Build More Powerful Web Applications in Less Time with BSP Extensions and the MVC Model

Karl Kessler

Karl Kessler joined SAP AG in 1992 as a member of the Basis modeling group, where he gained experience with SAP's Basis technology. In 1994, he joined the product management group of the ABAP Development Workbench. Since 1997, Karl has been Product Manager for SAP's business programming languages and frameworks.

(complete bio appears on page 26)

With Release 6.10 of the SAP Web Application Server (Web AS), SAP introduced a new approach to web development called Business Server Pages (BSP). For the first time, developers could create web applications by combining HTML directly with ABAP on a single page — a BSP page. A typical 6.10-based BSP application consists of a couple of BSP pages whose data retrieval is coded in ABAP and whose layout is defined via standard HTML, so that any ABAP programmer with even a modest understanding of HTML can easily write web user interfaces and applications using the ABAP Workbench.

SAP realized that this approach was not without room for improvement. BSP pages can quickly grow into large sizes that are difficult to maintain or reuse, and applications can become hard to modularize effectively. Plus, developing user interfaces in HTML can be cumbersome given the large number of tags and options to learn. SAP Web Application Server 6.20[1] provides relief through two key enhancements:

1. **BSP extensions:** Essentially a library of custom tags, BSP extensions allow developers to add complex user interface elements like table views to their pages with only a few lines of code. SAP provides predefined BSP extensions with Web AS 6.20, or you can create your own custom extensions.

2. **Support for the Model-View-Controller (MVC) model:** The well-known MVC model allows you to "compartmentalize" the user interface, business logic, and flow logic of a web application so they can be readily reused and more easily maintained.

[1] Release 6.20 is the technological basis of R/3 Enterprise and other SAP solutions, in contrast to 6.10, which is offered on a standalone basis.

Listing 1: Classic BSP Code for Displaying an Internal Table

```
 1 <%@page language="abap"%>
 2 <html>
 3
 4   <head>
 5     <link rel="stylesheet"
 6          href="../../sap/public/bc/bsp/styles/sapbsp.css">
 7     <title> Flight display </title>
 8   </head>
 9
10   <body class="bspBody1">
11     <table class="bspTbvStd">
12         <th class="bspTbvHdrStd">Carrid</th>
13         <th class="bspTbvHdrStd">Connid</th>
14         <th class="bspTbvHdrStd">Fldate</th>
15     <% loop at flights into flight.%>
16     <% data: str type char10.
17        write flight-fldate to str.%>
18         <tr>
19             <td class="bspTbvCellStd"><%=flight-carrid%></td>
20             <td class="bspTbvCellStd"><%=flight-connid%></td>
21             <td class="bspTbvCellStd"><%=str%></td>
22         </tr>
23     <% endloop.%>
24     </table>
25   </body>
26
27 </html>
```

This article describes and demonstrates how to use these key enhancements,[2] and provides examples and tips to help you avoid common difficulties. By the time you finish reading this, you will understand how, when, and why to use both BSP extensions and the MVC model to build more powerful, more maintainable, and more reusable web applications with less code and in less time.[3] Let's get started!

[2] BSP extensions and the MVC model are new, optional technologies for building better BSP applications. The "standard" BSP approach is still supported, so 6.10 applications will still work on the 6.20 platform.

[3] Basic knowledge of the BSP programming model is assumed. For an introduction to BSP programming, see my previous article, "A Developer's Guide to Creating Powerful and Flexible Web Applications with the New Web Application Builder," in the January/February 2002 issue of this publication.

What Exactly Do BSP Extensions Do?

Suppose you want to write a simple BSP application to output a list of flights from the flight table of the SAP IDES training system. If written in a 6.10 system (i.e., prior to the advent of BSP extensions), the code for your page might look something like **Listing 1**.

So what's wrong with Listing 1? While it technically will work, it is not ideal. As you can see, standard HTML is mixed with ABAP code, which makes the application difficult to read. The construction of the BSP page is also highly procedural, and does not leverage the declarative nature of HTML. Plus, since

Listing 2: Rewrite of Listing 1 Using the HTMLB Extension

```
 1 <%@page language="abap" %>
 2 <%@extension name="htmlb" prefix="htmlb" %>
 3 <htmlb:content>
 4   <htmlb:page>
 5     <htmlb:form>
 6       <htmlb:tableView id    = "t1"
 7                        table = "<%= flights %>" >
 8       </htmlb:tableView>
 9     </htmlb:form>
10   </htmlb:page>
11 </htmlb:content>
```

HTML treats all data as characters, you must program data type conversions manually, on your own.

For example, in line 17 of Listing 1, note that you must first write (with the ABAP *write* statement) the date field to a temporary string variable before outputting it in line 21. This ensures that the date is output in a format preferred by the user (e.g., 01/20/2003), instead of the default raw database format — YYYYMMDD without any punctuation characters (e.g., 20030120). As the page becomes more complex, this can add up to a lot of extra work.

You will notice another deficiency when you first pull up the page in a browser: the complete table of data is displayed, without any scrolling functionality available. That is, all of the flights are displayed at the same time in the browser, instead of in more manageable "pages" with scroll buttons. If you've done web development before, you will realize how much HTML (and possibly JavaScript) code would be required to implement this paging approach. (More on scrolling functionality in a later section.)

BSP extensions offer a cleaner, more compact way to write BSP applications. Take a look at **Listing 2**, where you can see how I've used the predefined HTMLB extension to display the example flight list. Notice how much less code is used compared to Listing 1, and how the page looks much more like HTML, with no loops or other programming constructs to detract from its readability.

> ✓ ***Tip***
>
> *Another good reason to work with the HTMLB extension tags instead of plain HTML tags is that HTMLB is fully analyzed by the BSP compiler. For example, the compiler checks whether you have specified all mandatory attributes and whether inner tags are handled correctly. In contrast, the "plain" BSP approach offers only a rough syntax check that looks primarily at the ABAP code inside the BSP page, leaving errors that will appear only at runtime. Yet another aspect is portability — using HTMLB ensures that your HTML code is rendered according to the underlying platform, while plain BSP pages are almost always browser-dependent.*

Let's take a closer look at what is happening in Listing 2. On line 2, you'll notice a new BSP directive, *@extension*, which tells the BSP compiler that I wish to use the HTMLB extension in this page. When getting started, you must first specify which BSP extension(s) you would like to use. The

predefined HTMLB extension contains tags that encapsulate common user interface elements, such as input fields, pushbuttons, and table views.

Using the HTMLB extension tags is easy. Many of them mirror their HTML counterparts (e.g., instead of using the HTML *<form>* tag in your page, you now use the *<htmlb:form>* tag, as in line 5 of Listing 2).

When declaring the extension(s) to be used, you must specify an individual prefix to refer to a BSP extension, as I have done on line 2. In this example, I am using the prefix *htmlb*. This naming convention comes in handy if you are working with several extensions simultaneously. Appearing next is an *<htmlb:content>* tag (line 3). This tag is responsible for inserting an *<HTML>* opening tag in the page (plus others), and is required. All other HTMLB tags are placed inside this tag.

Next, you would typically specify two HTMLB tags: *<htmlb:head>* for the head and *<htmlb:body>* for the body of your document. However, if you are writing a simple page, like the one in the example here — e.g., a page that doesn't include JavaScript or Cascading Style Sheet (CSS) styles in the HTML *<head>* section — you can simply use the *<htmlb:page>* tag (line 4), which automatically generates both the header and body tags.

Next comes an *<htmlb:form>* tag (line 5), which corresponds to an HTML *<FORM>* tag.[4] Since the example table view will require interactive capabilities (i.e., scrolling functionality), I embed the *<htmlb:tableView>* tag inside the form (lines 6-8). The *<htmlb:tableView>* tag shows attributes *id* and *table*. The *id* attribute uniquely identifies the HTML table control within the HTML page (e.g., in case you need to reference it later via event handling). The *table* attribute refers to the internal table to be displayed.

[4] If you are new to HTML, *<FORM>* tags are used when building interactive pages with elements like input fields and buttons that gather information from the user.

> ✓ ***Tip***
>
> *In Listing 2, note that I am passing the name of an internal table, "flights," to the <htmlb:tableView> element (lines 6-8). This internal table was loaded by ABAP code placed in the page's Initialization event handler rather than in the BSP code itself. Then, to make the internal table ("flights") available within the BSP page, I declare it as a page attribute on the "Page Attributes" tab of the BSP page editor. Separating the data retrieval code from the layout code in this way is considered good programming practice, and makes applications easier to maintain and reuse.*

If you look at Listing 2 as a whole, I hope it starts to become clear as to why BSP extensions hasten web application development. The code is fully declarative and contains just a reference to a data source (the internal table to be displayed). The code is much more compact than in Listing 1 and can be extended in a variety of ways that I will discuss in this article.

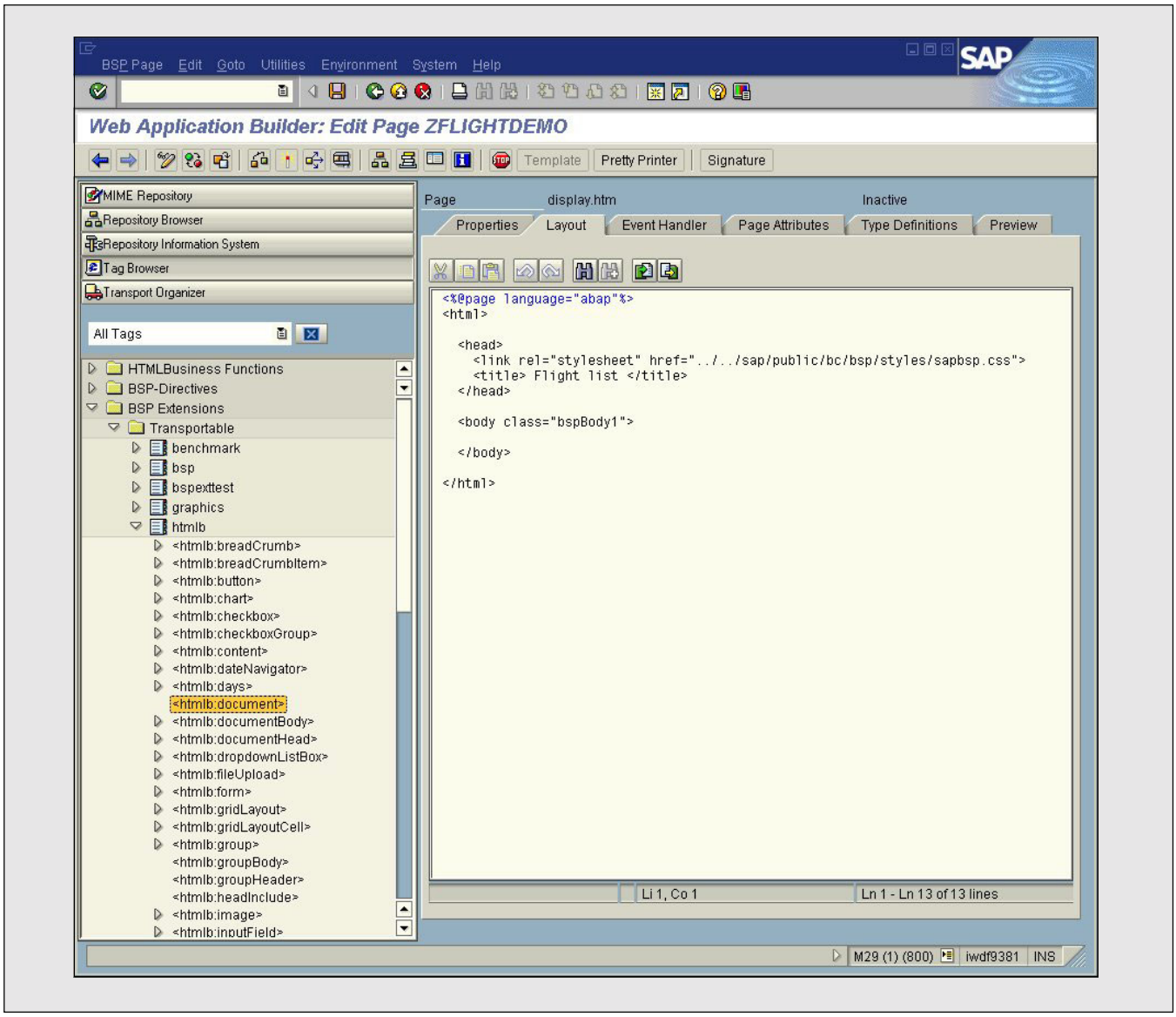
Creating a Web Application Using BSP Extensions

Let's now create the well-known flight demo application from scratch using BSP extensions (if you have access to a Web AS 6.20 system, I encourage you to follow along and try these steps for yourself):

1. Log on to Web AS 6.20. Go to the ABAP Workbench's Object Navigator (transaction *SE80*) and select *BSP Application* from the dropdown list box at the upper left of the screen. Enter a name for the new BSP application in the input field below (*zflightdemo* in the example).

2. Click the display button (). The system will ask you whether to create this application. Choose *Yes*. The system will create a new

Figure 1 *The Tag Browser and Page Editor Display*



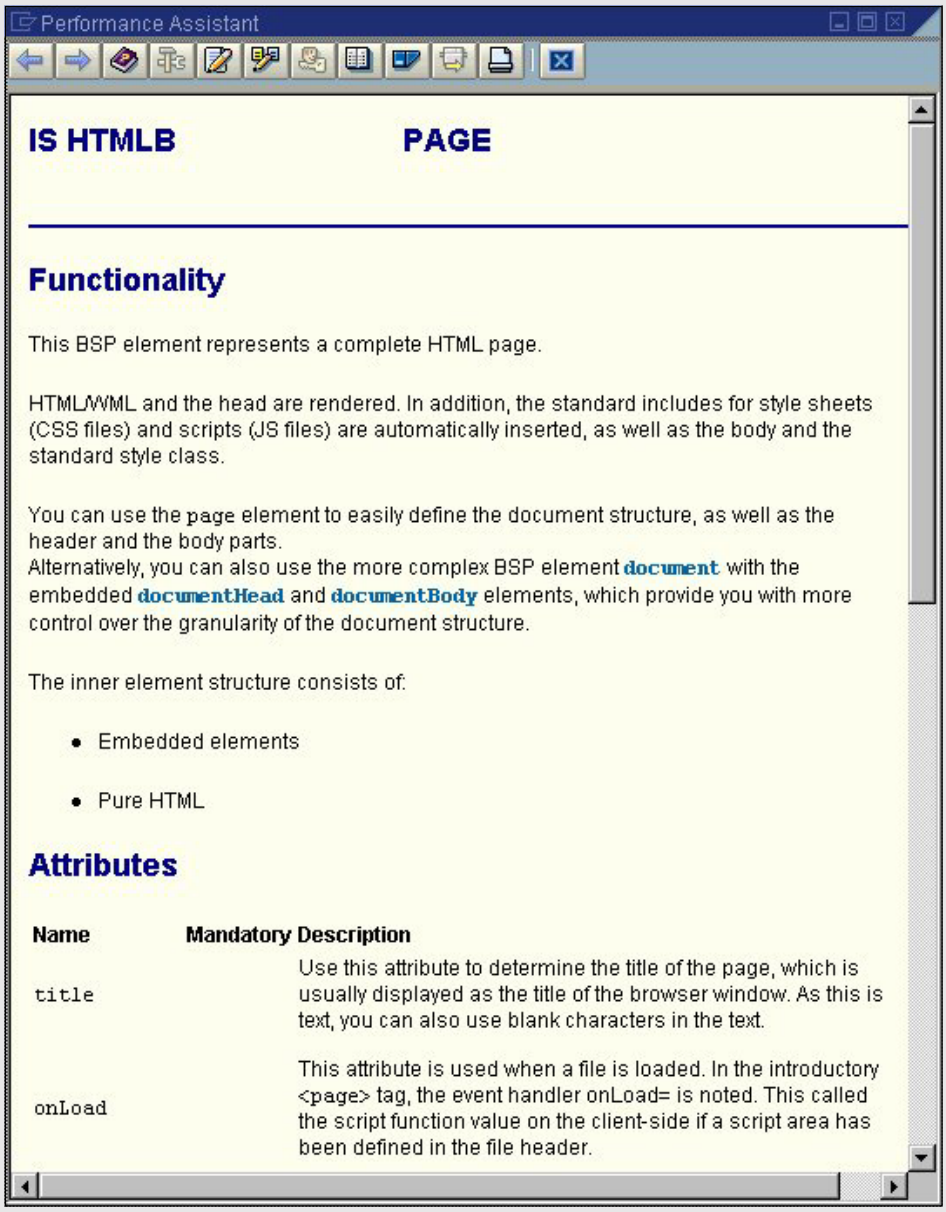
BSP application and display its outline in the lower left pane of the screen.

3. Position the cursor on the new BSP application name, right-click on it to open the context menu, and choose *Create* → *Page*. A dialog box pops up asking for the name of the page (and the page's flow logic). Enter a name for the page (*display.htm* in the example), along with a short description of the page (e.g., *Display*), and press the *Enter* key (or click ✔). The system creates the page with some default HTML code on the *Layout* tab of the BSP page editor.

4. You can now replace the default code by typing in the code from Listing 2, but there is an easier way to do this. Choose the *Tag Browser* button to open the Tag Browser[5] (see **Figure 1**).

[5] The Tag Browser displays all supported tags for the Web AS together with their attributes in a hierarchical display that you can use for drag and drop. You can also open the Tag Browser via *Environment* → *Web Tools* → *Tag Browser*.

Figure 2 HTMLB Documentation Display



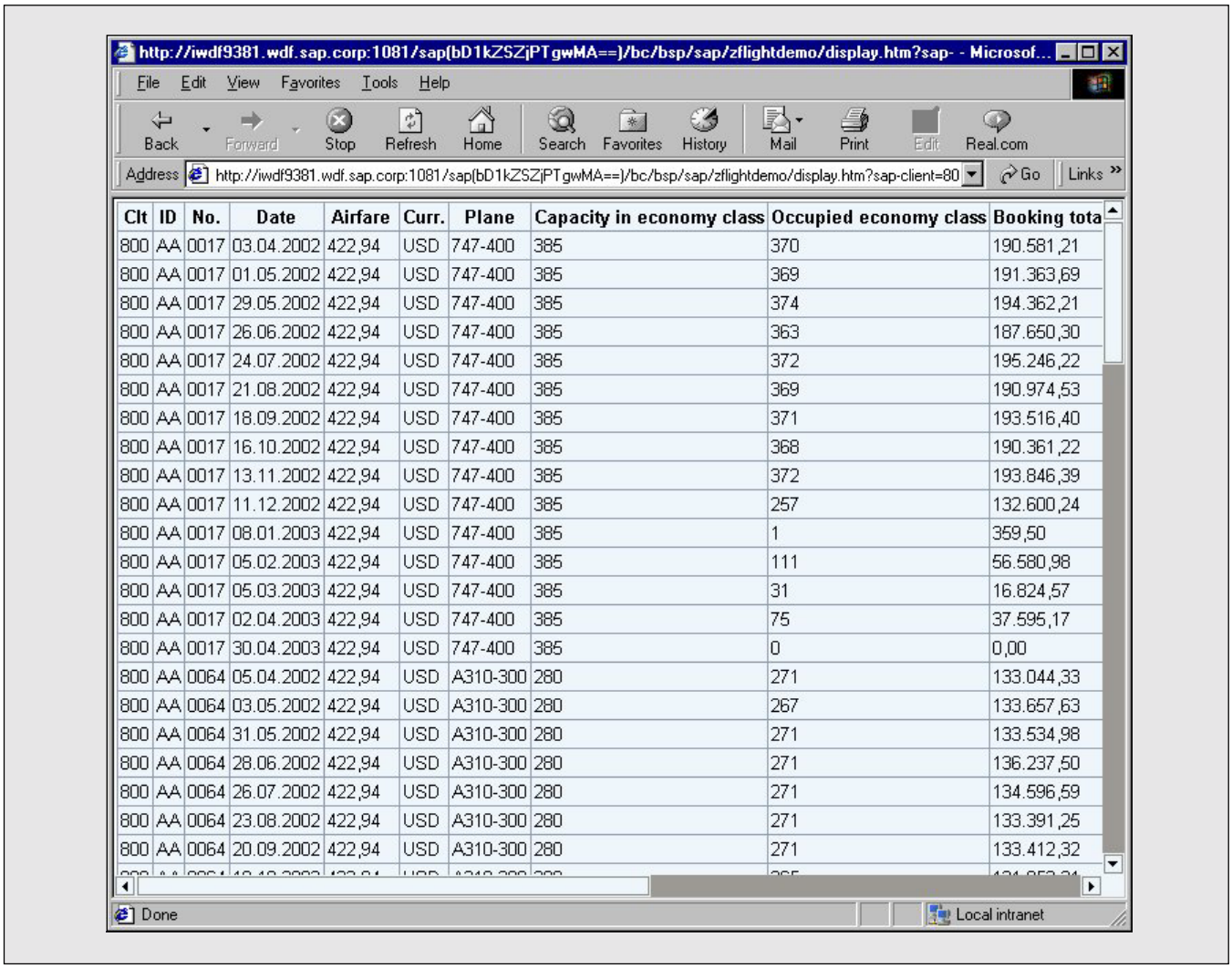
The system will then list all the available libraries in a hierarchical display on the left side of the screen.

5. Expand the node *BSP Extensions*. The system displays transportable and local extensions if any exist.

6. Expand the node *htmlb*. You will see the complete list of available HTMLB extension tags. Scroll down and double-click on <*htmlb:page*>. The system displays the online documentation for the selected tag (see **Figure 2**).

7. Drag the <*htmlb:content*> tag and drop it in

Figure 3 *The Flight List Display*



Clt	ID	No.	Date	Airfare	Curr.	Plane	Capacity in economy class	Occupied economy class	Booking tota
800	AA	0017	03.04.2002	422,94	USD	747-400	385	370	190.581,21
800	AA	0017	01.05.2002	422,94	USD	747-400	385	369	191.363,69
800	AA	0017	29.05.2002	422,94	USD	747-400	385	374	194.362,21
800	AA	0017	26.06.2002	422,94	USD	747-400	385	363	187.650,30
800	AA	0017	24.07.2002	422,94	USD	747-400	385	372	195.246,22
800	AA	0017	21.08.2002	422,94	USD	747-400	385	369	190.974,53
800	AA	0017	18.09.2002	422,94	USD	747-400	385	371	193.516,40
800	AA	0017	16.10.2002	422,94	USD	747-400	385	368	190.361,22
800	AA	0017	13.11.2002	422,94	USD	747-400	385	372	193.846,39
800	AA	0017	11.12.2002	422,94	USD	747-400	385	257	132.600,24
800	AA	0017	08.01.2003	422,94	USD	747-400	385	1	359,50
800	AA	0017	05.02.2003	422,94	USD	747-400	385	111	56.580,98
800	AA	0017	05.03.2003	422,94	USD	747-400	385	31	16.824,57
800	AA	0017	02.04.2003	422,94	USD	747-400	385	75	37.595,17
800	AA	0017	30.04.2003	422,94	USD	747-400	385	0	0,00
800	AA	0064	05.04.2002	422,94	USD	A310-300	280	271	133.044,33
800	AA	0064	03.05.2002	422,94	USD	A310-300	280	267	133.657,63
800	AA	0064	31.05.2002	422,94	USD	A310-300	280	271	133.534,98
800	AA	0064	28.06.2002	422,94	USD	A310-300	280	271	136.237,50
800	AA	0064	26.07.2002	422,94	USD	A310-300	280	271	134.596,59
800	AA	0064	23.08.2002	422,94	USD	A310-300	280	271	133.391,25
800	AA	0064	20.09.2002	422,94	USD	A310-300	280	271	133.412,32

the page editor for the *display.htm* page. The system will automatically insert the corresponding opening and closing tags. Similarly, you can also drag and drop the *<htmlb:page>*, *<htmlb:form>*, and *<htmlb:tableView>* tags to the page editor session.

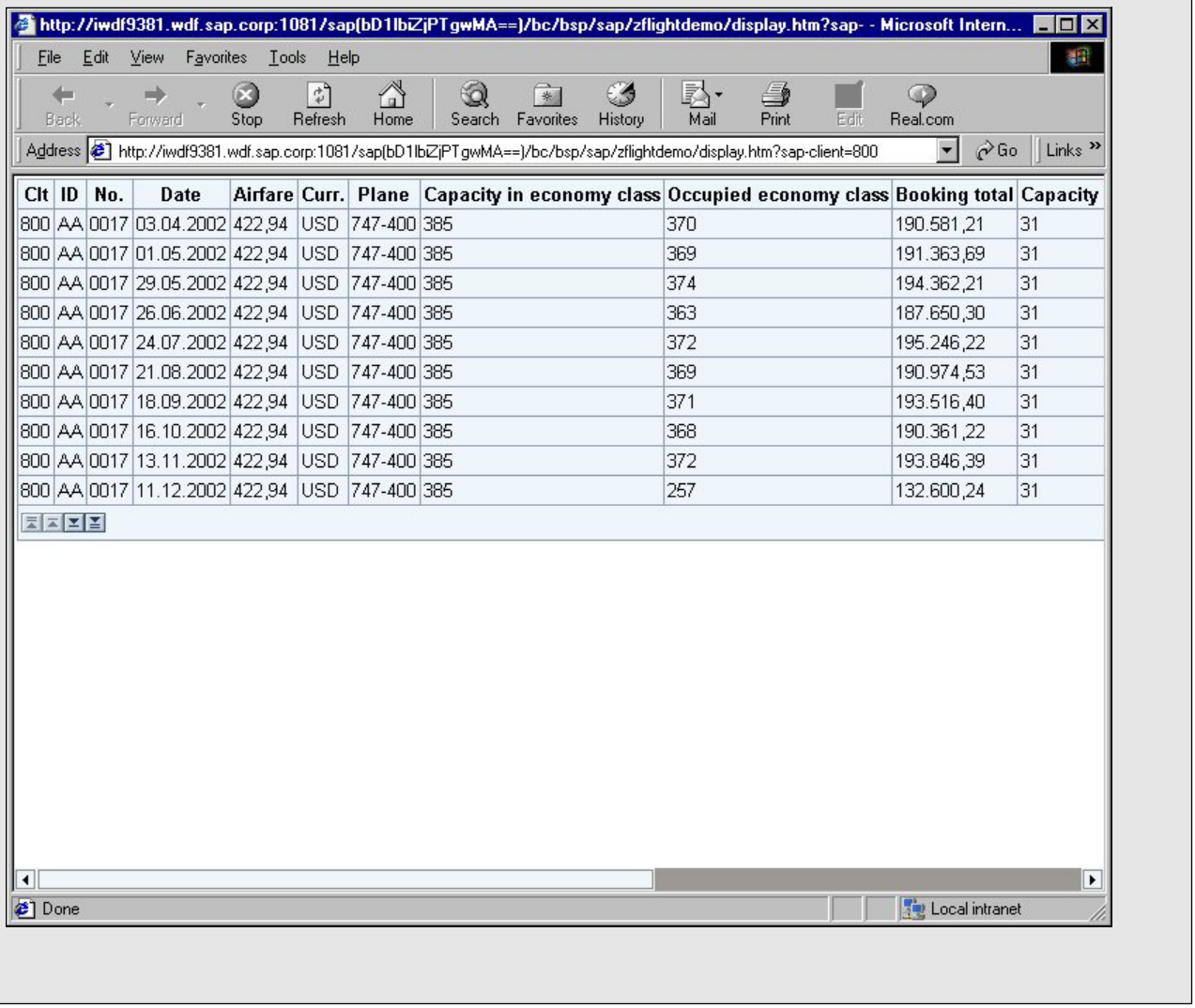
8. Expand the tag *<htmlb:tableView>*. All the attributes of the *<htmlb:tableView>* tag are displayed. As with the tags in the previous step, you can drag and drop the attributes to the page editor session.

9. Go to the *Page Attributes* tab and create a page attribute named *flights* of type *FLIGHTTAB*, and provide the following code for the event *OnInitialization*:

```
Select * from sflight into
  corresponding fields of table
  flights up to 100 rows.
```

10. Save ( ), activate ( ), and run the BSP application by pressing *F8*. The system automatically starts the web browser and displays the flight list, as shown in **Figure 3**.

Figure 4 Flight List Display with Scroll Buttons Added



Clt	ID	No.	Date	Airfare	Curr.	Plane	Capacity in economy class	Occupied economy class	Booking total	Capacity
800	AA	0017	03.04.2002	422,94	USD	747-400	385	370	190.581,21	31
800	AA	0017	01.05.2002	422,94	USD	747-400	385	369	191.363,69	31
800	AA	0017	29.05.2002	422,94	USD	747-400	385	374	194.362,21	31
800	AA	0017	26.06.2002	422,94	USD	747-400	385	363	187.650,30	31
800	AA	0017	24.07.2002	422,94	USD	747-400	385	372	195.246,22	31
800	AA	0017	21.08.2002	422,94	USD	747-400	385	369	190.974,53	31
800	AA	0017	18.09.2002	422,94	USD	747-400	385	371	193.516,40	31
800	AA	0017	16.10.2002	422,94	USD	747-400	385	368	190.361,22	31
800	AA	0017	13.11.2002	422,94	USD	747-400	385	372	193.846,39	31
800	AA	0017	11.12.2002	422,94	USD	747-400	385	257	132.600,24	31

If you look at the flight list table in Figure 3, you see that each column is formatted appropriately for its data type, as declared in the ABAP Data Dictionary. Suitable header text for each field is also automatically retrieved from the Data Dictionary definition. Compare this to the plain BSP page in Listing 1, where you have to hard-code the column titles in the layout (see lines 12-14 in Listing 1).

Now that you've got some background on using BSP extensions, let's take a look at some of the more advanced ways you can use them to enhance your web applications.

Enhancing a Web Application with BSP Extensions

To further explore the potential of BSP extensions, let's add a few bells and whistles to the example flight demo application.

Listing 3: Limit the Columns Displayed with the <htmlb:tableViewColumn> Tags

```
 1 <%@page language="abap" %>
 2 <%@extension name="htmlb" prefix="htmlb" %>
 3 <htmlb:content>
 4   <htmlb:page>
 5     <htmlb:form>
 6       <htmlb:tableView id = "t1"
 7                        visibleRowCount = "10">
 8                        table = "<%= flights %>" >
 9         <htmlb:tableViewColumns>
10           <htmlb:tableViewColumn columnName = "carrid" />
11           <htmlb:tableViewColumn columnName = "connid" />
12           <htmlb:tableViewColumn columnName = "fldate" />
13           <htmlb:tableViewColumn columnName = "details"
                                        type = "user" >
14             <htmlb:button id="$connid$" text="Details" />
15           </htmlb:tableViewColumn>
16         </htmlb:tableViewColumns>
17       </htmlb:tableView>
18     </htmlb:form>
19   </htmlb:page>
20 </htmlb:content>
```

Restricting the Display and Adding Scrolling

First, let's make large data sets more manageable by restricting the number of rows that can be displayed at once in the browser. The attribute *visibleRowCount* of the <*htmlb:tableView*> tag limits the rows that are displayed. When including this attribute in an <*htmlb:tableView*> tag, the system automatically adds scroll buttons () to the bottom of the table (see **Figure 4**). Contrast this to the plain BSP approach, where you have to code the scrolling logic yourself!

Next, let's limit the display shown in the browser to only three columns (fields) of the internal table. (By default, the <*htmlb:tableView*> tag displays all columns.) **Listing 3** shows how this change is easily accomplished by placing <*htmlb:tableViewColumn*> tags within the <*htmlb:tableView*> tags. See how easy it is to use BSP extensions? All you need to quickly build feature-packed applications is knowledge of the available BSP tags and attributes!

Finally, to insert a variable column that is not present in the internal table, we can embed a user-defined column, as I have done in line 14 of Listing 3. Here, I've added a column that contains a pushbutton for requesting details on individual flights. Each of the buttons gets its own ID based on the value of the *connid* column. Note that in this context, to have the system evaluate *connid* as a variable instead of as a constant, you must surround it with dollar signs (*$*).

Listing 4: Layout of the Search Page

```
 1 <%@page language="abap" %>
 2 <%@extension name="htmlb" prefix="htmlb" %>
 3 <htmlb:content>
 4   <htmlb:page>
 5     <htmlb:form>
 6       <htmlb:label for = "carrier"
 7                    id = "lab1"
 8                    text = "Carrier" />
 9       <htmlb:inputfield id = "carrier" />
10       <htmlb:button id = "button"
11                     onClick = "search"
12                     text = "Go"
13
14     </htmlb:form>
15   </htmlb:page>
16 </htmlb:content>
```

> ✓ ***Tip***
>
> *The BSP extensions framework provides variables that can be useful in your pages, including a standard unique ID named "TVCID" (table view counter ID) that you can use for identification purposes using the "$...$" syntax. For example, if you do not have a unique field on hand, like "connid" in the example, you can use "$TVCID$," which generates consecutive unique IDs that can be used in event handling.*

Adding a Search Page

To make the flight demo even more useful, let's add an interactive search page to enable users to narrow their searches by carrier ID. First, let's define the page's layout and provide a suitable event handler.

To begin, create a new page called *search.htm* and insert the code shown in **Listing 4**. Again, you can either use the Tag Browser or type in the code directly. **Figure 5** shows what the finished page looks like when accessed from a web browser.

In Listing 4, you'll notice three new HTMLB tags: *<htmlb:inputfield>*, which defines an input field called *carrier*; *<htmlb:label>*, which produces a label with the caption *Carrier* to its left; and *<htmlb:button>*, which renders the form's submit button (labeled with the caption *Go*). The value of the *onClick* attribute (*search* in this case) is passed as a parameter to the registered HTMLB event handler, described in the next section.

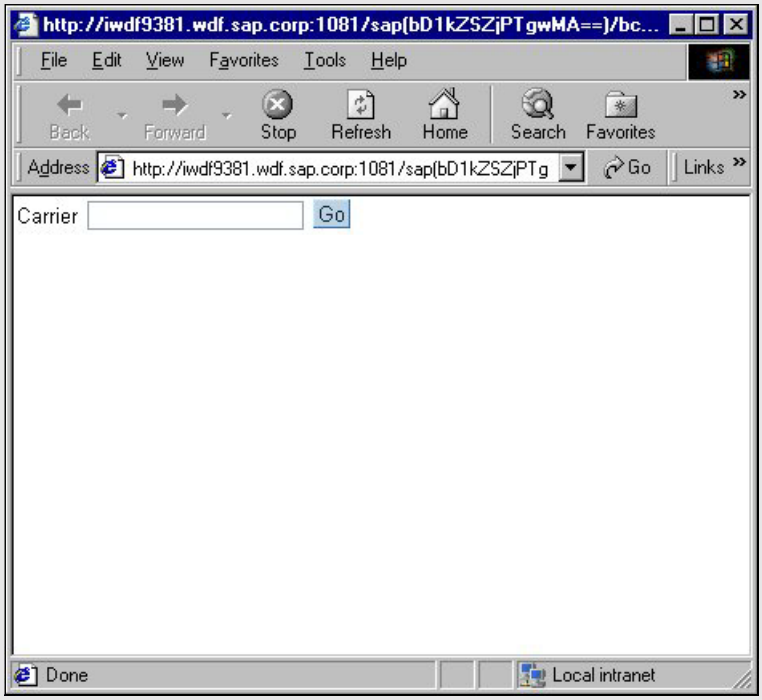
Handling Events

All HTMLB event handling takes place in the standard BSP event *OnInputProcessing*. Select the *Event Handler* tab of your *search.htm* page and open the page editor. Enter the event handling code shown in **Listing 5**.

The code shown in Listing 5 is straightforward, so I will only point out a few key things to note:

- Line 2 obtains a reference to the HTMLB event object (*cl_htmlb_event*), which contains important event information, such as which button the user pressed.

Figure 5 The Search Page Displayed in a Web Browser



Listing 5: Event Handling Code for the Search Page

```
 1 OnInputProcessing
 2 data : event type ref to cl_htmlb_event,
 3        car type string.
 4
 5 class cl_htmlb_manager definition load.
 6
 7 IF event_id = CL_HTMLB_MANAGER=>EVENT_ID.
 8    event = CL_HTMLB_MANAGER=>get_event( request ).
 9    If event->name = 'button'.
10       car = request->get_form_field( 'carrier' ).
11       navigation->set_parameter( name = 'carrier'
12                                  Value = car ).
13       navigation->goto_page( 'display.htm' ).
14    Endif.
15 Endif.
```

> ✓ ***Tip***
>
> *Instead of handling all HTMLB events in the OnInputProcessing event, it is also possible to register an event handler. You must implement your own class (e.g., CL_HTMLB_EVENT_EXAMPLE in the code example below) that implements the IF_HTMLB_EVENT interface. The HTMLB manager (line 3 below) will then call your event handler. This offers a better structural approach if you have to handle various events that otherwise would all be present in one code section.*

```
1 DATA: event_handler TYPE REF TO CL_HTMLB_EVENT_EXAMPLE.
2   CREATE OBJECT event_handler.
3   CL_HTMLB_MANAGER=>dispatch_event(
4     request       = runtime->server->request
5     event_handler = event_handler
6     page_context  = page_context ).
```

- Line 7 checks whether the standard parameter *event_id* indicates that this event is an HTMLB event. This is purely precautionary since in the example we have not mixed HTML submit buttons and HTMLB objects.
- Line 9 determines if the user clicked the one (and only) submit button on the example search page (the *Go* button) using the *event->name* property.
- Line 10 retrieves the value the user entered in the *carrier* input field using the standard BSP *request* object.
- Finally, via method calls on the *navigation* object (*set_parameter* and *goto_page*), lines 11-13 pass the entered carrier ID, along with control, to the *display.htm* page developed earlier.

To complete the enhancements, add an automatic page attribute named *carrier* to *display.htm*[6] and modify the *WHERE* clause of the *OnInitialization* event (coded earlier in step 9 on page 9) to narrow its selection to only that for the carrier ID specified by the user.

[6] Page attributes can be maintained on the *Page Attributes* tab of the BSP page editor (*SE80*).

Building Your Own BSP Extensions

You have seen how the predefined HTMLB extension provides a powerful set of tags and features to simplify tedious tasks like table formatting (the sidebar on the next page examines some of the details of this particular extension). Nevertheless, you will inevitably encounter opportunities to develop your own objects (e.g., specialized graphics or custom elements like an address element), and you may wish to encapsulate this work in your own "library" of elements (i.e., your own extension).

> ✓ ***Note!***
>
> *While it is not difficult to create a new, custom BSP extension with its own tags, developing a rich toolkit like the HTMLB extension is a major undertaking.*

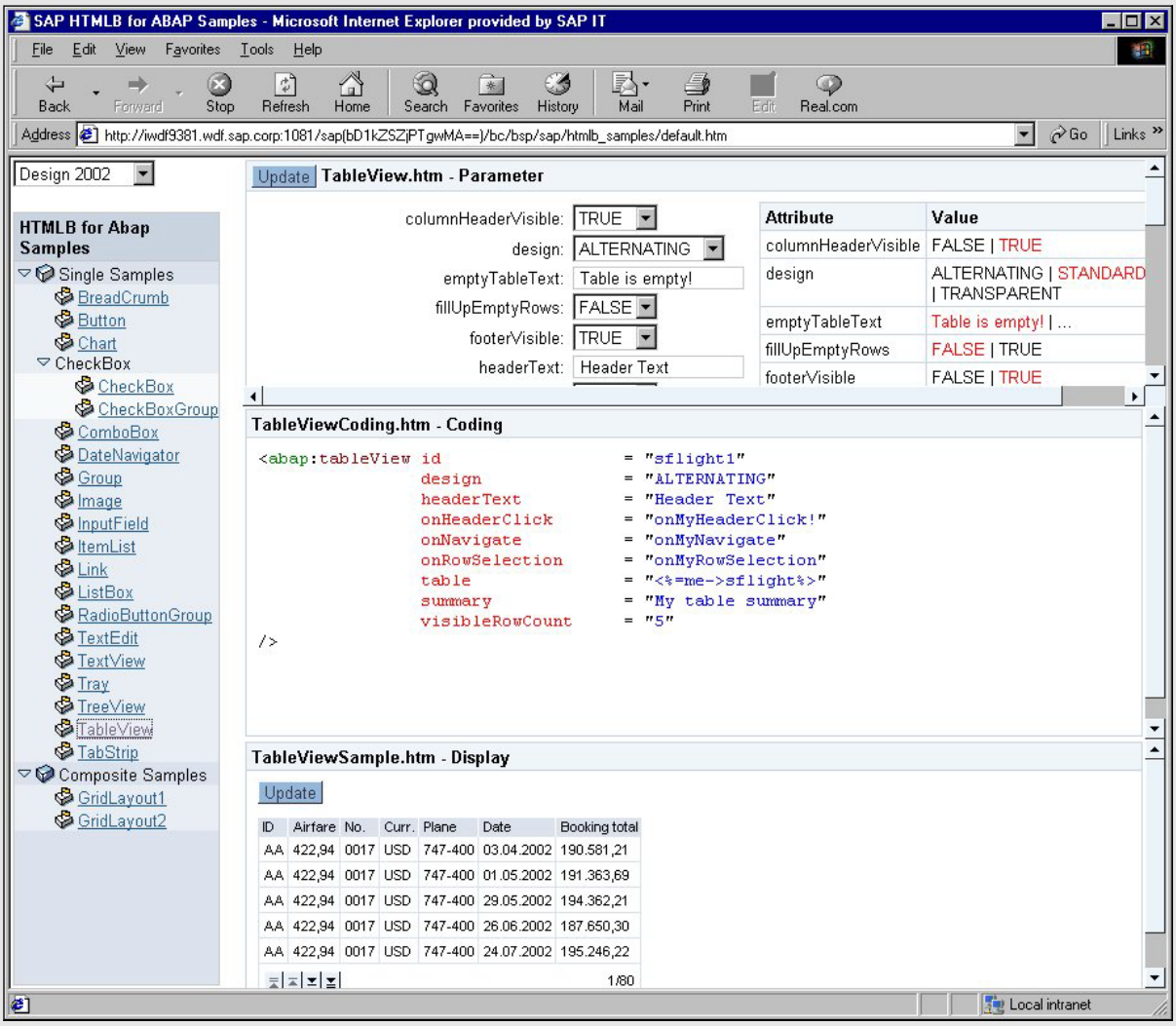
A BSP extension is simply a library of BSP elements identified by tag names. For each element

Delving into the Details of the HTMLB Extension

Inside the ABAP Workbench's Object Navigator (transaction *SE80*), you will find a demo BSP application (shown in the screenshot below) called *HTMLB_SAMPLES*. This sample application demonstrates all the various HTMLB tags.

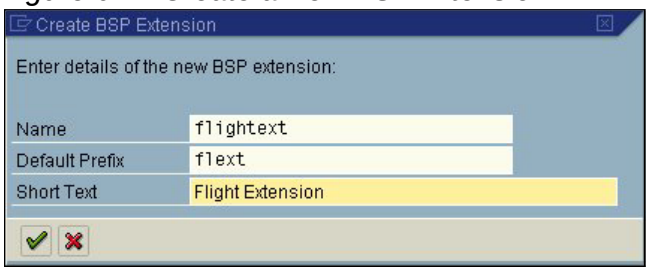
Once you open the demo application, on the left-hand side of the screen you see a navigation frame that allows you to choose from samples of the different HTMLB tags, including the tags for elements such as an input button, a radio button, and a table view. The right-hand side is divided into three parts: the top frame allows you to modify the attributes of each tag, the bottom frame shows the immediate effects of any modifications of the tags, and the middle frame displays the corresponding source code.

Especially for those new to BSP extensions, this demo application is valuable for exploring and mastering the library of available BSP extensions.



you must provide a handler class to render it. The handler class accommodates any attributes the element will support (e.g., the *onClick* attribute of the <*htmlb:button*> tag) through public class attributes that can be subsequently accessed from within the element's rendering method(s).

Figure 6 Create a New BSP Extension



To create your own BSP extension:

1. Start the ABAP Workbench's Object Navigator (transaction *SE80*) and select *BSP Extension* from the dropdown list. Enter a new extension name in the upper left of the screen (*zflightext* here).

2. Click the button. On the resulting pop-up (see **Figure 6**), type in a prefix (e.g., *flext*) and some short text (e.g., *Flight Extension*), and then press

Figure 7 Create an Extension Element

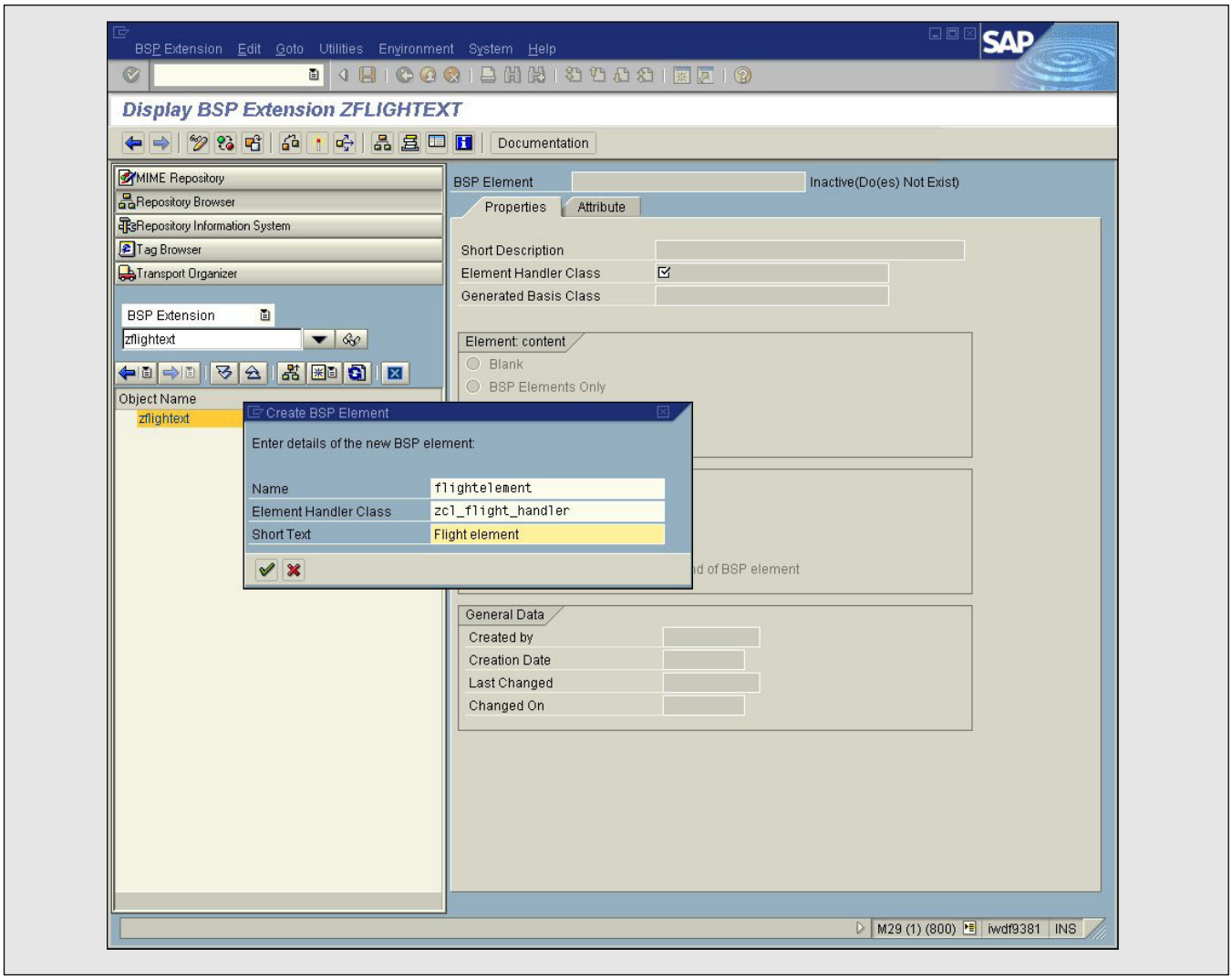
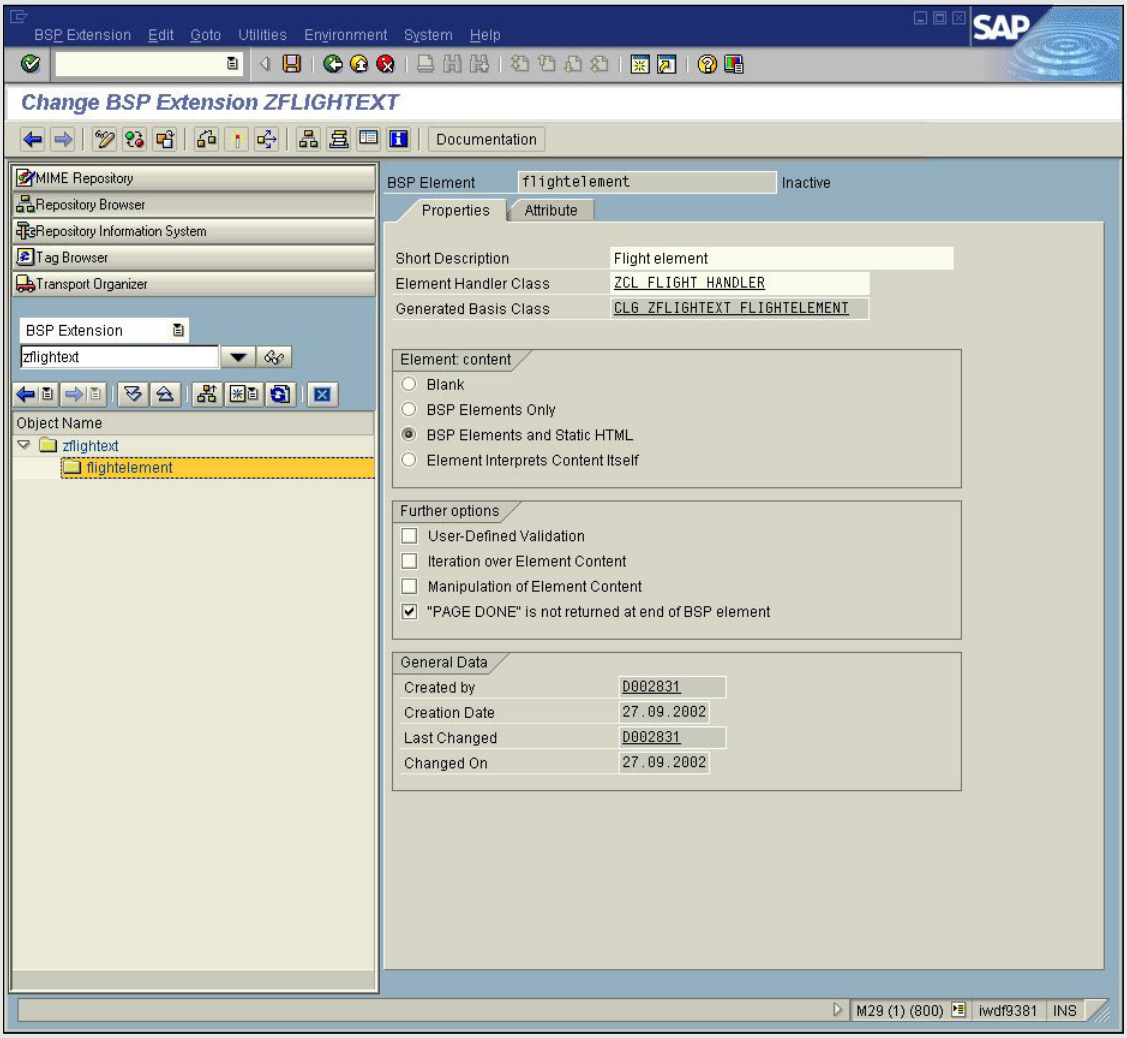


Figure 8 *Basic BSP Extension Properties*



the *Enter* key (or click ✔) to return to the Object Navigator screen (*SE80*).

3. Position the cursor on the newly created BSP extension (*zflightext* in the example), right-click on it to open the context menu, and choose *Create → BSP element*.

4. On the resulting pop-up (see **Figure 7**), type in the element name (*flightelement*), the name of the handler class (*zcl_flight_handler*), some short text (*Flight element*), and press the *Enter* key (or click ✔). The system displays the basic properties of your BSP extension on the right-hand side of the *Properties* tab (see **Figure 8**).

5. Double-click on the element handler class (*ZCL_FLIGHT_HANDLER*). The system navigates to the Class Builder, where you can see the derived methods of the BSP extension framework.

6. Click the redefine button ( ) to override the method *DO_AT_BEGINNING*. The system opens the ABAP Editor, where you type in the following code:

```
...
data : out type ref to
if_bsp_writer,
       out = get_previous_out( ).
       out = print_string( 'my
             custom tag' ).
```

   The code first obtains a reference to the HTML output stream and adds to it the string constant *my custom tag* (note that you could add any HTML code you wish here).

Once you save and activate your changes in the Class Builder, you can open the Tag Browser and expand your newly defined BSP extension. As with elements in the HTMLB extension, you can now drag and drop BSP elements from your extension onto a BSP page. If you run your page, the custom string defined in the code above (*my custom tag*) will be rendered in your page.[7]

Helpful Hints for Using BSP Extensions

Here are some helpful hints for when you begin working with BSP extensions on your own:

☑ For the purpose of readability, use the same design both for input fields and related pushbuttons.

☑ You can significantly reduce the rendering time of the date navigator if you specify a small number of months (e.g., one or two) for each page display.

☑ When working with table views, the system automatically fetches header texts from the Data Dictionary. If you want to use your own text, just overwrite the provided header texts using the <*htmlb:tableViewColumn*> tag (and set the *title* attribute).

☑ Always supply the *onClick* or *onClientClick* events when you define an input button. Otherwise the button will be rendered inactively.

☑ Use the <*htmlb:documentHead*> tag to define embedded JavaScript and CSS styles. Be careful with JavaScript — it is highly browser-dependent and may cannibalize all your HTMLB efforts. The <*htmlb:documentBody*> tag is the container for all your visible HTMLB tags and any standard HTML. Again, the BSP compiler does not check the standard HTML portion for syntactical correctness or browser compatibility.

☑ Use dropdown list boxes to display small sets of selections. Large data sets should always be displayed using the <*htmlb:tableView*> tag.

☑ Take care of accessibility constraints. For example, when displaying images, always provide the *alt* attribute as an additional help text that is accessed by screen reader software.

☑ BSP extensions provide helpful rendering capabilities, but using plain BSP pages is still a good idea in certain scenarios (e.g., if your page layout is done externally). You can combine pages that contain plain HTML with those containing only HTMLB.

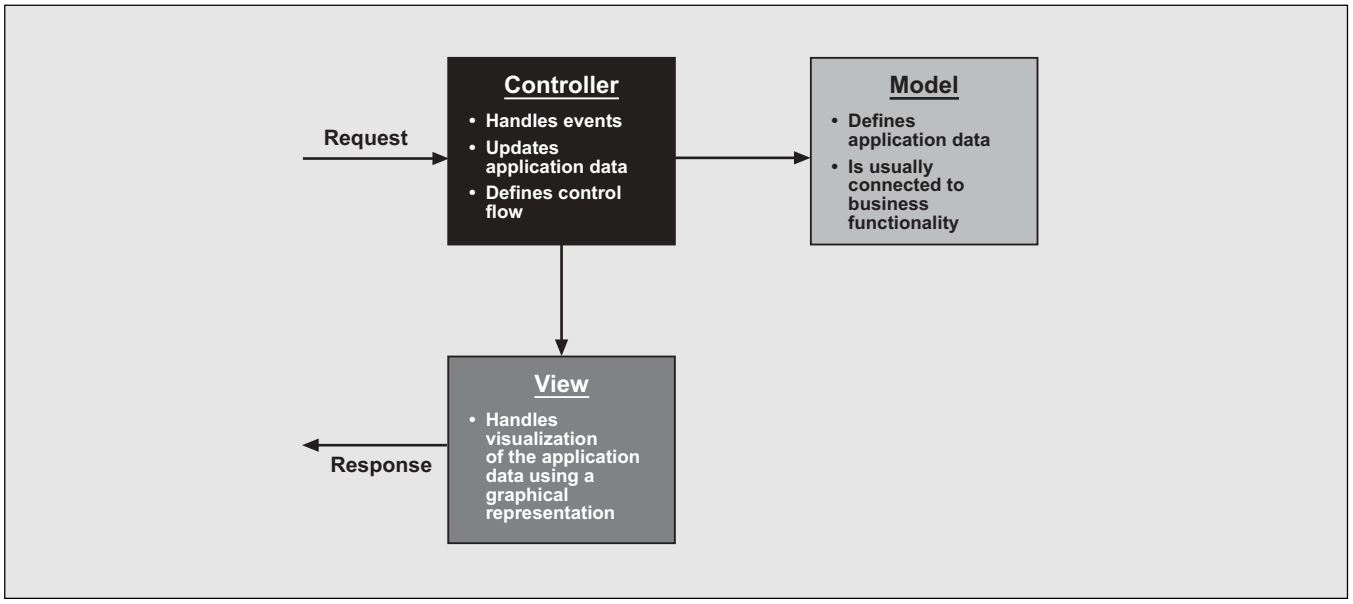
☑ Use the Tag Browser to add tags. Tags are case-sensitive; they all start with a lowercase letter (e.g., <*htmlb:checkbox*>). For structuring large names, uppercase letters are used (e.g., <*htmlb:checkboxGroup*>). Using the Tag Browser ensures that capitalization is always correct.

The MVC Model

While BSP extensions go a long way toward helping you streamline individual BSP pages, they don't address the larger issue of how to structure an entire

[7] A complete discussion of the BSP extension framework is beyond the scope of this article. You can find more information in the SAP Web AS 6.20 online documentation at **http://help.sap.com** (select *SAP Web Application Server → Web Applications → Web Application Server → Web Applications and Business Server Pages→ Programming Model → BSP Extensions*).

Figure 9 An Overview of the MVC Model



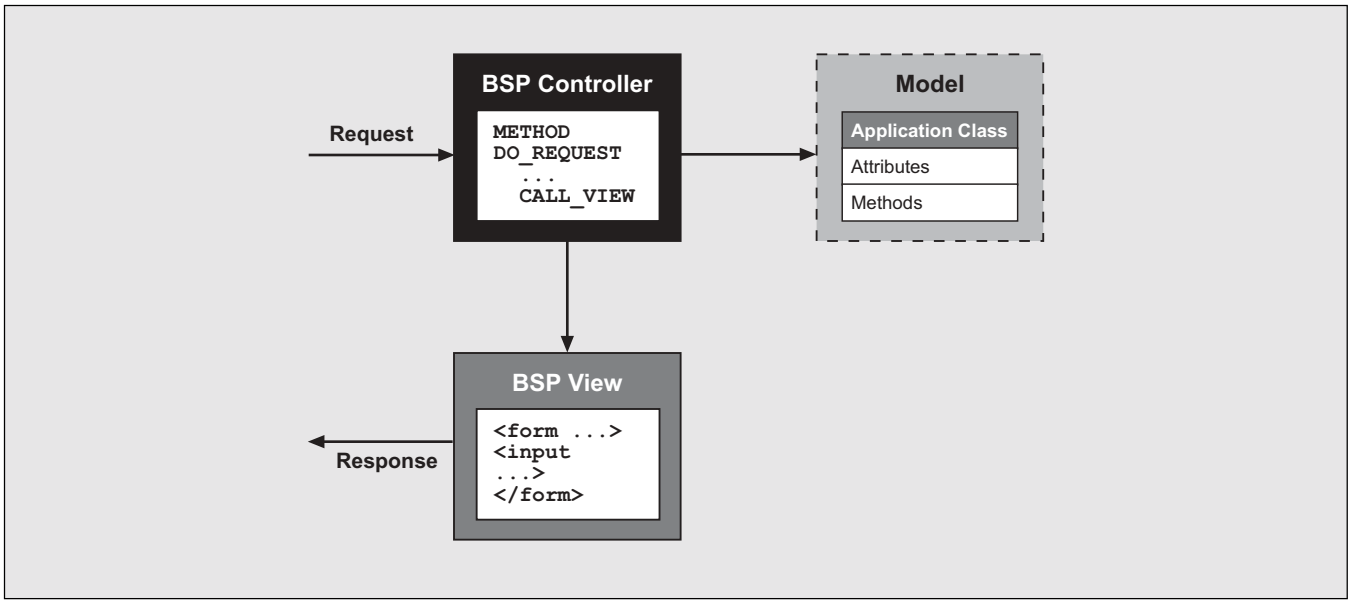
application in a way that maximizes code reusability and maintainability. The Model-View-Controller (MVC) model, which Java developers have successfully used for years, is now available with Web AS 6.20 and addresses this key issue.

With the amount of attention the MVC model has received in some circles over the last few years, you would think it was some hot new technology. It's not. The MVC model is simply an approach to structuring applications that separates the user interface into three distinct parts (see **Figure 9**):

- The *model* encapsulates the actual business data and business functionality. It serves as the data source for any kind of visualization. The model provides a single point for updating or retrieving data. In the flight example discussed in this article, business objects such as flights or bookings constitute the data for the model.
- The *view* visualizes the application data using a graphical representation. The same application data can be displayed in different ways (e.g., a table can be represented using a grid view or a chart view). When the application data (i.e., the model) changes, all dependent views typically need to be updated. In the flight example here, the three-column list of flights we coded in Listing 3 was a single view of the flight data.

> ✓ ***Note!***
>
> *Web AS 6.10 helped developers encapsulate reusable portions of code or HTML through the use of server-side "includes" — separate BSP page fragments that could be included in other pages at design time. For example, many developers create header and footer pages to house banner images and text that is shared across all pages within a web application. These header and footer pages are then inserted into each page within the BSP application through a single "include" directive. As you'll see, the MVC model provides a much more comprehensive approach to encapsulation than simple "include" files.*

Figure 10 Using the MVC Model When Building BSP Pages



- The *controller* manages interactions between the end user on one side, and the model and view on the other. The controller is responsible for handling user events, for updating the application data, and for handling navigation requests. In the flight example here, the code that queries the flight database, books a flight, or navigates from an overview to a detailed view would be the controller.

The MVC model helps to simplify the construction of complex user interfaces, limits the potential impact any changes might have, and raises the degree of reusability and maintainability of most BSP applications.

> ✓ ***Example***
>
> *Imagine that one of your users asks to see a current table of data sorted in a new way. Using the MVC model, you only need to add a new view, and perhaps a line or two of code, to the controller. The model, a notably sensitive component of the application, would (in most cases) need no modification at all.*

Building BSP Pages Using the MVC Model

So how does support for the MVC model change the way you build BSP pages? The most significant change, as you can see in the diagram in **Figure 10**, is that traditional BSP pages, originally containing HTML layout and event handlers, are divided conceptually into two parts: a *BSP controller* and a *BSP view*.

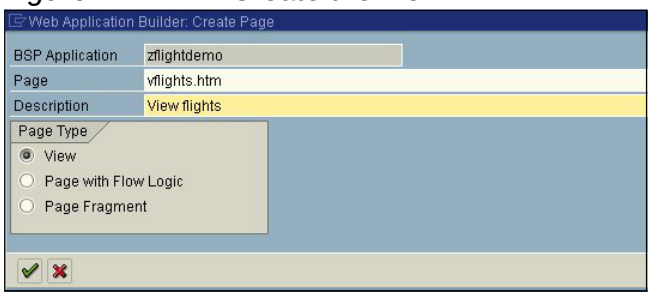
The controller (a BSP page) is responsible for handling all incoming navigation requests or data submissions. It communicates with the underlying model, which is implemented as an ABAP class. After interacting with the model to retrieve or post data as needed, the controller invokes the appropriate view, passing any data needed by the view to render the final HTML page.

To demonstrate, let's modify the flight demo application using the MVC model. Follow these steps:

1. Display the outline of the flight demo application in transaction *SE80*, as you normally would.

2. Position the cursor on the BSP application (*zflightdemo* in the example), right-click on it to open the context menu, and choose *Create* → *Page*. On the resulting pop-up, enter a name for the page (*vflights.htm*), add a short description (*View flights*), and select the page type (*View*), as shown in **Figure 11**. Press the *Enter* key (or click ✔).

Figure 11 Create the View



3. The system opens the page editor for the view (**Figure 12**), where you can choose to specify the layout in plain HTML, use a BSP extension (like HTMLB, for example), or use your own custom extension.

Figure 12 The Layout of the View

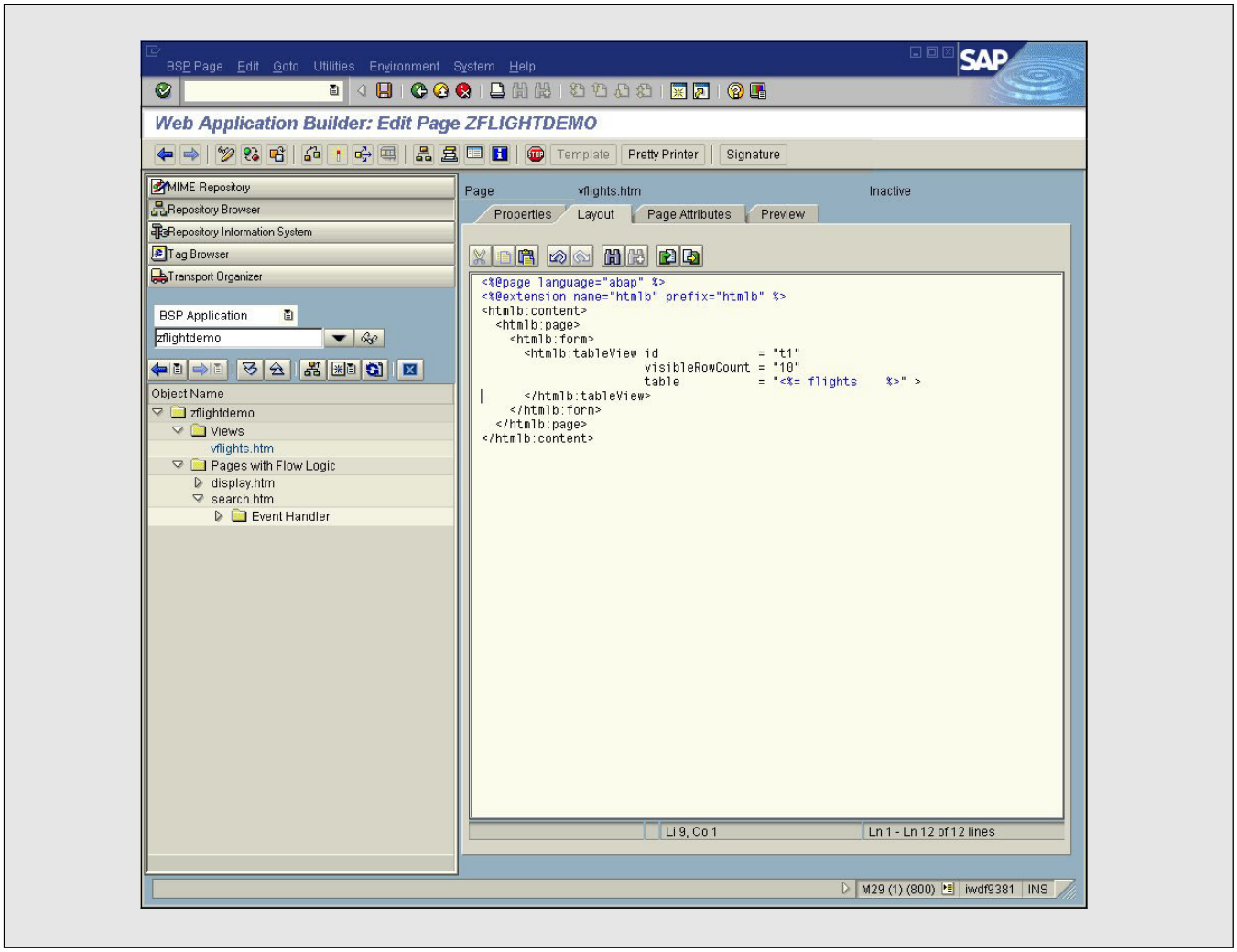
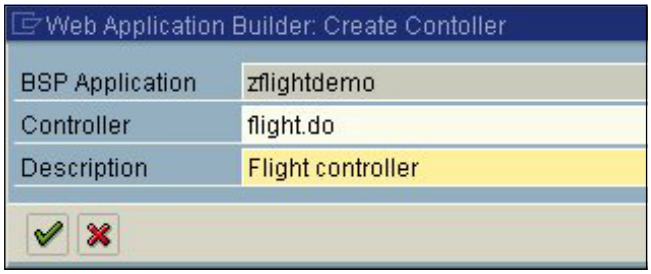


Figure 13 Create the Controller



Here, we will use HTMLB, so copy the code from Listing 2 into the page editor. (Note that the *visibleRowCount* attribute discussed in Listing 3 is also included to restrict the display.)

4. Go to the *Page Attributes* tab and create a new attribute *flights* of type *FLIGHTTAB*, in essentially the same way we did in the section "Creating a Web Application Using BSP Extensions."

Figure 14 The Properties of the Created Controller

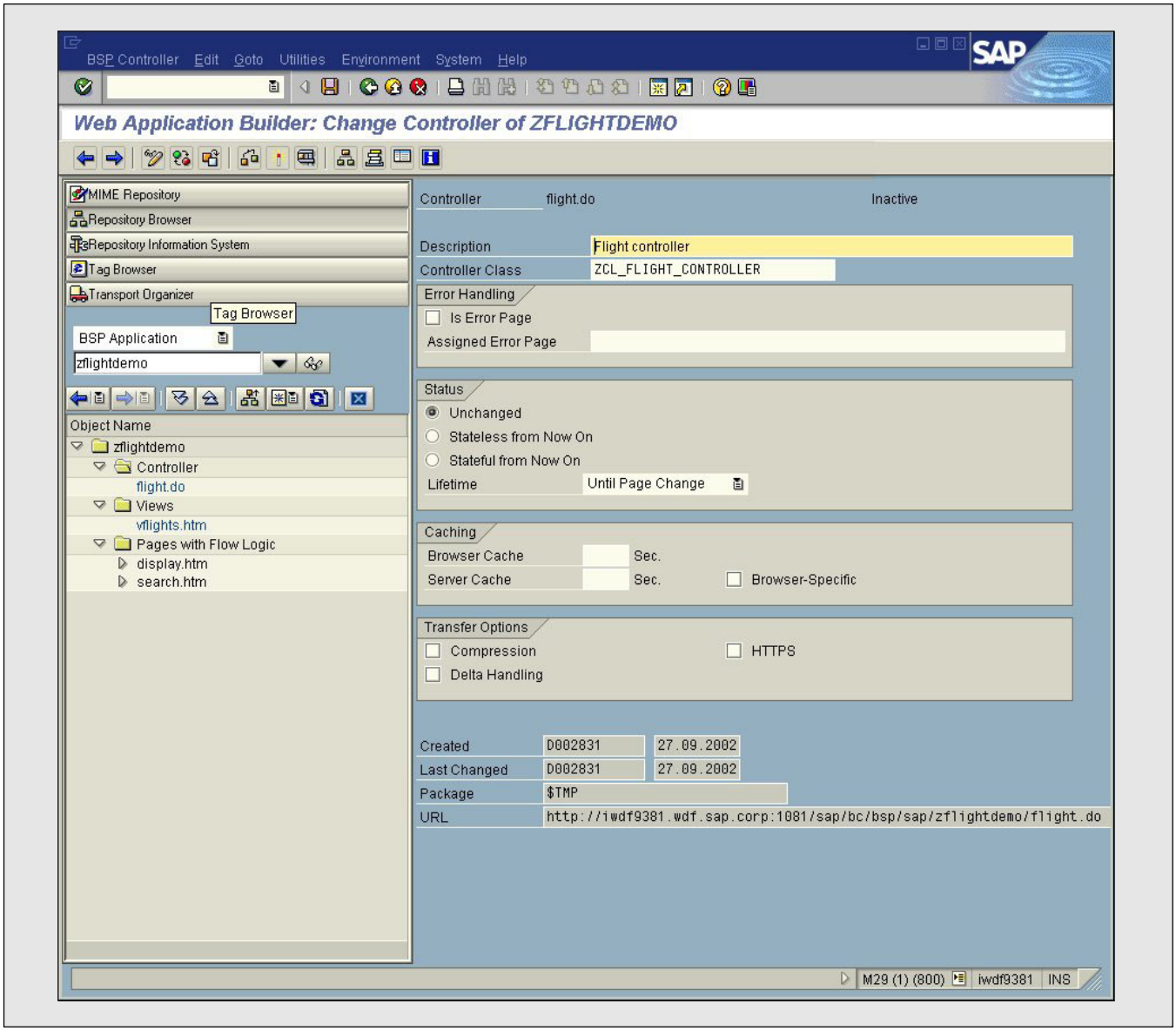
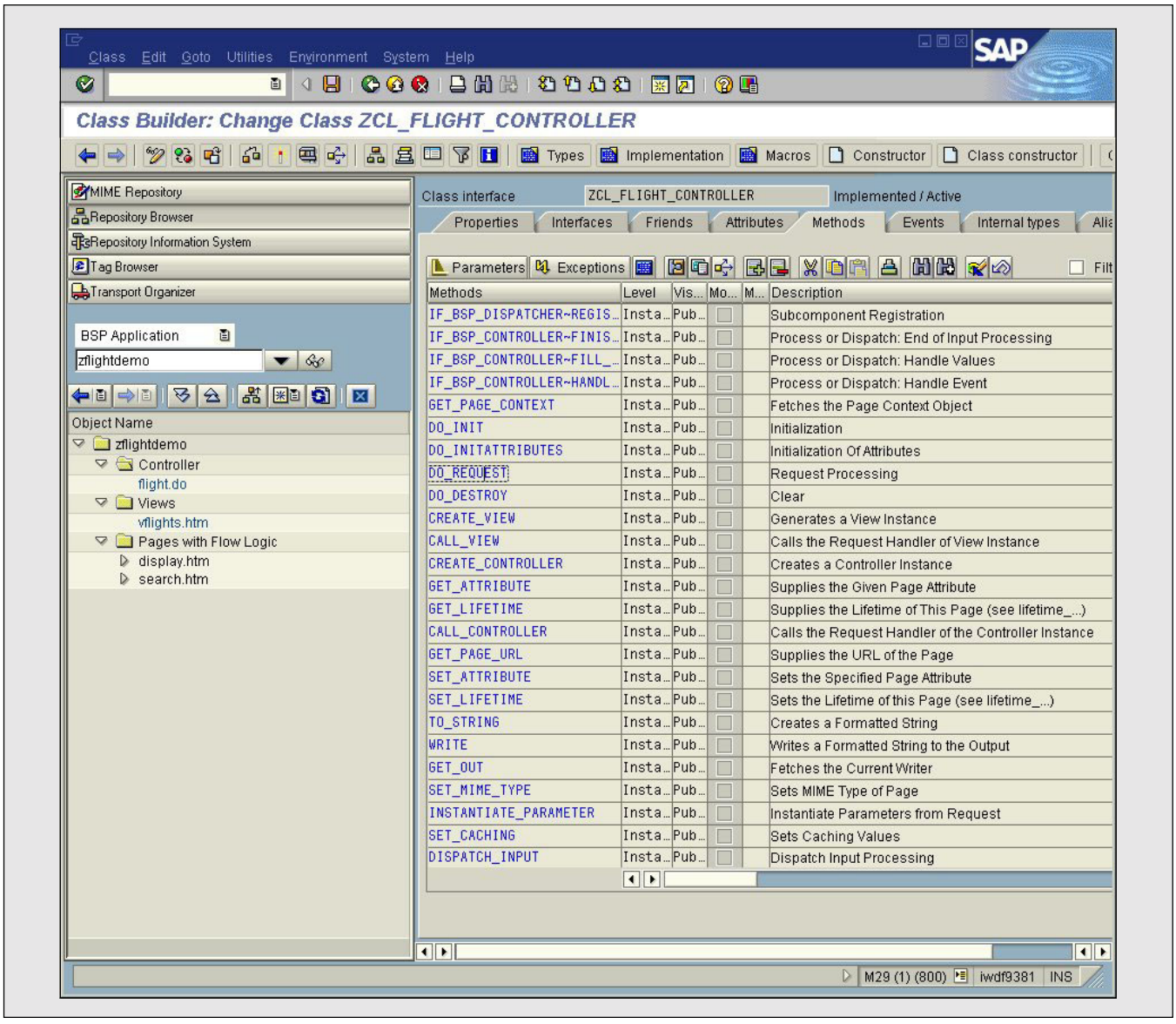


Figure 15 *The Inherited Controller Class Methods*



5. Position the cursor on the BSP application (*zflightdemo* in the example), right-click on it to open the context menu, and choose *Create* → *Controller*. On the resulting pop-up, enter a name for the controller (*flight.do*) and a short description (*Flight controller*), as shown in **Figure 13**. (By default, the controller's file name extension is *.do*.) Press *Enter* (or click ✔).

6. The system displays the properties page of the newly created controller (**Figure 14**). This page defines the attributes of the controller, and also the controller class, which implements the controller's behavior. To create the controller class, enter *ZCL_FLIGHT_CONTROLLER* in the *Controller Class* field and save (💾).

7. Double-click on the class name. The system asks whether you want to create this new class. Choose *Yes*. The system navigates to the Class Builder and displays the inherited methods (**Figure 15**). All controller classes are subclasses

Listing 6: Specify the Controller Actions

```
1 method DO_REQUEST.
2   data : view1 type ref to if_bsp_page,
3          flights type flighttab.
4          select * from sflight
5            into corresponding fields of table flights up to 100 rows.
6          view1 = create_view( view_name = 'vflights.htm').
7          view1->set_attribute( name = 'flights' value = flights ).
8          call_view( view1 ).
9 endmethod.
```

of the system class *CL_BSP_CONTROLLER2*, and thus they inherit the system class' methods.

8. Switch to change mode ( ) and, as shown in Figure 15, position the cursor on the method *DO_REQUEST*, which is the central point of the controller, where all the event handling takes place. Since a controller has its own URL, it can be requested directly from a web browser. The *do_request* method's implementation is then called immediately.

9. Choose the redefine button ( ). The system opens the source editor, where you can specify the actions to be executed when the controller is requested. Type in the code shown in **Listing 6**.

10. Save ( ) and activate ( ) your work, and then position the cursor on the controller and press *F8*. The system automatically starts the web browser and displays the list of flights as before (refer back to Figure 3), only now you have much more flexibility to quickly and easily add new views.

The controller's code needs some explanation. In line 2 of Listing 6, the variable *view1* is declared to store a reference to the view object instantiated in line 6. The internal table *flights* (line 3) is filled with records from table *sflight* in line 4. The controller then sets the internal table as a view attribute (line 7) and finally calls the view in line 8.

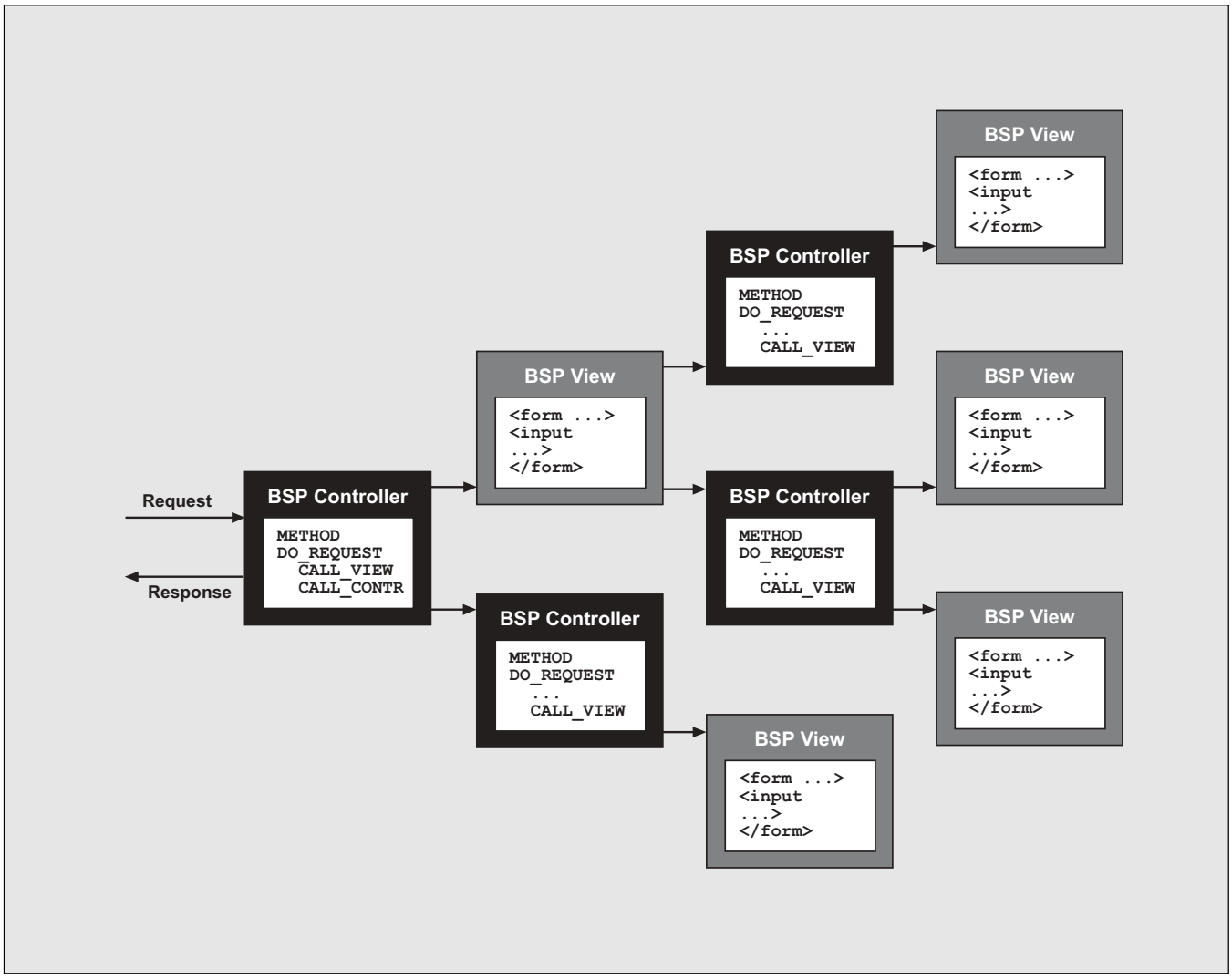
I hope it has become clear from the example shown here how the MVC model offers more flexibility than the plain BSP programming model: the view is cleanly separated from the request-handling code. In contrast to plain BSP pages, which both retrieve data and generate the final HTML page, BSPs based on the MVC model delegate this responsibility to a central controller class that can dynamically choose from one of many views as needed. In this way, new views can be added quickly without having to entirely rewrite the data retrieval logic.

Another benefit is efficiency. In the 6.10 BSP runtime environment, navigation between pages is handled by redirect requests. If a new page must be displayed, a redirect request is sent to the browser, which causes additional network traffic. Even if the application is stateless, all context information is lost between the two pages. With the MVC model, the controller constructs the view and then sends the view's HTML output back to the browser, allowing the context to remain valid between page requests.

Helpful Hints for Using the MVC Model

The MVC model has many advantages over the traditional BSP programming model. In particular:

Figure 16 More Complex MVC Scenarios



☑ BSP controllers provide a central entry point for all kinds of user requests and navigation handling. In addition, BSP controllers are callable units that can be reused in other applications more easily than in traditional BSP event handling.

☑ BSP views are passive objects — they serve to simply display data and variables by means of HTML or HTMLB. Since scripting code is not allowed in views, you obtain a much better separation of display and application logic (i.e., separation of concerns).

☑ BSP controllers and views can be cascaded to build "compartmentalized" web user interfaces instead of large unstructured HTML pages. BSP controllers can delegate tasks to subcontrollers that directly interact with a particular view; such a view might then in turn call another controller (see **Figure 16**).

☑ Rather than connecting each controller with the underlying application model, you can define individual models for each controller/view pair that holds the context for the fields in the view. A large application model can thereby be split into more manageable pieces.

Conclusion

BSP extensions and support for the MVC model, both now available with SAP Web Application Server 6.20, offer tremendous relief and opportunity for developers of Web AS applications. BSP extensions leverage pure HTML code and offer a lot of built-in functionality for handling tedious tasks like table formatting and scrolling. The MVC model enables developers to easily separate the user interface, data storage, and application flow portions of their web applications into separate components. BSP extensions and the MVC model are both well integrated into the ABAP Workbench environment, and can be used together or independently to increase the flexibility, maintainability, and reusability of your BSP applications.

Karl Kessler studied Computer Science at the Technical University of Munich, Germany. He joined SAP AG in 1992 as a member of the Basis modeling group, where he gained experience with SAP's Basis technology. In 1994, he joined the product management group of the ABAP Development Workbench. Since 1997, Karl has been Product Manager for SAP's business programming languages and frameworks. He can be reached at karl.kessler@sap.com.

