How to Use the Column Layout in a Web Dynpro for Java Application

Applies to:

Web Dynpro for Java 7.11

For more information, visit the User Interface Technology homepage.

Summary

This tutorial explains the ideas behind the Web Dynpro `ColumnLayout` layout manager and explains how to use this layout in Web Dynpro views.

The tutorial application demonstrates how this layout can be used in a real-world scenario and provides a view for interactively playing around with layout properties.

Author: Web Dynpro Java Team

Company: SAP AG

Created on: 29 June 2010

Table of Contents

Introduction 3
Prerequisites 3
Objectives 3
Some theory 4
How are UI elements arranged on the grid? 5
RowSpan and ColSpan 5
The Tutorial Application 6
The Sales Order Creation Screen 6
The ColumnLayout playground 11
Restrictions 16
Copyright 17

Introduction

In this tutorial you will learn how to use the Web Dynpro `ColumnLayout` layout manager.

The `ColumnLayout` is unique among the Web Dynpro layout managers because it allows alignment of UI elements from different views or even different Web Dynpro components. In contrast, all other Web Dynpro layout managers can operate only on UI elements within a single UI element container.

Prerequisites

You need to install the NetWeaver Developer Studio (Version 7.11 or later) in order to compile and deploy the tutorial application. The SAP Java AS to which this application is deployed should have the same or newer version as the NWDS.

The tutorial application is available as a development component (DC). You need to import the software component HM-WDUIDMKTCNT, which contains the DC `tc/wd/tut/layout/column`. The exact steps are described in a separate document.

Objectives

After reading through this tutorial, you should

- Understand the concept behind the `ColumnLayout`
- Be able to use `ColumnLayout` to align elements from embedded views and components

Some Theory

The `ColumnLayout` defines a grid structure. This grid contains a fixed number of columns determined by the `colCount` property, and a number of rows determined by the layout algorithm. The grid structure can be inherited either by embedded views or embedded interface views derived from child components. This allows alignment of UI elements over view boundaries and at different levels of embedding.

A UI element container with a `ColumnLayout` may occupy between one and five so called *major columns* of the grid. The `colCount` property defines the number of major columns.

Each major column consists of three *minor columns*:

- A label column
- A field column and
- A spacing column.

The spacing column is managed completely by the framework and can be ignored by the application developer.

Each (major) grid cell is subdivided into a label and a field *region*.

A UI element placed on the grid can occupy several (major) columns and several rows. The number of occupied columns and rows can be specified using the `colSpan` and `rowSpan` properties of the element's layout data. If the `colSpan` or `rowSpan` properties are not specified, then the UI element's natural grid size is used. The natural grid size varies for each UI element. For example an input field has a default grid size of 1 row x 1 col. A radio button group with 3 radio buttons per row and 2 rows has a grid size of 2 x 3 etc.

Other grid sizes are defined as follows:

- Embedded containers with `ColumnLayout`: *m* x *n*, where *m* is the number of rows needed to place all elements of the embedded container and *n* is the `colCount` of the embedded container
- Embedded containers with any other layout: one row, all columns of the outer grid
- `TabStrip`, `Tree`: one row, all columns
- `Table`: as many rows as given by table's visible row count, all columns
- `ViewContainerUIElement`: As many rows and columns as determined by the root element of the contained view.

The layout data for each UI element placed inside a `ColumnLayout` are defined by instances of the interface `IWDColumnLayoutData`. See right.

LayoutData [ColumnLayoutData]	
colSpan	-1
interlineSpacing	false
LayoutData [ColumnLayoutHeadData]	
colSpan	2
interlineSpacing	true
requiresEmptyNewRow	false
rowSpan	-1

In keeping with other Web Dynpro layout managers, a sub-interface called `IWDColumnLayoutHeadData` exists that forces the UI element to be positioned at the start of a new row.

The additional property `requiresEmptyNewRow` causes the UI element to be placed at the next, completely empty row of the grid. All rows that contain cells of other UI elements spanning multiple rows will be skipped.

How are UI elements arranged on the grid?

The layout algorithm fills the grid top-to-bottom and left-to-right. Since it is possible to embed views within each other that may each use `ColumnLayout`, it is therefore quite possible that recursive calls of the layout algorithm will be required.

The placement algorithm traverses the visible children of the UI element container and determines the next grid position for the current UI element using the following set of (simplified) rules:

- If the current element has `IWDColumnLayoutHeadData`, a new row is started and the element is placed in the label region of the first free cell in the next row
- If the current element is a label (or a drop-down list used as label), it is placed in the next free label region. Labels are always contained in label regions and cannot span columns or rows.
- If the current element is not a label, it is placed in the next free region. If this is a label region, then the element spans into the next field region as well.
- If the current element is an `InvisibleElement`, it is placed in the next free (label or field) region. `InvisibleElement` instances can be used to fine-tune the placement of other UI elements.
- If the current element has a built-in label (such as a `CheckBox` or a `RadioButton`), it is placed entirely into the next field region. The adjacent label region remains empty. This rule automatically aligns such elements with input fields etc, without having to use an `InvisibleElement` to skip the label region.
- If the current element is a container that uses a `ColumnLayout`, the children of this container are laid out and the resulting grid is placed inside the parent grid.

RowSpan and ColSpan

If a UI element defines an explicit `colSpan`, then this element occupies the corresponding number of **major** columns. This implies that the element will span all *minor* columns of the spanned major columns.

If a UI element defines a `rowSpan`, it spans the corresponding number of cells in the subsequent rows including the label regions. That means for example, if you have a UI element in the field region of row 3 with a row span of 2, it occupies the field region in rows 3 and 4, and it reserves the label region in row 4.

 **Important:**
Nesting of `ColumnLayout` containers is only possible when using a `TransparentContainer` ***without*** scrolling. Other containers like `Group` or `Tray` cannot be used. To implement titles for such nested containers, add a `SectionHeader` UI element to the container and set the `labeledBy` property of the container to the ID of that section header. Additionally, the `isLayoutContainer` property must be set to `false`.

A general recommendation is to enable wrapping for all labels (disabled by default). Label wrapping keeps the screen usable even if it gets rather small.

Input fields inside a column should be given the same width (e.g. 100%) to make the screen look more calmly.

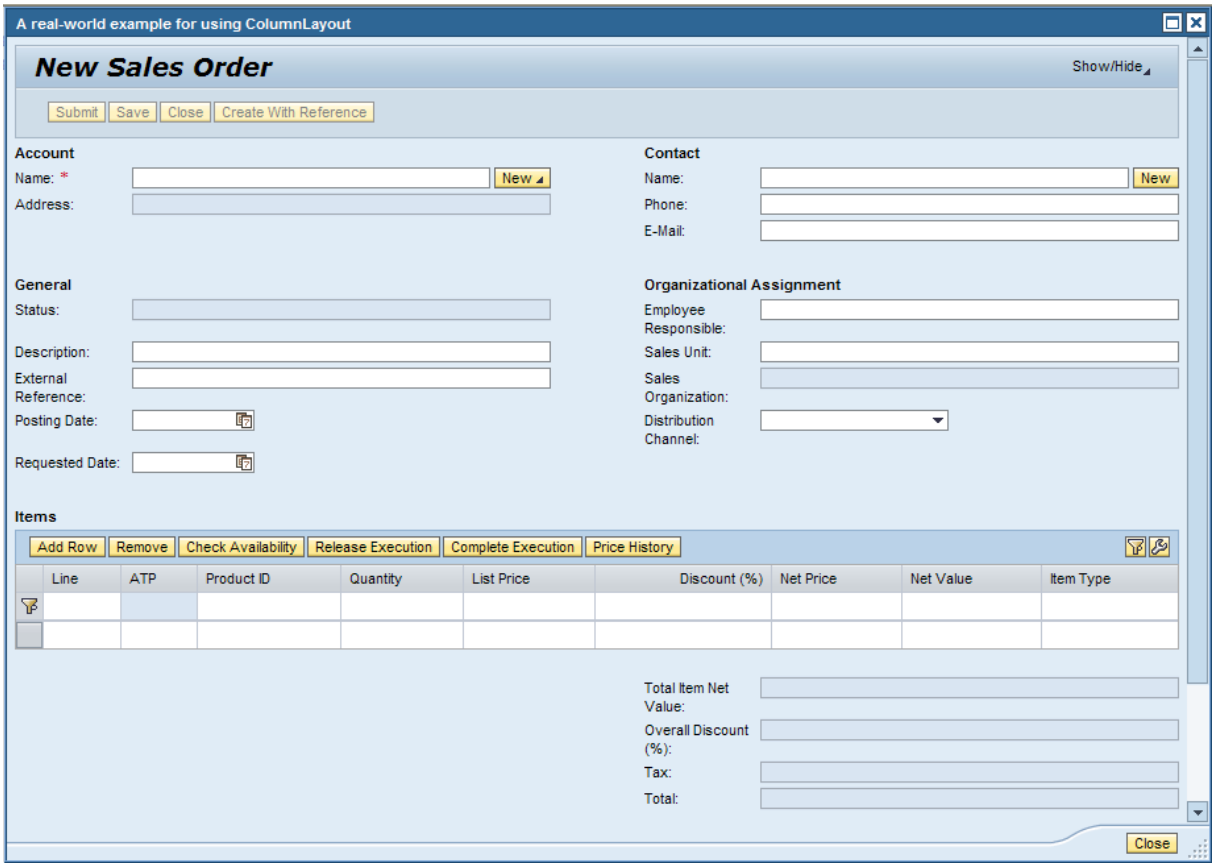
The Tutorial Application

The tutorial application contains two windows. The first window shows a sales order creation screen and is composed of a number of embedded views. You can show and hide each embedded view interactively and observe how the screen is adapted.

The second window contains a “playground” where you can interactively change the layout data of UI elements inside a ColumnLayout and where you can interactively embed a group of checkboxes and a complete view.

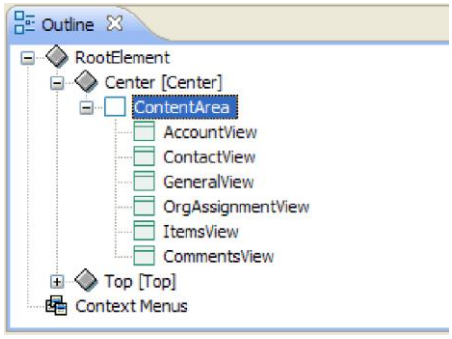
The Sales Order Creation Screen

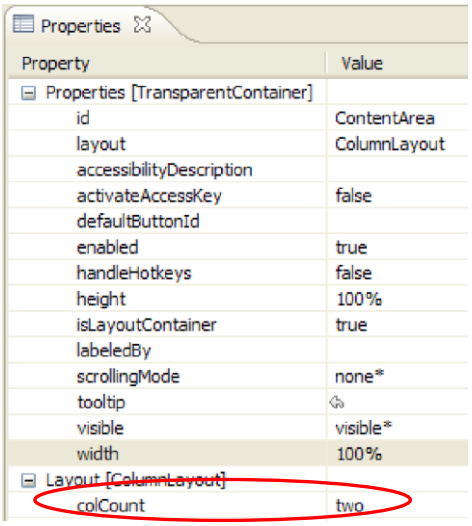
After clicking on the link to open the real-world example, the following window will appear:



The screen shows a sales order creation form and is composed of several sub-screens which can be identified by their titles like “Account” etc.

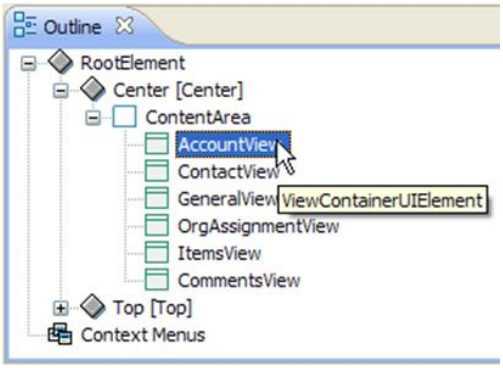
The two-column layout has been achieved by assigning the content area container a `ColumnLayout` with `colCount` = `two`:

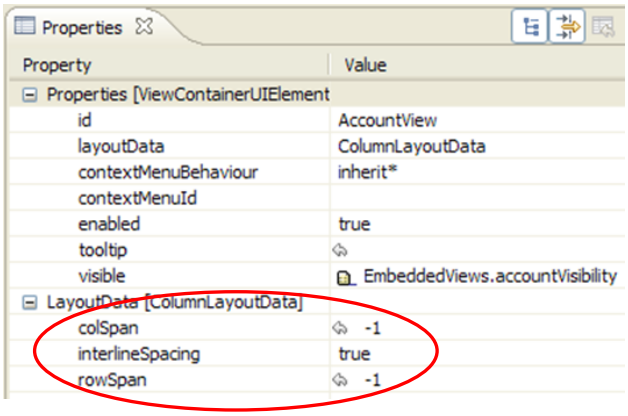




Each embedded view is represented by a `ViewContainerUIElement`.

Note that you have to enable the “Show Advanced Properties” setting in the “Properties” view to see the column layout data:





The `colSpan` and `rowSpan` properties have their default value `-1` which means that the number of spanned columns and rows is determined automatically.

The `interlineSpacing` property has been set to `true` which causes some white space to be inserted between the given view and its preceding row.

Change the width of the popup window and observe how the layout adapts to the changed size. The label and field columns are resized in proportion to the window width. When needed, labels are wrapped across multiple lines and the fields and labels in the other columns are aligned accordingly. If you don’t want certain labels to be wrapped, just leave the label’s `wrapping` property set to its default value of “no wrapping”.

A horizontal scrollbar will appear only when the table in the “Items” area cannot become any smaller.

This behavior of the `ColumnLayout` keeps a screen usable even when the window size or the screen resolution varies.

What makes the `ColumnLayout` unique in Web Dynpro is the fact that it allows UI element alignment across view and component boundaries. In our example each sub-screen has been implemented using a separate view (which could also belong to a different Web Dynpro component).

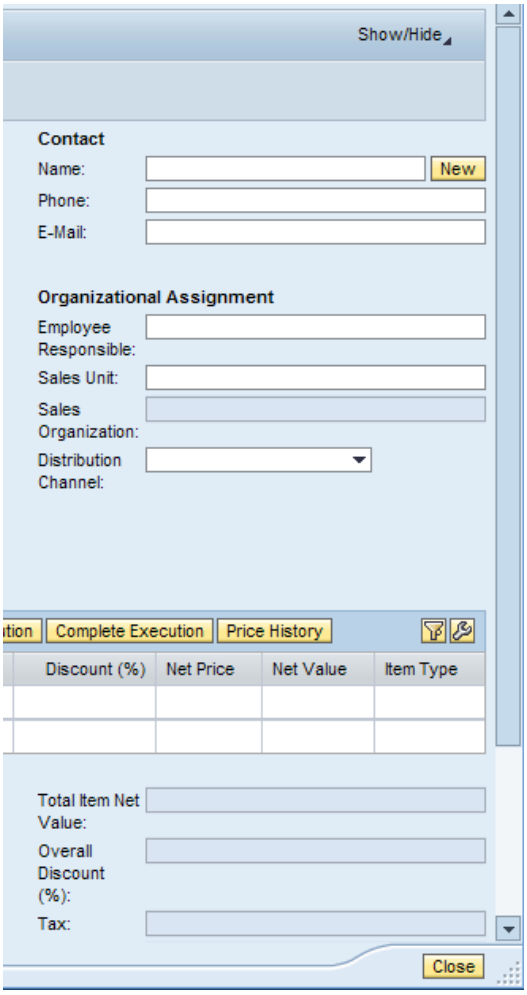
Have a look at the “General” and “Organizational Assignment” sub-screens. These views are aligned vertically at the top, and their labels and fields remain vertically aligned even if some labels need to be wrapped.

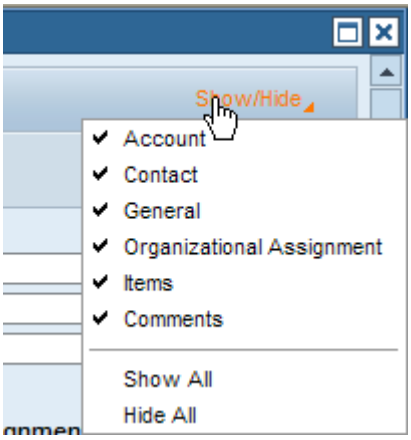
Now let's have a look at the **column** structure of the screen. The sub-screens are aligned horizontally according to the two-column structure. The "Items" view consists of a table (which spans two columns) and a form with the fields "Total Item Net Value" etc. Even this form is correctly aligned with all other views in the second column:

The "Items" view is an example of an embedded view in which certain parts cannot align their content on the column grid (the table), but other parts need to remain aligned (the form fields).

Note the empty rows above the "General" and the "Items" view. This is a consequence of setting the value of the `interlineSpacing` property to `true`.

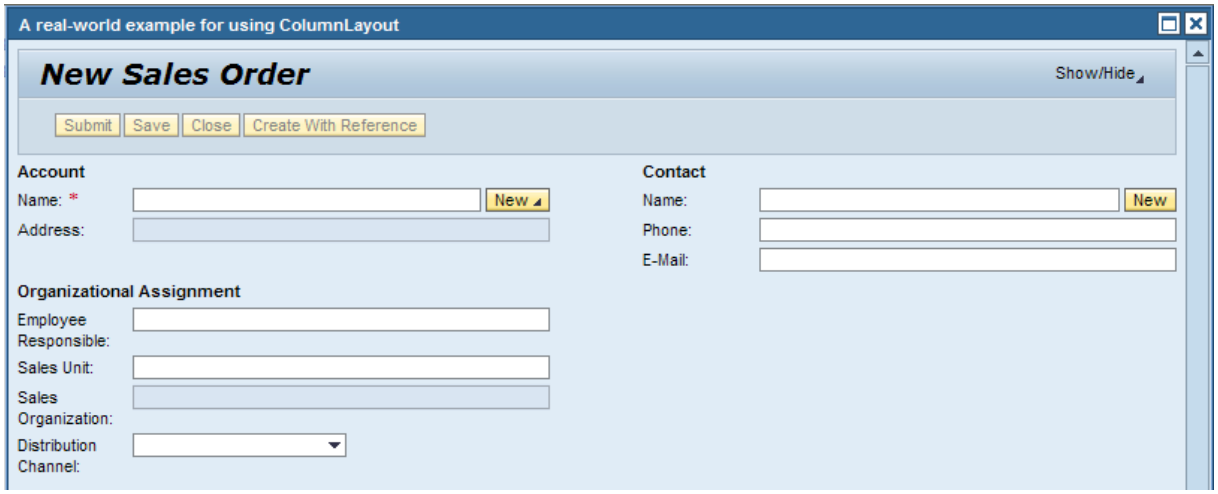
The "Account" and "Contact" views have this setting too, but because they are positioned in the very first row, the empty row is suppressed automatically.





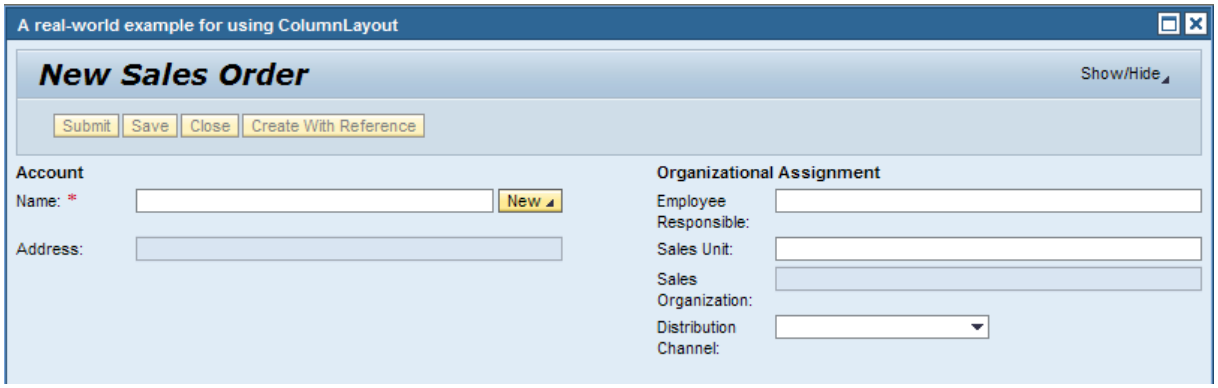
In the header of the screen you find a menu "Show/Hide" with entries to show or hide the sub-screens:

If you hide the "General" view, then observe that the "Organizational Assignment" view jumps to the first column and occupies the place where the "General" view was located.



If you now hide the “Contact” sub-screen, observe that the “Organizational Assignment” sub-screen jumps across to the second column of the first row and the layout adapts again.

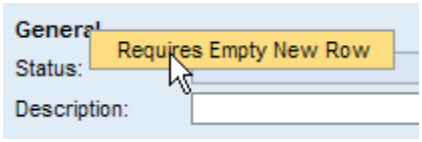
Through all of these rearrangements, the horizontal and vertical alignments on the grid remain intact.



Using the `ColumnLayout` you can design screens that remain correctly aligned and contain no holes even if parts of the screen are suppressed or displayed dynamically.

If needed, this flexibility can also be restricted. Have a look at the “General” sub-screen. Here we have forced the “General” sub-screen to always start ***after*** a new empty row. This makes the initial screen look slightly better.

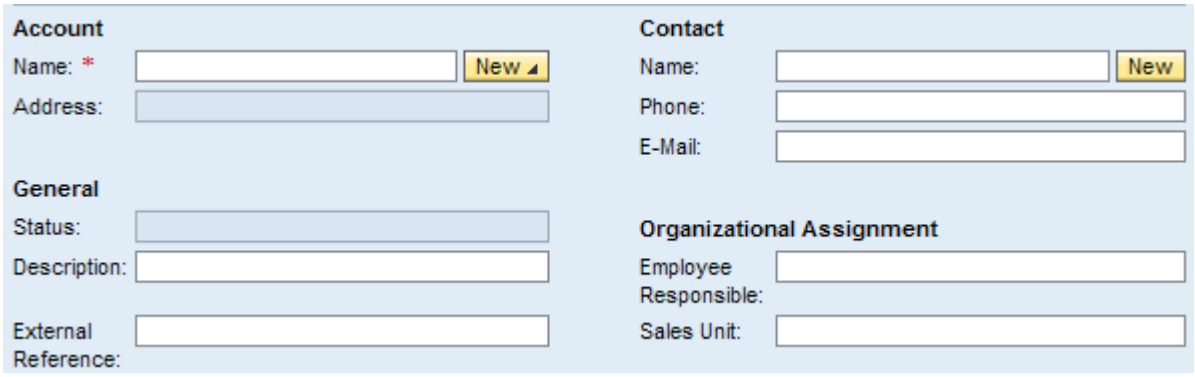
How would the screen have looked without this setting?



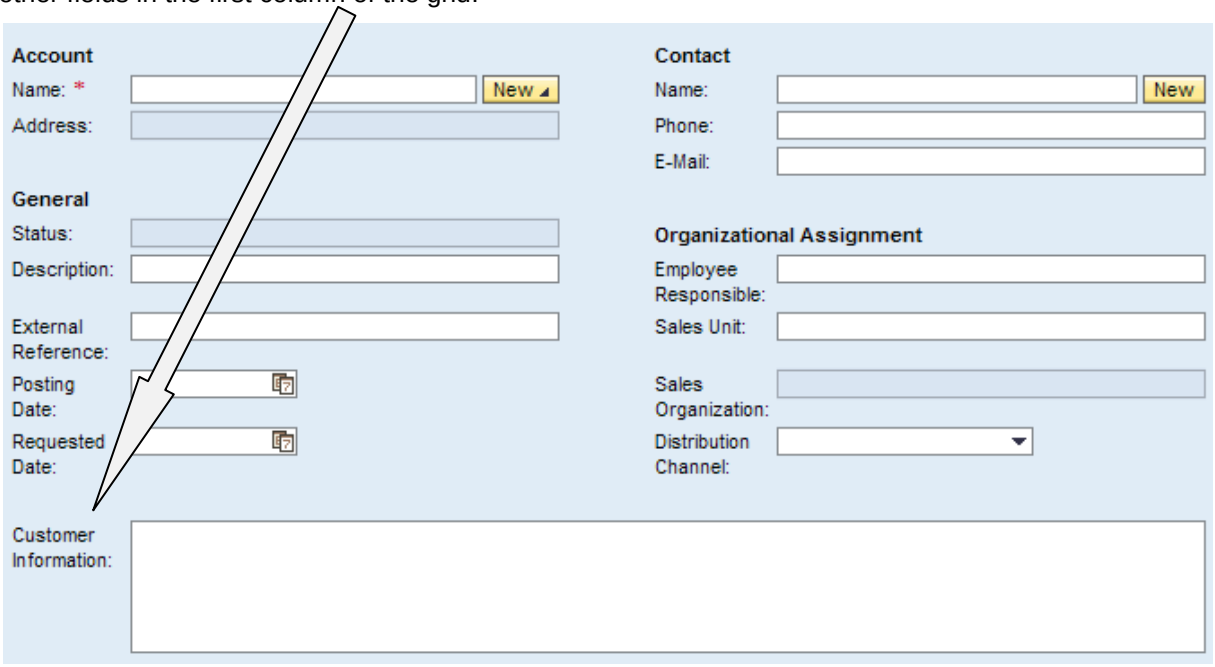
Right-click the header of the “General” sub-screen and deselect the “Requires Empty New Row” entry.

Note that the header moves one row up and is not aligned anymore with the “Organizational Assignment” header. The reason for this behavior is that the grid cell under the “Address” field was still free, and therefore has been used to place the next UI element (the section header in this case).

Only by forcing a new empty row can this free cell be skipped.



Hide the “Items” sub-screen and observe the (untitled) sub-screen with the “Customer Information” label. The `TextEdit` has been defined to span two columns. Therefore, it takes the complete width including the spacing columns and the label region of the second column. The left border of the `TextEdit` is aligned with all other fields in the first column of the grid.

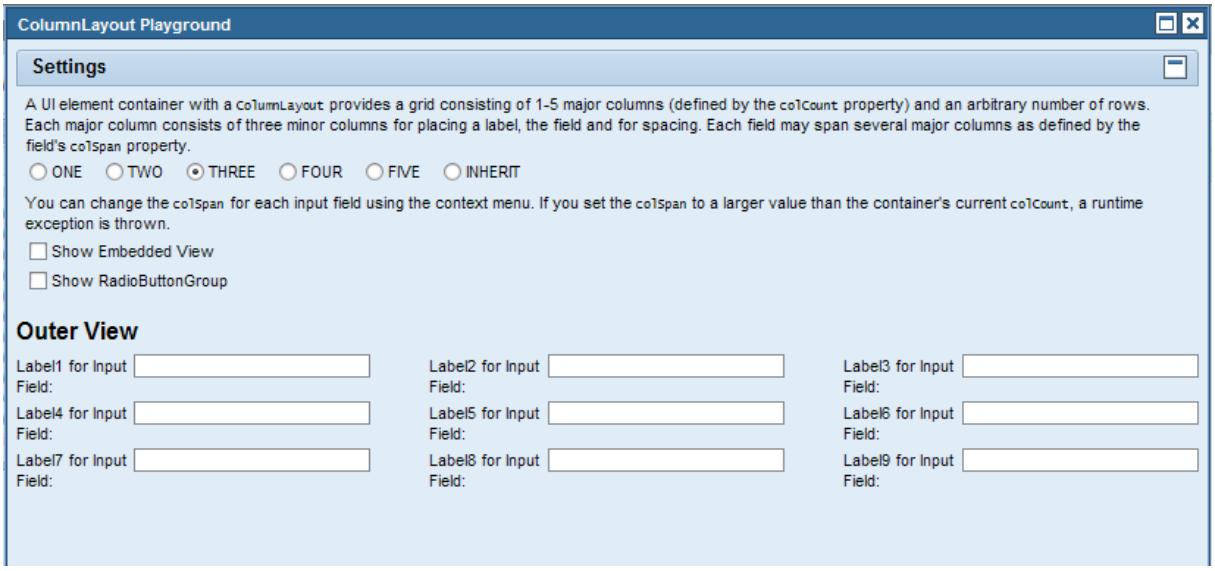


All sub-screens have been implemented using a non-scrolling `TransparentContainer` with a `ColumnLayout`. The sub-screen headers have been implemented using a `SectionHeader` that is connected to its `TransparentContainer` by setting the `labeledBy` property to the identifier of the `SectionHeader` element.

The ColumnLayout playground

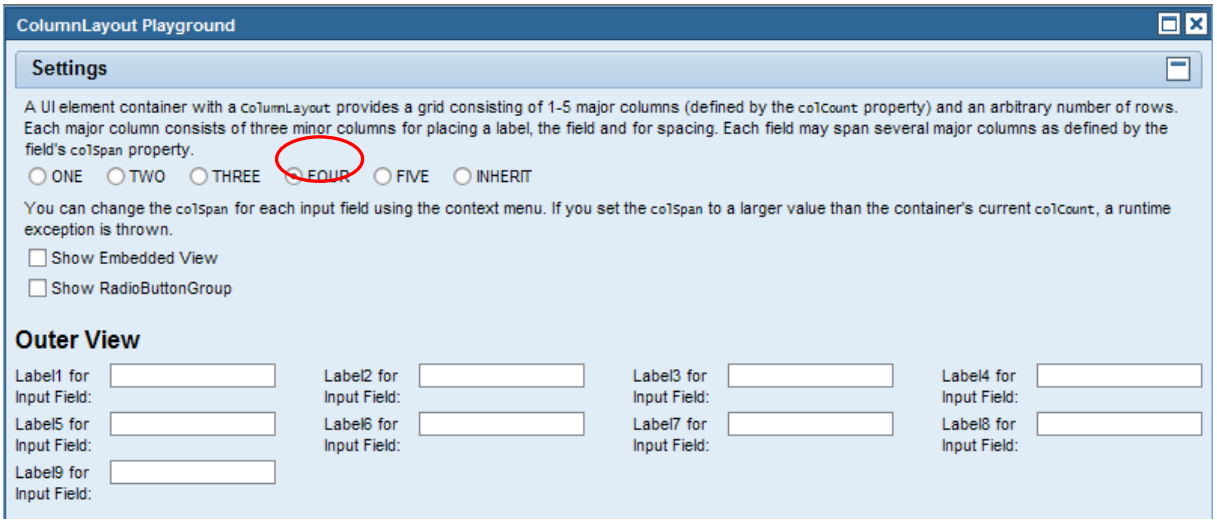
The second popup window of our tutorial application allows you to play around with some properties of the `ColumnLayout` interactively. You can also embed a view interactively to demonstrate how embedded views can inherit the column structure of the grid into which they have been embedded.

When you open the playground window, the following screen appears:



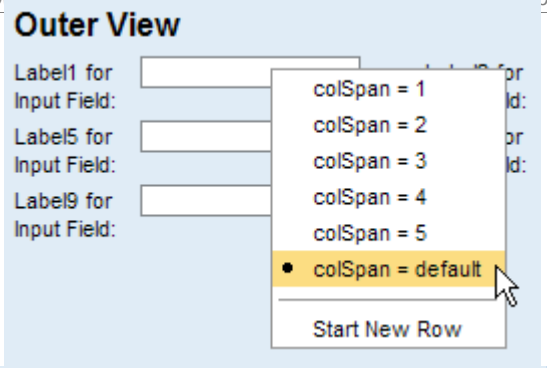
The screen initially displays a settings area on top and a content area below. The content area displays a form consisting of nine label/input field pairs arranged in three columns.

Use the radio buttons to change the column count of the `ColumnLayout`:



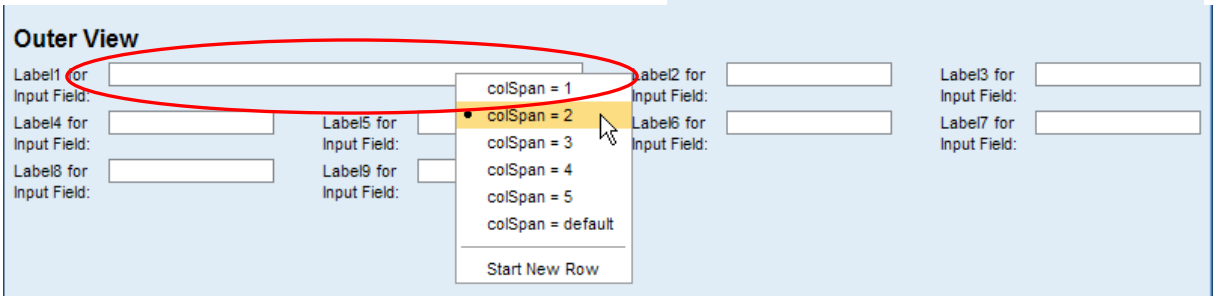
As expected the label/field pairs are arranged in the selected number of columns. The “INHERITED” setting leads to a single column. This is just a default value for the case that there is no outer column layout from which the column count could be inherited.

The `colSpan` for each input field can be changed using a context menu:



The initial value “default” has the effect that the label field takes just one cell. For other UI elements this might be different. For example, by default, a table occupy all the columns and one row.

Change this `colSpan` value and observe how the layout changes:



When a field has a `colSpan` larger than 1, it spans the field region of its first cell and both the label and field region of the spanned cells.

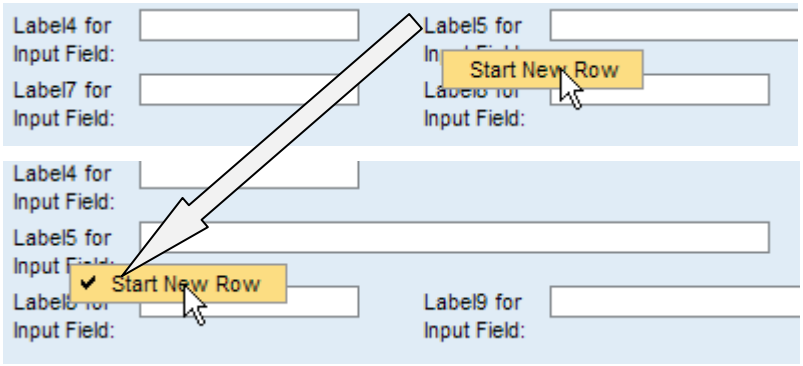
What happens if we set the `colSpan` to a larger value than the available `colCount`? In that case, the Web Dynpro runtime will throw an exception and the application is terminated:

500 Internal Server Error **SAP NetWeaver Application Server/Java AS**

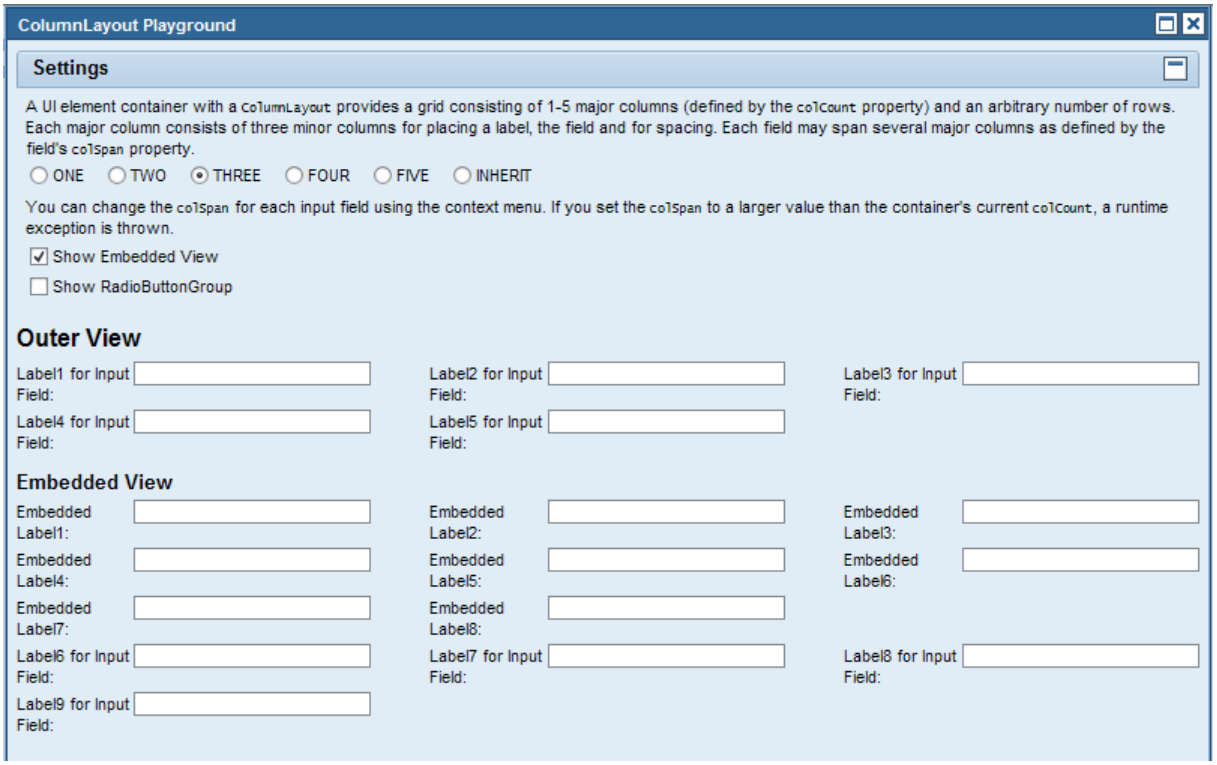
The initial exception that caused the request to fail, was:

```
com.sap.tc.webdynpro.services.exceptions.WDRuntimeException:
com.sap.test.tc.wd.tut.layout.column.View1.InputField1: Cannot span more
columns (5) than available (4)
   at com.sap.tc.webdynpro.clientserver.uielements.adaptbase.AbstractAdapter._assertionFailed
(AbstractAdapter.java:589)
   at com.sap.tc.webdynpro.clientserver.uielib.standard.uradapter.ColumnLayoutAdapter.access$1800
(ColumnLayoutAdapter.java:78)
   at
com.sap.tc.webdynpro.clientserver.uielib.standard.uradapter.ColumnLayoutAdapter$PackedGrid.getMinorColumnSpan
(ColumnLayoutAdapter.java:1840)
   at com.sap.tc.webdynpro.clientserver.uielib.standard.uradapter.ColumnLayoutAdapter$PackedGrid.arrange
(ColumnLayoutAdapter.java:1689)
   at com.sap.tc.webdynpro.clientserver.uielib.standard.uradapter.ColumnLayoutAdapter$PackedGrid.<init>
(ColumnLayoutAdapter.java:1563)
   ... 91 more
```

You can also interactively specify if a label or input field should start in a new row. This will automatically change the layout data of this element from `IWDColumnLayoutData` to `IWDColumnLayoutHeadData` and vice-versa.

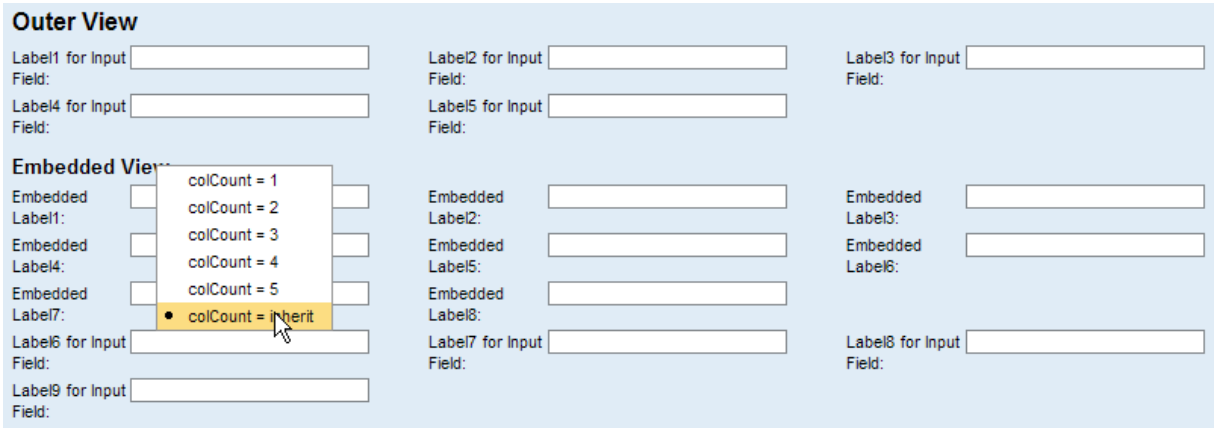


Reset the playground to its initial settings (by closing and reopening the window) and select the checkbox "Show Embedded View":

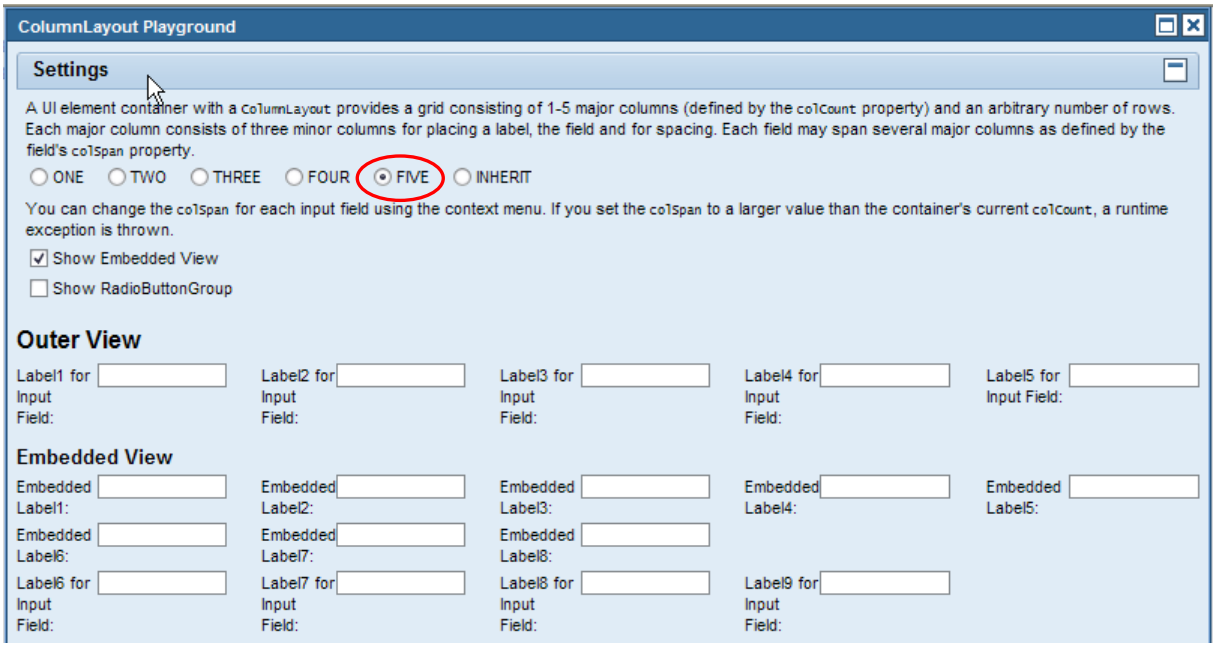


The embedded view is placed in a new row after the 5th input field which corresponds to the position of the `ViewContainerUIElement`. The embedded view's root is a `TransparentContainer` that uses `ColumnLayout`, therefore the elements of the embedded view are positioned in the grid inherited from the embedding view.

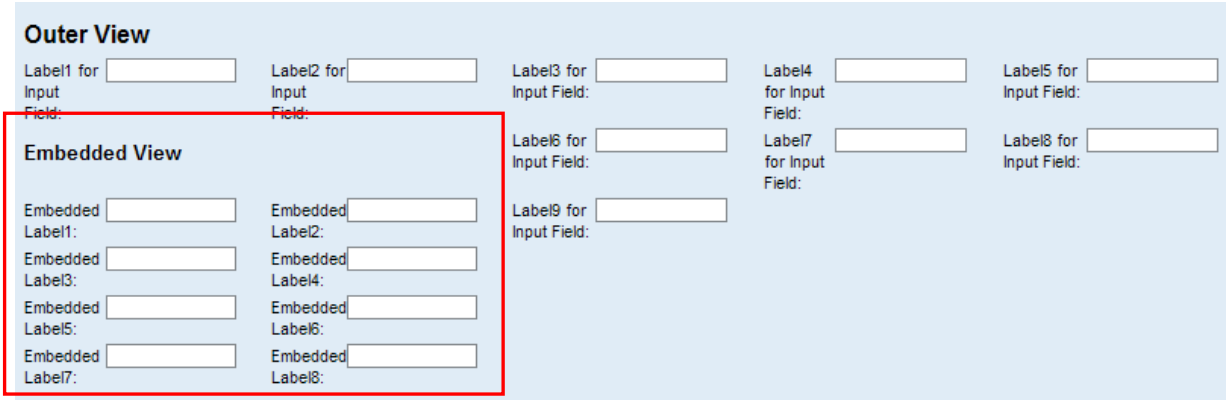
The embedded view initially has `colCount = inherit`:



Therefore the child (or embedded) container uses the same number of columns as the parent (embedding) container. Change the `colCount` of the outer container and observe how the `colCount` of the child container adapts automatically:



Change the `colCount` of the embedded view to two:



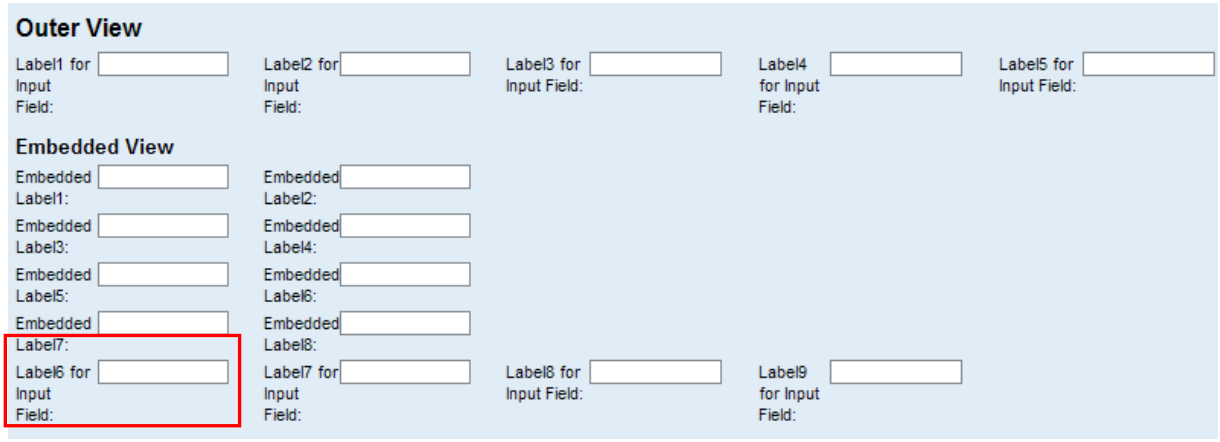
The fields of the outer view are placed top-to-bottom, left-to-right in five columns as determined by the outer views column count. After the 5th field, the inner view is embedded.

As the `colCount` of the inner view has been set to two, the inner view occupies two columns of the grid and as many rows as needed to place all its children.

Note that the next free cell (after having embedded the inner view) is not the cell directly to the right of the embedded view's header, but the cell at the next column. The reason for this is that the `ColumnLayout` always reserves a rectangular area for embedded containers.

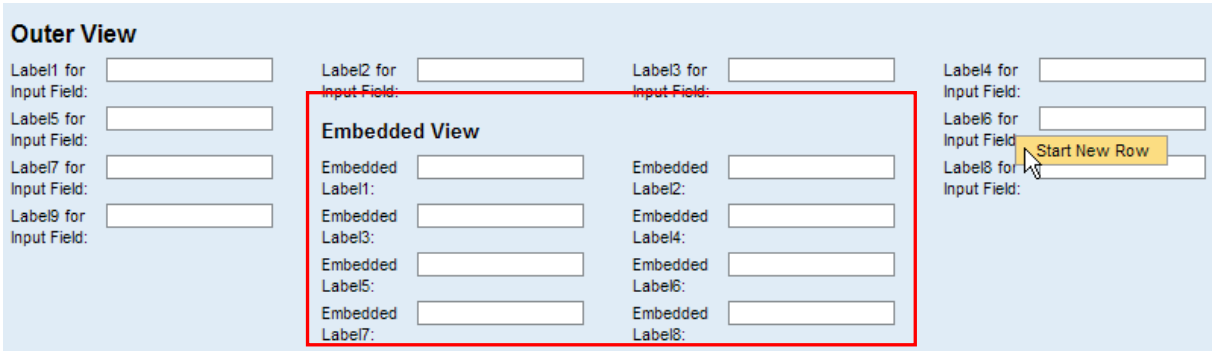
Therefore the placement continues in the third column and the labels and fields of the outer container are placed row-wise in the available free cells.

Right-click the 6th label of the outer view and select "Start New Row": The placement now continues in the first columns of the row under the embedded view:

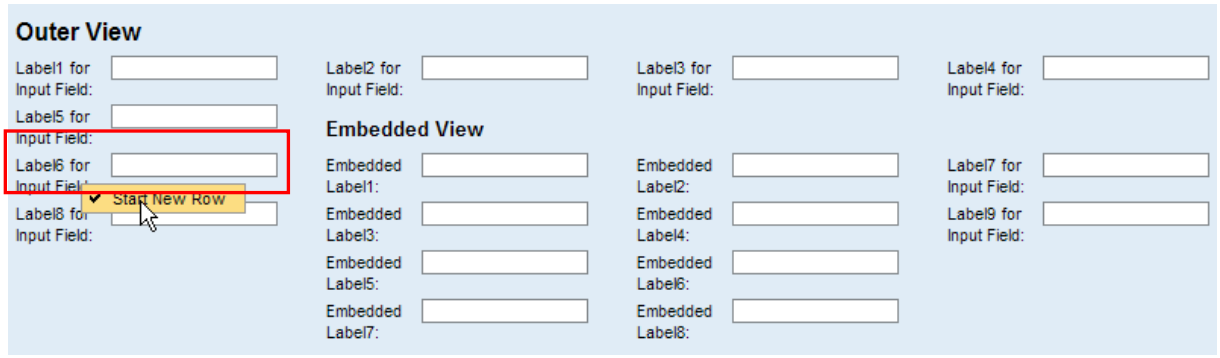


But what would have happened if the embedded view did not occupy the first columns?

This can be checked by changing the `colCount` of the outer view to "FOUR". Now the embedded view starts in the second column and occupies five rows. The next free grid cell for the 6th label is now the cell below the 4th label:

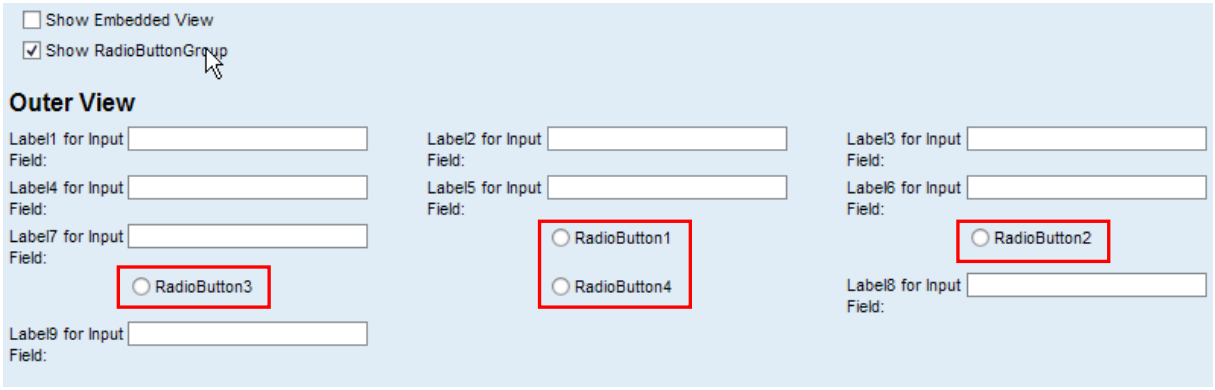


If we select the "Start New Row" entry again, the 6th label moves to the first column of the 3rd row and the placement of the other elements occurs in a row-wise fashion, skipping the embedded view:



Only if we additionally set the `requiresEmptyNewRow` property, would the 6th label be placed at the row below the embedded view.

The “Show Embedded RadioButtonGroup“ setting demonstrates how a set of radio buttons can automatically placed in the field regions of the corresponding cells without the need for using placeholders to skip the label regions:



Restrictions

The View Designer in the NWDS is not yet able to display `ViewContainerUIElement`s inside a `ColumnLayout`.

Copyright