Paper 60-28

Introduction to the SAS® Custom Tag Library

Robert Girardin, SAS Institute Inc., Cary, NC

ABSTRACT

The SAS Custom Tag Library, available in Version 3 of AppDev Studio™, is a collection of custom tags that provide a wide range of functionality, such as the ability to

- view and edit relational data via tables views or custom forms.
- explore multidimensional data stored in OLAP server cubes.
- perform common data management tasks such as sorting, filtering, adding computed items.
- attach JDBC™ data sources to standard HTML form controls via model/view communication.
- develop web applications for both web and wireless.

This paper will provide an introduction to building and deploying JavaServer Pages™ (JSP) with useful examples that illustrate the SAS custom tags mentioned above.

INTRODUCTION

JavaServer Pages technology offers a simple and fast way to build dynamically generated web applications that leverage the power of Java server-side processing. The JavaServer Pages specification defines standard tags for the use and modification of JavaBeans™ components. In addition to the standard tags, the JSP specification describes how you can create custom tag libraries, which are collections of reusable components that are to be used from within a JSP page. This paper focuses on the SAS Custom Tag Library, available in Version 3 of AppDev Studio.

One main advantage of using JSP custom tags is that non-Java web developers and programmers can use familiar tag-based libraries to easily create JSP pages. Another advantage is that tag libraries are portable, and can be used within a JSP page in any servlet container. Grouping of similar, reusable components for development are yet another benefit which allows faster and cleaner implementation.

The SAS Custom Tag Library is a collection of JSP custom tags that encapsulate the functionality of our specialized Java TransformationBean™ components, which reside in the `com.sas.servlet.tbeans` packages. The tag library allows you to perform a wide variety of tasks through the adding of tags to your JSP page. The SAS tags are prebuilt and bundled in a jar file, which can be dropped into any web application to aid in speeding up development. Each custom tag has a corresponding tag handler class, which can be found in the `com.sas.taglib` package. Note that the older `com.sas.servlet.beans` and `com.sas.taglib.servlet.beans` packages have been deprecated in Version 3.0 of AppDev Studio. Validation classes are also included, which insures that the SAS Custom Tag Library is used properly. Using the webAF Java development environment within AppDev Studio, you can drag and drop many of the tags into a page of your web application and have the tag code automatically generated for you.

The library is made available to the JSP page through a `taglib` directive, which maps a tag prefix to a unique URI identifier and tag library descriptor file. A tag is added to the page by specifying the defined prefix followed by the tag's name in an XML syntax. Attributes and body tags can be added as well, such as in the following code example:

```
<%@ taglib uri="http://www.sas.com/taglib/sas"
prefix="sas" %>

<sas:TableView id="myTable" cellPadding="2">
   <sas:Columns>
        <sas:Column modelIndex="2"/>
   </sas:Columns>
   <sas:Rows>
        <sas:Row/>
   </sas:Rows>
</sas:TableView>
```

Below are listed some of the key tags and features of the SAS Custom Tag Library:

- TableView Tags – view and edit relational table data
- OLAPTableView Tags – explore data within OLAP server cubes
- Selector Tags – perform data management tasks such as sorting, filtering, and selecting items
- Form Tags – create customized HTML forms for gathering and validating user input or connect to JDBC data sources
- Graphics Tags – add graphic elements such as bar charts, pie charts and more
- MenuBar Tags – design menu and submenu structures
- TreeView Tags – display hierarchical lists of items
- IPage Tags – develop applications for wireless devices, such as mobile phones or PDA's

JAVA SERVLET AND JSP TECHNOLOGIES

JavaServer Pages are an extension to the Java Servlet technology. Java servlets provide a component-based, platform-independent method for building Web applications. Servlets are Java classes that run on the server side within a servlet container, and can take advantage of all of Java's benefits, such as using any of the Java APIs, portability across platforms and reusability of previously developed components. They also combine built-in support with solid performance for HTTP-specific calls. Additionally, unlike CGI, servlets use a single process for all requests against the server. Therefore, servlets reduce the risk of overloading the server.

JavaServer Pages technology is an extension of servlet technology that is designed to support HTML and XML page development. It makes it easier to combine static template data with dynamic content. With the aide of scriptlet and other JSP elements, developers can create pages which combine HTML and the power of running server-side Java.

The JavaServer Pages specification also defines standard tags for the use and modification of JavaBeans components. The JSP specification describes how you can create custom tag libraries, which are collections of reusable tag components that can be used within a JSP. Custom tags are a great way to encapsulate common functionality, expand the JSP technology and speed up web application development time. They are also very portable since they can be embedded in a jar file to be used in multiple web applications.

For a JSP to use a custom tag library, it must include a taglib directive at the top of the page, such as:

```
<%@ taglib uri="http://www.sas.com/taglib/sas"
prefix="sas" %>
```

The `taglib` directive contains two attributes: a `uri`, whose value is a Uniform Resource Identifier which eventually maps to the Tag Library Descriptor, and a `prefix` for the tags using this library within the page. The Tag Library Descriptor (TLD) is a file that contains information about the tag library and is used by the servlet container running the web app. The TLD contains information such as tag and attribute names, required attributes, validation information and information about JSP scripting variables. Here is a simple example of a SAS CheckBox custom tag that will write out the HTML code to the requesting client:

```
<sas:CheckBox id=”myCheckBox” />
```

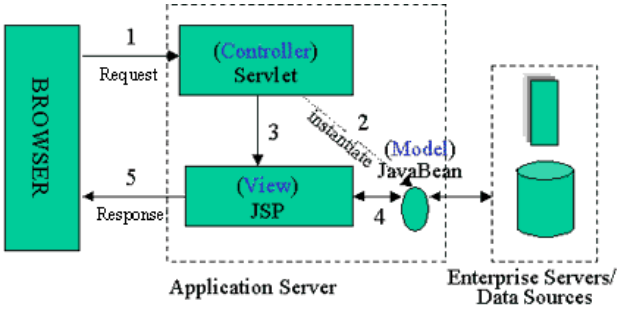
When a JSP page containing custom tags is first accessed by a client, it is parsed, validated, translated into a servlet, and finally executed. During the parse phase, the TLD file is used to make sure the tag library has proper JSP syntax. If present, validation classes are then used to verify that a particular tag library is being used as intended, such as the correct nesting of tags and the proper use of tag attributes. Validation messages will be sent back to the browser if one occurs. Translation involves compiling the page into a Java servlet, which includes translating the tags to Java code and setting up scripting variables. Requests made to the JSP page are sent back to the client in the execution step via a response.

Another important aspect of custom tags is the use of scripting variables. Classes that extend the TagExtraInfo class and are added to the TLD file can be used to add scripting variables for use within a JSP. Scripting variables, whether defined in a servlet, scriptlet code, or another custom tag, allow attributes which reside on a particular JSP scope (page, request, session or application) to be accessible within a tag.

MODEL VIEW CONTROLLER ARCHITECTURE

JavaServer Pages technology can complement servlets within Web application development. JSP technology enables the user interface view to be separated from the business logic of the application. This is known as Model-View-Controller or MVC Type II, which relies on the ability of a servlet to delegate, or "dispatch," a request to another resource (Figure 1). A front-end controller, which can be a servlet (Example 1), handles request dispatches, initialization and cleanup. The requests are forwarded to a JSP page which acts as the view component of the architecture.

Figure 1: MVC Architecture



A request is made from the client that goes through the controller servlet (1). The controller servlet performs any initialization and creates the model to be used by the view (2). The servlet then forwards the request (3) to the JSP. Below is a sample of code controller servlet for use with the SAS custom tags.

Example 1: Controller Servlet

```
public class ControllerServlet extends
HttpServlet
{
/*
 * doPost()
 * Respond to the Post message.
 */
public void doPost(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{

HttpSession session = request.getSession();
HttpActionProvider ap =
(HttpActionProvider)session.getAttribute("ap1");

if (ap == null){
   ap = new HttpActionProvider();
   ap.setName("actionprovider");
   ap.setControllerURL("Controller”);
   session.setAttribute("ap1", ap);
}
else{
   ap.executeCommand(request, response,
   response.getWriter());
}

RequestDispatcher rd =
getServletContext().getRequestDispatcher(“view.j
sp”);
rd.forward(request, response);

}

public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException
{
        doPost(request, response);
}
}
```

Before the request is forwarded to a JSP, the servlet controller initializes the SAS ActionProvider, which is required for many of the SAS TransformationBean components. The ActionProvider is comprised of a set of SAS classes that work with Swing and JSP component viewers to provide the user actions available for each renderable portion of the view. The ActionProvider allows for customization of the user actions by overriding default actions (like drilldown, etc) or adding new actions to the viewers (e.g. link to external web site). If no ActionProvider is found on the session, then one is created and set on the session. The `setControllerURL` call is necessary since we want the ActionProvider to go back through the controller servlet instead of the current JSP page. If an ActionProvider is found on subsequent calls, then it is executed. For more information on the ActionProvider framework, you can refer to the `com.sas.actionprovider.HttpActionProvider` API.

The `RequestDispatcher` code segment is used to forward the request to a JSP, in this case `view.jsp`.

Page or user interface designers, that may have little or no Java experience, can focus on writing the HTML and JSP that controls the view of the page. The JSP may use JavaBeans (4) to further customize the view. The last step in the Model View Controller architecture involves sending the response back to the requesting client (5).

Version 3 of AppDev Studio allows developers to easily create Web Application Projects. Web applications using servlets and JSP in a Model View Controller can be easily developed within the webAF Java development environment via the application project wizard.

DATA VIEWER TAGS

The SAS Custom Tag library included in Version 3 of AppDev Studio introduces many new custom tags and subtags, including those for viewing and editing relational data and for exploring multidimensional data stored in OLAP server cubes.

The first thing that must be done is to create the model as described earlier in the Model View Controller Architecture. This would be setup in the controller servlet. The SAS TableView custom tag will display a `javax.swing.table.TableModel` model, such as the `javax.swing.table.DefaultTableModel`. There are a variety of additional types of models that can be used, since there are now many adapters for transforming a model to the correct `TableModel` Java interface. Two of these adapters are the `JDBCToTableModelAdapter` and the `BusinessQueryToTableModelAdapter`. Here is an additional part of the controller servlet that creates a DefaultTableModel needed by the TableView Tag in the JSP page:

```
DefaultTableModel dtm = new DefaultTableModel();
String columnNames[] = {"NAME","AGE","SEX"};
String row0[] = {"Bob","10","M"};
String row1[] = {"Sue","11","F"};
String row2[] = {"Joey","10","M"};
String row3[] = {"Albert","9","M"};
String row4[] = {"Carol","11","F"};
String row5[] = {"Joanne","10","F"};
String row6[] = {"Mike","11","M"};

Object data[][] = new Object[6][3];
data[0] = row0;
data[1] = row1;
data[2] = row2;
data[3] = row3;
data[4] = row4;
data[5] = row5;

// add additional rows and data

dtm.setDataVector(data,columnNames);
session.setAttribute("m1", dtm);
```

Example 2 shows an example JSP page that writes out a TableView. The controller servlet would forward the request to this page after the model was initialized and put on the session via the `setAttribute` method.

Example 2: TableView

```
<%@ taglib uri="http://www.sas.com/taglib/sas"
prefix="sas" %>
<html>
<head><link href="styles/sasComponents.css"
rel="STYLESHEET" type="text/css"> </head>
<body>

<sas:TableViewComposite id="t1" model="m1"
actionProvider="ap1" scope="session">
   <sas:TableView/>
</sas:TableViewComposite>

</body>
</html>
```

The sasComponents.css file is the default style sheet for all of the SAS Java components and should be included on each JSP page. Because the tag's scope is session, the `actionProvider` and `model` attributes will be searched for on the session and should be found since they were setup in the controller servlet and stored on the session. Figure 2 shows the resulting table shown in the browser.

Figure 2: TableView

	NAME	AGE	SEX
1	Bob	10	M
2	Sue	11	F
3	Joey	10	M
4	Albert	9	M
5	Carol	11	F
6	Joanne	10	F

This shows the default styles applied. The default action of moving columns is available to the DefaultTableModel by clicking on a column heading (Figure 3 and 4).

Figure 3: TableView (column actions)

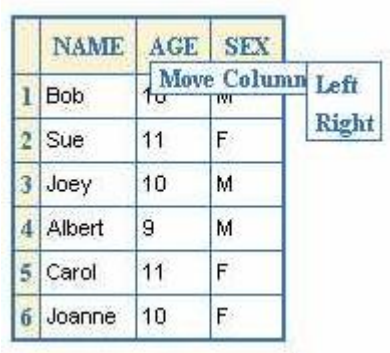


Figure 4: TableView (after column move)

	NAME	SEX	AGE
1	Bob	M	10
2	Sue	F	11
3	Joey	M	10
4	Albert	M	9
5	Carol	F	11
6	Joanne	F	10

Adding a CellRenderer to the TableView can further customize the look of the table (Example 3). The resulting table has an alternate color every other row (Figure 5).

Example 3: TableView with CellRenderer

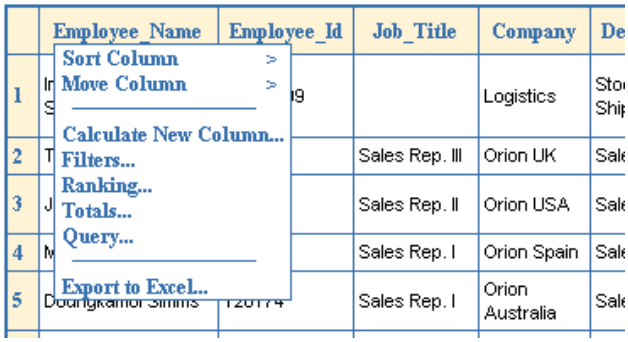
```
<sas:TableViewComposite id="t1" model="m1"
actionProvider="ap1" scope="session">
   <sas:TableView>
        <sas:CellRenderer bgColor="dcdcdc"
        startRow="1" endRow="-1"
        repeatRowFactor="2" startColumn="1"
        endColumn="-1" repeatColumnFactor="1"/>
   </sas:TableView>
</sas:TableViewComposite>
```

Figure 5: TableView with CellRenderer

	NAME	AGE	SEX
1	Bob	10	M
2	Sue	11	F
3	Joey	10	M
4	Albert	9	M
5	Carol	11	F
6	Joanne	10	F

Other types of default actions are available depending on the model. Figure 6 shows the default actions available when connecting to the BusinessQueryToTableModelAdapter.

Figure 6: TableView Actions in BusinessQuery Model



Editing a JDBC data source can be used within the TableView custom tag (Example 4 and Figure 7):

Example 4: TableView (JDBC Editing)

```
<sas:TableViewComposite id="t1" model=" m1"
actionProvider="ap1" scope="session">
   <sas:TableView>
        <sas:Edit enabled=”true” />
   </sas:TableView>
</sas:TableViewComposite>
```

Figure 7: TableView (JDBC Editing)

		NAME	SEX	AGE
1		.	.	23
2		dave	F	33333333333333
3		Barbara	M	13
4		Carol	F	10
5		Henry	M	12
6		James	M	15
7		Jane	F	14
	+			

Rows 1 - 7 of 32

Viewing OLAP cubes is performed in a similar way by using the OLAPTableView tags (Example 5). A model must be setup of the type `com.sas.storage.olap.OLAPDataSetInterface` and set on the session to be used by the tag. The resulting OLAPTableView using a BusinessQuery model is shown in Figure 8. The default actions are shown in Figure 9.

Example 5: OLAPTableView

```
<sas:OLAPTableViewComposite id="t2" model="oM1"
actionProvider="ap1" scope="session">
   <sas:OLAPTableView/>
</sas:OLAPTableViewComposite>
```

Figure 8: OLAPTableView

All TIME > Year
All PRODUCTS > Product Line
All GEOGRAPHY > State

			AMOUNT	GROSS_P
State	Product Line	Year		
Idaho	Clothes & Shoes	1998	10549	449841.89
		1999	10567	446898.46
		2000	10700	455053.39
		2001	10718	452192.22
		2002	10754	462600.59
Montana	Clothes & Shoes	1998	6592	279798.47
		1999	6716	284028.76
		2000	6633	279612.70

Figure 9: OLAPTableView Actions

There is a lot of functionality built into the default MenuBar within the TableView, such the ability to calculate new measures, apply advanced filtering to data, sort data based on a particular column plus many others.

SELECTOR TAGS

The selector tags involve components that allow data selection in a variety of views. The TreeView custom tag allows a model to be passed into the tag or for TreeNode subtags to be used to define the tree model. Example 6 lists a simple example of the TreeView tag with most attributes left as default. The resulting TreeView is shown in Figure 10.

Example 6: TreeView

```
<sas:TreeView id="tv1" nodeLoading="PROGRESSIVE"
rootNodeVisible="true" expansionLevel="1"
scope="session">

   <sas:TreeNode name="root" text="App Data">
        <sas:TreeNode name="GEO" text="Geography"
        description="Country, Region, Country,
        State & City">
<sas:TreeNode name="COUNTRY" text="Country">
   <sas:TreeNode name="USA" text="U.S.A.">
        <sas:TreeNode name="NC" text="North
        Carolina">
               <sas:TreeNode name="CARY"
               text="Cary"/>
        </sas:TreeNode>
        <sas:TreeNode name="CA"
        text="California"/>
...<!—Rest of TreeView -->...
```

Figure 10: TreeView

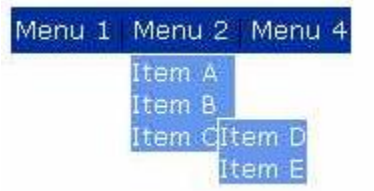
Customization of the TreeView can be used to include hyperlinks, control images, and set loading style from fully expanded (al nodes loaded) to progressive loading which loads TreeNodes as they are expanded.

As mentioned earlier, a version of the MenuBar is embedded within certain types of TableViews. However, developers can use the MenuBar tag itself (Example 7 and Figure 11).

Example 7: MenuBar

```
<sas:MenuBar id="mb1" separator="|"
menuType="DROP_CLICK">
   <sas:Menu label="Menu 1" URL="x.jsp "/>
   <sas:Menu label="Menu 2">
        <sas:MenuItem label="Item A"
        URL="A.jsp"/>
        <sas:MenuItem label="Item B"
        URL="B.jsp"/>
        <sas:Menu label="Item C">
             <sas:MenuItem label="Item D"
             URL="D.jsp"/>
             <sas:MenuItem label="Item E"
             URL="E.jsp"/>
        </sas:Menu>
   </sas:Menu>
   <sas:Menu label="Menu 4" URL="x.jsp"/>
</sas:MenuBar>
```

Figure 11: MenuBar



One last TransformationBean selector that has an associated SAS Custom Tag is the DualListSelector. It uses a javax.swing.ListModel interface as its model. As in most of the other selector tags, developers can setup the model in a controller servlet and pass the model to the tag via the model attribute. The following example, however, uses the DualListSelector tag and

the Item tag, which is used to define List model elements (Example 8 and Figure 12).

Example 8: DualListSelector

```
<sas:DualListSelector>
   <sas:Item id="a" value="item1"
   description="Item 1" selected="false" />
   <sas:Item id="b" value="item2"
   description="Item 2" selected="false" />
   <sas:Item id="c" value="item3"
   description="Item 3" selected="false" />
</sas:DualListSelector>
```

Figure 12: DualListSelector

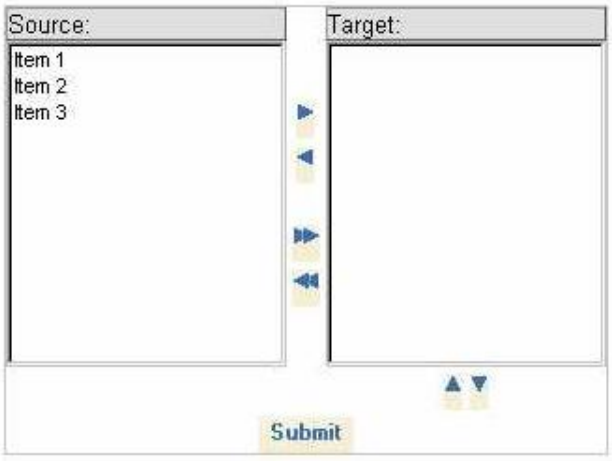
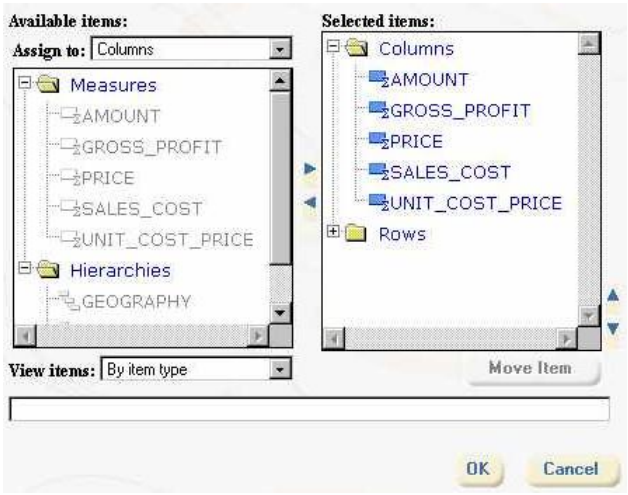


Figure 13 shows the DualListSelector used within OLAPTableView's Calculate New Measure action menu item.

Figure 13: DualListSelector in OLAPTableView



FORM ELEMENT TAGS

The Form Element components help in creating customized HTML forms for gathering and validating user input. The Form tags include the CheckBox, ChoiceBox, ComboBoxView, Form, Hidden, Image, LIstBox, ListBoxView, Password, PushButton, Radio, TextEntry and TextArea tags.

Example 9 is a simple example that reads the ChoiceBox model for available states from a previously defined model on the session.

Example 9: Form Elements

```
<sas:Form id="form" method="get"
action="submit.jsp" >
   <table border="0">
   <tr>
        <td>Name</td>
        <td>
        <sas:TextEntry id="name" />
        </td>
   </tr>
   <tr>
        <td>Address</td>
        <td>
        <sas:TextEntry id="address" />
        </td>
   </tr>
   <tr>
        <td>State</td>
        <td><sas:ChoiceBox id="state"
        model="stateModel" scope="session"/>
        </td>
   </tr>
   </table>

   <sas:PushButton id="submitButton"
   name="submit" text="Submit Form"
   type="submit" />
</sas:Form>
```

In addition to regular HTML Forms, the DataBean and DataBeanForm custom tags can be used to connect to a databean. The WebAF DataBean Wizard allows you to connect to a JDBC data source and create a databean that contains a property for specified columns in the data source.

Contained within the DataBeanForm tag are form elements whose names correspond to a property of the databean. This is how the DataBeanForm determines which databean property to access and set on the form element. The model attribute for the DataBeanForm is the databean. In addition to the model, an ActionProvider must be set on the DataBeanForm. Editing capability is also available. For more information on databeans and data driven applications in general, refer to paper 49-28, *Developing Data-Driven Applications Using JDBC and Java Servlet/JSP Technologies.*

IPAGE TAGS

The IPage TransformationBeans are a set of components that allow developers the ability to create wireless web-based applications with speed and ease. The IPage beans get a request from a client device, and will return back the appropriate markup language for that device, whether it is the Wireless Markup Language (WML), the older Handheld Device Markup Language (HDML) or HTML. Currently there are three IPage components: IText, IForm and IMenu.

The IText component is used to write out text to the client. Example 10 shows the JSP code will generate an entire page of text for the requesting client device.

Example 10: IText

```
<%@taglib uri="http://www.sas.com/taglib/sas"
prefix="sas"%>
<sas:IText>
Hello World! Here is a SAS IText Transformation
Bean.
```

```
</sas:IText>
```

If a HTML browser requests the page containing the above tag, HTML will be written to the browser as seen below (Figure 14). The same JSP accessed by phone would return back WML as the response to the client device (Figure 15).

Figure 14: IText (HTML Output)

Figure 15: IText (WML Output)

The IMenu writes out a menu which links to other menus or external links. A sample of the IMenu is shown in Example 11.

Example 11: IMenu

```
<%@taglib uri="http://www.sas.com/taglib/sas"
prefix="sas"%>
<sas:IMenu>
   <sas:IMenuItem URL="IText1.jsp"
   description="Menu Item 1"/>
   <sas:IMenuItem URL="IText1.jsp"
   description="Menu Item 2"/>
</sas:IMenu>
```

The corresponding WML for the IMenu example (Example 11) which is automatically generated by the component is listed below:

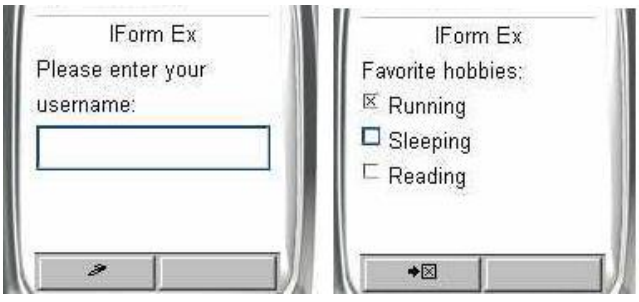
```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML
1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
   <template>
        <do type="prev">
              <prev/>
        </do>
   </template>

   <card id="IMenu">
        <p>
        <select>
             <option onpick="IText1.jsp">Menu
             Item 1</option>
             <option onpick="IText1.jsp">Menu
             Item 2</option>
        </select>
        </p>
   </card>
</wml>
```

The final IPage component is the IForm, which allows user information to be obtained by the JSP developers via form submission. Example 12 shows the IForm and some of the subtags that can be used to obtain information from the user. Figure 16 shows two of the WML cards generated by the IForm on a phone emulator.

Example 12: IForm

```
<%@taglib uri="http://www.sas.com/taglib/sas"
prefix="sas"%>
<sas:IForm name="form" action="IFormResults.jsp"
title="IForm Ex">
   <sas:IFormTextEntry name="name"
   prompt="Please enter your username: "/>
   <sas:IFormPassword name="password"
   prompt="Enter your password: "/>
   <sas:IFormListBox name="hobbies"
   prompt="Favorite hobbies: ">
        Running
        Sleeping
        Reading
   </sas:IFormListBox>
   <sas:IFormRadio name="colors"
   prompt="Favorite color: ">
        Red
        Green
        Blue
        Yellow
   </sas:IFormRadio>
   <sas:IFormHidden name="foo" value="bar" />
</sas:IForm>
```

Figure 16: IForm (IFormTextEntry and IFormListBox)



Note that the IPage custom tags should not be mixed with other HTML generating tags. The IPage components write out an entire page of well-formed markup and adding other HTML components can create malformed markup, which may not display at all on

devices such as phones.

GRAPH TAGS

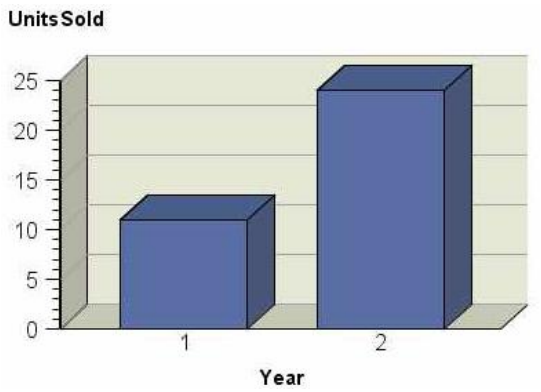
The SAS Custom Tags also provide many customizable graph tags, such as the BarChart, LineChart, PieChart, RadarChart and ScatterPlot.

The graphics and chart tags usually set up the appropriate type of graph model and pass it in via the model attribute (Example 13). A sample BarChart may look like Figure 17.

Example 13: BarChart Graph

```
<sas:BarChart id="barChart1" name="Chart1"
model="barChartModel" scope="session">
   <sas:BarChartModel dimension="3D"/>
</sas:BarChart>
```

Figure 17: BarChart Graph



DEPLOYMENT AND CUSTOM TAGS

Deployment of web applications, JSP's, servlets, and custom tags can sometimes be one of the more difficult tasks of the development process. The list below contains some tips to make sure the deployment of your JSP's which use the SAS Custom Tag Library goes smoothly.

- Check that the web application contains all the correct subdirectories. The default webAF structure contains an images, scripts, styles and templates directories. Care must be taken when modifying this area since many of the files contained there are required for the components to function.
- The SAS Custom Tag Library and its associated TLD file are bundled in the sas.servlet.jar file. Verify that the jar file is within the web application's WEB-INF/lib directory, which is where the servlet container will look for it.
- Check the application's jar files and their locations to avoid classpath issues.
- Make sure that the taglib directive listed on the JSP contains the proper uri attributes which map to a TLD file.
- Cleanup or remove any information that may have been placed on the session or application scopes at the appropriate time. For example, if a SAS Custom Tag uses `scope="session"` and contains an id attribute, a scripting variable will be placed on the session which should be cleaned up when it is no longer needed.
- Review the SAS Custom Tag Library Reference for more complete tag syntax, examples and overviews.

CONCLUSION

The SAS Custom Tag Library gives you the ability to easily add SAS component functionality to your web applications. Using these tags will decrease the amount of Java code written in your JSP pages, expand the flexibility of your application, and speed up the overall web development process.

More information about the SAS Custom Tags can be found on the AppDev Studio site at http://support.sas.com/rnd/appdev/index.htm and at the SAS Custom Tag Library Reference, http://support.sas.com/rnd/appdev/webAF/taglib.htm.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Robert Girardin
SAS Institute, Inc.
Work Phone: (919) 531-2189
Email: Robert.Girardin@sas.com