ABAP Web Dynpro

by

vCentric
Outcome driven Solutions.
Contents

Unit 9: Dialog Boxes (Popups)
Dialog Boxes (Popups)

Unit 10: Dynamic Modifications at Runtime
Dynamic Modifications at Runtime

Unit 11: Special Topics
Adaptation of Web Dynpro Applications
SAP List Viewer for ABAP Web Dynpro
Portal Integration
Integrating Adobe Forms

Appendix 1: Phase Model

Appendix 2: Dynamic Programming - Advanced Topics

Appendix 3: Value Help - Advanced Topics

Appendix 4: Adaptation of Web Dynpro Applications – Advanced Topics

Appendix 5: Portal Integration
Unit 9

Dialog Boxes (Popups)

Overview
Dialog boxes are used to display concrete information or possible settings in a view popping up on top of the browser window the user clicked on. After the dialog has been exited, either the browser window underneath becomes active again or another screen may appear.

There are two different types of dialog boxes:

• External dialog box:
  An external dialog box is opened in a separate browser window and can be moved around the screen independently of the original window. External dialog boxes are generally modeless.

• Modal dialog box:
  A modal dialog box is opened in the current browser window. The modal dialog box may display a window of the same component, the interface view of a used component, or a confirmation dialog.

Common code section
Independent what kind of dialog box has to be created, the first part of the source code is identical. This consists of the following parts.

1. First, the reference to the component controller's functionality (API) has to be determined. To do so, the method `wd_get_api()` has to be called for the controller reference `WD_COMP_CONTROLLER`.

2. This method call will return a reference of type `IF_WD_COMPONENT`. To store this reference, a corresponding reference variable has to be created (`lo_api_component`).

3. Next, the reference to the so called window manager has to be determined. This object offers functionality to create all kinds of dialog boxes. To do so, the method `lo_api_component->get_window_manager()` has to be called.

4. This method call will return a reference of type `IF_WD_WINDOW_MANAGER`. To store this reference, a corresponding reference variable has to be created (`lo_window_manager`).

5. Now, the different kinds of dialog boxes can be created by calling the appropriate method of `lo_window_manager`.
6. Independent what kind of dialog box you created, you will receive a reference to an object
representing the dialog box. This can be used to open or close (only for modal popups) the dialog box. This reference is of type `IF_WD_WINDOW`. A corresponding reference variable has to be created.

![Image 1](image1.jpg)

**Figure 118: Dialog Box: Common Source Code**

If you have a SAP system based on SAP NetWeaver 7.0 with a support package level < 10, then you have to create the code manually. For higher SP levels, the Web Dynpro Code Wizard will generate the complete code for sending windows as modal dialog boxes. If you want to create an external dialog box or if you want to create a confirmation dialog box, you have to replace the generated method call (step 5) by the correct method call.

**External Dialog Box**
To create an external dialog box, the method `create_external_window()` of the window manager object is used. This method's parameter allow to set the window title and the URL of the object to be displayed in the dialog box. In addition there exist boolean parameters to display or hide the browser's menu bar, scroll bars, status bar, tool bar and address bar. Finally, the develop can decide whether the browser window should be resizeable or not.

```abap
METHOD onactionnext_popup.
  DATA lo_window_manager TYPE REF TO if_wd_window_manager.
  DATA lo_api_component TYPE REF TO if_wd_component.
  DATA lo_window TYPE REF TO if_wd_window.
  lo_api_component = wd_comp_controller->wd_get_api().
  lo_window_manager = lo_api_component->get_window_manager( ).

  lo_window = lo_window_manager->create_external_window( url = 'http://www.sap.com'
                                                        title = 'SAP Homepage' ).

  lo_window->open( ).
ENDMETHOD.
```
Modal Dialog Boxes
There are three different kinds of modal dialog boxes, that can be implemented with ABAP Web Dynpro:

1. Confirmation dialog box:
   This dialog box is used to display simple texts in a window with a standardized set of buttons to define the next step.

2. Dialog box displaying a window of the same component: This dialog box can be used to display any window of the same component as a popup. The content is arbitrary.

3. Dialog box displaying a window of a used component: This dialog box can be used to display any interface view of a used component as a popup.

Confirmation Dialog Box
A confirmation dialog box is a modal popup displaying an arbitrary text. In order to generate a confirmation dialog box, the method `create_popup_to_confirm()` of the window manager object has to be called. Parameters allow to set the text to be displayed by the confirmation dialog box and its title. Optionally, an icon can be displayed on the left side of the text indicating its significance (e.g. warning).

Buttons can be displayed in the dialog box, one of these buttons can be defined as the default button. To each of the buttons displayed by the dialog box a certain action can be assigned. The popup's close button in the upper right corner can be displayed or hidden.
METHOD onactionconf_popup.

DATA lo_window_manager TYPE REF TO if wd_window_manager.
DATA lo_api_component TYPE REF TO if wd_component.
DATA lo_window TYPE REF TO if wd_window.

DATA lt_string TYPE TABLE OF string.
INSERT 'This is a confirmation Popup INTO TABLE lt_string.

lo_api_component = wd_comp_controller->wd_get_api().
lo_window_manager = lo_api_component->get_window_manager().

lo_window = lo_window_manager->create_popup_to_confirm(
  text = lt_string,
  button_kind = 1).

lo_window->open().
ENDMETHOD.

Figure 121: Confirmation Dialog Box: Example

Figure 122: Confirmation Dialog Box: Source Code
Dialog Box displaying a Window of the same Component

In order to display a window of the same component as a dialog box, the method `create_window()` of the window manager object has to be called. Parameters allow to define, which window of the same component is to be displayed as a modal dialog box. The title of the modal dialog box can be set.

Buttons can be displayed in the dialog box, one of these buttons can be defined as the default button. To each of the buttons displayed by the dialog box a certain action can be assigned. The popup’s close button in the upper right corner can be displayed or hidden. Optionally, an icon can be displayed on the left side of the dialog box indicating the significance of the content (e.g. information).

```plaintext
METHOD onactionown_window.

DATA lo_window_manager TYPE REF TO if_wd_window_manager.
DATA lo_api_component TYPE REF TO if_wd_component.
DATA lo_window TYPE REF TO if_wd_window.

lo_api_component = wd_comp_controller->wd_get_api().
lo_window_manager = lo_api_component->get_window_manager().

lo_window = lo_window_manager->create_window(
  window_name = 'POPUP_WINDOW',
  title = 'This is a window of own comp.',
  button_kind = 1).

lo_window->open().
ENDMETHOD.
```

Figure 123: Window of same Component as Dialog Box: Example
In order to display a window of a used component as a dialog box, the method `create_window_for_cmp_usage()` of the window manager object has to be called. The title of the modal dialog box can be set. The name of the used component and the name of its interface view to be displayed as a dialog box has to be defined by setting the related interface parameters of the method.

Buttons can be displayed in the dialog box, one of these buttons can be defined as the default button. To each of the buttons displayed by the dialog box a certain action can be assigned. The popup's close button in the upper right corner can be displayed or hidden. Optionally, an icon can be displayed on the left side of the dialog box indicating the significance of the content (e.g. information). However, these settings can not be defined by just passing information to the method `create_window_for_cmp_usage()` , but by calling subsequent methods.

To interchange data between the consumer component and the used component containing the popup window, external context mapping can be used. A second way of passing data from the consumer component to the used component is to call a method defined in its interface controller. Passing data back to the consumer component can be implemented by raising an event defined in the interface controller using the event parameters. A method defined in any controller of the consumer component may then register for this event.
METHOD onactioncmp_window.

DATA lo_window_manager TYPE REF TO if_wd_window_manager.
DATA lo_api_component TYPE REF TO if_wd_component.
DATA lo_window TYPE REF TO if_wd_window.

lo_api_component = wd_comp_controller->wd_get_api().
lo_window_manager = lo_api_component->get_window_manager().

lo_window = lo_window_manager->create_window_for_cmp_usage(
  interface_view_name = 'POPUP_WINDOW',
  component_usage_name = 'SUB_CMP',
  title = 'This is a window of used component').

lo_window->open().
ENDMETHOD.

Figure 125: Window of used Component as Dialog Box: Example

Figure 126: Window of used Component as Dialog Box: Source Code
Exercise 13: Sending Window of used Component as Popup

Exercise Objectives
After completing this exercise, you will be able to:
• Send a window of a used component as a modal dialog box (popup)

Business Example
To display details for a given data set you can reuse an existing component. The interface view of this component can be embedded in a View-Container that is defined in one of the views related to the consumer component. However, you want to display this interface view as a modal dialog box (popup).

Template
Template: VCT_COMP_S1 (consumer component)
Template: VCT_COMP_S2 (used component)
Solution: VCT_POP_S1 (consumer component)
The used component will not be changed.

Task 1:
Copy your Web Dynpro component ZVCT_COMP1_## or the template VCT_COMP_S1 to Web Dynpro component ZVCT_POP1_##. This component will act as the consumer component.

1. Copy the template, activate your component, and create a Web Dynpro application.

Task 2:
You want to display the customer details in a popup. Thus the view container defined on view OUTPUT_VIEW can be deleted. In addition, embedding the interface view in the window of the consumer component has to be deleted, too.

1. Delete the ViewUIElementContainer UI element from the view OUTPUT_VIEW.
2. Delete the embedding of the interface view in the view container that is defined in the consumer component window.

Task 3:
The customer details should be displayed if the user marks a flight or a booking on view OUTPUT_VIEW. Since both events are related to the same action, you can add the code for displaying the modal dialog box to this action handler method.

1. Create and open a modal dialog box to display the interface view MAIN_WIND of the used component. Create the code in method ONACTIONSHOW_CUSTOMER of the view OUTPUT_VIEW.
2. Activate and test your consumer component.
Solution 13: Sending Window of used Component as Popup

Task 1:
Copy your Web Dynpro component _ZVCT_COMP1_## or the template _VCT_COMP_S1_ to Web Dynpro component _ZVCT_POP1_##. This component will act as the consumer component.

1. Copy the template, activate your component, and create a Web Dynpro application.
   a) Perform this step as in previous exercises.

Task 2:
You want to display the customer details in a popup. Thus the view container defined on view OUTPUT_VIEW can be deleted. In addition, embedding the interface view in the window of the consumer component has to be deleted, too.

1. Delete the View UI Element Container UI element from the view OUTPUT_VIEW.
   a) Edit the view OUTPUT_VIEW. Open the Layout tab.
   b) Mark the View UI Element Container UI element. Delete it by selecting the corresponding entry from the UI element's context menu. Save.

2. Delete the embedding of the interface view in the view container that is defined in the consumer component window.
   a) Edit the consumer component's window. Open the node related to view OUTPUT_VIEW.
   b) Mark the entry related to the interface view of the used component. This entry is marked with a special sign because the container has already been deleted.
   c) Delete the interface view by selecting the corresponding entry in its context menu.

Task 3:
The customer details should be displayed if the user marks a flight or a booking on view OUTPUT_VIEW. Since both events are related to the same action, you can add the code for displaying the modal dialog box to this action handler method.

1. Create and open a modal dialog box to display the interface view MAIN_WIND of the used component. Create the code in method ONACTIONSHOW_CUSTOMER of the view OUTPUT_VIEW.
   a) Edit the view OUTPUT_VIEW. Select the tab Methods. Double click on the entry ONACTIONSHOW_CUSTOMER to display the method's source code.
   b) Locate the cursor under the call of component controller method CUSTOMER_READ. Use the Web Dynpro Code Wizard to create and open a modal dialog box for the window MAIN_WIND of the used component.
   c) If you are working in a system having not installed support package 10 or higher proceed as follows:

Use the Web Dynpro Code Wizard to call the component controller method WD_GET_API.
This method hands back the reference to the component's API. Create a reference variable of type `IF_WD_COMPONENT` to store this reference.

Use the pattern functionality to call the method `GET_WIN-DOW_MANAGER` for the reference (component controller API) you obtained in the last step. This method hands back a reference to the window manager. Create a reference variable of type `IF_WD_WINDOW_MANAGER` to store this reference.

Use the pattern functionality to call the method `CREATE_WIN-DOW_FOR_CMP_USAGE` for the window manager reference. This will hand back a reference to the dialog box object. Create a reference variable of type `IF_WD_WINDOW` to store this reference.

Finally you have to open the window. Call the method `OPEN` for the reference to the dialog box object you obtained in the last step.

d) To set the title displayed by the modal dialog box, you can pass an appropriate text literal to the interface parameter `title` of method `CREATE_WINDOW_FOR_CMP_USAGE`.

2. Activate and test your consumer component.
   a) Perform this step as in previous exercises.

Result

**Model Solution: VCT_POP_S1, Method ONACTIONSHOW_CUSTOMER**

```
METHOD onactionshow_customer.
  data lo_window_manager type ref to if_wd_window_manager.
  data lo_api_component type ref to if_wd_component.
  data lo_window type ref to if_wd_window.
  wd_comp_controller->customer_read().
  lo_api_component = wd_comp_controller->wd_get_api().
  lo_window_manager = lo_api_component->get_window_manager().
  lo_window = lo_window_manager->create_window_for_cmp_usage(  
    interface_view_name = 'MAIN_WIND'  
    component_usage_name = 'CUSTOMER_COMP_USAGE'  
    title = 'Customer Details'  
    close_in_any_case = abap_true  
    message_display_mode = if_wd_window=>co_msg_display_mode_selected  
  ).
  lo_window->open().
ENDMETHOD.
```
Unit 10

Dynamic Modifications at Runtime

Dynamic Runtime Modifications: Overview
Dynamic modifications can be made at runtime as follows:

• **Dynamic Context Manipulation**
  Context nodes and context attributes can be created and deleted. The meta data of already existing context elements can be modified.

• **Dynamic UI Manipulation**
  UI elements can be created and added to the UI element hierarchy. Existing UI elements can be removed from the hierarchy, and their position or their properties can be changed at runtime.

• **Dynamic Assignment of Actions to UI Elements**
  Statically created actions can be assigned to statically or dynamically created UI elements.

Usually, you would declare the structure of a controller’s context at design time, and then create a view’s layout statically in order to display the data stored in the declared context. However, it is also possible to create a context structure and a view’s layout partially or completely at runtime. It is preferable, though, to create as much of the context and the view’s layout as possible at design time. Dynamic programming comprises a significant amount of manually written source code. There are several situations in which dynamic modifications could be required:

• If the structure of the data is not known at design time
• If the behavior of the screen will be generic
• If Web Dynpro utility components have to be developed, working in a generic manner
Dynamic Context Modifications

The figure above shows an example of a controller context structure to be created at runtime. The coding steps include the generation of two context nodes, one being a sub-element of the other, and context attributes related to these nodes.

Creating an Independent Context Node
All context nodes created by the application developer must have some other node acting as their parent. This is why a context is always supplied containing a root node. The root node is the anchor point for all other nodes, irrespective of whether they are created statically at design time or dynamically at runtime.
Thus, in order to create a new independent context node a reference to the meta data of the context root node has to be obtained first. This reference is obtained by calling the method `wd_context->get_node_info()`.

Next, the meta data of the new node has to be defined. To create a new node, the method `add_new_child_node()` from the reference to the context root node can be used. This method hands back the reference to the meta data of the new node.

An important parameter of the method `add_new_child_node()` is `is_static`. Only if this parameter is set to `abap_false` can the related node can be deleted at runtime.

**Figure 128: Dynamic Node Creation (1)**

**Coding steps:**

1. Obtain a reference (lo_nd_info) to the metadata of the context node that will act as the new node’s parent. In this case, we get a reference to the metadata of the root node.
2. Call method `lo_nd_info->add_new_child_node()` to define new node.
Creating Context Attributes

Each attribute of the previously created context node can be defined by calling the method `add_attribute()` from the reference to the meta data of the node.

```plaintext
DATA lo_nd_info_root TYPE REF TO if_wd_context_node_info.
DATA lo_nd_info_flights TYPE REF TO if_wd_context_node_info.

* get meta data info of root context node
  lo_nd_info_root = wd_context->get_node_info( ).

* create node with name 'FLIGHTS' (without attributes)
  lo_nd_info_flights = lo_nd_info_root->add_new_child_node( 
    name = 'FLIGHTS',
    is_mandatory = abap_false,
    is_multiple = abap_true,
    is_mandatory_selection = abap_false,
    is_multiple_selection = abap_false,
    is_singleton = abap_true,
    is_initialize_lead_selection = abap_true,
    is_static = abap_false ).
```

Figure 129: Dynamic Node Creation (2)

```plaintext
DATA lo_nd_info_root TYPE REF TO if_wd_context_node_info.
DATA lo_nd_info_flights TYPE REF TO if_wd_context_node_info.
DATA ls_attribute TYPE wdr_context_attribute_info.

* get meta data info of root context node
  lo_nd_info_root = wd_context->get_node_info( ).

* create node with name 'FLIGHTS' (without attributes)
  lo_nd_info_flights = 
    lo_nd_info_root->add_new_child_node( ... ).

* define attribute CARRID
  ls_attribute-name = 'CARRID',
  ls_attribute-type_name = 'S_CARR_ID',
  ls_attribute-value_help_mode = 'O'.

  lo_nd_info_flights->add_attribute( 
    attribute_info = ls_attribute ).
```

Figure 130: Dynamic Attribute Creation (1)
Creating a Context Node and Context Attributes from a Structure

Creating a context node and its attributes comprises a lot of manually written coding since the method `add_attribute()` has to be called once for each attribute.

However, the method `add_new_child_node()` not only allows you to create a context node, but also to create related attributes as follows:

- An ABAP Dictionary flat structure type (structure, database view or transparent table) can be passed to the method using the parameter `static_element_type`. All attributes for this structure are created automatically.
- An RTTI structure descriptor can be passed to the method using the parameter `static_element_rtti`. The attributes are created accordingly.
- Finally, a table, with each line containing one attribute description, can be passed to the method using the `attribute` parameter. Attributes are created accordingly.
**Coding steps:**

1. Obtain a reference (lo_nd_info) to the metadata of the context node that will act as the new node’s parent. In this case, we get a reference to the metadata of the root node.

2. Call method
   
   ```
   lo_nd_info->add_new_child_node(...).
   ```

3. Use parameter
   
   ```
   static_element_type
   ```
   to pass the name of DDIC structure to the method.

---

**Figure 132: Dynamic Creation of a Context Node from a DDIC Structure (1)**

---

```plaintext
DATA lo_nd_info_root   TYPE REF TO if_wd_context_node_info.  
DATA lo_nd_info_flights TYPE REF TO if_wd_context_node_info.  

* get meta data info of root context node
  lo_nd_info_root = wd_context->get_node_info( ).

* create node with name 'FLIGHTS' of DDIC type SFLIGHT
  * with all attributes in one step

  lo_nd_info_flights = lo_nd_info_root->add_new_child_node(  
    name            = 'FLIGHTS',
    static_element_type = 'SFLIGHT',
    attributes       =  
    is_static        = abap_false ).
```

**Figure 133: Dynamic Creation of a Context node from a DDIC Structure (2)**
Creating Dependent Context Nodes
Creating a sub-node of an already existing context node is very much the same as creating an independent context node. However, now the reference to the meta data of the parent node has to be obtained, which is different from the reference to the context root node.

If a context node is created dynamically, the reference to the new node is returned by the creation method. From this reference, further sub-nodes can be created. However, the reference to a node’s meta data can also be obtained at any time by determining the reference to the context node instance, followed by getting the reference to its meta data:

```
lo_nd_<node> =
wd_context->get_child_node( name = '<node>' )
lo_nd_info_<node> =
lo_nd_<node>->get_node_info( )
```

It is also possible to obtain the reference to the context root node's meta info in the first step and determine the reference to the node’s meta data afterward:

```
lo_nd_info_root =
wd_context->get_node_info( )
lo_nd_info_<node> =
lo_nd_info_root->get_child_node( name = '<node>' )
```

The second approach is preferable, since the existence of the parent node is guaranteed. From the reference to the meta info of an existing node, information about sub-nodes can be determined (for example, by using method get_child_nodes( )). This allows you to check the existence of sub-nodes before accessing them.
Coding steps:
1. Obtain a reference (lo Nd_info) to the metadata of the context node that will act as the new node’s parent. In this case, we get a reference to the metadata of the nodeFLIGHTS.
2. Add a child node and related attributes.

```abap
DATA lo Nd_info_root TYPE REF TO if_wd_context_node_info.
DATA lo Nd_info_flights TYPE REF TO if_wd_context_node_info.
DATA lo Nd_info_bookings TYPE REF TO if_wd_context_node_info.

* get reference to meta data of node FLIGHTS
  lo Nd_info_root = wd_context->get_node_info ( ).
  lo Nd_info_flights = lo Nd_info_root->
    get_child_node( name = 'FLIGHTS' ).

* create node with name 'BOOKINGS' of DDIC type SBOOK
  * as sub node of node 'FLIGHTS'
  lo Nd_info_bookings =
    lo Nd_info_flights->add_new_child_node( static_element_type = 'SBOOK'
      name = 'BOOKINGS'
      is_static = abap_false ).
```

Figure 134: Dynamic Sub-Node Creation (1)

Figure 135: Dynamic Sub-Node Creation (2)
Additional Interesting Methods Acting on Context Meta Data

The following table lists methods that are not discussed in this section. All of these methods belong to the interface `IF_WD_CONTEXT_NODE_INFO`.

<table>
<thead>
<tr>
<th>Method Meaning</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_attributes()</code></td>
<td>Returns the meta data of all attributes related to a given node</td>
</tr>
<tr>
<td><code>get_attribute()</code></td>
<td>Returns the meta data of a certain attribute related to a given node</td>
</tr>
<tr>
<td><code>get_child_nodes()</code></td>
<td>Returns the meta data of all sub-nodes related to a given parent node</td>
</tr>
<tr>
<td><code>get_child_node()</code></td>
<td>Returns the meta data of a certain sub-node related to a given parent node</td>
</tr>
<tr>
<td><code>get_parent()</code></td>
<td>Returns the meta data of the parent node to a given context node</td>
</tr>
<tr>
<td><code>remove_attribute()</code></td>
<td>Deletes an attribute for a given dynamic context node</td>
</tr>
<tr>
<td><code>remove_dynamic_attributes()</code></td>
<td>Deletes all dynamically created attributes for a given context</td>
</tr>
<tr>
<td><code>remove_child_node()</code></td>
<td>Deletes context node for a given parent node</td>
</tr>
</tbody>
</table>

Dynamic UI Modifications

To display the value of dynamically created context attributes or to keep the UI generic, the UI element hierarchy of a view can be manipulated at runtime. The source code for all of these changes has to be defined in the standard hook method `wddomodifyview()`, since only in this method is a reference to the UI element hierarchy (parameter `view`) provided by the Web Dynpro runtime.

`Wddomodifyview()` is called just before the view layout is rendered and it is called every time the view controller is processed. However, because Web Dynpro is a stateful technology, the developer has to take note of how often the same coding section is interpreted by the Web Dynpro runtime. The Web Dynpro runtime provides a parameter, `firsttime`, that can be used to assure that the source code is processed only once in the view controller's lifetime. This parameter has the value `abap_true` if the view's layout is rendered the first time; otherwise it has the value `abap_false`. 
The following coding principles must be adhered to during UI element manipulation:

• Only perform direct manipulation of UI element objects when it is not possible to control their behavior through context binding.
• UI manipulation is only permitted within the \texttt{wddomodifyview()} method of the view controller.
• \texttt{wddomodifyview()} has a Boolean parameter called \texttt{firsttime}. Typically, you will only build a dynamic UI element hierarchy when \texttt{firsttime = abap\_true}. This avoids unnecessarily rebuilding the UI element hierarchy.
• Do not implement coding that modifies the context in \texttt{wddomodifyview()}!

The context should be considered —read-only” during the execution of this method.
Accessing Elements
If new elements are going to be added to the UI element hierarchy of a view layout, the first step is to get a reference to the container element acting as the parent for the new element. This container element could be the $\textit{RootUIElementContainer}$ (for example, when beginning with an empty view layout) or any other container element. To obtain this reference, methods of the parameter \textit{view} have to be used.

To access an arbitrary element, the method $\text{get_element()}$ can be called. To access the $\textit{RootUIElementContainer}$ the special method $\text{get_root_element()}$ is appropriate, since no additional information has to be provided.

Both methods return a reference of type $\text{IF\_WD\_VIEW\_ELEMENT}$, which is a general interface implemented by all UI element classes. Accessing the special properties of a container element requires that you cast the reference to an object having the correct type (down cast). The super class for all container UI elements is $\text{CL\_WD\_UIELEMENT\_CONTAINER}$. All UI elements are described by inheritors of the superclass $\text{CL\_WD\_UIELEMENT}$.

\textbf{Hint: } To get familiar with the class hierarchy describing all the UI elements, check the inheritance tree related to the abstract superclass $\text{CL\_WD\_VIEW\_ELEMENT}$
Figure 138: Dynamic UI Manipulation: Accessing the Container Element (1)

DATA lo_ui_root TYPE REF TO if_wd_view_element.
DATA lo_container TYPE REF TO cl_wd_uielement_container.

* check, whether this method has been processed before
  IF first_time = abap_true.
* get reference to any node (here to root node)
  lo_ui_root = view->get_element(
    id = 'ROOTUIELEMENTCONTAINER' ).
* OR:
* get reference to root node
  lo_ui_root = view->get_root_element( ).
* down cast:
* necessary to access all container props of UI element
  lo_container != lo_ui_root.

Add UI elements to container, using reference lr_container

ENDIF.

Figure 139: Dynamic UI Manipulation: Accessing the Container Element (2)
Before adding new child elements to a container element, the element's properties should be adjusted. This is especially important for the Layout property, since this influences the properties of the child elements. To adjust properties that are not inherited by the container's child elements, methods of the container reference can be called (for example, set_width( )).

```
* set container properties
* define Layout
  cl_wd_matrix_layout=>new_matrix_layout(
    container = lo_container ).
* set width
  lo_container=>set_width( value = '100%' ).
...
```

Figure 140: Adjusting Container Properties and Assigning the Layout

To assign the desired layout to the container element, a static method of the class representing the layout must be called. All layout classes are subclasses of CL_WD_LAYOUT.

**Adding UI Elements to a Container**

Before a new UI element can be added to the UI element hierarchy, the UI element has to be created. For each kind of UI element, a corresponding class describing the element exists. All of these classes are subclasses of CL_WD_UIELEMENT.

These classes follow the naming convention CL_WD_<element_kind>, so the class describing a Label UI element would be CL_WD_LABEL.

To create a new element, all of these classes offer a static method. These methods have the name new_<element_kind>() and return a reference to an object representing the UI element. Set the properties of the UI elements by calling the appropriate methods of the generated object.

To define the value of the property LayoutData, an object representing this property has to be assigned to the UI element. For all possible values of the property LayoutData, a corresponding class exists, having a static method to create the property object and assign it to a UI element. All of these classes are subclasses of CL_WD_LAYOUT_DATA and follow the naming convention CL_WD_<layout_data_kind>. The static methods follow the naming convention new_<layout_data_kind>().

Thus, to create an object representing LayoutData = GridHeadData, the static method new_grid_head_data() of the class CL_WD_MATRIX_HEAD_DATA must be called.
To add a UI element to the UI element hierarchy, the `add_child()` method of the parent element has to be used. The `index` parameter allows you to define where the new element is inserted. If this parameter is not used, the new element is appended.

**Hint:** For composite UI elements like the `table` element, special methods exist to create the parent UI element and all of its child elements.

Examples can be found in the helper class `CL_WD_DYNAMIC_TOOL` (for example, method `create_table_from_node()`).

![Figure 141: Adding a New UI Element to the UI Element Hierarchy (1)](image-url)
Additional Methods Acting on UI Element Hierarchy

The following table lists some methods that may be used to manipulate the context of a view layout.

**Method Meaning Available for**

- `get_<prop>( )`: Returns the value of the all UI elements property `<prop>`
- `set_<prop>( )`: Sets the property `<prop>` all UI elements to the desired value
- `bind_<prop>( )`: Binds the property's value all UI elements to a context attribute/node
- `bound_<prop>( )`: Returns the context all UI elements attribute node to which the property `<prop>` is bound
- `set_on<event>( )`: Binds action to property UI elements, on `<event>` of UI element able to fire events
- `get_on<event>( )`: Gets action bound to UI elements, property on `<event>` of UI able to fire element events
- `get_child( )`: Returns the sub-element(s) container UI of a given UI element elements
- `get_children( )`: Deletes sub-element(s) of container UI elements
- `remove_child( )`: Deletes sub-element(s) of container UI elements
- `remove_all_children( )`: Adds UI element to container UI element hierarchy of elements container element; position can be defined.

Assigning Actions to UI Elements Dynamically

Certain UI elements can trigger client-side events (such as toggling a *Checkbox* or selecting a *Table* row). Web Dynpro uses the concept of *actions* for the client-side event to trigger the execution of a server-side method. Actions can be assigned to UI element events either declaratively at design time or dynamically at runtime.
Actions assigned dynamically can only refer to existing server-side action handler methods. It is **not** possible to define the source code of an action handler method dynamically. But it is possible to define, which existing action handler will be called when a client-side event is caught.

You can assign an action to an UI element dynamically when creating the UI element or at a later point in time. Any UI element that allows triggering client-side events has a property `on_<event>` for each of the events. The static method used to create such a UI element has an import parameter for each of the client-side events. This allows you to hand over the name of the action to be assigned as a string.

For example, a *Button* has the property `on_action` related to the client-side event —pressing the button*. The static method `new_button()` of the class `CL_WD_BUTTON` is used to create the button. This method allows you to assign the action using the import parameter `on_action`. If an action is to be assigned to an already existing UI element, a corresponding setter method can be used for the object representing the UI element. This method follows the naming convention `set_on_<event>( )` (an action can be assigned to a button, using the method `set_on_action( )`).
DATA lo_button TYPE REF TO cl_wd_button.

* create button
lo_button = cl_wd_button->new_button(
    id = 'BUTTON_DISP'
    on_action = 'SELECT_FLIGHTS'
    text = 'Text of Button' )

* OR: assign Action at runtime to already existing button
lo_button->set_on_action( value = 'SELECT_FLIGHTS' ).

Figure 144: Assigning an Action to a UI Element (1)

Figure 145: Assigning an Action to a UI Element (2)
Parameter Extraction

If an action is assigned to multiple UI elements, the action handler should be able to distinguish between the UI elements in order to offer element specific functionality. All action handler methods have the interface parameter `wdevent`, which is filled by the Web Dynpro runtime if a client-side event is fired. The following information can be extracted from this parameter:

- `wdevent name`: The name of the client side event (for example, `ON_ACTION`).
- `wdevent parameters`: This internal table contains additional information about the UI element that fired the event. The table has the columns `name` and `value`. The ID of the UI element that fired the event is always contained in the table. Additional `name` / `value` pairs may be available depending on the type of UI element that fired the event (such as `CHECKED = 'X'` or `CHECKED=''` for a checkbox).

![Figure 146: Action Handling: The Interface Parameter WDEVENT](image)

If the information available from the interface parameter `wdevent` is not sufficient, additional information can be passed from the UI element to the action handler method. This technique is called `parameter mapping`. Details can be found in the appendix of this course.
Coding Example: Data Browser

Figure 147: Example for Dynamic Programming (1)

```plaintext
DATA lv_tabname TYPE ddobjname.
DATA lv_ddic_check TYPE dd02v-tabclass.

* get table name from context
  ...
  TRANSLATE lv_tabname TO UPPER CASE.

* check if this is the name of a transparent table
  * in the ABAP DDIC
    lv_ddic_check = cl_ws_helper=>get_ddobjtype( 
      tabname = lv_tabname ).

  IF lv_ddic_check <> 'TRANS'.
  * message handling
    ...
    ENDDIF.
```

Figure 148: Example for Dynamic Programming (2)
* define context structure and read data
  wd_this->check_user_input().

**WDDOBEBEFOREACTION**

* define context structure and read data
  wd_this->define_context().

* set global boolean variable: context node and content
  * exists now
  wd_this->gv_initialized = abap_true.

**ONACTIONDISPLAY**

Figure 149: Example: **WDDOBEBEFOREACTION** and the Action Handler Method

DATA lv_tabname TYPE dddobjname.
DATA lo_nd_info_root TYPE REF TO if_wd_context_node_info.
DATA lo_nd_info_dyn TYPE REF TO if_wd_context_node_info.
DATA lt_child_nodes TYPE wdr_context_child_info_map.
DATA ls_child_node TYPE wdr_context_child_info.

...

* delete old context node having wrong structure (if existing)
  lo_nd_info_root = wd_context->get_node_info()
  lt_child_nodes = lo_nd_info_root->get_child_nodes()
  READ TABLE lt_child_nodes INTO ls_child_node
    WITH KEY name = 'DB_TAB'.
  IF sy-subrc = 0.
    lo_nd_info_root->remove_child_node('DB_TAB').
  ENDIF.

* create new context node with attributes
  lo_nd_info_dyn = lo_nd_info_root->add_new_child_node(
    name     = 'DB_TAB'
    static_element_type = lv_tabname
    is_static       = abap_false ).

**DEFINE_CONTEXT**

Figure 150: Example: Checking User Input
Figure 151: Example: Building the Context

```abap
DATA lv_tabname TYPE ddobjname.
DATA lr_db_tab TYPE REF TO data.
DATA lo_nd_dyn TYPE REF TO if_wd_context_node.

FIELD-SYMBOLS: <lt_db_tab> TYPE ANY TABLE.

... 

* create data object of correct type and assign field symbol
CREATE DATA lr_db_tab TYPE TABLE OF (lv_tabname).
ASSIGN lr_db_tab->* TO <lt_db_tab>.

* read DB content
SELECT * FROM (lv_tabname) INTO CORRESPONDING FIELDS OF TABLE <lt_db_tab> UP TO 100 ROWS.

* bind table to context
lo_nd_dyn = wd_context->get_child_node( name = 'DB_TAB' ).
lo_nd_dyn->bind_table( new_items = <lt_db_tab> ).
```

Figure 152: Example: Reading Data and Filling Context

```abap
DATA lo_group TYPE REF TO cl_wd_group.
DATA lo_table TYPE REF TO cl_wd_table.
DATA lo_nd_dyn TYPE REF TO if_wd_context_node.

* Check if context is defined and data are read
CHECK wd_this->gv_initialized = abap_true.

* Get reference to group, which shall be modified
lo_group = view->get_element( id = 'GROUP_2' ).

* remove table, if it does not exist, nothing happens
lo_group->remove_child( id = 'DB_TABLE' ).

* Define TABLE bound to node having the name gv_nodename ...
lo_table = cl_wd_dynamic_tool->create_table_from_node( ui_parent = lo_group
table_id = 'DB_TABLE'
node = lo_nd_dyn ).

* hide mark column
lo_table->set_selection_mode( value = '06' ).
```

"06 = NONE"
DATA lo_group  TYPE REF TO cl_wd_group.
DATA lo_table  TYPE REF TO cl_wd_table.
DATA lo_nd_dyn TYPE REF TO if_wd_context_node.

* Check if context is defined and data are read
  
  CHECK wd_this->gv_initialised = abap_true.

* Get reference to group, which shall be modified
  
  lo_group ?= view->get_element( id = 'GROUP_2' ).

* remove table, if it does not exist, nothing happens
  
  lo_group->remove_child( id = 'DB_TABLE' ).

* Define TABLE bound to node having the name gv nodename
  
  lo_table = cl_wd_dynamic_tool->create_table_from_node(
    ui_parent = lo_group
    table_id = 'DB_TABLE'
    node    = lo_nd_dyn ).

* hide mark column
  
  lo_table->set_selection_mode( value = '06' ).  "06 = NONE"

| WDDOMODIFYVIEW |

**Figure 153: Example: Defining the Layout**
Exercise 14: Dynamic Modifications at Runtime

Exercise Objectives
After completing this exercise, you will be able to:
• Create context nodes and context attributes dynamically
• Add elements to the UI element hierarchy dynamically

Business Example
You want to develop a data browser based on ABAP Web Dynpro. Since the name of the database table to be displayed is not known at design time, you want to define the context structure and the UI at runtime.

Template : VCT_DYN_T
Solution : VCT_DYN_S

Task 1:
Copy the Web Dynpro component VCT_DYN_T to Web Dynpro component ZVCT_DYN_##. Test the functionality of the template Web Dynpro application.

1. Copy the template and create a Web Dynpro application.
2. Activate the component and test the application.

Result
The template component consists of two views: one to enter the name of a database table, and a second one to display the content of the database table. The user input on the first view is checked. If the name entered in the input field is not the name of a transparent table, an error message is displayed. Otherwise, the second view is processed. In the template component nothing is displayed on the second view, since neither the context structure for storing the database table content nor the TABLE UI element to display this content has been created yet.

Task 2:
In the second view (RESULT_VIEW), method HANDLEFROM_START_VIEW, define a context structure for the selected database table. This structure should consist of one node having attributes for each database table column. Read the first 100 data sets from the selected database table and store the data in the dynamically defined context structure.

1. Edit the method HANDLEFROM_START_VIEW of the view RESULT_VIEW.
2. Get the reference to the meta data of the context root node.
3. Call the method ADD_NEW_CHILD_NODE( ) for this reference. Use the pattern for calling methods to generate the statement.

Evaluate the export parameter STATIC_ELEMENT_TYPE with the name of transparent table to be displayed (variable LV_TABNAME).

Evaluate the export parameter NAME, which is the name of the context node to be created
(suggested name: DB_TABLE).
The method returns a reference to the generated context node. Store the returned reference in
the variable LO_ND_INFO_DYN. This reference variable has already been created.

4. Create the data reference variable LR_DB_TAB. This variable will be used to hold the data
sets read from the database table. Type this variable with an internal table type corresponding
to the database table the user wants to display.

Hint: Use the statement: CREATE DATA lr_db_tab TYPE TABLE OF (lv_tabname), where
lv_tabname is the variable containing the name of the transparent database table.

5. Dereference LR_DB_TAB into the field symbol <LT_DB_TAB>. This field symbol has already
been declared and will be used in the SELECT statement.

6. Select the first 100 rows of the database table requested by the user and store them in field
symbol <LT_DB_TAB>.

Hint: The table name can be used dynamically in the select statement as follows: SELECT *
FROM (lv_tabname) INTO ...

7. Determine the reference to the context node you created previously. The reference can be
stored in the variable LO_ND_DYN, which has already been defined.

8. Call the method BIND_TABLE( ) for this reference to store the selected data in the context.
Use the pattern for calling methods to generate the statement.

Task 3:
In the second view (RESULT_VIEW), define a TABLE UI element as a child of the already
existing GROUP UI element (name: GROUP_1). Display the data stored in the context
structure you defined in the last task.

1. Edit the hook method WDDOMODIFYVIEW( ) of the view RESULT_VIEW. Check if this
method is called for the first time (parameter FIRST_TIME).
2. Get the reference to the GROUP UI element GROUP_1.

Call the method GET_ELEMENT( ) for the parameter VIEW. Use the pattern for calling methods
to generate the statement.

Hint: The method returns a reference of type IF_WD_VIEW_ELEMENT. To handle the object
correctly, you have to cast this reference. The correct type for GROUP UI elements is
CL_WD_GROUP. You can either work with two reference variables and cast after having called
the method, or change your statement in order to cast the receiving parameter directly
(GET_ELEMENT( ) is a functional method).

3. Define a reference variable of type CL_WD_TABLE (suggested name: LO_TABLE). This will
point to the TABLE UI element to be created later.
4. Define a reference variable of type IF_WD_CONTEXT_NODE (suggested name: LO_ND_DYN). Store the reference to the node containing the selected data sets in this reference variable.

5. Call the static method CL_WD_DYNAMIC_TOOL=>CREATE_TABLE_FROM_NODE( ) to create the TABLE UI element, and add it to the UI element hierarchy. Use the pattern for calling methods to generate the statement. Use an arbitrary UI element ID (parameter TABLE_ID; suggested name: TABLE).

Pass the reference to the GROUP UI element to the method using the parameter UI_PARENT. Pass the reference LO_ND_DYN to the method using the parameter NODE. Store the returned reference to the created TABLE UI element in the variable LO_TABLE.
Solution 14: Dynamic Modifications at Runtime

Task 1:
Copy the Web Dynpro component VCT_DYN_T to Web Dynpro component ZVCT_DYN_##. Test the functionality of the template Web Dynpro application.

1. Copy the template and create a Web Dynpro application.
   a) Perform this step as in previous exercises.
2. Activate the component and test the application.
   a) Perform this step as in previous exercises.

Result
The template component consists of two views: one to enter the name of a database table, and a second one to display the content of the database table. The user input on the first view is checked. If the name entered in the input field is not the name of a transparent table, an error message is displayed. Otherwise, the second view is processed. In the template component nothing is displayed on the second view, since neither the context structure for storing the database table content nor the TABLE UI element to display this content has been created yet.

Task 2:
In the second view (RESULT_VIEW), method HANDLEFROM_START_VIEW, define a context structure for the selected database table. This structure should consist of one node having attributes for each database table column. Read the first 100 data sets from the selected database table and store the data in the dynamically defined context structure.

1. Edit the method HANDLEFROM_START_VIEW of the view RESULT_VIEW.
   a) Perform this step as in previous exercises.
2. Get the reference to the meta data of the context root node.
   a) Refer to the source code at the end of this exercise.
3. Call the method ADD_NEW_CHILD_NODE( ) for this reference. Use the pattern for calling methods to generate the statement.

Evaluate the export parameter STATIC_ELEMENT_TYPE with the name of transparent table to be displayed (variable LV_TABNAME).

Evaluate the export parameter NAME, which is the name of the context node to be created (suggested name: DB_TABLE ).

The method returns a reference to the generated context node. Store the returned reference in the variable LO_ND_INFO_DYN. This reference variable has already been created.

a) Refer to the source code at the end of this exercise.

4. Create the data reference variable LR_DB_TAB. This variable will be used to hold the data sets read from the database table. Type this variable with an internal table type corresponding to the database table the user wants to display.
**Hint:** Use the statement: `CREATE DATA lr_db_tab TYPE TABLE OF (lv_tabname)` , where `lv_tabname` is the variable containing the name of the transparent database table.

a) Refer to the source code at the end of this exercise.

5. Dereference LR_DB_TAB into the field symbol <LT_DB_TAB>. This field symbol has already been declared and will be used in the SELECT statement.
   a) Refer to the source code at the end of this exercise.

6. Select the first 100 rows of the database table requested by the user and store them in field symbol <LT_DB_TAB>.

**Hint:** The table name can be used dynamically in the select statement as follows: `SELECT * FROM (lv_tabname) INTO ...` .

a) Refer to the source code at the end of this exercise.

7. Determine the reference to the context node you created previously. The reference can be stored in the variable LO_ND_DYN, which has already been defined.
   a) Refer to the source code at the end of this exercise.

8. Call the method `BIND_TABLE( )` for this reference to store the selected data in the context. Use the pattern for calling methods to generate the statement.
   a) Refer to the source code at the end of this exercise.
Task 3:
In the second view (RESULT_VIEW), define a TABLE UI element as a child of the already existing GROUP UI element (name: GROUP_1). Display the data stored in the context structure you defined in the last task.

1. Edit the hook method WDDOMODIFYVIEW( ) of the view RESULT_VIEW. Check if this method is called for the first time (parameter FIRST_TIME).
   a) Refer to the source code at the end of this exercise.

2. Get the reference to the GROUP UI element GROUP_1. Call the method GET_ELEMENT( ) for the parameter VIEW. Use the pattern for calling methods to generate the statement.

   Hint: The method returns a reference of type IF_WD_VIEW_ELEMENT. To handle the object correctly, you have to cast this reference. The correct type for GROUP UI elements is CL_WD_GROUP. You can either work with two reference variables and cast after having called the method, or change your statement in order to cast the receiving parameter directly (GET_ELEMENT( ) is a functional method).
   a) Refer to the source code at the end of this exercise.

3. Define a reference variable of type CL_WD_TABLE (suggested name: LO_TABLE). This will point to the TABLE UI element to be created later.
   a) Refer to the source code at the end of this exercise.

4. Define a reference variable of type IF_WD_CONTEXT_NODE (suggested name: LO_ND_DYN). Store the reference to the node containing the selected data sets in this reference variable.
   a) Refer to the source code at the end of this exercise.

5. Call the static method CL_WD_DYNAMIC_TOOL=>CREATE_TABLE_FROM_NODE( ) to create the TABLE UI element, and add it to the UI element hierarchy. Use the pattern for calling methods to generate the statement. Use an arbitrary UI element ID (parameter TABLE_ID; suggested name: TABLE).

   Pass the reference to the GROUP UI element to the method using the parameter UI_PARENT.
   Pass the reference LO_ND_DYN to the method using the parameter NODE.

   Store the returned reference to the created TABLE UI element in the variable LO_TABLE.
   a) Refer to the source code at the end of this exercise.
RESULT

MODEL SOLUTION: VCT_DYN_S1, METHOD
HANDLEFROM_START_VIEW

METHOD handlefrom_start_view.

DATA lo_nd_info_root TYPE REF TO if_wd_context_node_info.
DATA lo_nd_info_dyn TYPE REF TO if_wd_context_node_info.
DATA lo_nd_table_name TYPE REF TO if_wd_context_node.
DATA lo_nd_dyn TYPE REF TO if_wd_context_node.
DATA lo_el_table_name TYPE REF TO if_wd_context_element.
DATA lv_tabname TYPE string.
DATA lr_db_tab TYPE REF TO data.
FIELD-SYMBOLS <lt_db_tab> TYPE ANY TABLE.

* read db table name from context
  lo_nd_table_name = wd_context->get_child_node( name = wd_this->wdctx_table_name ).
  lo_el_table_name = lo_nd_table_name->get_element().

  lo_el_table_name->get_attribute( EXPORTING name = 'TABLENAME' IMPORTING value = lv_tabname ).

  TRANSLATE lv_tabname TO UPPER CASE.

  * get meta data info of root context node
    lo_nd_info_root = wd_context->get_node_info( ).

  * create context structure
    lo_nd_info_root->add_new_child_node( EXPORTING static_element_type = lv_tabname name = 'DB_TABLE' RECEIVING child_node_info = lo_nd_info_dyn ).

  * create data object of correct type and assign field symbol
    CREATE DATA lr_db_tab TYPE TABLE OF (lv_tabname). ASSIGN lr_db_tab->* TO <lt_db_tab>.

  * read DB content
    SELECT * FROM (lv_tabname) INTO CORRESPONDING FIELDS OF TABLE <lt_db_tab> UP TO 100 ROWS.

  * bind table to context
    lo_nd_dyn = wd_context->get_child_node( name = "DB_TABLE" ).
    lo_nd_dyn->bind_table( EXPORTING new_items = <lt_db_tab> ).

ENDMETHOD.
METHOD wddomodifyview.

DATA lo_group TYPE REF TO cl_wd_group.
DATA lo_table TYPE REF TO cl_wd_table.
DATA lo_nd_dyn TYPE REF TO if_wd_context_node.

* Check, whether this method has been processed before
IF first_time = abap_true.

* Get reference to group, which shall be modified
lo_group ?= view->get_element( id = 'GROUP_1' ).

******************************************************************************
* Define table bound to node DB_TABLE
******************************************************************************
lo_nd_dyn = wd_context->get_child_node( name = 'DB_TABLE' ).
cl_wd_dynamic_tool=>create_table_from_node(
EXPORTING
  ui_parent = lo_group
  table_id = 'TABLE_DB_TABLE'
  node = lo_nd_dyn
RECEIVING
  table = lo_table).

** hide mark column
* CALL METHOD lo_table->set_selection_mode
* EXPORTING
  * value = '06'. "NONE

ENDIF.

ENDMETHOD.
Unit 11

Special Topics

Adaptation of Web Dynpro Applications
The UI of applications defined with ABAP Web Dynpro can be adapted in different ways and by different user groups. There are two categories of application adaptation: configuration and personalization.

**Configuration** is a concept that lets the developer create configuration data sets containing values for UI element properties or context attributes (typically bound to UI element properties). This allows the developer to overwrite many of the statically defined UI element properties, resulting in a different look and feel of the application (UI elements may be set to invisible, tables may have an alternating row color, and so on).

In contrast to configuration, **personalization** allows any user of the application to change the UI element properties at runtime. However, these changes are very restricted (for example, for simple UI elements like the `TextView`, only the visibility can be changed; for the `Table` element, the order of the columns can also be altered). Personalization is user dependent.

In contrast, **Customizing** is the ability to personalize Web Dynpro applications for all users in a client. To possibilities of customizing are much more far-reaching than the possibilities offered by personalization. To be able to customize an application, special authorizations are required.

**Adaptation Hierarchy**
The concepts of configuration, customizing, and personalization can be combined to define the final adaptation. Here, the changes defined by personalization overwrite the changes defined by customizing, and customizing overwrites the configuration adaptation. On the other side, the parameters available for configuration can be set to final so they cannot be changed using customizing or personalization. Parameters available for customizing can be set to final so they cannot be changed using personalization.
ABAP Web Dynpro

Figure 154: Personalization, Customizing, and Configuration

Implicit and Explicit Adaptation
ABAP Web Dynpro offers configuration, customizing, and personalization out of the box. This means that adapting a Web Dynpro application is possible without any programming effort. This kind of adaptation is called implicit adaptation. If the flexibility provided by implicit adaptation is not sufficient, the developer can modify the application in a way that all adaptations based on changing context attributes are possible. However, for this explicit adaptation, programming effort is required.

Figure 155: Implicit and Explicit Adaptation
**Implicit Configuration**

To change a Web Dynpro application using implicit configuration, configuration data sets must be created. Configuration data sets are related to components or applications. **Component configurations** allow you to change properties of UI elements defined in any view of a single component. For each component, an arbitrary number of component configurations can be defined. **Application configurations** are bound to Web Dynpro applications. They define which component configuration is used for which component in this application. For each application, an arbitrary number of application configurations can be created. Application changes related to configuration affect every user of this application in every client.

The following steps have to be executed to define a component configuration:

1. Choose *Create/Change Configuration* from the *Context* menu of a Web Dynpro component. This starts the configuration tool in the Web browser.
2. In the *Component Configuration* tray, enter the name of the component configuration to be created. **Caution:** In a customer system, you may only choose names in the customer namespace (beginning with *Z* or *Y*).
3. In the *Functions* tray, choose *Create*. This will open a dialog where you can change UI element properties.
4. On the *Implicit Configuration* tab, choose the UI element you want to manipulate. Change the property values and set the *Final* flag if you do not want this property to be changed by customizing or by personalization.
5. Finally, press the *Save* button in the *Changed Configuration* tray.
6. Close the browser. In the ABAP Workbench, refresh the object list. The component
configuration can be found as a sub-element of the Web Dynpro component.

After having defined a component configuration for each component used in a Web Dynpro application, the application configuration can be created:
1. Choose Create/Change Configuration from the Context menu of a WebDynpro application. This starts the configuration tool in the Web browser.
2. In the Application Configuration tray, enter the name of the application configuration to be created. **Caution:** In a customer system you may only choose names in the customer namespace (beginning with Z or Y).
3. In the Functions tray, choose Create. This will open a dialog where you can choose which configuration will be used for which component.
4. In the Assignment to Component Configuration group, choose a component usage. In the group below, choose which configuration should be used for this component usage.
5. Finally, press the Save button in the Changed Configuration tray.
6. Close the browser. In the ABAP Workbench, refresh the object list. The application configuration can be found as a sub-element of the Web Dynpro application.

**Using an Application Configuration**
As long as the application configuration is not assigned to the related application, the changes defined by the application configuration are not visible. There are two ways to assign an application configuration to a Web Dynpro application:

1. Dynamic assignment
   Add the query string sap-wd-configID=<appl_config> to the URL used to start the Web Dynpro application. Here, <appl_config> is the name of the application configuration.

2. Static assignment
   Edit the parameter list related to the Web Dynpro application. Add the parameter WDCONFIGURATIONID. In the Value column, enter the name of the application configuration.

**Implicit Customizing**
Implicit Customizing is provided by the Web Dynpro runtime and can be used by everyone having sufficient authorizations. To be able to customize a Web Dynpro application, the application must be started by adding the query string

SAP-CONFIG-MODE=X to the application's URL. Customizing is conducted by right-clicking on any UI element. A context menu appears. By selecting the entry Settings for current Configuration the customizing dialog is displayed.

Before the customizing dialog starts, the user's authorization for the authorization object S_DEVELOP is checked. If a sufficient authorization for S_DEVELOP is not found, the authorization for the authorization object S_WDR_P13N is checked. For each UI element, a predefined number of properties can be changed. Elements that are excluded from customizing by configuration are not available. Selecting the Final checkbox for any property excludes this property from personalization. The personalization data sets are independent of the user.
Implicit Personalization

Implicit personalization is provided by the Web Dynpro runtime and can be used by anyone starting any Web Dynpro application. Personalization is conducted by right-clicking on any UI element. A context menu appears. For each UI element the visibility can be changed. For more complex elements like the Table UI element, additional properties may be altered (such as the order of columns). Elements that are excluded from personalization by customizing or configuration are not available. The personalization data sets are user dependent.

Figure 157: Implicit Customizing

Figure 158: Implicit Personalization
Explicit Configuration
To allow an explicit configuration of a component, a configuration controller has to be created. A configuration controller is a special custom controller. Only one configuration controller may exist for each Web Dynpro component. All attributes that will be accessible via configuration have to be defined in the context of the configuration controller. Using context mapping and data binding, these attributes can then be used to change UI element properties directly in any view. However, any other functional changes based on these attributes are also possible, since the attributes are visible to all controllers that declare the usage of the configuration controller.

To create a configuration controller, a custom controller has to be created for the component. From the context menu (Re)Set as Config. Controller this controller can then be transformed to the component's configuration controller.

The developer has to decide which attributes are to be defined in the component controller and how changing these attributes will influence the functionality and the UI of this component. When defining a component configuration, not only predefined UI element property values can be changed, but also the values of the attributes defined in the configuration controller. These attributes are accessible via the Explicit Configuration tab.

![Figure 159: Explicit Configuration](image)

**Hint:** Attributes defined in the configuration controller are not automatically available for customizing and personalization!
Exercise 15: Special Topics: Adaptation via Configuration

Exercise Objectives
After completing this exercise, you will be able to:
• Create a component configuration
• Create an application configuration
• Assign an application configuration statically to a Web Dynpro application
• Use configuration controllers to implement explicit configuration

Business Example
You have created a Web Dynpro component and a Web Dynpro application to access the component. Now you have to adapt the look and feel of the Web Dynpro application for different user groups. Depending on the user group, the properties of some UI elements need to be modified. In addition, a text displayed on the screen should be adapted according to the users' needs. To implement these tasks, you want to use the concept of implicit and explicit configuration.

Template: NET310_CONF_T
Solution: NET310_CONF_S

Task 1:
Copy the template VCT_CONF_T to Web Dynpro component ZVCT_CONF_##. Test the functionality of your copy.
1. Copy the template, activate your component and create a Web Dynpro application.
2. Test the functionality of your Web Dynpro application. What happens if a nonexistent connection is chosen? Analyze the component to understand the behavior.

Task 2:
To make the text on the view NOTHING_FOUND_VIEW adaptable, you have to create a configuration controller. In the controller's context, define an attribute that will serve as the mapping origin for the attribute SETTINGS.EMPTY_TABLE_TEXT of the view NOTHING_FOUND_VIEW. This is the prerequisite to access the text via explicit configuration.

1. In your Web Dynpro component, create a custom controller (suggested name: CONF_CTRL) and make this controller the component's configuration controller.
2. In the context of the configuration controller, define a node (suggested name: SETTINGS) and an attribute (suggested name: EMPTY_TABLE_TEXT) of type STRING.
3. Map the SETTING node of the view NOTHING_FOUND_VIEW to the node of the configuration controller you defined in the last step. Before you can map the nodes, you have to declare the usage of the configuration controller in the properties of the view NOTHING_FOUND_VIEW.

Task 3:
Adapt the look and feel of your component using implicit configuration. Change the text that appears if no flights is found by using explicit configuration.
1. Create a component configuration for your component. Name the component configuration `ZVCT_COMP_CONFIG_##`
2. Begin with the implicit configuration. Change the following properties.

**View UI Element Property Value**
- INPUT_VIEW GROUP_1 Design secondary- boxcolor all labels Design highlighted OUTPUT_VIEW
- GROUP_2 Design secondary- boxcolor FLIGHTTAB Design Alternating FLIGHTTAB_PLANETYPE Visibility Invisible

3. Explicit configuration: Choose a text for the attribute SETTINGS.EMPTY_TABLE_TEXT.

**Task 4:**
Create an application configuration for your Web Dynpro application. Assign this application configuration to your Web Dynpro application statically.

1. Create the application configuration (name: `ZVCT_APPL_CONFIG_##`).

**Task 5:**
Assign the application configuration to your Web Dynpro application statically. Test your Web Dynpro application.

1. Add the name/value pair WDCONFIGURATIONID /ZNET310_APPL_CONFIG_## to the parameter list of your WebDynpro application.
Solution 15: Special Topics: Adaptation via Configuration

Task 1:
Copy the template VCT_CONF_T to Web Dynpro component ZVCT_CONF_##. Test the functionality of your copy.
1. Copy the template, activate your component and create a Web Dynpro application.
   a) Perform this step as in previous exercises.
2. Test the functionality of your Web Dynpro application. What happens if a nonexistent connection is chosen? Analyze the component to understand the behavior.
   a) If an existing connection is chosen, a table containing all flights of this connection is displayed. If a nonexistent connection is chosen, the table disappears. This is because the view containing the table (OUTPUT_VIEW) is replaced by another view (NOTHING_FOUND_VIEW) that only contains a TEXTVIEW UI element. The text property of this element is bound to the context attribute SETTINGS.EMPTY_TABLE_TEXT. However, the value of this context attribute is not set, so no text appears on the screen.

Task 2:
To make the text on the view NOTHING_FOUND_VIEW adaptable, you have to create a configuration controller. In the controller’s context, define an attribute that will serve as the mapping origin for the attribute SETTINGS.EMPTY_TABLE_TEXT of the view NOTHING_FOUND_VIEW. This is the prerequisite to access the text via explicit configuration.
1. In your Web Dynpro component, create a custom controller (suggested name: CONF_CTRL) and make this controller the component’s configuration controller.
   a) Create the custom controller by choosing the option from the context menu of your component.
b) From the context menu of the custom controller, choose (Re)Set as Config. Controller to make this custom controller the component's configuration controller.
2. In the context of the configuration controller, define a node (suggested name: SETTINGS) and an attribute (suggested name: EMPTY_TABLE_TEXT) of type STRING.
   a) Perform this step as in previous exercises.
3. Map the SETTING node of the view NOTHING_FOUND_VIEW to the node of the configuration controller you defined in the last step. Before you can map the nodes, you have to declare the usage of the configuration controller in the properties of the view NOTHING_FOUND_VIEW.
   a) Perform this step as in previous exercises.

Task 3:
Adapt the look and feel of your component using implicit configuration. Change the text that appears if no flights is found by using explicit configuration.
1. Create a component configuration for your component. Name the component configuration ZVCT_COMP_CONFIG_##.
   a) Create the component configuration from the context menu of your component. The configuration
2. Begin with the implicit configuration. Change the following properties.

**View UI Element Property Value**
- **INPUT_VIEW GROUP_1**
  - Design secondary-boxcolor: all labels Design highlighted
- **OUTPUT_VIEW GROUP_2**
  - Design secondary-boxcolor
  - FLIGHTTAB Design Alternating
  - FLIGHTTAB_PLANETYPE Visibility Invisible

a) Select the **Implicit Configuration** tab.
b) Open the hierarchical list to find the elements. Change the element properties.
c) Choose the **Save** button located in the group box on the left side of the screen.

3. Explicit configuration: Choose a text for the attribute SETTINGS.EMPTY_TABLE_TEXT.

a) Select the **Explicit Configuration** tab.
b) Select the SETTINGS context node. The value of the attribute EMPTY_TABLE_TEXT appears on the right side. Change the text.
c) Choose the **Save** button located in the group box on the left side of the screen.

**Task 4:**
Create an application configuration for your Web Dynpro application. Assign this application configuration to your Web Dynpro application statically.

1. Create the application configuration (name: **ZVCT_APPL_CONFIG_##**).
   a) Create the application configuration from the context menu of your application. The configuration editor for Web Dynpro opens in the browser. Enter the configuration name in the corresponding field on the upper-left side of the browser window.
   b) Click the **Create** link that appears in the **Functions** group box. A dialog opens on the right side.
   c) Enter a description in the appropriate field on the right side.
   d) Using the value help, choose the component configuration you created in the last task. This is entered in the **Configuration of Root Component** group box.
   e) Choose the **Save** button, located in the group box on the left side of the screen.
   f) Finish the configuration process by closing the browser window.

**Task 5:**
Assign the application configuration to your Web Dynpro application statically.

1. Add the name/value pair WDCONFIGURATIONID / ZNET310_APPL_CONFIG_## to the parameter list of your Web Dynpro application.
   a) Edit your Web Dynpro application. Choose the **Parameters** tab.
   b) Enter **WDCONFIGURATIONID** in the **Parameter** column and the name of your application configuration in the **Value** column.
   c) Start your application. Check the changes in respect to the design and to the text that appears if
Lesson: SAP List Viewer for ABAP Web Dynpro

Lesson Overview
This lesson discusses the integration of the SAP List Viewer (ALV) in ABAP Web Dynpro applications.

Lesson Objectives
After completing this lesson, you will be able to:
• Use the ALV for Web Dynpro to display mass data
• Explain how more sophisticated ALV functionality can be implemented

Business Example
You have created a Web Dynpro application. However, you know the old ABAP Dynpro and the SAP List Viewer were used there to display mass data. You want to learn about using the SAP List Viewer in Web Dynpro to realize a more sophisticated UI.

SAP List Viewer for Web Dynpro
The Table UI element is the standard UI element in the Web Dynpro context to display mass data. Basic functionality like selecting rows and scrolling up and down is provided out of the box. The developer can set a large number of table properties at design time to change the visual design and the functionality offered to the user at runtime. However, in the classical SAP GUI, the developer and the user can work with the SAP List Viewer, which offers a much larger functionality.

Using the ALV in ABAP Web Dynpro
The SAP List Viewer is not implemented as a standard or complex UI element, but as a Web Dynpro component. The name of this component is SALV_WD_TABLE. If the ALV is to be displayed in a view of any component, this component has to declare the usage of the ALV component. The interface view TABLE has to be embedded in a view container of the view that will display the SAP List Viewer. Finally, data have to be exchanged between the main component and the ALV component using external context mapping.
The single steps of embedding are:
• Add a component usage of `SALV_WD_TABLE` to the list of used components (`Used Components` tab).
• In the view that will embed the ALV, define a `View Container UI Element`.
• In the window containing this view, embed the ALV interface view `TABLE` in the `View Container UI Element`.
• Define an interface controller usage for the ALV component usage (from the context menu of the ALV component usage in the object tree of the workbench).
• Edit the interface controller usage:
  Æ Declare the component controller of the main component as a used controller.
  Æ Define an external mapping between the `data` node of the interface controller usage and the node of the main component controller containing the data to be displayed.

Having performed all of these steps, the basic functionality of the SAP List Viewer can be used. This includes:
• Sorting
• Filtering
• Selecting rows
• Scrolling up/down
• Exporting to Microsoft Excel
• Exporting to PDF

The `Settings` link can be used to open a dialog box, which allows you to adapt additional settings:
• Hiding columns
• Defining sorting columns and sorting direction
• Defining complex filters
• Changing table display settings
• Defining settings for PDF export
These settings can be saved and used as the user's default display variant.

**Hint:** If the Web Dynpro application is started in Customizing mode (adding `sap-config-mode=X` to the URL), the settings can be saved for all users. In addition, the settings can be added to a transport request using a Customizing task.
Advanced ALV Functionality: Configuration Model

When instantiating the ALV component usage, a configuration model is generated automatically. The configuration model offers a large number of methods that allow adaptation of the look and feel of the ALV. A reference to the configuration model can be obtained by calling the interface controller method `get_model()`, which is available for all ALV component usage instances. The model reference is of type `CL_SALV_WD_CONFIG_TABLE`. This class implements several interfaces, allowing changes in the following parts of the ALV:

**ALV Configuration Model: Interfaces Implemented by the Model Class**

**Interface Name Usage**
- `IF_SALV_WD_COLUMN_SETTINGS` Settings for the columns in the ALV output
- `IF_SALV_WD_CONTROL_SETTINGS` Controls which spreadsheet application is used with button (e.g., use Excel in Place)
- `IF_SALV_WD_EXPORT_SETTINGS` Settings for data export to Microsoft Excel
- `IF_SALV_WD_FIELD_SETTINGS` Settings for the fields of the internal data table (e.g., group aggregation)
Interface Name Usage
IF_SALV_WD_FUNCTION_SETTINGS Settings for user-defined functions (e.g., position)
IF_SALV_WD_PDF_SETTINGS Settings for data export to PDF (e.g., margins)
IF_SALV_WD_STD_FUNCTIONS Settings for generic ALV functions (e.g., hide Print Version button)
IF_SALV_WD_TABLE_HIERARCHY Settings for hierarchical ALV list display (e.g., collapse hierarchical view)
IF_SALV_WD_TABLE_SETTINGS Settings for entire ALV output (e.g., make ALV available for edit)

 Calling methods defined in these interfaces is possible from the standard hook methods wddoinit( ) and wddomodifyview( ) of a main component's controller.

Advanced ALV Functionality: Eventing
The ALV component fires a number of events to influence ALV processing. This may happen at certain points of time (for example, before rendering a table cell) or as a reaction to a user action (for example, selecting a row or clicking a button in a table cell). Methods, defined in the main component, can register to these events. For example, when the user clicks on a link or on a button defined in a table cell, the ALV component fires the event ON_CLICK. If a method of the main component has registered to this event, it will be processed after clicking. The event ON_DATA_CHECK is triggered after the user has changed the value of a table cell. Data checks can be performed in the main component.

Figure 162: Advanced SAP List Viewer Functionality
Exercise 16: Special Topics: Using the SAP List Viewer

Exercise Objectives
After completing this exercise, you will be able to:
• Embed the SAP List Viewer into a view of your component
• Use basic functionality of the SAP List Viewer

Business Example
You have created a view that contains a TABLE UI element to display mass data. The functionality of this element is restricted. Thus, you want to display the data using the SAP List Viewer, which offers more features, even in the basic version (without having to add custom code).
Template: NET310_POP_S1
Solution: NET310_ALV_S
Solution: NET310_ALV_S_OPT (optional part)

Task 1:
Copy your Web Dynpro component ZVCT_POP1_## or the template VCT_POP_S1 to Web Dynpro component ZVCT_ALV_##.
1. Copy the template.
2. Create a Web Dynpro application, activate the component, and test the application.

Task 2:
Embed the SAP List Viewer component SALV_WD_TABLE in your component. Replace the flight table on the view OUTPUT_VIEW with the interface view TABLE of the ALV component.

1. Declare the usage of the ALV component in your component.
2. Map the DATA node, defined in the interface controller of the ALV component usage, to the FLIGHTTAB node of the consumer component's component controller.
3. In the view OUTPUT_VIEW, replace the TABLE UI element for displaying flight information with a view container. Embed the interface view TABLE of the ALV component usage into this view container.

Task 3:
Test your Web Dynpro application.
1. Activate the component with all of its constituents and test the application.

Task 4:
OPTIONAL: Selecting a new flight by marking a row in the ALV flight table does not result in displaying the customer details in a dialog box. This is because marking the line in the ALV flight table has not yet been handled yet. Thus you have to create an event handler method for the event ON_LEAD_SELECT of the ALV component. From the source code of this method the action handler method ONACTIONSHOW_CUSTOMER has to called.

1. Add the interface controller of the ALV component usage to the list of used controllers for view
OUTPUT_VIEW.
2. Create an event handler method (suggested name: H_ON_LEAD_SELECT).
3. Register this method for the event ON_LEAD_SELECT of the ALV component usage.
4. Implement the event handler method. Call the action handler method ONACTIONSHOW_CUSTOMER.
Solution 16: Special Topics: Using the SAP List Viewer

Task 1:
Copy your Web Dynpro component ZVCT_POP1## or the template VCT_POP_S1 to Web Dynpro component ZVCT_ALV##.
1. Copy the template.
a) Perform this step as in previous exercises.
2. Create a Web Dynpro application, activate the component, and test the application.
a) Perform this step as in previous exercises.

Task 2:
Embed the SAP List Viewer component SALV_WD_TABLE in your component. Replace the flight table on the view OUTPUT_VIEW with the interface view TABLE of the ALV component.
1. Declare the usage of the ALV component in your component.
a) On the Used Components tab of your component, enter the name of the ALV component (SALV_WD_TABLE) in the Component column and enter the usage name in the Component Usage column (suggested name: ALV_FLIGHTS).
2. Map the DATA node, defined in the interface controller of the ALV component usage, to the FLIGHTTAB node of the consumer component's component controller.
a) In the navigation tree for your component, open the node Component Usage ALV_FLIGHTS. Double-click on the sub-node INTERFACECONTROLLER_USAGE to edit the interface controller of the ALV component usage.
b) In the object window on the right side, press the Controller Usage button and add the consumer component's component controller to the list of used controllers.
c) Drag and drop the DATA node of the interface controller usage to the FLIGHTTAB node of the consumer component's component controller. Now you have established external context mapping between the two controllers.
3. In the view OUTPUT_VIEW, replace the TABLE UI element for displaying flight information with a view container. Embed the interface view TABLE of the ALV component usage into this view container.
a) Edit the layout of view OUTPUT_VIEW. Delete the UI element FLIGHTTAB. Instead, insert a VIEWCONTAINERUIELEMENT at this position.
b) Edit the window of your component. Declare the interface controller of the ALV component usage as a used controller (Properties tab).
c) Open the Window tab. Select the view container and embed the interface view TABLE of the ALV component usage into this container. Use the context menu of the view container.

Task 3:
Test your Web Dynpro application.
1. Activate the component with all of its constituents and test the application.
a) Perform this step as in previous exercises.
Hint: All columns of the database table SFLIGHT are displayed, because the context node FLIGHTTAB, which holds the data, is typed with the dictionary structure SFLIGHT. You can use the ALV configuration option to hide columns and save the result as your default variant.

Task 4:
OPTIONAL: Selecting a new flight by marking a row in the ALV flight table does not result in displaying the customer details in a dialog box. This is because marking the line in the ALV flight table has not yet been handled yet. Thus you have to create an event handler method for the event ON_LEAD_SELECT of the ALV component. From the source code of this method the action handler method ONACTIONSHOW_CUSTOMER has to called.

1. Add the interface controller of the ALV component usage to the list of used controllers for view OUTPUT_VIEW.
2. a) Edit the view OUTPUT_VIEW. Select the Properties tab.
   b) Press the button with the label Create Controller Usage to add the interface controller of the ALV component usage to the list of used controllers for view OUTPUT_VIEW.

2. Create an event handler method (suggested name: H_ON_LEAD_SELECT).
   a) Edit the view OUTPUT_VIEW. Select the Methods tab.
   b) After having entered the method's name, choose the method type (Event Handler) before pressing Enter.
3. Register this method for the event ON_LEAD_SELECT of the ALV component usage.
   a) On the Methods tab, use the value help of the field Event to select the ALV event.
4. Implement the event handler method. Call the action handler method ONACTIONSHOW_CUSTOMER.
   a) Use the Web Dynpro Code Wizard to call the method.
   b) Parameters send by the ALV event can be passed to the action handler method by passing the parameter WDEVENT.

Result

OPTIONAL PART - Model Solution:
VCT_ALV_S_OPT, Method H_ON_LEAD_SELECT

METHOD h_on_lead_select.

wd_this->onactionshow_customer( wdevent = wdevent )

ENDMETHOD.
Lesson: Portal Integration

Portal Integration

The portal offers a single point of access to SAP and non-SAP information sources, enterprise applications, information repositories, databases, and services in and outside your organization, all integrated into a single user experience. It provides you the tools to manage and analyze this knowledge, and to share and collaborate on the basis of it. With its role-based content and personalization features, the portal enables users from employees and customers to partners and suppliers to focus exclusively on data relevant to daily decision-making processes.

Figure 163: SAP Enterprise Portal Portal Content Structure
Content can be integrated into the SAP Enterprise Portal by generating a related iView. An iView (integrated view) is a logical portal content building block representing a given visual application or a part thereof. iViews let you extend the reach of your portal to any available information resource, regardless of where it may be stored. A portal page holds iViews and other portal pages organized in a layout. This allows you to combine multiple iViews in rows and/or columns. Each iView is placed on a tray, so it can be opened or closed when displaying the page in the portal.

Worksets let you bundle iViews and pages in folder hierarchies, for example, for roles. Worksets represent generic, reusable structures or modules that can be added to roles. A workset may be used in any number of roles, and a role may consist of a number of different worksets. Roles are the largest semantic unit within the content objects. A role is a folder hierarchy comprising other content objects (worksets, pages, or iViews).

Roles are assigned to users. This means that users can only access the content that is relevant for them if they have the appropriate role.

Integrating Web Dynpro Applications into the SAP Enterprise Portal
Web Dynpro applications can be integrated into the SAP Enterprise portal; that is, they can be bound into portal navigation as an iView. In addition, it is possible to address portal functions from within a Web Dynpro application. For this purpose, you can access portal manager methods (interface IF_WD_PORTAL_INTEGRATION) as source code templates for the different functions by calling up the Web Dynpro Code Wizard. This includes:

- Using portal events
- Navigation between Web Dynpro applications within the portal or to any other portal content using:
  - Object-based navigation
  - Absolute navigation
  - Relative navigation
The main steps in the process of generating an iView for a Web Dynpro application are:

• Defining the system in the SAP Enterprise Portal system landscape
• Defining the mapping between portal users and users in the previously defined ABAP system
• Creating iViews for Web Dynpro applications in the ABAP system
• Combining different iViews that have to interact via the portal using a portal page
• Adding the portal page to a portal role to make it usable by user groups to which this role is assigned

**Portal Eventing**

If one Web Dynpro application needs to interact with another Web Dynpro application, it can declare the usage of the other component. The other component's interface is then visible and usable by the consumer component. This type of coupling is called **tight coupling**. In the SAP Enterprise Portal, you can process different application types in special iViews on the same portal page. Here, iViews can be included using different technologies (such as Web Dynpro or Business Server Pages). The communication between these iViews takes place through an event function called **portal eventing** (or client-side eventing).

A Web Dynpro application can register for portal events. In this way, the Web Dynpro application can react to an event that was triggered in another iView in the portal. Therefore, it does not matter what technique you used to set up the application that is the basis for the other iView. The assignment of which event handler is to be called when an event occurs is stored in the Web Dynpro application that has registered itself on the portal event. Similar to registration, a Web Dynpro application can trigger any portal event. In this case, the event is passed to the portal by the
respective iView. The portal passes the event to all iViews that have registered for this event. The application that finally handles this event can, in turn, have been set up with a different technique than the Web Dynpro application triggering it.

**Caution:** Portal eventing functions only between iViews that are on the same browser window. Events between iViews in different browser windows cannot be transported. All participating iViews must also belong to the same domain; otherwise, portal eventing cannot work due to JavaScript restrictions.

![Portal Eventing](image)

**Figure 166: Portal Eventing**

**Raising a Portal Event from an ABAP Web Dynpro Application**

The source code for firing a portal event can be generated using the Web Dynpro Code Wizard. Portal events have to be fired from a controller method of a view. Before a portal event can be fired, the reference to the portal manager has to be determined. Three parameters must be exported when firing an event:

- **portal_event_namespace**
  In each event namespace, each event name is unique. The event namespace is an arbitrary string.

- **portal_event_name**
  Name of the fired event (arbitrary string). The combination of event namespace and event name identifies an event uniquely.

- **portal_event_parameter**
  This string contains all data that will be exchanged between the application firing the event and the application that registered for the event.
* Register for Portal Event

```plaintext
* WDDOINIT

DATA lo_api_component TYPE REF TO if_wd_component.
DATA lo_portal_manager TYPE REF TO if_wd_portal_integration.
DATA lo_api_controller TYPE REF TO if_wd_view_controller.

lo_api_component = wd_comp_controller->wd_get_api( ).
lo_portal_manager = lo_api_component->get_portal_manager( ).
lo_api_controller = wd this->wd_get_api( ).

lo_portal_manager->subscribe_event(
    portal_event_namespace = 'SAP',
    portal_event_name = 'SHOW_BOOKINGS',
    view = lo_api_controller,
    action = 'GET_KEY_FIELDS').
```

Name of action. Related method is used to handle portal event

---

Figure 167: Firing a Portal Event

---

* Fire Portal Event

```plaintext
DATA lo_api_component TYPE REF TO if_wd_component.
DATA lo_portal_manager TYPE REF TO if_wd_portal_integration.
DATA lv_parameter_string TYPE string.

lo_api_component = wd_comp_controller->wd_get_api( ).
lo_portal_manager = lo_api_component->get_portal_manager( ).

lo_portal_manager->fire(
    portal_event_namespace = 'SAP',
    portal_event_name = 'SHOW_BOOKINGS',
    portal_event_parameter = lv_parameter_string).
```

---

Figure 168: Registering for a Portal Event
Thus, to be able to interact via portal eventing, the application firing the event and the application registering for the event have to use the same names for the event name space and for the event name. In addition, the method of extracting the information from the event parameter must be known to the application that registered for the portal event.

Registering for a Portal Event in an ABAP Web Dynpro Application

The source code for registering for a portal event can also be generated using the Web Dynpro Code Wizard. Only a view can register with an action handler method for a portal event. Registering for the portal event is to be implemented in the standard hook method `wddoinit()` of the view. The name of the action whose related method will handle the portal event has to be exported.

In the event handler method, information about the portal event can be extracted from the interface parameter `WD_EVENT`. The attribute `WDEVENT->PARAMETERS` is an internal table containing name/value pairs in each row. In the portal event context three lines are filled containing the event namespace, the event name, and the event parameter. Extracting the event parameter can be implemented by reading the related table line using the `keyname = 'PORTAL_EVENT_PARAMETER'`. Since the `value` field of the corresponding table line is of type `ref to data`, dereferencing the value using a field symbol (of type `string`) is necessary.

Figure 169: Handling the Portal Event

```
* Handle Portal Event:
* Extract parameters

DATA ls_params TYPE wdr_event_parameter.

FIELD-SYMBOLS <lv_params> TYPE string.

READ TABLE wdevent->parameters INTO ls_params
  WITH KEY name = 'PORTAL_EVENT_PARAMETER'.
ASSIGN ls_params-value-* TO <lv_params> CASTING.

SPLIT <lv_params> AT ....
```
Lesson: Integrating Adobe Forms

Lesson Overview
This lesson discusses the integration of Adobe forms into ABAP Web Dynpro application views.

Lesson Objectives
After completing this lesson, you will be able to:
• Embed Adobe form documents into Web Dynpro views

Business Example
You have created a Web Dynpro application. To allow printing the UI content displayed by the Web Dynpro clients, you want to integrate Adobe forms in your application.

Adobe Interactive Forms
As of SAP NetWeaver 2004 (in SAP Web Application Server), you can use a new solution to create interactive forms and print forms for the optimization of your form-based business processes. This solution uses Portable Document Format (PDF) and software from Adobe Systems Inc. that has been integrated into the SAP environment.

Figure 170: Adobe Form Integration
Interactive Form Scenario
For collaborative business processes, you can design interactive forms containing dropdown menus, buttons, text fields, and other elements that allow user entries. The form is rendered in PDF format and you provide it to users in, for example, a browser. Users fill in the form and save their entries inside the form in XML format. The SAP system extracts the data saved in the form and stores it in the database for further processing. In general, you can use such forms in two different scenarios:

• Online: The user is logged on to an SAP system while filling in the form.
• Offline: The user is not logged on to an SAP system while filling in the form. After completing the form, the user sends the form back to the form provider (for example by e-mail). The provider’s SAP system then extracts the data from the form.

Form Processing Scenario
You can create PDF-based forms for printing, sending by e-mail, or faxing by merging SAP system data with a form template. You can rely on the proven principle of separation of data collection and form layout, which provides the required flexibility in the case of changes to one or the other.

Online Interactive Form Scenario: Example
You can use Adobe interactive forms in scenarios where an employee typically needs to fill in a form to start or advance a related business process. Let’s take the example of an assembly line worker. The worker needs to order new materials to keep the work going.

In a traditional setup, he would fill in a paper form, sign it, and send it by in-house mail. In the case of an interactive form, the worker can log on to the internal company portal and call up the required form, which is displayed in PDF format in the browser.

The interactive form looks like the old paper-based form. During the time that the form is displayed, the worker is logged on to the system, where the required information will be processed further. When the form is displayed, it already contains all the relevant information on the worker, such as his name, the location and cost center to which he is assigned, and so on. This information is determined based on the user logon, and is automatically filled in by the system. Next, the worker fills in the required information on the form. When finished, he submits the form back to the system using, for example, a button. The data is written to the system, and the corresponding workflow moves the process to the next step. The worker also has the ability to print out the interactive form locally.
Offline Interactive Form Scenario: Example

While the last example was for an online scenario in which the user maintains a system connection, Adobe interactive forms offer new opportunities through offline usage as well. In this scenario, a company running a marketing campaign from its mySAP CRM system determines that certain data from a customer is missing. The company wants to obtain the data through a customer visit. Triggered by the CRM system, the existing relevant customer data is prefilled in the corresponding form, which also contains fields for entering the missing data. The form is automatically e-mailed to the responsible sales representative.

The sales representative travels to the customer and, together with the customer contact, fills in the form, which he has received by e-mail and downloaded to his local hard drive (or PDA). When finished, he prints the form for the customer’s records, which is facilitated by the PDF format.

The sales representative then forwards the completed form to the SAP system. He can do so by attaching it to an e-mail or by uploading it to the corresponding site in the internal company portal upon his return to the office. The mySAP CRM system receives the data entered by the sales representative, processes it, and automatically triggers the next step in the business process.
Adobe Forms: What Do They Look Like?
PDF forms look like your original paper form, and thus provide the familiar look and feel users need to execute business processes quickly. Visual integrity and fidelity of a form increases user acceptance. Special functions of Adobe Reader allow users, among other features, to save PDF forms locally, to distribute them via e-mail or an enterprise portal, to print them out, or (from SAP NetWeaver 7.0) to sign them digitally before submitting them to the system.

All user entries made in a PDF form are stored in XML format and can, thus, easily be transferred back into the SAP system. The integration of Adobe technology into the SAP development environment also allows for prefilling form fields with current system information, including context-sensitive list boxes (value help) comparable to SAP's F4 help.

Upon form creation, the necessary data is automatically extracted from the back end system. When the form is returned to the system by the user, the XML-based data is automatically returned to the system.
Adobe Forms in ABAP Web Dynpro

PDF forms can be created and used for Web Dynpro user interfaces. The following scenarios are available for Web Dynpro ABAP:

- Print, offline, and display scenario: Here, unlike in the interactive scenario, it is not only interactive PDF forms that are used.
- Interactive scenario: For this scenario, interactive PDF forms are used.

The procedure for creating or using the relevant forms is largely the same for all scenarios. For Adobe integration, the Adobe Library with the Interactive Form UI element is provided in the Web Dynpro View Designer.
Forms can be created and maintained independently of Web Dynpro applications using the Form Builder (transaction SFP). You can integrate a form within every view of any component. To include an existing form into a view, proceed as follows:

- In the Web Dynpro Explorer, create a view for your component or select a view into which to integrate the form.
- In the layout of this view, add the UI element `InteractiveForm` to the UI element hierarchy.
- For the `templateSource` property, enter the name of the selected form (an input help is available). Based on the interface of the selected form, a context node with attributes is automatically created for the UI element `InteractiveForm`. The `dataSource` property of the UI element is automatically bound to this context node.
- To define that your form is an interactive form, select the checkbox for the `enabled` property of the `InteractiveForm` UI element. By default, `enabled` is inactive; that is, it is usually a non-interactive form.
- Fill the created context structure, for example, by using a suitable supply function.
- Activate and test the application.

**Hint:** You must only mark the checkbox for the `enabled` property if your application is to be used interactively. Selecting the checkbox means that the Active Components Framework (ACF) is used to render in the client (generally the browser of the user) and must therefore be installed. The ACF is installed centrally from SAP NetWeaver, usually automatically when a relevant application is first called. However, it is possible that a user may not have sufficient authorization for the local installation of the ACF-CAB files. In this case, the user cannot display the form. If a form is used without checking the `enabled` checkbox, that is, if it is not interactive, the ACF is not required for rendering.

![Diagram](image.png)

**Figure 174: Adobe Forms in ABAP Web Dynpro**
Appendix 1

Phase Model

The following figure schematically shows the process flow during data transport.

Figure 175: Phase Model: Processing views
The process flow of event handling is shown in detail in the following figure.

Figure 176: Phase Model: Event Handling in Detail
Appendix 2

Dynamic Programming - Advanced Topics

Parameter Mapping
All action handler methods have the interface parameter wdevent, which is filled by the Web Dynpro runtime if a client-side event is fired. Information about the UI element that fired the event can be extracted from the action handler’s interface parameter wdevent parameters. If the information available from the interface parameter wdevent is not sufficient, additional information can be passed from the UI element to the action handler method. To do this, a list of name/value pairs can be mapped to any event of the UI element. This must be performed in the hook method wddomodifyview().

The parameter list is mapped to an event of the UI element object by using the appropriate mapping method. For each client-side event, a different parameter list can be added. The mapping methods follow the naming convention map_on_<event>.

Figure 177: Parameter Mapping
If an event of the UI element is fired, the related parameter list is sent back to the server. The name/value pairs are attached to the internal table `wdevent parameters` by the Web Dynpro runtime. However, the value column of this internal table is of the generic data reference type `DATA`. Thus, before this value of a certain event parameter can be accessed, it has to be casted to a variable of the correct type. This is typically done by de-referencing the reference variable into a field symbol using the addition `casting type <type_name>` . However, the Web Dynpro runtime types the reference variables with dynamically generated types that are not compatible to the statically available types defined in the class builder. Thus a different way of de-referencing the variable value into a field symbol has to be used:

At runtime, the type of a variable can be described by an object using RTTI (run time type information). On the other side, a RTTI type description can be used to type variables using the addition `type handle to` . This addition can also be used when de-referencing a reference variable into a generically typed field symbol (`castingtypehandleto`). Thus, accessing the event parameters involves the following steps:

- First the dynamically created reference type of an event parameter is obtained by using RTTI (e.g. the method `describe_by_data_ref` of class `CL_ABAP_TYPEDESCR`).
- Next, the parameter value is de-referenced in a field symbol using the addition `casting type handle to <RTTI_type object>`.
- If the static type of this parameter is known, the field symbol can finally be casted to a reference variable of this static type.
**DATA** ls_parameters TYPE wdr_event_parameter.
DATA lv_id TYPE STRING.
DATA lo_object TYPE REF TO cl_wd_usielement.
DATA lo_rttssref TYPE REF TO cl_abap_refdescr.
FIELD-SYMBOLS <lu_n> TYPE ANY.
FIELD-SYMBOLS <lu_obj> TYPE ANY.

* get value of data reference N from parameter WDEVENT
  READ TABLE wdevent->parameters INTO ls_parameters
  WITH KEY name = 'N'.
  ASSIGN ls_parameters-value->* TO <lu_n>.

* get reference to object reference OBJECT from parameter WDEVENT
  READ TABLE wdevent->parameters INTO ls_parameters
  WITH KEY name = 'OBJECT'.
  lo_rttssref ?= cl_abap_typedescr->describe_by_data_ref(
    p_data_ref = ls_parameters-value).
  ASSIGN ls_parameters-value->* TO <lu_obj>
  CASTING TYPE HANDLE lo_rttssref.
  lo_object ?= <lu_obj>.

**Figure 179: Action Handler Method: Accessing the Mapped Parameters**

**Hint:** Since the UI element ID is unique in respect to the UI element hierarchy of one view, it is sufficient to use the element's ID to identify the element. Thus, parameter mapping is rarely used in ABAP Web Dynpro programming.
Appendix 3

Value Help - Advanced Topics

Object Value Selector (OVS)
There are situations when reusing existing ABAP dictionary search helps is not sufficient. Say, for example, the value help should take into account or fill multiple input fields of your view composition. However, the different input fields are related to different value nodes, which in turn refer to different ABAP Dictionary structures. ABAP Dictionary search helps are not able to fulfill this task, since the maximum scope of such a search help is the structure to which it is related.

To overcome this restriction, complex value help can be implemented as part of the Web Dynpro component. This kind of value help is referred to as the Object Value Selector (OVS).

Figure 180: Object Value Selector (OVS)
A reusable Web Dynpro component is responsible for displaying the selection screen, allowing you to enter restrictions for data retrieval and for displaying the values fitting these restrictions. The consumer of this component (the parent component) has to care for specific tasks such as defining the selection screen setup and data retrieval. Between the two components, data has to be interchanged. Thus, the OVS component has to offer an interface that can:

- Receive information about which input fields should be displayed on the selection screen,
- Receive information about the initial values of these input fields
- Hand back the user’s input into these input fields
- Receive the result from the data retrieval in order to display the list
- Hand back the selected data set to the consumer program

**Phase Model of the Object Value Selector**

After having pressed the value help button of an input field in the consumer component, the OVS component is processed. However, the OVS component needs additional information from the consumer component at runtime (for example, should an additional selection screen be displayed, which fields should be displayed on this selection screen, which information should be displayed as the value help, and so on). Thus, at certain points in time, control has to be given back to the consumer component. This is realized by using standard event handling technique.

Every time the OVS component requires information from the consumer component, the OVS event is fired by the OVS component. An additional parameter, `phase_indicator`, indicates the type of information that is requested. An event handler method of the consumer component has to subscribe for the OVS event. Depending on the value of the `phase_indicator` parameter, different coding sections can be implemented to collect the required information and hand it back to the OVS component.

**Note:** For some of the phases, handing back information to the OVS component is optional.

In order to hand back the information to the OVS component, appropriate methods of this component have to be called. A reference to the object ( `ovs_callback_object` ) containing these methods is also provided by the OVS event.
Phase 0 (phase_indicator = 0): This phase is also called the configuration phase. Here, the consumer can define which texts are to be displayed by the OVS component, either on the selection screen dialog box or on the value help list. In addition, the number of columns used to structure the selection dialog box can be defined.

To export the information to the OVS component, the method set_configuration( ) of the callback object should be called.

Hint: Calling the method set_configuration( ) in phase 0 may be omitted if no selection dialog box is to be displayed by the OVS component.

Hint: Label texts not provided by the consumer component are obtained from the ABAP Dictionary if the related input fields are typed accordingly.

Phase 1 (phase_indicator = 1): In this phase, the consumer can define which input fields are to be displayed on the selection dialog box. This is done by exporting an arbitrary structure containing these fields. Default values for the selection fields can be provided by setting the structure fields to the corresponding value. To export this information back to the OVS component, the method set_input_structure( ) of the callback object should be called. Then the selection screen can be rendered.

Hint: Calling the method set_input_structure( ) in phase 1 may be omitted. In this case the selection screen will not be displayed. This is useful if the user has already entered values in the input fields of the consumer component, so displaying the same values again on the OVS selection dialog box before displaying the value help is not needed.

Phase 2 (phase_indicator = 2): This phase is processed after the user has pressed the button on the OVS selection dialog box (if processed). In this phase, the consumer has to collect the data that
ABAP Web Dynpro

will then be displayed as the value help list by the OVS component. However, to collect this information, the consumer component needs to know what the user has entered into the fields of the selection dialog box. The reference (data reference) to this information is available from the query_parameters parameter of the callback object.

To export the value list to the OVS component, the method set_output_table() of the callback object has to be called. The value list will be displayed.

- The OVS event has four phases, and each is called once
- The different phases can be recognized with the attribute OVS_CALLBACK_OBJECT > PHASE_INDICATOR

![Figure 182: Object Value Selector: Distinguish Between Phases](image)

**Phase 3 (phase_indicator = 3):** After having selected a value from the value list, this selection has to be transported back to the consumer component. Thus, the OVS event is fired a fourth time. The reference to the user selection is available from the data reference variable selection which is a parameter of the callback object. In the related coding section of the consumer component's event handler method, these values have to be written back to the context. The selected data is displayed as input field values of the consumer component's view.

**Cookbook for Using the OVS**

The following steps must be performed in order to use the OVS concept:

- A component usage of the component WDR_OVS has to be declared by the consumer component.
- A usage of the OVS interface controller has to be declared in a controller of the consumer component for example, the component controller or the displaying view controller).
- The input help mode Object Value Selector has to be chosen for the attribute under consideration.
- A handler method for the event OVS of the OVS component usage has to be created and implemented.
User-Defined Value Help

In addition to the value help technique discussed above, a developer can define a completely user-defined value help. Technically, this kind of value help is implemented as a Web Dynpro component implementing the Web Dynpro component interface `IWD_VALUE_HELP`. To be able to use the value help for a certain input field, the following steps are necessary:

- A component usage of the value help component (HC) has to be declared by the consumer component (CC).
- A usage of the HC interface controller has to be declared in the CC view.
- The input help mode `User-Defined Programming` has to be chosen for the attribute that is bound to the input field under consideration. The HC usage must be related to this attribute.

The component interface of the HC has only one method: `set_value_help_listener()`. This method is called by the Web Dynpro runtime if the value help button of the input field under consideration is clicked. The HC has to be implemented as follows:

- The method `set_value_help_listener()` has an import parameter. This means that the reference to the listener, provided by the Web Dynpro runtime, is passed to the user-defined HC. This reference has to be saved as a user defined controller attribute.
- To close the help value dialog box, the `close_window()` method of the listener has to be used.
- All views have to be embedded into a window having the name `WD_VALUE_HELP`. This name is used by the Web Dynpro runtime.
- To exchange data between the CC and the HC, context mapping can be used.

**Hint:** For implementing the component interface `IWD_VALUE_HELP`, you need to choose the `Reimplement` button in the HC. The implementation status changes to green and the events and the method of the component interface are visible in the HC controller.
Appendix 4

Adaptation of Web Dynpro Applications - Advanced Topics

Explicit Customizing and Personalization

If the functionality offered by implicit customizing and implicit personalization is not sufficient, the developer can extend the functionality nearly arbitrarily. However, here the developer has to write the complete source code to bring up and process the adaptation dialog. This includes:

• Defining a UI element, which is used to start the adaptation process
• Defining the source code in the related event handler method to bring up the adaptation dialog
• Defining the dialog, which consists of the standard implicit adaptation functionality (optional) and the additional explicit adaptation functionality
• Defining the logic to close the adaptation dialog (typically via Save and Cancel buttons).
• Defining the functional changes in the component related to the user’s input in the adaptation dialog

Figure 183: Combining implicit and explicit Customizing / Personalization
To combine the functionality of implicit and explicit customizing / personalization in one dialog, the component interface `IWD_PERSONALIZATION` has to be declared as a used component (interface) by the component offering the adaptation functionality. This interface offers an interface view (`imp_personalization`) that can be embedded in a view container of the main component in order to realize a layout combining implicit and explicit customizing / personalization. Typically, a view containing a `TabStrip` with two tabs is used to define the layout. One tab contains all fields that are used for explicit customizing / personalization. The other tab contains only a `ViewContainerUIElement`. The view is embedded in an extra window, which is necessary to display the windows content as a dialog box. The interface view `imp_personalization` is then embedded in the `ViewContainerUIElement`.

In addition, the interface offers the method `init_personalization()`, allowing hand over of a reference to the UI element that is adaptable implicitly (with all of its sub-elements; this could also be the `UIElementRootContainer`).

**Hint:** All controllers that need to access the interface of the personalization component implementing `IWD_PERSONALIZATION` have to declare the usage of this interface. Typically this is the view containing the link to start customizing / personalization and the view embedding the interface view `imp_personalization`.
Portal Integration

Creating a New System
To access the Web Application Server (Web AS) from the SAP Enterprise Portal, a new system has to be defined. This is done by choosing System Administration system Configuration system Landscape. On the Browse tab, you may first create a new folder to group all the systems. Choose New Folder from the context menu that appears when you click on the main Portal Content node or any child node. Having selected a folder, a new system is created from the context menu by choosing New System (from template). In the wizard, choose SAP system using... . There are three entries allowing you to log on to an application server, to use load balancing, and to use an SAP router string. On the next screen, enter an arbitrary System Name and System ID. Choose Finish. On the next screen, choose Open the object for editing.
Choose **Object** from the **Display** dropdown box. Then choose **Web Application Server (Web AS)** as the **Property Category**. Enter the System ID, the URL (including the port), the path to the Web Dynpro applications (default: sap/bc/webdynpro), and the protocol **http**. Change the **Property Category** to **User Management**. On the related screen, select **Logon Method = UIDPW** and **User Mapping Type = admin,user**. This will allow everyone to log on, filling user and password in a logon dialog box that appears when starting the application from the portal. In the next step, an alias has to be created for the system. Choose **Display = System Alias**. Enter an alias and press the **Add** button. Choose **Save**.

![Figure 185: Portal: Creating a New System (2)](image)

**User Mapping**

At least for testing purposes, it is easiest to map the portal user to a user in the back-end system. Choose **User Administration > Identity Management**. Enter your portal user name in the input field and choose **Go**. Your user appears in the table below.

Mark the related table line. Now, below the table listing your user the details of your user appear. Tabs allow to select details of your user. Select the right most tab having the label **User Mapping**. Select the alias of your system from the drop down box **System**. Press **Modify**. Enter your user and password in the appropriate fields. Choose **Save**.
Creating iViews for ABAP Web Dynpro Applications

Now, that the technical system and the user mapping are defined, iViews can be created for the Web Dynpro applications.
1. To start the process, choose Content Administration   Portal Content . On the right side of the browser window, select the Browse tab. Open the Portal Content node and the sub-node you want to use to group the content you develop (to create a new folder, select New Folder from the context menu that appears when you click on the Portal Content node or any child node).

Select New iView from the context of the selected node. On the right side, select the type of iView wizard you want to use.

2. Select the iView template radio button. This opens a special wizard offering you a large set of predefined iView patterns.

3. On the first screen of this wizard (Step 1), select SAP Web Dynpro iView.

4. Next (Step 2), enter a name and an ID for the iView.

1. On the third screen, ABAP has to be chosen as the Web Dynpro Definition Type.

2. Finally, on screen four, application parameters have to be provided as follows: The system alias has to be entered in the System field. SAP is the default Namespace. This value will be added to the path to the WebDynpro application. The Web Dynpro application name has to be entered in the Application Name field. Optionally, the name of an application configuration and/or application parameters can be added (for example, sap-config-mode=X).

3. After having finished the iView generation, the result can be checked by choosing Preview from the context menu of the iView.
Combining Views in a Page

Multiple iViews can be arranged on one visible page using a portal page object.

1. To create a page and its content, first choose Content Administration Portal Content. On the Browse tab, localize the node that should be used as the page's parent node. Create a portal page from the context of the node (choose New Node).
2. On the following screen, enter name and ID of the page.
4. Next, the page layout has to be defined. To display two iViews, one below the other, choose 1 Column (Full Width). Choose Add, then Next.
5. To adjust the properties of the page and to assign the iViews, select Open the object for editing on the next screen. Choose OK.
1. On the right side, a list of the iViews that are already embedded in the portal page appears. To add another iView to the page, choose Add iView to Page Delta Link from the context menu of the iView to be embedded.
2. By switching from Page Content to Page Layout, the resulting structure of the page can be analyzed.

The last step in creating the page is to adapt the page’s properties. There are properties related to the page object and there are properties related to the embedded iViews.

1. An important group of properties is the Property Category Appearance Size. For the portal page, the Height Type should be adjusted to Full Page. For the single iViews, this property should be adjusted to Automatic so the iView’s height depends on the space the Web Dynpro application occupies.
2. You can test the result by choosing Preview, which can be found on top of the tab page.