Data-Aware HTML Controls, 2

by Steve Troxell

Back in the October issue, we started building a markup language engine that allowed us to add our own tags to standard HTML. This engine provided support for a set of Delphi classes, which we called data objects, to provide database access. This combination gave us a simple language with which we could author data-driven web pages. Our new tags would be dynamically replaced with HTML and data-driven content. Figure 1 shows an example of standard HTML containing three embedded custom tags. It also shows how those tags are replaced and filled with data from a data object.

HTML snippet with embedded SML tags:

```
<BODY>
<H3>Stats for a specific employee:</H3>
<%
<DATAOBJECT class="TEmployee" name="Emp" oid="4"/>
<CONTROL type="edit" property="Emp.FirstName"/>
<CONTROL type="edit" property="Emp.LastName"/>
%>
</BODY>
```

SML tags expanded into HTML with data-driven content:

```
<BODY>
<H3>Stats for a specific employee:</H3>
<INPUT type="text" value="Bruce">
<INPUT type="text" value="Young">
</BODY>
```

➤ *Figure 1*

In Our Last Episode

Let's briefly recap the system architecture. The web pages themselves are authored as standard HTML pages with custom tags embedded in them between `<%` and `%>` delimiters. The custom tags we referred to as SML (*Steve's Markup Language*) just to give them a name. These pages are referred to as page templates because they will be passed through a page generator which will convert the SML tags into the applicable HTML that is desired when the page is requested. The final expanded page is sent back to the browser with data-driven content.

The markup language extensions we've developed are simply a custom set of tags, following XML syntax conventions. We define the tag name, attributes and subtags for whatever functionality we want. In Figure 1, the `<DATAOBJECT>` tag is used to create a reference to a particular data object; the two `<CONTROL>` tags are used to create an HTML control on the page containing data from that object. Since the tag syntax is based on XML, we can use a standard third-party XML parser to interpret the SML tags.

Within our markup language engine, we have a set of tag resolver classes that are responsible for interpreting each SML tag we define. Each individual SML tag maps one-to-one to a class descending from the `TTagResolver` base class we developed last time. For example, the `TDataObjectTag-Resolver` class handles all the processing for the `<DATAOBJECT>` tag. Adding new tags to the markup language is as simple as adding another tag resolver class to the library.

In addition to the tag resolver classes, we also have the data object classes, which provide all direct access to the database. These classes are the interface between the markup language and the database. The data object classes contain all the code to directly access the database, through SQL, a native API, or any other means of data access. In the example in Figure 1, we have a `TEmployee` class which exposes all the data elements for a given employee, such as the employee's first name and last name. The `TEmployee` class is responsible for knowing how to pull those elements out of the database and publish them as properties. Web page authors need not be concerned about the details of the database tables or SQL. They only need to be familiar with the usage of the data objects. This is the great advantage of this system over others such as ASP, iHTML and so on.

All the components of the markup language are contained within an application server that runs alongside the web server. In the actual website, links to pages are written as calls to a CGI program, called the router, with a parameter that identifies the desired page template. The CGI program connects to the application server which then handles all the work of loading the page template, parsing the SML tags, instantiating data objects, and replacing SML with generated HTML. The final page is handed back to the CGI caller, which in turn hands it back to the web browser.

Where We Go From Here

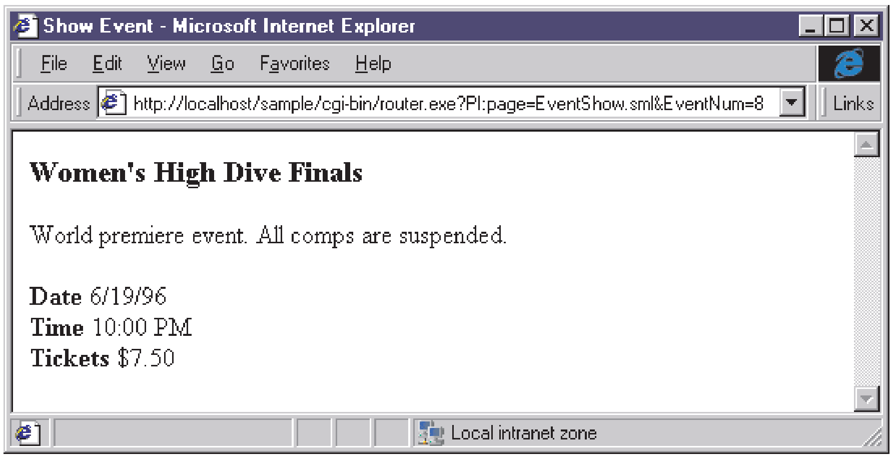
This month, we'll tie together the pieces we started on in October and add support for capturing data entry on the web pages and posting changes in the database through the data objects. We're going to build a live website using our markup language to display and modify the contents of the `DBDEMOS` database. All the actual code for this implementation can be found on this month's disk. Be sure to read the README.TXT file for specific setup instructions before attempting to run the demo software.

Displaying A Read-Only Page

Figure 2 shows a page from our sample website and Listing 1 shows the template for that page.

```
<html>
<head><title>Show Event</title></head>
<body>
<%<DATAOBJECT class="TEvent" name="Event" oid="${EventNum}"/>%>
<H3><%<TEXT>${Event.Event_Name}</TEXT>%></H3>
<%<TEXT>${Event.Event_Description}</TEXT>%>
<BR><BR>
<B>Date</B> <%<TEXT>${Event.Event_Date}</TEXT>%><BR>
<B>Time</B> <%<TEXT>${Event.Event_Time}</TEXT>%><BR>
<B>Tickets</B> <%<TEXT>$${Event.Ticket_Price}</TEXT>%>
</body>
</html>
```

➤ *Listing 1*


Show Event - Microsoft Internet Explorer
File Edit View Go Favorites Help
Address http://localhost/sample/cgi-bin/router.exe?PI:page=EventShow.sml&EventNum=8
Links
Local intranet zone

Women's High Dive Finals

World premiere event. All comps are suspended.

Date 6/19/96
Time 10:00 PM
Tickets $7.50

➤ *Figure 2*

This page shows a single entry from the Events table in the DBDEMOS database. The URL shown in the browser's address box illustrates how we call into this system to retrieve a page. We pass parameters into the CGI router which identify the page and content we want to see. The app server is set up to recognize certain parameters as 'processing instructions' which control the actions of the app server. All processing instructions are parameters with names beginning with PI:. The specific instruction to return a page is PI:PAGE. In Figure 2, we are making a request to see the page EventShow.sml, which is the name of the file that holds the template shown in Listing 1.

Many pages will need some additional information to identify the data we want to see on the page. In Figure 2 we are seeing a single row from the Events table, but we'll need some way of telling the app server which row we want to see. Since we're using the TEvent data object, we need to provide the object identifier (OID) for the particular event we are interested in. The OID for TEvent happens to be the EventNo field in the table. In this example, we pass along the event number as a URL parameter named EventNum. In the page template, the <DATAOBJECT> tag makes use of the EventNum parameter to set the tag attribute OID.

Notice the special syntax in the OID attribute. For the other <DATAOBJECT> attributes, Class and Name, the value for those attributes is taken as a literal string. For the OID attribute, we do not want the literal string EventNum. Rather, we want the contents of the variable EventNum. So the special syntax of ${<variable>} is taken to mean 'return the named value'. All URL parameters and page content variables, as well as data object properties, are available as variables.

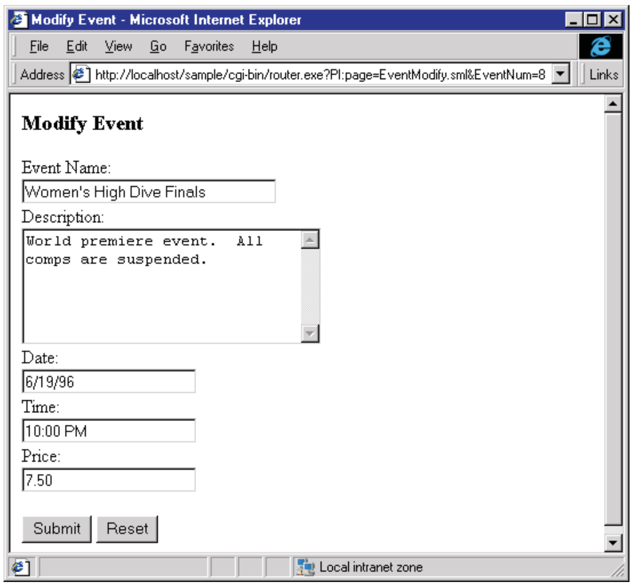
Also in Listing 1 we see a new SML tag called <TEXT>. This tag simply returns whatever literal value is between the <TEXT> and </TEXT> tags. On the surface this seems like a useless function, but the <TEXT> tag provides a point where we can use the ${} syntax to inject data values. We are using <TEXT> on this page to display several properties of the Event object. We use <TEXT> rather than <CONTROL> because we only want to display the values, not provide a data entry control to manipulate them.

Display A Data Entry Page

Figure 3 shows a web page that allows us to modify an entry in the Events table, and Listing 2 shows the template for that page. There is no difference in the URL call between a page that allows data entry and a page that does not, other than the page name itself obviously. There are some differences in how the template is

➤ *Listing 2*

```
<html>
<head>
  <title>Modify Event</title>
  <SMLScripts>
</head>
<body>
<FORM>
<%<DATAOBJECT class="TEvent" name="Event" oid="${EventNum}"/>%>
<SMLVars>
<H3>Modify Event</H3>
Event Name:<BR>
<%<CONTROL type="edit" name="edtName" property="Event.Event_Name"/>%><BR>
Description:<BR>
<%<CONTROL type="memo" name="edtDesc" property="Event.Event_Description"
cols="30" rows="6"/>%><BR>
Date:<BR>
<%<CONTROL type="edit" name="edtDate" property="Event.Event_Date"/>%><BR>
Time:<BR>
<%<CONTROL type="edit" name="edtTime" property="Event.Event_Time"/>%><BR>
Price:<BR>
<%<CONTROL type="edit" name="edtPrice" property="Event.Ticket_Price"/>%><BR>
<BR>
<%
  <BUTTON type="submit" caption="Submit">
    <ACTION type="update" value="Event"/>
    <ACTION type="page" value="EventMain.sml"/>
  </BUTTON>
  <BUTTON type="reset" caption="Reset"/>
%>
</FORM>
</body>
</html>
```



Modify Event - Microsoft Internet Explorer

Address http://localhost/sample/cgi-bin/router.exe?PI:page=EventModify.sml&EventNum=8

Modify Event

Event Name: Women's High Dive Finals

Description: World premiere event. All comps are suspended.

Date: 6/19/96

Time: 10:00 PM

Price: 7.50

➤ *Figure 3*

arranged, however. We still use the `<DATAOBJECT>` tag to instantiate the desired data object in order to prepopulate the data controls with current values. We still use the `<CONTROL>` tag to define data entry controls for each of the properties we want to modify. In Part 1 we saw how to process these tags to produce HTML controls. Now we need to focus on the additional infrastructure that is needed when the web page is submitted and the new data values in the controls need to be posted back to the database.

First things first. For a standard HTML page, values for all the data entry controls that appear within the `<FORM>` tag are sent back to the web server when the form is submitted. Normally, the `<FORM>` tag would include an `Action` attribute that identifies a CGI program to process the form variables. For an SML template we do not provide the `Action` attribute on the `<FORM>` tag; the submit action will be set programmatically via JavaScript attached to the `Submit` button.

Down at the bottom of Listing 2 we see the new tag `<BUTTON>` being used to create a submit button and a reset button. When we process the `<BUTTON>` tag for the submit button, the markup language engine will include a special JavaScript function on the page to set the form submit action programmatically. The actual submit action will be a call to our CGI router program with special URL parameters to indicate what kind of action to take. The `<ACTION>` subtags for the button tag tell us what special processing to do when this page is submitted. In this example, we want to update the contents of the `Event` object and then return to the EventMain.sml page. The HTML generated from the `<BUTTON>` tag for our submit button is shown in Listing 3. When the button is clicked, we execute the JavaScript function `DoSubmit`, which sets the form parameters and submits the form.

The `DoSubmit` function is added to the page automatically by the markup language engine. However, we need a way to tell the engine where to put it in the page. Inside the `<HEAD>` section of Listing 2 you'll see a tag called `<SMLScripts>`. This is an arbitrary placeholder tag that the markup language engine will replace with whatever JavaScript functions are required by the SML in that template.

Since the form submit action is set programmatically, we can have multiple submit buttons on a form doing different actions. For example, we could add a delete button on this form to delete the event. Clicking this button would call the CGI router with the parameter `PI:DELETE=Event`.

We have to keep in mind that the instance of a data object we used to generate a page is released once we send the generated page back to the browser. When a user submits a page to post changes, we have to reacquire and reinstantiate the same data object again and set its properties with new values from the form variables. When a form is submitted in this fashion, the only information we have at our disposal is the variables on the form and whatever URL parameters were passed in the call to the CGI program. How do we know which form variables map to which object properties? For that matter, how do we know what data object was used to fill those values in the first place? The original bindings were determined in the `<CONTROL>` tag in the page template, but all that information was lost when the SML tags were replaced with HTML. So we have to have some way of carrying the bindings with us in the page.

This binding information must be available when we submit the page, so it has to be contained within an HTML control on the form. We don't want it to be visible to the user, so we'll use a hidden control. A hidden control is like any other HTML control; it holds a string value, but it is not visible in the browser. In the template shown in Listing 2, under the `<DATAOBJECT>` tag is an additional tag called `<SMLVars>`. This is a placeholder that will be substituted with the HTML for this hidden control we will use to store data value/property bindings. Notice that the `<SMLVars>` tag must be within the `<FORM>` tag since we want this variable to be posted when the form is submitted.

The value/property bindings must be compiled when the page is generated and stored within the page itself. For each value in an HTML control, we need to know the class name of the data object that produced it, the OID of the

➤ *Listing 3*

Generated submit button:

```
<INPUT type=button value=Submit
onclick=DoSubmit("PI:update=Event&PI:page=EventShow.sml&EventNum=8")>
```

DoSubmit JavaScript function:

```
function DoSubmit(params) {
  document.forms[0].method='post'
  document.forms[0].action='../cgi-bin/router.exe?' + params
  document.forms[0].submit()
}
```

```
<html>
<head>
  <title>Modify Event</title>
  <SCRIPT>
    function DoSubmit(params) {
      document.forms[0].method='post'
      document.forms[0].action='../cgi-bin/router.exe?' +
        params
      document.forms[0].submit()
    }
  </SCRIPT>
</head>
<body>
<FORM>
<INPUT type=hidden name=SMLDataBindings value='
<databindings>
  <instance class="TEvent" oid="8" name="Event">
    <bindings>
      <binding>
        <control>edtName</control>
        <property>Event_Name</property>
      </binding>
      <binding>
        <control>edtDesc</control>
        <property>Event_Description</property>
      </binding>
      <binding>
        <control>edtDate</control>
        <property>Event_Date</property>
      </binding>
      <binding>
        <control>edtTime</control>
        <property>Event_Time</property>
      </binding>
      <binding>
        <control>edtPrice</control>
        <property>Ticket_Price</property>
      </binding>
    </bindings>
  </instance>
</databindings>
'>
<H3>Modify Event</H3>
Event Name:<BR>
<INPUT type="text" value="Women's High Dive Finals"
  name=edtName size=30 maxlength=30><BR>
Description:<BR>
<TEXTAREA name=edtDesc maxlength=100 cols=30 rows=6>
  World premiere event.  All comps aresuspended.
</TEXTAREA><BR>
Date:<BR>
<INPUT type="text" value="6/19/96" name=edtDate><BR>
Time:<BR>
<INPUT type="text" value="10:00 PM" name=edtTime><BR>
Price:<BR>
<INPUT type="text" value="7.50" name=edtPrice><BR>
<BR>
<INPUT type=button value=Submit
onclick=DoSubmit("PI:update=Event&PI:page=EventMain.sml")>
<INPUT type=reset value=Reset>
</FORM>
</body>
</html>
```

➤ *Listing 4*

specific instance of the data object, and the name of the object property itself. Since we have to store this information in a string, XML is a logical choice for representing the data. XML works well for representing complex data in a string and we already have XML parsing technology built into our system since that is what we are using to parse the SML tags.

Listing 4 shows the full HTML page generated from the template in Listing 2. Notice that the `<SMLVars>` tag has been replaced with a hidden control named `SMLDataBindings`. The value for this control is an XML-formatted string. Each `<INSTANCE>` tag describes the specific data object instance through the attributes `Class`, `OID` and `Name`. The `Name` attribute holds the name that was given to that data object instance when the page was generated. That's how we'll refer to it when we generate the instructions to modify it. Under the `<INSTANCE>` tag are a series of tag groups that define each HTML control holding a value from a property of that data object instance. In this example, there is an HTML control on the page that holds a value from the `Event_Name` property of a `TEvent` data object. Using this information, we can instantiate the data object and copy values from the control variables into the object's properties and save the data back in the database. This is simply the reverse process of what we did to pull the data out of the objects in the first place.

Making It Happen

Up to now, I've basically described how the system will work from an external view. Now let's start to look at the programming required to make this work. The core of the system is the application server, which we will implement as a simple COM object. In our COM object we define a single method called `GetContent` which accepts a single widestring parameter and returns a widestring. The idea behind `GetContent` is that we will pass all our parameters about the page request, including URL parameters and page content variables, as a single comma-separated string parameter to the function. `GetContent` will then return the entire HTML page that should be displayed in response to that request.

The Router

The CGI router program is very thin and simplistic. Its only job is to capture an HTTP request, translate the HTTP variables into a comma-separated widestring and call the `GetContent` method in our application server. Using the Web Broker components, all we really have to do is capture the information provided in the `TWebRequest` packet, and set the `Content` property of the `TWebResponse` packet to the value returned by the app server's `GetContent` method. I won't go into details here; there really isn't that much to it and the full source code for the CGI router is on the disk.

It is important to realize, though, that the router does not do anything but hand off the request to the app server. The app server is responsible for all logic and processing necessary to produce content. The router is simply a conduit between the web server and the app server through which the web request flows. Different web servers may support different means of capturing HTTP requests. All servers should handle CGI, but you may want to tune the performance by using a server-specific technology such as ISAPI or DSAPI. By keeping the router reduced to this simple communication role, we can easily substitute other mechanisms to hook into the web server without impacting the core page generation code in the application server.

The COM Object

As we said before, the `GetContent` method is the primary point of communication with our COM object. A single comma-separated string of URL and content variables is passed into this function.

GetContent scans this list of variables and determines whether the request is a page request (PI:PAGE) or a submit request (PI:UPDATE, PI:DELETE, etc). These are the two main branches of processing a request, so we break them out into two separate classes: TPageHandler and TSubmitHandler. TPageHandler does all the work of producing an HTML page from a template. TSubmitHandler does all the work of posting page values back to the original data objects.

Page Generation

TPageHandler is merely a container for the core tag substitution logic we built in October. It is responsible for loading the page template from an external file, parsing out the SML tags, and calling the tag resolver classes to interpret the tags. Delphi's TPageProducer component is a handy tool to start with for parsing SML tags from a template. TPageProducer is designed to read an HTML document and fire an event when certain types of tags are found. The tag can be replaced with dynamically generated text within the event handler. This is exactly what we want to do with SML tags, but there is a catch: SML tags are delimited with <% and %> markers. Unfortunately, TPageProducer does not respond to these delimiters.

It turns out all we really have to do is create a custom descendant of TPageProducer and tweak it to parse based on <% and %> delimiters. We'll call our new component TusPageProducer. We simply need to override the ContentFromStream method. This method already contains logic to find tags that are denoted by a < character followed by a # character. So we simply have to add parallel logic to look for tags denoted by a < character followed by a % character. When we find such a tag, we'll expose it to a new event handler called OnScriptingBlock which is defined as:

```
procedure (Sender: TObject;
  Body: string; var ReplaceText:
  string) of object;
```

TPageHandler simply loads the page template into a TusPageProducer component and every time a <% token is found, the OnScriptingBlock event is fired. Within this event, we have the SML tags available in the Body parameter and we pass this into the tag resolver code we built last time to return the generated HTML, which we pass back through the ReplaceText parameter. The TusPageProducer component replaces our SML tags with the generated HTML within the document. The source of this component is on the companion disk in the UHTTPApp.pas file.

Changes To TTagResolver

Obviously we'll need to expand the implementation of TTagResolver to accommodate the extra functionality we discussed earlier. We've added support for primitive expression evaluation with the ${<variable>} syntax. Coincidentally, that also implies that tag resolvers need access to the URL parameters of the page request. For example, in Figure 2 the URL parameter EventNum needs to be available to the <DATAOBJECT> tag resolver to load that specific event into the object. Finally, the tag resolvers could generate data that is not part of the HTML that replaces the tag. I'm talking about the possibility of including the DoSubmit JavaScript function and the data bindings that must be generated for modifiable data. This information is created by the tag resolver, but is not written to the page in place of the tag itself. The data must be accumulated internally until the whole page has been processed, and then written in specially designated spots on the page. Generated scripts are written where we've put the <SMLScripts> tag, and generated data bindings go where we've put the <SMLVars> tag.

We will create a class called TSMLContainer which will serve as the workspace for the TTagResolver classes. A TTagResolver object is created and destroyed for each SML tag encountered, so we need an external place to hold our working data. A single instance of the TSMLContainer class will be created for the page and all instances of TTagResolver will hold a pointer to this container. The TSMLContainer class will hold lists for generated scripts and data bindings. It will also hold the list of variables available to that page and a list of data objects that have been instantiated for that page.

Expression Handling

With the list of variables and the list of data objects, support for the expression syntax ${<variable>} is fairly simple. The <variable> may be either a named URL variable in our variable list, or an object property reference. We already know how to retrieve a value from an object property when we constructed the <CONTROL> tag in Part 1.

Listing 5 shows the EvaluateExpression function we added to TTagResolver. To support expressions in a tag attribute or body, we simply use this function to convert the expression. If the value passed in does not have the ${} syntax, then it is returned as a literal value. Otherwise the value of the corresponding page variable or object property is returned.

Handling Page Submits

We do not have to change much in the TDataObject base class to support read/write objects. For our purposes here, TDataObject is a

➤ *Listing 5*

```
function TTagResolver.EvaluateExpression(aExpression: string): string;
begin
  Result := Trim(aExpression);
  if Copy(Result, 1, 2) = '${' then begin
    aExpression := Copy(Result, 3, Length(Result) - 3);
    if FContainer.Variables.IndexOfName(aExpression) <> -1 then
      Result := FContainer.Variables.Values[aExpression]
    else
      if Pos('.', aExpression) <> 0 then
        Result := FContainer.ObjectCache.GetPropertyValue(aExpression)
      else
        raise Exception.CreateFmt('Unknown expression: %s', [aExpression]);
  end;
end;
```

simple wrapper around a TQuery. We access the query fields through the PropertyByName method. This method returns a TField so we can write new values to the TField as easily as we can read existing values from it. We only need to ensure that the SQL statement we've provided for the object is a 'live' query. In addition we've added a few more features to TDataObject to put the dataset into edit or insert mode, delete a row, and post changes. These are all simple wrappers around TQuery functionality, so you can look to the sample source code on the disk for the details.

The TSubmitHandler class takes care of requests to write changes back to data objects and is fairly straightforward. All the information we have to work with is given to us in the form of a string list of variable=value pairs. All the URL parameters and page content variables for the page we just submitted are in this list. One value of critical importance is the SMLDataBindings variable that was created during page generation (see top of Listing 4). This is the road map for processing the request. Since this is XML formatted data, we can use the TusXMLParser class we created last time to transform this raw data into a structured XML document.

We get into the TSubmitHandler object by having a PI:UPDATE parameter on the URL. The value of this parameter tells us which data object to update. So we have two main tasks: instantiate the named data object and set its properties according to the road map found in SMLDataBindings. Given the name of the data object, we loop through our XML document until we find an <INSTANCE> element whose Name attribute matches the one we are looking for. Once found, the Class attribute will tell us the class name of the data object. From that we can transform the class name into a class type using Delphi's GetClass function. Then we can instantiate the data object and load it with the specific OID given to us in the OID attribute. The code for TSubmitHandler can be found in the svrSubmitHandler.pas unit on the companion disk.

Once we have the data object, all we need to do is loop through the <BINDINGS> information and call the object's PropertyByName function to set the value of each property. PropertyByName returns a TField object, so we must use the TField.AsVariant property to set data, since our values all come to us in the form of strings. AsVariant will perform the necessary data conversions from string to numeric, date, time, etc. Since the data object is really just a wrapper around a TQuery, once we've set the properties, we just post the dataset to write the changes back to the database.

Obviously there is an issue of concurrency here. What if another user modified the database contents after we generated the page, but before the page was submitted? The most straightforward approach is to rely on optimistic locking. Each record has an additional field, like a timestamp. Whenever an update is made to the record, the timestamp field is updated. When reading data to generate a page, we store the timestamp of the data in the data binding info for that object. When we submit that page, if the current timestamp of the data no longer matches the timestamp we captured when we generated the page, somebody changed the data. At that point, we can return an error page that directs the user to try again with the current data.

Conclusion

What I've provided here is a very simplistic model for a web page system. There are certainly many more details to work out for this to be turned into a viable production system. My goal here was simply to focus on the mechanics of interpreting a custom tag language in order to move data in and out of simplistic data objects. The advantage of this approach over other technologies is a clear, clean distinction between the page authoring work and the data processing work.

Steve Troxell is a software engineer with Ultimate Software in the USA. He can be reached via email at Steve_Troxell@UltimateSoftware.com