11 Creating your own BSP Extension Element

We already have seen in the last few chapters how powerful the BSP Extension Framework is. Fortunately, this is also an open technology framework that allows SAP's customers to build their own BSP extensions and to combine existing extensions to create composite elements.

11.1 Creating a BSP Extension Element

In Chapter 9, we took a close look at how BSP extension elements are structured in order to better understand how to use them. However, this only scratched the surface of what lies within the BSP extension element. Before we begin the process of writing our own elements, it is important to study in detail the most important part of an extension element: its element-handler class.

11.1.1 Extension Framework Hierarchy

The element-handler class actually represents an inherited hierarchy of class objects that all come together to form the extension framework. It is important to build this inheritance hierarchy correctly, because much of the functionality we will code within our handler will be placed inside of redefinitions of inherited methods.

The core extension framework comprises two objects, IF_BSP_ELEMENT and CL_BSP_ELEMENT. IF_BSP_ELEMENT defines all the core methods and attributes for the extension framework. CL_BSP_ELEMENT implements the IF_BSP_ELEMENT interface and provides the basic functions that support all BSP extension elements.

There are two more objects within this hierarchy, both specific to the individual extension element. The first is a generated basis class, usually created with the following naming standard:

(Z)CLG_<EXTENSION>_ELEMENT

This class is automatically generated by the BSP development environment. When you define attributes for your extension element inside the BSP extension editor, these attributes will be generated as public attributes of this basis class. This class should also inherit from CL_BSP_ELEMENT and provide the specific constructor for the element. By dynamically generating this class, all the attributes of your element can be strictly typed and checked at compile time.
The final object is the core handler class itself. It should inherit from the generated basis class, and its name is completely user definable. However it is probably good form to following a naming standard such as the following:

(Z)CL_<EXTENSION>_<_ELEMENT>

This class is where you will be spending most of your time as an element author. This class has the method redefinitions and any specific methods or attributes needed to implement the element.

11.1.2 User-Defined Validation

BSP extension elements are unique among the ABAP language tools in their approach to input validation. The extension runtime gives you the opportunity to code different validation routines that will be executed at runtime and compile time. That means that the syntax check of a BSP page will fire validation code that you can write. This gives you the ability to throw compiler errors for your own elements.

In order to implement user-defined validation, we must redefine two methods in our handler class that were inherited down from IF_BSP_ELEMENT.

The first method is COMPIL_TIME_IS_VALID. This is where we will code our compile time checks. SAP provides a series of validation methods (in class CL_BSP_ELEMENT_CT_ATTR_VALID), which assist in this process. In addition to providing simple checks, these methods also properly convert attribute-input string values into Boolean and integers values where necessary.

What follows are coding examples for the validation routines of a fictional BSP Extension Element. They contain common types of checks in order to demonstrate the different possible techniques.

METHOD if_bsp_element~compile_time_is_valid.
  validator->to_enum( name = 'Color' 
                     enums = 'RED/BLUE/GREEN' ).
  validator->to_enum( name  = 'alignment' 
                     enums = 'LEFT/RIGHT' ).
  validator->to_boolean( name = 'disabled' ).
  validator->to_integer( name = 'size' ).
  valid = validator->m_all_values_valid.
ENDMETHOD.

We also have the method RUNTIME_IS_VALID. This method is useful for checking attribute values that are supplied dynamically, such as through BSP expressions.
Creating a BSP Extension Element

(\%\%\=\%\%\) only at runtime, or for attributes whose values are transformed into another data type.

METHOD if_bsp_element-runtime_is_valid.
    get_class_named_parent(
        class_name = 'CL_HTMLB_CONTENT' ).
    IF runtime_parms = '/*/' OR runtime_parms CS 'alignment'.
        alignment = m_validator->to_enum(
            name = 'alignment'
            value = alignment
            enums = 'LEFT/RIGHT'
            required = space ).
    ENDIF.
    IF runtime_parms = '/*/' OR runtime_parms CS 'disabled'.
        disabled  = m_validator->bindable_to_boolean(
            name = 'disabled'
            value = disabled
            binding_path = _disabled
            page_context = m_page_context ).
    ENDIF.
    IF runtime_parms = '/*/' OR runtime_parms CS 'size'.
        size = m_validator->bindable_to_integer(
            name = 'size'
            value = size
            binding_path = _size
            page_context = m_page_context ).
    ENDIF.
ENDMETHOD.

The first line in the runtime validation method checks that this BSP element is used with an <htmlb:content> element. It is not possible to check this at compile time, as different elements can be used in different views, and these are compiled separately. We surround each of our dynamic value checks with an IF check for performance. That way we only perform validation routines on attributes that actually have values set dynamically.

It is important to note that these two validation methods will only be called if the User-Defined Validation option is selected in the BSP Element Properties. That way, if you have no validations that you wish to perform in your element, you can save the time that it would have taken for the framework to make calls into simply empty methods.
11.1.3 Element Content

Three methods in our element-handler class control the flow of creation of element content. They are `DO_AT_BEGINNING`, `DO_AT_ITERATION` and `DO_AT_END`.

`DO_AT_BEGINNING` is always accessed by the runtime at the beginning of the element processing. You can control the flow of processing after `DO_AT_BEGINNING` by setting the return parameter `RC`. If your processing is simple and only requires logic in the `DO_AT_BEGINNING`, you can set `RC` to `CO_ELEMENT_DONE`. Processing is then completed and returned to the BSP runtime. However setting `RC` to `CO_ELEMENT_CONTINUE` will allow processing to move on to the body of the element.

This means that all inner tags are given the change to render themselves. A small example might be that within a tabstrip there are many `<lib:tabStripItem>` elements. But, only one tabstrip item is required and must be rendered. Consequently, each item checks whether it is active and visible. Those that are not active are set `rc` to `CO_ELEMENT_DONE` to skip the processing of all inner elements, as this not needed for rendering. Only the one active tab strip item will actually continue with processing of its inner tags to generate the required HTML.

If the option Iteration Through Element Content was selected in the element properties screen, the method `DO_AT_ITERATION` can be called following `DO_AT_BEGINNING`. This method allows the element handler to make several passes over its inner content.

The method `DO_AT_END` is accessed after all other processing is completed. At this point, all the element content is available and can be further manipulated. This method is especially useful for BSP elements that contain inner elements.

The combination of `DO_AT_BEGINNING` and `DO_AT_END` methods are very similar to the structure of basic HTML. They are most useful for their ability to render before and after their inner content.

Let us assume the following example:

```html
<htmlb:link href="http://www.sap-press.com">
  SAP PRESS
</htmlb:link>
```

In this example, the `DO_AT_BEGINNING` method of the `<htmlb:link>` element will render out the HTML `<a href="http://www.sap-press.com">` and then set `CO_ELEMENT_CONTINUE`.

This causes the runtime to process the inner body, which in this case only outputs the string "SAP PRESS". Thereafter, the `DO_AT_END` method is called, which corresponds very much to the end tag in HTML. This method will render out the HTML...
sequence. Here the two methods very much reflect the way that HTML is structured with leading/trailing markup, allowing for efficient rendering.

11.2 Writing a Composite Element

Very often, we find the same pattern repeated on some or all of our BSP pages. Although such coding can be easily placed on all pages with cut-and-paste programming, it quickly becomes tedious and error-prone. Modifications suddenly require code updates over all BSP pages.

One approach to this problem, is to place the specific pattern into a page fragment and simply include it on every page where required. This has the advantage that changes are only required once in the page fragment. However, it still has the disadvantage that the code inside the page fragment is expanded inline into each BSP page. This increases the size of each page and can result in a GEN_BRANCHOFFSET_LIMIT error when generation limits are reached.

11.2.1 Designing a New Composite Element

What we most would like to have is a principle of composition. Usually these reusable patterns are just a collection of HTMLB elements. Would it not be nice if we could combine such a collection into one composite element? Well, we can, by creating our own BSP Extension Element.

Let us first look at an example application that could benefit from a redesign with composite elements in mind. This example shows the typical process of navigating back and forth inside a simple form. Normally, this can be done by using the <htmlb:button> element, with the new previous and next designs. Let us assume that we would like to place two navigation buttons at the bottom of each page.

```xml
<%@extension name="htmlb" prefix="htmlb"%>
<%@extension name="phtmlb" prefix="phtmlb"%>
<htmlb:content design="design2003">
  <htmlb:page>
    <!-- body comes here -->
  </htmlb:page>
</htmlb:content>
```

Figure 11.1 Composite Element Example
Creating your own BSP Extension Element

The goal is to replace this entire navigation rendering with one simple element. The expected final code on each BSP page would then be:

```html
%@extension name="htmlb" prefix="htmlb"%>
%@extension name="ybook" prefix="ybook"%>
<htmlb:content design="design2003">
<htmlb:page>
<htmlb:form>
...body comes here...
<ybook:pager prev = "Page In-1" next = "Page In+1" />
</htmlb:form>
</htmlb:page>
</htmlb:content>
```

We want one element that takes a previous and/or next attribute with the text to display. As we are slightly lazy in this example, we assume that pages are named exactly the same as the descriptive text, except that they are without spaces, and terminated with our typical `.bsp` extension.

The definition in the workbench, transaction SE80, of the new BSP element is quickly done. It has only two string attributes. Once this BSP element has been defined and activated, the above example BSP page will actually compile and run. It will just not yet render any output.
Now that we have defined our new `<ybook:pager>` element and already written the test program, it is time to complete the code for the composite element itself. Before processing other elements, it is important to understand how elements are processed on BSP pages. It is only possible to use existing BSP elements within our new element in this way.

A BSP element is written on a page using an XML format. As a first step, the BSP compiler must map the XML name onto a specific handler class. This class name can be seen in the workbench, when looking at the BSP element. The compiler generates code to create a new temporary variable to hold the reference to the handler class (data: statement), and then to create an instance of this BSP element-handler class. Next, the compiler generates the source code to initialize each attribute with its specified value. Finally, the BSP element is pushed onto a stack, which contains all elements that are currently in process, and the do_at_beginning method is called.

```plaintext
<xyz:element Ai = "Vi”>
  DATA %_e123 TYPE REF TO CL_XYZ_ELEMENT.
  CREATE OBJECT %_e123.
  %_e123->Ai = Vi.
  push( %_e123 ).
  %_e123->DO_AT_BEGINNING( ).
  DO.
  ...body of element ...
  %_e123->DO_AT_ITERATION( ).
  pop( %_e123 ).
</xyz:element>
```

Figure 11.2  BSP Element Properties and Attributes

**11.2.2 Processing Other BSP Elements**

Now that we have defined our new `<ybook:pager>` element and already written the test program, it is time to complete the code for the composite element itself. Before processing other elements, it is important to understand how elements are processed on BSP pages. It is only possible to use existing BSP elements within our new element in this way.

A BSP element is written on a page using an XML format. As a first step, the BSP compiler must map the XML name onto a specific handler class. This class name can be seen in the workbench, when looking at the BSP element. The compiler generates code to create a new temporary variable to hold the reference to the handler class (data: statement), and then to create an instance of this BSP element-handler class. Next, the compiler generates the source code to initialize each attribute with its specified value. Finally, the BSP element is pushed onto a stack, which contains all elements that are currently in process, and the do_at_beginning method is called.

```plaintext
<xyz:element Ai = “Vi”>
  DATA %_e123 TYPE REF TO CL_XYZ_ELEMENT.
  CREATE OBJECT %_e123.
  %_e123->Ai = Vi.
  push( %_e123 ).
  %_e123->DO_AT_BEGINNING( ).
  DO.
  ...body of element ...
  %_e123->DO_AT_ITERATION( ).
  pop( %_e123 ).
</xyz:element>
```

Figure 11.3  Element Processing Flow
In the simplest case, the element has no body or is defined as empty. In this case, the do_at_end method is called directly afterwards. If the element has a body, it is processed between the two method calls. It is also possible for the BSP element to request that the body be skipped, for example if you have an inactive tabstrip body. In the most extreme case, the BSP element can request that it reiterates over its body, which results in the body being processed as long as the do_iteration method requests that this be done.

In principle, it is difficult to know the exact code required to process each specific BSP element. It can also happen that, the BSP element is changed over time, and then has a different execution sequence.

You can break down the element-processing parts into the following phases.

▶ Each BSP element is first instantiated, and then its attributes are set correctly. This coding is very specific for the BSP element and will be different for each one.
▶ Thereafter, preamble coding is required to get the element onto the stack and process the do_at_beginning method call. This generic code is the same for each element.
▶ The body is processed. The body depends completely on the element being used. It can contain more BSP elements, or even raw HTML code can be rendered.
▶ More coding is required either to complete the processing of the BSP element, or to set it up correctly for a new iteration.

Specifically the fact that a BSP element can iterate over its body implies that some form of loop will be required. In addition, framework coding is required before and after the body to ensure correct processing. In order to encompass all these aspects, the following processing model was designed:

```
... user written factory and attribute initialization code ...
WHILE m_page_context->element_process( the_element )
    = CO_ELEMENT_CONTINUE.
    ... body of element ...
ENDWHILE.
```

This approach leaves the programmer the freedom to initialize the specific BSP element correctly. Thereafter, only one WHILE construct is required to process any BSP element in any of its variations. The element_process method will be called as many times as required to ensure that the BSP element is processed correctly.
Let us now look at a few detailed examples of processing existing elements. For our first example, let us assume that we have the following code on our BSP page:

```xml
<htmlb:button
text     = "Page In-1"
design   = "PREVIOUS"
onClick  = "pageIn-1.bsp" />
```

Then the correct code to process this `<htmlb:button>` dynamically would be:

```abap
DATA: myBtn TYPE REF TO CL_HTMLB_BUTTON.
CREATE OBJECT myBtn.
myBtn->text    = 'Page In-1'.
myBtn->design  = 'PREVIOUS'.
myBtn->onClick = 'pageIn-1.bsp'.
WHILE m_page_context->element_process( element = myBtn )
  = CO_ELEMENT_CONTINUE.
ENDWHILE.
```

The workbench must be used to find the correct class that implements this specific BSP element.

Alternatively, you can use the factory method that is automatically generated onto all BSP elements. The benefit of the factory method is that you can double-click on it to see the exact list of required parameters, and the ABAP language compiler is used to enforce required attributes.

```abap
DATA: myBtn TYPE REF TO CL_HTMLB_BUTTON.
myBtn = CL_HTMLB_BUTTON=>FACTORY( text    = 'Page In-1'
                                      design  = 'PREVIOUS'
                                      onClick = 'pageIn-1.bsp' ).
WHILE m_page_context->element_process( element = myBtn )
  = CO_ELEMENT_CONTINUE.
ENDWHILE.
```

Now let us look at a slightly more complex example. Assume that we are using an `<htmlb:link>` element that contains, as body, both an `<htmlb:image>` element and normal text. The source code on a BSP page would be:

```xml
<htmlb:link id = "lnk" reference = "http://www.sap.com" >
  <htmlb:image src = "logo.gif" />
</htmlb:link>
```
To process this sequence dynamically, the correct coding would be:

```plaintext
DATA: myLnk TYPE REF TO CL_HTMLB_LINK.
myLnk = CL_HTMLB_LINK=>FACTORY( id = 'lnk'
WHILE m_page_context->element_process( element = myLnk )
   = CO_ELEMENT_CONTINUE.
DATA: myImg TYPE REF TO CL_HTMLB_IMAGE.
myImg = CL_HTMLB_IMAGE=>FACTORY( src = 'logo.gif' ).
WHILE m_page_context->element_process( element = myImg )
   = CO_ELEMENT_CONTINUE.
ENDWHILE.
DATA: out TYPE REF TO IF_BSP_WRITER.
out = m_page_context->get_out().
out->print_string( 'SAP' ).
ENDWHILE.
```

The WHILE loop to process the `<htmlb:image>` is placed inside the WHILE loop of the `<htmlb:link>`. This reflects the fact that the image is part of the body of the link. In addition, text or raw HTML can be rendered as body of an element being processed. This is done by obtaining a reference to the active writer at the top of the stack and writing the relevant text.

Do not attempt to cache this writer reference. In all cases, always do the `get_out` call again after any `element_process` call. It is always possible for any new element on the stack to also push an additional writer onto the stack. The `get_out` call always returns the active writer.

In later support packages, there is a helper method called `print_string` that should be inherited from the super class `CL_BSP_ELEMENT`. This method already contains the logic to correctly retrieve the writer reference, allowing you to simplify your coding. In the example above, you could replace the text output with the following single line of code.

```plaintext
print_string( 'SAP' ).
```

Often, while writing the code to process a BSP element dynamically, you get weird error messages from the compiler. For example, the code snippet below produced the error "Field WHILE unknown."
Writing a Composite Element

The reason for this is very subtle. Inside BSP elements, strings are written using XML syntax with double quotes. Typically, code is cut-and-pasted from BSP pages directly into an ABAP class for the processing sequence. However, the double-quote character in ABAP starts a comment sequence that extends up to the end of the line. So, in the above source, the ABAP compiler will see `myImg->src = WHILE`.

The correct coding is:

```
* <htmlb:image src = "logo.gif" />
myImg->src = 'logo.gif'.
WHILE ...
```

11.2.3 Writing the Composite BSP Element

We already defined a test page that shows us the required rendering. Furthermore, we defined a new BSP element. As this will be an empty BSP element, we redefine only the `do_at_beginning` method and paste the code from the test page into this method. The code changes become straightforward, given the examples above.

Below is an extract of the code.

```abap
METHOD if_bsp_element~do_at_beginning.
  ... <phtmlb:horizontalDivider/> ... 
  * <phtmlb:matrix width = "100%" />
  DATA: phtmlb_matrix TYPE REF TO cl_phtmlb_matrix.
  phtmlb_matrix = cl_phtmlb_matrix=>factory( 
    width = '100%' ).
  WHILE m_page_context->element_process( 
    element = phtmlb_matrix ) = co_element_continue.
  * <phtmlb:matrixCell hAlign = "RIGHT" />
    phtmlb_matrix->mc_halign = 'RIGHT'.
    phtmlb_matrix->do_set_data( 
      element_name = 'matrixCell' ).
    ... prev button ...
  * space between two buttons
    me->PRINT_STRING( '&nbsp;' ).
    ... next button ...
```
Creating your own BSP Extension Element

11.2.4 Handling of Inner Data BSP Elements

Often we will find constructs where child BSP elements are used to feed information into the parent BSP element for later rendering. A typical example is `<htmlb:breadCrumb>`.

```plaintext
<htmlb:breadCrumb id = "myBreadCrumb0">
  <htmlb:breadCrumbItem key="k1" value="text1" />
  <htmlb:breadCrumbItem key="k2" value="text2" />
  <htmlb:breadCrumbItem key="k3" value="text3" />
</htmlb:breadCrumb>
```

Each item has only stub code for finding the parent and supplying the configured parameters:

```plaintext
METHOD if_bsp_element~do_at_beginning .
  DATA: breadcrumb  TYPE REF TO cl_htmlb_breadcrumb.
  breadcrumb ?= get_class_named_parent( 'CL_HTMLB_BREADCRUMB' ).
  breadcrumb->append_item( key = key , value = value ).
  rc = co_element_done.
ENDMETHOD.
```

However, for each item on the BSP page, code must be generated to instantiate a new `<htmlb:breadCrumbItem>`, set its attributes, and then to process the element. This is very high overhead for simply adding additional configuration information to the parent item. To improve the performance for this typical usage pattern, a new BSP element of type `Data` was created.

Effectively, the name of the parent handler class is specified for the new BSP element. The BSP library generator will then place all the attributes of the `Data` ele-
ment onto the parent element, using the camel-case abbreviation of the name as key to prefix the attributes. For example, for <phtmlb:matrixCell>, the camel-case abbreviation will be mc_. The <phtmlb:matrixCell> has at least two attributes: col and row. For these defined attributes, new attributes mc_col and mc_row are generated on the handler class of the parent.

When the <phtmlb:matrixCell> is used on a BSP page, the BSP compiler keeps a list of the surrounding BSP elements. It sees that <phtmlb:matrixCell> is a data element attached to the class cl_phtmlb_matrix. As a result, the following code is generated:

%_matrix_6->mc_col = 1.
%_matrix_6->mc_row = 2.
...
%_matrix_6->DO_SET_DATA( element_name = 'matrixCell' ).

The %_matrix_6 is the outer instance of type <phtmlb:matrix>. The attributes are set on the parent class, and the DO_SET_DATA call is placed, giving the name of the actual data element being processed. This way, data can be moved into the parent element with better performance.

### 11.3 A Deeper Look at BSP Extensions Events

#### 11.3.1 Introduction to BSP Extension Events

HTML/HTTP does not support the concept of server events. At the lowest level, the only building block that is available is forms in HTML, which can be submitted to a server. When a form is submitted, all input fields—including hidden input fields—are transported to the server. Therefore, event-handling in the browser is reduced to setting up specific predefined input fields, usually type="hidden", with values that reflect the event to be sent to the server, and then submitting the form.
When using the HTMLB family of rendering libraries, it is very seldom that any raw HTML is required. The rendering libraries already have sufficiently extensive sets of controls. However, once some HTML is required, you are immediately faced with a few perplexing problems. One is the question of transporting events from the browser to the server.

```html
<htmlb:form id="myform" >
  <input type="hidden" name="s_event_id" value ="TEST"/>
  <SCRIPT language="JavaScript">
    function myEventHandler(event_id) {
      document.myform.s_event_id.value = event_id;
      document.myform.submit();
    }
  </SCRIPT>
  <button id="Test" onclick="myEventHandler('button_clicked');">
    Submit!    
  </button><br>
  Event = <%= server_event_id %>
</htmlb:form>

Listing 11.1  Triggering a “Server Event” via the HTML form submit
```

The HTMLB library comes with its own event-handling system, which also includes a large piece of JavaScript code. If native HTML code, such as the code listing above, is added on a page that bypasses the HTMLB event system, the HTMLB library could be negatively affected.

One typical example is the `<xhtmlb:protectDoubleSubmit>` element. This item hooks into the HTMLB event system in the browser and will display a wait message once an event is sent to the server. Therefore, it is helpful for other library writers, and for people writing native HTML, to use the HTMLB event system for their event handling as well.

### 11.3.2 Rendering Events

During rendering, each element might require one or more events. This is usually done by wiring the HTML `onClick` attribute with some JavaScript code that will handle the event. This specific, required JavaScript code is obtained by a call to the method `cl_htmlb_manager=>render_event_call`.

This method will return a sequence of JavaScript code, which consists of one or more calls to the different JavaScript functions that are available for event handling in the browser. The output of this method is for internal use only. This output has been improved a number of times. Do not try to concatenate this Java-
Script output together directly, as this will cause problems if the underlying event-handling code is modified.

Rendering Phase:

```html
<htmlb:button id="myBtn" onClick="button_clicked"/>
...
CL_HTMLB_BUTTON
... event = CL_HTMLB_MANAGER=>RENDER_EVENT_CALL(...).
... render onclick="htmlbSubmitLib(...)"
```

In the above examples, the JavaScript function `htmlbSubmitLib` is shown. However, the exact call that will be generated depends on a number of factors, for example whether a client-side event is also involved, or whether the event is listed in a predefined dictionary. Consider the output of the `RENDER_EVENT_CALL` method as a black box.

In the browser, once a control event is triggered, the JavaScript code in the `onclick` handler is executed. This code calls the defined JavaScript code, which packs the relevant event information into hidden input fields and then submits the form:

In Browser:
1. User clicks on button
2. `onclick` is triggered, calls `htmlbSubmitLib(...)`
3. Sets up a number of input fields with correct values
4. Calls `form.submit();`

**RENDER_EVENT_CALL Method**

The `render_event_call` can only be used within a BSP element. One of the checks that this method does is to see if it is used within an HTMLB form. This is verified by checking the processing stack of all BSP elements, looking for an `<htmlb:form>` element. This method has a relatively complex interface that is discussed below in detail.

<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsp_element IF_BSP_ELEMENT</td>
<td>This is the actual element that is rendering the event. From this interface, the library name, the element name and the ID will be used for event-rendering. The first two values are generated into the base class of the element. The ID string must be set by the element.</td>
</tr>
</tbody>
</table>

Table 11.1 Parameters of Method RENDER_EVENT_CALL
<table>
<thead>
<tr>
<th>Parameter Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>event_type STRING</td>
<td>This string indicates what type of event was fired by the element. Typically, a button could fire a click event, a pager could fire page up or down events, and a table could fire a row-select or header-click event. This string has no further meaning for the HTMLB event system, and is transported transparently.</td>
</tr>
<tr>
<td>server_event STRING</td>
<td>This string is defined by the user of the element for the event. A typical example would be to write <code>&lt;htmlb:button onClick=&quot;myHandler&quot;/&gt;</code>. This string can contain information to help the user to handle the event correctly. This string has no further meaning for the HTMLB event system, and is transported transparently.</td>
</tr>
<tr>
<td>client_event STRING</td>
<td>This string reflects the typical <code>onClientClick</code> attribute used on many elements. It must contain valid JavaScript code that will be executed in the browser. This string is not returned to the server. At a minimum, one of the server or client events must be specified. Otherwise, no event-handling code will be generated.</td>
</tr>
<tr>
<td>client_event_inlined XFELD Default SPACE</td>
<td>Initially, it was up to the control to render out a JavaScript function that had a predefined name containing the <code>client_event</code> code. However, during HTML-parsing, small JavaScript functions cause a high overhead for the HTML-rendering. Thus, the inline flag leaves the rendering of the JavaScript code to the HTMLB manager class. It only creates a JavaScript function if this event should actually be triggered. We highly recommend that you set the value always to &quot;X&quot;.</td>
</tr>
<tr>
<td>event_defined STRING</td>
<td>Many events require a minimal string to contain additional information for the event. Instead of using additional parameters, you can use this one string for carrying the information. This string has no further meaning for the HTMLB event system, and is transported transparently.</td>
</tr>
<tr>
<td>param_count I</td>
<td>Number of parameters that will be transported in this event. This value must be set correctly for the <code>render_event_call</code> method.</td>
</tr>
<tr>
<td>param_string STRING</td>
<td>A comma-separated string of parameters (strangely starting with a comma!). This list of parameters is copied verbatim into the generated event handling function. It is also possible to embed the names of JavaScript variables in the event-parameter string with this format, which is then automatically used during the event-handling.</td>
</tr>
<tr>
<td>param_1 ... _9 STRING</td>
<td>An alternative option is to specify the <code>param_string</code> string as single parameters from <code>param_1</code> to <code>param_9</code>. The parameters are copied together during the rendering of the event. If the parameters are supplied individually, each parameter is considered to be a constant string, and will be rendered with quotes.</td>
</tr>
</tbody>
</table>

| Table 11 | Parameters of Method RENDER_EVENT_CALL (cont.) |
11.3.3 Handling Incoming Events

On the server, the event-handling system will look at the incoming HTTP request. If it detects form fields with well-known names, for example all HTMLB element-event input fields having a prefix `htmlbevt_`, it will signal an HTMLB event. The runtime then unpacks the relevant fields into an event object.

On Server:

```plaintext
event = CL_HTMLB_MANAGER=>GET_EVENT_EX( request )
```

This action of unpacking the relevant fields into an event object is done by the class `cl_htmlb_manager`. It will map the event onto the correct class, which is by default the same class used for rendering the BSP element. It instantiates a new copy of this class and then does a query for the `if_htmlb_data` interface.

The method `event_initialized` will be called with all the standard attributes of an HTMLB event. The values are restored onto the event attributes defined on the interface `if_htmlb_data`. The last call will be to `event_set_parameters` with all additional parameters that were available in the incoming HTTP request. These are also restored into the class attributes.

11.3.4 Rendering an Event via the `<bsp:htmlbEvent>` Element

It is useful to understand the way to directly interact with the `CL_HTMLB_MANAGER=>RENDER_EVENT_CALL` method if you are going to create your own custom BSP elements. This method call can be included in the rendering code of your ABAP class.
What if you simply want to render an event in-line in your BSP page and attach it to some standard HTML or another BSP Element? For this task, SAP provides the `<bsp:htmlbEvent>` element. This element can either return the event JavaScript code for later use, or it can generate a JavaScript function that, when called, will fire an event back to the server.

For example, the control can be used as:
```html
<bsp:htmlbEvent name="fireMyEvent" p1="a" p2="b" />
```

It will write into the output stream the following:
```html
<script> function fireMyEvent(a,b) {...} </script>
```

This function can now be called directly from HTML or JavaScript:
```html
<button onclick="return fireMyEvent('myButton',123)">myButton</button>
```

With this design, it is actually possible to use the HTMLB event system, without even knowing what is rendered out. The `<bsp:htmlbEvent>` element renders out a wrapper function that can be called directly, and it even allows additional parameters to be transported.

Another approach is to request that the `<bsp:htmlbEvent>` element return the JavaScript code for direct use. By flagging an attribute on the element as a reference attribute, it will get a reference to a local variable, and then can write back the information. In the example below, `event_code` will be updated by the `<bsp:htmlbEvent>` element with the final generated JavaScript code, and the code can now be used directly inline when writing HTML.
```html
<% DATA: event_code TYPE string. %>
<bsp:htmlbEvent event_defined="myBtn2" event_code="<%=event_code%>" />
<button onclick = "<%=event_code%>">myButton2</button>
```

### 11.4 Event Handling in Composite Elements

Earlier in this chapter, we built a composite element, but you may have noticed that the example did not fire any events and was not tied into the HTMLB event manager. Now that we have studied the HTMLB event manager in detail, we are ready to return the earlier example and improve it by changing the fired `<htmlb:button>` events into real native events from this element. In addition, we will add support for a data interface.
11.4.1 Extending the Design of the Composite Element

As the names of all IDs and events used in the previous example were hard coded, it was not possible to use two pagers on the same HTML page. For example, this could be interesting in scenarios where a split screen showing two logical independent sequences is used, and can be paged separately. Thus, we need to begin our enhancements by adding an ID attribute.

In addition, one never knew what the current page was. The pager only handled the previous and next pages. We will also add a current attribute, which is the name of the current page. This will also be rendered left-aligned on screen.

Last, we are adding an onPage attribute to allow us to configure the event handler that must be called on return. Note that we will have both pagePrevious and pageNext events. The onPage is just a string that is the user’s handle for the event. Although in most elements we define an onX per event, it is not required. Using one such onX string for a number of events is perfectly acceptable.

![BSP Element Properties and Attributes](image)

As Figure 11.5 demonstrates, we have created a new element for these enhancements so as to keep the older example for reference. But it is also possible simply to change the original code.

11.4.2 Using the Composite Element

Before we start looking under the hood at the code that will be needed to complete the work, let us first use the new element. This will give us a good idea of what must be supported. The test program will be similar to that used previously. We only have to set additional attributes for the element.
For each page, we define the following source code:

```html
<htmlb:content design="design2003"><htmlb:page><htmlb:form>
  ...body comes here...
  <ybook:pager2 id = "<any id string>"
    prev    = "<name of previous page>"
    current = "<name of this page>"
    next    = "<name of next page>"
    onPage  = "<name of event handler>" />
</htmlb:form></htmlb:page></htmlb:content>
```

For the `onInputProcessing` code, we would now like to use code that is similar to that of the HTMLB library:

```abap
DATA: event TYPE REF TO if_htmlb_data.
  event = cl_htmlb_manager=>get_event_ex( request ).
  IF event IS NOT INITIAL AND event->event_id = 'myPager'.
    navigation->goto_page( event->event_defined ).
  ENDIF.
```

In addition, the element should support minimal data retrieval, where it is possible to query the previous, current, and next pages. The typical code for the data call is:

```abap
DATA: pager TYPE REF TO ycl_bsp_book_extension_pager2.
  pager ?= cl_htmlb_manager=>get_data( request = request
       name    = 'ybook:pager2'
       id      = 'myPager' ).
  * use here pager->current, pager->next, pager->prev
```

Notice that for the `get_data` call it is important to also supply the library and element name. The HTMLB manager has no other help available to determine the correct handler class. The library name is not that of the prefix used in the layout, but the original name under which the library was created. This allows the HTMLB manager to again determine the correct handling class.

We see from the above coding that we wish to achieve a new pager element that will work transparently with the HTMLB manager. Any consumer of our new element should not be able to see a difference between it and any other standard SAP-delivered element.

### 11.4.3 Use of IDs

The first significant aspect is the handling of the element ID. Once we allow the option that the same element can be used multiple times on the same page, each must have a unique ID. First, the element was given a new required ID attribute.
The pager element itself does not really do any rendering. Primarily, it uses two `<htmlb:button>` elements. Each of these buttons requires an ID. This at first was solved by just hard coding the ID string.

```vbnet
htmlb_button = cl_htmlb_button=>factory(
    id = 'ybook_pager_next'
... )
```

With the new approach, we would like to have IDs that are unique and independent of the usage count. This goal can be achieved by using the ID of the element as the basis for creating new IDs. All new IDs will typically be of the form `<id>_<sub string>`. This is such a common pattern when building composite elements that the factory methods were extended to handle the concept of an `id`, plus a postfix string that must be attached.

```vbnet
htmlb_button = cl_htmlb_button=>factory(
    id = id
    id_postfix = '__Previous'
... ).

htmlb_button = cl_htmlb_button=>factory(
    id = id
    id_postfix = '__Next'
... ).
```

The factory method will concatenate the `id` and `id_postfix` strings together to create the new ID for the specific button.

### 11.4.4 Integrating into the HTMLB Manager

The HTMLB manager interacts with the element-handler class via the `if_htmlb_data` interface. The interface has four methods used for the data and event handling.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESTORE_FROM_REQUEST</td>
<td>This method is called by the HTMLB manager to restore view state from the incoming request. This is always triggered by the <code>get_data</code> call.</td>
</tr>
<tr>
<td>EVENT_INITIALIZE</td>
<td>The <code>get_event_ex</code> call will result in a call to this interface, with the event data already decoded. The code has to fill the <code>event_*</code> attributes of the <code>if_htmlb_data</code> interface.</td>
</tr>
</tbody>
</table>

Table 11.2 Methods of Interface IF_HTMLB_DATA
When implementing these methods, the biggest problem is the interaction between data and events. For example, assume that we have an `<htmlb:group>` element and that the minimize button was pressed. Using only the `get_data` call, the view state would actually indicate that the group container is still maximized. After restoring the previous view state, you therefore must check whether the incoming event must be applied onto the data. Similarly, if only the `get_event_ex` call is used, it is usually practical that the rest of the view state data is also restored, so that no additional `get_data` call is required.

For this reason, we always implement the `event_initialize` code to also call the `restore_from_request` method, thereby simulating a `get_data` call. The `restore_from_request` code uses the `event_id` as a flag to determine whether it is called from the event-handling code, in which case it continues to restore data, or whether it is triggered from a `get_data` call, in which case it will use an HTMLB manager call to apply an event if required.

**METHOD** `if_htmlb_data-event_initialize`.
* Initialize event_* parameters
  
  `me->if_htmlb_data-event_* = ...`
* Restore all data from the request
  
  `me->if_htmlb_data-restore_from_request(
    request = p_request
    id      = if_htmlb_data-event_id ).`
  
  ...now apply event onto restored data...

ENDMETHOD.

**METHOD** `if_htmlb_data-restore_from_request`.
* Use event_id as flag to check whether we also have an * event. Let it do work.

  `IF me->if_htmlb_data-event_id IS INITIAL AND`
  
  `CL_HTMLB_MANAGER=>CHECK_AND_INITIALISE_EVENT(
    instance  = me
    request   = request
  )`
249

11.4.5 Data-Handling

We require the pager to be able to restore the values of the previous, current, and next pages. We must keep in mind that any control on the page can trigger an event to the server, and thus it is not always possible to retrieve this information from the event data.

The best technique for storing the view state within an HTML page is to use hidden input fields. This information is not rendered and will be returned to the server when the form is submitted.

The following code is used within the do_at_beginning method to render the view state into the response, so that it will be returned to the server on the next request:

```plaintext
DATA: html TYPE STRING.
CONCATENATE`<input type="hidden" name="` id `__valPrev" value="` prev `">
`<input type="hidden" name="` id `__valCurrent" value="` current `">
`<input type="hidden" name="` id `__valNext" value="` next `">
INTO html.
print_string( html ).
```

Notice the use of the ID with sub strings to create new names for each hidden input field. The values are taken from the current element attributes.

To restore the values, the code below is used in the restore_from_request method:

```plaintext
me->id = id.
CONCATENATE me->id `_valPrev` INTO name.
me->prev = request->get_form_field( name ).
```
Creating your own BSP Extension Element

---

**11.4.6 Event-Handling**

Event-handling is slightly more complex. The pager element uses two `<htmlb:button>` elements. As such, when one of these buttons is pressed, a HTMLB button-clicked event is returned to the server. What we actually want is to present a pager event.

The problem is that the HTMLB manager has the class `cl_htmlb_button` defined as handler class for the button-click. We want our new pager class defined as the handler for these events. For this, the HTMLB manager supports an escape mechanism. Usually events are encoded in the HTML in the form:

```
htmlb:button:click:null
```

But, it is also possible to add an additional handler classes onto this string, using `::` as separator sequences.

```
htmlb:button:click:null::<handler_class>::<event_defined>
```

This means that even although a button-click event is received, the newly specified handler class must be called to decode the event. As it is not possible to configure these escape strings when processing another element, the HTMLB manager will also accept these escape sequences when they are attached to the event-server name, on `X` strings.

```
DATA: htmlb_button TYPE REF TO cl_htmlb_button.
htmlb_button = cl_htmlb_button=>factory(
  id            = id
  id_postfix    = '__pagePrevious'
  text          = prev
  design        = 'PREVIOUS'
).
CONCATENATE onPage '::YCL_BSP_BOOK_EXTENSION_PAGER2::'
  prev '.bsp'
  INTO htmlb_button->onclick.
WHILE m_page_context->element_process( htmlb_button )
  = co_element_continue.
ENDWHILE.
```
In the previous example we have hard-coded the class name. This approach is simple but can lead to problems if you rename your handler class. If you want to use the element handler as the event handler, it is best to retrieve the class name dynamically using the CLG (base) class.

```plaintext
CONCATENATE onPage '::' me->m_class_name '::'
    prev '.bsp'
INTO htmlb_button->onclick.
```

Instead of just writing `onClick = onPage`, we are now adding our `YCL_BSP_BOOK_EXTENSION_PAGER2` class into the escape string to function as the handler for this specific button-click event. We use our event-defined string to carry the name of the previous page. With this small change, our handler class will always be called when one of the buttons is pressed.

For an incoming event, the `event_initialize` method will be called with the information about the button click. First, we set up all the `event_*` attributes. Afterwards, we would like to map a button-click event onto a pager `pagePrevious` or `pageNext` event.

Our first step is to set the new event name to the name of this BSP element (pager2). As a next step, the ID has to be set correctly. Remember that the initial ID was post-fixed with a constant string `__<direction>`. Therefore, we split the string at `__` to get the original ID again and the event type, which was effectively encoded as a sub string in the ID.

```plaintext
METHOD if_htmlb_data~event_initialize.

* Copy those parameters which we keep verbatim
  if_htmlb_data~event_id = p_event_id.
  if_htmlb_data~event_type = p_event_type.
  if_htmlb_data~event_class = p_event_class.
  if_htmlb_data~event_name = p_event_name.
  if_htmlb_data~event_server_name = p_event_server_name.
  if_htmlb_data~event_defined = p_event_defined.
  if_htmlb_data~event_intercept_depth = p_event_intercept_depth.

* The pager uses two `<htmlb:button>` elements. Massage the
  event to be pager event.
* Event name will be 'button', should be our 'pager2'.
* Event Id will be <id>__pageNext or <id>__pagePrevious
* Event Type will be click from the button. The actual value
```
we want was already encoded into the ID before.

```c
if_htmlb_data-event_name = me->m_name.
SPLIT if_htmlb_data-event_id AT '__'
    INTO if_htmlb_data-event_id
        if_htmlb_data-event_type.

* Restore view state from the request
  if_htmlb_data-restore_from_request( request = p_request
    id      = if_htmlb_data-event_id ).
ENDMETHOD.
```

With the above changes, events are now presented as pager2 events, as shown in Figure 11.6

---

**Figure 11.6** Intercepted Events

---
Index

<bee:html> 214
<bsp:bee> 205, 290
<bsp:call> 283
<bsp:findAndReplace> 268, 295
<bsp:htmlEvent> 243, 269
<bsp:parameter> 432
<bsp:portalEvent> 271
<bsp:portalNavigationAbsolute> 272
<bsp:portalNavigationRelative> 272, 273
<bsp:portalNavigationToObject> 272, 273
<bsp:root> 214
<btf:editor> 257, 343
<graphics:chart> 261, 262, 266
<graphics:custom> 266
<graphics:data> 262, 264, 266
<graphics:nativexml> 262
<htmlb:breadCrumb> 238
<htmlb:button> 202, 231, 244
<htmlb:content> 174, 395, 408
<htmlb:document> 178
<htmlb:documentBody> 178
<htmlb:documentHead> 178
<htmlb:dropDownListBox> 175, 223, 322, 338
<htmlb:fileUpload> 180
<htmlb:form> 179, 197, 241, 360
<htmlb:gridLayout> 194, 200
<htmlb:gridLayoutCell> 195
<htmlb:group> 248
<htmlb:headInclude> 178
<htmlb:image> 206
<htmlb:inputField> 206, 322, 323, 351, 353
<htmlb:inputField> as BEE 208
<htmlb:label> 308, 312, 338, 351
<htmlb:listBox> 323
<htmlb:page> 177
<htmlb:tableView> 187, 205, 206, 215, 322, 422, 443
<htmlb:tableViewColumn> 189
<htmlb:tableViewColumns> 189
<htmlb:textView> 223
<htmlb:tray> 337, 434, 461, 468
<htmlb:tree> 192
<htmlb:treeNode> 192
<phtmlb:comboBox> 417, 419
<phtmlb:containerContentItem> 199
<phtmlb:containerTabStrip> 197
<phtmlb:containerTabStripItem> 199
<phtmlb:formattedText> 455
<phtmlb:formLayout> 200, 298
<phtmlb:formLayoutCheckBox> 202
<phtmlb:formLayoutDropDownListBox> 202
<phtmlb:formLayoutInputField> 202
<phtmlb:formLayoutItem> 202
<phtmlb:formLayoutTextEdit> 202
<phtmlb:formLayoutX> 202
<phtmlb:matrix> 194, 200, 290, 463
<phtmlb:matrixCell> 195, 239
<phtmlb:popupMenu> 198, 323, 331, 374
<phtmlb:buttonGroup> 180, 182, 185
<phtmlb:pager> 191
<phtmlb:protectDoubleSubmit> 196, 240
<phtmlb:toolBar> 326
<phtmlb:toolBarButton> 327
<1x1 URL> 127

A
ABAP Class Builder 278
ABAP Kernel 15
ABAP Look-and-Feel Service, ALFS 398
ABAP Workbench 15, 25
ABAP XML API 453
Accept, HTTP request header 35
Accept-Encoding, HTTP request header 36
Accept-Language, HTTP request header 36
accessibility flag 473
ActiveX 358, 449
Adobe Forms 423
ALFS 398
integration to Web AS 402
source code 404
alias handling 74
alias texts 347
ALV Grid 187, 367, 422, 424
ALV Grid Field Catalog 217
anonymous service 108
applets 133
authentication 97, 137
anonymous service 108
basic 98, 113
form-based 109
logon application 111
SSO 114
Authorization, HTTP request header 36

B
Base64 100, 477
basic authentication 98
cancelled 100
de-authentication 113
BEE 205, 253
creating own 227
element content 230
error handling 215
for Excel download 374
help function 311
HTML 209
table 211
user-defined validation 228
XML 212
Big Endian 342
binary string 370, 474
breakpoint 29
browser cache, expiration 476
browser rendering time 157
BSP Application Event 144
BSP element
  composite 231
  inner data 238
  processing 233
  processing flow 233
  writing a composite 237
BSP Element Expressions 205
BSP Extension Element
  help function 311
BSP Extension Framework 227
BSP extensions 16, 167, 253
designs 172

element 186
extension framework 168
library 268
technology 168
using 168
BSP extensions event 239
BSP library 268
BSPs, interaction with SAP GUI 144
BTF 253, 343
BTF content 258
BTF document 258
BTF editor 253
BSP extension element 256
database storage 255
functionality 254
toolbar 254
Unicode 255
Business Communication Service (BCS) 481
Business Server Pages (BSP) 15
Business Text Framework (BTF) 253
Byte Order Mark (BOM) 342, 369, 371

C
category, IGS 263
certificate
de-authentication 114
digital 105
chart data 262
chart designer 267
chart model class 264
checkbox 48
CL_BCS 481
CL_BSP_APPLICATION 479
CL_BSP_ELEMENT 227
CL_BSP_PAGE 480
CL_BSP_RUNTIME 474
CL_BSP_SERVER_SIDE_COOKIE 481
CL_BSP_SERVICES 478
CL_BSPUTILITY 474
CL_HTTP_MANAGER 477
CL_HTTP_EXT_BASE_HANDLER 481
CL_HTTP_SERVER 477
CL_HTTP UTILITY 476
CLASSIC 172
compiler 27
component 282
composite element 231
  event handling 244
  using 245
composition 231
Connection, HTTP request header 38
Content-Length, HTTP response header 38
Content-Type, HTTP response header 38
Control Framework 253
controller 277
  eventing 281
  methods 278
  model lifetime 280
  modifications 293
sub- 282
cookie 54
  basic authentication 102
  SSO 102
  SSO2 91
Cookie, HTTP request header 36
credentials 97
CSS file 395
customization 395

data manipulation 366
data point, IGS 263
data-handling 249
date format 352
DDIC_UTILS 351
de-authentication 113
debugger 29
DEC VAX 342
decode_url 83
design pattern 275
DESIGN2002 172
DESIGN2003 172
development environment 25
DIAG protocol 136
digital certificates 105
dirty indicator 272
document handling 355
display in new window 366
display inline 361
display inside HTML 363
Excel 368
  non-HTML 358
download 49
dropdown list box 48
Dynpro 133, 187, 253, 288, 322

E
element class 312
element ID 246
Employee Self Services (ESS) 295
Enterprise Core Component 5.00 16
Enterprise Portal Client Framework (EPCF) 271
error handling 72
error pages 408
event
  handling in composite elements 244
  handling incoming 243
  handling manually 181
  rendering 240
  rendering via <bsp htmlbEvent> 243
event dispatching 180
  IF_HTMLB_EVENTS 183
  onClick 185
event-handling 250
  image click 267
eventing 281
Excel 355, 474
  interactive 449
Excel download 367
Excel Unicode text file 369
Extensible Stylesheet Language Transformations (XSLT) 456
extension framework 168

F
F1 help 307
  controller method 318
  model 318
F4 help 322
field help 307
field history 417, 419
field label 351
Flash 358
flow logic 16
form-based authentication 109, 111
FQDN 89
  browser requirements 91
Fully Qualified Domain Names 89
G
GET 43
GET_COLUMN_DEFINITIONS 218
get_cookie 57
getter method 286
GUID 364
GZip 392

H
HANDLE_REQUEST 72, 86
handler class 70
help UI 307
Host, HTTP request header 36
HTML 31
  mapping onto HTTP 46
HTML BEE 209
HTML forms 41
HTML Viewer 133, 142, 309
HTMLB 26, 168, 172, 174
  event system 180
HTMLB event manager 244, 271
HTMLB event system 240
HTMLB manager 247, 250
HTTP 21, 31, 97
  Header/Body Separator 37
  redirect 58
  request 34
  request body 37
  request headers 35
  request status line 35
  response 34
  response body 39
  response headers 38
  response status line 37
  return codes 37
  structure 32
HTTP cache 363
HTTP form field 475
HTTP handler 67
HTTP proxy 32
HTTP server 477
HTTP trace tool 153

I
ICF tree 67
ICM File Handler 357, 389
ID 246

IF_BSP_APPLICATION_EVENTS 119, 123
IF_BSP_BEE 205
IF_BSP_ELEMENT 227
IF_BSP_MODEL_SETTER_GETTER 286
IF_BSP_NAVIGATION 480
IF_BSP_RUNTIME 473
IF_HTMLB_DATA 182
IF_HTMLB_EVENTS 183
IF_HTTP_EXTENSION 69, 86
IF_HTTP_FORM_FIELDS 481
IF_HTTP_HEADER_FIELDS 72, 481
IF_HTTP_HEADER_FIELDS_SAP 72
IF_HTTP_REQUEST 40, 480
IF_HTTP_RESPONSE 40, 480
IF_HTTP_SERVER 119
IF_MR_API 481
IF_PHTMLB_EVENTS 184
IF_XHTMLB_EVENTS 184
IFrame 323, 426
IGS 260
  BSP extension 261
  chart data 262
  chart designer 267
  customizing 266
  image 355
  image click 267
  image conversion 268
  images 69
  input field 46
  input help 325
Intel X86 342
internationalization 337
Internet Communication Framework (ICF) 24, 67, 97, 156, 409
Internet Communication Manager (ICM) 21, 97, 115, 152, 357
  Configuration 91
Internet Graphics Service (IGS) 260
Internet mail extensions (MEs) 276
Internet Transaction Server (ITS) 260
invalid URL 64

J
Java applets 133
JavaScript libraries 319
L
List processing 423
Little Endian 342, 369, 371
load testing 161
load-balancing 115
logon application 110, 412
logon errors 412
logon language 338
switch 341
long texts 347

M
mangling 83
Microsoft Office 449
Microsoft Outlook Web Access 463
MIME objects
direct import 356
inline 357
MIME repository 30, 355, 481
alternatives 389
MIME repository browser 356
MIME type setting 343
Mini-Portal 458, 460
within SAP GUI 464
missing resource 62
model 276
lifetime 280
serialized 293
model binding 284
dynamic 288
model view binding 308
Model View Controller (MVC) 16, 184, 275, 417
Motorola 68000 342
Multi Display Multi Processing (MDMP) 343
multiple language support 337
MVC 275, 417
application structure 275
controller 277
model 276
pattern engine 295, 301
stateless 291
view 283
mySAP ERP 2005 16
MYSAPSSO2 110, 114, 151
MYSAPSSO2 cookie 102

N
namespace mapping 92
network latency 154
non-secure warnings 63

O
Office integration 367
Office Web Components (OWC) 449
onClick 182
onHeaderClick 188
onInputProcessing 124, 181
Online Text Repository (OTR) 346, 473, 476, 481
OWC control 450

P
Parameters 429
pattern engine 295, 301, 417
PDF 355, 358, 422
pending timeout 128
performance 65, 151
browser rendering time 157
runtime analysis 159
server processing time 155
SQL trace 165
statistical record 157
test 152
personalization 468
PHTMLB 16, 26, 172
pictures 358
ping 154
portal eventing 271
portal integration 270
portal navigation 272
POST 43
printing 422
dialog 426
PRINT method coding 424
PRINT method interface 423
processing timeout 121
PTHMLB 173

Q
Quick Info 351
quick sizing 153
R
R/3 337
R/3 Document Management System 390
R/3 Enterprise 16
radio button 48
RDF site summary (RSS) 451
rdisp/no_statistic 156
rdisp/plugin_auto_logout 129
redirect 58
Referer, HTTP request header 37
relative URL 63
RENDER_CELL_START 222
render_event_call 241
RENDER_ROW_START 220
Repository Browser 169
resource caching 60
restart 123
RFC destination maintenance 260
right to left (RTL) rendering 176, 473
RSHTTP01 452
RSS feed 451
RSSTATISTIC 156
runtime 28
runtime analysis 159
Runtime Type Information (RTTI) 329, 437
Runtime Type Services (RTTS) 289, 317, 440, 479

S
SAP Content and Cache Server 390
SAP Enterprise Portal 174, 270, 395, 458
SAP GUI 133, 464
interaction with BSPs 144
SAP GUI event 147
SAP GUI session 115
SAP Internet Transaction Server (ITS) 15
SAP NetWeaver 16
SAP Unified Framework 461
SAEVENT 144
sap-language 338
SAPMHTTP 157
SAPscript 253, 309
Select-Options 429
series, IGS 263
server cookie storage 292
server processing time 155
server runtime measurement 156
Server, HTTP request header 37
server-side printing 422
session 138
session ID 116
session identification 115
session management 115
browser-based 125
session timeout 118
in browser 120
set_cookie 56
setter method 286
shared-memory class 469
Single Sign-On (SSO) 91, 102, 110, 137, 151, 338
de-authentication 114
small loader page 143
Smalltalk 275
SmartForms 423
SPARC 342
SQL trace 165
SSO2 cookie 91
statistical record 157
stress testing 161
subcontroller 282
SVG 358
system logon 412
sy-uname 105

T
Table BEE 211
table query handler 78
table view iterator 215, 217, 422
Tag Browser 169
TCP/IP 21, 115, 136
testing 152, 161
text editor 253
Theme Editor 397
theme export 396
theme root 406
white list 406
timeout 118
pending 128
timestamp 476
transaction
DWDM 133
RZ11 118
SE16 406, 410, 442
SE30 159
SE38 156
SE63 350
SE80 25, 166, 232, 355
SM04 115, 124
SM30 105, 204
SM59 260, 344
SMICM 23, 121
ST22 38, 458
STRUST 105
SU01 340
SU3 460
translation 349
Translation Workbench 350
Transport Management System (TMS) 15
troubleshooting 62

U
Unicode 255, 338, 341, 369, 474
and Excel 369
in BSP 343
Unified Renderer 174, 396
upload 49
URL 83
1x1 127
detach in browser 149
URL escaping 95
URL mangling 83, 161
URL mapping 68
URL parameter 93
URL syntax 71
User-Agent, HTTP request header 37
UTF-X 342, 369, 371
utility classes 473

V
value help 307, 322
view 283
Visual Composer 174

W
weather display 466
Web Application Stress Tool 161, 162
Web Dynpro 174, 319
Web Dynpro ABAP 16
Window Open Behavior 139
WorkProtect 272

X
X.509 certificate 105
XHTMLB 26, 172, 173
XML BEE 212
XML serialization 291
XMLGROUP 81
XSLT 80

Z
ZIP 355
ZIP tool 392