XML Where Angels Fear To Tread: SB/MVC on the Server

Michael Corning, Microsoft Corporation

Introduction

This session is a follow-on to "XML for the Criminally Insane: An Introduction to the Schema-Based/Model-View-Controller Design Framework", and moves the SBPNews sample program from that session to the server. The first part of this session describes an ASP version of SBPNews. The second part shows you how to use XSL ISAPI 2 to serve wireless clients as well. Included in the second part is an application that persists data in the Model to the file system, as individual stories used by XSL ISAPI2.

Michael Corning, mcorning@microsoft.com

Michael is a Memetic Engineer on the Application Server Team at Microsoft. Currently, his job is to automate the App Server test process using XML and ASP. Corning's fifteen minutes of fame came from being fortunate enough to have coauthored (with Steve Elfanbaum and David Melnick) the first book on ASP, Working with Active Server Pages (Que, 1997). Since then Corning has written extensively on topics of interest to ASP and XML developers. One European reviewer of his book aptly noted that "the author could scarcely contain his enthusiasm." Corning brings that patented passion to the podium before audiences the world over whenever he gets a chance to speak about software that, he believes, will have a measurable impact on the course of human development.

Fifth Generation Web Sites

I consider fifth generation web sites to be those who use an object model for client-side data. Fifth generation web sites support XML on the client. SB/MVC applications not only use XML to give an object model to their data, they also bring an object model to the application architecture itself. The Model encapsulates data access, the Views provide the user interface, and the Controller is the glue that binds all the parts together.

The SBPNews application, introduced in the first session, illustrates all the significant features of a Schema-Based/Model-View-Controller (SB/MVC) application. SBPNews consists of three files. The largest, `sbpnews.xml`, stores the raw data used by the Model at runtime. `sbpnews.xslt` is the XSLT template that is used once when the client makes its first request to the server; the output of `sbpnews.xslt` is pure xhtml (that is, well-formed html). The Controller uses JScript on the client as the mechanism to mediate between the user and the application, and between the Views and the Model. Since the JScript is kept in the Model, the first thing that `sbpnews.xslt` must do is fetch the script from the Model and strip off the XML that may surround that script. SBPNews uses XML wrapped around script to provide a version control mechanism, a tutorial mechanism, and, most importantly, a way to bind script to events and XSLT transforms that will take data in other parts of the Model and render them as html. The third file comprising the SBPNews sample is the `sbpnews.css` cascading stylesheet file; it controls the appearance of the application.

Basic Downlevel Browser Support

If you're like me, the first thing you thought when you initially saw ASP was that the heartburn caused by maintaining different versions of a web site to support different browser capabilities was a thing of the past. Some things never change, though. Even after all these years, we still have many browsers of differing capabilities to support. The second version of SBPNews, driven by `sbpnews.asp`, is one answer to providing some level of SB/MVC support for browsers that cannot handle XML themselves. This section describes the changes made to the original SBPNews application to make it work on ASP, and the last section describes how we brought in XSL ISAPI 2 to server wireless clients with their tiny screens and tiny bandwidth.

SB/MVC on ASP

A main advantage of true SB/MVC applications is speed. Since all the data is on the client, selections made by the user don't result in a round trip to the server and there is no screen flash while the browser displays the new data. Once ASP is in the picture, this advantage is lost. `sbpnews.asp` still returns html and script to the client, and the Controller's role as mediator between the user and the application remains intact, but the duties of the Controller have changed. Now `sbpnews.asp` assumes part of the role of Controller. The server-side script in `sbpnews.asp` uses the XSLTemplate and XSLProcessor objects to pass parameters to the cached stylesheet (see my article in the September issue of XML Developer for details) that returns html to the client. As a result, subsequent transformations will reflect the intentions of the user. The server-side version of the application also requires other changes to the original `sbpnews.xml` and `sbpnews.xslt` files.

In `sbpnews.xml`, the Model is slightly modified and the `sbpnews.xslt` transform is substantially altered. Since an abbreviated version of the client-side script is sent to downlevel browsers, the XML that surrounds the script in the Model must provide `sbpnews.xslt` with the information it needs to properly filter the script, when the XML file is accessed directly or when the ASP version of the application is called. If an XML file is requested, the full suite of script functions is returned. If the `.asp` file is requested, only the function pointer assignments to the onclick event handler and a single line function to recall `sbpnews.asp` survive transformation.

In the interest of simplicity, we are leaving out any explicit browser capability queries by ASP. We assume that downlevel browsers are using sbpnews.asp and Internet Explorer 5 users are calling sbpnews.xml. When we turn to XSL ISAPI 2 we do so, in part, because browser sniffing comes for free.

The modified client-side script, used when sbpnews.asp is called, maps all html onclick events to the modified **notify()** function:

```
function notify()
{
   location.href="sbpnews.asp?"+_
      "event="+this.event+"&story="+event.srcElement.id;
}
```

As you can see, this function merely calls the sbpnews.asp file, passing in the event and selected story. This is why it's fairly safe that, once a downlevel browser finds sbpnews.asp, it's likely to continue to use it.

The original sbpnews.xslt was rather elegant. The transform basically dumped a little client-side script and some empty html tags on the client. As events (such as startUp) fired, the html picked up data from the Model.

The sbpnews.xslt used by sbpnews.asp is another matter altogether. This modified transform begins with a collection of XSL parameters and variables passed in by sbpnews.asp. These variables and parameters help the stylesheet populate the usually empty html tags with data from the Model.

```
  <xsl:output method="xml" indent="yes" omit-xml-declaration="yes" />
  <xsl:variable name="siteName" select="model/edition/@name" />
  <xsl:param name="event">startUp</xsl:param>
  <xsl:param name="mode">XML</xsl:param>
  <xsl:param name="selectedStory">story01</xsl:param>
  <xsl:variable name="thisStory"
                select="/model/edition/story[@id=$selectedStory]"/>

  <xsl:template match="codeBlock">
    <xsl:if test="@mode=$mode">
      <xsl:apply-templates />
    </xsl:if>
  </xsl:template>
```

The **bold** instructions are the ones that move the script out of the Model and down to the client. This template uses the $mode variable reference to determine whether all the script or merely the modified initialize() and notify() methods are called for.

Here's the <body> tag from the sbpnews.xslt used by sbpnews.asp (note the **bold** text, it's been added to the original sbpnews.xslt file):

```
      <body id="bodyTag">
        <table width="900" border="0">
        <caption id="viewMainHead" title="Click to see Tutorial">
        <xsl:value-of select="$siteName" /></caption>
          <col width="200" /><col width="500" /><col width="200" />
          <tr>
          <td>
```

```
          <h3 id="viewHeadlineHead" event="startUp"
              title="Click to see all Abstracts">Headlines</h3>
          <div id="viewHeadlines" event="headlineSelected">
            <xsl:if test="$event='startUp' or $event='headlineSelected'">
              <xsl:for-each select="model/edition/story/headline">
                <h4 id="{../@id}"><xsl:value-of select="."/></h4>
              </xsl:for-each>
            </xsl:if>
          </div>
        </td>
        <td>
          <h3 id="viewStoryHead" title="Click to see all Abstracts">
          <xsl:choose>
          <xsl:when test="$event='startUp'">
            <xsl:value-of select="/model/@view" />
          </xsl:when>
          <xsl:otherwise>
            Story
          </xsl:otherwise>
          </xsl:choose>
          </h3>
          <div id="viewStory" event="headlineSelected">
          <xsl:choose>
          <xsl:when test="$event='startUp'">
            <xsl:for-each select="model/edition/story">
              <h4 id="{@id}" onmouseover="status=this.getAttribute('id')">
                 <xsl:value-of select="headline"/>
              </h4>
              <div class="abstract"><xsl:value-of select="abstract"/></div>
            </xsl:for-each>
          </xsl:when>
          <xsl:when test="$event='headlineSelected'">
            <xsl:for-each select="$thisStory">
              <h4><xsl:value-of select="headline"/></h4>
              <div class="story"><xsl:copy-of select="content"/></div>
            </xsl:for-each>
          </xsl:when>
          </xsl:choose>
          </div>
        </td>
        <td>
          <div id="viewReferences"></div>
        </td>
        </tr>
        <xsl:copy-of select="model/tutorial[@step='1']" />
        <tr>
          <td colspan="3"><div id="viewTutorial"></div></td>
        </tr>
      </table>
    </body>
```

Basically, that **bold** script populates the normally empty html tags with data from the Model. Each subsequent request uses different data and returns the whole html file to the client (as opposed to the true SB/MVC version that only updates individual html tags, as necessary).

Developing this ASP version of an SB/MVC application bothered me a bit. I saw the familiar and reassuring structure of a pure SB/MVC framework slip a little. There's too much processing going on in `sbpnews.xslt`. I have to keep track of logic in two files (the `.xml` and the `.xslt` files). I also had to move some Controller functionality out to ASP, and that bothered me a bit too. Over time I'll rethink some of these preliminary techniques, probably using two different XSLT files and caching them both in ASP Application scope (we're already using `XSLTemplate` and `XSLProcessor` objects).

My misgivings notwithstanding, if there were no wireless form factors, I could probably get away with this version of SBPNews, but, ever since I bought my first pocketPC a few months ago, I've been coming out of denial. Fortunately, about that time the Adaptive UI team gave us XSL ISAPI 2. Let's see why it's so important.

Before I proceed, a note about sbpnews.asp: I have not fully implemented all the UI behavior in the .asp. If you click the Story heading you will not see all the abstracts reappear as they do when IE 5.x processes sbpnews.xml. I leave that additional work in sbpnews.asp as an exercise for the reader. No sense in me having all the fun.

Dynamic XML Using XSL ISAPI 2

I just completed an article for "Active Web Developer" showing how to use XSL ISAPI 2 to process static XML. That article used SBPNews. To get SBPNews to work on the pocketPC and on cell phones I had to persist each story in the Model (that is, in sbpnews.xml) to the file system. For completeness, I've included here a short .asp file that copies stories from the Model to the file system. Here's that script:

```
<OBJECT ID="Model" PROGID="MSXML2.FreeThreadedDOMDocument"
RUNAT="SERVER"></OBJECT>
<OBJECT ID="story" PROGID="MSXML2.FreeThreadedDOMDocument"
RUNAT="SERVER"></OBJECT>
<OBJECT ID="xslt" PROGID="MSXML2.FreeThreadedDOMDocument"
RUNAT="SERVER"></OBJECT>

<%@ Language=JScript %>
<%

  Model.async=false;
  Model.load(Server.MapPath("sbpNews.xml"));
  var stories=Model.selectNodes("model/edition/story/@id");

  xslt.async=false;
  xslt.load(Server.MapPath("sbpNewsStories.xslt"));

  xslTemplate.stylesheet=xslt;
  xslProc=xslTemplate.createProcessor();
  xslProc.input=Model;
  xslProc.output=story;
  var storyID=null;
  while (storyID=stories.nextNode())
  {
    xslProc.addParameter("story",storyID.text);
    xslProc.transform;
    story.save(Server.MapPath("sbpnews"+storyID.text+".xml"))
  }
  Response.Write("All stories saved successfully.")
%>
```

And here's the sbpNewsStories.xslt transform:

```
<xsl:transform version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >

  <xsl:output method="xml" indent="yes" omit-xml-declaration="yes" />
  <xsl:param name="story">story01</xsl:param>

  <xsl:template match="/">
```

```
  <xsl:processing-instruction name = "xml-stylesheet">
  type="text/xsl" server-config="sbpNewsConfig2.xml"
  </xsl:processing-instruction>

  <xsl:for-each select="model">
    <xsl:copy>
      <xsl:copy-of select="@*"/>
      <xsl:for-each select="edition">
        <xsl:copy>
          <xsl:copy-of select="@*"/>
          <xsl:apply-templates select="story[@id=$story]"/>
        </xsl:copy>
      </xsl:for-each>
    </xsl:copy>
  </xsl:for-each>

  </xsl:template>

  <xsl:template match="story">
    <xsl:copy>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="*">
    <xsl:copy>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="abstract"/>

</xsl:transform>
```

However, for this session, I'm going to use a technique written by Bogdan Popp of the Adaptive UI team. I've modified Bogdan's original design slightly, mainly to make it as simple as possible.

ContentGenerator.pasp

```
<%@ LANGUAGE = VBScript %>

<%

'Content generator PASP file
'
'The purpose of this file is to generate dynamic content from the XML data
'file. The filtered nodes will be part of the web content returned to the
'client and are based on the parameters from the http request.
'For example:
'http://www.myweb.com/contentgenerator.pasp?id=story01&abstract=no&reference=no
'will delete all the "abstract" and "reference" child nodes
'of the story node which has the id attribute equal to story01.
'As the result of the node filtering, we can write XSL stylesheets easily
'for different content generation (see the XSL stylesheets)

  Dim doc, node, sourceFile, storyid, abstract, content, reference, headline
  Dim attrib, tag

  ' Initialize variables with the values from the http request
  storyid=Request.QueryString("id")     ' story ID identifier
  abstract=Request.QueryString("abstract")
```

```
content=Request.QueryString("content")
reference=Request.QueryString("reference")
headline=Request.QueryString("headline")

'Load the XML data into a MSXMLDOM object
sourceFile = Server.MapPath("sbpnews.xml")
Set doc = Server.CreateObject("Microsoft.XMLDOM")
doc.async = False
doc.load(sourceFile)

'Get the collection of all the "story" nodes into a node collection object
' that are not the requested story.
If (storyid<>"") Then
  Set selectedNodes =
    doc.selectNodes("/model/edition/story[@id!='" & storyid & "']")

  For Each node In selectedNodes
    node.parentNode.removeChild node
  Next

End If

Set selectedNodes = doc.selectNodes("/model/edition/story")
'Filtering out the unwanted information (unwanted nodes)
For Each node In selectedNodes
    attrib= node.getAttribute("id")
  If abstract="no" Then
      For Each tag In node.childNodes
        If tag.nodeName="abstract" Then node.removeChild tag
      Next
  End If
  If content="no" Then
    For Each tag In node.childNodes
       If tag.nodeName="content" Then node.removeChild tag
      Next
  End If
  If reference="no" Then
    For Each tag In node.childNodes
       If tag.nodeName="reference" Then node.removeChild tag
      Next
  End If
  If (headline="no")Then
    For Each tag In node.childNodes
       If tag.nodeName="headline" Then node.removeChild tag
      Next
  End If

  'Before processing the dynamically generated XML,
  'we need to create a "link" attribute for every "headline" node.
  'As you can see, we are generating more or less content from the
  'original XML data regarding which device will receive the data.
  'For example: linkWML has the most data filtered out because we
  'can't show the same amount of information on a cell phone compared
  'to a PocketPC or IE on the desktop.

    For Each tag In node.childNodes
      If tag.nodeName="headline" Then
        tag.setAttribute "linkPIE", "contentgenerator.pasp?id= _
             "&attrib&"&abstract=no&content=yes&reference=yes&headlines=yes"
        tag.setAttribute "linkWML", "contentgenerator.pasp?id=
             "&attrib&"&abstract=no&reference=no&headlines=no"
      End If
    Next
Next

'in case you want to have the dynamically generated XML in a file
```

```
  'doc.save(Server.MapPath("resultdata.xml"))

  'the dynamically generated XML data is sent for further processing
  'to the XSLISAPI2 filter
  Response.Write (doc.xml)

%>
```

After I briefly cover how XSL ISAPI 2 works, I'll come back to this script for the highlights.

How does XSL ISAPI 2 work?

XSL ISAPI 2 is an ISAPI filter (not an ISAPI extension because the `Server.Transfer()` method is used by an XSL ISAPI 2 system file, `RedirectorPASP.asp`) that processes any request for `.pasp` files or `.xml` files (unless some other ISAPI extension has claimed it). You can download the XSL ISAPI 2 from http://msdn.microsoft.com/downloads/webtechnology/xml/xslisapi.asp. The setup instructions are clear and simple. Copy the SBPNews sample application into a folder on the vroot that you create for XSL ISAPI 2, and you should have a working system. To confirm everything works with either the pocketPC or a cell phone, download the appropriate sdk. See http://www.microsoft.com/pocketpc/ and http://www.phone.com/index.html for information on obtaining the emulators you can use to simulate running SBPNews on wireless devices.

For complete details on how XSL ISAPI 2 processes dynamic XML, see my series on "Active Web Developer".

How does contentGenerator.pasp work?

The dynamic XML version of SBPNews works when a story request has come from a wireless client with a querystring that specifies which story the reader wants. Once that input data is processed, `contentGenerator.pasp` strips out all story nodes from its in-memory copy of `sbpnews.xml`, except the story node for the requested story.

If no story is requested, then the client is implicitly asking for all story abstracts. In either case, the next thing the script does is create a nodeList of stories. If the reader selected a story, this nodeList has one node in it.

The script then walks this nodeList and keeps only nodes explicitly included in the querystring. This stripped down XML then picks up two new attributes, linkPIE and linkWML. If the client is a pocketPC, the stylesheet assigned in the `sbpnewsconfig.xml` file will then transform the SBPNews XML and will reference the linkPIE attribute, for the URL exposed by the client, so that, when selected, XSL ISAPI 2 will return the correct story content to the client.

Conclusion

This brief summary of dynamic XML covered two ways to get state-of-the-art schema-based programming applications, such as SBPNews, to clients incapable of processing the XML. This has always been important (otherwise you alienate a segment of the Internet community), but it will be vital in the future. Before you might have written off downlevel browsers, rationalizing that attrition will eventually solve your problem for you. But the future of the Internet is wireless and, today, when an XML-challenged client visits your site it's not because the browser's out of date, it's because the client is state-of-the-art.

Besides, if you're like me, the wireless client will probably be you...