMQueryTableProducer.Content: String;
var
  i: Integer;
begin
  Result := '';
  if Query <> nil then begin
    Query.Close;
    if Assigned(FRequest) then
      for i:=0 to Pred(Query.ParamCount) do
        Query.Params[i].Value :=
          FRequest.Item[Query.Params[i].Name];
    Query.Open;
    if DoCreateContent then
      Result := Header.Text +
        HTMLTable(Query, Self, MaxRows) + Footer.Text
  end
end {Content};
end.
```

➤ *Listing 7*

minor changes to show the output of the `TDataSetTableProducer` (see Listing 6). The result looks just great (Figure 3).

TQueryTableProducer

Now that we've seen how to use the output of a `TDataSetTableProducer`, let's take it one step further: to the `TQueryTableProducer` (which is why we've been using a `TQuery` so far and not a `TTable` component). The `Query TableProducer` is the most advanced of the WebBroker components, as it connects the input fields (found in the `Request`) to the `Query` parameters. For example, we can modify the above query to work with a parameter, by specifying the following SQL query:

```
SELECT * FROM BIOLIFE.DB AS B
  WHERE (B."Length_in" >= :LEN)
```

to specify that we're looking for fish that have a length of `LEN` inches or more. Note that before we can activate (open) this query, we first have to specify the parameter information for `LEN` (`DataType` is `ftInteger`, `ParamType` is `ptInput`, and as default `Value` we can assign it to the integer value 7). As soon as we open the parameterised query, we can connect it to a `TQueryTableProducer` and use the columns editor as before to shape the appearance of the HTML output just as you want.

The only problem is that `TQueryTableProducer` relies on its `Dispatcher` property (of type `TCustomWebDispatcher`) to provide the `Request` of type `TWebRequest`. Without a `Dispatcher`, there is no way to obtain the `Request` values that can be matched to the query parameters. We could add a `WebDispatcher` component to the data module, turning it effectively into a web module (or use a web module from the start). However, the web dispatcher will never be actually dispatched: we're inside an Active Server Object, remember? So the `Request` object of type `TWebRequest` will never be filled with name-value pairs, and a regular `TQueryTableProducer` will not be able to match its input values with the `Query` parameters.

ASO QueryTableProducer

Dr.Bob to the rescue! What we need is a way for the `TQueryTableProducer` to get to the Active Server Object `IRequest` interface instead of the WebBroker `Request` object. And while we can't just change the source code of the `TQueryTableProducer`, we can derive a new component from it, called `TDMQueryTableProducer` and override the function `Content` in which the actual matching of parameters and production of HTML takes place. Apart from this new `Content` function, we also need a new property `Request` of type `IRequest`, that's right, the new `TDMQueryTableProducer` will get a pointer to

```
procedure TDrBob42.ASProduce;
var Name: String;
begin
  Name := Request.Form['LEN'];
  Response.Write('<H1>Hello, '+Name+'!</H1>');
  Response.Write('<HR>');
  Response.Write(DataMod.MidasPageProducer1.Content); // InternetExpress!
end;
```

➤ *Listing 8*

the `IRequest` interface from its own Active Server Object. And once inside the `Content` function, it's easy to walk through the parameters of the query and for each one try to find a matching value inside the request (just by assigning the `Request.Item` of the `Query.ParamName` to the `Query.ParamValue`). And this, lo and behold, works like a charm!

The source code for the new `TDMQueryTableProducer` component can be seen in Listing 7.

Just install this component on your component palette (maybe on the `Internet` tab instead of the `DrBob42` tab, but that's your own choice), and you're ready to go. Active Server Objects utilising the full power of the WebBroker components. Just one more step and we're ready to take on InternetExpress...

Producing XML

So far, we've seen how to use the HTML producing components from WebBroker. Apart from Active Server support, Delphi 5 introduced another big feature named InternetExpress (the combined force of WebBroker and MIDAS), including a `MidasPageProducer` that produced (D)HTML and XML, which was displayed

using a number of additional JavaScript files. It would be very nice if we could use the `MidasPageProducer` inside our Active Server Object as well, so let's sit back for our last big experiment this month.

Drop a `DataSetProvider` component on the data module and connect its `dataset` property to the `Query` component (we won't be needing a truly multi-tier solution, we've already seen in *The Delphi Magazine* Issues 51 and 52 that a standalone InternetExpress solution works just fine). Now drop an `XMLBroker` and `MidasPageProducer` component. Set the `XMLBroker.ProviderName` to `DataSetProvider1` and double click on `Connected` to make sure data is flowing from the `Query` via the `DataSetProvider` to the `XMLBroker`.

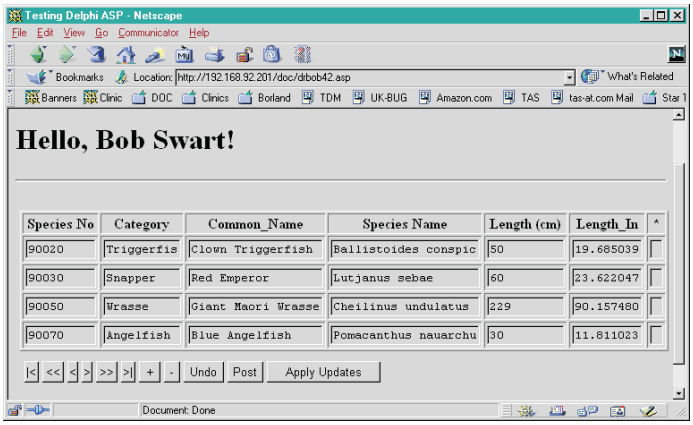
Now, right click on the `MidasPageProducer` to get to the Web Page Editor. In here, right click again to add a `DataForm`, and as sub-components add a `DataGrid` with `DataNavigator` component. Set the `XMLComponent` property of `DataNavigator` to the `DataGrid` and make sure the `DataGrid`'s `XMLBroker` property points to `XMLBroker1`, so we can see the meta data shaping the grid (the grid columns and their titles appear). We're not done yet, because we need to right click on the `DataGrid` to add all the fields we want (remember: if we don't explicitly add any fields here, we won't see any at runtime either). I just select all the fields except for `Memo` and `Graphic`.

Now, close the Web Page Editor, and return to the `MidasPageProducer`. Inside its `IncludePathURL` property we must specify where the JavaScript files can be found (see the SOURCE\WEB MIDAS directory of your Delphi 5 installation for these files).

At this time, we're ready to return to the Active Server Object, specifically the `ASProduce` method, and make use of the `MidasPageProducer`, which can be done as shown in Listing 8.

The result looks and feels fantastic (Figure 4): interactive data inside the web browser, using XML to contain the data, and all this generated by a `MidasPageProducer` used by an Active Server Object.

There's one big question: like the `QueryTableProducer`, which



➤ *Figure 4*

required its input from the `Request` class (and not the `IRequest` class), the `Apply Updates` button also communicates with the world behind (beyond?) the browser. `Apply Updates` normally connects to the WebBroker application and sends the delta data packets so the database can be updated and/or a reconcile error page can be shown. In this case, the Active Server Object has no 'action' item that can be called, and neither did we specify a special 'update' method.

To tell you the truth, currently I have no idea how to invoke the Active Server Object when you click on the `Apply Updates` button. So, I've decided to take the easy road and (for now at least) declare the use of InternetExpress within Active Server Objects to be read-only. As a consequence, we can remove the insert, delete, undo, post and, obviously, apply updates button and focus on navigating only. You can even set the fields to be read-only (although this doesn't improve their readability inside Internet Explorer, so you have to decide for yourself if you want to do that, I don't).

I leave it as an exercise for the reader to derive a new `QueryMidasPageProducer` that will connect the input field `Len` to the query parameter `Len`, just like the regular `QueryPageProducer`. If you're not sure how to do it, check my website (at www.drbob42.com) where I plan to present this component in an upcoming *Dr.Bob Examines...* column.

Debugging Support For ASO

Another item that I promised you for this month was a little bit of enhanced debugging support for Active Server Objects.

They are really hard to debug, believe me, and the text on pages 49-4 and 49-5 from the *Delphi 5 Developer's Guide* is, I'm afraid, just plain wrong. It doesn't work if you try to set your web server as the host application as for ISAPI DLLs, since the Active Server Object is loaded by the ASP.DLL ISAPI DLL, and not by your web server. Oh well, it was too good to be true anyway.

Fortunately, there are a number of methods you can still use to show the progress and other things of your Active Server Pages. Message boxes, for example, are always helpful. Personally, I'm very fond of CodeSite from Raize Software (www.raize.com). This is a two-part debugging aid that comes with a CodeSite message viewer and a CodeSite object that you can use to send just about everything to the viewer. The only important thing you have to do when you start to debug Active Server Objects this way is to make sure they can actually 'talk' to the desktop (or the CodeSite message viewer for that matter), otherwise a messagebox or CodeSite method will have no effect at all. The trick is to go to the `Control Panel`, select the `Services` applet and then go to the `IIS Admin Service` (the one that you have to unload to free your Active Server Object from memory, remember?). Select this service, click on `startup` and make sure the `allow service to interact with desktop` option is selected. This works for NT4 Server with SP5 and IIS4 as the web server.

ASP 3.0 And MTS

Finally, I want to show you that, for Delphi 5 developers, the difference between ASP 2.0 Objects and ASP 3.0 Objects is not that much. So far, we've only seen ASP 2.0 Objects, using the `OnStartPage/OnEndPage` event-level methods. Windows 2000 and IIS 5.0 now support ASP 3.0, which uses an Object Context and MTS to handle much of the internal implementation details. The base class `TASPObject` for ASP 2.0 is replaced by `TASPMTSObject` for ASP 3.0. Apart from that, however, the 'usage' interface for Delphi 5 developers is almost 100% identical. We again have `Request`, `Response`, `Session`, `Application` and `Server` objects, and we still communicate using `Form`, `QueryString` and `Cookies`, and the `Response.Write` method. The main difference is that with ASP 3.0 we use MTS behind the scenes, which of course means that the target machine must have a working MTS configuration! Apart from that, the only Delphi 5 coding difference is that an ASP 3.0 Object does not have the `OnStartPage` and `OnEndPage` events (so you have to create your data module inside the constructor, for example).

Conclusions

Let's look back at what we've learned this time and last time. First of all, Delphi 5 contains 'basic' support for Active Server Objects, with the best part being that it shields us from most of the ASP 2.0 and ASP 3.0 differences. Second, we can use the existing WebBroker components as additional HTML producing aids. However, when it comes to combining ASP with InternetExpress we're stuck with read-only support (at least for now, until I've found a way to connect the `Apply Updates` button back to the `XMLBroker` component).

Finally, we've seen that although Active Server Objects are really difficult to debug, we can make sure they can interact with the desktop so we can use old fashioned debugging techniques like messagebox and other helpful debugging tools like CodeSite.

Next Time

Next time we'll continue with the MIDAS 3 coverage (from Issue 57) and turn our attention to Object Pooling (of remote data modules) and Object Brokering (of connections): two techniques in MIDAS 3 that are very interesting, to say the least. We'll see what these concepts mean and how we can use them to our advantage, *so stay tuned...*

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