WHITESTEIN Technologies

Living Systems® Process Suite

# Todo and Document Forms

## Living Systems Process Suite Documentation

3.2
Tue Jan 12 2021

Whitestein Technologies AG | Hinterbergstrasse 20 | CH-6330 Cham
Tel +41 44-256-5000 | Fax +41 44-256-5001 | http://www.whitestein.com

# Contents

# Chapter 1

# Main Page

**Important:** This guide documents the forms as implemented by the `ui` module, as opposed to the `experimental implementation in the forms module` of the Standard Library.

The *form* defines the content of a `to-do` or `document` which is displayed in the front-end application, the *Application User Interface* (To-dos are produced by the `User Task` and dedicated objects called `documents`. It serves to obtain information from the front-end user: The user opens the to-do or document, fills out the form, and submits it: On submit, the document instance ceases to exist, and the to-do becomes accomplished and the execution of the process instance can continue.

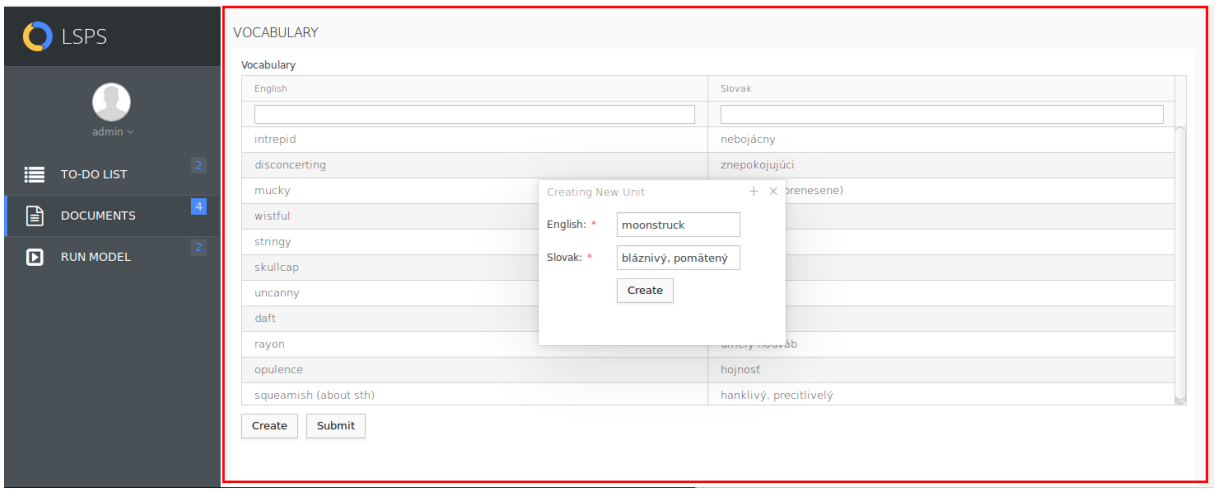



**Figure 1.1 Form with a table in a document**

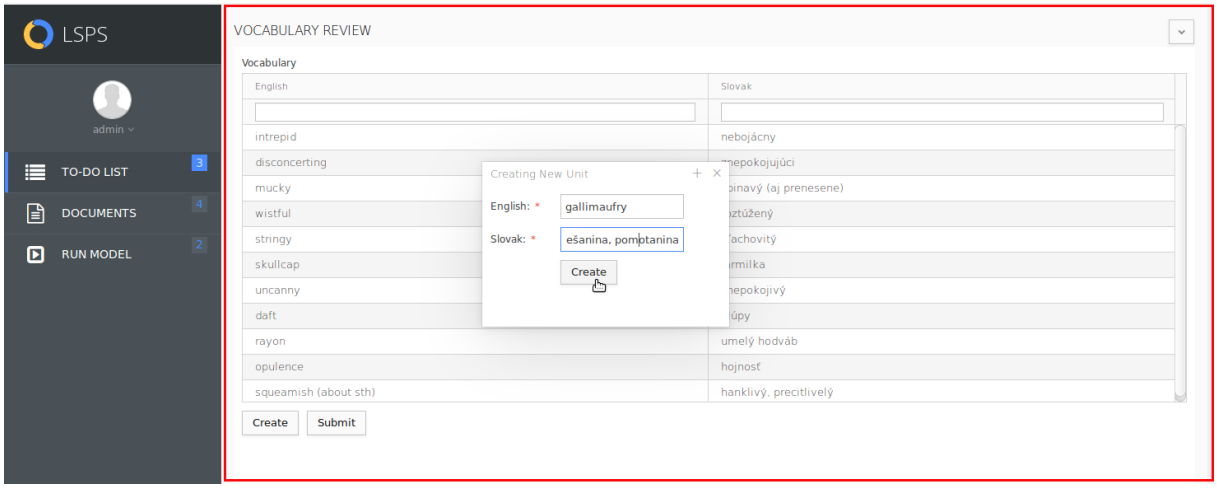



**Figure 1.2 Rendered to-do with the form**

When the user opens a document or to-do:

1. A context level, called *screen context*, is created.

   The *screen context* isolates the form data from the data of the model instance, which exists on the *base* level: the form context exists on another level, called the *screen level*, which overlays the model instance execution level. With the contexts separated, the user can change the data in the form without changing the data in the model instance. Only when the user submits or persists the form is the data from the form context merged into the data of the model instance context.

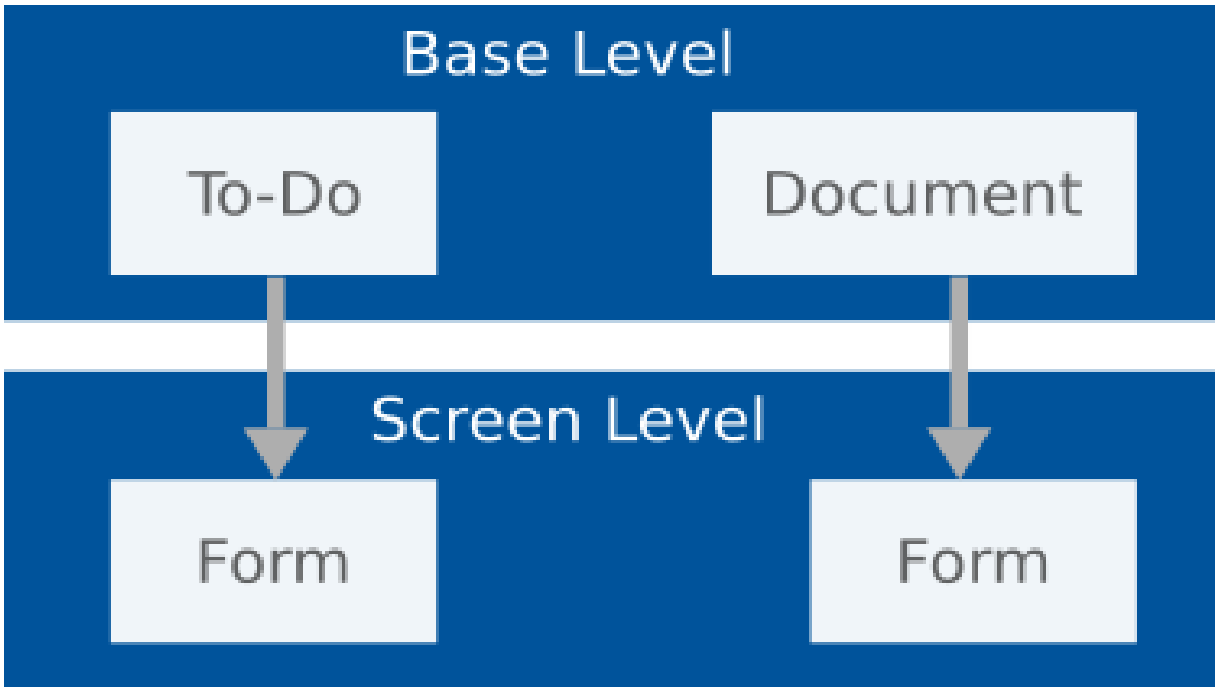



**Figure 1.3 Schema with execution levels of a model instance, and a to-do and form**

2. The data from the model context required in the form is copied to the *screen context*.

3. In the *screen context*, the form variables are initialized, the form is built and rendered as web content.

   The form is build based on the form definition, which holds a tree of form components, such as, input fields, radio lists, etc. Each component produces events whenever it records some action. The events can be caught by listeners which are attached to the component. When a listener catches its event, the event is queued and processed in the event-processing cycle. The way the event is processed is defined by the listener.

For example, if the user clicks a button, the button fires an event, called the *ActionEvent*. The event is caught by the *ActionListener* attached to the button component and sent to the event-processing cycle. Based on the listener properties, it can cause refresh of form components, persisting of data, submit of the to-do or document, etc.

You can `create a custom form component` that will be available in the form-editor palette just like standard components.

# Chapter 2

# Event Processing

When the user or the system performs an action on a form component, the form component produces an event of a certain type. The type is defined by the action that caused it.

If the form component has a listener for the type of the event, the event triggers the event-processing cycle (An exception are the non-immediate ValueChangeEvents. These events are queued and processed later when an immediate event appears.)

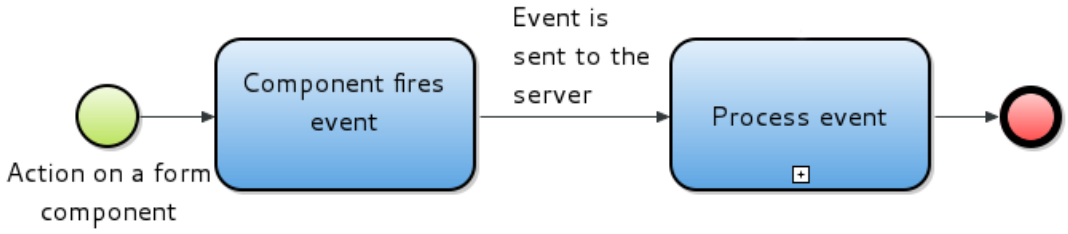



**Figure 2.1 Form event firing**

An event is processed in the **event-processing cycle**, or request-response cycle. and the exact way it is processed is defined by its listeners.

Here is a simplified descriptions of the cycle (we omit here the case when there are non-immediate ValueChange↩ Events in the queue):

1. An event is thrown.
2. All listeners catch the event.
3. All error messages in the form are cleared.
4. Event is processed based on the listener properties:
    - Internal values in the form, the values referenced by the bindings of form components, are updated.
    - Internal values are validated.
    - The handle expressions of listeners are executed.
    - Application events.
    - View Models are merged or cleared.
    - The Navigation expression is calculated.

5. Components are refreshed: If a new event arises as a result, the whole processing is repeated. If no event arises, the processing continues to listener actions:

    - **Submit**: defines if the document or to-do is submitted as part of the event handling;
    - **Persist**: if enabled, the relevant data in the processed components is persisted as part of the event handling; unlike on submit action, if the form is used by a to-do, the to-do does not become finished; if part of a document, the document remains open; persist is performed after the merge to the screen level before the transaction is committed.
    - **Save action**: defines if the to-do or document is saved;
    - **Navigation**: defines the location where to navigate after the event is processed; you can navigate to a to-do or document, URL, custom application page (refer to the descriptions of the data types defined in the human.navigation.datatypes resource in the Standard Library)

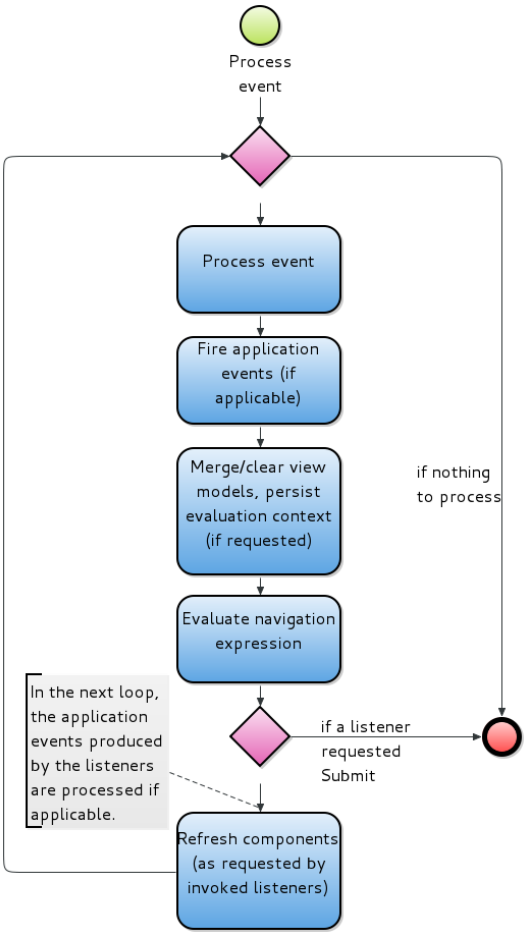



**Figure 2.2 Event processing cycle**

The event processing and the listener actions are executed within a single database transaction so that if the processing fails at any stage, the database is rolled back to a consistent state.

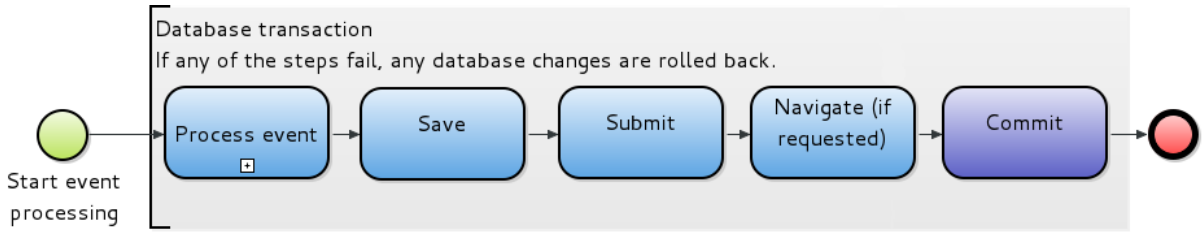



**Figure 2.3 Database transaction on event processing and listener actions**

You can find a detailed schema of the cycle here.

Events Listeners

## 2.1 Events

When the user performs an action on a form component, such as, changing a value, clicking a button, moving a widget, etc. the component produces an event of a type: the type depends on the action. The produced event holds data about the actions.

If the component has a listener that is listening for an event of the given type, it catches the event. These caught events are add to the event queue. Most events, called immediate events, trigger the request-response cycle and hence are processed immediately along with all the events in the event queue. If the event is not immediate, it waits in the queue until an intermediate event enters the queue and triggers the request-response cycle.

Some event types are immediate by default; other event types are non-immediate and can be set as immediate explicitly if required.

All queued events are processed simultaneously: the way the events are handled and the actions they cause are defined by the listeners that caught them; for example, a listener can request refreshing of form components, data validation, etc.

Events have their properties stored in their fields: for example, the ChartClickEvent contains fields with details on where exactly on the chart the click occurred so the listener can define different actions depending on the location where the user clicked.

Form components produce events of different types depending on the action that created the event. Here is a brief summary:

- All form components
  - **InitEvent** produced when a component is initialized (a form or a previously hidden component is shown)
  - **ApplicationEvent** produced by any event as part of the event processing
    Application events can be fired by an event of any type and is broadcast to all form components.
- All input components
  - **ValueChangeEvent** produced when the user changes a value in an input component

  The time when the change-value event is processed depends on the setting of immediate mode.
- Button and Action Links
  - **ActionEvent** produced when the user clicks an Action Component

- File Download
  - **FileDownloadEvent** produced when the user clicks a file-download button of a file-download component
- File Upload
  - **FileUploadEvent** produced on file-upload completion
- Text Box and Text Area
  - **AsynchronousValueChangeEvent** produced on every key stroke
- All Chart components
  - **ChartClickEvent** produced when the user clicks a chart
- Widget
  - **WidgetChangeEvent** produced when the user adds a widget to the dashboard, resizes, moves, or hides a widget
- Calendar
  - **CalendarCreate event** produced when the user selects a time period by clicking and dragging
  - **CalendarEdit event** produced when the user clicks a calendar entry
  - **CalendarReschedule event** produced when the user drags a calendar entry
- Geolocator
  - **Geolocation event** produced when the Geolocator component acquires user's location
- Map Display
  - **MapClickedEvent** produced when the user clicks on the map
  - **MarkerClickEvent** produced when the user clicks on a map marker
  - **MarkerDraggedEvent** produced when the user drags a map marker
- Table of the Paged Type
  - **TablePageSizeChangeEvent** produced when the user changes the size of the table page
- Tree
  - **TreeEvent** produced when the user expands or collapses a tree item, be it in a tree or a tree table component
- Popup
  - **PopupCloseRequestEvent** produced when the user clicks the close button in a popup
- Context menu
  - **MenuEvent** produced when the user clicks an item in the context menu
- Grid Column
  - **RendererClickEvent** produced when the user clicks an item in the Grid Column

### 2.1.1 InitEvent

The InitEvent is fired when a component is displayed, be it for the first time or after it was saved or when it became visible.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | UIComponent | Component that produced the event |
| isFirstLoad | Boolean | true for a component displayed for the first time (the property is true also when a hidden component is displayed for the first time) |
| isFirstLoadAfterSave | Boolean | true for a component displayed for the first time or for the first time after save |

### 2.1.2 ValueChangeEvent

The *ValueChangeEvent* is fired when the user changes a value in an input component or when they select an option.

When the ValueChangeEvent is actually processes and takes effect depends on whether its input component is in immediate mode, which is defined by the Immediate property of the component.

When the component has the Immediate property set to `true`, the event is processed as follows:

- If on a Text Box or Text Area, the event is produced and handled as defined by the event-processing cycle, when the component loses focus, that is, after you had edited the value of the component and clicked out of the component or pressed **Enter**.
- If on Lists, Check and Combo Boxes, the event is produced and handled as defined by the event-processing cycle, when an item is selected.

If the property is set to `false`, then the event is produced at the same time but remains unhandled, that is, the request-response cycle is not triggered: Instead it is added to an event queue. The queued events are processed on the next run of the request-response cycle, that is, when a ValueChangeEvent with activated immediate mode or an event of another type is handled. This typically happens when the user confirms the value changes by clicking a button with an ActionListener.

When a ValueChangeEvent occurs, the new value undergoes conversion validation. On conversion validation, the server checks, if the new value has the correct form. If this validation fails, the component is marked as invalid and the ValueChangeEvent, any ApplicationEvents and ActionEvents are not processed any further. Otherwise, the events are processed.

Note that when you change a value in an input component, the value change is applied on the bound entity, such as, a variable, even if no listener for the value change event exists. To prevent this behavior, use nested contexts.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | UIComponent | Component that produced the event |
| oldValue | Object | Object with the value before change |
| newValue | Object | Object with the value after change |

### 2.1.3 AsynchronousValueChangeEvent

The AsynchronousValueChangeEvent is fired and processed immediately when the user changes a value in a Text Box or Text Area component.

The event is fired asynchronously: a new one can be fired even if the old one is still being processed.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | UIComponent | Component that produced the event |
| text | String | String with the input |

### 2.1.4 ActionEvent

The ActionEvent is fired when an action or image component is clicked, when a file upload is started, and when ENTER is pressed on the TextBox component.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | UIComponent | Component that produced the event |

### 2.1.5 FileDownloadEvent

The FileDownloadEvent is fired when the File Download component is clicked.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | UIComponent | Component that produced the event |

### 2.1.6 FileUploadEvent

The FileUploadEvent is fired when file uploading finishes. Note that on upload start, an ActionEvent is fired.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | UIComponent | Component that produced the event |
| uploadedFiles | Set<Files> | Set of uploaded files |
| errorMessage | String | Error message returned if the upload fails |

### 2.1.7 ChartClickEvent

The ChartClickEvent is fired when a data point of a chart element is clicked.

The event contains the data series, key, and values of the clicked chart location. This allows you to implement, for example, data drill-down.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | UIComponent | Component that produced the event |
| series | String | Label of data series that was clicked |

| Property | Data Type | Description |
|---|---|---|
| key | Object | Key value for the data point |
| value | Decimal | First value defining the data point |
| value2 | Decimal | Second value defining the data point |
| payload | Object | Payload of the data point |

### 2.1.8 WidgetChangeEvent

The WidgetChangeEvent is fired when a widget is added, removed, resized, moved, or hidden.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | UIComponent | widget component that produced the event |
| widgetId | String | ID of the widget that produced the event set in the Widget ID parameter |
| configuration | WidgetConfiguration | Widget configuration with details about the widget visualization properties |

### 2.1.9 CalendarCreateEvent

The CalendarCreateEvent is fired by a calendar component when the user clicks and drags over a period in a calendar. The event holds the selection data as its payload and the data can be used to create a new calendar entry.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | Calendar | Calendar component that produced the event |
| from | Date | Start date of the selected period |
| to | Date | End date of the selected period |
| allDay | Boolean | If the entry is a whole-day event (if selected across days, the entry is an allDay entry; if the selected area is across hours, the allDay property is false and the exact hours are included) |

### 2.1.10 CalendarEditEvent

The CalendarEditEvent is fired by a calendar component when a calendar entry is clicked. Note that the event has as its payload the business object of the calendar entry that was clicked.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | Calendar | Calendar component that produced the event |
| data | Object | Business object of the calendar entry that was clicked |

### 2.1.11 CalendarRescheduleEvent

The CalendarRescheduleEvent is fired by the calendar component when a calendar entry is dragged-and-dropped to a different date. Note that the event has as its payload the business object of the rescheduled calendar entry.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | Calendar | Calendar component that produced the event |
| from | Date | Start date of the new period |
| to | Date | End date of the new period |
| data | Object | Business object of the calendar entry that was rescheduled |

### 2.1.12 GeolocationEvent

The GeolocationEvent is fired by the Geolocator component after the component has acquired the geographical position of the user or when the request for location times out.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | Geolocator | Geolocator component that produced the event |
| position | Geoposition | Data on position including latitude and longitude, speed, altitude, etc. |
| failure | GeolocatorError | Type of error if locating failed |

### 2.1.13 MapClickedEvent

The MapClickedEvent is fired by the Map Display component when the user clicks into the map.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | MapDisplay | Map Display component that produced the event |
| point | GeographicCoordinate | point that was clicked |

### 2.1.14 MarkerClickedEvent

The MarkerClickedEvent is fired by the Map Display component when the user clicks a marker.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | MapDisplay | Map Display component that produced the event |
| makerData | Object | underlying marker business object (the respective object of the set defined in the Markers property of the Map Display component) |

### 2.1.15 MarkerDraggedEvent

The MarkerDraggedEvent is fired by the Map Display component when the user drag-and-drops a marker.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | MapDisplay | Map Display component that produced the event |
| makerData | Object | underlying marker business object (the respective object of the set defined in the Markers property of the Map Display component) |
| newLocation | GeographicCoordinate | new marker position after dropped |

### 2.1.16 MenuEvent

The MenuEvent is fired when the user clicks an item in the context menu.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | UIComponent | Component with the context menu |
| id | Object | id of the clicked MenuItem object |

### 2.1.17 TreeEvent

The TreeEvent is fired when the user expands or collapses a tree item, be it in a tree or a tree table component.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | UIComponent | Parent Tree or TreeTable of the treeItem |
| treeItem | TreeItem | TreeItem object that produced the event |

### 2.1.18 TablePageSizeChangeEvent

The TablePageSizeChangeEvent is fired when the user changes the size of a paged table.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | Table | Parent Table |
| pageSize | Integer | page size after the change |

### 2.1.19 PopupCloseRequestEvent

The PopupCloseRequestEvent is fired when the user clicks the close button in the caption of a popup component.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | Popup | Popup that requested close |

### 2.1.20 RendererClickEvent

The RendererClickEvent is fired when the user clicks an item that uses the renderer in a Grid column.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| source | GridCellRenderer | The renderer of the clicked Grid cell |
| rowObject | Object | The object of the clicked row |

### 2.1.21 ApplicationEvent

The ApplicationEvent can be produced by any listener: it is the only event, the user can define out-of-the-box↩ : When a listener catches its event, it can fire an ApplicationEvent as part of its logic. The event can be caught by any ApplicationEvent listener in the form, and that including listeners on any hidden components and reused forms.

**Table with event properties**

| Property | Data Type | Description |
|---|---|---|
| eventName | String | Custom name of the ApplicationEvent |
| payload | Object | Custom event data |

For example, let's assume a form for placing orders. It contains multiple nested reusable forms: one contains customer details, another ordered items, and the last one invoicing details. All three reusable forms need to be validated before the order can be placed. The Place Order button located in the parent form, will fire an Application↩ Event that will be handled by ApplicationEventListeners on the reusable forms. The listeners will trigger validation and any other actions needed as part of the event handling.

To distinguish application events, use the event fields.

> For example, let us assume an application form with personal details, application details has Reject and Accept buttons, and a Reusable form with additional data on reject. If the user rejects the application, the provided data must be validated and the reject form must appear and they must provide the relevant rejection data. If they accept the application, no comment is required. However, all other components must be still validated.
>
> The underlying form will define the two buttons with listeners that throw the OnSubmit Application↩ Event. However, the Reject button will throw an ApplicationEvent with the REJECT value as its payload field, while the Accept button throws an ApplicationEvent with the ACCEPT value as its payload field. The input component for the comment must therefore define an ApplicationListener that will check that the ApplicationEvent payload is ACCEPT or if the payload is REJECT, it will check that the comment component is not empty.

## 2.2 Listeners

Listeners define how an event of a particular type from a particular component is processed in the event-processing cycle when it occurs.

When the component produces an event of the expected type, the listener catches the event and sends it to the event-processing cycle. The event is processed as defined by the properties of the listener.

Listeners defines the following:

- Basic properties determine what events they catch and how they are handled:
  - **Listener type**: the type of event the listener handles
    A listener catches only events with the same type as the listener type. You can use the Generic Listener if you want it to handle any event on the component.
  - **Refresh components**: comma-separated list of component IDs that are refreshed, that is, their content is recounted and the components are re-rendered.
  - **Validators**: validation expression
    The event is handled only if the validation expressions are `true`.
  - **Data validation**: validation expressions that validate against Constraints
    When using data constraints, the expression should include a `validate()` call that will trigger the constraint validation.
  - **Execute if other validations failed**: select to ignore failed validation on other listeners (refer to Validating UI Form Data).
  - **Execute even if invalid components**: enforces execution of the listener logic even if the form contains incorrect data, such as, "aaa" instead of an integer, to allow operations as refresh or reset on click.
  - **Handle expression**: expression that is executed
    The expression represents the action the listener performs to handle the event.
  - **Event identifier**: identifier of the received event
    You can use the identifier in the Handle expression to acquire the event fields with event details, such as the component that produced the event, uploaded files, etc.; refer to Filtering Events on Listeners.
- Advanced properties define the context of the execution and actions as well as handling pre-condition:
  - **Process components**: events of the components that are processed when this listener event is processed (refer to Ignoring Queued Non-Immediate ValueChangeEvents)
  - **Execution context**: execution context of the listener
    All expressions defined in the listener properties, such as, its precondition, validation, etc. are evaluated in this context level.
    - default: the context level of the component with the listener
    - top level: the screen context of the form
    - component: ID of the component which represents the target execution context
      **Important:** The execution context setting does not influence how View Models are merged: this is set by the merge type property on the View Model.
  - **Execute only if visible components**: comma-separated list of component IDs; the event handling takes place only if all specified components are displayed in the form.
  - **Clear/merge view model components**: IDs of view models (their contexts are cleared or merged to the target execution level).
  - **View model init**: expression executed right after the merge or clear of view model components
    The expression serves as a hook after the merge-clear phase.
  - **Precondition**: pre-condition for event handling
    If the pre-condition evaluates to false the event is not handled.
- Actions properties define what actions are performed on the to-do or document after the event is processed:
  - **Submit**: defines if the document or to-do is submitted as part of the event handling;
    If enabled, the relevant data in the components is persisted and the to-do becomes finished or the document closes.

  - **Persist**: if enabled, the relevant data in the processed components is persisted as part of the event handling; unlike on submit action, if the form is used by a to-do, the to-do does not become finished; if part of a document, the document remains open; persist is performed after the merge to the screen level before the transaction is committed.

    You can define a persist action, which is performed immediately after the persist.

  - **Save action**: defines if the to-do or document is saved;

    The save action saves the current state of the to-do or document for later editing, which includes saving the provided data; it is identical to clicking the Save button on a to-do or document in the LSPS Application User Interface; the save action does not persist the data; therefore make sure to define the Persist property if required. If a newer version of the saved to-do or document is available, the application displays a notification.

    > **Note:** A saved document is persisted in the system database as the *SavedDocument* record. If saved repeatedly, the same record is overwritten; hence only the last saved document version is available. On submit, the persisted document is removed. For more information on the record type and related functions, refer to the Standard Library documentation.

  - **Navigation**: defines the location where to navigate after the event is processed; you can navigate to a to-do or document, URL, custom application page (refer to the descriptions of data types defined in the human.navigation.datatypes resource in the Standard Library)

    Navigation takes place after persisting, which allows you to use the persisted data in the navigation expression.

    > **Important:** If the form is used in documents, the form navigation is overridden with the document navigation.

  - **Fire application event**: ApplicationEvent the listener produces

- Expression tab allow you to define the entire listener as an expression: when you select the **Listener is defined by expression** option, the properties defined on the other tabs are reflected in the generated expression

# Chapter 3

# Creating ui::Forms

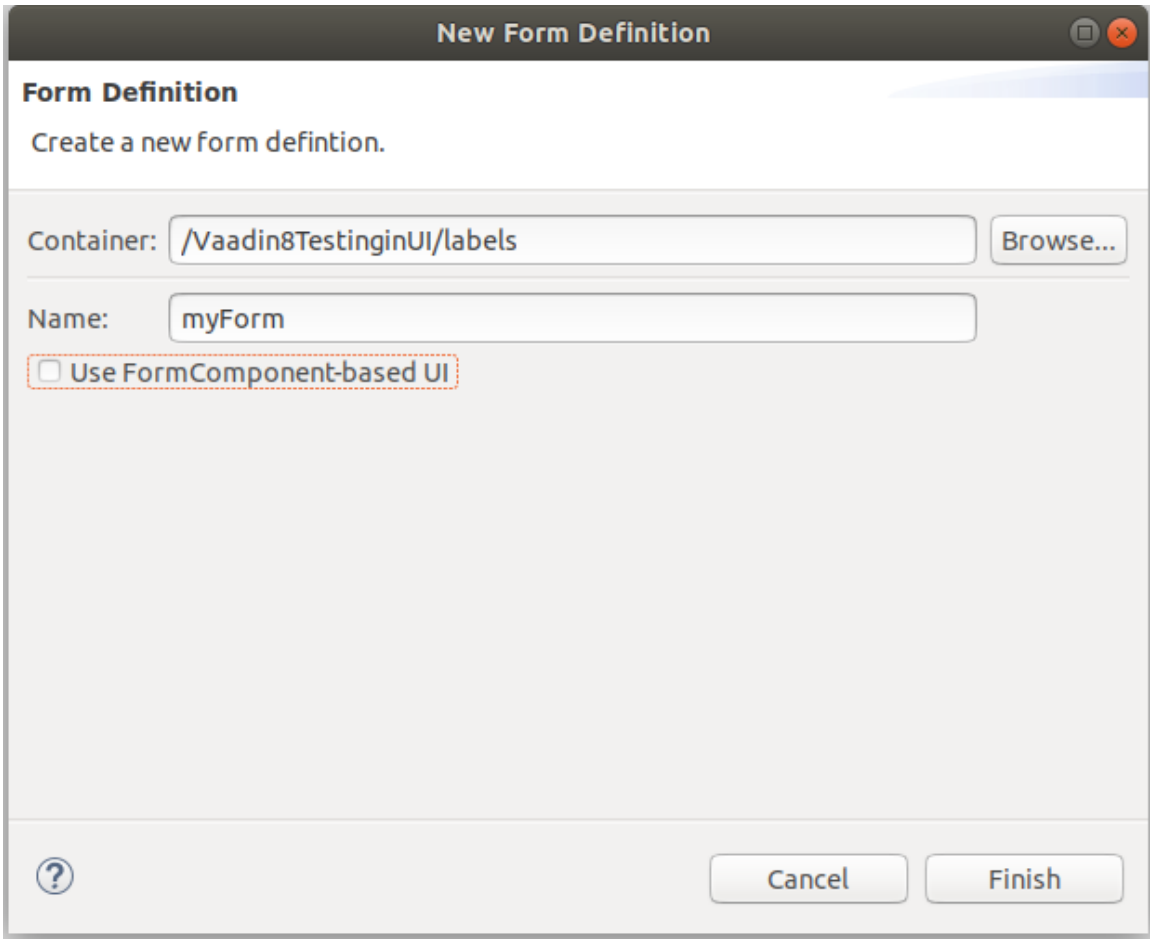
When creating ui::forms, you typically do the following:

- Create a form definition.
- Create form parameters and variables.
- Design the form content.

## 3.1 Creating ui::Form Definition

To create a UI form definition, do the following:

1. In GO-BPMN Explorer, right-click a module.
2. Go to **New** > **Form Defining**.
3. In the popup, enter the name of the form.
4. Unselect the *Use FormComponent-based UI*.




**Note:** If the *Use FormComponent-based UI* option is selected, `a form based on the forms module` will be created.

## 3.2 Defining ui::Form Parameters

Forms can require parameters.

To define form parameters, do the following:

1. Open the form in the form editor: double-click it in the GO-BPMN Explorer.
2. In the Outline view, right-click the form and select **New** > **Parameter**.

   Alternatively, you can open the form properties in the Properties view and click Add on the Parameters tab.
3. In the Property view, define the parameter.

The parameters are required: when creating the form, the call must provide arguments for the parameters.

```
 //example call that creates a parametric form:
applicationForm(user -> admin, requestedHardware -> Hardware.ssd)
```

## 3.3 Defining ui::Form Variables

Forms can define form variables, which are accessible from within the form. They allow you to separate presentation data and business data: You should prefer form variables over global variables whenever possible.

Form variables are initialized when the form is displayed along with the InitEvent being fired.

To define form variables do the following:

1. Open the form definition.
2. In the Outline view, right-click the form and go to **New** > **Variable**.
3. In the displayed Properties view, define the variable properties.

## 3.4 Designing ui::Form Content

To create the content of your form, open the form definition file (double-click it in the *GO-BPMN Explorer*). Then either click the required component in the palette and then click into the canvas, or right-click the canvas, go to *Insert Component* and select the component from the context menu.

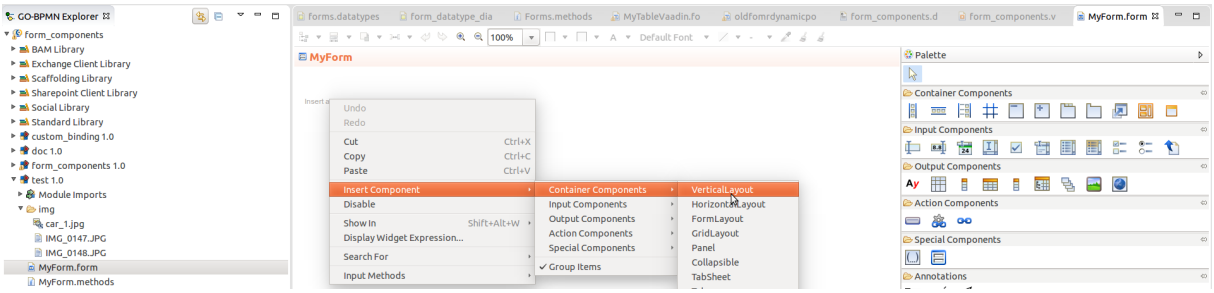



**Figure 3.1 Inserting a component into the form from the context menu**

> **Note:** You can create forms also with the `dynamic-gui` functions, such as *createAndAdd()*. However, these functions are not maintained and will not be enhanced.

Once you have created your form or while doing so, you can do the following:

- define basic behavior of the form and its components
- define validation
- reuse form components
- modify presentation properties of components
- define mobile forms

### 3.4.1 Inserting a Parent Component in ui::Forms

You can select one or multiple form Components and wrap them with another components in the Form graphical editor.

To insert such a parent form component over another component, do the following:

1. In the Form editor, right-click the component.
2. In the context menu, go to Insert Parent and select the component to use as the wrapper component.

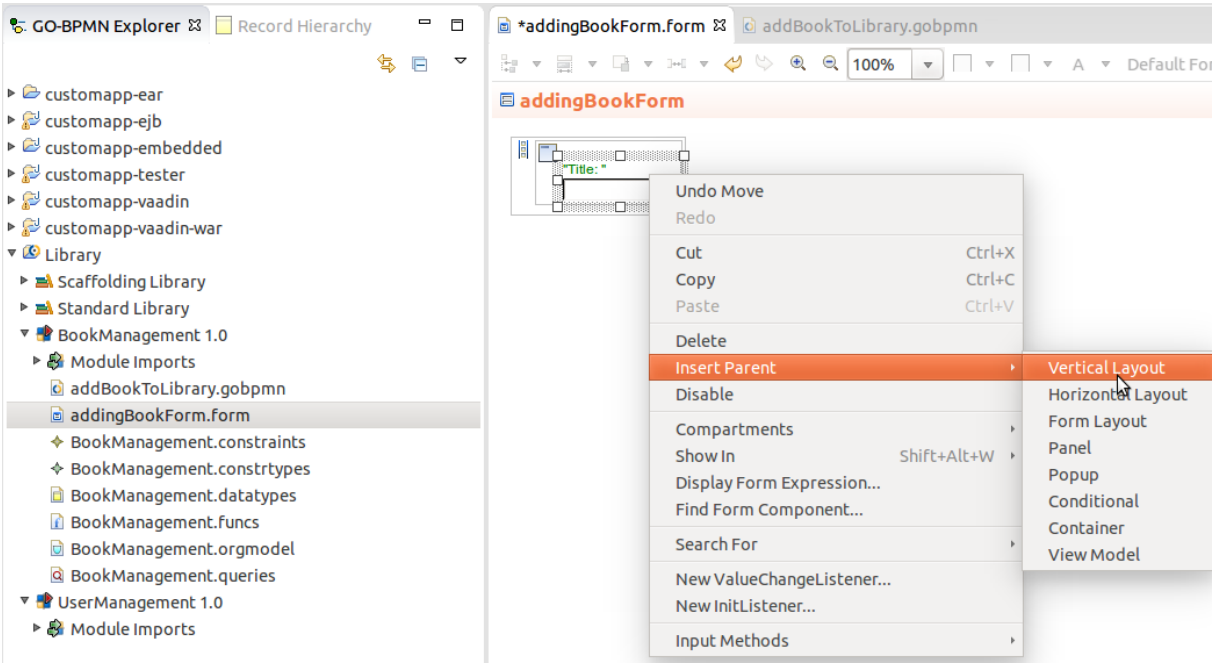



**Figure 3.2 Wrapping component in a layout component**

### 3.4.2 Deleting a Parent Component in ui::Forms

To delete a only a parent component and preserve its child components, right-click the parent component and click **Shift Children Up**.

Note that the option is only available if the children can be accommodated in the form after the deletion; for example, it is not possible to delete a Vertical Layout with multiple child components if it has a Panel component as its parent.

### 3.4.3 Previewing a ui::Form

While creating a form, you can display its preview in the browser.

Make sure, PDS is connected to an LSPS Server: When displaying a form preview, the server runs a persisted instance of the model that contains the form definition (global variables are initialized, the Form parameters are initialized, the form screen context is created). The model instance is persisted.

To display preview of your form in your web browser, do the following:

1. In the GO-BPMN Explorer, right-click the form.
2. Select Run As -> Form Preview and on the displayed application page in your browser, log in to the application.

For parametric forms, add the argument to the form preview configuration.

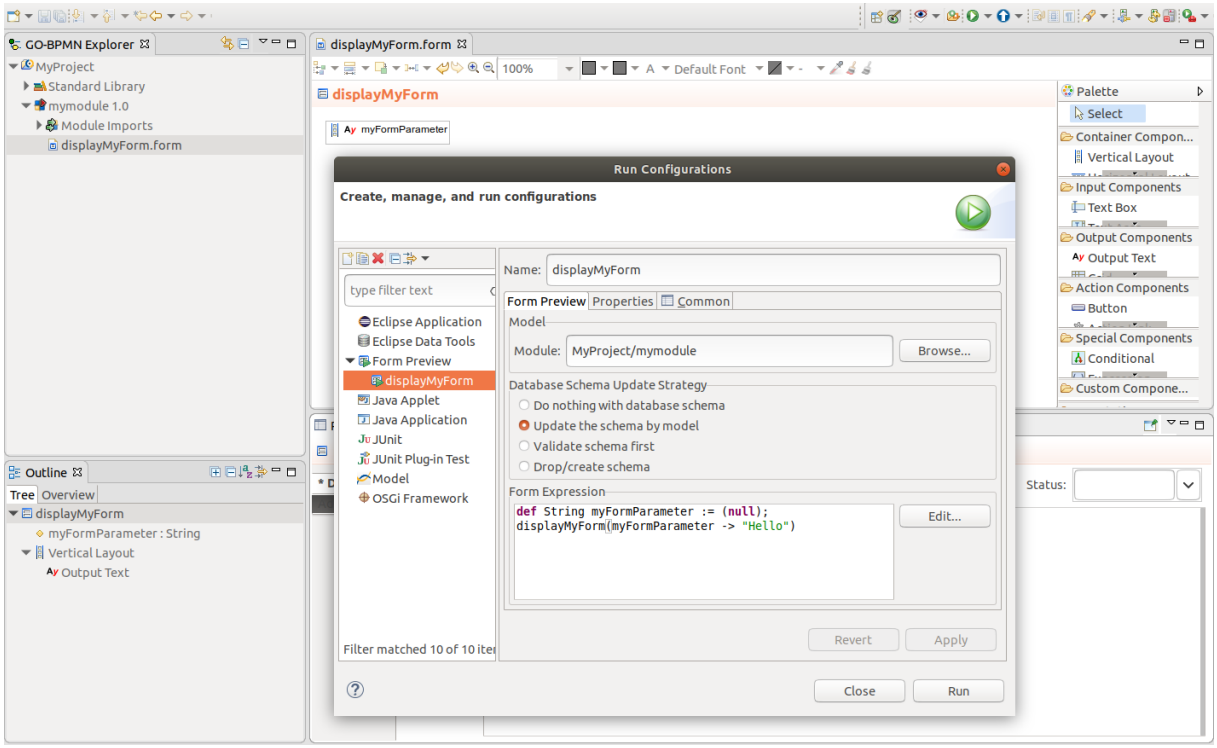



**Figure 3.3 Form preview configuration of a parametric form**

In the Management perspective, you can delete any model instances that you triggered as form previews in the Model Instances view: right-click anywhere into the view and click Remove All Form Preview Model Instances.

### 3.4.4 Displaying the ui::Form Source Code

Since forms are defined as functions and their components as variables of the respective record type on the level of the Expression Language, you can display their source in the Expression editor:

1. In the Form editor, right-click any form component.
2. On the context menu, select Display Form Expression.

### 3.4.5 Searching for a ui::Form Component

If an error on a form component occurs during runtime, the server returns an error message with the modeling ID of the form component. To find the form component, do the following:

1. Go to Search > Find Form Component.
2. In the displayed search dialog, enter the modeling ID.

### 3.4.6 Defining a Context Menu in ui::Forms

Every form component can define its context menu. The menu and its content is defined as a component property. When the user right-clicks the component, the context menu with its items is displayed. The user can then click a context menu item. On click, a MenuEvent is produced. The MenuEvent has the id of the clicked menu item as its payload so the form can define the MenuListener that will trigger the respective actions.

The context menu can be defined as static or dynamic: A static context menu is calculated only once before the form is rendered, while a dynamic context menu is recalculated on every right-click. This might cause performance issues since it could require accesses to server on every right-click.

If you define both a static and a dynamic context menu on right-click, the displayed context menu will contain the static menu items on top and the dynamic menu items below.

To define a context menu on a component, do the following:

1. Select the component in the form definition.
2. In the Properties view, open the Context Menu tab.
3. On the tab define either a static context menu or a dynamic context menu.

```
[new MenuItem(
            caption -> "Open the Detail",
            htmlClass -> "contextmenu",
            id -> 0,
            submenu -> [new MenuItem(
                    caption -> "Open in a New Tab",
                    htmlClass -> "contextmenu",
                    id -> 1),
                    new MenuItem(
                    caption -> "Open in This Tab",
                    htmlClass -> "contextmenu",
                    id -> 2)
                    ]
            )
]
```

4. Define the action that should occur when the item is clicked in a MenuListener.

   **Example Handle expression on a MenuListener**

```
_event.id == 1 ?  (MENUITEMCLICK.content := { -> "Menu item with id 1 has been clicked." })
```

### 3.4.7 Defining a Listener in a ui::Form

You can define listeners on any form component; however, every component type allows only listeners for events it can actually produce; for example, a Button component can define only an Action listener, which is fired when the user clicks a button. It cannot define a ValueChange listener since there is no value to be changed on a Button. On the other hand, a component might produce various events: A button produces just like all components an InitEvent when it is created: An Action listener on the Button will ignore the event: it catches only the events of a particular type on its component.

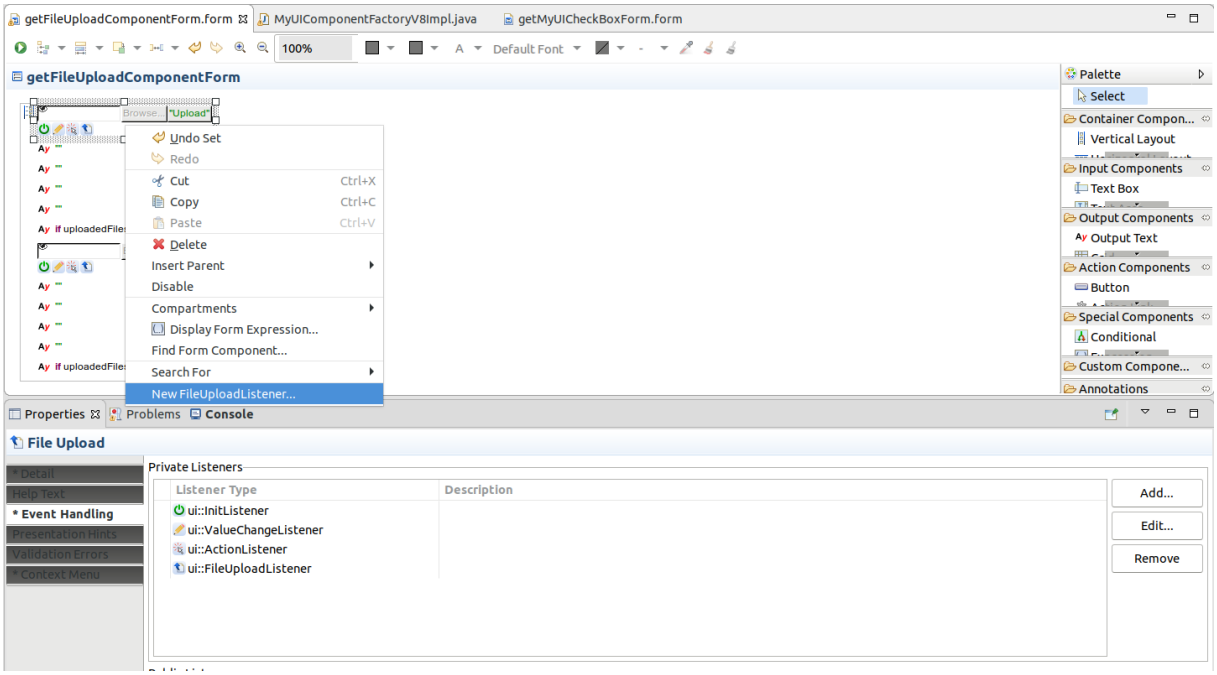
**Note:** To listen for an event on another component, you can do one of the following:

- To listen for events from child reusable forms or to listen on events from parent forms, use Container interface with public listeners and registration points.
- Produce an application event.

To create a listener on a form component:

1. In the Form editor with your form definition, right-click the form component.

2. In the context menu, select the listener type.




3. In the Add Listener dialog, define the listener properties:

   - You can do so by gradually going through the *Basic*, *Advanced*, and *Actions* tabs

   - Alternatively, you can go to the *Expression* tab and define your listener as an expression: if you have previously defined some properties on the *Basic*, *Advanced*, and *Actions* tabs, they are these are reflected in the listener expression.

Note that the event caught by the listener is available to expressions on all tabs with the exception of the *Handle expression*; however, the type of the event is not recognized so you might need to cast it to its type, for example, `_event.cast(InitEvent)`.

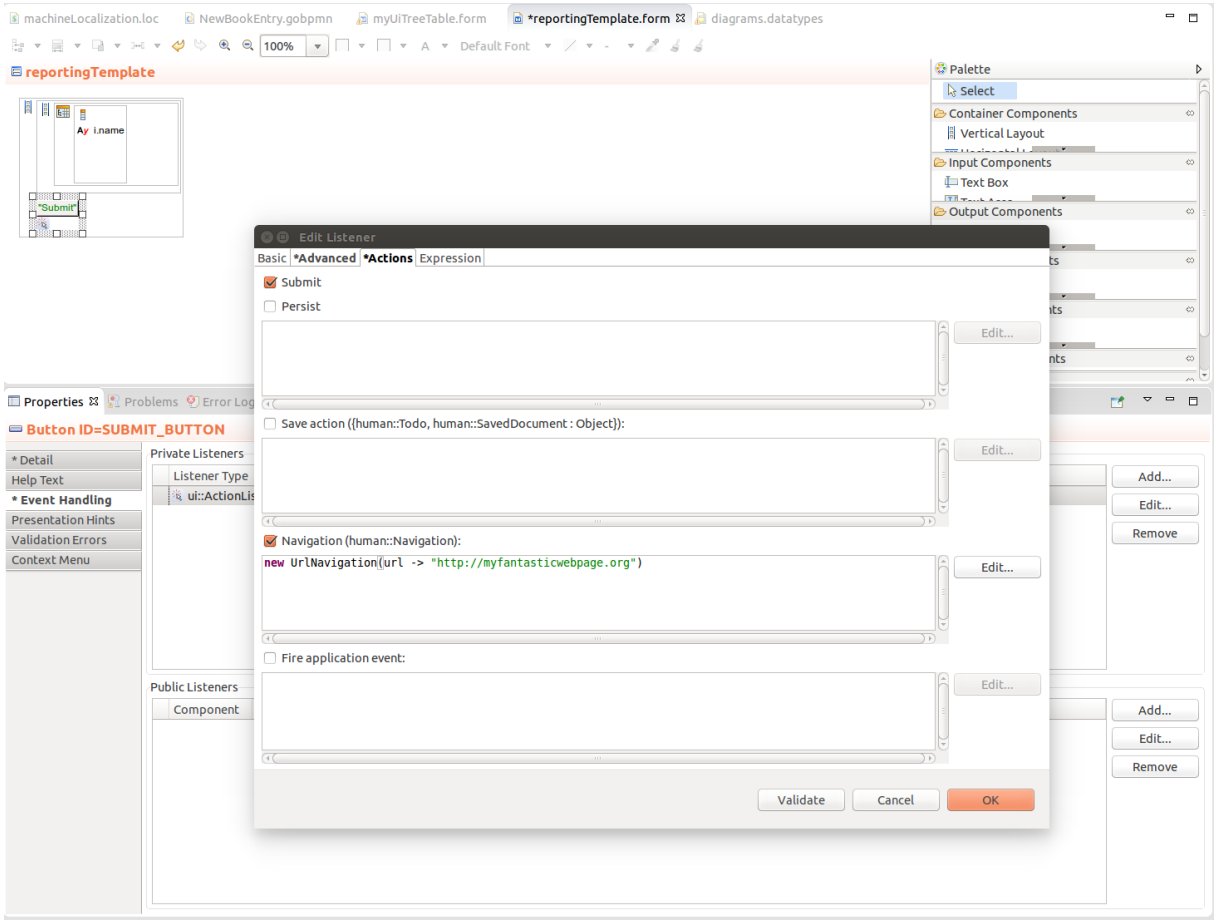



**Figure 3.4 Defining properties of an ActionEvent listener on a submit Button component**

You can display listener properties by double-clicking the listener in the form or on the *Event Handling* tab of the component's Properties view.

#### 3.4.7.1 Disabling a Listener in ui::Forms

To disable a listener on design time, open its properties and select *Listener is disabled* on the Basic tab.

#### 3.4.7.2 Filtering Events on Listeners in ui::Forms

To filter out an event so the listener does not catch it, use one of the following listener properties on the Advanced tab:

- **Execute only if visible components**: The event is processed by the listener only if the component that produced the event is visible.
- **Precondition**: The event is only processed if the precondition evaluates to `true`.

```
if ((_event as ApplicationEvent).payload as Integer) = 1
 then true
 else false
end
```

- **Event name** (available only for ApplicationEventListeners): An application event is only handled by the listener, if the name of the event matches the defined event name.

  **Note:** An event might not be processed also if the validation process fails.

#### 3.4.7.2.1 Ignoring Queued Non-Immediate ValueChangeEvents

Sometimes you want to ignore queued events from other components, for example, in forms with multiple tabs, you want to ignore events on tabs that are not focused at the given moment, or you want to prevent validation on components while resetting the content of another component, etc.

To ignore events produced by other components when a particular event is processed, set the *Process component* property on the *Advanced* tab in your listener properties:

- **all**: any queued events are processed
- **this**: only events from the component with this listener are processed
- **components**: events from the listed components and the component with this listener are processed

For example, let's assume a table with column A and column B. Both columns contain input components that are not-immediate and buttons used to submit the values from the given column. When the user changes values, ValueChangeEvents from both columns are kept in the event queue. Then the user clicks the submit button in column A. The value change listener must define as its Process component only column A. If it defines as its Process component column B as well, all the ValueChangeEvents will be processed.

#### 3.4.7.3 Refreshing a Component in ui::Forms

To refresh the content of form components, do the following:

1. Define IDs on the component you want to refresh.
2. Create a private listener on the component that should cause the refresh.
   (a) In the Properties view of the component, click the *Event Handling* tab.
   (b) In the *Private Listeners* section, click **Add**.
   (c) In the Add Listener view on the Basic tab, define the Listener properties:
      - Listener type: the type defines the event type the listener handles.
      - Refresh components: define IDs of the components that should be refreshed when the event is handled.

Alternatively, you can call the *refresh()* function from the handle expression (for example, `refresh([MYTABLE, MYPOPUP])`).

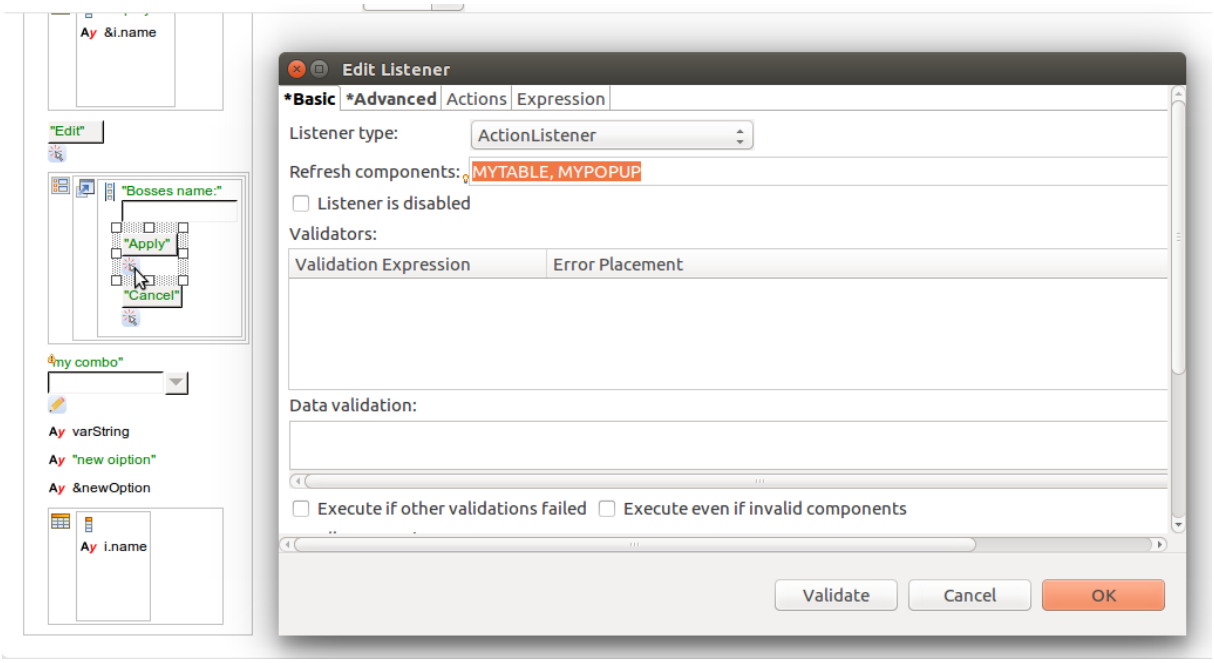



**Figure 3.5 Listener that refreshes multiple components**

#### 3.4.7.4 Persisting Data in ui::Forms

To persist the context data of a document or to-do, enable the Persist action on the *Action* tab of the respective listener or call the *persist()* function from its handle expression.

To perform an action immediately after persist, define it in the text area below the **Persist** checkbox,

The Persist action is performed after the merge to the screen level before the transaction is committed.

#### 3.4.7.5 Saving a To-Do or Document in ui::Forms

To save the state of the to-do or document for later editing, define the *Save* action on a listener. The action is identical to clicking the Save button on a to-do or document in the LSPS Application User Interface. Note that the save action does not persist the data, therefore make sure to activate the Persist action if required.

> **Note:** A saved document is persisted in the system database as a **SavedDocument** record. If saved repeatedly, the same record is overwritten. On submit, the persisted document is removed.

To process a saved to-do or document, define the *Save action* closure below the **Save** option: it has the saved to-do or document as its input parameter.

Note that by default, Save does not preserve Column states (width and collapse state). To save these, proceed as described in Saving Column Width and Collapsed State.

#### 3.4.7.5.1 Saving a To-Do or Document in a Custom Data Source

To store a saved to-do or document in your own data source, define the following:

- a shared Record that is in the 1:1 relationship to the human::SavedDocument record
- a document or a todo that will contain a form with a listener that will save the data as your SavedDocument subrecord in its Saving Action expression
- implementation that will recover the saved documents, for example, another dedicated Document with a navigation
- possibly the option for deletion of saved documents

### 3.4.7.6 Submitting a ui::Form

When the user submits the document or to-do, the data is saved, typically in the underlying DB via shared Records, and; in the case of documents, the model instance ceases to exist, and, in the case of to-dos, the user task finishes.

Note that the document will remain available in the list of documents since the list contains the types of documents, not their instances: Unlike To-Dos, documents availability does not depend on a Task; they are available as long as the definition of the document is on the server and their instances are created on request.

To define, when the document or to-do is submitted, do one of the following:

- on the component that should submit the document, create a listener of the required type and on the Action tab in the listener properties, select Submit.

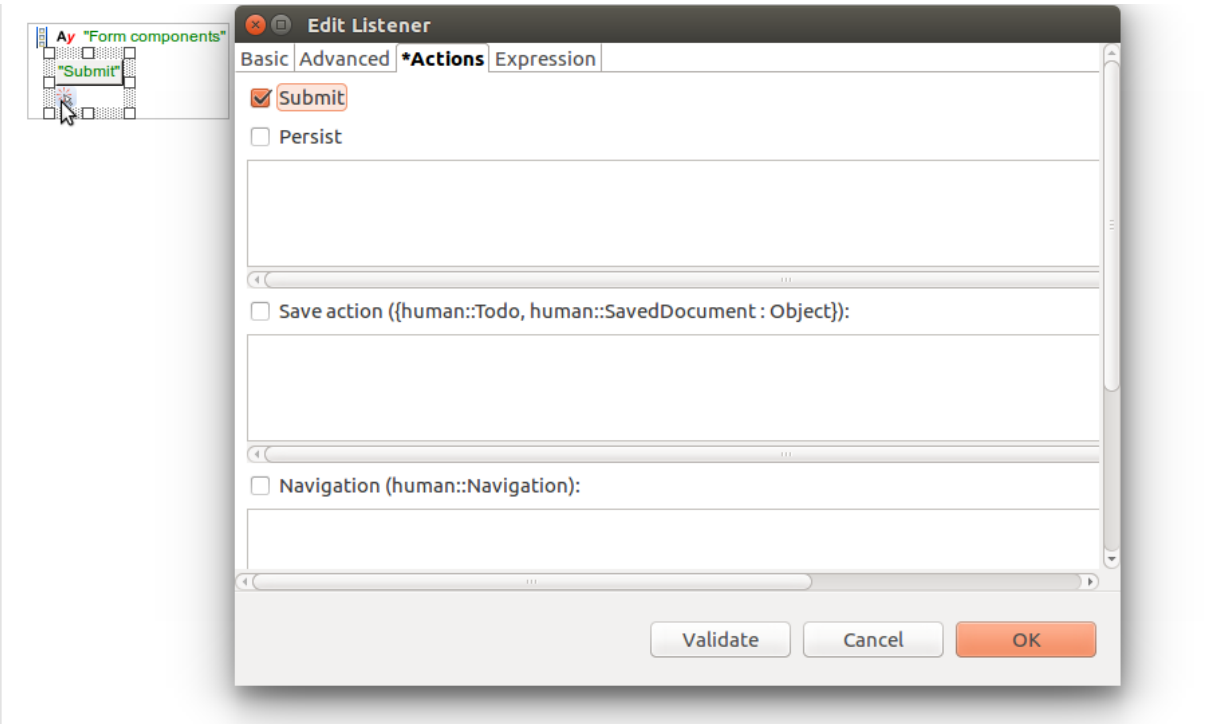



**Figure 3.6 Submit action on a click listener**

- on the component that should submit the document, create a listener of the required type and call the *requestSubmit()* or *requestSubmitAndNavigate()* from the Handle expression.

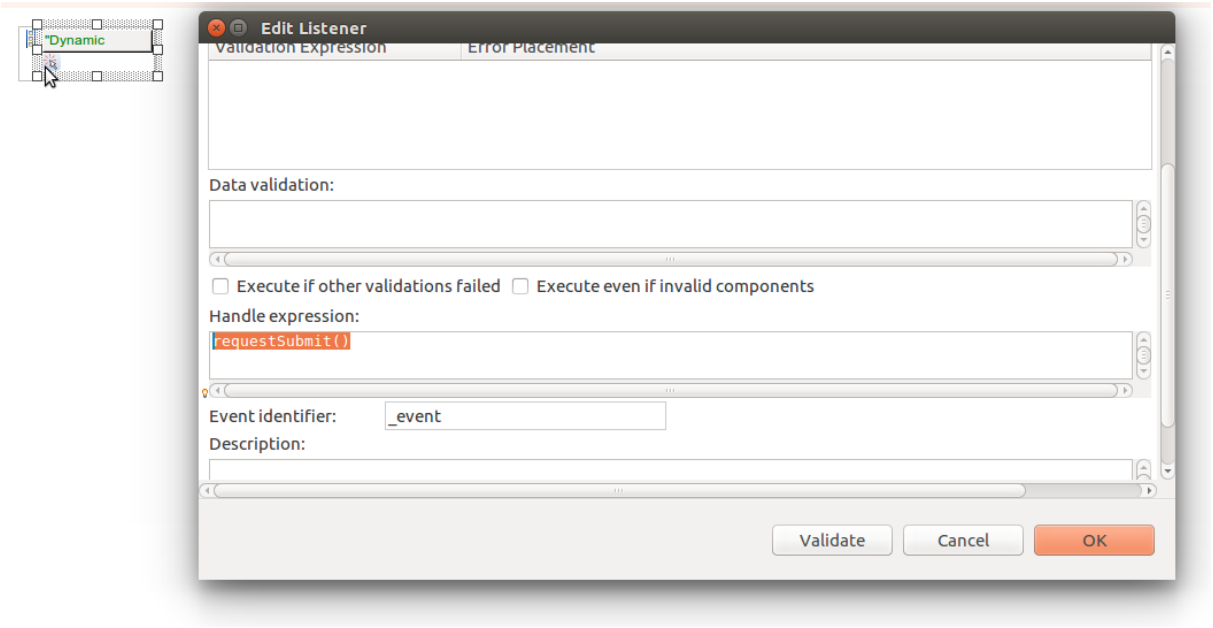



**Figure 3.7 Submit call from a listener handle expression**

#### 3.4.7.7 Navigating from a ui::Form

To navigate to a location when the form is submitted or saved, define the *Navigation* on the Action tab of the listener: you can navigate to a to-do or document, URL, custom application page (refer to the descriptions of data types defined in the human.navigation.datatypes resource in the Standard Library).

> **Note:** You can define a Navigation also on the document definition: This Navigation overrides the Navigation defined on the form (that is, if you define a button that submits the form, the form will navigate as defined in the Navigation closure on the document; not in the navigation of the submit component).

**Navigation expression**

```
 //redirect to the document Confirmation when the event is handled:
new DocumentNavigation(documentType -> confirmationDocument())
```

> **Note:** Navigation expression is evaluated right after persisting, which allows you to use the persisted data in the navigation expression. However, the action is taken only after the submit or save action is performed.

## 3.5 Validating UI Form Data

To validate a value during event processing of an event, define one of the following:

- To validate simple values in the form components, for example, a string in an input field, define a **validator**.
- To validate record values:
  - define a **data validation** expression to check if the value meets the record constraints
  - or define validation in the handle expression to control validation directly in the form.

Generally validation part of the event-processing cycle passes if:

- all validators of the processed events return `null` and
- data validations of the processed events do not return any validation constraint violations.

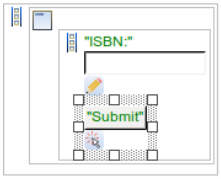
However, mind that for ValueChangeEvents it is enough if *their own validation passes*: ValueChangeEvents are queued for processing always when their validation passes. Other listeners are ignored. Consequently, the *Execute if other validations failed* setting of ValueChangeEvents is ignored.

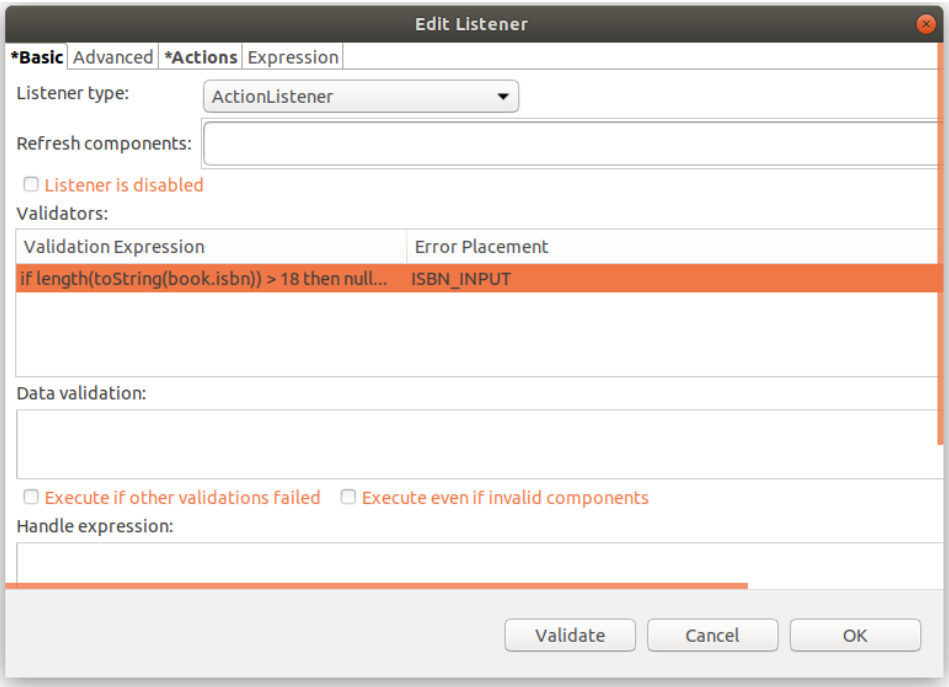
On the other hand, the validity of ValueChangeEvents has no impact on the validity of non-ValueChangeEvents.

**Example:** Consider the form below:

- The ISBN text box has a ValueChangeListener with a validator for the entered text.
- The Submit button has an ActionListener with the *Execute if other validations failed* property set to `false` and executes submit (the Submit property is selected).

If the user enters an invalid ISBN and hits Submit, the form is submitted and the invalid ISBN value is persisted: the ActionListener on the Submit button ignores that the validation of the ChangeValue↩ Listener failed. To prevent the form submit in such cases, move the validation into the ActionListener.



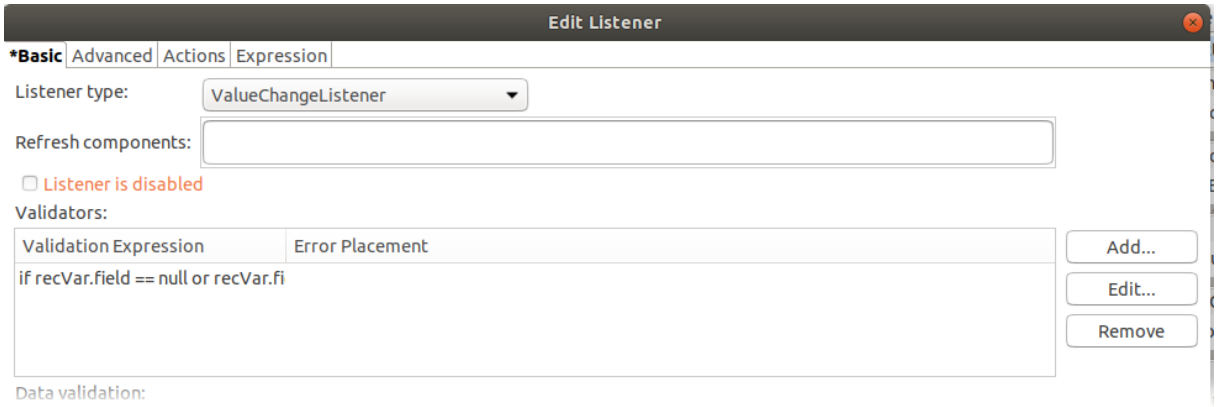



### 3.5.1 Validating a Value of a UI Form Component

To validate a value of a simple data type automatically, use validator expressions: the expressions are checked automatically during the validation phase of they event processing)

To create validators and validate a value of a simple data type automatically, do the following:

1. Open the listener that should cause the validation.
2. On the *Basic* tab in the *Validators* section define the validator expressions: The expressions return a string with the error message when the validation fails, for example, `if searchVar!=null and length(searchVar)>0 then null else "Provide search string!" end`. The message is displayed as an error message either on the current component or on the component defined by their *Error placement* property.




**Note:** Mind that failed validation on ValueChangeListeners does not cause fail of the form validation: other listeners will still be processed. For further information refer to the section on validation

### 3.5.2 Validating a Record Value in a UI Form

To validate values of record properties entered in a form, define the Data Validation expression on the listener: the expression is checked during the validation phase of the event processing. If it returns a list of ConstraintViolations with messages, the messages are displayed on the components that are bound to the properties.

To obtain the list of violated constraints, call the `validate()` function.

If you need to display a constraint violation only on a particular component, create a constraint violation with a record or property set to the same binding as your component; for example: `[new ConstraintViolation( payload -> null, record -> selectedRecord, property -> LookupValue.displayName, guid -> null, id -> null, message -> "Incorrect value.") ]`

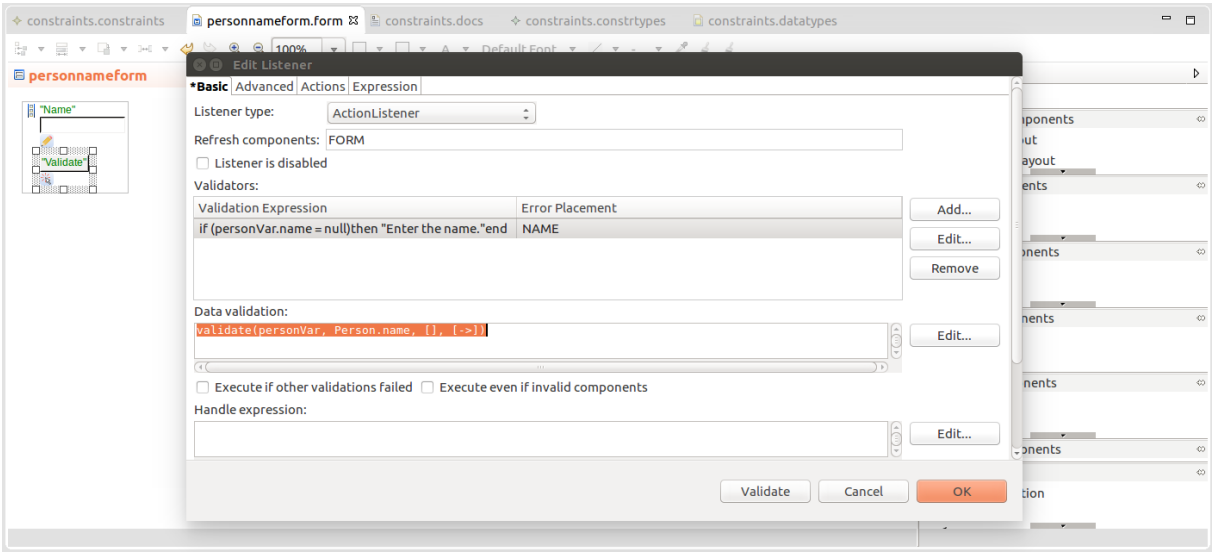



**Figure 3.8 Listener with validators and data validation**

**Note:** Mind that if you define validation on ValueChangeListeners, the failure of the validation does not cause the overall form validation to fail: other listeners will still be processed. For further information refer to the section on validation

### 3.5.3 Defining Validation in Listener Handle

To perform validation manually from a Listener Handle expression, you will need to

- collect a set of constraint violations, typically, by calling the `validate()` function on the respective Record,
- call the `showConstraintViolations()` function to display the violations.

For further information on the functions and constraints, refer to the documentation on Record validation.

When validating a Record in the Handle expression of a Listener, do the following:

1. Collect a list with constraint violations; typically you will use the `validate()` function on the record or property.
2. Call `showConstraintViolations()` function to display the error messages of the violation on the respective form components.

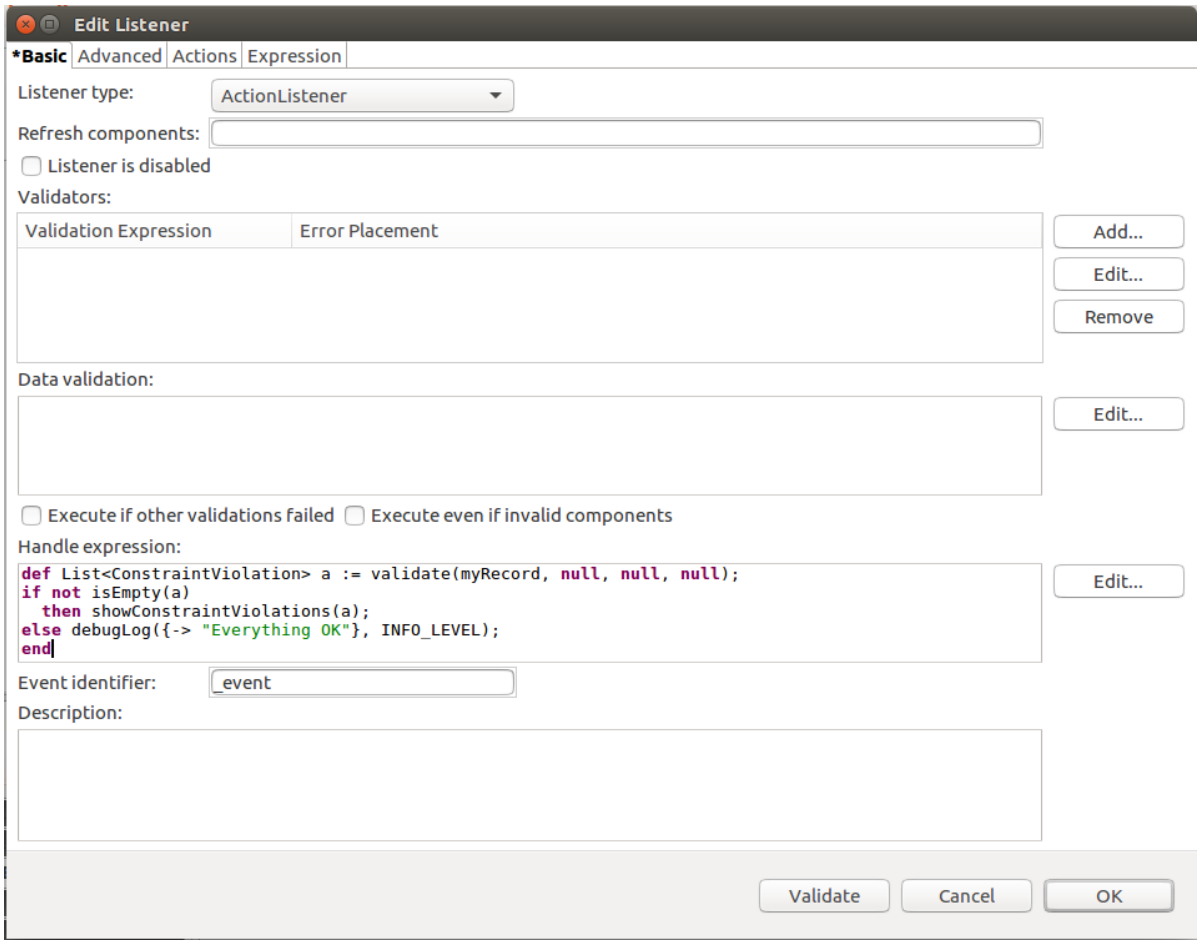



**Figure 3.9 A record validated as part of a listener's Handle**

**Note:** Mind that if you define validation on ValueChangeListeners, the failure of the validation does not cause the overall form validation to fail: other listeners will still be processed. For further information

### 3.5.4 Handling an Event When Validation Failed

To process a listener even when the form validation fails, select *Execute if other validations failed* of the listener.

It is enough that the validation on that particular listener passes for the event to be handled: failed validations on other listeners are ignored.

### 3.5.5 Filtering Validation Errors

When using validation of constraints on form components, by default, any validation errors are displayed on the form component that contains the value with the error. However, you can define explicitly which validation errors are displayed or ignored by the component on the *Validation Errors* tab of its properties:

- **Exclude Validation Error**: intended for filtering of errors; only a subset of the errors is displayed on the component.

- **Include Validation Error**: additional validation errors displayed on the component.

  Typically, you will include here errors that would otherwise remain hidden; for example, when editing multiple record fields, which result in an overall record constraint violation, the error for the entire record is not be displayed since there is no form component that references the entire record.

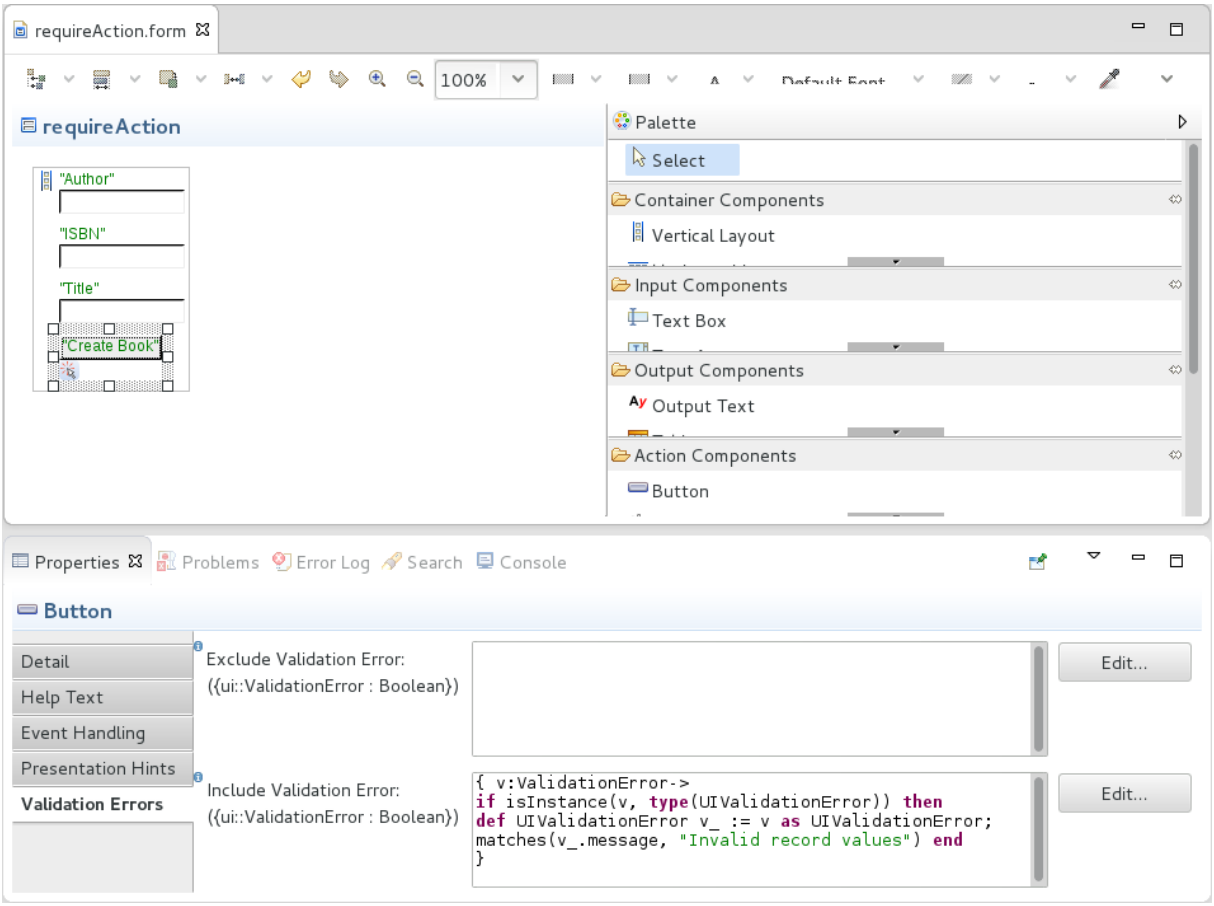



**Figure 3.10 Including a Validation Error on a Form Component**

If you want to ignore invalid data in a form and force processing of an event, select the **Execute even if invalid components** setting for the given listener: the listener will be always executed as part of the next event processing.

### 3.5.6 Validating Initialized Forms

To validate initialized forms, design the forms as follows:

1. On the form, define an InitListener which fires an ApplicationEvent.
2. Define an ApplicationEventListener that listens for the ApplicationEvent.
3. On the ApplicationEventListener, define the validation expression.

## 3.6 Reusing Forms

To reuse an existing ui form, insert into your current form the Reusable Form component and set it to reference the existing form: On runtime, the Reusable Form inserts the referenced form into your form tree and renders it as its part.

Note that events that occur in Reusable Forms are not visible to the parent form and vice versa. To enable handling of events between the reused form and the parent form, you will need to define interface elements using the Container component:

- **To catch an event produced by the parent form and handle it in the reused form:**

  1. Wrap the reused form content in a Container component.
  2. Define a Public Listener that will listen for the event in the parent form.
  3. Add the Public Listener to the parent form.

  Detailed instruction are available in Sending Events to a Reused Form.

  > **Note:** If a public listener needs to listen on another higher parent context, you can register the listener as a delegate listener on the mediating Container.

- **To catch an event produced by a reused form and handle it in the parent form:**

  1. Wrap the reused form in a Container component.
  2. Define a Registration Point that will expose its component.
  3. Define a Private Listener on the parent form that will be registered with the Registration point and listen for the event.

  Detailed instruction are available in Receiving Events from a Reused Form.

  > **Note:** Registration points are implemented as references to the set of listeners defined on the Container component. If an event of the given type is fired within the form, it is handled as defined by the registered listener of the parent form.

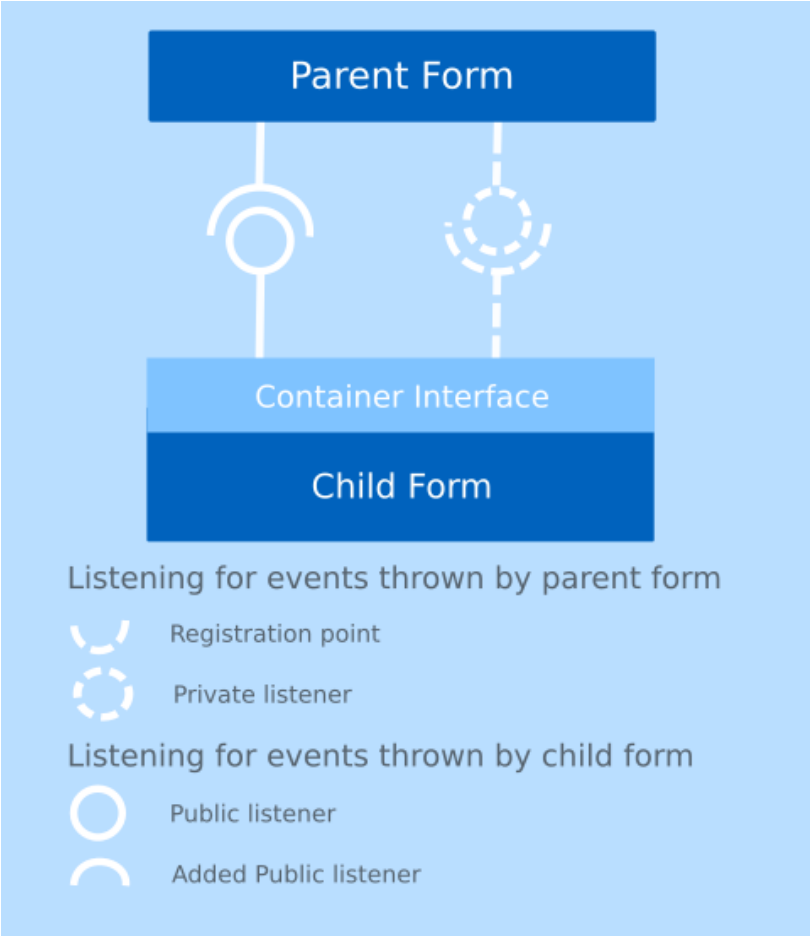



**Figure 3.11 Schema of listener exposure in reusable forms with indications on how the events flow**

Another way to handle events with listeners that are not defined on the component is to use Application Events: Application Events can be created by a listener of any type and handled by one or multiple listeners of any form component and that even if the component is hidden or located in a reusable form. Therefore it is not necessary to define any interfaces to handle Application Events. Note that this mechanism can result in an involute form definitions which are difficult to debug; therefore it is recommended to use Application Events sparingly.

### 3.6.1 Receiving Events from a Reused Form

If a form needs to handle an event that occurs in the form of its Reusable Form component, you need to expose the event using a registration point on the form referenced by the Reusable Form component and define a private listener with the required Actions on the Reusable Form component.

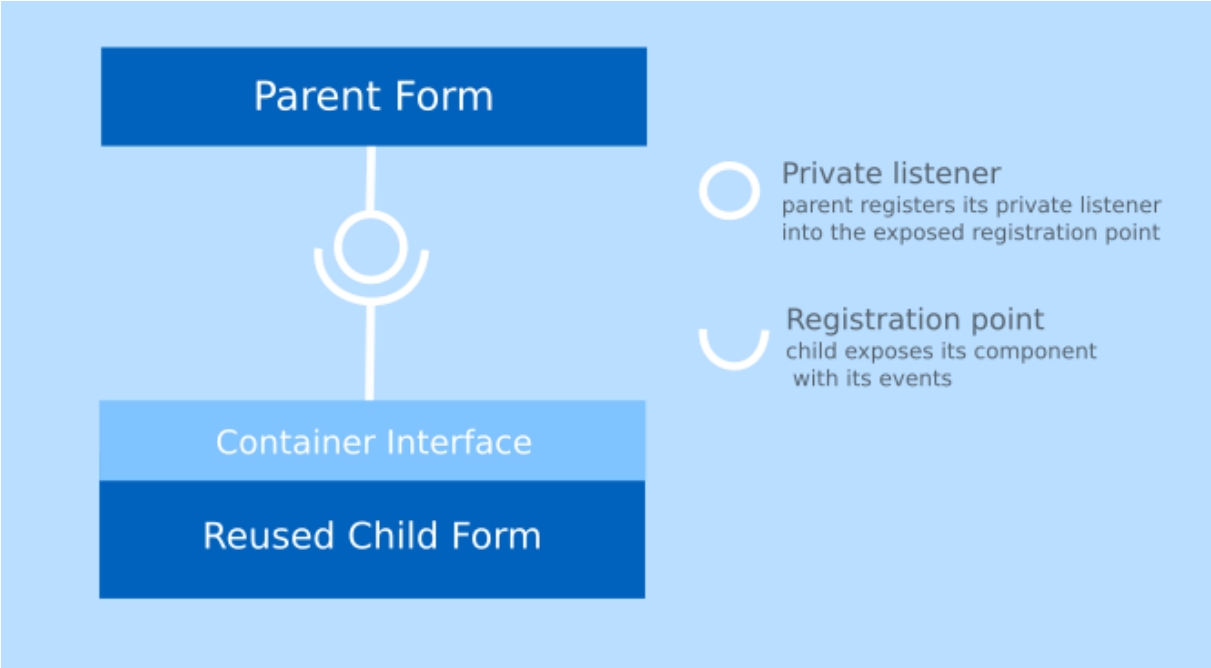



**Figure 3.12 Listening for events that occur in a Reusable Form component**

Note that if you need to expose an event that occurs in a form that is included via multiple Reusable Form components, you need to mediate the registration points into upper forms.

To define such event handling, do the following:

1. Create or open the child form that will be referenced by the Reusable Form component of a parent form:

   (a) Insert the Container component as the root component of the form.

   (b) On the Container, define the registration point that will register the listener of the parent form:

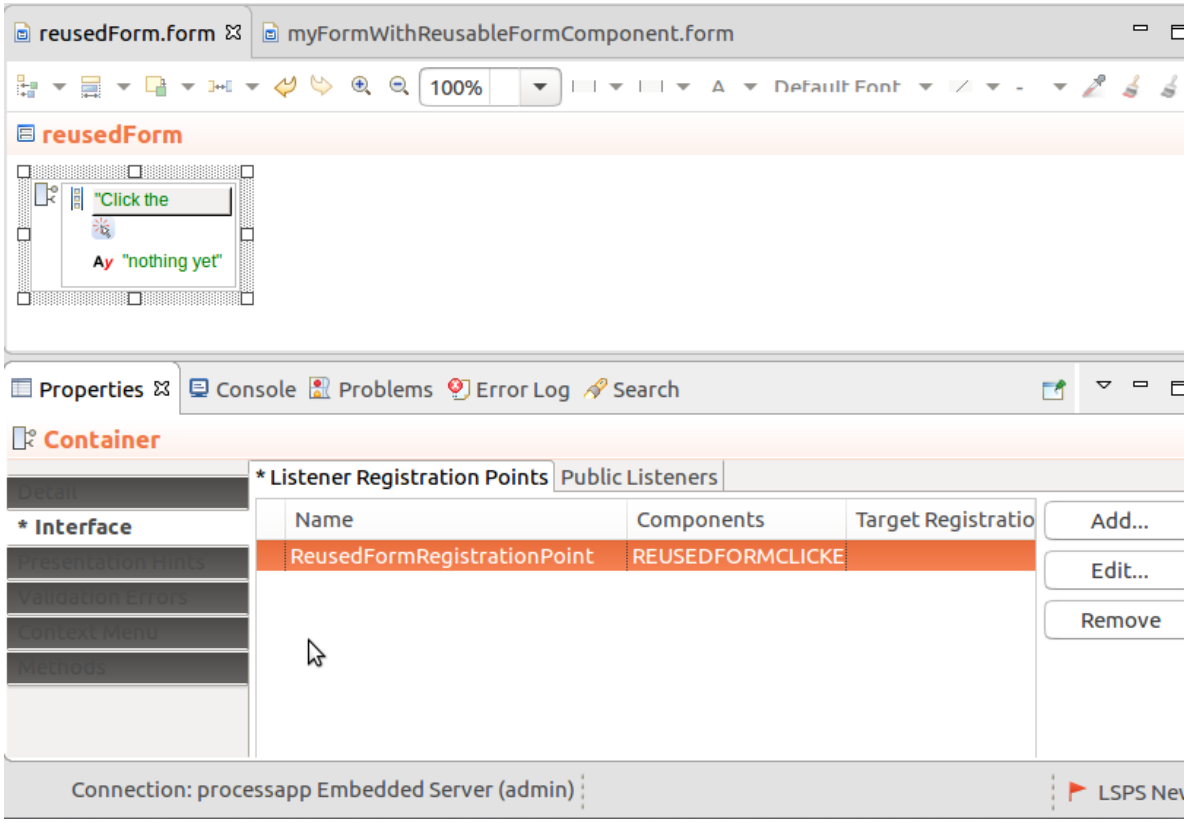
      i. Select the Container.

      ii. Go to the Properties view.

      iii. In the Interface properties, click the *Listener Registration Points* tab and click **Add**.

      iv. In the Add Listener Registration Point dialog box, define the registration point properties:

         - Name: name of the listener defined on the Reusable Form component
         - Components: IDs of the components the listener listens on




2. Create or open the form definition that will reuse the form you just created:

   (a) Insert the Reusable Form component available under the Special Components and define its properties:

   (b) As its *Form* property, set the form from the previous step.

   (c) On the *Event Handling* tab in the *Private Listeners* section, create a private listener with the following properties:

   - **Listening on registration point**: the registration point you defined on the Container of the reused form (use autocompletion)
   - Any other relevant listener properties.

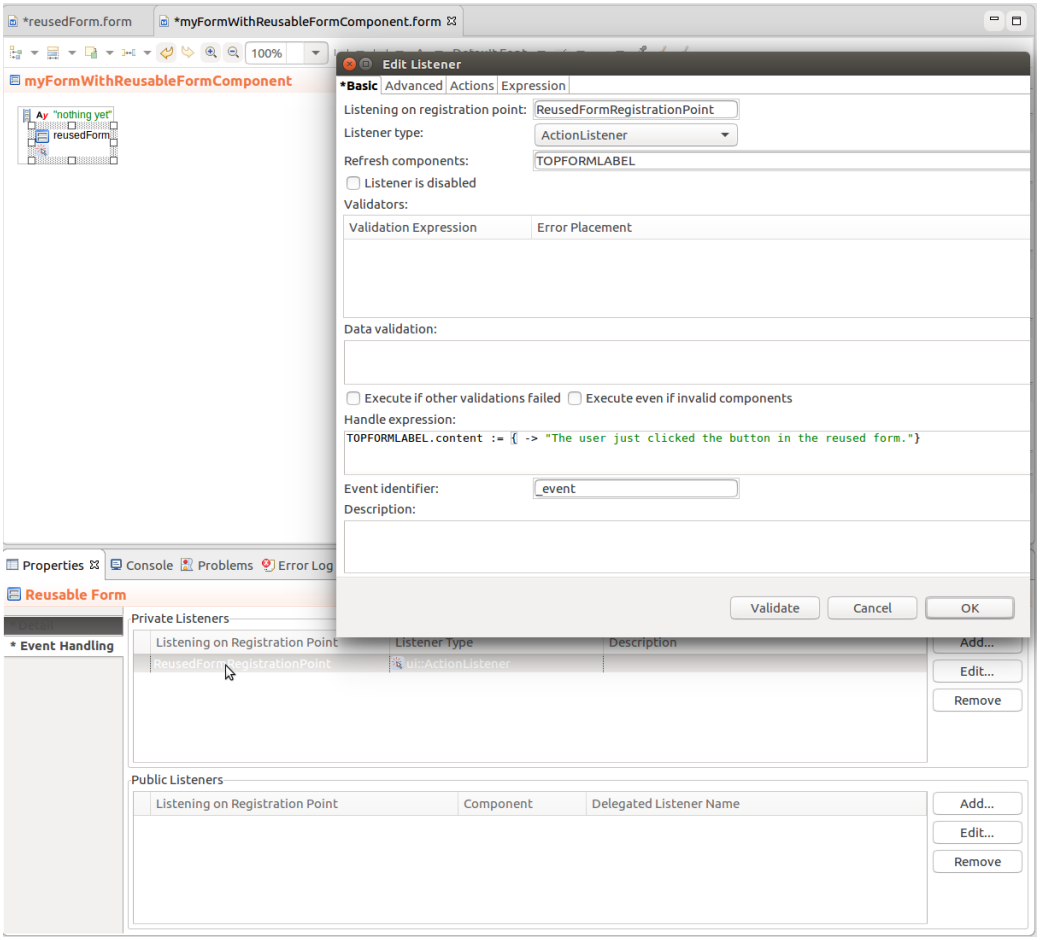



**Figure 3.13 Reusable form component with a private listener for a registration point**

3. Test your form on the Embedded Server: right-click the form definition, go to **Run As** > **Form Preview**.

#### 3.6.1.1 Receiving Events from a Reused Form across Multiple Reusable Forms

If a form needs to handle an event that occurs in a Reusable Form of another Reusable Form component, you will need to delegate the Registration Point through the mediating Reusable Forms:

1. Create the child form that will be referenced by multiple Reusable Form components.

   The form must have the Container component with the registration point as its root component.

2. Create or open the mediating form:

   (a) Insert the Container component as the root component.

   (b) Insert the Reusable Form component that references the reused form step 1. Make sure to define its ID.

   (c) On the Container component, create Registration Points that will mediate the Registration point from the reused form:

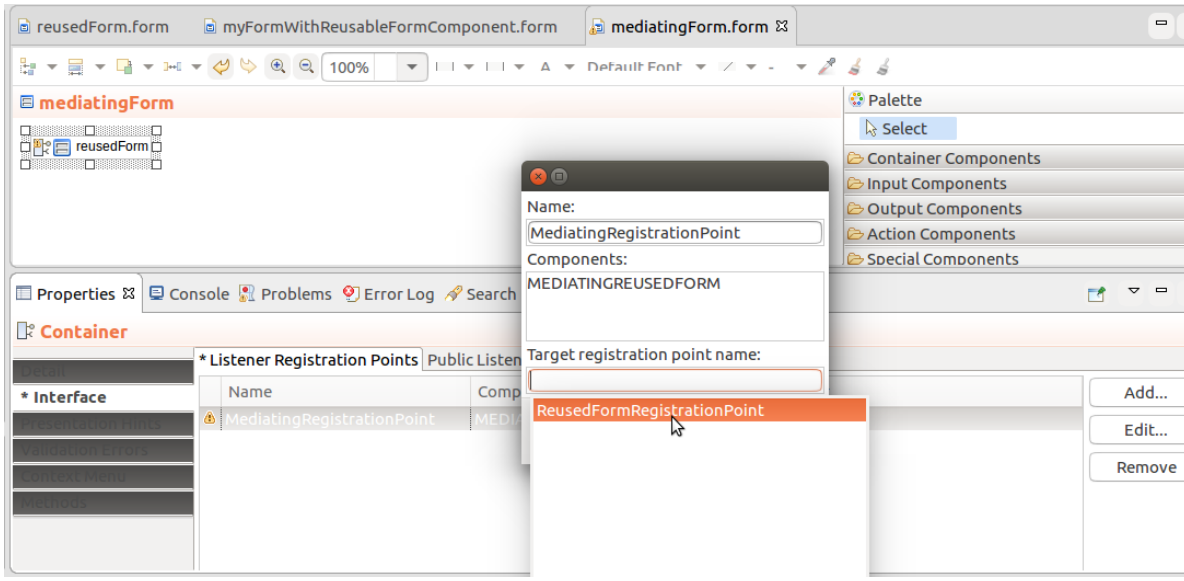
      i. Open the Properties view of the Container.

      ii. On the *Interface* tab, click the *Listener Registration Points* tab and click **Add**.

      iii. In the dialog box, define the registration point properties:

         - Name: name of the registration point
         - Components: ID of the Reusable Form component

- Target registration point name: registration point of a child Reusable Forms (use autocompletion)
  It is this property that connects the form to the Registration Point in the child Reusable Form.




3. Create the form that will handle the mediated event: insert the Reusable Form component that references the respective mediating form and create a private listener on the component.

### 3.6.2 Sending Events to a Reused Form

If a Reused Form needs to handle an event from other form components, you need to insert a public listener into the Reusable Form component that will listen for the event.

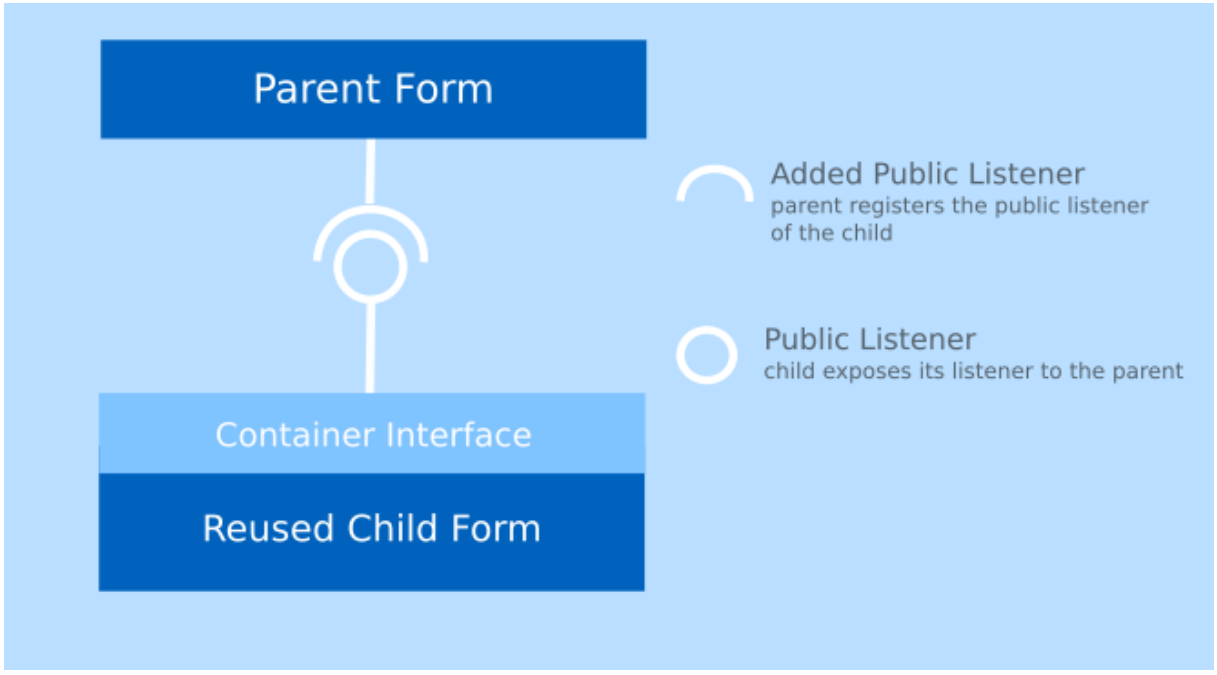



**Figure 3.14 Public listener and its counterpart: added public listener on the parent**

Note that if you need to send an event via multiple Reusable Form components, you need to expose the public listeners using delegated listeners

To define such event handling, do the following:

1. Create the form you want to use as the Reusable Form (this form will listen to an event on its parent form):
   (a) Create the form.
   (b) Insert the Container component as the root component of the form.
   (c) Define a public listener on the Container component:
      i. On the Interface tab of the Container properties, select the *Public Listeners* tab.
      ii. Click **Add** next to the New Listeners section.
      iii. Define its properties: make sure to select the correct listener type.

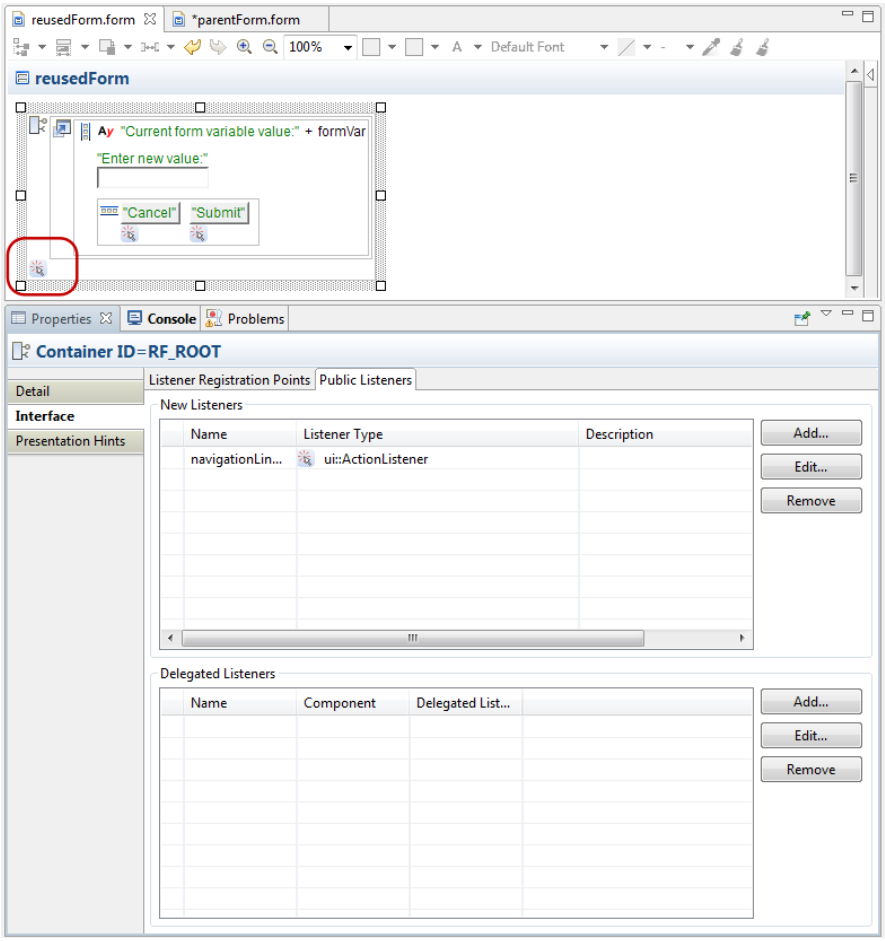
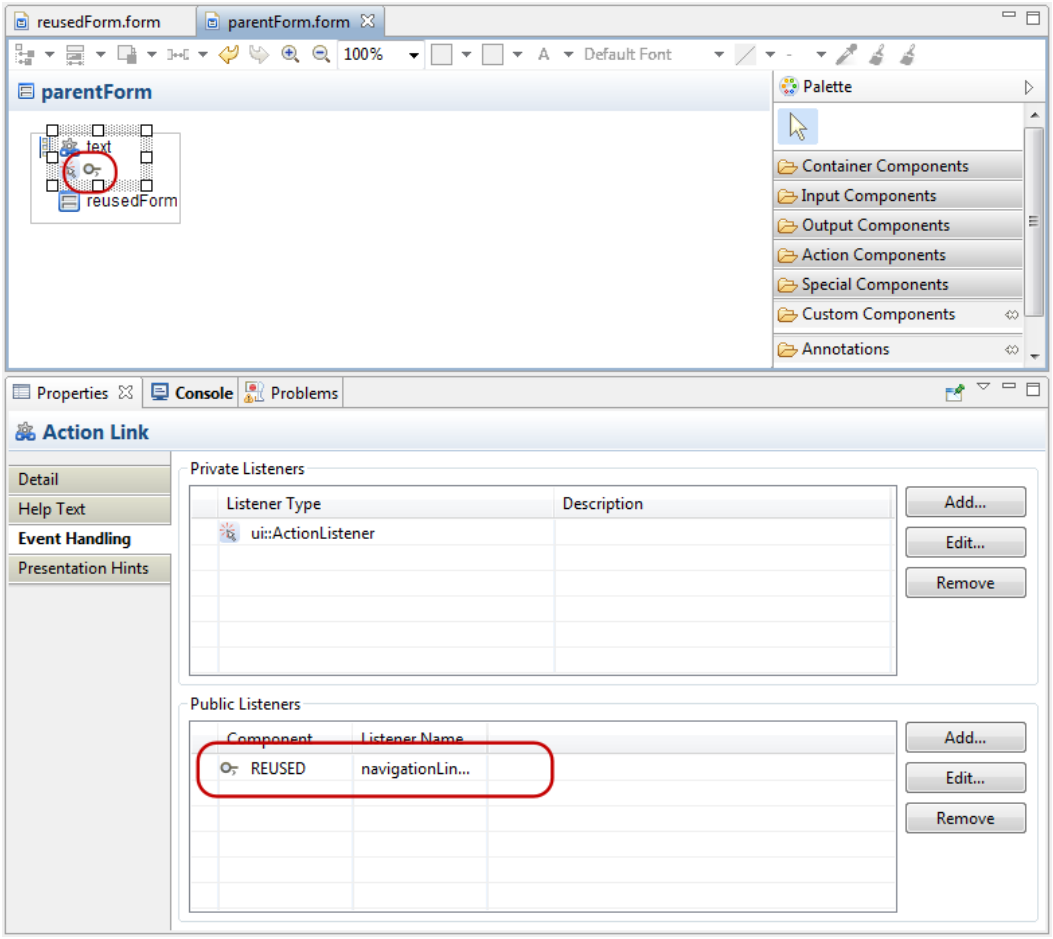



**Figure 3.15 Container with the public listener navigationLinkClick**

2. Open or create the form that will use the created form in the Reusable Form component:
   (a) Insert the Reusable Form component available under the Special Components and define its properties: Make sure to define its ID and the referenced form.
   (b) Create the component that will throw the event you will process in the reused form.
   (c) Register the listener on the component:
      i. Display its Event Handling tab in the Properties view.
      ii. Click **Add** in the Public Listeners section and define the properties of the public listener:

- Component: the Reusable Form component that should process the event
- Listener name: name of the listener you defined on the form in the previous step (use auto-completion)

If applicable, consider setting the *Immediate* property on the input component to `true`.




#### 3.6.2.1 Sending Events to a Reused Form across Multiple Nested Forms

To expose a listener of a reused form across multiple parent forms, define a delegated listener in the mediating forms: This will allow the child reusable form to listen for events that occur in other than the immediate form.

1. Create the form you want to use as the Reusable Form with the Container component with a public listener.
2. Delegate the public listener through the Container interface of other forms with Reusable Forms components:
   (a) Create or open the mediating form with the Container component.
   (b) Insert the Reusable Form into the Container. Make sure to define its properties including its ID and the referenced form.
   (c) Delegate the public listener of the Reusable Form:
      i. Select the Container component, and click the *Interface* tab in its Properties view.
      ii. Select the *Public Listeners* tab and click the Add button in the **Delegated Listeners** section.
      iii. Specify the delegated listener parameters:
         - Name: name of the delegated listener
         - Component: the Reusable Form ID

- Listener name: name of the listener of the reusable form (the one you are mediating; use autocompletion)

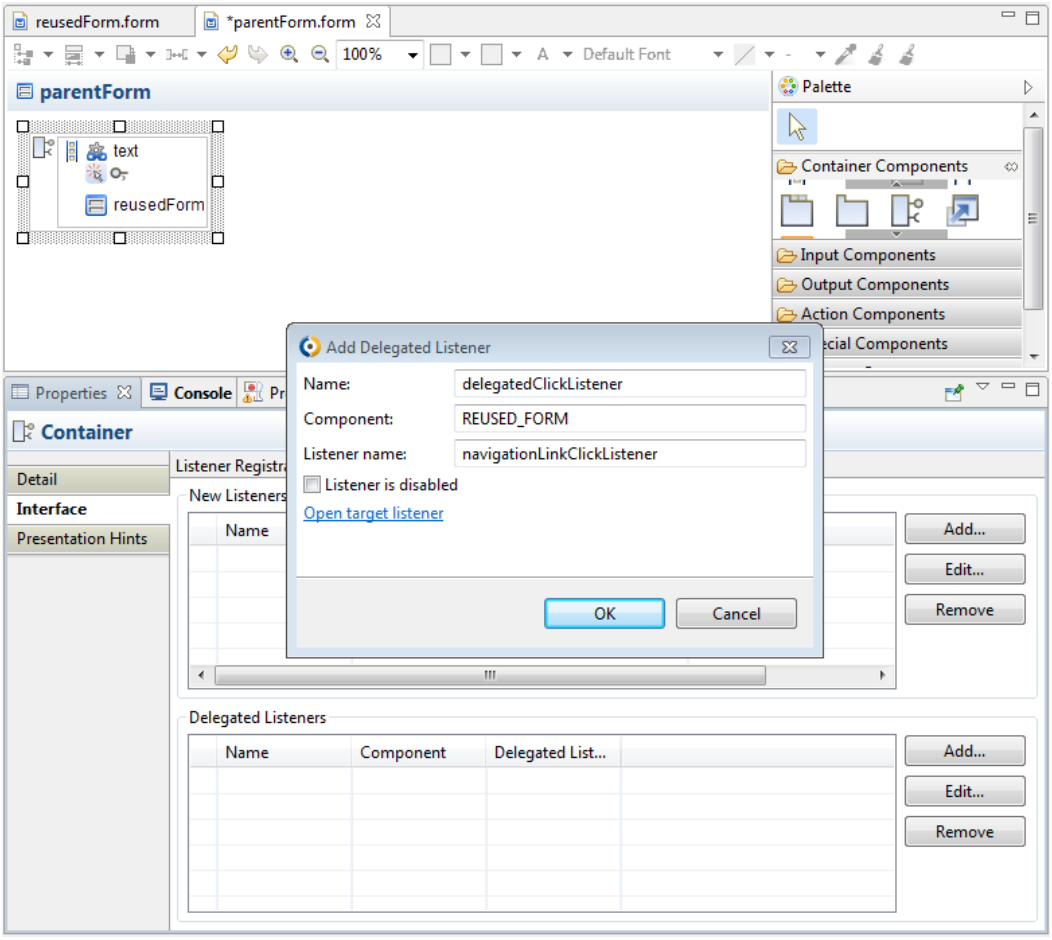



**Figure 3.16 Defining properties of a delegated listener**

3. To delegate across further Reusable Form components, repeat the previous step.
4. Open or create the form that will use the created form in the Reusable Form component and create a public listener on the Reused Form.

### 3.6.3 Broadcasting an Event

To broadcast an event across the entire form, throw an ApplicationEvent: an ApplicationEvent is thrown as part the event-processing lifecycle based on a listener definition. You can then define ApplicationEventListeners on the form to catch and process the event.

To define an application event that will be thrown by the event-processing cycle, do the following:

1. Create a listener on the component that should give rise to the event.
2. Open the properties of the listener (on the Event Handling tab of the Properties view, double-click the listener).
3. In the Edit Listener window, go to the Actions tab.
4. In the lower part of the tab, check the Fire application event checkbox.
5. In the area below, enter the expression that creates the application event.

```
new ApplicationEvent(eventName -> "idCreateRequest", payload -> idInfo)
```

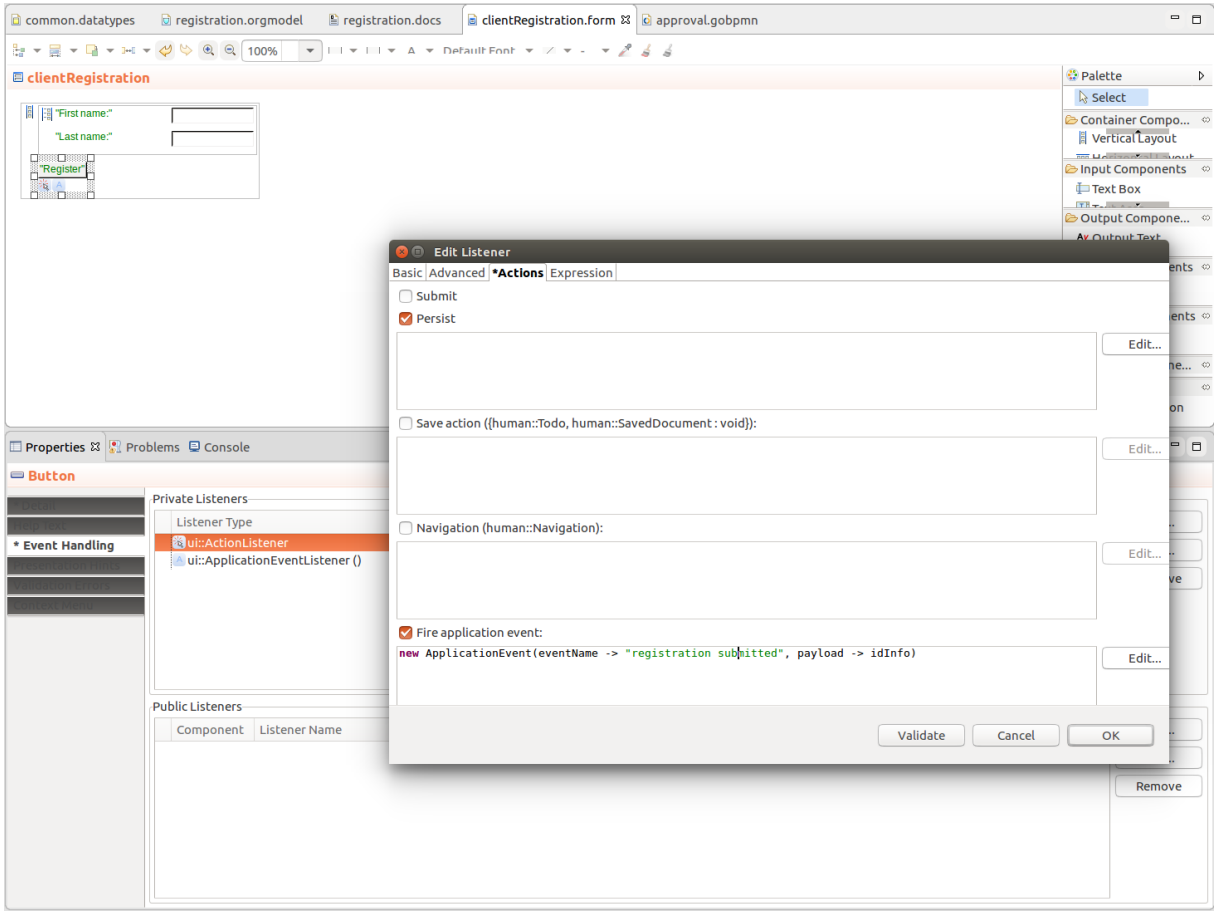



**Figure 3.17 Throwing an application event**

You can now define the ApplicationEventListeners for the event.

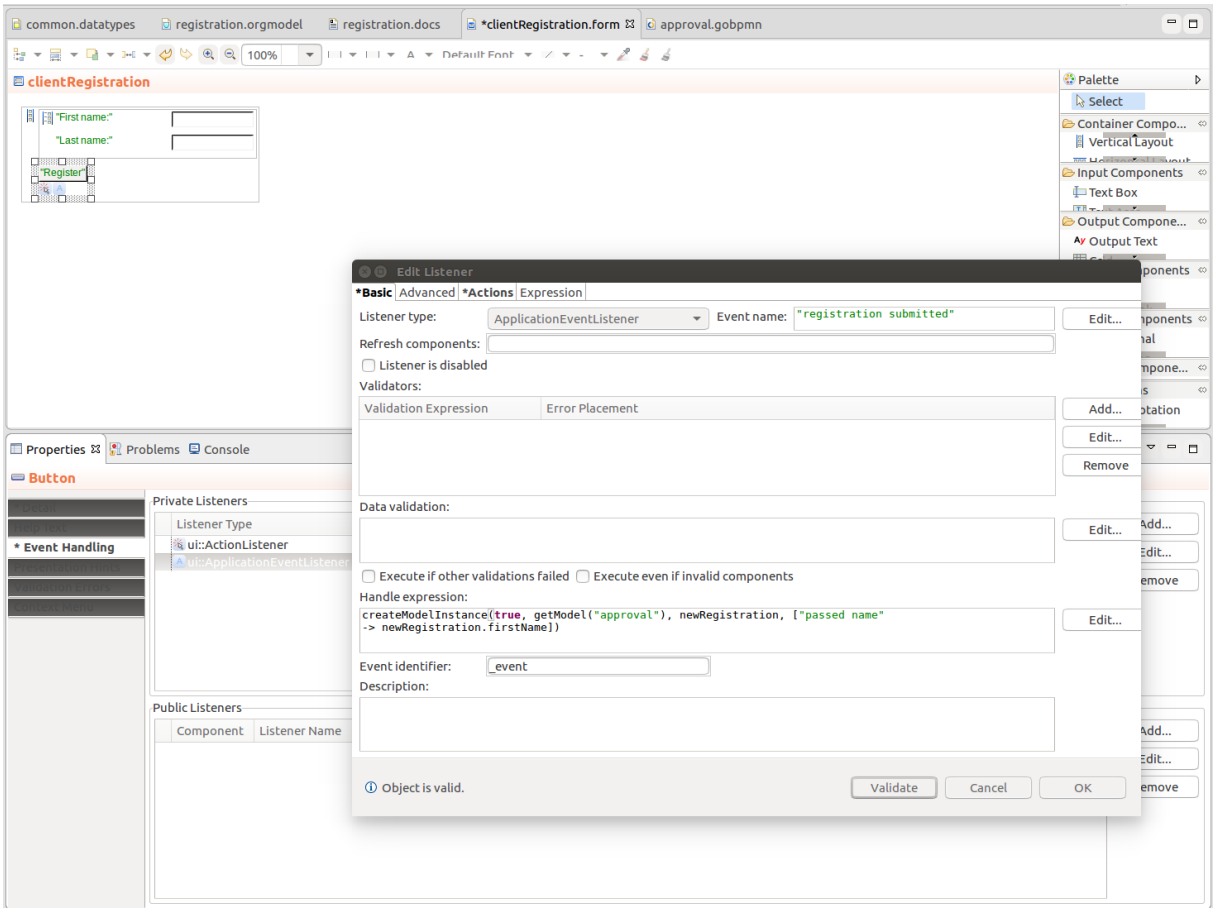



**Figure 3.18 Catching an application event**

## 3.7 Modifying Presentation of Components

To modify presentation properties of a form component, such as, its size, position, alignment, or possibly properties of a custom form component, use *presentation hints*. In the Application User Interface, hints translate into classes of the element.

You can use the hints defined in the Standard Library or you can define your custom hints.

> **Important:** Any hint assigned to a component is inherited by its child components. Note that some hints are defined implicitly and are not visible in the Form editor; for example, layout components have their width set to 100% by default and their children therefore have the width set to 100% as well. To ignore inherited hint values, override the hint value on the child component with another value or the `null` value if you want to erase the value.

### 3.7.1 Standard Library Hints

The ui module comes with a set of default presentation hints stored in the in the `ui::ui.hints file in the Standard Library`.

#### 3.7.1.1 Assigning Hints From the Standard Library

To assign a form component a hint from the Standard Library, do the following:

1. In Form editor, select the form component.
2. In its Properties view, select the Presentation Hints tab.
3. Display the Hint Table tab.
4. Click the Add button to create a new hint or select an existing hint in the table and click Edit.
5. In the Edit dialog window, define the hint properties:
   (a) In the Hint name text field, select a predefined hint in the drop-down menu. You can also enter a hint name manually.
   (b) Either select a predefined hint value in the Predefined option drop-down or define its value in the Expression text area.

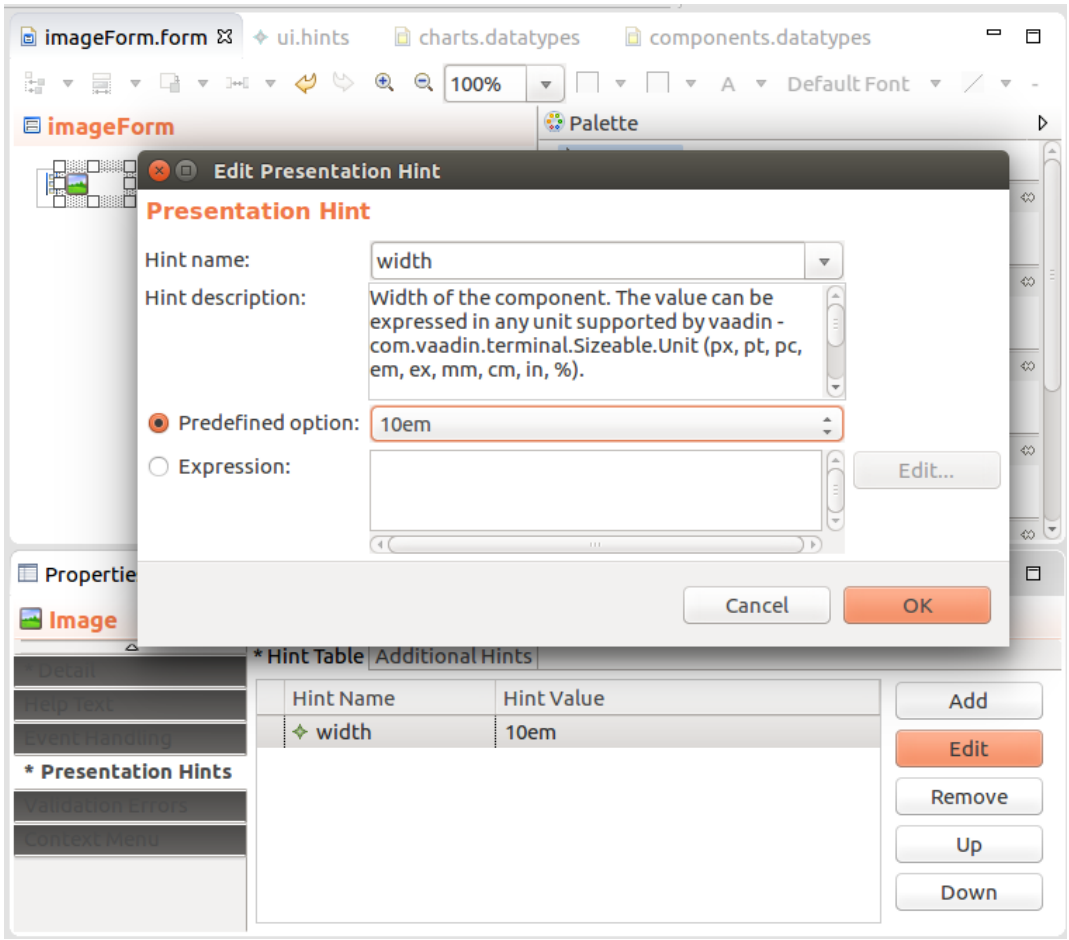



**Figure 3.19 Editing a presentation hint**

### 3.7.2 Custom Hints

Hints can be defined either directly on a form component or in a hints definition file. The hints defined in a hints definition file are available in the entire parent module.

Make sure that the component implementation accepts and processes the hint. Otherwise, the hint is ignored.

To define hints that can be used by all forms in the module, do the following:

1. Create a hint definition file (go to File New Hint Definition, and select the parent module and provide the definition file name).

2. In the GO-BPMN Explorer, double-click the hint definition file.

3. In the Hint Editor, click the **Add** button.

4. On the right, define the hint properties:

   - Name: hint name displayed in the Hint name drop-down menu.
   - Component: comma-separated list of form components that can use the hint
   - Hint options: value options available for the hint in the Predefined option (the label holds the displayed name of the option and the expression holds the value it is translated to when rendered)

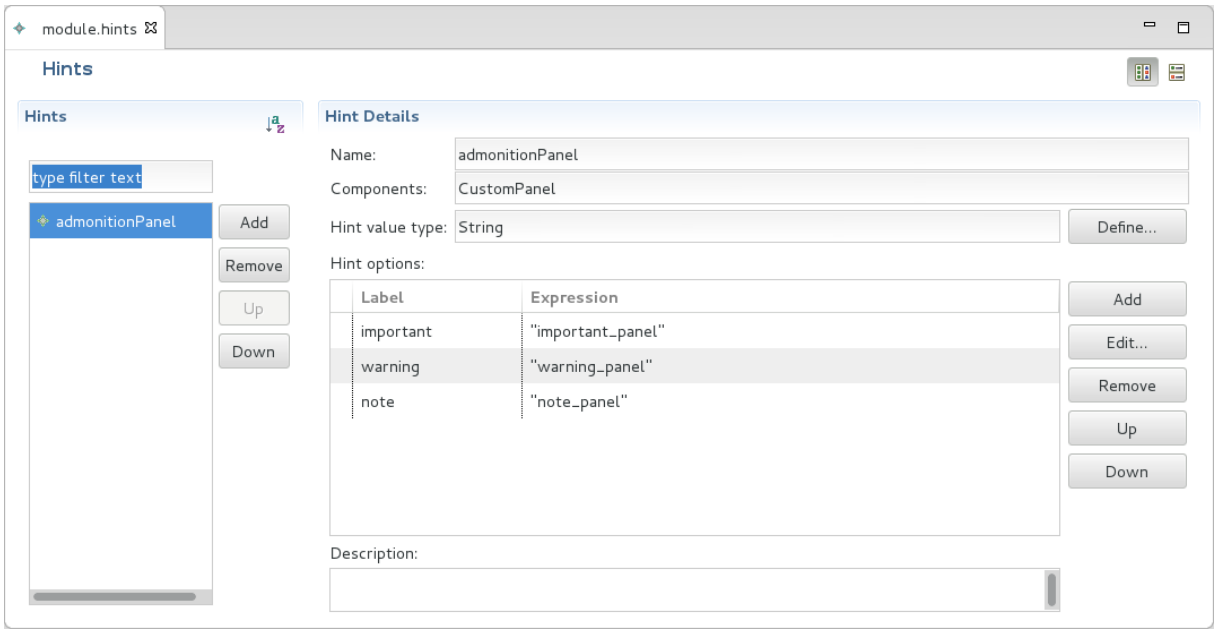



**Figure 3.20 Hints editor with a hint definition**

#### 3.7.2.1 Assigning Custom Hints

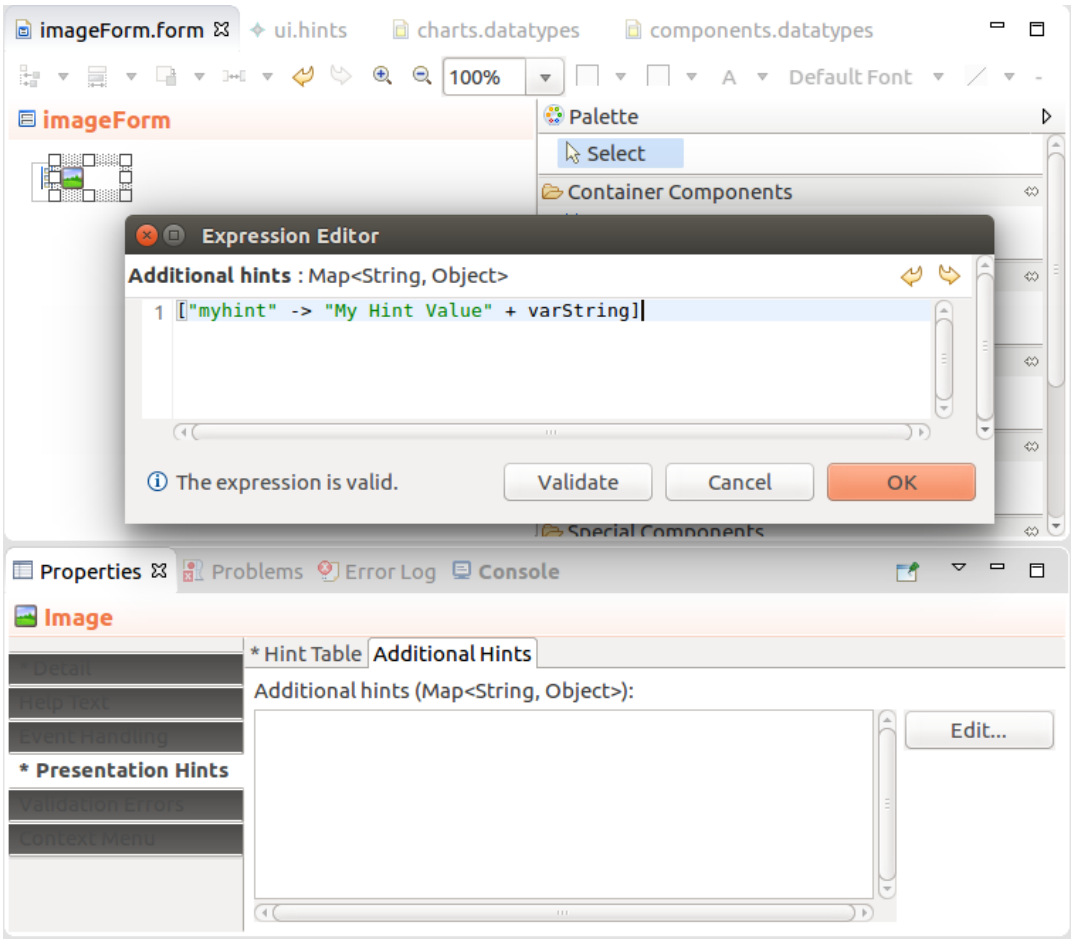
To assign a form component a custom hint or a standard-library hint, do the following:

1. In the form editor, select the form component.

2. In its Properties view, select the Presentation Hints tab.

3. On the Additional Hints tab click Edit.

4. Define the hint name and values as a map of a String and Object (for example, `["myhint" -> "My Hint Value" + varString]`).




### 3.7.3 Using Hints

#### 3.7.3.1 Aligning Form Components

To define alignment of child components in the Vertical, Horizontal, and Grid Layout, and table Column, use the `align hint`. Note that the *top*, *middle*, and *bottom* align settings are simply ignored.

On a table Column, the alignment is applied also on the column header. If necessary, you can override the setting with the *header-align* hint.

#### 3.7.3.2 Resizing Form Components

Component size is defined by the height and width component hints. Where applicable, the size hints are defined implicitly. However, you can be override them with an explicit hint definition.

Hint values can be defined in the following units:

- font-relative units: size definition relative to the font size
  - em: size in factors of the letter `m` size
  - ex: size in factors of the letter `x` size
- absolute size: component size in pixels (px), picas (pc), points (pt), mm, cm, or inches (in)

- relative size to the parent: component size in relation to the size of the immediate parent

  You can use the predefined values:

  - `fillparent`: identical to `100%`,
  - `wrapcontent`: component size is set to the sum of sizes of children components

- `expand`: Sets the expand ratio for a component. If there is only one component with an expand ratio, the component will be expanded to the maximum possible size within its parent. This hint can only be applied to a table Column or the direct child of a Vertical or Horizontal Layout. Form Layout is unsupported.

#### 3.7.3.2.1 Default Size Hint Values

Form components define implicitly presentation hints with default values. These values are applied unless you define another value for the hint.

Note that the default values might be in conflict. As a result, the rendered form might not meet your requirements or render at all. For example, consider a table in a vertical layout:

- Vertical layout has the default width hint defined to `wrapcontent` so it checks the size of its child components and sets its width to the maximum child width.
- A table has the default width hint value defined as `fillparent` so it uses the parents' width.

Hence there is no setting for the table or the layout and you need to explicitly set to one of the widths to a required value.

**Default Widths and Heights**

| Component | Default Width | Default Height |
|---|---|---|
| Container Components | fillparent | wrapcontent |
| Grid Layout | fillparent | wrapcontent |
| Form Layout | fillparent | wrapcontent |
| Text Box | wrapcontent | wrapcontent |
| Text Area | wrapcontent | wrapcontent |
| Check Box | wrapcontent | wrapcontent |
| Combo Box | wrapcontent | wrapcontent |
| Lazy-Loading Combo Box | wrapcontent | wrapcontent |
| Form Layout | fillparent | wrapcontent |
| Single-Select List | wrapcontent | wrapcontent |
| Multi-Select List | wrapcontent | wrapcontent |
| Check-Box List | wrapcontent | wrapcontent |
| Radio-Button List | wrapcontent | wrapcontent |
| File Upload | wrapcontent | wrapcontent |
| Output Text | fillparent | wrapcontent |
| Table | wrapcontent | wrapcontent |
| Calendar | fillparent | fillparent |
| Action Button | wrapcontent | wrapcontent |
| Browser Frame | fillparent | wrapcontent |
| Button | wrapcontent | wrapcontent |
| Cartesian Chart | fillparent | 400px |
| Gauge Chart | fillparent | 400px |
| Pie Chart | fillparent | 400px |

| Component | Default Width | Default Height |
|---|---|---|
| Polar Chart | fillparent | 400px |
| Conditional | fillparent | wrapcontent |
| Dashboard | fillparent | fillparent |
| Dashboard Widget | 0, 0, 200, 300 | overrides DashBoard |
| File Download | wrapcontent | wrapcontent |
| Geolocator | invisible | invisible |
| Horizontal Layout | wrapcontent | wrapcontent |
| Image | wrapcontent | wrapcontent |
| Lazy Table | wrapcontent | wrapcontent |
| Tree Table | wrapcontent | wrapcontent |
| Map Display | fillparent | 400px |
| Message | fillparent | wrapcontent |
| Navigation Link | wrapcontent | wrapcontent |
| Panel | fillparent | wrapcontent |
| Popup | fillparent | wrapcontent |
| Repeater | wrapcontent | wrapcontent |
| Tabbed Layout | fillparent | wrapcontent |
| Table Column | counted by the table | counted by the table |
| Tree | wrapcontent | wrapcontent |
| Vertical Layout | fillparent | wrapcontent |
| View Model | fillparent | wrapcontent |

#### 3.7.3.3 Defining Common Presentation Properties

With the *html-class* hint, you can define the presentation of a component, such as, border rendering, highlighting, emphasis, overflow, disabling of text, etc. Refer to the `hints documentation in the Standard Library documentation`.

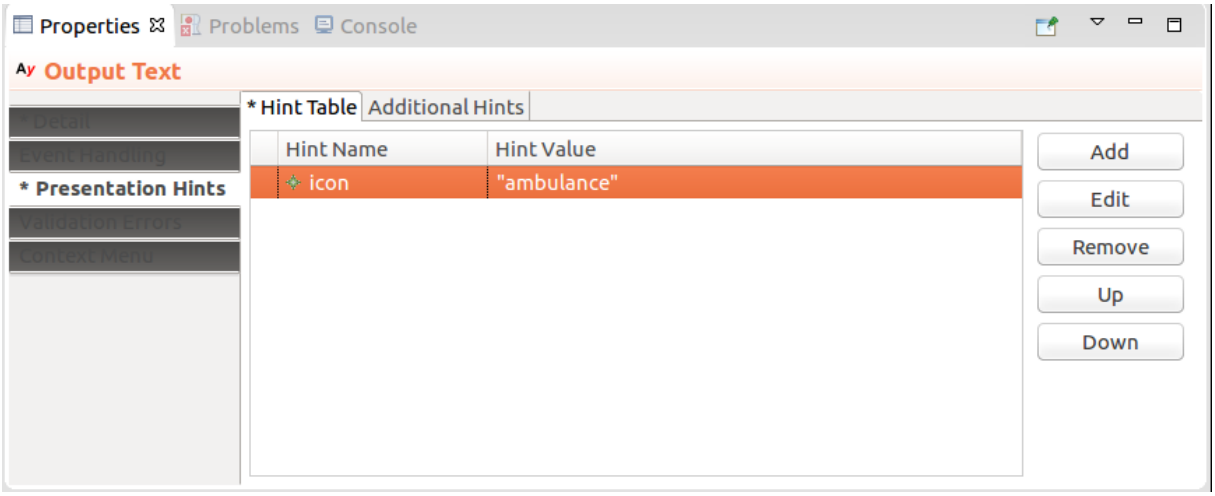
#### 3.7.3.4 Adding a CSS Class to a Form Component

To add a class attribute to the rendered component, use the `html-class` hint.

Note that the class must be defined in the Application User Interface. For instructions on how to add new style sheets, refer to `Custom Application User Interface Guide`.

#### 3.7.3.5 Adding a Font Icon to a Form Component

To add an icon from the Awesome font to your component class, assign the `icon` presentation hint to your component and insert the name of the icon as its value.




#### 3.7.3.6 Setting the Maximum Text Size on a TextBox and a TextArea

The TextBox and TextArea components can define the max-text-size hint, which defines the maximum length of the text the user can enter. If the component is bound to a shared record field, the size must take into account the size of the underlying database column, that is the column to which the field is mapped.

You can hence generate the hint for TextBox and TextArea components that are bound to shared record fields. Note that the hint generation is applied on all open GO-BPMN projects of your workspace. Close any projects you want to exclude.

To generate or adjust the max-size-hint for all TextBox and TextArea components in all open GO-BPMN projects, do the following:

1. Go to **Project** > **Update max-text-size hint**.
2. In the Update max-text-size hints dialog, select the applicable choices:
   - Add missing max-text-size hint to generate the hint on any TextBox and TextArea components bound to a shared record field if the hint is missing
   - Update existing max-text-size hints to overwrite any defined values of the hint with the database column sizes

     **Important:** If a max-text-size hint contains a non-integer value, the hint is invalid.
   - Ignore if value of max-text-size hint is smaller than DB length to prevent overwriting of the field length that are smaller than the length of the underlying database column size.

## 3.8 Creating Mobile Forms

LSPS provides support for creating mobile forms that can be used in mobile LSPS applications.

> **Important:** A mobile LSPS application is by no means intended to substitute the desktop application and should be considered an extension of the desktop application.

The specific support for mobile forms in LSPS includes the following:

- mobile presentation hint for the vertical and horizontal layout component

### 3.8.1 Guidelines

- Use horizontal layout with the mobile hint only to hold a set of buttons.
- Wrap more complex forms either in the vertical layout component with the mobile hint or in the Tabbed container component. Every Tab component can then contain a part of the form. In the Tab component, use preferably icons instead since the used font is rather small.
- Do not nest a layout component with the mobile hint into another layout component with the mobile hint. The exception is a horizontal layout component with buttons.
- Adapt your forms to different screen sizes and resolutions: use Conditional components with different "versions" of the form.

  To detect the screen size on runtime do the following:

  – Create the custom `getBrowserWindowWidth()` that returns an Integer. The function is implemented as the `getBrowserWindowWidth()` method in `com.whitestein.lsps.app.↩ vaadin_touchkit.touchkit.MobileUtils` and returns the width of the displayed page in DP(density independent pixel).
  – Define how to handle the ApplicationEvent with the ID `internal_windowresize`. The event is produced when the width of the browser window changes and therefore also when the device is rotated. It bears the window size in DP as its payload (held by the `ui::Dimension` record instance).

**Important:** If the user provides invalid values in the form and the window width changes, the invalid values are not transferred to the refreshed form. Consider handling such situations in your form.

# Chapter 4

# Components

When you design the component of a form, you insert various types of form components to the form content. The components constitute a tree with each component defining

- listeners for the type of events that can occur on the component and
- properties required to render the component, such as, data to display, presentation styles, size, etc.

The following are the properties that are common to all components:

- **ID**: ID of the component on design time (Only capital letters, digits, underscores and dollar signs are allowed.)

  ID can be used by other form components and listeners in the form.
- **Modeling ID**: ID of the component used on runtime

  Modeling IDs are populated with a random unique value. You can change the value manually if required. If an error occurs in runtime, the server returns an error with the modeling IDs of the involved components. The feature is disabled by default. You can enable it with a parameter.
- **Visible**: visibility of the component and its child components

  If the property expression is `true` or `null`, the component is visible. If `false`, the component is not visible. The default value is `true`. Note that invisible components and their child components are not recalculated on refresh.

  Components with an expression in the *Visible* property are marked with the icon  .

  > **Important:** The **Visible** property substitutes the **Show** property, which was deprecated and removed. However, if your forms still contain the **Show** property, it will be handled as expected. If both of the properties are defined, the Show property is ignored.
- **context menu**: menu displayed when the user right-clicks the component
- **component-specific properties** that define the behavior of the component (refer to sections on individual components)
- presentation hints define the presentation style of the component

Properties and events specific to components or the group they belong to, are described in the respective sections:

- Container Components
- Input Components
- Output Components
- Action Components
- Special Form Components
- Text Annotations and Associations
- Deprecated Components

## 4.1 Container Components

Container components nest components and define the form structure.

They can fire only InitEvents with the exception of the Dashboard Widget, which can fire a WidgetChangeEvent as well.

### 4.1.1 Vertical Layout

The Vertical Layout component ( ) is a container component that can hold multiple form components, which are arranged in a vertical manner.

Specific Vertical Layout properties:

- **Label**: object that is displayed as the title of the layout; label is rendered by the parent component; hence if the layout is the root component, Label is not rendered.

### 4.1.2 Horizontal Layout

The Horizontal Layout component ( ) is a container component that can hold multiple form components, which are arranged in a horizontal manner.

Specific Horizontal Layout properties:

- **Label**: object that is displayed as the title of the layout; label is rendered by the parent component; hence if the layout is the root component, Label is not rendered.

### 4.1.3 Form Layout

The Form Layout component ( ) is a container component that can hold multiple form components arranged in two columns. The child form components are rendered with their labels placed in the left column and any other component parts rendered in the right column.

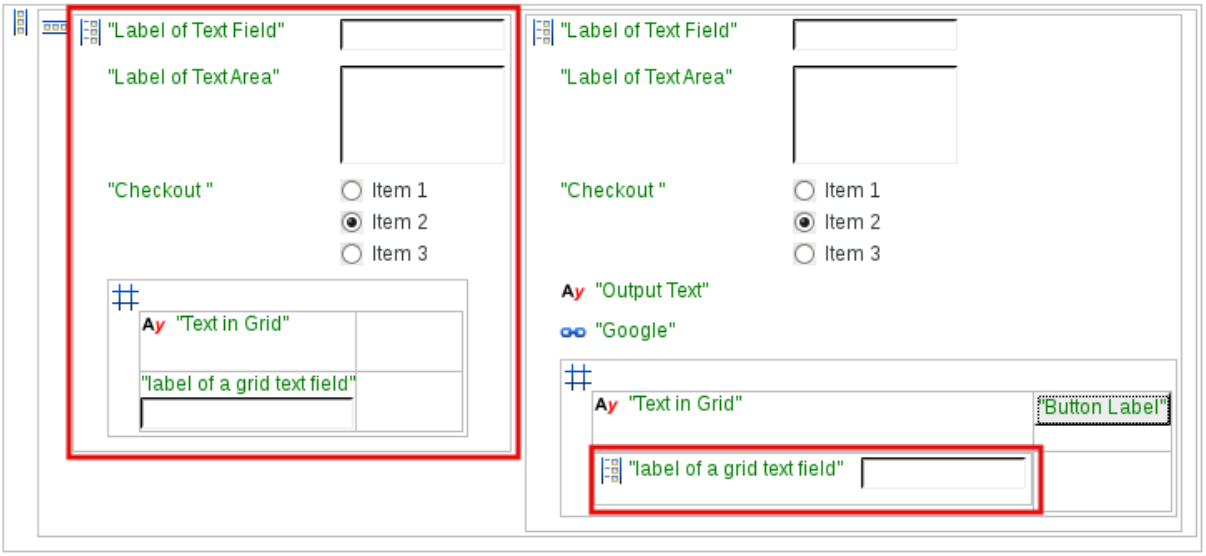



**Figure 4.1 Form with multiple Form Layouts**

### 4.1.4 Panel

The Panel component ( ) is a container component that serves for adding a frame and a caption to one, possibly a layout, component.

A scrollbar is rendered in panel child component automatically if applicable.

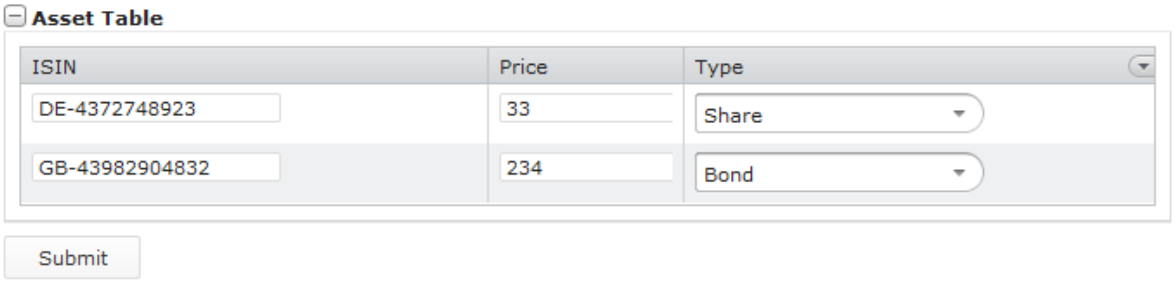



**Figure 4.2 Asset Table nested in a collapsible panel (note the collapse box next to the table label)**

The Panel component has the following properties:

- **Title**: label displayed on the panel The Title property supports HTML content.
- **Collapsed**: if set to true, the panel is rendered as collapsible

### 4.1.5 Grid Layout

The Grid Layout component ( ) is a container component that can hold multiple form component elements, which are arranged in a grid manner, that is in a table-like way. You can add rows and columns to the grid using the Add ( ) and Delete ( ) buttons on the grid component.

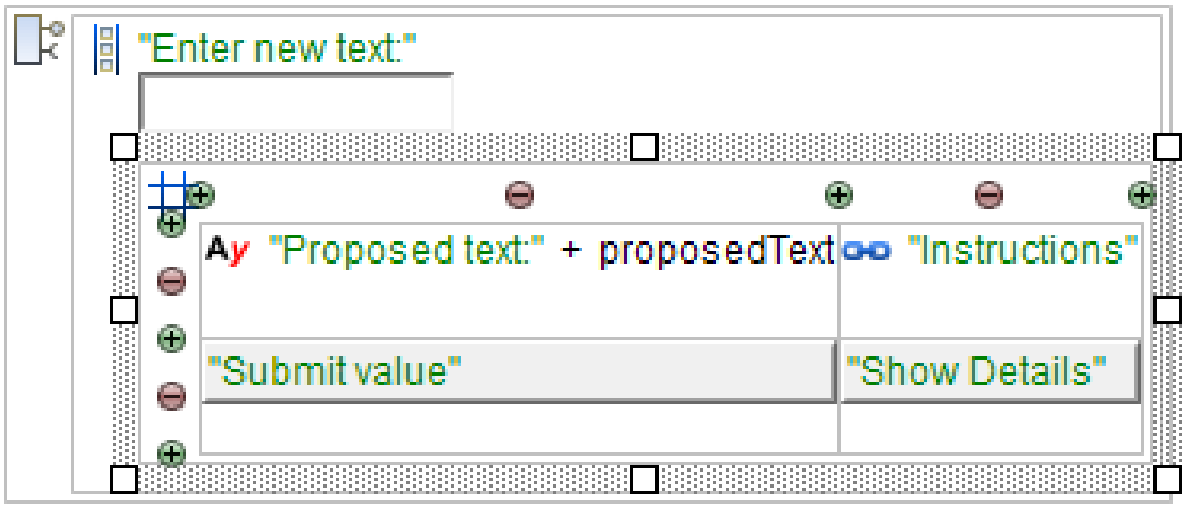



**Figure 4.3 Grid Layout component**

### 4.1.6 Tabbed Layout

The Tabbed Layout component (  ) is a container component that can hold multiple Tab components.

#### 4.1.6.1 Tab

The Tab component (  ) is a component rendered as a tab. It that can contain one form component and its parent component must be the tabbed layout component.

A tab content and its children are initialized along with their parent. However, a tab produces an init event and is refreshed when displayed.

The Tab component has the following properties:

- **Title**: tab label
- **Visible**: boolean expression defining the tab visibility

  If the visible expression evaluates to true or undefined on init or refresh, the tab is displayed.

##### 4.1.6.1.1 Focusing a Tab

To focus a tab in a tabbed layout, use the `selectTab()` function, for example, .

To focus a tab on init, call the function from an InitListener on the first Tab or use dynamic tabs.

#### 4.1.6.2 Dynamic Tabs

You can add, remove, and select a Tab component programatically from listener handle expression using the `addTab()`, `removeTab()`, `selectTab()` functions.

```
 //create dynamic tab:
def Tab myDynamicTab := new Tab(text -> { -> "Dynamic Tab" }, content -> new HorizontalLayout());
//add the tab to a tabbed layout:
addTab(MYTABBEDLAYOUT, myDynamicTab);
//focus the tab:
selectTab(TABBED, myDynamicTab);
```

### 4.1.7 Container

A Container component (  ) is a form component that allows you to define an interface for a form or a form component and its child components. The interface provides a mechanism for listening for events either on nested forms or on parent forms (for details refer to Container Interface).

### 4.1.8 Popup

The Popup component (  ) is rendered as a pop-up window with a maximize/minimize and close button in its caption. It can be modal: when a modal popup is displayed, you cannot work with the underlying form until it is closed.

Since Popups are included directly in the form, they are calculated when the form is initialized. For large popups with a lot of data, this can result in poor form performance since these popups are part of the form all the time even if you might not need them at all. In such a case, consider using a dynamic popup which is created only when requested.

For an example popup design and usage, refer to `this tutorial`.

#### 4.1.8.1 Dynamic Popup

Dynamic popups are created at the moment they are requested so that you can prevent existence of potentially unnecessary Popups in the form hierarchy. This can save you some headache over the performance of your form.

Mind that a dynamic popup component is created on the given screen or execution level unlike the other form components which are created on the screen level: if you create a dynamic popup in a view model, the dynamic popup component cannot be accessed from the screen level.

To create a dynamic popup from a Listener handle expression or as part of a reusable form, do the following:

1. Define the popup and its content (for example `new ui::Popup p; p.child := ...`)
2. Create the popup with the call `createAndShow()`.
3. Define the logic of the popup, possibly using `clear()` and `merge()` functions.
4. To remove the popup from the component tree, call the *hideAndDestroy()* function on the popup.

```
 //This is a dynamic Popup (possibly created in a Reusable form or a component listener handl
def ui::Popup p := new ui::Popup (
  visible -> { ->
    true
  },
  listeners -> {
    new PopupCloseRequestListener(
      handle -> { e ->
        hideAndDestroy(p)
      }
    )
  }
);


 //p.visible := { -> true};
def TextBox tx := new TextBox(binding -> &varString);
~
p.child := new VerticalLayout(
      children -> [ tx ]
);
createAndShow(p);
```

More dynamic functions are available from the *dynamics.funcs* file in the *ui* module, for example, to search for components higher or lower in the hierarchy, such as, `findTopmostComponents()`, `findTopmost↩Containers()`, `getChildren()`, etc.

#### 4.1.8.2 Closing a Popup

You can close a popup by setting its visibility to `false` and refreshing it from any listener so it disappears or with the *hideAndDestroy()* function call on the popup.

If you want to close your popup with an X button in its caption, define the behavior in the popups *PopupCloseRequestListener*.

```
def ui::Popup p :=
 new ui::Popup (
   visible -> { -> true },
   listeners -> { new PopupCloseRequestListener(
     handle -> { e ->  hideAndDestroy(p) }
     )
   }
 );
```

### 4.1.9 Dashboard

The Dashboard component  is a container component that can hold multiple widget components.

When working with a rendered dashboard in the client application, use the plus (+) button on the dashboard to display a widget. Visualized widgets can be positioned anywhere within the dashboard, resized, and visualized as necessary. Only one instance of the given widget can be visualized on the dashboard.

Note that dashboard has a toolbar, which can contain any form element apart from a widget.

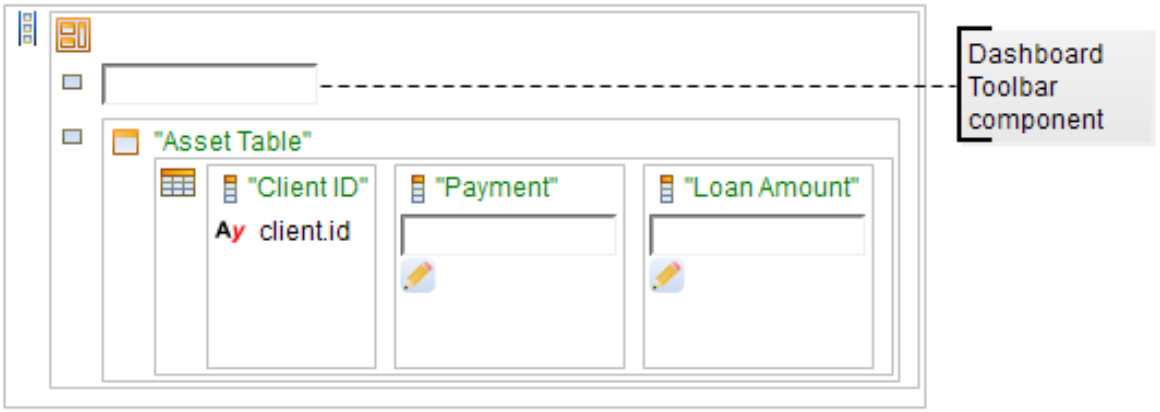



**Figure 4.4 Form with a Dashboard component**

**Note:** By default, dashboard size adapts to the requirements of the visualized widgets with the minimum size 650x490 px. To alter the size, use presentation hints on the dashboard (refer to the Standard Library document).

#### 4.1.9.1 Dashboard Widget

The Dashboard Widget component  is a form component that is rendered as a widget on a dashboard: Therefore it must have the Dashboard component as its parent. A Dashboard Widget can hold one form component.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *WidgetChangeEvent* when the user adds a widget to the dashboard, resizes, moves, or hides a widget

  The event contains information on the widget properties in the WidgetConfiguration field.

By default, when a dashboard is visualized, it remains empty. The user can then add the available widgets to the dashboard using the plus (+) button on the dashboard. To visualize a widget immediately when a dashboard is displayed, define the widget's Configuration property: the property is defined as its WidgetConfiguration property.

For example, if the Configuration property is set to `new WidgetConfiguration(visible -> true, width -> 300, height -> 300, top -> 300, left -> 350)`. The widget will be visible when the dashboard is displayed; its size will be 300x300 pixels; the coordinates of the upper left corner will be 300:350 (its upper left corner will be positioned 300 pixels below the top of the dashboard and 350 pixels to the right).

The Dashboard Widget component has the following properties:

- **Title**: widget title
- **WidgetID**: ID used in the WidgetChangeEvent
- **Configuration**: WidgetConfiguration object defining the visualization properties of the widget when the dashboard is visualized

  WidgetConfiguration defines the following:

  - `visibility`: boolean value that defines whether the widget is displayed by default when the dashboard is displayed

    In the default Application User Interface, previously closed or invisible widgets can be visualized by clicking them under the + plus button of the dashboard.
  - `width`: default width of the widget
  - `height`: default height of the widget
  - `top`: distance of the top widget border from the dashboard top border
  - `left`: distance of the left widget border from the dashboard left border
  - `zIndex`: vertical stacking order of the widget

    When the widget produces a WidgetChangeEvent, that is whenever it is clicked, moved, or resized, the zIndex is raised so that the widget is on top of any other widgets in the dashboard.
  - `maximized`: boolean value that defines if the widget is maximized
  - `minimized`: boolean value that defines if the widget is minimized

    In the default Application User Interface, minimized widgets are available at the bottom of the dashboard.

**Note:** Note that widgets keep its last size value separate from its maximized and minimized status↩ : if you set a widget to a particular size, then maximize it, and minimize it, when restored it will be maximized. If you then double-click the widget caption, it will be restored to the original size. If you want to move a widget to the foreground, click the widget.

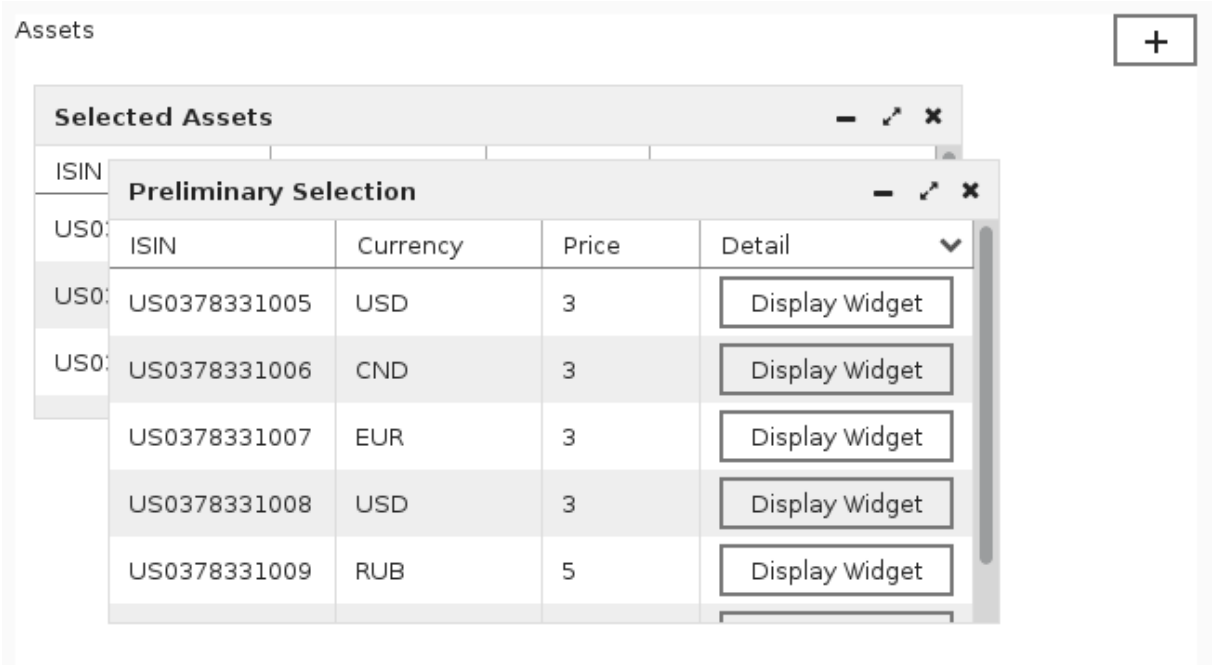



**Figure 4.5 Dashboard with Widgets**

## 4.2 Input Components

Input components serve to acquire input from the user: When the user provides input, the component produces a value change event, which can be caught by a value change listener and processed in the next response-request cycle.

### 4.2.1 Text Box

The Text Box component ( ) is rendered as single-line input field.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* when the component has changed content and loses focus (to process it also on focus loss; set the Immediate property to true)
- *ActionEvent* when the user presses ENTER while the Text Box is focused
- *AsynchronousValueChangeEvent* whenever the user inserts or removes a character

  Note that the binding of the Text Box does not have to change immediately due to possible validation; hence binding of the Text Box might appear to be out-of-date.

The Text Box component has the following properties:

- **Label**: visible text label

- **Placeholder**: input prompt (text displayed in the text box if the value of the Binding reference is null)
- **Required**: If true, a mark indicating that the value is required is rendered in the component.

  **Important:** The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless an additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.
- **Binding**: reference to a slot that holds the text value (for example, a form variable or global variable)

  If a Text Box binds to a Date type, it is rendered as the Date Picker: for the Date Picker you can define the format of the date the Date Picker will accept in the **Format** property. The available formats are defined in the `additional formats` hint.
- **Format**: required input format for values such as, date, integer, etc. defined as a string; it follows the rules of java.util.Pattern, java.text.SimpleDateFormat, and java.text.DecimalFormat

  If the entered value does not follow the format pattern, a validation mark is displayed next to the Text Box.
- **Read-only**: editability of the text

  If true, the text renders as grayed out and cannot be edited.
- **Immediate**: setting of Immediate mode

  If true, the immediate mode is active: the value change event triggers request-response cycle on focus change.
- **Help text**: tooltip text

#### 4.2.1.1 Defining Suffix on a Text Field

To add a suffix to a Text Field component, such as, `PCS`, define on the Text Field the *suffix* Hints with the value of the suffix in a String. You can do so on the *Presentation Hints* tab of the Properties view.

### 4.2.2 Text Area

The Text Area component (  ) renders as multi-line text input Area to allow the user to provide longer text input.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* when the component contains changed content and loses focus (to process it also on focus loss; set the Immediate property to true)
- *AsynchronousValueChangeEvent* whenever the user inserts or removes a character

  Note that the binding of the Text Area does not have to change immediately due to possible validation; hence binding of the Text Area might appear out-of-date.

The Text Area component has the following properties:

- **Label**: visible text label
- **Required**: If true, a mark indicating that the value is required is rendered in the component.

  **Important:** The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless an additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.

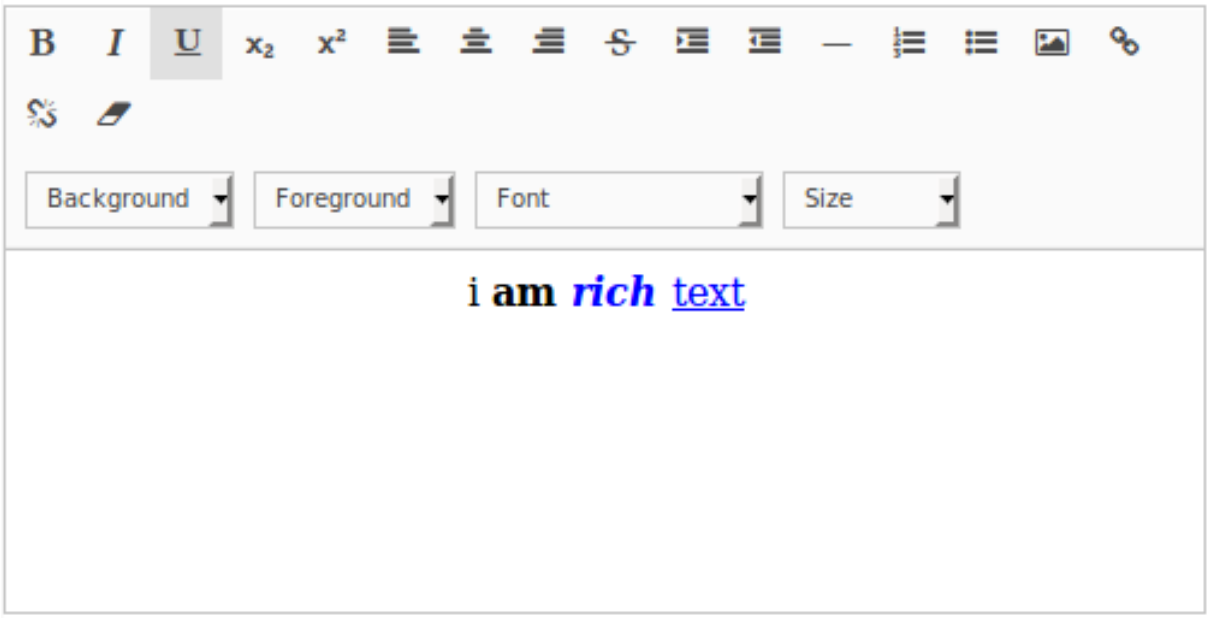
- **Binding**: reference to a slot that holds the text value (for example, a form variable or global variable)
- **Read-only**: editability of the text
- **Immediate**: setting of the Immediate mode

  If true, the immediate mode is active: the value change event triggers request-response cycle on focus change.
- **Placeholder**: input prompt (text displayed in the text area if the value of the Binding reference is null)

  If true, the text area is grayed out and cannot be edited.
- **Is Rich Text**: if true, the Text Area is rendered with a formatting toolbar

  The input text is translated into html.

  **Important:** Unlike Text Areas, Rich Text Areas don't produce AsynchronousValueChangeEvents.
- **Help text** (on the Help Text tab): tooltip text




### 4.2.3 Check Box

The Check Box component ( ) renders as a check box with a Label and allows the user to provide a Boolean value.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* when the component was selected or unselected

The Check Box component has the following properties:

- **Label**: visible label displayed next to the checkbox
- **Required**: The property is not applicable for Check Box components.

- **Binding**: reference to a slot that holds the Boolean value (for example, a form variable or global variable)
- **Read-only**: editability of the check box

  If true, the check box renders as grayed out and cannot be edited.
- **Immediate**: setting of the Immediate mode

  To process the ValueChangeEvent at the moment the user clicks the check box, set the Immediate property to `true`.
- **Help text**: tooltip text

### 4.2.4 Combo Box

The *Combo Box* component (  ) allows the user to select one option from a set of options available in a drop-down list. The selected option is stored and taken from the binding of the combo box.

> **Note:** If the value stored in the binding is not in the current options, the value is not displayed.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* when the user selects an option

The Combo Box component has the following properties:

- **Label**: visible label above the combo box
- **Placeholder**: input prompt (text displayed in the text field if the value of the Binding reference is null)
- **Required**: If true, a mark indicating that the value is required is rendered in the component.

  > **Important:** The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless an additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.
- **Binding**: reference to a slot that holds the selected option value (for example, a form variable or global variable)
- **Read-only**: editability of the combo box

  If true, the combo box renders as grayed out and no option can be displayed or selected.
- **Immediate**: setting of the Immediate mode

  To process the ValueChangeEvent at the moment the user select an option, set the Immediate property to 'true'.
- **Options**: list of options

  The list items are displayed in the drop-down. Make sure the defined expression resolves to an object of type List<ui::Option>.

  Note that the Option data type has the value and label field so that you can define the label displayed in the drop-down list that represents the given value.

```
 collect(
  literals(type(AssetType)),
  { e ->
    new ui::Option(
       value -> e,
       label -> literalToName(e)
    )
  }
)
```

- **Create new option**: closure that is called if the user enters a custom value

  When the user enters a custom value in the text field of the combo box, the closure is called with the value as its argument. The closure returns an object that is set to the binding value.

  > **Important:** An event that creates a new option is created only when the user presses ENTER after they input the new value.

- **Help text**: tooltip text

### 4.2.5 Lazy-Loading Combo Box

The *Lazy-Loading Combo Box*  component allows the user to select one option from a set of options available from a drop-down list, just like the regular Combo Box. However, unlike in the Combo Box component, the options in the drop-down are paged and queried as the user enters parts of the option so the component is useful if you need to select an item from many options.

It produces:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* when the user selects an option

When creating a Lazy-Loading Combo Box component, define the following properties:

- **Label**: visible label
- **Required**: If true, a mark indicating that the value is required is rendered in the component.

  > **Important:** The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless an additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.

- **Binding**: slot that holds the selected option value, such as a form or global variable
- **Read-only**: editability of the combo box

  If true, the combo box renders as grayed out and is disabled.

- **Immediate**: setting of the Immediate mode

  To process the ValueChangeEvent at the moment the user clicks an option, set the Immediate property to 'true'.

- **Options**: list of options

  The closure returns a List<Object>, where each list object represents an option. What is actually displayed as a label for the option is defined in the Formatter property.

  The closure has the following input parameters:

  - value in the text field (the value is a String and can serve to filter out the options in the drop-down list)

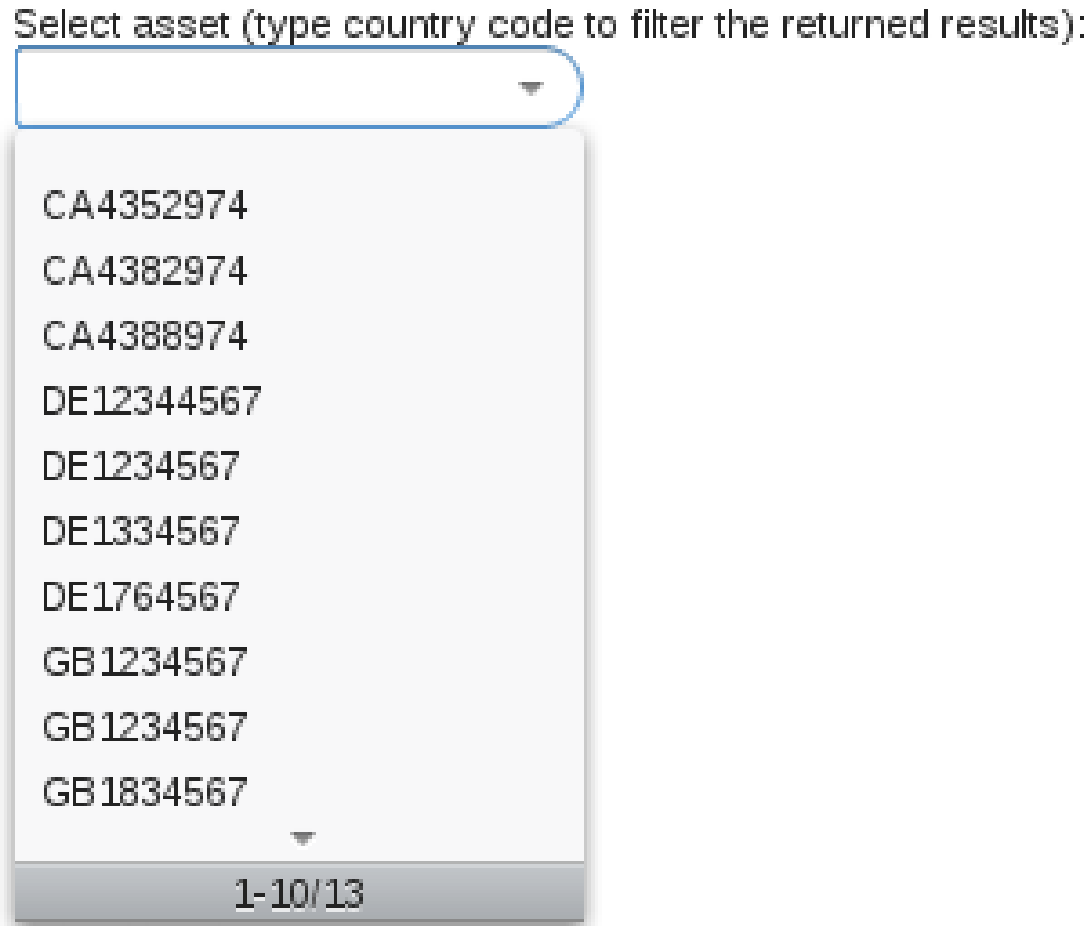



**Figure 4.6 Lazy-Loading Combo Box with no value in the text field**

  - index of the first option (options of items starting from this index can be displayed)
  - displayed option count (the number of options from the first option to be displayed)

- **Option count**: count of displayed options

  Make sure the defined closure returns an Integer value. Typically you want to use a count query.

```
{code -> getCurrencies_count(code, null, null)}
```

- **Formatter**: closure that returns a string that is displayed in the drop-down list

  The string represents the respective object from the Options.

```
{c:Currency -> c.currencyCode}
```

- **Create new option**: closure that is called when the user enters a custom value

  If the user enters a custom value in the combo text field of the combo box, the closure accepts the value as its parameter and returns it to the binding entity.

- **Placeholder**: input prompt (text displayed in the text field if the value of the Binding reference is null)
- **Visible**: whether the component is displayed (if not, the component is not initialized)
- **Help text**: tooltip text in the *Help Text* tab

#### 4.2.5.1 Creating a Lazy-Loading Combo-Box

Typically, a lazy-loaded combo box offers the available options as the user inputs characters: in the background, the options need to be loaded in batch required by the combo box.


**Figure 4.7 Lazy-Loading Combo Box with options for the user input**

To create such a lazy-loading combo box, do the following:

1. Define the *Binding* property with the target slot for the selected option.
2. Define the *Options* property so that the closure returns the results for what the user typed into the combo box. The user input is passed as the first closure parameter.

   ```
    {
   userInput, firstPageItem, batchSize ->
      getCurrenciesContaining(userInput, firstPageItem, batchSize)
   }
   ```

   In the example above, the query is defined so as to return:

   - only results that contain the user input, for example, you can define a `standard query` with a condition, such as, `currentCurrency.code like userInput +"*"`
   - and that in batches defined by the closure input parameters `firstPageItem` and `batchSize`
3. Define the *Options count* property so that the closure returns the number of options returned by the *Options* property.

   ```
   {userInput -> getCurrenciesCountForFiltered(userInput)}
   ```

   Consider using the `count query` to obtain the options count.
4. Define the *Formatter* property so that it returns the string displayed in the drop-down menu.

   ```
   { c:Currency -> c.code }
   ```
5. Define the *Placeholder* property with the string that should be displayed if no option is selected. Alternatively, in the *Formatter* property handle the object for null value.

### 4.2.6 Single-Select List

The Single-Select List component (  ) is a form component that displays a list of options and allows the user to select exactly one option.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* when the user selects an option (to process it also, set the Immediate property to true)

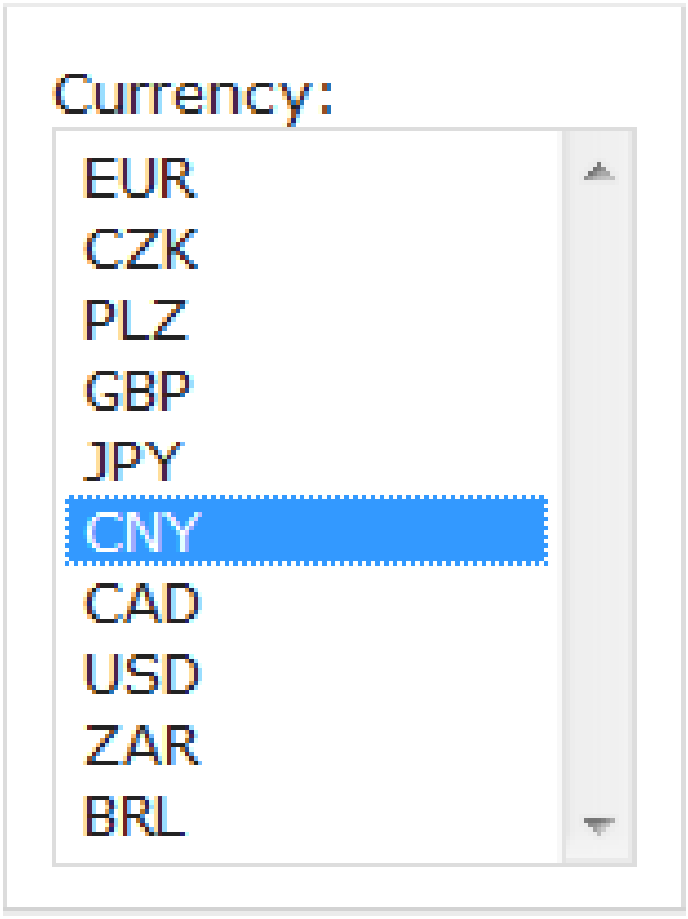



**Figure 4.8 Single Select List in a Panel**

- **Label**: visible label
- **Required**: If true, a mark indicating that the value is required is rendered in the component.

  **Important:** The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless an additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.

- **Binding**: reference to a slot that holds the selected option value (for example, a form variable or global variable of the respective type)

- **Options**: list of options

  The options are displayed as a vertical list. Make sure the defined expression resolves to an object of type List<ui::Option> object.

  Note that the Option data type has the value and label field so that you can define the label displayed in the drop-down list that represents the given value.

- **Read-only**: editability of the Single Select List

  If true, the list renders as grayed out and no option can be selected.

- **Immediate**: setting of the Immediate mode

  To process the ValueChangeEvent at the moment the user selects an option, set the Immediate property to 'true'.

- **Help text**: tooltip text

### 4.2.7 Multi-Select List

The Multi-Select List component (  ) is a form component that displays a list of options and allows the user to select multiple options.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* when the user selects an option (to process it also, set the Immediate property to true)

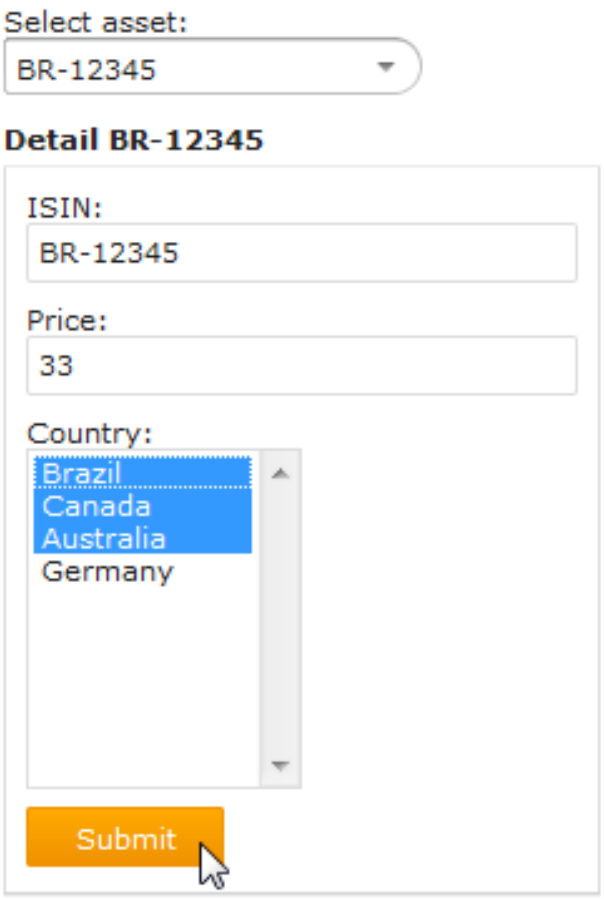



**Figure 4.9 Multi-Select List**

- **Label**: visible label

- **Required**: If true, a mark indicating that the value is required is rendered in the component.

  **Important:** The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless an additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.

- **Binding**: reference to a set of objects that holds the selected values

- **Options**: list of options

  The list is displayed in the drop∗down menu. Make sure the defined expression resolves to an object of type List<ui::Option>.

- **Read-only**: editability of the Multi Select List

  If true, the list renders as grayed out and no option can be selected.

- **Immediate**: setting of the Immediate mode

  To process the ValueChangeEvent at the moment the user selects an option, set the Immediate property to 'true'.

- **Help text**: tooltip text

### 4.2.8 Check-Box List

The Check-Box List component is a form component that displays a list of options with check boxes and allows the user to select multiple options using the check boxes.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden

- *ValueChangeEvent* when the user selects an option (to process it also at this point, set the Immediate property to true)


**Figure 4.10 Check-Box List in a Panel component**

- **Label**: visible label
- **Required**: If true, a mark indicating that the value is required is rendered in the component.

  > **Important:** The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless an additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.

- **Binding**: reference to a set of objects that holds the option values
- **Options**: list of options

  The list is displayed as entries in the check list. Make sure the defined expression resolves to an object of type List<ui::Option> object.

- **Read-only**: editability of the Check Box List

  If true, the list renders as grayed out and no option can be selected.

- **Immediate**: setting of the Immediate mode

  To process the ValueChangeEvent at the moment the user selects an option, set the Immediate property to 'true'.

- **Help text**: tooltip text

### 4.2.9 Radio-Button List

The Radio-Button List component is a form component that displays a list of options and allows the user to select exactly one option by clicking a radio button.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* when the user selects an option (to process it also at this point, set the Immediate property to true)


**Figure 4.11 Radio Button List in the Panel component**

- **Label**: visible label
- **Required**: If true, a mark indicating that the value is required is rendered in the component.

  **Important:** The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless an additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.

- **Binding**: reference to a slot that holds the selected option value (for example, a form variable or global variable of the respective type)
- **Options**: list of options

  The options are displayed as a vertical list with radio buttons on the left. Make sure the defined expression resolves to an object of type List<ui::Option>.

- **Read-only**: editability of the Radio Button List

  If true, the radio button list renders as grayed out and no option can be selected.

- **Immediate**: setting of the Immediate mode

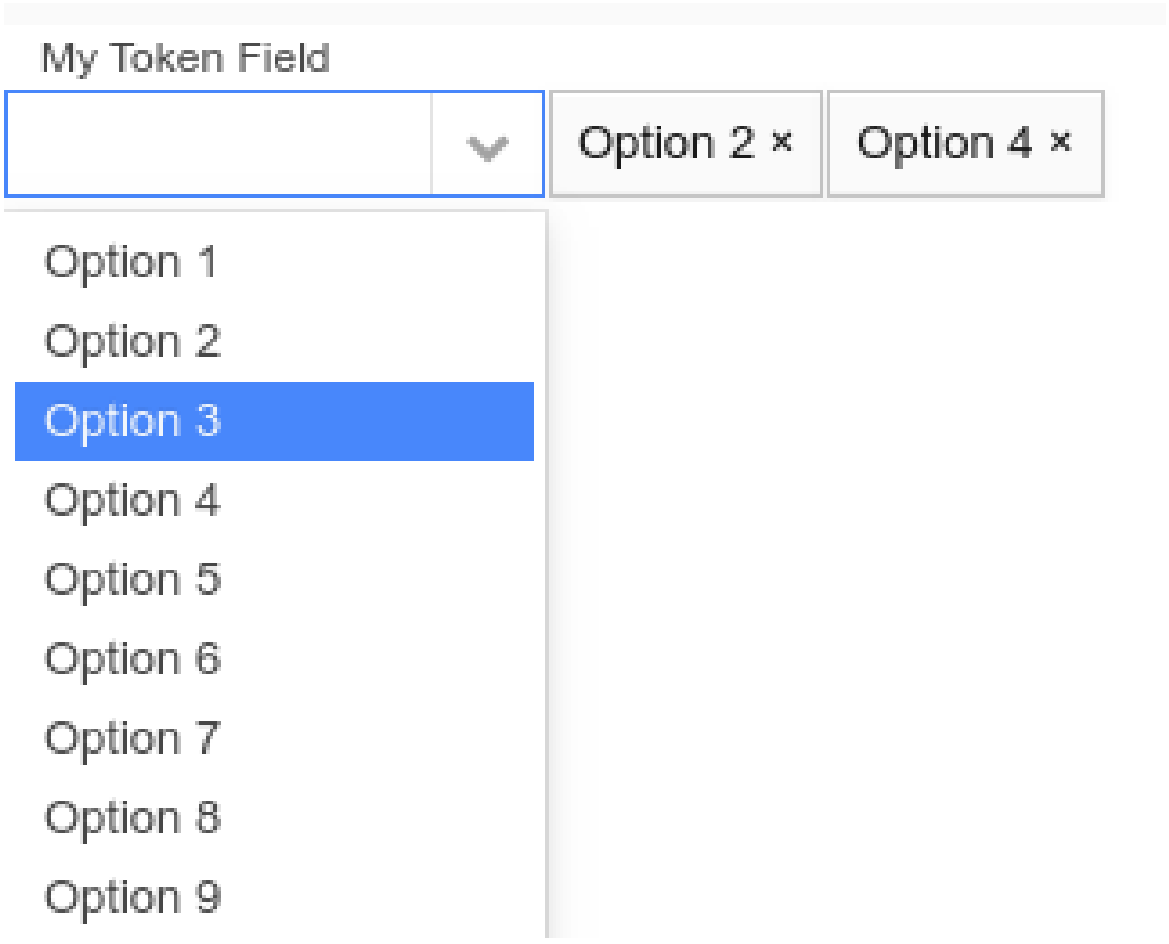
  To process the ValueChangeEvent at the moment the user selects an option, set the Immediate property to 'true'.

- **Help text**: tooltip text

### 4.2.10 Token Field

The Token Field component (  ) serves for selection of multiple options. Whenever the user selects or removes an option, the component fires a ValueChangeEvent.

The component is rendered by default as a combo box, so the user can click the arrow on the combo box to display the options. However, you can render the Token Field as a Text Field by setting its `html-class` presentation hint to `tokentextfield`.




- **Label**: visible label
- **Required**: If true, a mark indicating that the value is required is rendered in the component.

  **Important:** The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless an additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.

- **Binding**: reference to a slot that holds the selected options

  Typically a form variable of the collection type that holds the same type as the option value, for example, `Set<String>`; mind that validation does not Check the object type in the set.)

- **Read-only**: editability

  If true, the token field is grayed out and no option can be selected.

- **Immediate**: setting of the Immediate mode

  If true, the Immediate mode is active: any value changes are processed immediately.

- **Options**: list of options

  **Option expression**

```
 collect(
  1..10,
  { x:Integer -> new Option(label -> "Option " + x, value -> "Value " + x) }
)
```

  The options are displayed when the user enters characters that appear in one of the option values.

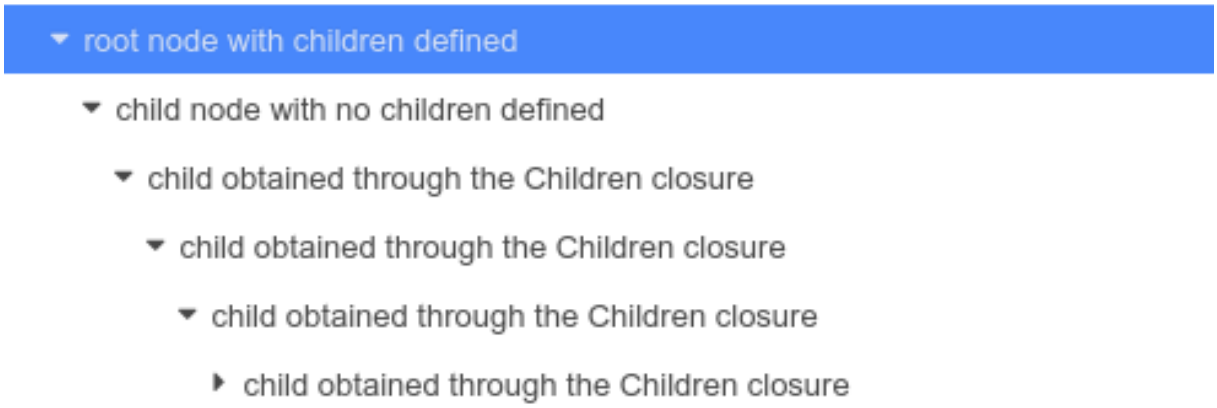
- **Help text**: tooltip text

  Define the Help text on the Help Text tab in the Properties view.

### 4.2.11 Tree

The *Tree* component (  ) allows the user to select a node from a expandable tree structure. The node is defined as a *TreeItem* object and can be expandable: When expanded, the Tree component produces a TreeEvent and tries to render child nodes:

1. If the *children* property of the *TreeItem* are defined, the child nodes are obtained using this property.
2. If the *children* property of the *TreeItem* is `null`, the Tree calls the closure defined in the *Children* property to acquire the child nodes.




When the user selects a node, the object defined in the *data* property of the TreeItem is stored in the *Binding* object and the Tree produces a ValueChangeEvent.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed when previously hidden
- *TreeEvent* when the tree node is expanded or collapsed
- *ValueChangeEvent* when the user selects a node

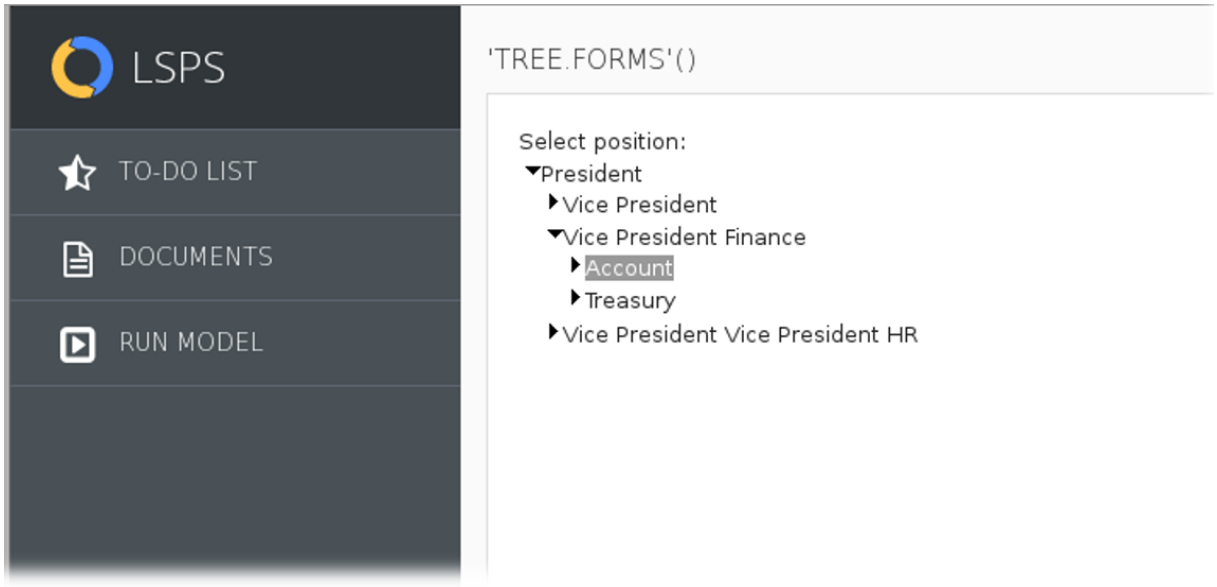



**Figure 4.12 Tree with Multiple Nodes Expanded**

The Tree component has the following properties:

- **Label**: visible label
- **Required**: If true, a mark indicating that the value is required is rendered in the component.

  **Important:** The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless an additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.
- **Binding**: reference to a slot that holds the selected tree option value (for example, a form variable or global variable)
- **Read-only**: editability

  If true, the tree renders as grayed out and no option can be selected. The selection uses the value of the binding slot.
- **Root**: collection of TreeItem that are root items of the tree
- **Children**: closure that returns a list of tree items which are displayed when a node is expanded

It is called if the children property of a *TreeItem* is `null`.

- **Immediate**: the setting of the immediate mode of ValueChangeEvents

  To process the ValueChangeEvent at the moment the user selects a node, set the Immediate property to 'true'.
- **Help text**: tooltip text

### 4.2.12 File Upload

The File Upload component allows the user to upload a file to the server. It renders as an input field for a file path and a Browse and Upload button.

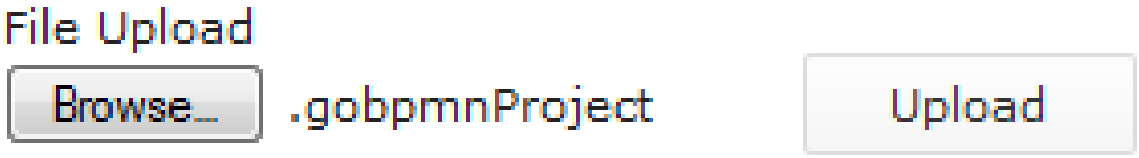



**Figure 4.13 File Upload**

It produces events of the following events:

- *InitEvent* produced when the component is initialized or displayed if previously hidden
- *ValueChangeEvent* produced when the user inserts a file name
- *ActionEvent* produced when the file upload is requested (on the upload button click)
- **FileUploadEvent** produced when file has been uploaded

  Since file upload is an asynchronous process, on runtime, the component produces an ActionEvent when the upload button is clicked and an FileUploadEvent when the uploading finishes.
- **Label**: visible label
- **Required**: If true, a mark indicating that the value is required is rendered in the component.

  > **Important:** The property **does not provide any validation** whether the field contains a value and the user will be able to submit a null value unless an additional validation mechanism is defined. Such a validation can be implemented on a listener, possibly as a validation expression.
- **Binding**: reference to a set of objects that will hold the files
- **Read-only**: editability of the File Upload

  If true, the component is disabled and grayed out and no file can be selected.
- **Immediate**: setting of the Immediate mode

  To process the ValueChangeEvent at the moment the user selects a node, set the Immediate property to 'true'. In immediate mode, the Browse button is no longer be available and the user is prompted to select the file when they click the Upload button. When they select the file, the file is uploaded immediately.
- **Button text**: label of the upload button
- **Multiple**: setting for enabling upload of multiple files

  > **Important:** This option is currently not supported since native HTML forms do not allow selecting multiple files in one window.
- **Upload to memory**: If true, the file is uploaded to memory. If false, the file is uploaded to LSPS_BINARY_↩ DATA table in the application database.
- **Delete temp data**: if true, the files from the LSPS_BINARY_DATA are deleted when the respective HTTP session finishes, by default, when the user logs out. The setting is applicable only if *Upload to memory* is set to `false`.
- **Help text**: tooltip text

## 4.3 Output Components

Output components serve to display data in a certain way and include the following:

- Output Text
- Tabular Components
- Grid
- Repeater
- Image
- Navigation Link
- File Download
- Charts
- Browser Frame
- Calendar
- Map Display

### 4.3.1 Output Text

The Output Text component is rendered as a read-only, single-line text field.

- **Label**: label of the output box
- **Content**: object that is displayed in the output box
- **Format**: formatting applied on the object defined in the Content property

  The applicable format depends on the data type of the object from the Content property:

  - Format for a **Date** object: as java.text.SimpleDateFormat, for example `"EEE, MMM d, ''yy"`
  - Format for Integer and Decimal: as java.text.DecimalFormat, for example `"#0.00"`
  - Format for a **String** can be set to:
    - `html`: HTML tags are supported.
    - `plaintext`: Line breaks in the Content object are ignored.
    - `preformatted`: Line breaks are respected.
  - Any objects of **other types** are transformed with the `toString()` function and no Format setting applies.

The component throws *InitEvent*.

### 4.3.2 Tabular Components

Tabular components include the Table and Tree Table components which share common features: they serve to display data in a tabular manner and have the Column components as their immediate children. Both work with the same types of data sets and support ordering and filtering of their content.

#### 4.3.2.1 Table

The **Table** component displays the data of its data set in the child components of the table's columns: The system iterates through individual objects of the data set and each object becomes the *data iterator* value for a row. The iterator is then used by the child components of table columns to access the data object.

It supports ordering, filtering, and grouping of the data.

The way the data set is acquired and the way the table is rendered is defined by the table type:

- **simple**: The table is on one page and the data is obtained in a single request.
- **lazy**: The table is rendered with a particular size and with a scroll. The data is loaded as the user scrolls through the table.
- **paged**: The table is rendered with a page navigation at the bottom. The data is acquired per page.

The data set itself can be defined as the following:

- **Type**: shared record or a shared record field

  The system fetches all shared record instances from the database.
- **Query**: query that returns a collection of objects

  The table iterates through the collection objects.
- **Collection**: collection of the data object

  The table iterates through the collection objects.
- **Data**: parametric closure that returns the collection of data objects

  The first parameter holds the index of the first entry; The second parameter is the count of entries per load or page. You will use this option typically when integrating with other systems, for example, `{currentIndex, count -> getEntryBatch(currentIndex, count)}`.

  This data kind requires the **Data count** property, expression that returns the total amount of entries to be loaded.
- **Generic**: an Object that results in any of the above on runtime

  The setting is used for generic reusable tables when the user wants to fill the same table with different data queried in different ways, typically when reusing the form in other forms.

##### 4.3.2.1.1 Creating a Simple Table

A simple Table is rendered as a table on a single page with all its items.

To create a simple Table, do the following:

1. Insert the Table component in a Form.
2. On the *Details* tab of the Properties view, define the following:
   - Data Kind: data source type and the related data
   - *Data Iterator*: reference to a local form variable of the same type as your Data objects.

     **Important:** The data iterator can be used solely as the table iterator: Using the iterator out of the Table only when creating the table dynamically. The practise is generally discouraged and results in a validation problem.
   - *Type*: `TableType.simple`
3. Insert Table Columns into the Table.
4. Into each column, insert the required components and bind them to the iterator data.

#### 4.3.2.1.2 Creating a Paged Table

A paged table data is rendered on pages with the number of rows defined by the *initial-page-size* presentation hint and page navigation at the bottom. It can be set to load a particular page using the *Show Index* property.

A paged table produces the **TablePageSizeChangeEvent** when the user changes the size of the table page.

To create a paged Table, do the following:

1. Select the Table component in a Form.
2. On the *Details* tab of the Properties view, define the following:
   - Data Kind: data source type
   - *Type*: `TableType.paged`
   - *Data Iterator*: reference to a local form variable of a type that can hold your Data objects

     **Important:** The data iterator can be used solely as the table iterator: using the iterator out of the Table results in a validation Error.
   - *Show index*: the index number of the page that should be opened on load.
   - *Index Iterator*: reference to an object that will hold the index of the row where the current Data object is used
3. Set the size of the page in the `initial-page-size` presentation hint.
4. Insert Table Columns into the Table.
5. Into each column, insert the required components and bind them to the iterator data.

#### 4.3.2.1.3 Creating a Lazy-Loaded Table

A lazy table is rendered as scrollable table with the number of rows defined by the *initial-page-size* presentation hint.

To create a Table, do the following:

1. Select the Table component in a Form.
2. On the *Details* tab of the Properties view, define the following:
   - Data Kind: data source type
   - *Type*: `TableType.lazy`
   - *Data Iterator*: reference to a local form variable of a type that can hold your Data objects

     **Important:** The data iterator can be used solely as the table iterator: using the iterator out of the Table results in a validation Error.
   - *Show index*: the index number of the page that should be opened on load.
   - *Index Iterator*: reference to an object that will hold the index of the row where the current Data object is used
3. Insert Table Columns into the Table.
4. Into each column, insert the required components and bind them to the iterator data.

#### 4.3.2.1.4 Defining Grouping on a Table

Groups serve to group the items in a table on multiple levels according to their properties in a tree-like way, for example, a list of songs according to their interpret and then according to the album.

Note that grouping takes place *on the data set* and renders the table type setting irrelevant.

The groups are defined as a subtype of the abstract `GroupSpec` record and that as one of the following:

- **PropertyGroupSpec**: the groups are defined as Record properties;

```
 [
//first-level grouping:
new PropertyGroupSpec (
    label -> "Surname",
    groupBy -> Author.surname),
    //second-level grouping:
    new PropertyGroupSpec
          (label -> "First name",
          groupBy -> Author.firstName)
]
```

- **ClosureGroupSpec**: the groups are calculated on runtime in a closure;

```
 [new ClosureGroupSpec(
   label -> "Surname",
   groupBy -> {a:Author -> a.surname}
   )
]
```

- **OptClosureGroupSpec**: the groups are calculated on runtime in a closure in optimized manner;

```
 [new OptClosureGroupSpec(label -> "Surname",
      groupBy ->
        { c:Collection<Author > ->
        def Set<String> surnames := toSet(collect(c, {a:Author -> a.surname}));
        //creates map with surname as the key and collection of authors as value
        map(surnames, {surname:String -> select(c, {a:Author -> a.surname == surname})})}
    )
]
```

Alternatively, if you are using the Type data kind, you can infer grouping: The system will "guess" the applicable groups. The Group Spec then becomes a collection of all bindings of child components of all columns. Therefore, grouping is inferred only on columns with one Input component with binding set to a simple data type entity.

To define grouping on a table, do the following:

1. In your form definition, select the table component.
2. In the Properties view, open the *Grouping* tab.
3. Uncheck Disable Grouping.
4. Define the grouping:
   - For your own grouping:
     (a) Define the GroupSpec and optionally Grouping that holds the currently applied grouping.
     (b) Optionally, on individual table columns, define the Group value on their Grouping tab. The value is used as a group header in the line that serves to expand and collapse the group.
     `{values:Collection<Author>->values[0].gender}`

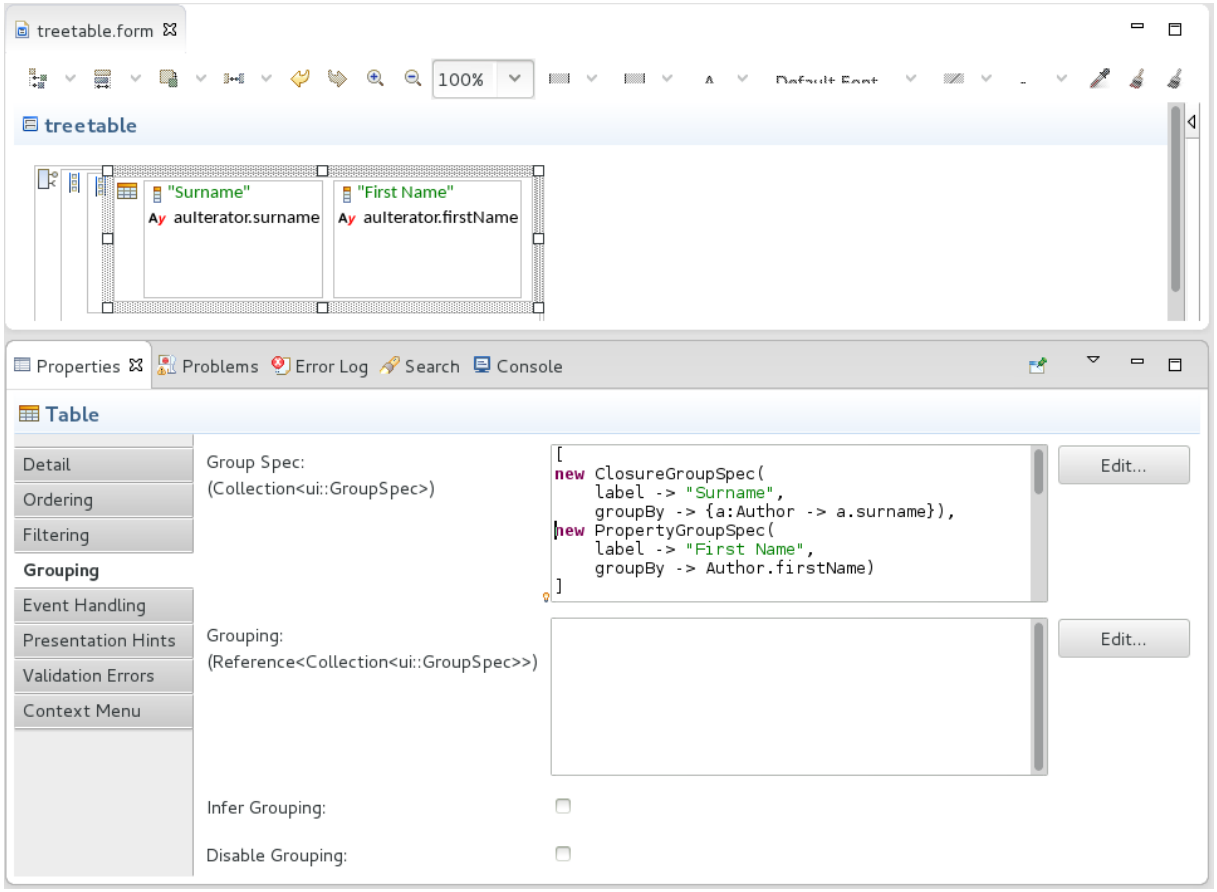



**Figure 4.14 Form with Table and its Grouping Definition**

- To infer grouping, select **Infer Grouping**.

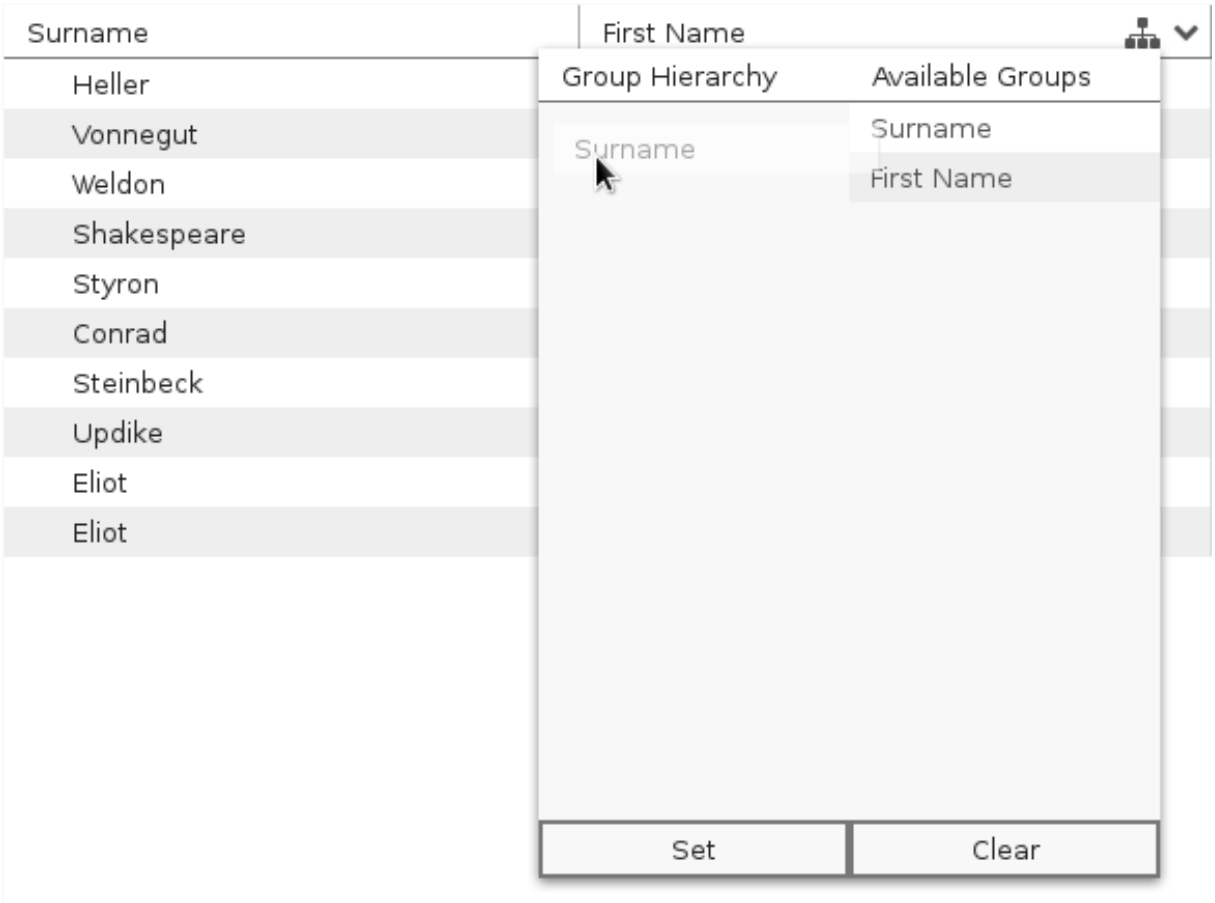
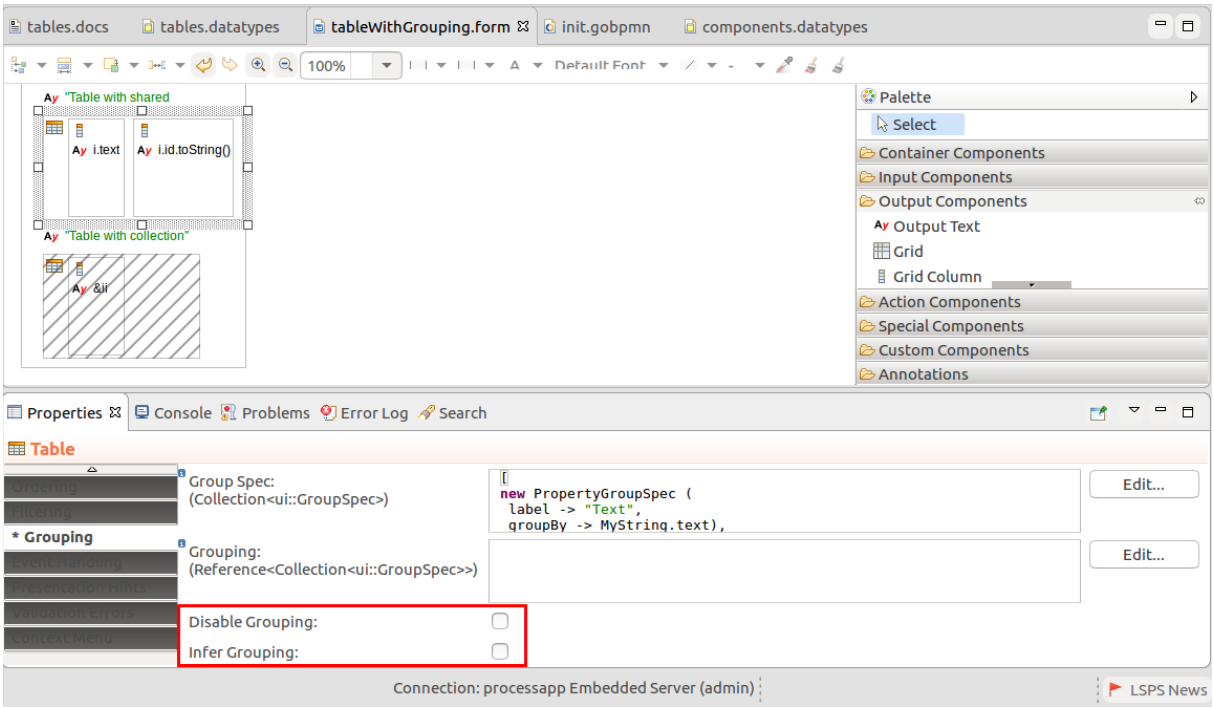



**Figure 4.15 Setting Grouping**

#### 4.3.2.1.5 Disabling Grouping on Tables

To disable grouping on a table, do the following:

1. In the Properties view of the Table component, go to the Filtering, Ordering, or Grouping tab.
2. Unselect the applicable grouping option:
   (a) Unselect **Disable Grouping** to disable grouping altogether.
   For the Columns with inferring of grouping, the inferring will be disabled. Other Grouping settings will be applied.
   (b) Unselect **Infer Grouping** to disable inferred grouping.




#### 4.3.2.2 Tree Table

The *Tree Table* component renders as a table with rows organized in a tree structure. The rows of the collapsible tree nodes are loaded on expand.

Individual nodes are represented by *TreeItem* objects, which hold the business data and information about the tree node and its child elements: The tree table component iterates through a collection of the root <TreeItem> objects. If a root object defines children, it iterates also through the child elements. Every element becomes the object for a row.

Tree Tables support ordering and filtering of their content.

The TreeItem objects define the following:

- data: business object the tree table item operates over (business data of a row)
- label: label you can use as component content
- expanded: whether the node is expanded by default
- parent: parent TreeItem object (element "above")
- children: child TreeItem object (elements "inside")

You can access the data through the *iterator* or *tree item iterator*: the iterator holds only the business data object, while the tree item iterator holds its TreeItem object.


**Figure 4.16 Tree Table with expanded nodes**

The Tree Table component has the following properties:

- **Root**: collection of root TreeItem elements
- **Children**: closure that returns a list of tree table items that are loaded when a node is expanded

  The closure is called when the children in the Root expression are null; note that when *children* is an empty collection, the node is considered a leaf and the closure is not called.
- **Iterator**: reference to an object of the business data type

  The iterator will hold the *data* object of the given row so it has to be able to hold any business data type used in the Tree Table: if the table uses different records, consider using the *Record* type or another parent data type.

  > **Important:** The data iterator can be used solely as the table iterator: using the iterator out of the Table results in a validation Error.
- **Tree Item Iterator**: reference to an object of the TreeItem type

  It will hold the TreeItem object of the current row.

  > **Important:** The iterator can be used solely as the table iterator: using the iterator out of the Table results in a validation Error.

#### 4.3.2.2.1 Creating a Tree Table

To create a Tree Table, do the following:

1. Insert the Tree Table component in a Form.
2. Define the *Root* property with the collection of the root nodes and possibly their children.

```
 [new TreeItem(
//business object
    data -> boss,
    label -> "Boss",
    expanded -> true,
    parent -> null,
    children -> [
```

```
            new TreeItem(
                data -> employee,
                label -> "Employee",
                expanded -> true,
                parent -> null,
                //if children is null, like below, then the Children property is used to retrieve
                children -> null)
        ]
    )
]
```

3. Define the Children property (the closure that returns the child objects for root objects with null value in *children*).

```
{ treeitem -> collect(getAllChildren(treeitem), convertToTreeitems())}
```

4. Define the Iterator and the TreeItem iterator: typically these are references to local form variables.
5. Insert Table columns into the Tree Table.
6. Insert the required components into the Columns, typically, *OutputTexts*, *Text Boxes*, etc.
7. Define the content of the Column components using the iterators where applicable.

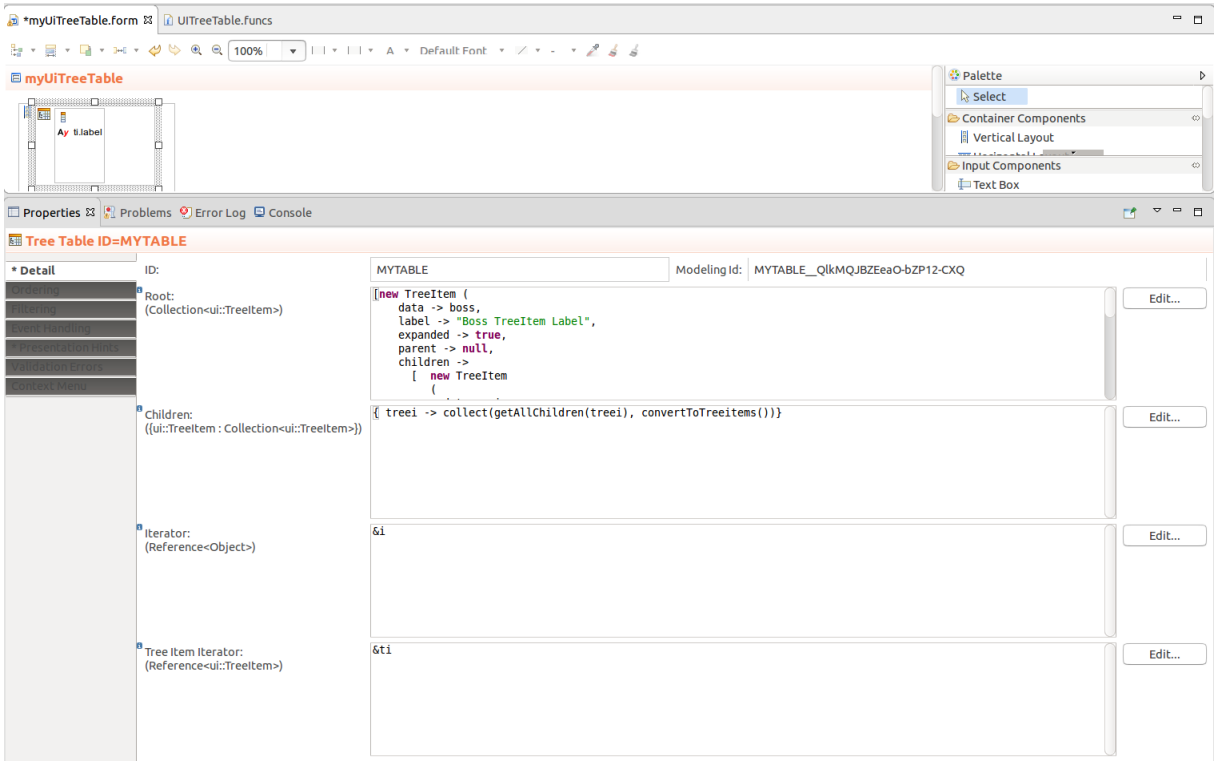



**Figure 4.17 Tree Table form with its Detail properties**

#### 4.3.2.3 Table Columns

The *Table Column* component can represent column of a Table or a Tree Table. It defines table column properties, including ordering and filtering options. Its children are components that can operate over the Table business data.

To define how children of a column are aligned use the `align` presentation hint: note that the hint is applied on the header as well.

**Warning:** Note that refreshing a table column does not refresh its underlying data set: only its header, hints, and any data not related to the data set, such as global variable references, are recalculated. This is justified by the fact that if you managed to refresh only the data set for a column, the table could be populated with inconsistent data. To refresh the underlying data set of a column, either refresh the parent table or the content of the column child components.

The Table Column component has the following properties:

- **Header**: column header
- **Visible**: visibility of the table column

  If false, the column is not visible or available in the column picker.

  The expression is evaluated whenever the column's parent component is refreshed.

#### 4.3.2.3.1 Collapsing a Table Column

To hide a Table Column, that is to display it collapsed, set the `hidden` presentation hint to true: this setting will be applied when the form is initialized.

#### 4.3.2.3.2 Saving Column Width and Collapsed State

To save the column width in a table and the collapsed state of a table when a document or a to-do is saved, call the `getColumnStates(<MYTABLE>)` function and store the returned column states. When applicable, call the `restoreColumnStates(<MYTABLE>, <COLUMNSTATE>)` function to restore the column widths and collapsed states.

### 4.3.2.4 Ordering and Filtering of Tables and Tree Tables

Tables and Tree Tables support

- ordering: sorting of data in an ascending or descending order when the user clicks a column label
- filtering: filtering of data so that only entries that meet the filter requirements are displayed

The features are enabled on the Table and Tree Table components; however, since their logic applies to Columns, that is where they are actually defined.

All the features allow their inferring, that is, setting up the feature automatically as best effort for the given Column data.

#### 4.3.2.4.1 Defining Ordering

Ordering is defined on the Column components in Tables or Tree Tables. By default, Columns are set to infer ordering so the feature is automatically set up. However, inferring is applicable only on Columns with exactly one input component and with their binding set to a simple data type. For columns with multiple or other than input components, the inferring of ordering is not supported. Also, if the data set is obtained as Data, you will need to define the ordering manually.

> **Note:** Ordering cannot be applied on columns with enumeration values. If you want to order a table according to an enumeration, use a query to acquire the data set already sorted. You can then define the enumeration as the sorting property on table columns.

To define ordering on a table, do the following:

1. Define the ordering properties on the Ordering tab of the Table or Tree Table:
   - **Ordering**: reference to a variable that holds the ordering applied to the table at the given moment
   - **Infer ordering**: if checked, inferring of ordering on the table is enabled (Columns can infer filtering).
     Ordering can be inferred only on tables that define a shared Record type as its data kind and that only on columns that contain exactly one input component (the infer guess is performed based on the binding reference of the component).

   Once you have defined ordering properties on the table, you will need to define ordering on table columns.
2. Define the ordering properties on individual columns: On the Ordering tab in the column Properties view, select the Ordering kind and define the respective Order By expression:
   - Property: the expression must return a record field of a simple data type. The returned type must be present in the row scope type. The table data will be sorted according to the values of this field in row scopes when the user clicks the column header.

     ```
     //Author is a shared record that is part of the row scope and surname is its field.
     Author.surname
     ```
   - Expression: the closure expression must return a field of a simple data type. The input closure parameter is the row scope. The returned type must be present in the row scope type. The table data will be sorted according to the values of this field in row scopes when the user clicks the column header.

     ```
     //a incoming parameter is the row scope, in this case, the Author instance; the closu
     {a:Author -> a.surname}
     ```
   - Enumeration: the name of the ordering enumeration. This setting applies only to data acquired using non-native *query* and defines entered ordering enumeration.
   - Infer: no Group By expression applies. Inferring is applied only on the particular column.
     If setting the value to **Infer**, make sure inferring of ordering is enabled on the parent table or tree table.
   - Disabled: ordering is disabled on the particular column.

#### 4.3.2.4.2 Tracking Current Ordering on Tables and Tree Tables

To track and programatically change the ordering of a Table or Tree Table, define the Ordering expression on the Ordering tab of the component properties. The value is a reference to a map and will be set to values in the form `[ORDER_BY -> ORDER_DIRECTION]`, for example `[MyForm::MyRecord.id -> ui::↩ OrderDirection.Ascending]`.

Other model resources can use the variable value to detect the current ordering. Also, if you set the reference to an expression, the expression will be evaluated when the component is visualized in the front-end application and used as default ordering on the Table or Tree Table.

#### 4.3.2.4.3 Defining Filtering

Filtering on Tables and Tree Tables is by default enabled and Columns are set to infer the filtering settings: by default your Tables and Tree Tables have filtering set up: However, inferring is applicable only on Columns with exactly one input component with binding to a simple data type. For columns with multiple or other than input components, you need to define filtering manually.

To define filtering on a table, do the following:

1. Define the filtering properties on the Filtering tab of the Table or Tree Table:
   - **Filtering**: currently applied filtering expression
   - **Infer Filtering**: if checked, inferring of filtering on the table is enabled (Columns can infer filtering).

     **Note:** Filtering can be inferred only on table columns that contain exactly one input component. The guess is performed based on the binding reference.
2. Define the filtering properties on individual columns: On the Filtering tab in the column Properties view, do the following:
   (a) In the Filter UI field, define the filter type (for example, `new ui::SubstringFilterUI(substring -> "default substring"), new ui::RegExpFilterUI(regexp -> "default regex")`).
   (b) Select the Filtering type and define the respective Filter By expression:
   - **Property**: the expression must return a record field of a simple data type. The returned type must be present in the row scope type. The table data will be filtered according to the values of this field in row scopes.

     ```
     //Author is a shared record that is part of the row scope and surname is its field.
     Author.surname
     ```
   - **Expression**: the closure expression must return a field of a simple data type. The input closure parameter is the row scope. The returned type must be present in the row scope type. The table data will be sorted according to the values of this fields in row scopes when the user clicks the column header.

     ```
     //a incoming parameter is the row scope
     // in this case, the Author instance; the closure returns the instances surname
     {a:Author -> a.surname}
     ```
   - **Custom**: applicable only on Data (paged collection) custom filter definition (Filter must be of PropertyFilter, ClosureFilter, or CustomFilter)

     ```
     new CustomFilter(ui -> new TextBox(  binding -> &filterBinding),
         filterText -> {->"Surname Filter"},
         popup -> false)
     ```
   - **Infer**: no Filter By expression applies. Filtering is applied only on the particular column. If setting the value to **Infer**, make sure Inferring of Filtering is enabled on the parent component.
   - **Disabled**: filtering is disabled on the particular column.
3. On the Table, consider defining the `no-data-message` presentation hint: the hint value will be displayed in the Tables when it is empty.

#### 4.3.2.4.4 Disabling Filtering and Ordering on Tables and Tree Tables

To disable filtering, or ordering on a table, do the following:

1. In the Properties view of the component, go to the Filtering, Ordering, or Grouping tab.
2. Select the *Disable* option:
   (a) To disable any filtering and ordering, select **Disable Filtering** or **Disable Ordering**.
   (b) To disable only the inferred filtering or ordering, unselect **Infer Filtering** or **Infer Ordering**. For the Column with inferring of filtering, ordering, or grouping, the inferring will be disabled. Other Filtering, Ordering, or Grouping settings will be applied.

### 4.3.3 Grid

The *Grid* component displays single-line text data in a table layout and enables editing of the persistent data out-of-the-box.

The Grid data is lazy-loaded, which prevents potential performance issues. Unlike in Tables, no Vaadin components are created based on the content of cells: Grid columns do not contain components; they can define only how the data is presented in the Grid Column components.

If you plan on using complex content, such as images, charts, etc. consider using Table.


**Figure 4.18 Editable Grid with a validation message on incorrect value**

#### 4.3.3.1 Creating a Grid

To create a Grid with your content, do the following:

1. Insert the Grid component into your Form.
2. In the Properties view of the Grid, define the data kind:
   - **Type**: shared record or a shared record field
     The Grid iterates over all shared record instances.
   - **Query**: query that returns a collection of objects
     The Grid iterates through the collection objects.
   - **Collection**: data set is defined as a collection
     The Grid iterates through the collection objects.
   - **Data**: a collection of objects that are iterated through
     When using queries to get the data, define the input parameters for index and entries count and use them as paging definition, for example, `{currentIndex, count -> getEntryBatch(currentIndex, count)}`
   - **Generic**: an Object that results in any of the above on runtime
     Use this setting to fill one table with different data queried in different ways.
3. If you are using the *Data* data kind, define the **Data count** property with the total amount of entries to be loaded.
4. Insert the *Grid Column* components ( ) into the Grid.
5. Define the properties of every Grid Column.

#### 4.3.3.1.1 Setting Height of Grid

By default, the Grid defines a fixed minimum height. You can change the height using the *height-by-rows* presentation hint. If the grid displays more rows than the value set by the **height-by-row** hint, the grid becomes scrollable.

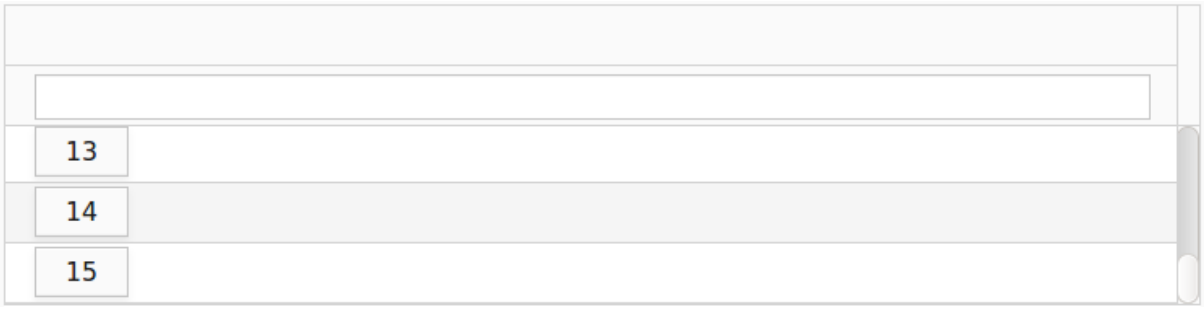



**Figure 4.19 Grid with a scroll**

### 4.3.3.2 Defining a Grid Column

A Grid Column is the child component of the Grid which displays the data of the data set: The column defines the Value Provider property that returns the content of the cell based on the object of the row and the Renderer that defines how the Value Provider value is rendered.

For Grid Columns, do the following:

1. Define the following properties:
   - *Content*: content of the Column
     It can be of the following types:
     - **Property path**: path to the Property of the row object (applicable if the row object is a Record)
     - **Closure**: closure with the row object as its input parameter
     - **Custom**: custom expression
   - *Renderer*: the way the value returned by the Value Provider is rendered in the cell
     - **None**: renders the Value Provider value as is
     - **HTML**: renders the Value Provider value as HTML
     - **Number**: renders the Value Provider value in the defined format
       Define the format as a String following the `DecimalFormat Java formatting rules`, for example, `"0000.00000"`
     - **Date**: renders the Value Provider value in the defined format
       Define the format as a String following the `SimpleDateFormat Java formatting rules`, for example, `"EEE, d MMM yyyy HH:mm:ss"` will result in formatting like `Wed, 7 Sep 2016 14:33:00`
     - **Button**: renders the Value Provider value as a Button
       The action that should be performed on click is defined as a closure with the Value Provider object as its input parameter.
       ```
       { clickRowObject:String -> varString := "The user clicked: " + clickRowObject.toStri
       ```
     - **Link**: renders the Value Provider value as a Link
     - **Theme image**: image from your Vaadin ThemeResource
       To be able to use this option, the Value Provider must return a String with the path to the image. The path is relative to current Vaadin theme directory, for example, `myapp-war/VAADI↩N/themes/lsps-blue` so the String path could be `"favicon.png"`.

2. Define the action listener on the renderer if required:

   (a) Select the renderer in the Grid.

   (b) In the Properties view on the *Event Handling* tab, define the listener.

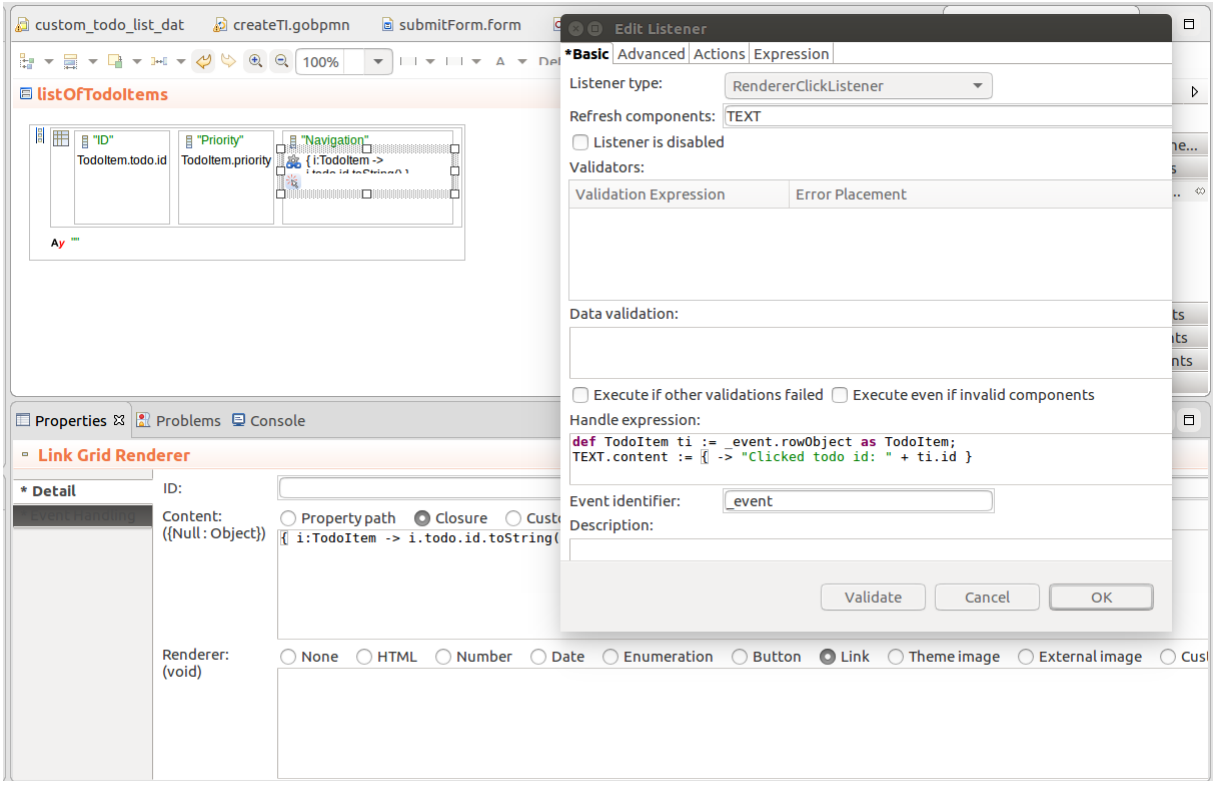
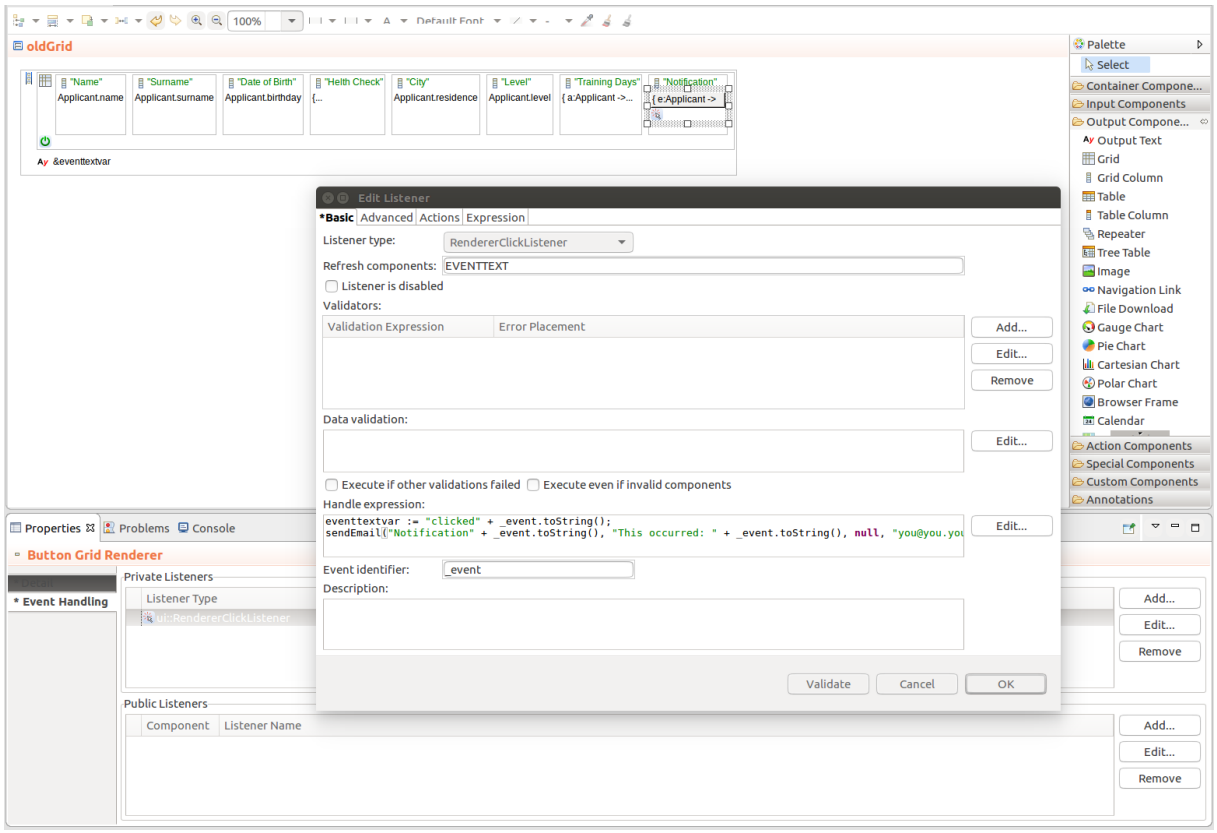



**Figure 4.20 RendererClickListener definition**

#### 4.3.3.2.1 Handling Action on Renderers in Grid Columns

In a Grid, you can handle clicks on Grid Columns with Button, Link, Theme image, and External image renderers:

1. Define the Grid Column with one of the Renderer types.
2. Select the Renderer in the Form to display its properties in the Properties view.
3. In the Properties view, click the Event Handling tab and click **Add**.
4. Define the listener type to RendererClickListener.

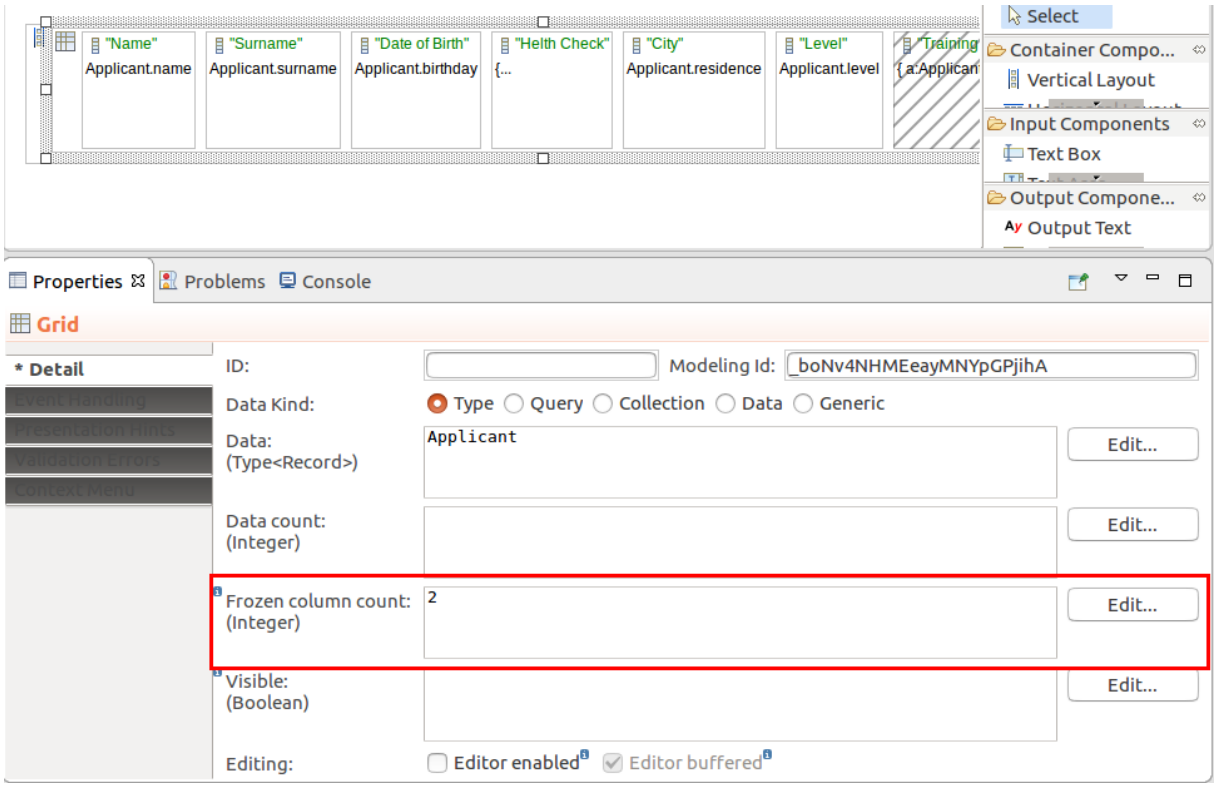



#### 4.3.3.2.2 Collapsing a Grid Column

To hide a Grid Column, that is to display it collapsed, set the `hidden` presentation hint to true: this setting will be applied when the form is initialized.

#### 4.3.3.2.3 Freezing Grid Columns for Horizontal Scroll

To freeze the first and any number subsequent columns of a Grid, set the *Freeze column count* property on the Grid to the number of columns to freeze. Frozen columns remain displayed when the user uses horizontal scroll,




#### 4.3.3.2.4 Enabling Editing of a Grid Column

You can make cells of a Grid Column editable in the following case:

- The **row object is a Record**.
- The Grid Column with the value has the **Value Provider set to Property Path**; otherwise the cell value will not be editable and that even if the Grid is editable.

  **Important:** If the underlying associated shared Record Property returns `null`, the cell will remain empty and uneditable.

To enable editing of shared Record Properties in a Grid, do the following:

1. Open the Properties view of the Grid:
   (a) Select the *Editor enabled* flag.
   (b) To save the changes only when the user clicks the *Save* button in the edited row, select the *Editor buffered* flag.
   If the *Editor buffered* is *not* selected, the changes are applied to the underlying Record instantly (on every change).
2. Enable editing on the Grid Column:
   (a) Open the properties of your Grid Column.
   (b) Select the *Editable* flag.
   (c) In the *Editor* field, define which editor should be used on the Value Provider object:
   - leave empty for Strings
   - *NumberEditor*: the user will be able only to provide a number (Decimal or Integer)
   - *DateEditor*: the user will be able to insert only a date with the option to use a date picker

- *EnumerationEditor*: the user will be able to select one of the Enumeration values from a drop-down box

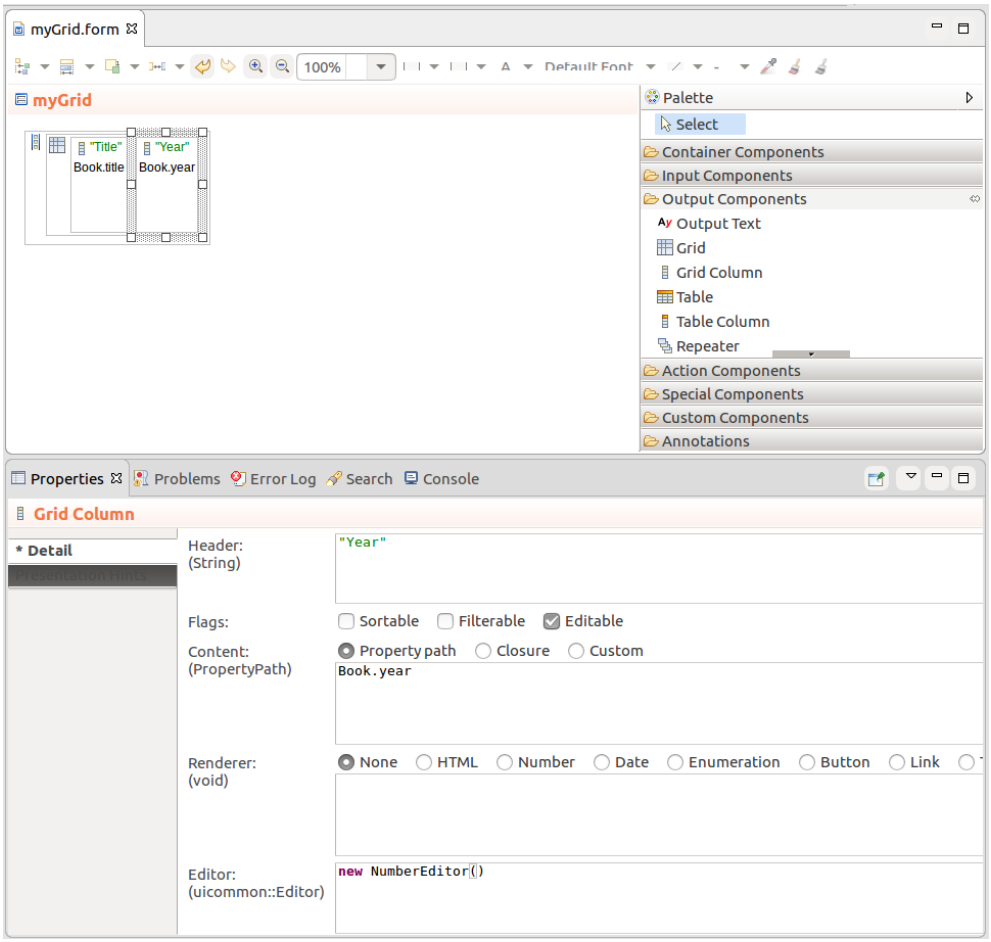
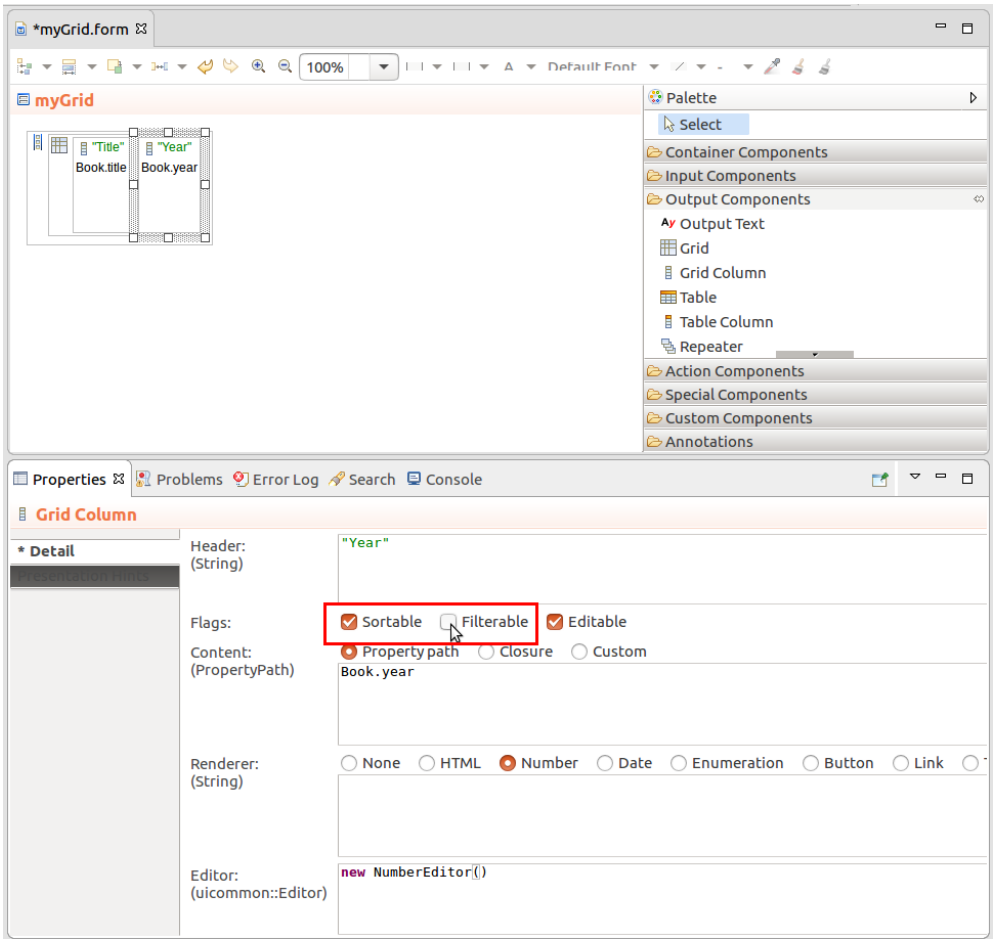



**Figure 4.21 Definition of a Grid Column with an editable number value**


**Figure 4.22 Editable Column with a validation message on incorrect value**

#### 4.3.3.2.5 Filtering and Sorting on a Grid Column

To enable filtering or sorting on a Column of your Grid, open the Properties view of the Column and check the *Sorting* or *Filterable* flag.




### 4.3.4 Repeater

The *Repeater* component is a Form output component that renders its child component multiple times.

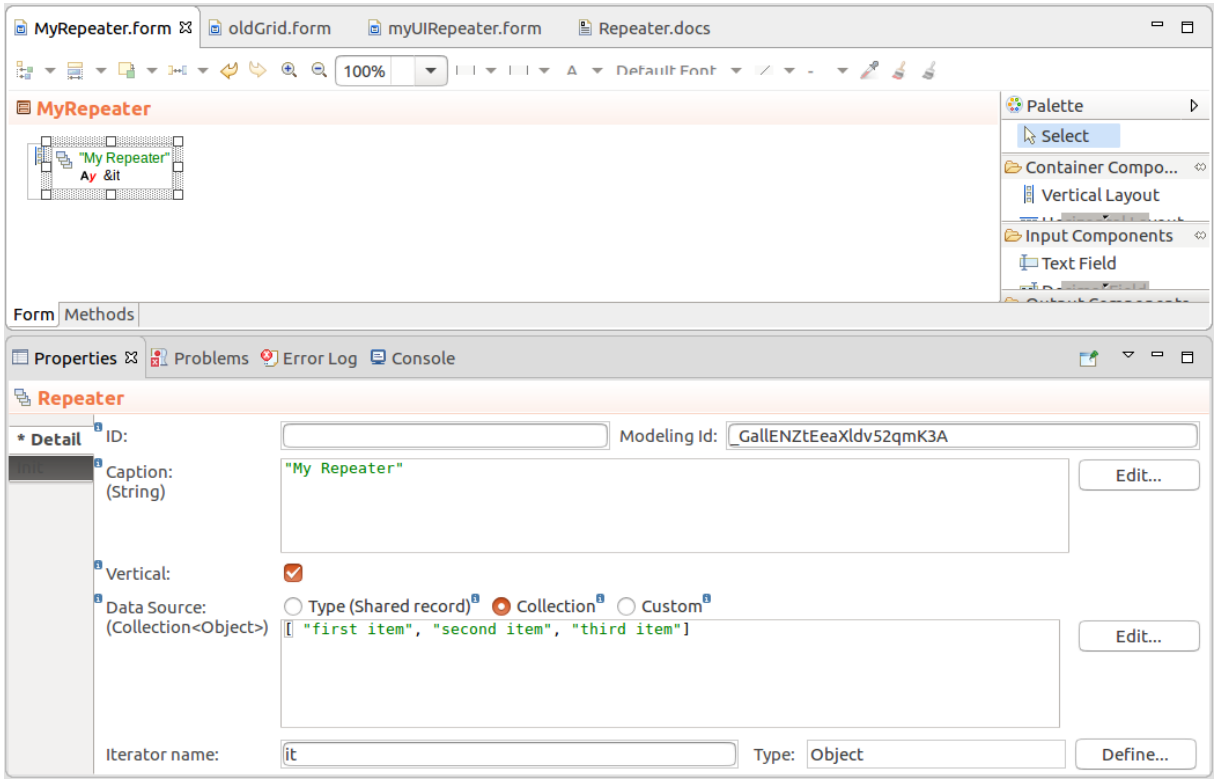
- **Data**: list of objects to iterate through
- **Data iterator**: reference to the iterated object (Since an iterator holds the value of the object for the current iteration, the referenced object must be of the same type as the Data list objects)

  **Important:** If you define a reference to a variable as iterator, make sure the variable is used solely as the repeater iterator.

- **Layout**: a RepeaterLayout that defines the layout applied on child elements
  - `RepeaterLayout.wrap`: repeated items are arranged horizontally; if their content is larger than the width wrapping, the lines overflow.
  - `RepeaterLayout.horizontal`: repeated items are arranged horizontally. Their size is ignored.
  - `RepeaterLayout.vertical`: repeated items are arranged vertically. Their size is ignored. If some of the child elements are not displayed in the repeater, set the child components width to `Wrap Content`.

    **Important:** The Layout property is not recalculated on refresh.

- **Index iterator**: reference to an Integer variable that holds the index of the currently iterated object




### 4.3.5 Image

The Image component is a Form output component that renders an image File object.

- **Content**: the image object to be displayed; you can use the `getResource()` function; for example, `getResource("myModule", "picture.jpg")`

  **Note:** The File object cannot be created directly over a file in the file system. Therefore, you need to import the image into your project (File > Import > General > File System) and create the File object over the imported image.

- **Text**: image caption
- **Help text**: tooltip text; you can define the Help text on the Help Text tab in the Properties view.

### 4.3.6 File Download

The *File Download* component renders as a hyperlink: when clicked, it produces the FileDownloadEvent and downloads a file.

It produces events of the following types:

- InitEvent when the component is initialized or displayed if previously hidden
- FileDownloadEvent when the user clicks the file-download button

File Download defines the following properties:

- **Content**: File object with the file
- **Text**: text of the download hyperlink
- **Help text**: tooltip text

### 4.3.7 Charts

**Important:** Before you design charts, make sure to purchase the Vaadin Charts license. Use of forms with charts in your Application User Interface does not require additional licenses.

#### 4.3.7.1 Pie Chart

The *Pie Chart* component renders as a circular chart that depicts data values as sections. When clicked, it produces a *ChartClickEvent* with data about what was clicked so the system can process the click as required.

It produces events of the following types:

- InitEvent when the component is initialized or displayed if previously hidden
- ChartClickEvent when the user clicks a chart

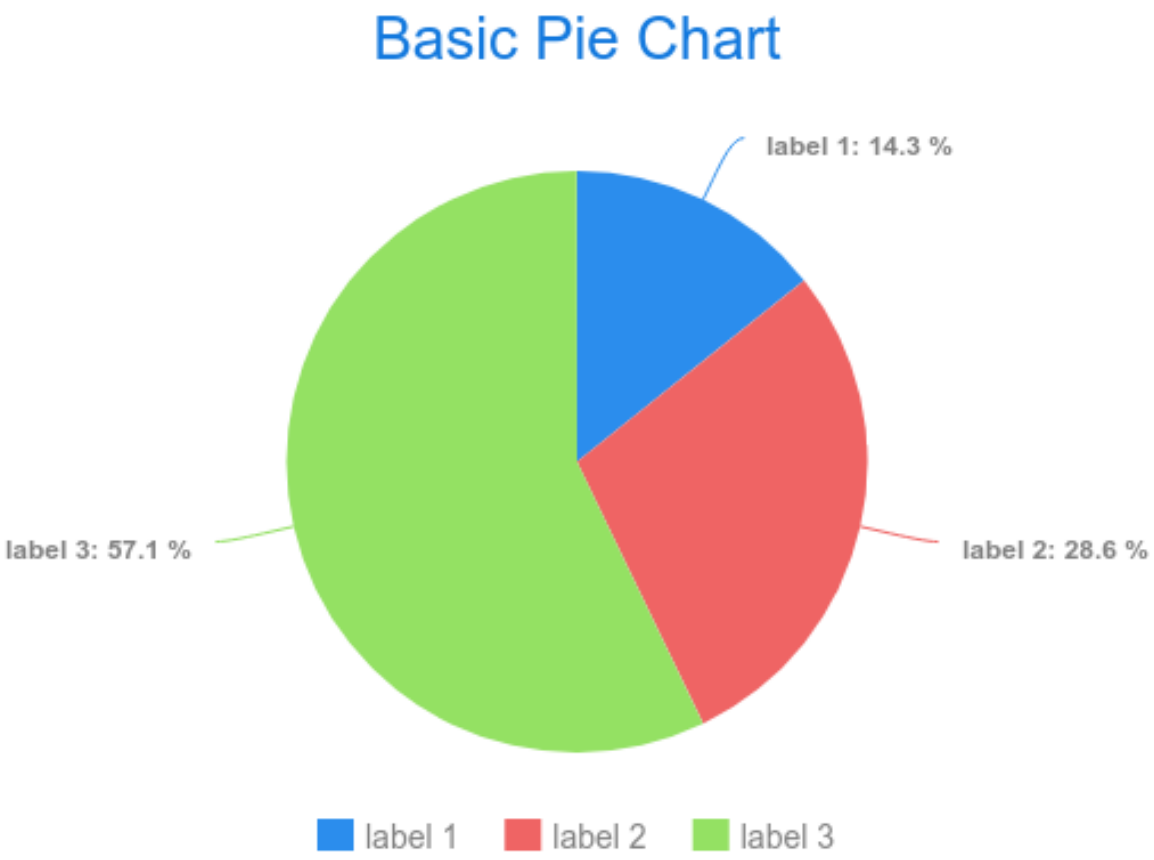



**Figure 4.23 Pie chart with legend**

The Pie Chart component has the following properties:

- **Title**: pie chart main title
- **Subtitle**: pie chart subtitle

- **Slices**: pie chart sections (a list of slices displayed in the pie chart)

  The slices define their label and value: the Slice value is defined as a decimal value and represents the mutual ratio of individual slices.

```
collect(
 getCurrentAssets(),
 {asset ->
   new ui::PieSlice(
     label -> asset.ISIN,
     value -> asset.currentAmount)
 }
)
```

- **Show legend**: chart legend setting (if true, the legend is visible)

#### 4.3.7.2 Gauge Chart

The *Gauge Chart* component renders as a chart with a circular Y-axis and a rotating pointer.

It produces events of the following types:

- InitEvent when the component is initialized or displayed if previously hidden
- ChartClickEvent when the user clicks a chart with information about the clicked data series

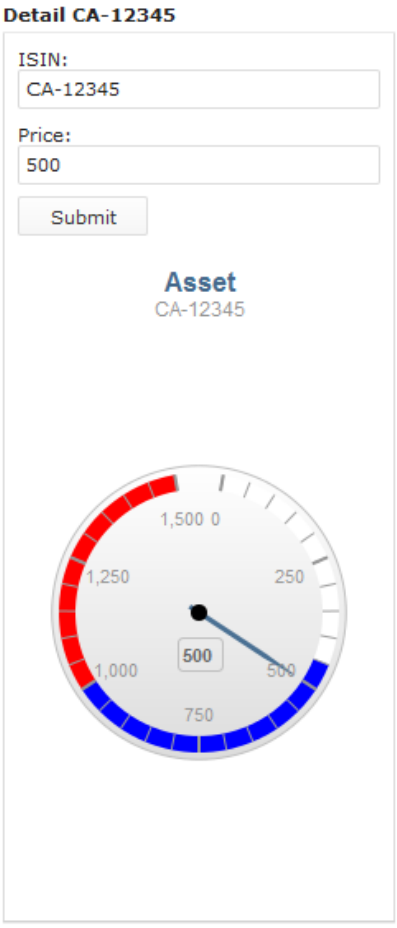



**Figure 4.24 Form with the Asset gauge chart displaying asset price**

**Gauge Chart Properties**

- **Title**: gauge chart main title
- **Subtitle**: gauge chart subtitle
- **Value**: value displayed by the pointer (needle)
- **Value name**: value name displayed in the pointer tooltip along with the Value
- **Axis**: gauge axis that defines the gauge chart scale, label, properties of the scale, bands, and chart position: The gauge axis constructor takes the following arguments:
  - min and max: minimal and maximal values on the gauge scale
  - label: label displayed directly on the gauge chart
  - opposite: position of the axis (if true, the axis is displayed on the edge of the gauge circle; if false, the axis is displayed inside the gauge circle)
  - bands: plot bands (shown white, blue, and red in the gauge chart figure above; CCS color definitions are supported)
  - startAngle and endAngle: start and end angle of the gauge axis
  - centerY: horizontal positioning of the gauge chart middle in percent (“50” places the chart directly under its title with no gap in between)

**Example Axis definition**

```
new ui::GaugeAxis(
 min -> 0,
 max -> 80,
 label -> "axis label",
 opposite -> null,
 bands -> [
   new PlotBand(from -> 0, to -> 20, color -> "#FFFFFF"),
   new PlotBand(from -> 20, to -> 40, color -> "blue"),
   new PlotBand(from -> 40, to -> 50,  color -> "red")],
   startAngle -> 300,
   endAngle -> 420,
   centerY -> 120)
```

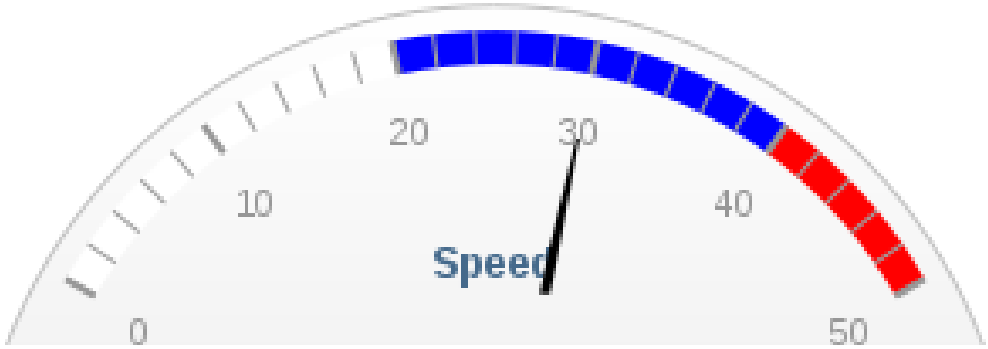



**Figure 4.25 Rendered gauge chart with the axis as defined above**

- **Show legend**: visibility of the chart legend

This property is not applicable for the gauge chart component.

#### 4.3.7.3 Cartesian and Polar Chart

The Cartesian Chart renders as a multi-dimensional chart with an arbitrary number of x and y axes. The rendering of the x axis depends on the given data series, while the y axis displays the value connected to the x value defined as a data point.

The Polar Chart is a variation of the Cartesian chart. It is rendered as a circle with the x axis on its circumference and its radius is the y axis. Just like the Cartesian chart, it is multi-dimensional chart, that is, it can have n arbitrary number of x and y axes. However, though this option is functional, the y axes are overlaid over each other in a single radius.

The charts produce events of the following types:

- InitEvent when the component is initialized or displayed if previously hidden
- ChartClickEvent when the user clicks into the chart

  The event holds data about the clicked data point: a listener can use this data, for example, to drill down into the chart or visualize details related to the clicked data point.

**Cartesian and Polar Chart Properties**

- **Title**: chart main title
- **Subtitle**: chart subtitle
- **Series**: a set of data series displayed in the chart (see data series)
- **X axes**: list of x chart axes

  You can define multiple x axes. The axes are arranged underneath or next to each other depending on their orientation.
- **Y axes**: list of y chart axes

  You can define multiple y axes. The axes are arranged underneath or next to each other depending on their orientation.

  Axes for Cartesian and Polar chart define the following properties:

  - min and max: minimal and maximal values on the axis
  - label: label displayed directly on the chart
  - opposite: position of the axis (if true, the axis is displayed as the opposite axis, that is x is displayed as y and vice versa
  - bands: plot bands (any CCS colors are supported)
- **Rotate axes**: Boolean value that defines whether the x and y axes are rotated (for example, if true, chart bars can be displayed horizontally)

  This option is not available for the Polar Chart component.
- **Show legend**: visibility of the chart legend (if true, the legend is visible)

Data series defines a set of data points that are displayed as values in the chart. It also defines general properties of the data series:

- `label`: the data series label in the legend and tooltip of the plotted data series
- `options`: plotting options
- `xAxisIndex`: index of the x axis the data series uses

- `yAxisIndex`: index of the y axis the data series uses

  Since chart axes are defined as lists, the xAxisIndex and yAxisIndex are defined as integers with the first defined axis being indexed `0`.

  **Important:** The ui::DataSeries record is abstract: Define the Data Series as one of its sub-types.

One chart can display multiple data series of different types. The type of a data series defines the way its x axis renders (note that a chart may contain multiple x or y axes):

- **ListDataSeries**: the x axis values are integers.

  Values are defined as a list of data points (**List**<**DataPoint**>) and are distributed evenly as depicted below.


**Figure 4.26 Cartesian chart with ListDataSeries (“Company A” and “Index”) and the series definition below**

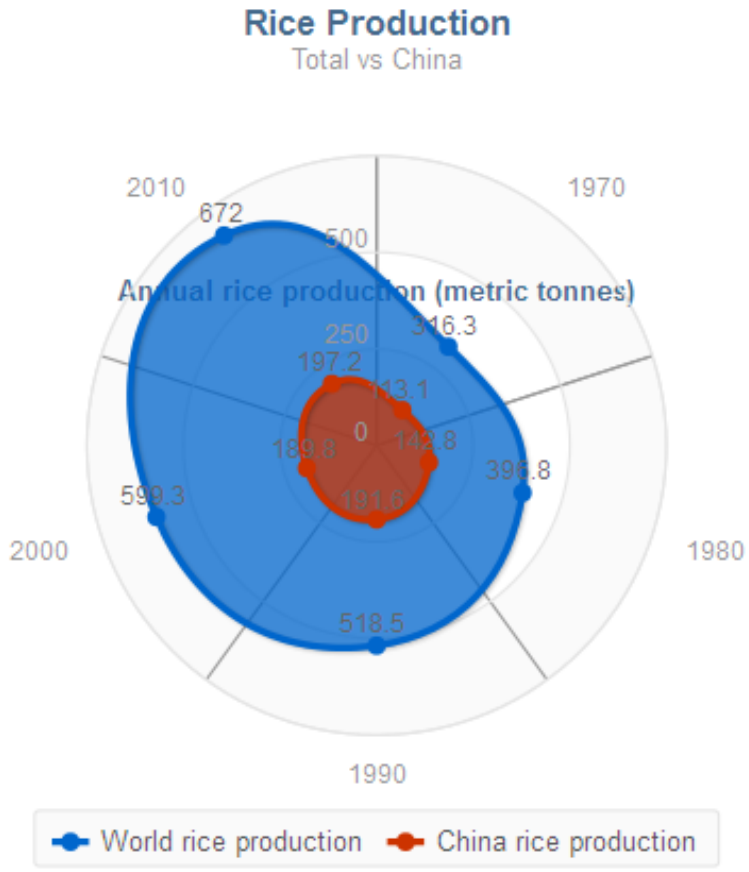



**Figure 4.27 Cartesian and Polar charts with CategoryDataSeries (“World rice production” and “China rice production”) and years as String values; the definition of the “World rice production” CategoryDataSeries below)**

```
 def List<DataPoint> companyAPriceList:=
[
   new DataPoint(value -> 40, value2 -> null, payload -> null),
   new DataPoint(value -> 60, value2 -> null, payload -> null),
   new DataPoint(value -> 35, value2 -> null, payload -> null),
   new DataPoint(value -> 10, value2 -> null, payload -> null),
   new DataPoint(value -> 30, value2 -> null, payload -> null),
   new DataPoint(value -> 30, value2 -> null, payload -> null)
];
def List<DataPoint> indexPriceList:=
[
   new DataPoint(value -> 30, value2 -> null, payload -> null),
   new DataPoint(value -> 90, value2 -> null, payload -> null),
   new DataPoint(value -> 45, value2 -> null, payload -> null),
   new DataPoint(value -> 15, value2 -> null, payload -> null),
   new DataPoint(value -> 34, value2 -> null, payload -> null),
   new DataPoint(value -> 10, value2 -> null, payload -> null)
];
[
   new ListDataSeries(
      label -> "Company A",
      options -> null,
      xAxisIndex -> null,
      yAxisIndex -> null,
      values -> companyAPriceList
      ),
```

```
    new ListDataSeries(
       label -> "Index",
       options -> null,
       xAxisIndex -> null,
       yAxisIndex -> null,
       values -> indexPriceList
       )
]
```

- **CategoryDataSeries**: the x axis values are arbitrary string values.

  Values are defined as a map of Strings and DataPoints (**Map<String, DataPoint>**): the String is used as the value on the x axis and the data point defines the values on the y axis.

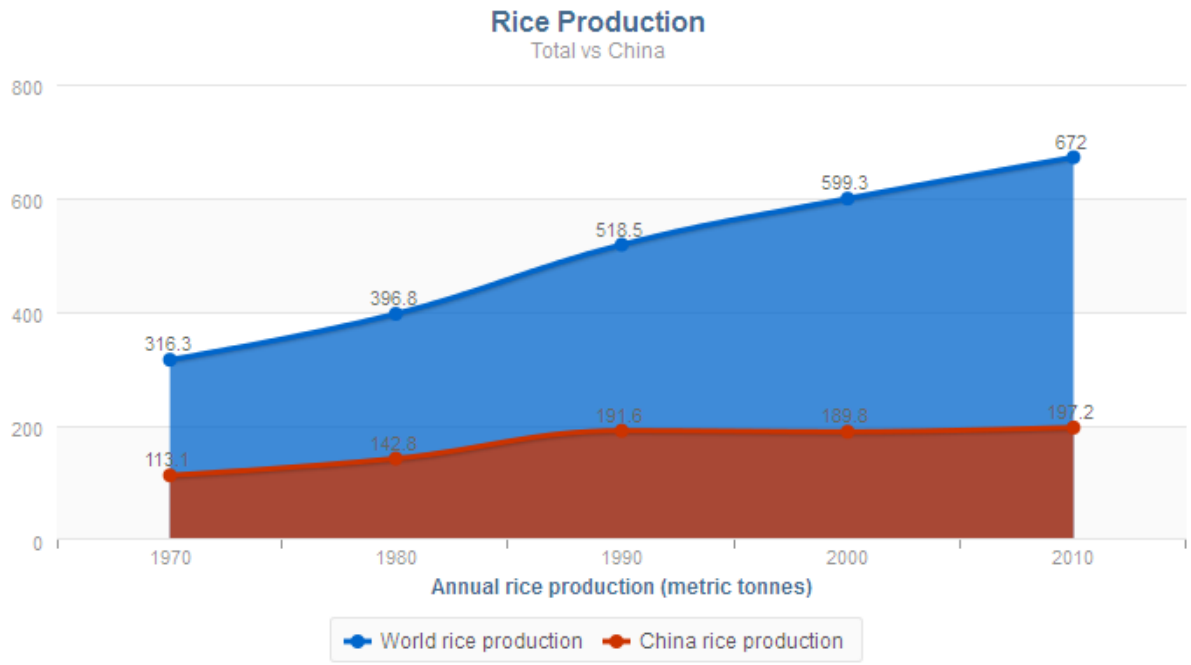



**Figure 4.28 Cartesian and Polar charts with CategoryDataSeries ("World rice production" and "China rice production") and years as String values; The definition of the "World rice production" CategoryDataSeries further below**

```
 def Map<String, DataPoint> worldData :=
  [
     "1970"-> new DataPoint(value -> 316.3),
     "1980"-> new DataPoint(value -> 396.8),
     "1990"-> new DataPoint(value -> 518.5),
     "2000"-> new DataPoint(value -> 599.3),
     "2010"-> new DataPoint(value -> 672)
   ];
def Map<String, DataPoint> chinaData :=
  [
     "1970"-> new DataPoint(value -> 113),
     "1980"-> new DataPoint(value -> 142),
     "1990"-> new DataPoint(value -> 191),
     "2000"-> new DataPoint(value -> 189),
     "2010"-> new DataPoint(value -> 197)
   ];
[
    new CategoryDataSeries(
```

```
        label -> "World rice production",
        options -> new PlotOptionsArea(
           color -> "#0066CC",
           stacked -> false,
           marker -> Marker.circle,
           lineStyle -> LineStyle.solid,
           lineWidth -> 3,
           spline -> true,
           range -> null,
           opacity -> null),
        xAxisIndex -> null,
        yAxisIndex -> null,
        values -> worldData
        ),
     new CategoryDataSeries(
        label -> "China rice production",
        options -> new PlotOptionsArea(
           color -> "red",
           stacked -> false,
           marker -> Marker.circle,
           lineStyle -> LineStyle.solid,
           lineWidth -> 3,
           spline -> true,
           range -> null,
           opacity -> null),
        xAxisIndex -> null,
        yAxisIndex -> null,
        values -> chinaData
        )
]
```

- **TimedDataSeries**: the x axis values are points in time.

  Values are defined as a map of Dates and DataPoints(**Map**<**Date, DataPoint**>): the date is used as the value on the x axis and the data point defines the values on the y axis.

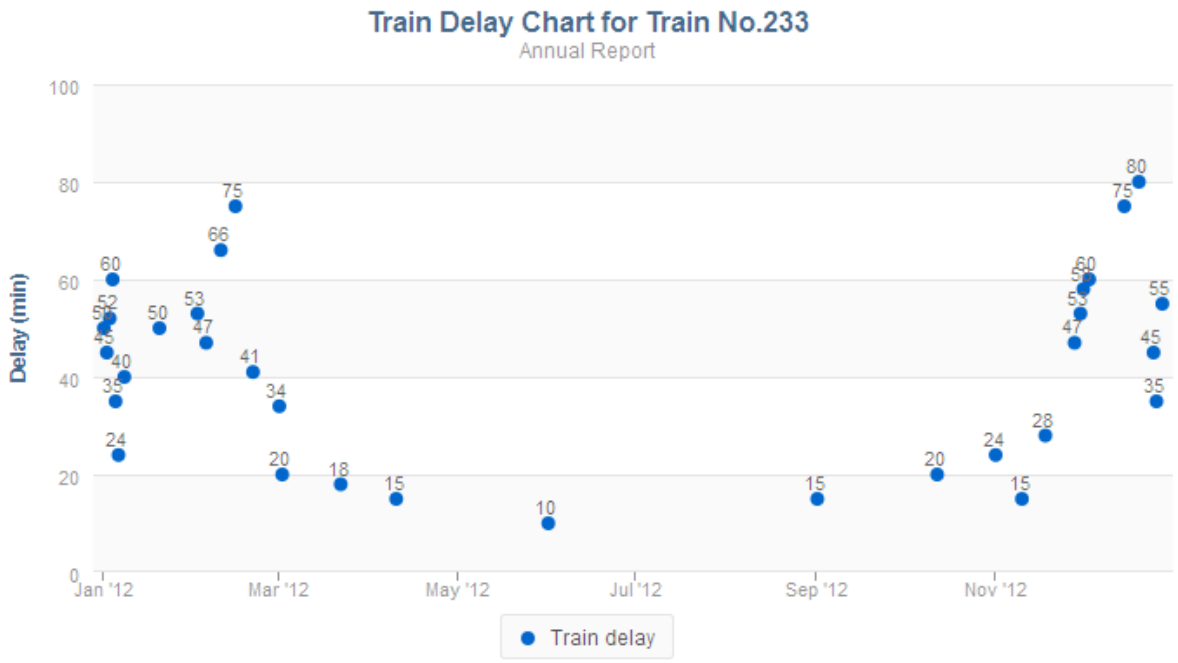



**Figure 4.29 Cartesian chart with TimedDataSeries (“Train delay”); the definition of the “Train delay” TimedDataSeries below**

```
def Map<Date, DataPoint> trainDelay :=
[
    date(2012, 1,1)->new DataPoint(value -> 50),
...
];

[
    new TimedDataSeries(
        label -> "Train delay",
        options -> new PlotOptionsScatter(color -> "#0066CC",
        showLabels -> true,
        stacked -> false,
        marker -> Marker.circle),
    xAxisIndex -> null,
    yAxisIndex -> null,
    values -> trainDelay)
]
```

**Figure 4.30 Cartesian chart with TimedDataSeries (“Train delay”); the definition of the “Train delay” TimedDataSeries**

- **DecimalDataSeries**: the x axis values are decimal numbers.

Values are defined as a map of Decimals and DataPoints (**Map<Decimal, DataPoint>**): the decimal is used as the value on the x axis and the data point defines the values on the y axis.

#### 4.3.7.4 Plotting Options

Plotting options define how a set of data renders. They are defined in the `options` property of every data series so that every data series in a chart can be plotted differently. If undefined, the default plotting properties for the given data series are applied.

The following plotting options are available:

- `PlotOptionsArea`: the data series is rendered as an area.
- `PlotOptionsBar`: the data series is rendered as a set of bars.
- `PlotOptionsBubble`: the data series is rendered as a bubble.

  **Note:** This plotting option is currently not supported.
- `PlotOptionsScatter`: the data series is rendered as a set of dots (scatter).
- `PlotOptionsLine`: the data series is rendered as a line.

```
[
    new TimedDataSeries(
        label -> "Train delay",
        options -> new PlotOptionsLine(
            color -> "#0066CC",
            showLabels -> true,
            stacked -> false,
            marker -> Marker.diamond,
            lineStyle -> LineStyle.solid,
            lineWidth -> 4,
            spline -> true),
        xAxisIndex -> null,
        yAxisIndex -> null,
        values -> trainDelay)
]
```

**Figure 4.31 TimedDataSeries with a PlotOptionsLine definition**

### 4.3.8 Browser Frame

The Browser Frame renders as a view to a URL. To adjust the size of the frame, use presentation hints (refer to the Standard Library).

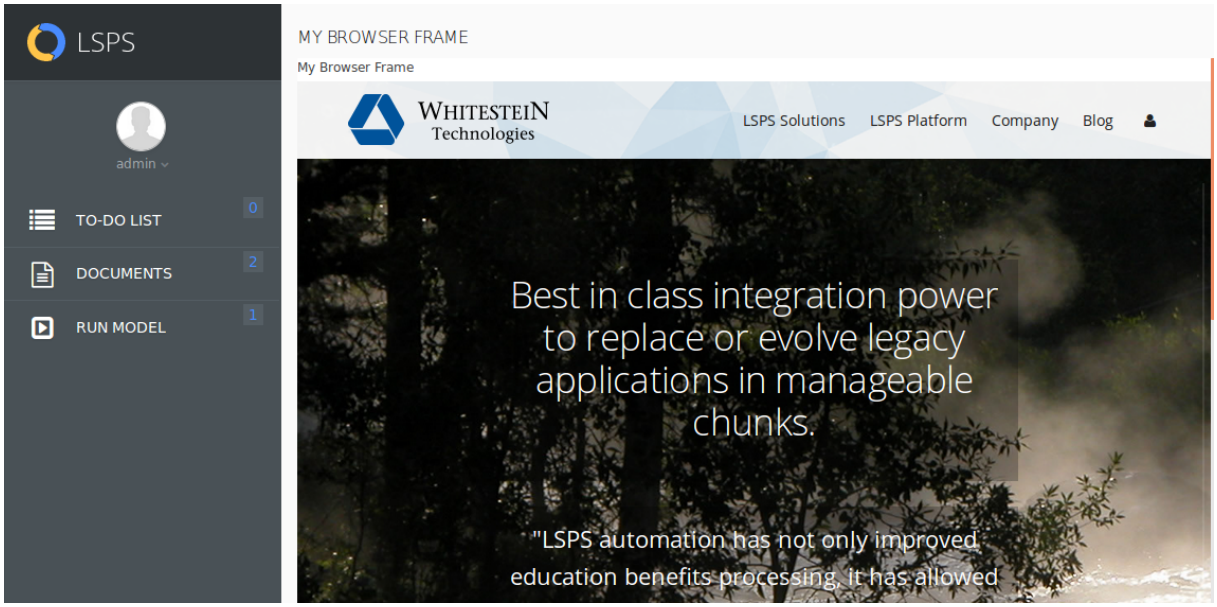



**Figure 4.32 Browser Frame to external page in a tab component**

Browser Frame defines the URL property with the URL displayed in the frame.

### 4.3.9 Calendar

The Calendar ( ) component is rendered as a calendar with calendar entries. The calendar entries are clickable and can be dragged-and-dropped. You can also select the calendar mode in the calendar. These actions fire the respective events.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *CalendarEditEvent* when the user clicks a calendar entry
- *CalendarRescheduleEvent* when the user drags a calendar entry
- *CalendarCreateEvent* when the user selects a time period by clicking and dragging

The calendar component is displayed in month mode by default. To display a day schedule, click the day date in the calendar cell. Note that if a calendar entry needs to be scheduled in the day mode and keep track of exact hours, the allDay property of the entry must be set to `false`.

- **Data**: closure that returns a set of business data for the calendar (the data contains information about individual calendar entries)

```
{
   x, y -> def Set<Object> result:={};
   foreach Note n in notes do
       if n.notetype == NoteType.MEETING
       then
         result:=add(result, n)
        end;
   end;
   result
}
```

- **To item**: closure that transforms the data from the set defined in the **Data** property to CalendarItem

```
{ mynote:Note -> new ui::CalendarItem(
                 caption -> mynote.description,
                 description -> "Imported MEETING note",
                 from -> mynote.time.from,
                 to -> mynote.time.to,
                 allDay -> false,
                 style -> null)}
```

- **Initial Date**: date that is selected in the calendar when first opened (By default, the current date is selected.)
- **Mode**: calendar display mode

  The property determines the way a calendar is displayed initially and after refresh. The possible values are `daily`, `weekly`, `monthly`.

- **Read only**: calendar renders as read-only and cannot be edited

  If read-only, CalendarRescheduleEvent, CalendarCreateEvent are not fired. CalendarEditEvent is fired to allow the form to display an event details.

### 4.3.10 Map Display

The Map Display component renders as an OpenStreetMap with the defined center location, zoom, and optionally also markers. You can drag the map to change the visualized area and zoom it in and out. The map markers can be draggable.

It produces events of the following types:

- InitEvent when the component is initialized or displayed if previously hidden
- MapClickedEvent when the user clicks the map
- MarkerClickEvent when the user clicks a map marker
- MarkerDraggedEvent when the user drags a marker

#### Map Properties

- **Center**: coordinates of the center of the rendered map
- **Zoom**: default zoom on initialization and refresh defined as an integer with value 0-18 (0 being the lowest zoom with the entire Earth displayed)
- **Markers**: set of business data that are used as map markers

```
 {
  [new Meeting(
          title -> "Whitestein Meeting",
          location -> whitesteinHeadquarters,
          can_reschedule -> false,
          attendees -> ["Vladimir", "Estragon"]
          ]
}
```

- **To marker**: expression that converts the business data from Markers to MapMarker objects

```
 { x:Meeting -> new ui::MapMarker
  (
    title -> x.title,
    location -> x.location,
    popup -> x.title + "<br/>Atendees: " + x.attendees,
    draggable -> x.can_reschedule
  )
}
```

## 4.4 Action Components

Action components produce an action event when clicked.

### 4.4.1 Button

The *Button* component renders as a button. On click the button produces an action event. The component can have an ActionListener that defines how the event should be handled.

It produces the events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ActionEvent* when the user clicks the component


**Figure 4.33 Tab with the Submit note Action Button**

The Button has the following properties:

- **Text**: text on the button
- **Disabled**: availability of the button
  If true, the button is grayed out and cannot be clicked.
- **Help text**: tooltip text

  **Note:** You can define the Help text on the Help Text tab in the Properties view.

### 4.4.2 Action Link

The Action Link component renders as a clickable link. On clicking the link produces an action event. The component can have an ActionListener that defines how the event should be handled. The handling typically involves navigation action.

It produces the events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- *ActionEvent* when the user clicks the component


**Figure 4.34 Tab with the Link to note Action Link**

The Action Link has the following properties:

- **Text**: link text
- **Disabled**: whether the link is disabled (rendered as grayed out and not clickable when true)
- **Help text**: tooltip text

  **Note:** You can define the Help text on the Help Text tab in the Properties view.

### 4.4.3 Navigation Link

The Navigation Link component renders as a hyperlink; when clicked, it redirects the user to navigation target.

- **Content**: the Navigation object to be used for navigation

```
new UrlNavigation(url -> "http://www.whitestein.com")
```

- **Text**: text displayed in the navigation link
- **Disabled**: whether the link is disabled (rendered as grayed out and not clickable)
- **Help text**: tooltip text

## 4.5 Special Form Components

### 4.5.1 Message Form Component

The *Message* component displays validation messages of failed validators in the chosen Form location. It is useful if you do not want to display these in the components with failed validation.

### 4.5.2 Expression Form Component

The *Expression* component defines an expression that returns a component. The expression is evaluated when the screen context is created and cannot be recalculated later. The component is intended for quick ad-hoc expression returning: It is recommended to preferable use the Reusable Form component.

### 4.5.3 Reusable Form

The *Reusable Form* component allows you to use an already existing form in your current form: it references a form, which is on runtime inserted into the tree of components. The form is called and resolved when the screen context is created.

Note that if you want to work with events of such injected reused form in other form components or process events from other nodes inside the reused form, you will need to explicitly allow such event distribution.

### 4.5.4 Conditional Form Component

The *Conditional Form* component is a form component that defines the visibility of its child components: if the Visibility property evaluates to `false` and the parent component is visualized or refreshed then the child components are not displayed; the children do not exist at all. Therefore it is not possible to operate over the child components unless the Conditional component defines them as visible.

### 4.5.5 View Model Component

With the *View Model* component, you can create context on execution levels that overlay the current execution level. This allows you to isolate a data set and apply or throw the changed data set without compromising the underlying data for example, when you want to edit data in a popup and apply them only when the user confirms the data.

The View Model component creates the data of its component subtree on a level called an **evaluation level**. Each evaluation level **holds the differences against the data in the context on the level they overlay**, first the screen level and if the View Model is inside another View Model then against another evaluation level.

When the user changes the data in components of a view model, the data on the level below remain unchanged until explicitly requested to be merged.

You can either apply the changes to the context in the underlying level or discard them. The level to which the changes are applied is defined by the merge type of the view model:

- `MergeType.oneLevel`: the view model context is merged into the immediate underlying level
- `MergeType.screenLevel`: the view model context is merged into the screen level

Hence, View Models are used, for example, when creating pop-ups: the user enters data into the popup without influencing the data in the rest of the form.

Note that the components inside the View Model still exist on the screen level. The exception are dynamic popups, which exist on the level they are created on.

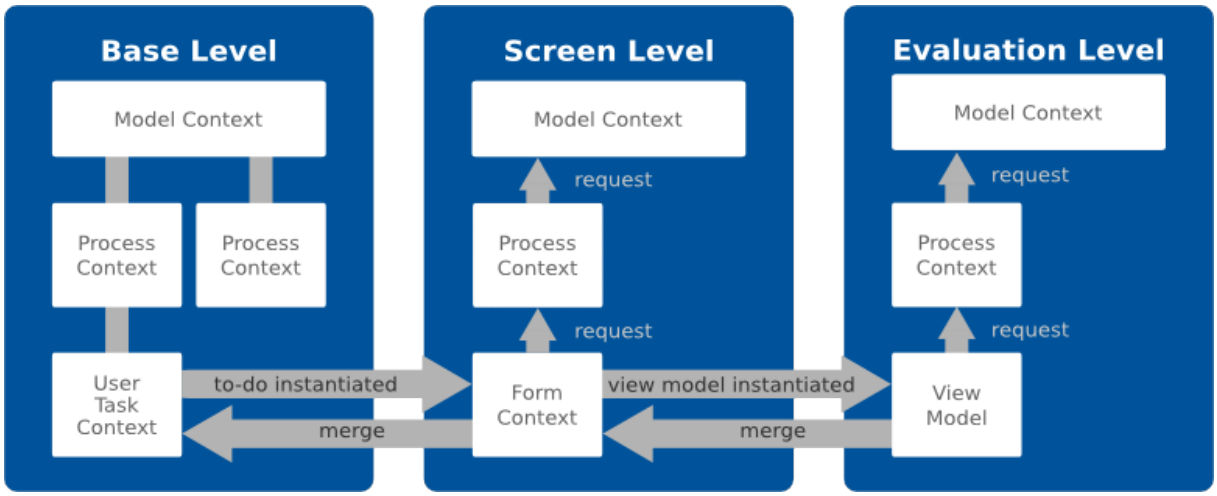



**Figure 4.35 Execution levels and contexts in Model instances**

For a view model use case, refer to Pop-up with Apply and Cancel Buttons.

#### 4.5.5.1 Isolating Transient Data

To isolate transient component data use the View Model component: The component provides a "commit mechanism" by creating a context on another evaluation level. A context on an evaluation level overlays the original context so the components inside the View Model work with data in their own space. You can use a view model, for example, to implement a cancel action when editing data: the user will edit the data in the View Model and the view model will be discarded or merged on a button click.

Generally you will proceed as follows:

1. Insert the *View Model* component into your Form. Make sure to define its name and merge type.
2. Into the View Model, insert Input Form components that will allow the user to modify the data.
3. In the View Model, create components with listeners that will merge or clear the data from your View Model: On the listener's Advanced tab:
   - To apply the data changes from a View Model, enter the View Model name to the Merge view model components property.
   - To discard the data change from a View Model, enter the View Model name to the Clear view model components.
   - You might want to define the View model init expression on the listeners: the expression is executed right after the merge or clear of view models.

Alternatively, you can call the *merge()* and *clear()* functions from the handle expression.

See `Pop-up with Apply and Cancel Buttons` for example usage.

For example, let us assume the form below.

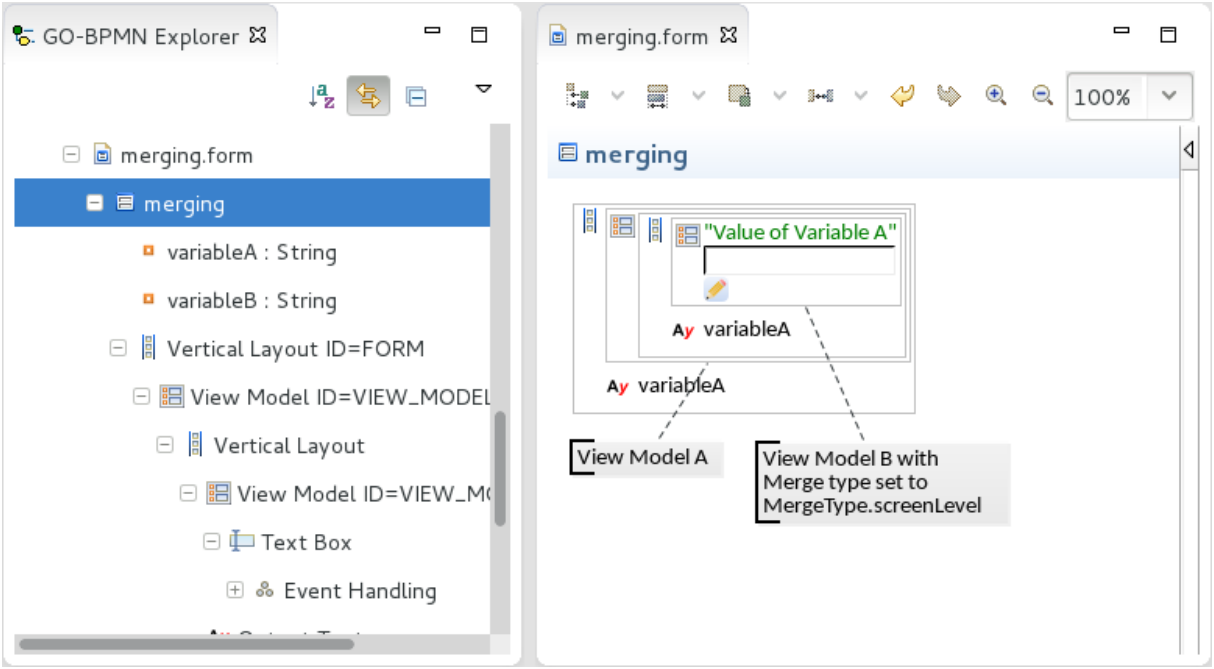



**Figure 4.36 Form with multiple view models**

Note the following:

- View Model B has the Merge type property set to `MergeType.screenLevel`
- The "Value of Variable A" TextBox is bound to variable A and has the Immediate property set to `true`.
- The ValueChangeListener refreshes the entire form and merges the View Model B (set in the listener properties).

On runtime, if you change the variable A value in the text box, the value will be merged to the screen context via the underlying evaluation level: hence also the context in of View Model A will have the change reflected and the displayed value will be updated.

For a view model use case, refer to `Pop-up with Apply and Cancel Buttons`.

### 4.5.6 Geolocator Component

The *Geolocator* component serves to acquire user's location. The location is detected on initialization and every component refresh. Note the component is not rendered in a form and is intended to provide input data for other components such as the map component.

The location is acquired either from the Wi-Fi or BTS location with accuracy from 300 to 3.000 meters, or from GPS with accuracy from 1 to 10 meters.

**Note:** When a form with a Geolocator component is rendered, the browser asks the user whether he wants to enable the locating.

It produces events of the following types:

- *InitEvent* when the component is initialized or displayed if previously hidden
- **Geolocation event** when the Geolocator component acquires user's location

  Note that as of the time of writing, Firefox version 24 and later do not support geolocation.

The Geolocator component has the following properties:

- **Detect**: enables or disables the location detection (If `false` the location is not detected on component refresh; this feature allows the user to disable the location detection, for example, via an action button.)
- **Position Options**: options of the location detection (for details, refer to *PositionOption* in `ui↩ ::components.datatypes`)

#### 4.5.6.1 Acquiring Location

To acquire location of the user in your form, do the following:

1. Insert a *Geolocator* component into your form.
2. On the Geolocator component, create the GeolocationListener.
3. To work with the received location, in the Handle expression, handle the event's position property.

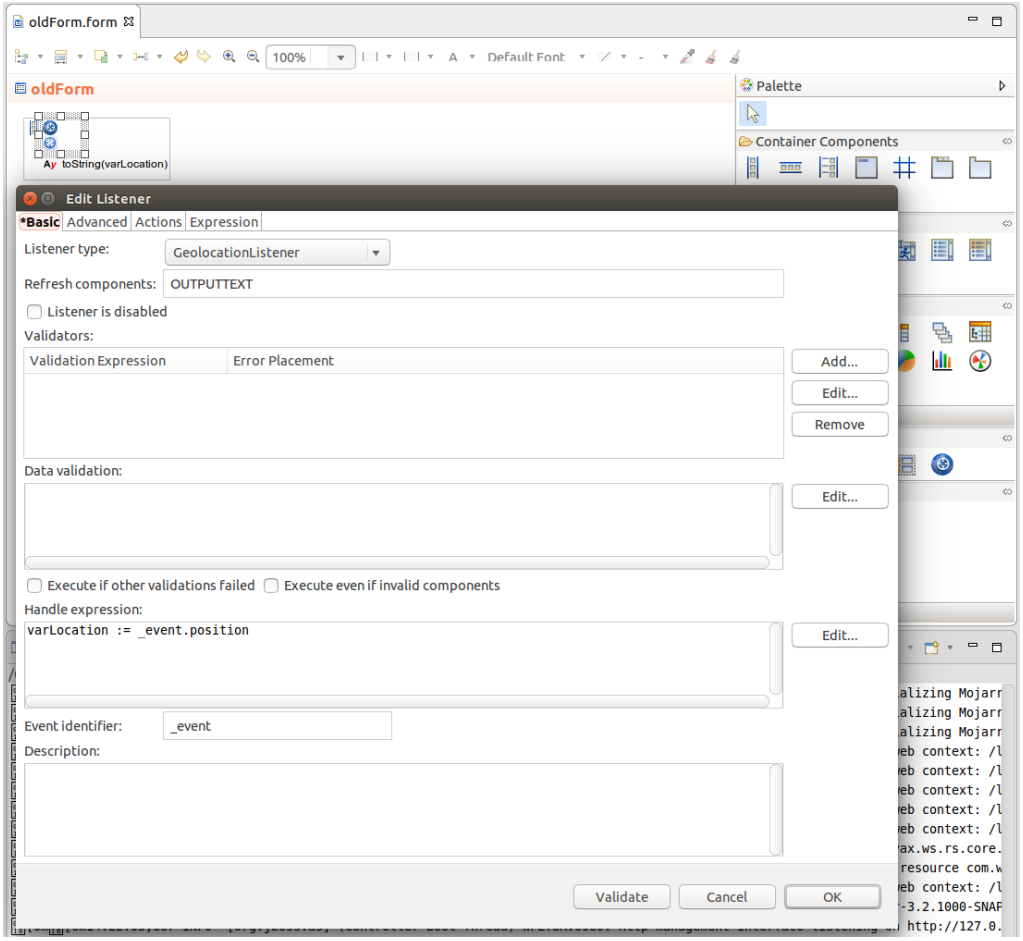



**Figure 4.37 Handling of Geolocation event on the Geolocator component**

## 4.6 Text Annotations and Associations

Annotation components serve to document the form: they display information about the and have no semantic value.

The Text annotation holds description or comments and can be connected to the related form component with a directed or non-directed association, which is a line, again, with no semantic value. You can use also a directed association, which is rendered as an arrow.

## 4.7 Deprecated Components

The components described in this section are deprecated and will be removed in the next release.

### 4.7.1 Tree

The Tree (  ) component renders as a tree of nodes. The nodes are expandable unless they are leaf nodes, that is, they do not have any children. Whenever a node is expanded its child nodes are lazy-loaded. To acquire the child nodes, the Tree component calls the closure defined in the Children property. The closure returns a List<TreeItem> objects.

The TreeItem objects define the following:

- label (String): arbitrary text displayed as the node label
- expanded (Boolean): whether the node is expanded by default
- data (Object): business object the tree item operates over

The Tree component serves as a picker of exactly one option (node). Once the user selects a node, the selection is stored in the Binding slot.

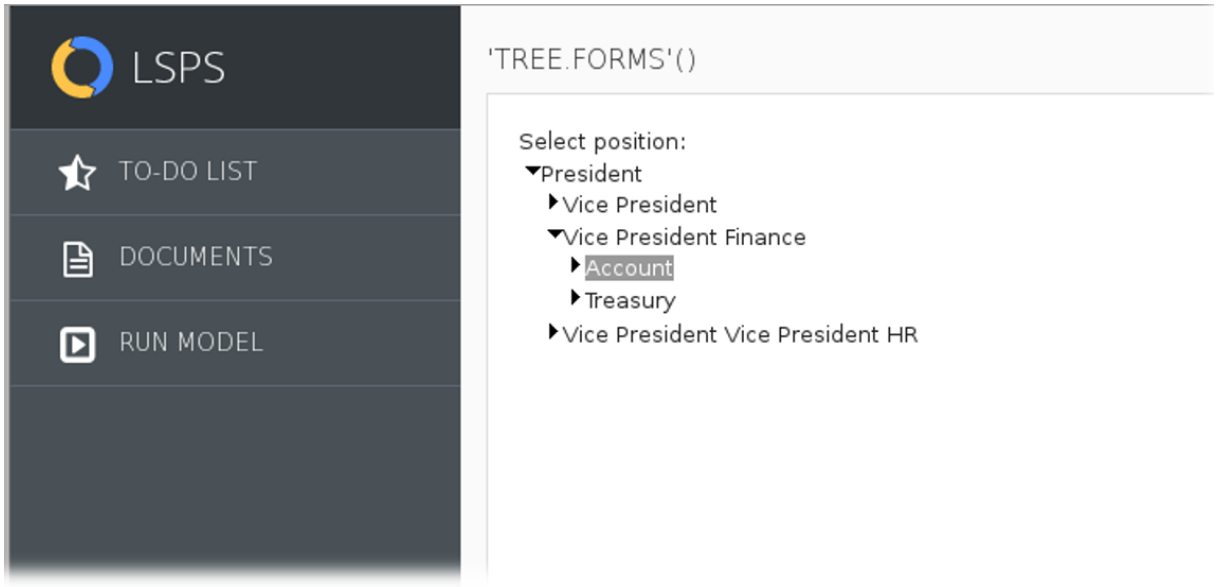



**Figure 4.38 Tree with multiple nodes expanded**

The Tree component has the following properties:

- **Label**: visible label
- **Required**: compulsoriness of the tree component

  If true, a mark indicating that the value is required is rendered in the component. > **Important:** The property **does not provide any validation**: the user > will be able to submit a null value even if it is set to `true`. To prevent > the submit, define a form validation on a listener.
- **Binding**: reference to a slot that holds the selected tree option value (for example, a form variable or global variable)
- **Children**: closure that returns a list of tree items which are displayed when a node is expanded

```
{ parent:Position, depth:Integer ->
   compact(
     collect(
       positions,
       { p:Position ->
         p.parent := parent ? new TreeItem (
           data ->p,
           expanded ->false,
           label -> p.name) : null
       }
     )
   )
}
```

- **Read-only**: editability

  If true, the tree renders as grayed out and no option can be selected. The selection uses the value of the binding slot.
- **Immediate**: setting of immediate mode

  If true, the Immediate mode is active: any value changes are processed immediately.
- **Help text**: tooltip text

  **Note:** You can define the Help text on the Help Text tab in the Properties view.

### 4.7.2 Tree Table

The Tree Table component renders as a table with rows organized in a tree structure with rows as tree nodes. Hence, child rows can be collapsed.

The component works similarly to the Table component. However while a table operates directly over business data types, the tree table component wraps the business data in the TreeTableItem data type objects. The object holds the along with business data also additional information. Based on the additional information, the tree table either expands the node that represents the business data or keeps it collapsed.

The tree table component iterates through a List<TreeTableItem> object. The TreeTableItem objects content can be then further processed (for example, displayed or provided for editing) in the table's columns.

The TreeTableItem objects define the following:

- expanded (Boolean): whether the node is expanded by default
- data (Object): business object the tree table item operates over (business data you want to work with in the tree table)


**Figure 4.39 Tree Table with multiple nodes expanded**

The Tree Table component has the following properties:

- **ID**: component ID unique in the form (Only capital letters, digits, and underscores are allowed.)
- **Children**: closure that returns a list of tree table items that are displayed when a node is expanded
- **Iterator**: reference to an object of the business data type

  The object is used as iterator for the table tree items.

  > **Important:**The iterator is not of the TreeTableItem type but of the type of your business data : the List<TreeTableItem> holds TreeTableItem objects, while the TreeTableItems objects hold the business objects in **data** and serve to provide the additional **expanded** property.

- **Help text**: tooltip text

  > **Note:** You can define the Help text on the Help Text tab in the Properties view.

# Chapter 5

# Enabling Error Reporting on Components

By default, runtime error reports do not contain information on which form component caused the error. Run your server with the `-Dcom.whitestein.lsps.vaadin.ui.debug=true` system property to include the modeling ID of your components in the reports (if you are running your server from PDS, go to **Server Connection** -> **Server Connection Settings**, select the Connection and click **Edit**).

You can then use the search to find the form component with the modeling ID: go to **Search** > **Find Form Component**.