

Advanced Techniques in SAP® Html Business (HTMLB): For Java Enterprise Portals EP6

By Bill Guderian, BK Systems, Inc.

Editor's Note: In this article for Portals and Java developers, Portals Editor Bill Guderian presents the facts and dispels the myths of using SAP's HTML for Business (HTMLB) code for generating Java Server Pages (JSP) in an EP6 environment. Bill begins with a candid discussion of the myriad flaws associated with JSP and suggests a do-able solution. His approach employs the well-worn KISS (Keep It Simple, Stupid) methodology that we all love, beginning with his simple but sound workaround: moving the bulk of the code to a single Java bean per page. Bill uses the Page Context object, the Form object, embedded objects in the TableView Control, and the Scroll Container (to embed a page within a page) to illustrate the power of HTMLB. Using practical code examples and project-based pointers, Bill shows that it really is possible to generate Java Server Pages that deliver results to your user community without compromising performance or security.

Introduction

In my last article, I discussed the details of how to get the Java Connectivity Architecture (JCA) up and running on EP6. Hopefully, the readers of that article gleaned enough information from it to allow them to get JCA up and running in little time. Quite a few readers contacted me concerning the article and told me that they found it helpful to them. In some cases, people stated that they had to live through many of the same pains that I had had in getting the architecture to work. They were thankful that someone had documented these matters, so that we don't all have to re-invent the wheel.

When I first started developing Web applications, I liked JSP and the idea of a three-tiered model-view-controller design. Now I've begun to develop a loathing for JSP.

In this article, I am going to cover a collection of techniques that I have used to improve the preparation of JSP pages that use HTMLB. For those readers who are not familiar with HTMLB, it is a collection of JSP custom tags produced by SAP, referred to mainly as a custom tag library. When the programmer develops pages using these tags, the pages embrace a consistent look and feel, and development of code is expedited. When HTMLB is used, it is generally not necessary to use HTML directly in the page, except under certain cases like adding a line break `
`, or something equally benign.

Before I get into the specific techniques, allow me to critique somewhat the use of JSP pages and SAP's

custom tag library. When I first started developing Web applications, I liked JSP and the idea of a three-tiered model-view-controller (MVC, Model 2 architecture) design, where the JSP page occupies the view layer, the layer closest to the user. Although I feel that a model-view-controller design is essential for any serious development, I've begun to develop a loathing for JSP. As an aside, JSP has really nothing to do with the concept of a 3-tiered architecture. It merely occupies the position of the view tier.

JSP suffers from the following idiosyncrasies:

- It is a script-like code.
- It is difficult to debug.
- It cannot be easily syntax checked.
- The pages cannot be nicely formatted.
- The pages can be difficult to organize and understand, particularly when you have nested tags.
- The lure of using ugly scriptlet code in the page is just too overwhelming.
- The development of your own custom tag library is not that straightforward, particularly in a portals environment.
- It can be difficult to "fine-tune" the format of the page on the page itself without the use of a long sequence of scriptlet code, which I try to avoid.

The bottom-line of this list of issues is that it can take a long time to properly develop and nicely format a page.

If this is not bad enough, there is a bug or “feature” somewhere in either Eclipse or SAP’s plug-in that causes Eclipse to jam in some cases, when a JSP page is edited, saved, compiled, and then exported. I was praying that someone would fix this in EP6, but to no avail. Sometimes, we can avoid this bug by closing all the pages before compiling, and exporting the application. However, this approach is not always effective, nor is it practical. When the bug occurs, it is necessary for the programmer to exit Eclipse and then restart it. It can take quite a while to restart Eclipse. After you have done this more than 5,000 times as I have, you will have lost some of your love for JSP. With this in mind, you will see my JSP pages are as simple as possible, and I avoid changing them frequently.

For these reasons, many of the techniques that I cover in this article attempt to improve this situation by allowing the developer to come off the page, and place his or her complex page layout in a bean, where it

can be more easily developed. Some developers would argue that we are violating the 3-tiered concept. I would counter this by saying that the JSP page and associated bean constitute the view layer, and let’s not worry about it.

Using the Page Context

As I mentioned in one of my past articles, it is a good idea for each JSP page to have one and only one page bean, for simplicity sake. The bean should have access to the `IPortalComponentRequest` object, so that it can gain access to information stored in the request, session, or context objects. The code at the top of the JSP page should look something like Figure 1.

The page should also contain HTMLB content, page, and form tags. These tags are shown in Figure 2.

Once we have established this, we can begin to use the bean to remove tags and long-winded scriptlet code from the pages. I will illustrate this with an example. Say we want to put a heading on a page. We can use a single line scriptlet as follows:

```
<%pageBean.getHeading(theContext);%>
```

```
<jsp:useBean id="pageBean" scope="page" class="beans.PageBean" />
<jsp:setProperty name="pageBean" property="request" value="<%=componentRequest%>" />
```

Figure 1: Use of the Page Bean in a JSP Page

```
<hbj:content id="theContext" >
  <hbj:page title="My Page">
    <hbj:form id="myFormID">
      </hbj:form>
    </hbj:page>
  </hbj:content>
```

Figure 2: Context, Page and Form Tags on JSP Page

The bottom line of this list of issues is that it can take a long time to properly develop and nicely format a page.

In this case, we are passing the variable “theContext” as a parameter to the `getHeading` method of the bean. The context variable is an instance of `IpageContext`. In our example here, the `getHeading` method is a relatively complex sequence of statements that we would not want to put in a scriptlet on a page. Here is the `getHeading` method in Figure 3.

In this case, we are wrapping a `TextView` and a `Link` inside a `GridLayout`. The `GridLayout` is further wrapped in a `Document`. The `Document` is then rendered to the context. Note that in this case, the `ResourceBundle` is used to get the text elements. We do this to internationalize the page.

Using the Form Object

The IPageContext technique is especially useful when we want to format a page that contains a large number of input fields. In this case, however, we need to modify the technique slightly to ensure that the input fields are wrapped in a Form. We do this to ensure that the input of the fields can be extracted from the request sent to the server. If the fields are not in a Form, they will not be available in the request, and we will search in vain in an attempt to read the values from the screen.

Therefore, the scriptlet for this type of construct is as follows:

```
<%pageBean.getFieldGrid(myFormID);%>
```

We defined the myFormID variable when we used the HTMLB form tag discussed previously. The method call getFieldGrid looks something like this:

```
public void getFieldGrid(Form form) {
```

The code shown in Figure 4 contains a number of nested layers. The Group contains a layout. The layout contains both input fields and field labels. All the fields are of the same type and use the same labeling technique.

When we retrieve the value of the fields after the form is submitted, we would use code that looks something like this:

```
String fieldValue = getPageContext().
getDataForComponentId("field_name_1").
getValueAsString();
```

At this point, the code is free to work with the value that was entered in the field. Notice that the method getPageContext() is only available in a class that extends JSPDynPage. This would be your controller class.

```
public void getHeading(IPageContext context) {

    Document document = new Document("heading");
    GridLayout layout = new GridLayout(1, 2);
    layout.setWidth("90%");
    TextView tv = new TextView("text_title");
    tv.setText(this.getTitle());
    tv.setDesign(TextViewDesign.HEADER1);
    layout.addCell(1, 1, new GridLayoutCell(tv));
    Link link =
        new Link("home",
            request.getResourceBundle().getString("home"));
    link.setOnClick("onEvent");
    GridLayoutCell cell = new GridLayoutCell(link);
    cell.setHAlignment(CellHAlign.RIGHT);
    layout.addCell(1, 2, cell);
    document.addComponent(layout);
    document.addRawText("<br>");
    document.render(context);
}
```

Figure 3: Use of Page Context ("Get Heading" Method)

```
Group group = new Group();
group.setDesign(GroupDesign.SAPCOLOR);
group.setTitle(this.getBoxTitle());
group.setWidth(width);
GridLayout layout = new GridLayout();
layout.setCellPadding(cell_padding);
this.addFieldToLayout(1, 1, "field_name_1", layout);
this.addFieldToLayout(1, 3, "field_name_2", layout);
group.addComponent(layout);
form.addComponent(group);
}

private void addFieldToLayout(
    int row,
    int column,
    String fieldName,
    GridLayout layout) {
    InputField field = new InputField(fieldName);
    field.setType(DataType.STRING);
    String labelText =
        request.getResourceBundle().getString("label_" + fieldName);
    Label label = new Label(labelText);
    label.setLabelFor(field);
    layout.addCell(row, column, new GridLayoutCell(label));
    layout.addCell(row, column + 1, new GridLayoutCell(field));
}
```

Figure 4: Using the Form Object to Construct a Grouping of Fields

In all my code, I store this value in a request helper, and then pass the request helper to a command that is called by the controller. This prevents my controller from growing out of control. The discussion of design patterns and how to implement a Model 2 design is beyond the scope of this article.

Embedding Objects in a Table Control: Links

It is desirable to embed HTMLB controls in other controls, and this is

possible, provided that the HTMLB control is a subclass of `com.sapportals.htmlb.Component`, which most HTMLB controls are. We can embed all types of components in other components. I already did this numerous times in the cases shown above.

There are times, however, when doing this can be a bit tricky. Such is the case when embedding components in the Table View control. One common case of this is when we want to put a link in the Table View con-

trol. This comes up quite frequently. Another case is when we want to put an input field into a Table View control. I will cover that a little later, as it is a bit more complex. The SAP documentation covers this, but I am going to put a little twist on it, so as to make it a bit easier to reuse.

We can embed all types of components in other components. There are times, however, when doing this can be a bit tricky.

```
<hbj:tableView
  id="myTableView"
  model="pageBean.model"
  design="STANDARD"
  headerVisible="false"
  footerVisible="<%=pageBean.isFooterVisible() %>"
  fillUpEmptyRows="false"
  navigationMode="BYPAGE"
  selectionMode="None"
  headerText="<%=pageBean.getHeaderText()%>"
  visibleFirstRow="<%=pageBean.getVisFirstRow()%>"
  visibleRowCount="<%=pageBean.getVisibleRowCount() %>"
  width="<%=warrantyPageBean.getTableWidth() %>"
  onNavigate="onEvent">
  <%
    pageBean.modifyTable(myTableView);
    myTableView.setUserTypeCellRenderer(new
      TableViewCellRenderer());
  %>
</hbj:tableView>
```

Figure 5: Table View HTMLB Control with Embedded Links

```
int columns = tv.getColumnCount();
for (int i = 1; i <= columns; i++) {
  TableColumn column = tv.getColumn(i);
  if (column.getIdentifier().equals("LinkColumn"))
    column.setType(TableColumnType.USER);
}
```

Figure 6: Setting Table Column Type

First, we put the table view on a JSP. We can use the technique outlined above concerning forms, but in this case, I chose to use the tag version.

The TableView tag looks something like this in Figure 5.

Notice above, the method `modifyTable(myTableView)`

Here is where we designate one or more columns of the table as type "user". The code for this is similar to the following in Figure 6.

Once we have done this, we can set a cell rendered for the column, as we did in Figure 5.

The cell renderer class can be coded as in Figure 7. Once again, we use the Resource Bundle to store text elements, so that the page can be translated into multiple languages.

Generally, we can use the same or similar table view cell renderer throughout an application. We can make it fancy by querying the render-

erContext and using “if” statements to step around undesired code.

We have to remember to code the Page Down function of the table view control. When the Page Down arrow is pressed, a TableNavigationEvent is thrown. From this event, we can get the next row to display. Generally, we store this information in the request and retrieve it in the page bean. In some cases, you may want to put the first visible row in the session, for example, in the event you want the user to hold his place throughout the application. The code for this is as follows in Figure 8.

```
public class TableViewCellRenderer implements ICellRenderer {
    public void renderCell(
        int row,
        int column,
        TableView tableView,
        IPageContext rendererContext) {

        String reference = tableView.getValueAt(row,
            column).toString();

        if (reference != null && reference.trim().length() > 0) {

            String linkName = "linkName";
            IPortalComponentRequest request =
                (IPortalComponentRequest)
                rendererContext.getRequest();
            String linkText =
                request.getResourceBundle().getString("link_text");
            Link link = new Link(linkName, linkText);
            link.setTarget("_blank");
            link.setOnClick("onEvent");
            link.setReference(reference);
            link.render(rendererContext);
        }
    }
}
```

Figure 7: Typical Table View Cell Renderer, Internationalized

```
if (event != null && event instanceof TableNavigationEvent) {
    TableNavigationEvent tne = (TableNavigationEvent) event;
    first_visible_row = tne.getFirstVisibleRowAfter();
}
Integer firstVisibleRow = new Integer(first_visible_row);
request.getNode().putValue("first_visible_row", firstVisibleRow);
```

Figure 8: Page Down Coding for Table View Control

```
AbstractDataType dataBox =
    getPage().getPageContext().getDataForComponentId(
        "tableViewName",
        componentName,
        i);
```

Figure 9: Getting Data From the TableView Control

Embedding Objects in Table View Control: Input Fields and Components

We can use the same approach, as discussed above, to put an input field into a table view control. This input field can be the actual HTMLB input field object, or could also be a drop-down list-box, check-box group, or other HTMLB control where the user adds input. In this case, it is not so clear as to how get the input value off the screen after the user submits the form. I had to search a long time to find the proper method call. SAP’s documentation tells us how to put fields in table view controls, but does not tell us how to get the data out of the table view control after the user has made input.

In this case, we must make use of the following method call in Figure 9.

Using this method, we can query the type of object returned using the Java instance of Key Word, cast the result to the appropriate object, and call the required method of the object. We could also, more concisely, call the method dataBox.getValueAsString(), which will return the user’s input as a String. This is what is most commonly done. Make sure, however, that the dataBox is not null before you call the method.

Notice that the `getPage()` method returns an instance of the `JSP-DynPage`, and is only available in a class that extends the `JSP-DynPage`. I always write a controller for my applications, and then store the `JSP-DynPage` object in a helper that is passed to a command that processes the logic when the user submits a request. This is a true three-tier approach. Also, the above method call requires the name of the component and the row. In order to achieve this, it is best to write a method that names the component as some consistent concatenation of the row and a variable, such as "myInputField_1". This way, we can get the name of the component (after the code that generated the component has completed), and also avoid the runtime error that will occur if two fields are named identically.

We need to develop a technique for storing the input values, especially when the user triggers a `TableNavigationEvent` by pressing the Page Down arrow. We can easily do this by using the method shown above in a loop, and then storing the result in an `ArrayList` that is held in the session. This way, the user's input is not lost when she or he proceeds either forward or back in the screen sequence of the application.

Bear in mind that in all cases of using a `Table View` control, the programmer cannot retrieve the values of text fields in the `Table View` after the `Table Control` has been rendered. We can only access key values and the values of input fields. Therefore, if you need to submit a `Table View` and then read the data in that `Table View`, it is best to store the necessary data as part of the key. I usually do this by making the key a delimited sequence of fields. The `StringTokenizer` class is valuable in this respect. Using this approach, the key can be retrieved from the table, and the required data in the row can be read.

Another approach is to use an input field, and then make that column invisible. You might try this. I have not used this approach, because it requires more programming, but I am fairly certain it will work.

The Scroll Container allows us to embed a page within a page, with scroll bars on the container, to allow the user to see the entire contents of the container.

Using the Scroll Container

This HTMLB control is new to EP6, and I really like it because it gives us a capability that we never had.

Quite often in Web development, we have multi-part forms that constitute a data entry process. When the user completes the multi-part form, he often cannot see the results of the previous step when completing the current step. This is common with things like shopping cart applications, where the user may enter his credit card information on a different page than where he enters the shipping information.

Wouldn't it be nice if the user could see the result of the previous step when completing the current step? I have been asked to do this on quite a few applications, and I have gener-

ally completed this by using a pop-up box. The pop-up box is a less than desirable solution to this problem. First off, it is a little bit annoying. Second, the user has to remember to close the pop-up box, or it will sit out there forever

There are many ways to code the pop-up box. Here is the method that I generally use. The pop-up box is enabled by clicking on a link. The target of the link is a new page. The link reference needs to be a new `iView` contained within the same project and par file, as defined in the `portalapp.xml` file in EP6. In a sense, this is a sub-component of the top level `iView`. The `iView` generally points to the same controller. A new JSP page will need to be prepared, and this JSP should contain JavaScript code to control the sizing of the box, as well as disable the menu bars. The JavaScript code is problematic, as it sometimes works well, but not always. All of this, collectively, is a lot of coding and hassle to generate a pop-up box.

The `Scroll Container` provides a method of avoiding all of this. The `Scroll Container` allows us to embed a page within a page, with scroll bars on the container (to allow the user to see the entire contents of the container). We can, therefore, keep the size of the container relatively small, so that it does not take over the page. Inside the scroll container, we can place summary information that describes what the user entered in the previous step or steps. The scroll container, therefore, allows us to keep our pages flat, and does away with the need to create the pop-up box.

Let me describe this with a little bit of code. First, I place a statement like this in the JSP page that contains the scroll container.

```
<%pageBean.getScrollContainer(theContext);%>
```

In this case, I only display the scroll container if the user has the appropriate data in the session. The code is shown in Figure 9.

In Figure 10, we placed a group on the JSP page. The group contains a Scroll Container. The Scroll Container, in turn, contains a Table View model with two columns. All of the text elements have been internationalized.

In this case, I only display the scroll container if the user has the appropriate data in the session.

Let me say a little bit about internationalization. The location of the resource bundle is contained in the portalapp.xml file. An excerpt of this file is shown in Figure 11.

Using this setup, the localization properties file is located in the /dist/PORTAL-INF/classes directory under your project root in the Eclipse IDE. Note the fully defined class name of the Controller for the application.

Figure 10: Coding the Scroll Container

```
public void getSelectedShipTo(IPageContext context) throws IOException {

    Object ob =
    request.getComponentSession().getValue("stored_data");
    if (ob == null)
        return;
    if (!(ob instanceof List))
        return;

    List list = (List) ob;
    if (list.isEmpty() || list.size() < 1)
        return;

    ScrollContainer scroll = new ScrollContainer();
    scroll.setHeight("75");
    scroll.setWidth("65%");

    //build column names
    Object[] colNames = new Object[2];
    colNames[0] = "COLUMN_NAME_1";
    colNames[1] = "COLUMN_NAME_2";

    //build data
    Object data[][] = new Object[list.size()][2];
    Iterator it = list.iterator();
    int row = 0;
    while (it.hasNext()) {
        Object element = it.next();
        if (element != null && element instanceof MyBean) {
            MyBean bean = (MyBean) element;
            data[row][0] = myBean.getAttribute1();
            data[row][1] = myBean.getAttribute2();
        }
        row++;
    }

    DefaultTableModel model = new
    DefaultTableModel(data, colNames);
    TableView tableView = new TableView("myTableView", model);
    tableView.setFooterVisible(false);

    //setting column titles
    for (int i = 1; i <= tableView.getColumnCount(); i++) {
        tableView.getColumn(i).setTitle(
            request.getResourceBundle().getString(
                tableView.getColumnName(i)));
    }
    scroll.addComponent(tableView);

    Group group = new Group();
    group.setDesign(GroupDesign.SAPCOLOR);
    group.setTitle(
        request.getResourceBundle().getString(
            "my_group_title"));
    group.setWidth(width);
    group.addComponent(scroll);
    group.render(context);
}
}
```

Field Validation Using SAP Supplied Validators

In EP6, SAP provided a collection of field validation classes, such as DateValidator, Length Validator, and Required Validator. We can apply these validators to the input field. The code can be done as shown in Figure 12.

The validator will place a red box around the field in the event that the field contents fail to pass the validation. In most cases, however, it is still necessary to perform server-side validation. For example, in the code above, the validation is performed when the form is submitted. If validation fails, the form will still be submitted, and the user will need to be directed back to the form. I have found no instances where we can easily modify the JavaScript in the validators, nor would it be practical to do this. We can retrieve the validators for the field with the method `getAllValidators()`, which places all the validators in an `ArrayList`. We can then loop through the `ArrayList` and get all the messages from the validators in the event the field has failed validation. Following that, we can submit the user back to the submitted form and display the error messages. We might use the `MessageBar` class to display the messages, but beware that users generally fail to see these messages, because they appear at the base of the screen.

I have not been tremendously successful in using the SAP validators in my own code. Apparently, I am not the only one.

```
<components>
  <component name="default">
    <component-config>
      <property name="ClassName" value="main.Controller"/>
      <property name="ResourceBundleName" value="localization"/>
      <property name="SecurityZone" value="com.sap.portal.pdk/low_safety"/>
    </component-config>
    <component-profile>
      <property name="tagLib" value="/SERVICE/htmlb/taglib/htmlb.tld"/>
      <property name="SystemIdentifier" value="SAP_CRM"/>
    </component-profile>
  </component>
</components>
```

Figure 11: Excerpt from portalapp.xml File

```
public void getDateField(Form form) {

    InputField field = new InputField("test_date_field");
    field.setType(DataType.DATE);
    DateValidator validator = new DateValidator();
    field.setValidator(validator);
    field.setRequiresValidation(true);
    field.setClientEvent(EventTrigger.ON_FORM_SUBMIT,
        validator.getJavascript(request.getLocale()));
    form.addComponent(field);

}
```

Figure 12: Use of a Validator on an Input Field: Date Validator

I have not been tremendously successful in using the SAP validators in my own code. Apparently, I am not the only one, as I have seen messages on the SDN forum concerning problems with validators. The validators usually suffer from the deficiency that they are not able to validate in the manner that is necessary for the business logic of the form, like ensuring that a date is in a specific range, or performing cross-field validation. In most cases, I have chosen to perform my own server-side validation, and, in the event that the field fails

validation, generate my own error messages and display these messages on the screen. I have found server-side coding like this very reliable.


I am going to defer complete coverage of field validation to another article. If some of my readers have had success in using the validators, I would greatly appreciate them sharing some code with me. I will then publish this information so everyone can benefit. I have found no documentation from SAP concerning the use of field validators.

Conclusion

In this article, I have covered a number of techniques to improve the generation of JSP pages using HTMLB. Most of these techniques have focused on keeping the JSP pages as simple as possible, while making extensive use of the embedding possibilities inherent with HTMLB.

I introduced the topic of field validation, and pointed out some of the issues associated with using some of the standard SAP supplied validators. In a future article, I plan to cover this matter in greater detail.

Hopefully, readers of this article will learn a few new tricks, and get some ideas that will help them prepare better code with less effort.

Bill Guderian, *BK Systems, Inc.* Bill is an independent contractor specializing in SAP Enterprise Portals and similar developments involving the Java programming language. He has almost ten years of SAP experience in a wide range of technical and functional areas, including extensive experience as an ABAP developer. Bill has spent the past four years focusing on the space where the SAP applications intersect with the Java programming language. He is a consultant in helping companies develop SAP bolt-on applications written in Java. Bill's email address is Bill.Guderian@SAPtips.com. 

SAPtips *Journal*

The information in our publications and on our Website is the copyrighted work of Klee Associates, Inc. and is owned by Klee Associates, Inc. NO WARRANTY: This documentation is delivered as is, and Klee Associates, Inc. makes no warranty as to its accuracy or use. Any use of this documentation is at the risk of the user. Although we make every good faith effort to ensure accuracy, this document may include technical or other inaccuracies or typographical errors. Klee Associates, Inc. reserves the right to make changes without prior notice. NO AFFILIATION: Klee Associates, Inc. and this publication are not affiliated with or endorsed by SAP AG. SAP AG software referenced on this site is furnished under license agreements between SAP AG and its customers and can be used only within the terms of such agreements. SAP AG and mySAP are registered trademarks of SAP AG. All other product names used herein are trademarks or registered trademarks of their respective owners.