Bitten by the ASP

James W. Cooper

In the March issue, I summarized some major, apparent differences between JavaServer Pages and Active Server Pages. Since then, a number of readers have told me that my summary was incomplete, incorrect, or even dead wrong. There is nothing more exhilarating or more humbling than to make statements in the column of a national magazine and have readers tell you their opinions of your work. We got a number of letters, and even though I have to admit that I made some oversimplifications, I was also pretty pleased to know that our readers will always keep me honest. Thanks for telling me what I didn't write right and thanks for taking the time to write.

The letters we got pointed out that there was a lot more to Active Server Pages than my short summary indicated. Some gently suggested the error of my ways and others bluntly suggested that I was totally half-ASPed.

Quite a few letters made the same two points, although they were individually written, the repetition of these 2 points was quite a coincidence.

1. Since you can include ASP files in other ASP files, this is the same as being able to call external subroutines.
2. Since you can call functions in DLLs from ASPs, this is quite analogous to having the ability to using the methods in a Java Bean to carry out computations in user input data.

Let's take a few minutes and look at JSPs and ASPs once again to try to understand their comparative features.

How Server Pages Work

When you fill out an HTML form, you tell it what system to call to get the proper response to your form request. For example, you might call an ordinary CGI-BIN procedure, frequently written in perl or in C to process the form fields and compute a response HTML page. These programs actually write the HTML data stream that you receive from the beginning <html> tag to the ending </html> tag. This has worked well for years but is a lot of trouble to write and to maintain.

Microsoft developed Active Server Pages (ASPs) a few years ago to simplify this process for the HTML programmer. In doing so, they developed a server page language that looked a lot like Visual Basic, that was never seen by the browser, but was instead interpreted on the server so that the result was computed HTML output. For example, the archtypical "Hello world" ASP looks like this:

```
<% @LANGUAGE = VBScript %>
<html>
<head><title>Simple ASP Program</title></head>
<h1 align="center">Simple ASP Program</h1>
<% Response.write "Hi there" %>
</body></html>
```

Note that there is only one actual language statement

```
<% Response.write "Hi there" %>
```

and that it is enclosed in HTML brackets and percent signs. This is a signal to the ASP processing DLL to interpret and execute that code to produce an output page. We call these ASP program pages by specifying them in the “action=” tag on an input form:

```
<html><head>
<title>Simple ASP Program</title>
</head>
<h1 align="center">Simple ASP Program</h1>

<form action=hiback1.asp method=POST>
Enter your name:
  <input type="text" name="Name" width=20><br>
<p><input type="submit" value="Submit" ></p>
</form>
</body></html>
```

Note that there is an input field called “Name” and that this form is sent to the hiback1.asp program for processing.

In that hiback1.asp program, we take the Name field from the input form and send it back in the output form:

```
<%
strName = Request.Form("Name")
Response.write "Hi there " & strName
%>
```

JavaServer Pages

JavaServer pages look very similar to ASPs. In fact Sun and its partners clearly learned a lot from the evolution of ASPs and tried to improve on the model, since they had the advantage of hindsight.

When you create a JavaServer page it always relates to a server bean class that catches the contents of the various form fields. This is a non-visual bean that is just a master Java class with a group of set and get methods. There can be any number of associated classes in the bean, and several pages can refer to the same bean. However, for every form element named xyz there should be a setXyz and a getXyz method in that associated bean. When you submit a JSP, these methods are called automatically, and the output JSP can use them to construct a response. In the JSP below the bottom of the page is filled in from the results of calling the server bean called mine.Jtest.

```
<jsp:useBean id="mine" scope = "page" class="mine.JTest" />
<jsp:setProperty name="mine" property="*"/>
<html><head><title>Hi</title></head>
<body>
<form method="post">
<input type="text" name="entryText" size="25"><br>
<input type="submit" value="submit">
</form>

<% if (mine.getEntryText() != null) { %>
 Hello there: <%= mine.getEntryText() %>
<% } %>
```

```
</body>
</html>
```

and the bean itself just contains the appropriate set and get methods:

```
//A simple Java bean for the Hi program
public class JTest {
   private String text; //text stored here

   public JTest() {
      text = null;            //initialize
   }
   //set text here
   public void setEntryText(String t) {
      text = t;
   }
   //get it back here
   public String getEntryText() {
       return text;
   }
}
```

First Level Comparisons

So how do we compare these two approaches? Well, at first they seem pretty similar in some ways. The server page language elements are quite similar. The difference is that while *all* JSPs have a related bean on the server whose get and set methods are called, this is not required by ASPs. This can be merely an implementation difference or it can be rather significant. Any page that interacts with a bean can use the bean as a repository of information between pages and thus provide a kind of persistence not directly available in ASPs. There are some important additional caveats to persistence we'll discuss later.

It is important to note that ASPs are interpreted when they are executed and that JSPs are interpreted and then compiled to Java servlets. The upshot is that the first time anyone hits an ASP he will get a fairly rapid response. The first user of a JSP will see a rather significant pause while the JSP is compiled and loaded. Future users will get much more rapid response, however.

Programming Style Differences

The programming style that ASPs encourage is one in which each page points to the next and passes parameters to it. For JSPs, on the other hand, the programmer is encouraged to start with a single page where various parts of the page are executed conditionally depending on the state of variables set in the related bean. Neither of these styles is required, however, and you can easily program around these assumptions to make the pages do whatever you want.

Calling Subroutines in Include Files

One somewhat overarching generality I included in the March column was that JSPs could call external subroutines but that ASPs could not. Many correspondents quickly pointed out that you can include one ASP file in another and thus that ASPs could also

call external programs. That's true, of course, and I use ASP include files all the time to include a common function or a common header or footer. In JSPs, you write

```
<@ include file="footer.jsp" %>
```

and in ASPs you use a comment-style structure:

```
<!-- #include file="footer.inc" -->
```

Include files are typically used for headers and footers and to include common subroutines, but they are not really the same as having the ability to call external subroutines. In the case of ASPs, they become part of the same interpreter image and each page that includes that file interprets it separately. In the competitive advantage race, this is a draw, and the feature is not really that significant in either system.

Calling External Subroutines

JSPs can call routines in the beans associated with each page. Usually, you can refer to a bean in one or more pages, but it is possible to have segments of a page each refer to different beans. Beans provide a convenient repository for computation and database access methods.

You can also access external routines from ASPs, but it is more involved and a little less type safe. You do this by writing an ActiveX DLL (also called a COM object) and call the methods in that object from any ASP. The coupling is not as tight is between as JSP and its bean, but you can use it in a similar way. Let's write the same program in both systems and compare the implementation details.

The Problem

Suppose we need to write a program to recognize a name entered on a form. Your user can enter her name either as

```
lastname, firstname
```

or as

```
firstname lastname
```

We'll show below how to write this same program as a JSP and Bean or as an ASP with an associated COM object.

The JSP Solution

Clearly, if these are the only possibilities, we can write a program to look for a comma or a space and determine the names easily. However, if there might be more two cases we might find it useful to define a base Namer class and then write two subclasses:

```
package Names;
//A simple base class for both orders of names
public class Namer {
   //store the names here
  protected String first, last;
```

```
  //return the first name
  public String getFirst() {
      return first;
  }
  //return the last name
  public String getLast() {
      return last;
  }
}
```

The FirstFirst subclass just separates the names at the space and stores them in the protected first and last variables where the inherited getFirst and getLast methods can retrieve them:

```
package Names;
public class FirstFirst extends Namer {
    //separates the leading first name
    public FirstFirst(String s) {
        int i = s.lastIndexOf (" ");
        if(i > 0) {
            first = s.substring(0, i).trim();
            last = s.substring(i + 1).trim();
        }
        else {
            first = "";
            last = s;
        }
    }
}
```

and analogously for the LastFirst class

```
package Names;
public class LastFirst extends Namer {
   //makes the last name all to the left of the first comma
   public LastFirst(String s) {
        int i = s.lastIndexOf (",");
        if(i > 0) {
            last = s.substring(0, i).trim();
            first = s.substring(i + 1).trim();
        }
        else {
            first = "";
            last = s;
        }
    }
}
```

Then, the Bean that drives all this is extremely simple. It selects one of the two subclasses of Namer and instantiates it. It then uses that instance to return the first and last names separately on request.

```
package Names;
public class NamingBean {
    private Namer namer = null;

    //This selects one of two Namer subclasses
    public void setName(String nm) {
```

```
        int i =nm.indexOf (",");
        if( i >0 )
            namer = new LastFirst(nm);
        else
            namer = new FirstFirst(nm);
    }
    //return the last name
    public String getLast() {
       return namer.getLast ();
    }
    //return the first name
    public String getFirst() {
       return namer.getFirst ();
    }
    //get the instance of the Namer class
    public Namer getNamer() {
        return namer;
    }
}
```

The setName method in the above bean is called automatically when your user clicks on the Submit button and sends the data to the server. Then, the bean instantiates one of the two derived Namer classes. This is an example of the Factory design pattern and is one of the advantages of programming in a language like Java.

The ASP Solution

Now lets consider the same problem using an ASP. We can write the underlying ActiveX DLL (or COM object) in Visual Basic, or in C or C++, but for simplicity, we'll write this one in Visual Basic. Many, if not most COM objects for ASPs are written in VB because of its superior development environment and ease of use. VB does not support inheritance, but it does support interfaces, and we can define VB interface Namer as follows:

```
'Interface class Namer
Public Sub init(nm$)
End Sub
'-------------------
Public Function getFirst() As String
End Function
'-------------------
Public Function getLast() As String
End Function
```

Then we can write two implementations of that interface in much the same way as we did in Java. Here is the FirstFirst class. The LastFirst class is entirely analogous.

```
'Class FirstFirst
Implements Namer
Private lname As String
Private frname As String
'-------------------
Private Function Namer_getFirst() As String
 Namer_getFirst = frname
```

```
End Function
'-------------------
Private Function Namer_getLast() As String
 Namer_getLast = lname
End Function
'-------------------
Private Sub Namer_init(nm As String)
Dim i As Integer
i = InStr(nm, " ")
 If i > 0 Then
  frname = Trim(Left$(nm, i - 1))
  lname = Trim$(Right(nm, Len(nm) - i))
 Else
  lname = nm
  frname = ""
 End If
End Sub
```

The main class in this program will be the factory that instantiates one of these classes and uses it to return the first and last names:

```
'Class NameSplit
Private Nmr As Namer
'-------------------
Public Sub setName(nm)
Dim i As Integer
Dim s As String
s = nm
i = InStr(s, ",")
 If i > 0 Then
  Set Nmr = New LastFirst   'instantiate last first class
 Else
  Set Nmr = New FirstFirst  'or first first class
 End If
Nmr.init s  'and pass it the name to be split
End Sub
'-------------------
Public Function getLast() As String
 getLast = Nmr.getLast
End Function
'-------------------
Public Function getFirst() As String
 getFirst = Nmr.getFirst
End Function
```

After we have written this program, we tell VB to compile it as an ActiveX DLL. This also automatically registers the DLL with the Windows Registry so it can be found by any calling programs.

The calling ASP asks for an instance of the NameSplit class from the NameFunct DLL

```
set fName = Server.CreateObject("NameFunct.NameSplit")
```

and uses it to split the name, as we show below.

```
<% @LANGUAGE = VBScript %>
```

```
<%
Option Explicit
Dim strName, fName, gotName
%>

<html>
<head>
<title>Simple ASP Program Using ActiveX DLL</title>
</head>

<h1 align="center">Simple ASP Program Using COM Object</h1>
<%
strName = Request.Form("Name")
set fName = Server.CreateObject("NameFunct.NameSplit")
fName.setName strName
gotName = fName.getFirst
Response.write "First name: " & gotName
gotName = fName.getLast
Response.write "<p>Last name: " & gotName
%>
</body></html>
```

Differences and Similarities

We can see from these two elementary examples that there are a number of similarities and differences here. The JSP is compiled into a Java servlet and its Bean is loaded into the same JVM as that servlet, guaranteeing type compatibility and good performance. The ActiveX DLL is a rather separate entity where all variables are passed from the interpreted ASP to the DLL as of type Variant with no compile-time type checking. This has the possibility of leading to greater error.

Further, if you compile the DLL on the same machine you use as a server, the DLL is registered with Windows automatically. If you need to move that DLL to another server, you will need to run the regsvr32.exe program there to register that DLL. If you do not have execute access to the server machine, you need to use a much longer invocation of the CreateObject method, which includes the long registry number generated during the compile step. This is a bit error prone, I'm afraid, but you can do it.

So, to summarize this section: Yes, both JSPs and ASPs can indeed call external subroutines, and these ActiveX DLLs are the external objects ASPs can use. I see them as a bit more disconnected from the server page system than JSPs and beans, but they are commonly used and I should have mentioned them in my first column.

The ActiveX DLLs generated by Visual Basic are single-threaded only. This appears to mean that only one user at a time can get access to the DLL. If you want to use a multi-threaded architecture for larger servers with many users, you have to write your DLL using Visual C++ instead of VB, and specify which of 4 threading models you want to use. JSPs are always multithreaded, with each user getting his own thread.

Further, calling a function in a DLL is not in any way a cross-platform solution, and thus this approach is really only applicable on Windows servers. VB and Visual C++ also provide a third approach called WebClass objects, which seem to provide a more intimate

link between browser-generated events and the server-side data. However, from what I can glean, this approach is neither platform nor browser independent.

Data Persistence

The issue of data persistence is quite simple for small sites using JSPs or ASPs but in both cases can grow into something rather elaborate for large sites. When a user accesses a JSP, a Session object is created automatically where that instance of the bean is stored along with any data that needs to be kept between pages. Remember, though, that HTML communication is stateless and that there must be some feature added somewhere to allow each user to be identified between page submissions. This is done under the covers by the servlet using a cookie and is completely transparent to the user. If the client does not allow cookies, the data does not persist between pages.

There is also a Session object in ASPs which allows you to store both variables and objects in a sort of named hashtable. Here we store the object connecting to the DLL in the Session variable “Ename.”

```
set fName = Server.CreateObject("NameFunct.NameSplit")
set Session("Ename") = fName
```

We can fetch that variable back with

```
set fName = Session("Ename")
```

These data are also actually stored by the ASP using cookies, and if they are not available, you need to devise an alternate solution.

There are, of course, a number of other ways to store data that persists between pages, and they are not ASP or JSP dependent. One is to store the data itself as hidden fields in the xSP page. Another is to store a session key in a hidden field and use it to look up data stored in a database by user session key. Then you are responsible for seeing that it is deleted.

Reader Mike Hodge also pointed out to me that in large systems, with multiple servers, data persistence is even more problematic, since each page may get sent to a different server, thus making any session persistence on the server useless. In this case, you can only provide persistence using one of the alternate solutions I just mentioned.

Some Conclusions

The ASP and JSP approaches have somewhat more in common than my last column indicated. However JSPs still have a couple of advantages. They are cross-platform object-oriented solutions and are not dependent on mysteries like the Windows Registry. JavaBeans are robust and pretty easy to write, the ActiveX DLLs are a much lower level solution that is not cross-platform and if improperly written can be a source of server instability. They are also a bit harder to install remotely.

But the single greatest distinction I’ve found is that you can learn almost all you need to know to write effective JSPs in an afternoon or at least in a day or so. However, there are many more layers to learning the details of using ASPs that can consume quite a bit

longer time. This is probably because ASPs evolved over time from a fairly simple system to one of much more complexity. More than anything else, it is this long learning curve that makes me happy to continue to recommend using JSPs whenever the server system allows it.