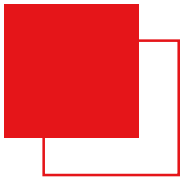


# Abap Web Dynpro

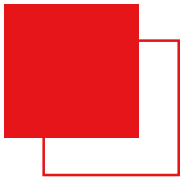
## Recommendations and best practices



# Abap Web Dynpro

Programming notes for AWD applications

- This document contains useful notes on programming Web Dynpro ABAP applications and optimizing performance. These notes are intended primarily for application developers who already have a sound knowledge of the Web Dynpro Framework. You can find the complete developer documentation about Web Dynpro ABAP under [Abap Web Dynpro Abap: Development in detail.](#)



# Special Features of Web Dynpro ABAP Programming

- There are some differences in programming Web Dynpro ABAP components to standard ABAP programming.
  - You cannot use lists, nor dynpro and control technology in Web Dynpro ABAP programming. For example,
    - CALL / LEAVE TO SCREEN
    - LEAVE TO LIST-PROCESSING
    - WRITE / ULINE / HIDE
    - MESSAGE
  - The program flow cannot be changed in Web Dynpro ABAP programming. You cannot use statements to exit the current session or to start a new one. For example:
    - CALL / LEAVE TO TRANSACTION
    - SUBMIT
    - LEAVE PROGRAM
  - You cannot use certain system commands in Web Dynpro ABAP programming. For example:
    - EDITOR-CALL
    - SYNTAX-CHECK/GENERATE
  - Functional enhancements of Web Dynpro components, such as methods and event handlers are called by Web Dynpro Framework. This fills all parameters. There is therefore no need to query the parameter interface.
    - IS SUPPLIED
    - IS REQUESTED
  - The integration of type pools and program includes is not supported in Web Dynpro ABAP programming.



# Number, Size, and Genericness of Components

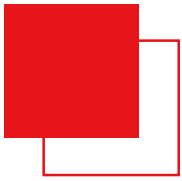
## The Ideal Design of a Component

- Communication between components requires work and makes the performance of the application suffer. Despite this, you should not make each component too big. You need to balance these requirements, while making sure that each component is a self-contained logical unit. You must always prevent different development groups from working on the same component. We have put together the following guideline for the optimum number of views in a component:



- Whenever possible, restrict your components to **a maximum of 15 views**.

- Make sure that the number of *controllers used* by each controller does not exceed 8. Remember, at runtime you create a load from your components. The size of this load depends to a great extent on the number of views, UI elements, controllers, controller usages, and the size of the context nodes. If this load is too big, the system produces a warning telling you that limited resources are available at runtime for the WDA application. Performance drops.



# Number, Size, and Genericness of Components

## View or Component

- A component is a reusable unit in the Web Dynpro Framework. A view can be displayed only once within a component; a component, however, can be instantiated multiple times. This means that, if you want a view to appear more than once in a window, you must make it a separate component and reuse it multiple times in a different component.



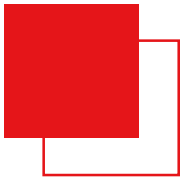
- A component is a visual unit. For this reason, we recommend that you embed only one interface view on the window of the "using" component for each declared component usage.
- Model component:
    - We no longer recommend that you create special model components, since they do not offer any benefits over model classes. One alternative is to use a shared assistance class to provide a model



# Number, Size, and Genericness of Components

## The Genericness of Web Dynpro Components

- Generic components are more difficult to maintain than explicitly programmed components, which is why we recommend that you make all your components only as generic as they absolutely need to be. This requirement needs to be balanced with the demands of distributed application development groups. These groups need to provide generic components that are then, for example, given a uniform layout at a later time or by a different group. As a guideline, we recommend that you make your applications generic to the extent that, when the resulting component is configured, around 80% of all applications are covered. Do not try to make your component more generic just to cover the remaining 20% of the possibilities.



# Context

- A context in Web Dynpro ABAP is a complex construct that gives you great freedom when you design your applications. The way an application handles quantities of data is usually of great significance for its performance; as a developer, you have an opportunity here to control the time needed by your application, but there are also risks to consider. Today, performance optimization is important for many applications, particularly more complex applications, and must be considered in the design of every new application or application group. The following sections offer you a range of useful information about how best to use various context properties and context features.
- If you want to use an element in multiple contexts, but intend to modify it only rarely if at all, it is best to create a copy of the element in the appropriate context. You must, however, ensure that the data always remains consistent. Another option is to store the content of the context node as an internal table initially, and to store it as an actual context node only within the relevant view.
- Define a mapping to context nodes of used components only in exceptional cases. This can cause a drop in performance, and any changes within the used component can cause errors in the main component.
- An alternative to this is the shared usage of an assistance class instance. For more information about this, read the description of the example application [DEMO\\_COMMON\\_ASSISTANCE1](#) and examine its implementation in your system in package SWDP\_DEMO.





# Context

## Handling Context Mappings

- A context mapping is a mechanism that enables you to make static connections between different contexts in different controllers. If, for example, you require a set of data in exactly the same structure but in two different views, it is a good idea to configure a suitable context node in the component controller. In a single action, you can then create a copy of the component controller node for each view controller in the Workbench's View Editor; at the same time, a mapping to the original node is created. This procedure is both easy to implement and very secure. It does, however, encourage you to start designing contexts in a component by designing the component controller context first, and then only defining the required mappings for the view contexts. Depending on the size of the context node (number and structure of the subnodes, and the absolute size of the data), this can have a significant effect on the performance of the application. To avoid using superfluous mappings in your component, consider carefully which context nodes you actually require in multiple contexts, and which nodes you only require locally.
- A context node is not filled until the corresponding context is called for the first time. Unlike a context node mapped to a central node, a context node created locally is empty, until the corresponding view is accessed. To avoid errors in dialogs, it is a good idea to provide the data for the context of the follow-on view before the actual navigation step is triggered. Since an instance of the follow-on view has not yet been created, you must now decide whether to configure a component controller context with an appropriate mapping, or whether to fetch and check the data in an auxiliary class first and only pass it to the context of the follow-on view after navigation is completed. An application developer would, of course, prefer to use the context mapping method; using an auxiliary class, however, improves performance significantly.



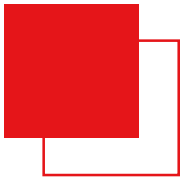




# Context

## Dynamically Created Attributes or Dynamically Created Nodes

- Context nodes and context attributes can be created dynamically in the Web Dynpro ABAP framework. The dynamic creation of these features is relevant from a performance perspective, but is not particularly significant if only one attribute is being created. If, however, you want to create multiple attributes for a context node dynamically and in parallel, it is a much better idea to first create a structure. The whole of the new node is then created in a single step. For an example of the dynamic creation of a node from a structure, refer to the Web Dynpro application DEMODYNAMIC in the package SWDP\_DEMO in your system. The method WDDOINIT in the view DYNAMIC\_NODE\_TYPE creates a structure first and then the required node information is created from this structure.



# Context

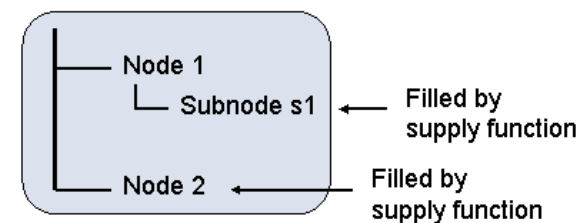
## Singleton Node

- When you create a context node below the root node, the "Singleton" checkbox is selected by default. If you have deep data structures with large amounts of data, we recommend that you consider only loading the data for the selected element of the main node into the memory. The other factor that you need to consider, however, is the performance loss connected to the required invalidation and refilling of the corresponding context node or nodes. You need to decide whether to minimize the memory load created by large amounts of data or minimize the runtime caused by invalidation and filling actions. Use singleton nodes only if the data in the memory will reach a critical amount if you take no action. If not, you may prefer to work with non-singleton nodes and delete any data that you do not yet need, or will never need, from the memory explicitly.

# Context

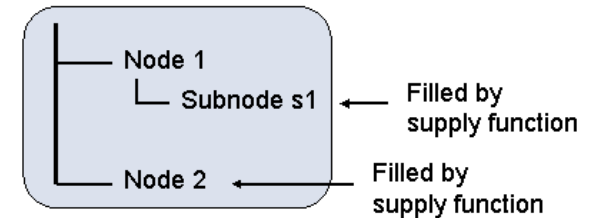
## Filling the Context: Supply function

- The "Supply" function lets you supply a node with node only when required. The supply function is not called until the content of the node is actually needed at runtime (and only then). Since runtime controls when the supply function is called (and not the code created by the application developer), it is especially important that no errors occur when the data is retrieved. For this reason, use the supply function only if you know that the data is available and correct in the back end.
- Supply functions are especially suited for filling subnodes, whether they are singleton nodes or non-singleton nodes. However, you must note the following: The required content of a subnode can be dependent only on one element of the corresponding superordinate node. This is the only way of ensuring that instances of all required nodes are created and that the values exist. The following figure illustrates the interactions:

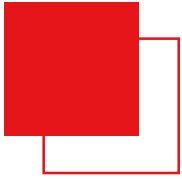


# Context

## Possible Dependencies of Supply Functions



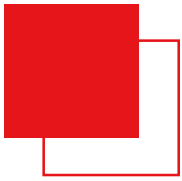
- The subnode s1 of the node node1 can always be filled by a supply function if the required data only depends on the value of an element of the node node1. In this case, there is an implicit guarantee that this value already exists, because an instance of the node node1 already exists. It can, however, be the case that the data selected for s1 depends on the value of an element of the node node2. However, there may not yet be an instance of the node node2 available when s1 is filled. If the node node2 is also filled by a supply function, it may occur that the supply function of s1 must itself first call the supply function of node node2. A cascading call of two or more supply functions is invalid and causes a runtime error. Therefore, if you need to access the data of a superordinate node other than the direct superordinate node to fill a context node, do not use the supply function; instead, program the required data retrieval function in an appropriate event handler method. If this is really not possible, at least ensure that any runtime errors are caught and handled correctly.
- If you want to trigger the call of a supply function explicitly before the content of the node is used for the first time, you can use the method `get_element()`.
- A supply function always recognizes the parameters `PARENT_ELEMENT` and `NODE` automatically. The parameter `PARENT_ELEMENT` is a reference to the element of the superordinate node (in the example above, an element of the node node1), whose value must be read to retrieve the data; the parameter `NODE`, however, is a reference to the node that is filled by the supply function (s1 in our example). You must use these two parameters. It is much more difficult to use a generic method involving generic context methods. There are no real benefits and it is also more prone to errors.
- Make sure that you never modify the `CONTEXT_NODE_INFO` of a context node within a supply function. A supply function must also never modify the `ValueSet` of a node. This causes complications since the order of the steps performed at runtime may no longer match the values.



# Context

## Context Change Log

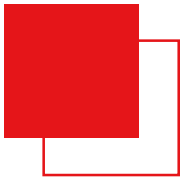
- Use the Context Change Log functions to detect user input. This has particular performance benefits while a user of the application modifies only a small amount of data in a view while displaying a large amount of mixed data.



# Context

## Properties of Context Attributes

- Each context attribute has four predefined properties: readOnly, visible, state and enabled. The values of these properties can be set with the interface `IF_WD_CONTEXT_ATTR`. The Web Dynpro Framework allows UI element properties with the same name to be bound to the respective attribute property, which means the number of the nodes can be significantly reduced in the context. This improves performance and reduces memory requirement. Refer to the Programming Manual in the document Properties of Context Attributes for more information.

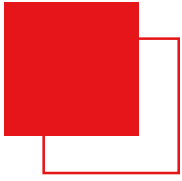


# Controller

## Handling Controller Methods

- As well as contexts, a controller also includes methods for handling the values of the context attributes, and for triggering and handling events. Important information about this is available in the Programming Manual for Web Dynpro for ABAP. Alongside these predefined method structures, you can create your own methods and call them from your application. The greater the size of a Web Dynpro application, the more effective it becomes to move its logic into separate auxiliary classes. This makes the methods much more flexible, since you can use these auxiliary classes in other application. Web Dynpro Framework itself implements an assistance class. The inherited type of this class gives it special properties that can be used as appropriate by Web Dynpro Framework. In addition, you can create your own auxiliary classes and structure them to match your application landscape.
- Independent auxiliary classes have the additional benefit of optional parameters.
- Remember that the methods from auxiliary and assistance classes can be called *directly* from view controllers as well



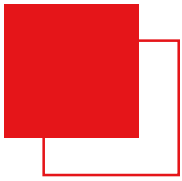


# Controller

## Handling Method Attributes

- We also recommend that you create method attributes in separate auxiliary classes, and reference them from the Web Dynpro methods.





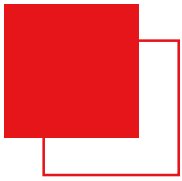
# Controller

## Events

- Events and event handling in Web Dynpro for ABAP are **not** identical to the similar concepts in ABAP.



- In particular, events can be caught and handled only if a controller usage is entered for both controllers involved.

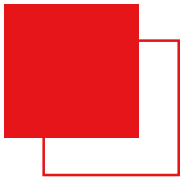


# Controller

## Notes on Assistance Classes

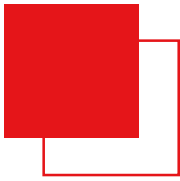


- Assistance classes are instantiated with their component.
- If a particular assistance class is accessed from various components in a component chain, it makes sense to create the class in the main component and then pass it to the subcomponents.
- More information is available under DEMO\_COMMON\_ASSISTANCE1



# User Interface

- You can use many different UI elements when designing the user interface of a Web Dynpro ABAP application, and you can combine and nest these elements in many different ways. The latter aspect of UI design can, however, cause overly complex views to be created that not only tax the user but are also bad for the application's performance. For these reasons, always avoid making the structures in your views overly complex if at all possible.



# User Interface


## Matrix Layout

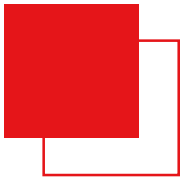
- In a view layout with a regular level of complexity, always use the matrix layout even if the default setting of the View Editor is a flow layout. A matrix layout benefits you by providing several formatting utilities that are otherwise very difficult to design. A matrix layout is not, however, suitable if you want your view to have a very simple layout. In this case, use a flow layout or grid layout for performance reasons.
- In a matrix layout, the properties “stretchedHorizontally” and “stretchedVertically” are set by default. Disable the “stretchedVertically” property, since it can often cause browser errors at runtime. For UI elements embedded in matrix layout, you can define values for height and width under LayoutData. The values you enter here are defined as pixel values ("px") by default. You can change this unit from "px" to "em" or "ex" (see the CSS Standards).



# User Interface

## Dynamic Programming

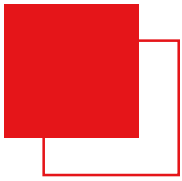
- Web Dynpro ABAP enables you to manipulate the view layout dynamically, which means you can create, modify, or remove specific UI elements at runtime. The user of the application can personalize some properties of UI elements though, making the application developer unaware of the values of these properties.
- If you use dynamic methods to create a UI element, make sure that you give each element an element ID. Technically, this is an optional property, but it is a great help when the element is processed. Implicit personalization can work only if an ID is set, since stable assignments of values can only be made to the same UI element. If you do not define an element ID, the elements are given new random IDs each time they are called and the personalized values cannot be assigned.
-  You often do not know how many elements need to be created at design time. For example, if you want to create a form dynamically, and you do not know its context nodes at design time, you must assign element IDs generically. In this case, ensure that your chosen algorithm provides unique results. Simply numbering the elements sequentially is not enough to ensure that each dynamic UI element appears in the right position in the view. For example, a "Price" attribute may appear as the third attribute in node A, but as the fifth attribute in the structure in node B. In this case, it is more effective to use the attribute name as the element ID.



# User Interface

## ALV Component

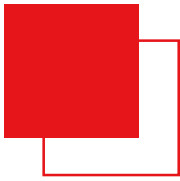
- The ALV component offers application developers a range of integrated services that they can use directly without any additional programming work. These services includes functions such as filters, sort functions, and total functions. However, each additional ALV component you use increases runtime significantly; the table UI control is quicker for displaying simple tables. Use the ALV component only if you are sure you need its extra functions. Always choose a simple Web Dynpro table if sufficient for your application.
- When you use an ALV component, you can pass the data for display using either a regular cross-component mapping or an external context mapping.



# User Interface

## Configuration Options

- Make use of the personalization and configuration options when you develop your applications. When you create generic components, however, remember that subsequent application developers may themselves want to add dynamic UI elements. It is not a good idea to define a maximum set of UI elements at the beginning and assume that many of these elements will be deactivated later. Restrict yourself only to those elements that you actually need when you create your components.

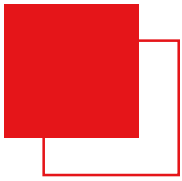


# Delta Rendering

## Configuration Options

- View-based delta rendering is used to improve the performance of user interaction in complex applications when only a part (view) of the displayed page has to be updated. Only the view that has been changed is newly rendered. If several views have been changed, the view rendered is the one containing the changed views and any views beneath it, which means that only a certain part of the page is replaced. For programming, this means that you should avoid unnecessary context updates, and avoid making changes to view elements.
  
- By default all Web Dynpro applications run without delta rendering. You can activate delta rendering for an individual application using application parameter “wdDeltaRendering” in the following way:
  - Setting Delta Rendering Using Application Parameter WDELTA\_RENDERING
  - or
  - Setting Delta Rendering Using URL Parameter sap-wd-DeltaRendering





# Delta Rendering

## Implementing Delta Rendering

- We recommend you use delta rendering for complex pages on which usually only a part of the page is updated.
- Critical for the success of delta rendering are:
  - The design of each view
  - Read-write access is assigned only to objects to be updated (context, view elements)

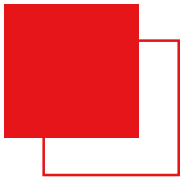


# Delta Rendering

## Delta Rendering Functions

- The purpose of delta rendering is to update only the area of the displayed page that has been changed by the application. The granularity for these updates is a view. Only one area at a time is updated, and accordingly the most inward view of all changed views is refreshed. In the most unfavorable case this is the top view of the application.
- To determine which views have to be updated, any changes to the Web Dynpro programming interface that could lead to a change in the visualization of a view are registered:
  - Navigation
  - Change to the UI element tree
  - Personalization change
  - Changes to Context and “ContextNodeInfos”
  - Messages
- No check is made whether this actually results in a changed value. Just a call to a changing method results in the view being updated.
- User interaction with screen elements also result in the view being updated
- When using context mapping, a change operation to a mapped context node and its subnodes results in an update to each view that maps to these nodes. It is irrelevant whether the subnodes or attributes in the development environment have been copied to the mapping or not.

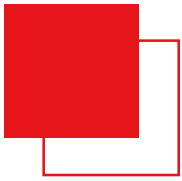




# Delta Rendering

## Effect of Delta Rendering & Restrictions

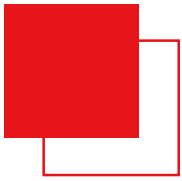
- Since delta rendering already starts before the "costly" rendering process, in appropriate scenarios significant performance improvements can be achieved in the server as well as in the Web Browser.
- The following UI elements are not (yet) supported by delta rendering:
  - TimedTrigger
  - Gantt
  - Network
  - InteractiveForm
  - OfficeControl



# Delta Rendering

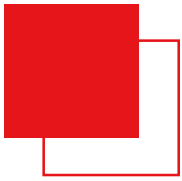
## Possible Side Effects

- If an application uses very fine-granular change operations (for example `SET_ATTRIBUTE` calls for each field of a context node instead of `BIND_TABLE`), delta rendering can lower performance.
- Delta rendering errors may occur due to JavaScript errors when calling `document.getElementById(...)` after a server roundtrip, or because screen areas have not been updated. If errors occurs see SAP Note 1021981.



# Gantt Charts

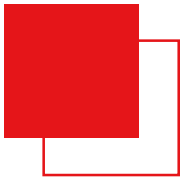
- A Gantt chart is used for displaying multilayer bar charts, usually over a defined time period. For instance, they can be used for displaying an overview of the progress of a project.
- Web Dynpro ABAP provides two very different technical implementations of Gantt charts:
  - JGantt Integration
  - IGS interface



# Gantt Charts

## JGantt Integration

- Web Dynpro ABAP provides a defined interface for a JGantt control. JGantt is a variant of the JNet control and is based on a specially formatted XML file.
- With this type of Gantt chart users can change values at runtime. For instance, you can move bars along the time axis or increase the length of bars. This flexibility is made possible by the high complexity of the JGantt control. The UI Element Gantt is one of the UI elements provided in Web Dynpro ABAP and can be found in the graphic category.



# Gantt Charts

Interface for Internet Graphics Service (IGS)

- In many cases you actually only want to show a static display of values and their relationships in a Gantt chart. If so, you do not need the flexible but highly complex options offered by the JGantt control, and instead you should use the BusinessGraphics UI element. This UI element itself is very flexible, and can be used in various display formats by using the attribute `chartType`, such as `BusinessGraphics (chartType gantt)`.



# Multiple Parallel Applications

## Separate Roll Areas

- Each Web Dynpro application runs in its own role area. Multiple applications can run in parallel or communicate with each other only if one of the following is used:
  - The database
  - shared objects/shared memory,
  - Portal events
- For this reason, it is not possible to close an application by closing another application (that is, closing a browser window). You must therefore make sure that a self-contained task creates only one application. This reduces both programming work and the number of browser windows for the user.







# Multiple Parallel Applications

## Dialog Box

- Dialog boxes (often called popups) are special display elements that open alongside a running application. Dialog boxes always start a separate phase model instance, which means that their source window cannot be refreshed or edited while the dialog box is open (while the phase model instance still exists). You can avoid dialog boxes in many cases by displaying content in an additional window view. This removes the source of many errors and improves performance.



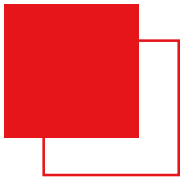
- Modeless dialog boxes are not supported.



# Using Components and Component Interfaces

## Components and Interface Definitions

- The Web Dynpro Framework enables you to use functions and data from one component in another by using a component in another component. A prerequisite for this is that a usage declaration is created in the "using" component at design time. This itself requires the used component to exist and be recognized by its name. This is not always the case, however. To help you in cases where the used component cannot be identified, Web Dynpro Framework offers a less rigid technology: Component interface definitions. In this case, you first enter the usage of a separately defined component interface. All components that implement precisely this interface definition can now be used by means of the usage declaration from the main component. The concrete implementation of this method is passed as a parameter at runtime.
- Example: If a component wants to embed two components that are not yet known at design time, all components that are able to be embedded must implement a common interface definition. The calling component creates two usages for the interface. At runtime, the implementation is passed as a parameter when the used components or interfaces are created.



# Using Components and Component Interfaces

## General Information About Handling Interfaces

- A component can implement multiple interfaces.
- Interfaces cannot be inherited.
- You can implement interfaces in the Application Configuration or by calling them as ULR parameters.



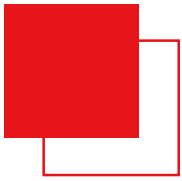
- Interfaces do not implement their own namespaces. We recommend strongly that you use prefixes when you name your interfaces.



# Using Components and Component Interfaces

## Implementation of Interfaces for Customer Developments

- Using interfaces in a Web Dynpro component benefits customers by giving them a clean basis for their own further developments. When creating a local development, you can implement a used interface in a separate component and add your own aspects to an application delivered by SAP.

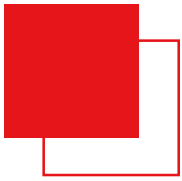


# Using Components and Component Interfaces

## Usage Groups

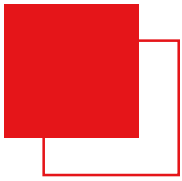
- In certain circumstances, you may want a main component to use an individual component more than once. To enable this, you can create multiple usages for the same component. The same applies to the usage of interface definitions.
- As long as the number of used components is reasonable and known already at design time, we recommend that you use a usage group instead of a static list of individual usages. Usage groups organize the dynamically programmed use of components or interface definitions.
- Before you decide to use a solution that involves dynamic component usages, remember that you also need to program navigation to any components whose use is implemented using usage groups. This makes navigation more complicated and also means that the application is more difficult to maintain.





# Creating and Deleting Components and Views

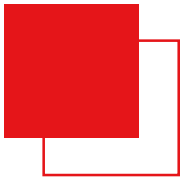
- The most important way to save memory is to only call on those *resources currently needed*. For Web Dynpro this means only keep those components and views in the memory that are currently being displayed.
- The prerequisite for this is that an application has a *modular structure*, which means it does not have to keep hold of all components at the same time.
- If this prerequisite is fulfilled, another important issue concerns dynamic creation and deletion of components and views.



# Creating and Deleting Components and Views

## Components

- Component COMP\_A has component usage USAGE\_A\_B in component COMP\_B. Web Dynpro instantiates component COMP\_B for component usage in the following cases:
  - An interface view from COMP\_B is made visible in a currently visible window of COMP\_A (see views)
  - A context mapping exists from COMP\_A to the context of COMP\_B
- Otherwise component COMP\_B is not automatically instantiated. If a component is no longer needed, this should be deleted using DELETE\_COMPONENT. Indeed, no component views must still be visible, and there must be no context mapping.

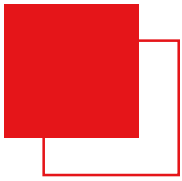


# Creating and Deleting Components and Views

## Views

- The lifespan of a view is defined by its visibility: A view is automatically instantiated as soon as it is made visible. If the view has setting Lifespan: when visible, the view is deleted as soon as it is no longer visible. Otherwise the view can only be deleted by removing the entire component.
- The concept, visible, is used in different ways in Web Dynpro and can therefore easily be misunderstood. What is actually visible on the screen ultimately depends on the views visible in the view assembly and on the visibility of the individual UI elements (property visibility).
- When we refer to the visibility of views, as above, we are referring to the view assembly. The view assembly is the tree of visible views of a window. One view is always visible for each window and view container. Whether a view is instantiated or not depends only on the view assembly and not on the visibility of UI elements.
- A view is made visible and therefore also instantiated when it is embedded in a visible window and marked as a default view, or is navigated to when one of its inbound plugs is activated. It becomes invisible when the user navigates to another view in the same window or view container.

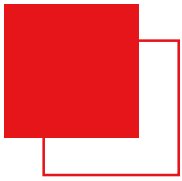




# Creating and Deleting Components and Views

## Memory Saving Scenarios

- Use of Lifespan: when visible
  - If you use this setting, the view is deleted when it becomes invisible as a result of user navigation (as described above). You should choose this setting if the view is not repeatedly used in the application.
  - If the view becomes invisible, the following happens: Method WDDOEXIT is called. The view controller and its attributes, the local view context, and the UI element tree are deleted. Note that the memory can only be released when these things are not being referenced by other objects. Therefore, do not keep any references from the view, context, UI elements, or context nodes, elements, node infos, and so on, outside of the view. Otherwise, application developers will be responsible for removing these references.
  - If you think you may want to return later to the UI status (for example, the first visible table row), you have to keep it outside of the view: For example, as an attribute of the assistance class or context node of the component controller that is mapped to the view.



# Creating and Deleting Components and Views

## Making Views Visible Dynamically

- In the view container for VIEW\_A embed VIEW\_B and EMPTY\_VIEW in the window. Set EMPTY\_VIEW as the default. For VIEW\_A create an outbound plug TO\_VIEW\_B and for VIEW\_B create an inbound plug IN. Connect both plugs in the window.
- VIEW\_B is now not automatically instantiated with VIEW\_A. Instead, EMPTY\_VIEW is used as the placeholder. By triggering plug TO\_VIEW\_B, VIEW\_B then becomes visible. You can then also use the plug to notify VIEW\_B which object it is to display.



# Creating and Deleting Components and Views

TabStrip: Make visible for active tab only

- Each tab of a TabStrip is to make another view visible. For each tab a “ViewContainerUIElement” must be created, and this in turn must have a view container in the window. The view to be displayed is embedded in the view containers. Since there must always be a default view for each view container, they are therefore all default views. This means that all views are immediately instantiated. Proceed as described in the example below.
- Example application DEMO\_TABSTRIP\_VIEWS