WHITESTEIN Technologies

Living Systems® Process Suite

Todo and Document Forms

Living Systems Process Suite Documentation

3.3
Mon Nov 1 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

A form defines the content that is displayed in the front-end application, the *Application User Interface*, when the user opens either a `to-do` or a `document`

It serves to obtain and process information, which is used by the application or model instances: The user opens a to-do or document, fills out the form, submits it, and the execution of the underlying entity, that is, user task or document model instance, continues.

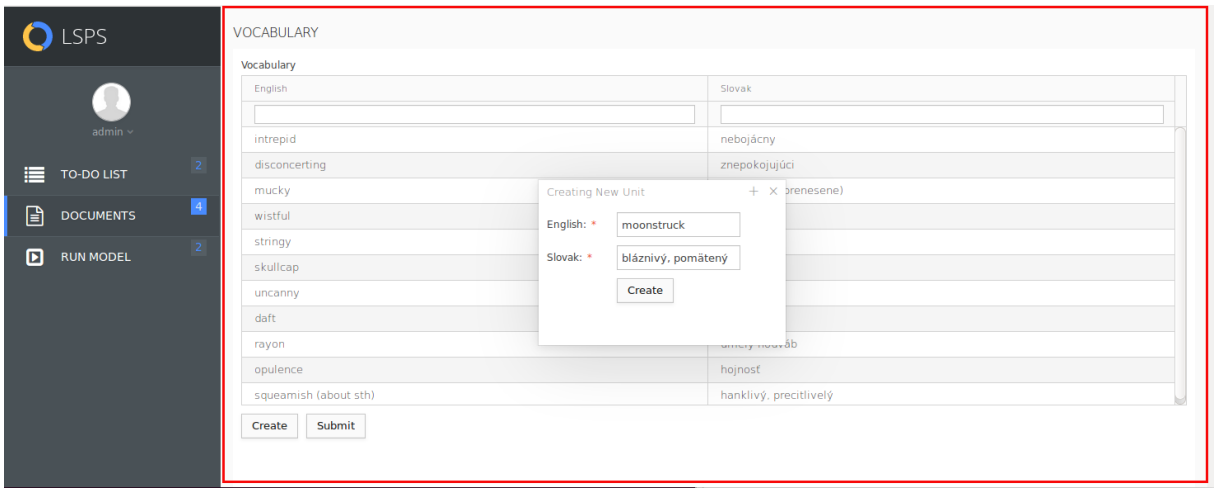


Figure 1.1 Form with a table in a document

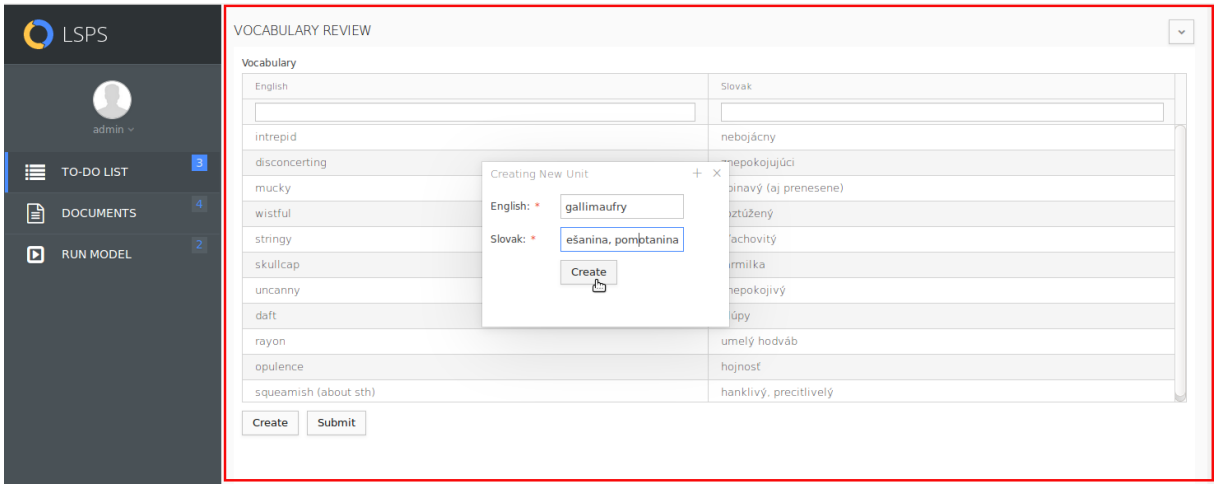


Figure 1.2 Rendered to-do with the form

Important: Chart components of forms are based on `Vaadin Charts` developed by a third party. Make sure to obtain the Vaadin Chart licenses before you use them. The users of the your Application User Interface and Management Console, do not require any additional licenses.

Note: This document cover the forms implementation provided by the *forms* module of the Standard Library. The Standard Library provides a `previous implementations` of forms with life-cycle driven processing of events. This implementation is now deprecated.

Chapter 2

Quickstart

We will create a Document, and a Process that will display your form. From the form, we will create a new shared Record instance.

2.1 Prerequisites

Before you can start working with form definitions, you will need to:

1. Run PDS.
2. Open the Modeling perspective.
3. Create a GO-BPMN project with a GO-BPMN module.
4. Create the *data* module with a Data Type definition: insert a shared Record with at least one Field into the definition.

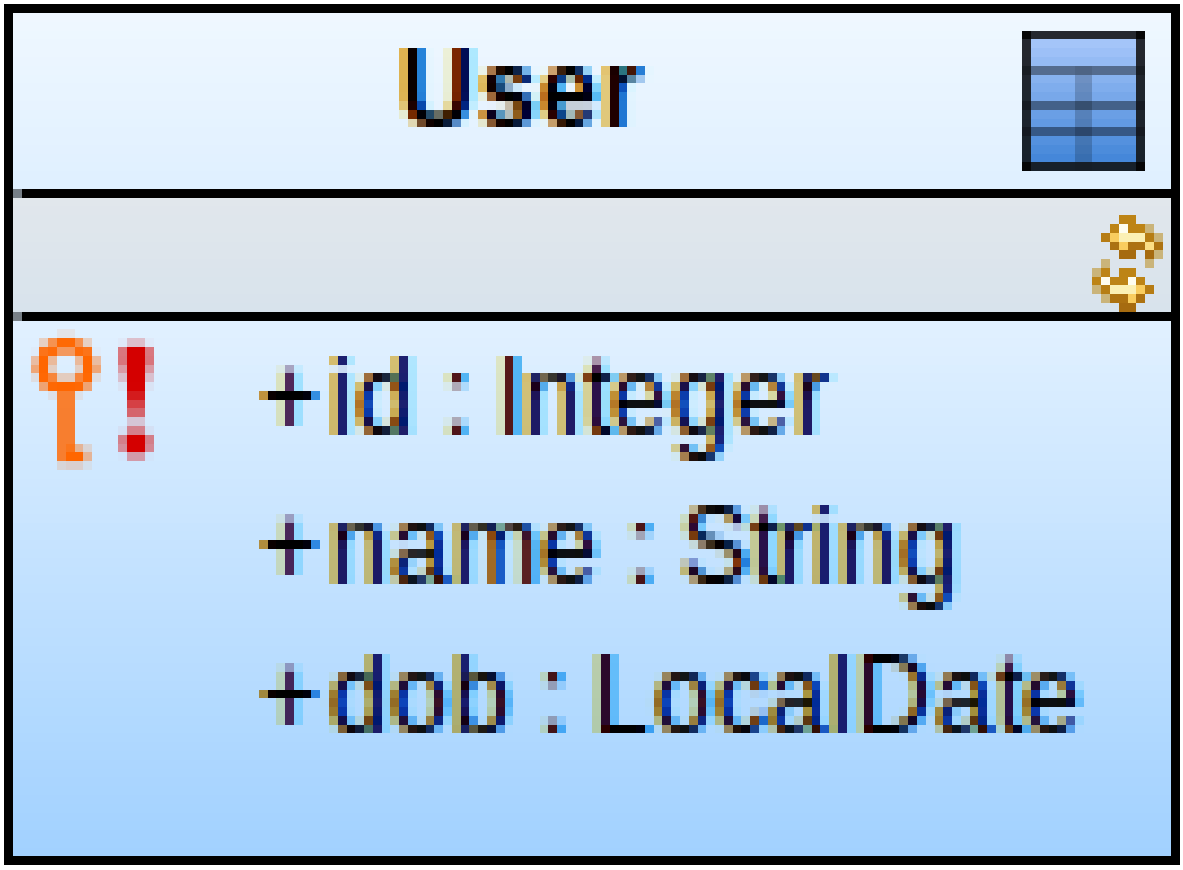


Figure 2.1 Shared Record

5. Create the *quickstart* module and import the *data* module into it.
6. Remove the *ui* module from module imports.

2.2 Form

> **Important:** Before you create your form, make sure you have removed the *ui* module import from your module. Having both, the *ui* module and the *forms* module, imported in a module could cause exceptions on forms in runtime.

To create the form, do the following:

1. Import the *forms* Module into your Module:
   (a) In the GO-BOMN Explorer, expand the Module and double-click its **Module Imports** node.
   (b) On the Imports tab of the *Properties* dialog, click **Add**.
   (c) In the *Add New Import* dialog, double-click the **Standard Library** > **forms** Module.
2. Create a Form Definition:
   (a) Right-click your Module > **New** > **Form Definition**

(b) Enter the form name and select *Use FormComponent-based UI*

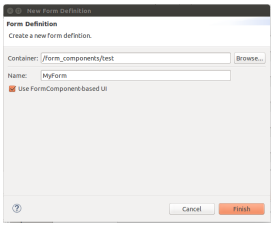


Figure 2.2 Creating a form definition

(c) Click **Finish**.

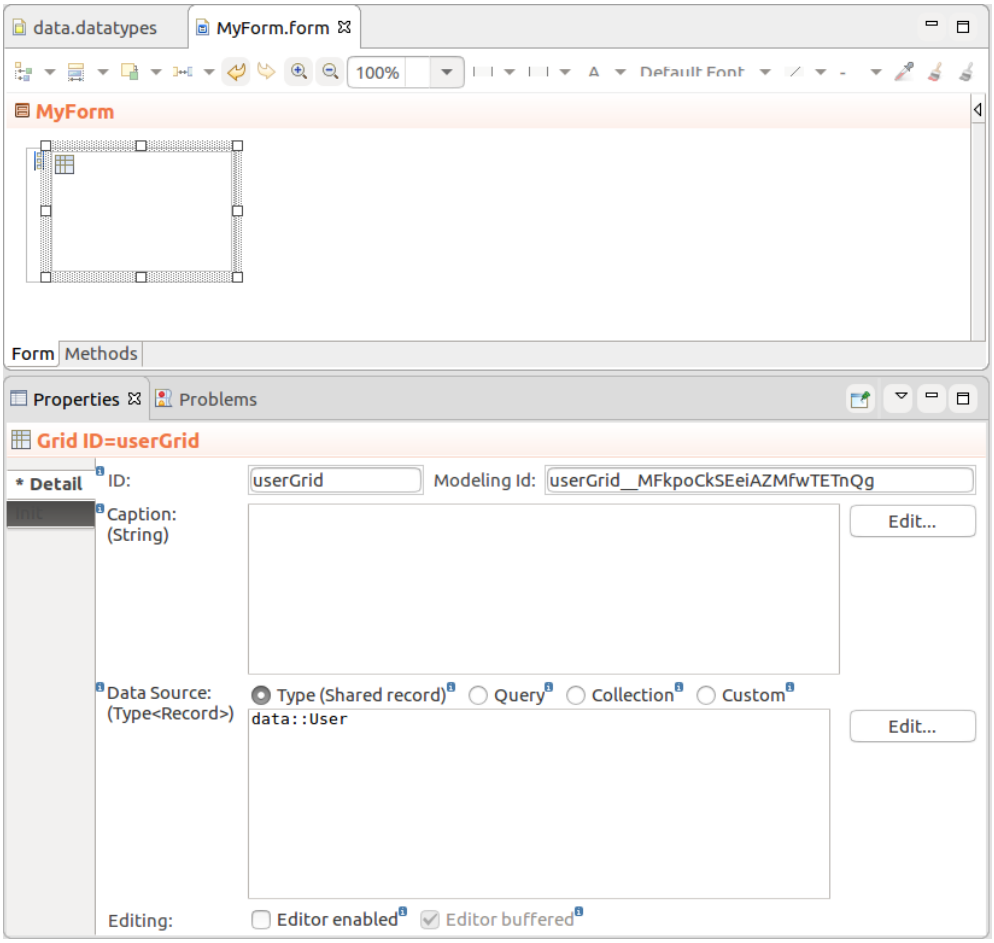
3. In the Form editor, create a grid that will display a list of users in a tabular manner:

   (a) Right-click the canvas, go to **Insert Component** > **Container Components** > **Vertical Layout**.

   (b) Right-click the Vertical Layout, go to **Insert Component** > **Output Components** > **Grid**.

   (c) Open the Properties view of the Grid and define the following:

   - ID unique within the form
   - data source as *Type* and insert the name of the shared Record so the table will use as its data source a list of all shared Records of the type.



   (d) Create a Grid Column with the user name:

   i. In the palette, click Grid Column  and then click into the Grid.

   ii. In the Column's Property view, under *Value Provider* select *Property path* and enter the property path to the name below.

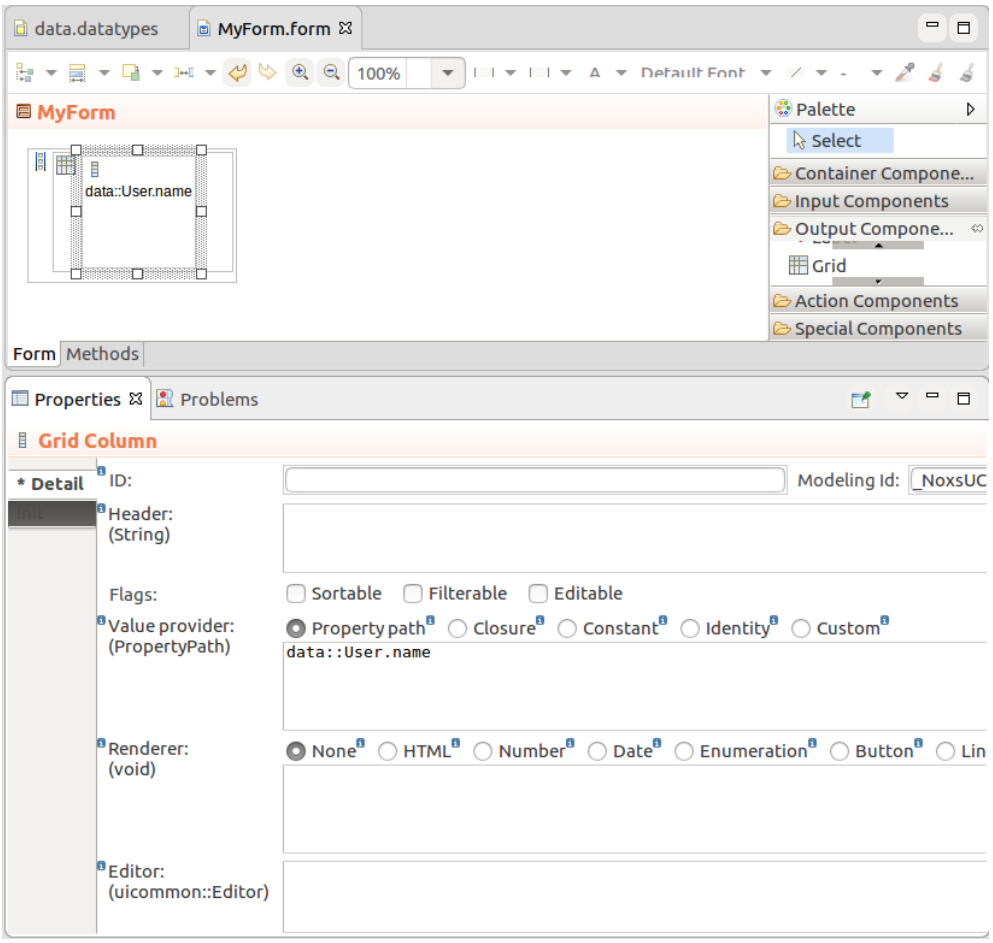


Figure 2.3 Column with record field

(e) Create another Grid Column with the date of birth:

i. Insert another Grid Column  into the Grid.

ii. Set *Value Provider* to *Property path* with the value `User.dob`.

iii. Set *Renderer* to *LocalDate*.

4. Create a popup for entering new user data:

(a) Insert the Popup component from the *Container Components* palette section below the Grid.

(b) Define its ID and Method signature for its factory call.

We will call the method as a button action to create and display the popup.

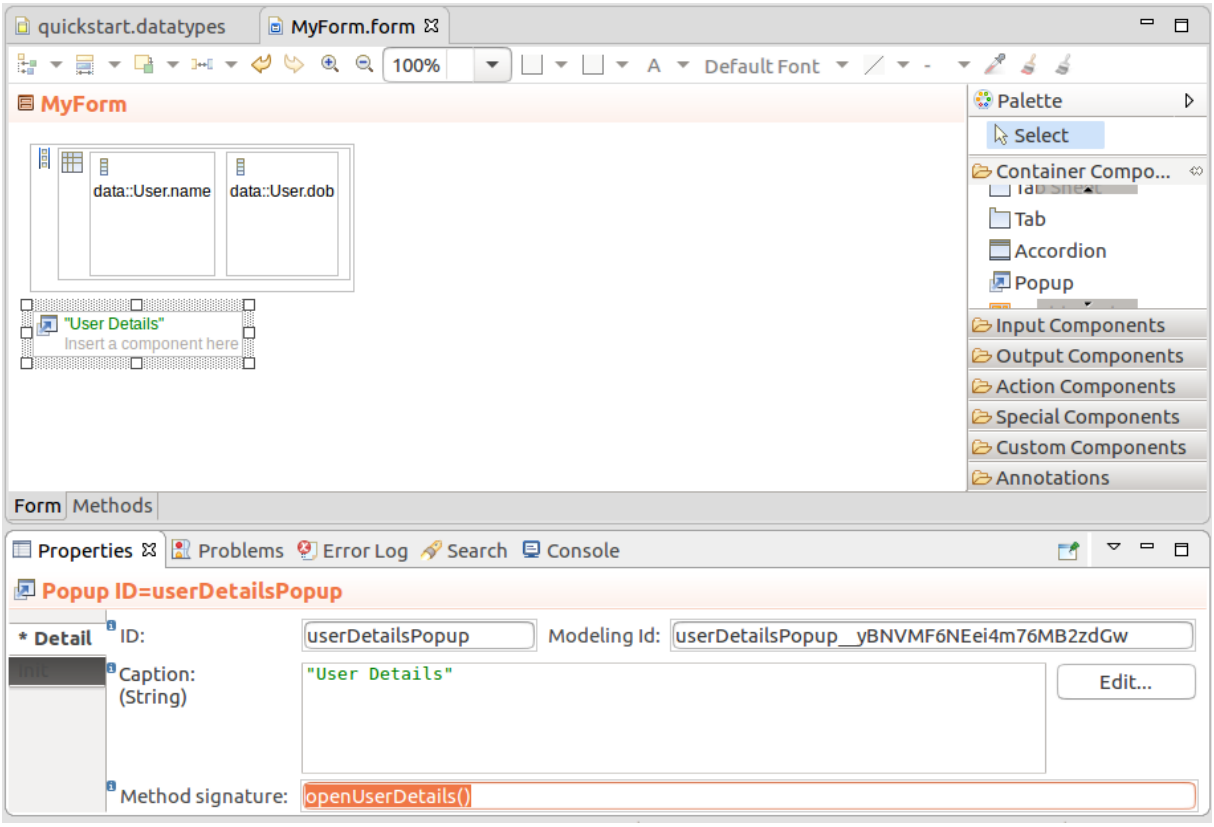


Figure 2.4 Popup

(c) From the Outline view, create local variables on the popup:

- variable for a record proxy set of type RecordProxySet with the initial value `createProxySet(null)`
- variable for the proxy of the new user of type User with the initial value `recProxySet.proxy(data::User)`

Proxy sets with proxies exist "above" the records: this makes sure that the user data is not persisted before the user confirms that the user should be created. The aim of proxy sets and proxies is also to allow us to discard the user data without having to clean up any transient data (for further information, refer to the modeling guide).

5. Design the popup content:

(a) Insert a Vertical Layout into the Popup: select the Margin and Spacing options in its properties.

(b) Insert a Form Layout into the Vertical Layout.

(c) Insert a Text Field into the Form Layout with its binding set to *Reference* and its value to the *name* property of the *user* variable.

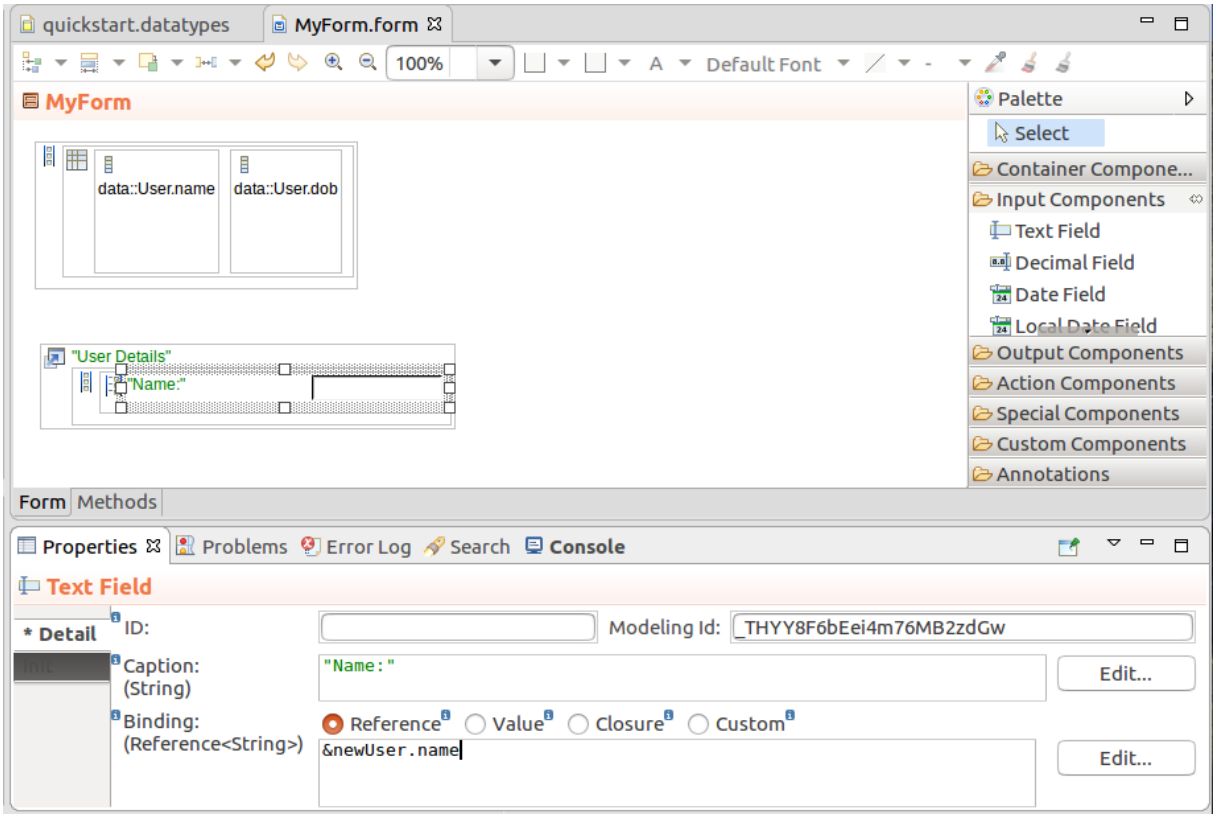


Figure 2.5 Text field with user name Binding

(d) Insert a Local Date Field into the Form Layout with its binding set to *Reference* and its value to the *dob* property of the User object.

(e) Insert the *Create* button and define the create logic as its click action:

- The Proxy newUser object merges: a "proper" User object is created.
- The table refreshes so that the new user appears in the table.
- The Popup closes.

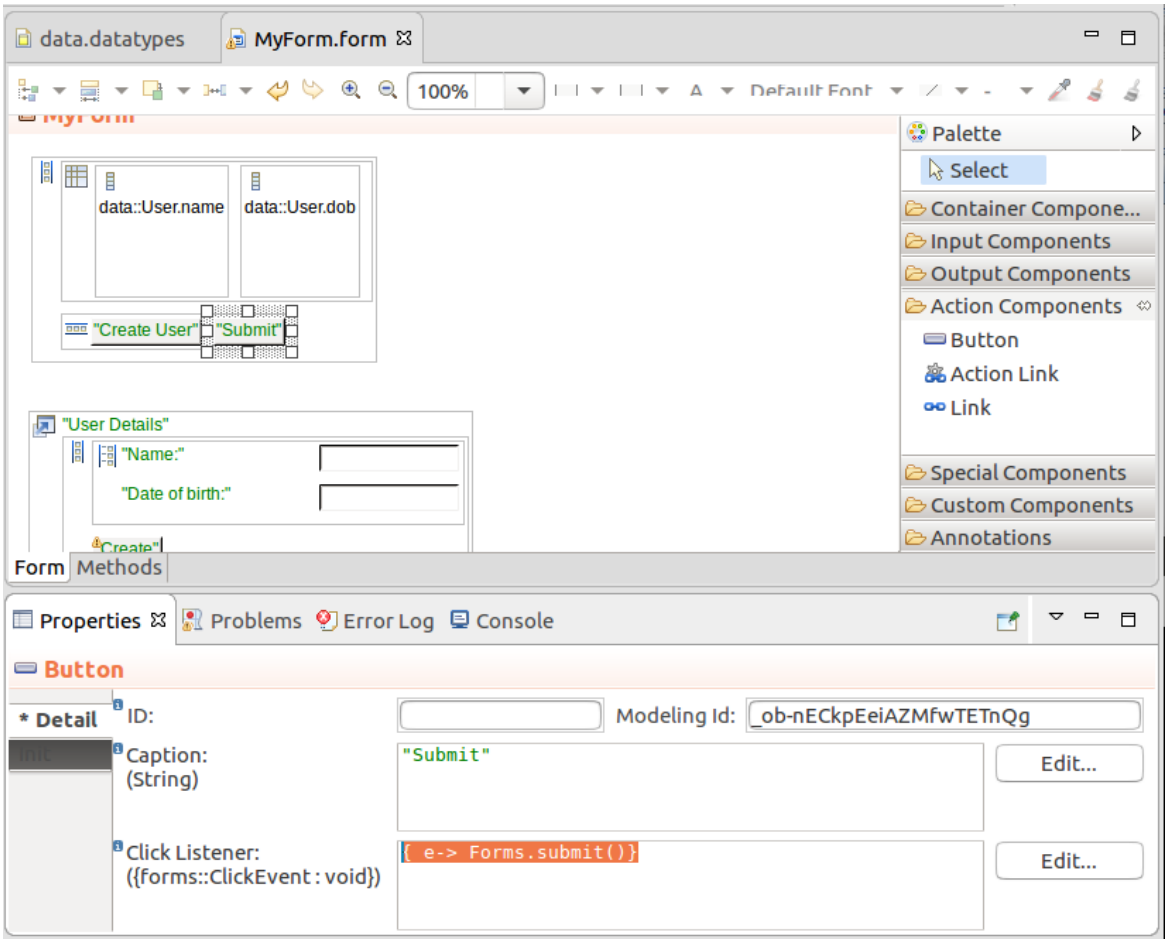
```
{   click ->
    recProxySet.merge(false);
    userGrid.refresh();
    userDetailsPopup.hide()
}
```

6. Below the Grid, create a button that will open the popup:

(a) Insert a Button component under the Grid.

(b) Define its *Click Listener* expression to open a popup: `{ click -> openUserDetails().show()}`

7. Finally, we want to add the option to submit the form on button click: add a Button to the form and call `{ c -> Forms.submit()}` from its click listener.



As we will demonstrate next, a form can be used by a User Task or a Document:

- When used by a User Task, on submit the execution of the User Task finishes and the process execution continues.
- When used by a Documents, the system navigates away: under the hood, the server creates a model instance whenever you open a document and finishes the instance when you submit the form.

2.3 Using the Form in a Document

To *create a Document with your Form*, do the following:

1. Create a document definition:
   (a) Right-click your Module > **New** > **Document Definition**; enter name and click **Finish**
   (b) In the document definition, click **Add**.
   (c) Define the document properties and its UI definition.

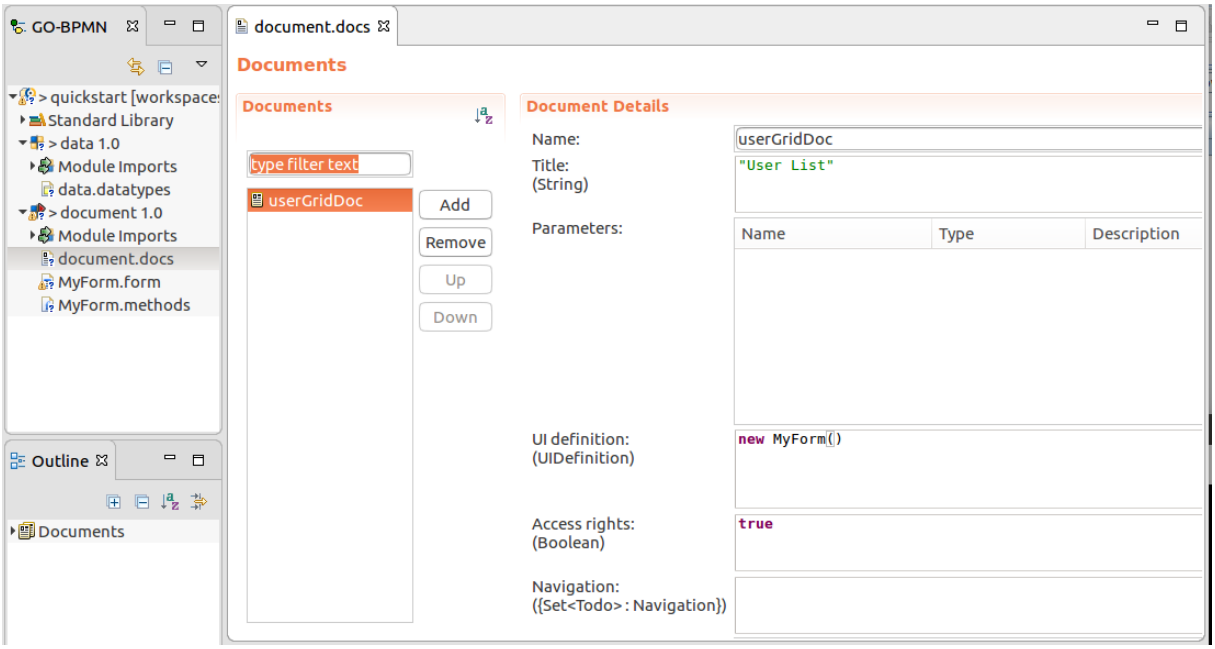


Figure 2.6 Document definition that will display our Form as its content

Open the document:

2. Make sure you are connected to an LSPS server or run the LSPS Embedded Server: go to **Server Connections** > **LSPS Embedded Server** > **Start**.

3. In GO-BPMN Explorer, right-click your Module, go to *Upload As* > *Model*.

4. Open a browser and go to your application URL (for the Embedded Server by default `http://localhost:8080/lsps-application/ui`)

5. After you log in, go to Documents and click the document.

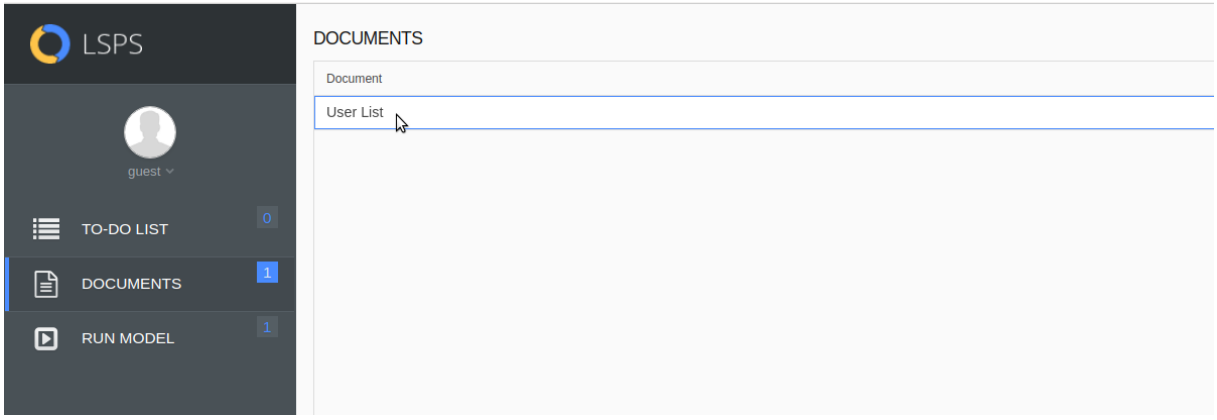


Figure 2.7 Opening Document

6. Create new users in the document.

7. When finished, click *Submit*.

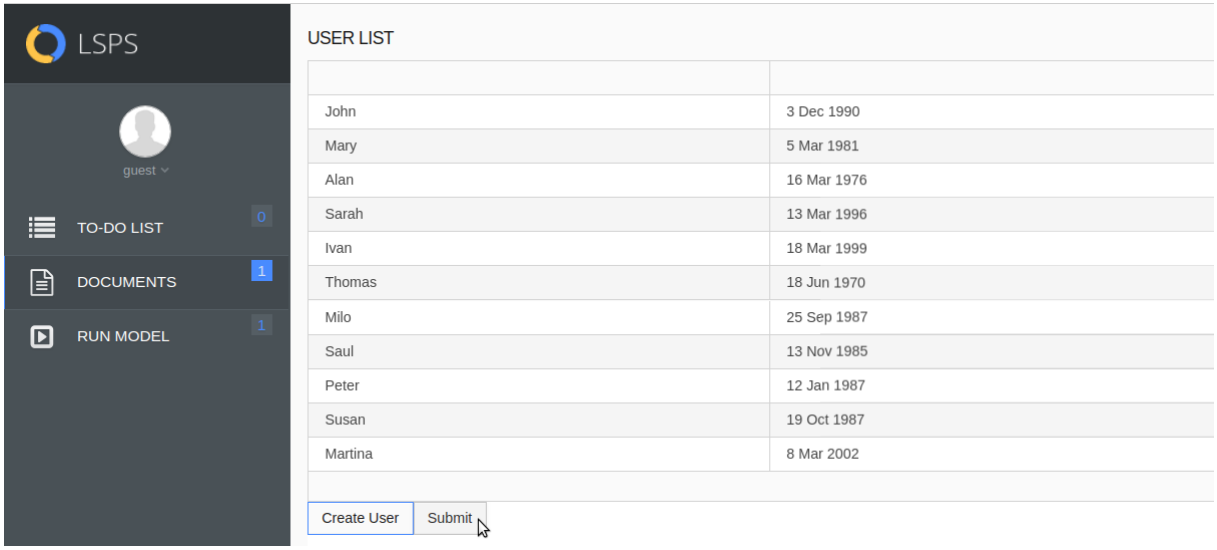


Figure 2.8 Submitting a filled-in document

2.4 Using the Form in a User Task

Create a Process with a User Task with your Form:

1. Create a process definition: right-click your Module > **New** > **Process Definition**; enter name and click **Finish**

2. In the process editor, right-click the canvas and go to **New** > **Task**.

3. In the **Select Task Type** dialog, select **User** from the *human* module.

4. On the *Parameters* tab in its Properties view, define the Task properties with its UI definition set to create an instance of the form.

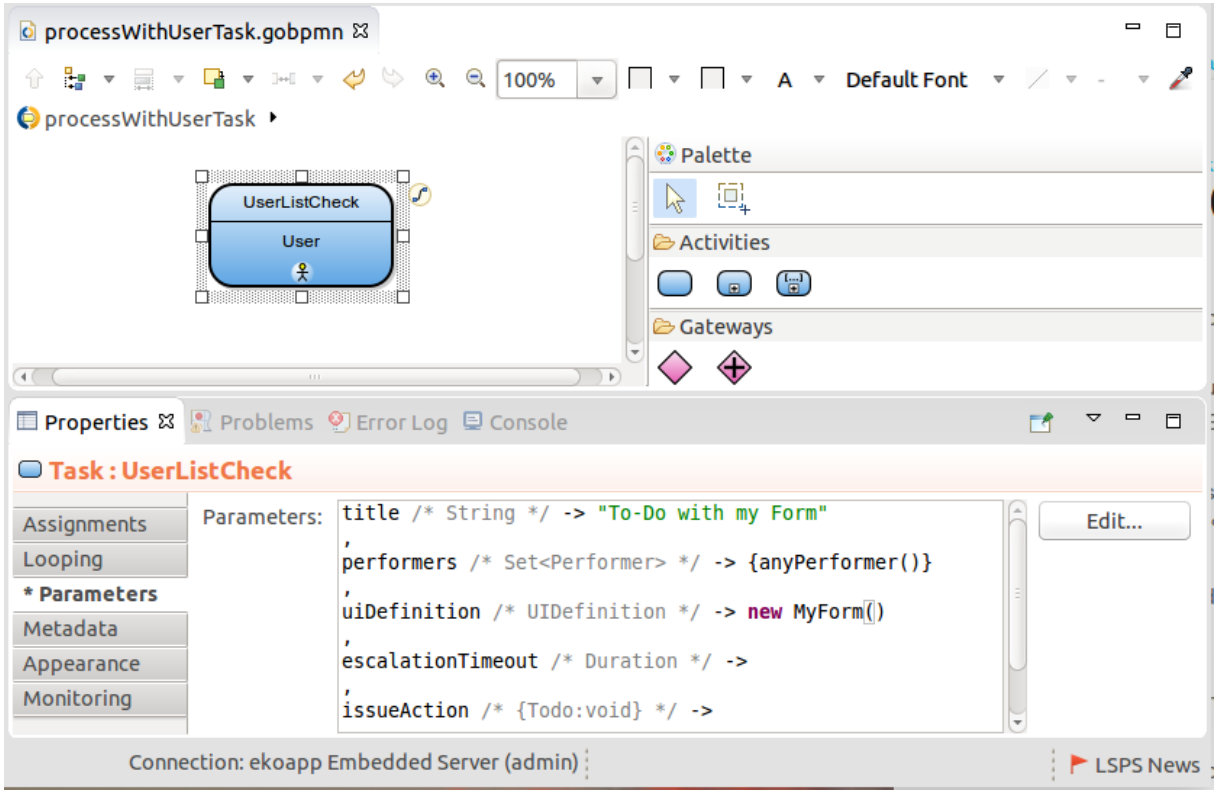


Figure 2.9 User task that will display your Form

To check the document and process and display the form, do the following:

5. Make sure you are connected to an LSPS server or run the LSPS Embedded Server (go to **Server Connections** > **LSPS Embedded Server** > **Start**).
6. In GO-BPMN Explorer, right-click your Module, go to *Run As* > *Model*.
7. Open a browser and go to your application URL (for the Embedded Server by default `http://localhost:8080/lsps-application/ui`)
8. After you log in, go to the To-Do List and open the to-do.

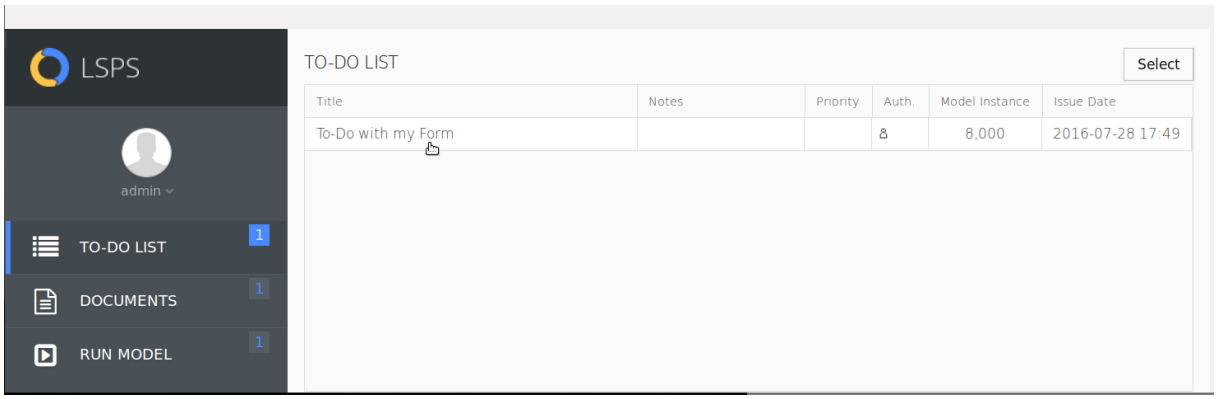


Figure 2.10 Opening To-Do

9. Create new users in the document.

10. When finished, click *Submit*.

Chapter 3

Form Definition

A *form definition* holds one form with its constructors, members (local variables and methods), and the tree of form components, such as, layout components, input components, output components, etc.

> **Important:** Chart components are based on `Vaadin Charts` developed by a third party. Make sure to obtain the Vaadin Chart licenses before you use them. The users of your application and Management Console, do not require any additional licenses.

3.1 Creating a Form Definition

To create a form definition file, do the following:

1. Make sure you have created a project and module.
2. Remove the *ui* module from module imports of your module and import the *forms* module.

   > **Important:** Having both, the ui module and the forms module, imported in your module could cause exceptions on your forms in runtime.

3. In the GO-BPMN Explorer view, right-click the module.
4. In the displayed context menu, go to **New Form Definition**.

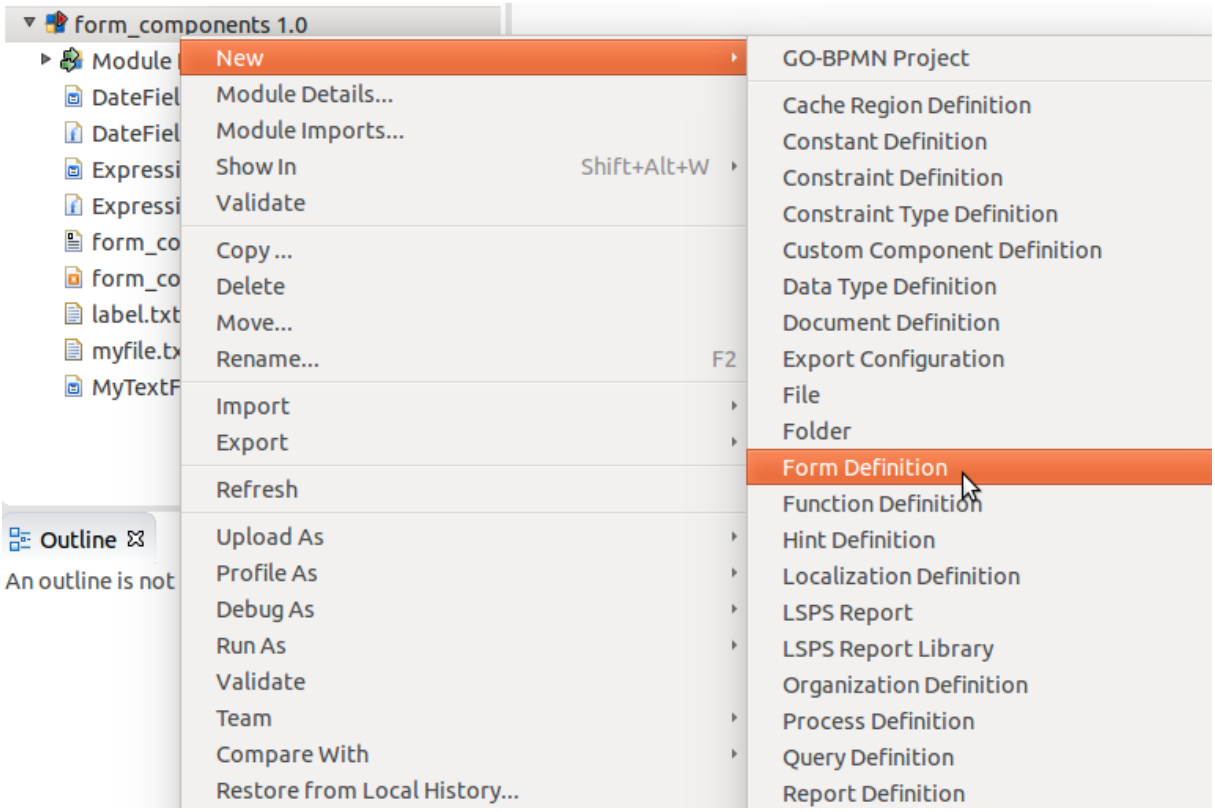


Figure 3.1 Creating a form definition

5. In the New Form Definition dialog, enter the form name and make sure the **Use FormComponent-based UI**.

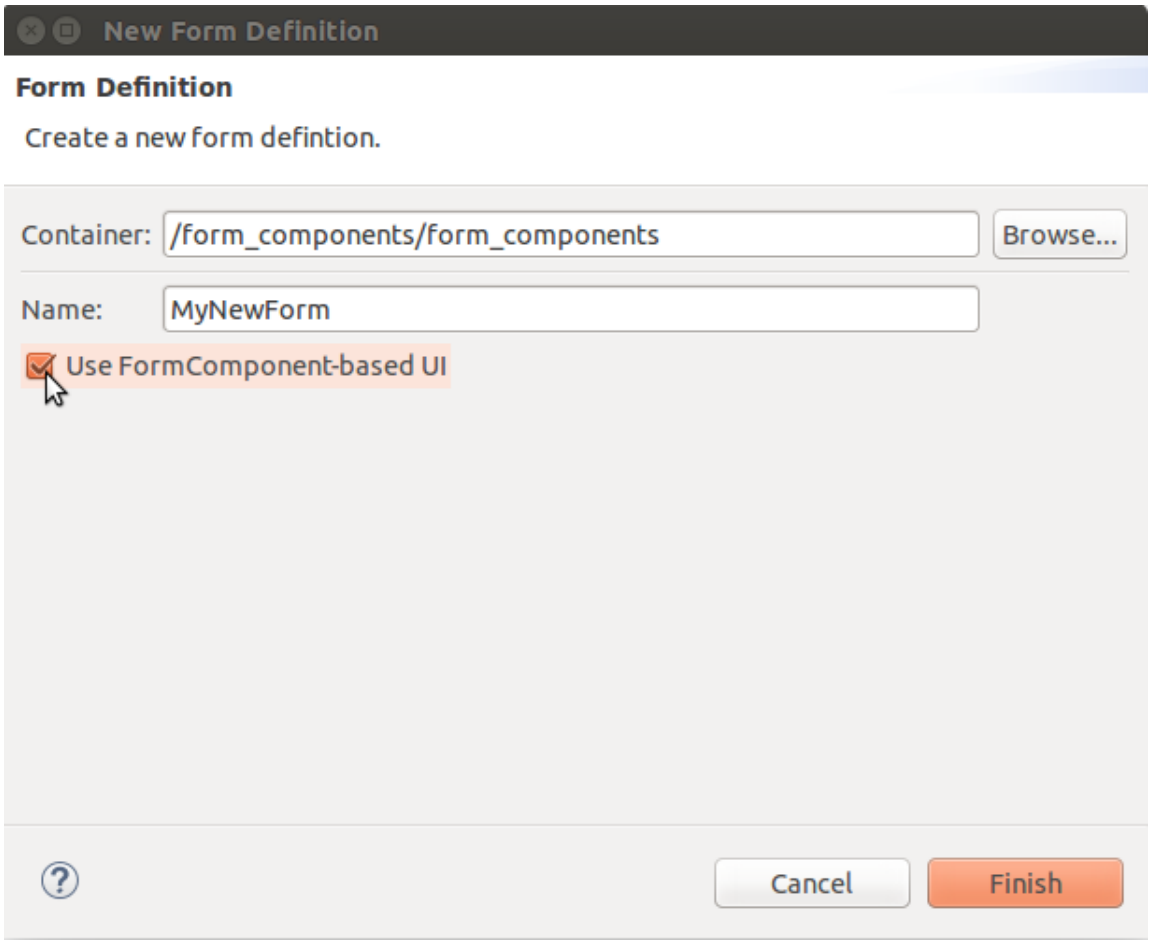


Figure 3.2 Flagging a form definition to create a form based on the forms module

3.2 Renaming a Form Definition

To rename a form definition and its form, right-click the Form root node in the Outline view or the file in the GO-BPMN Explorer and click **Rename** in the context menu.

Note that this will rename all the references to the Form root, the form definition file, the methods file.

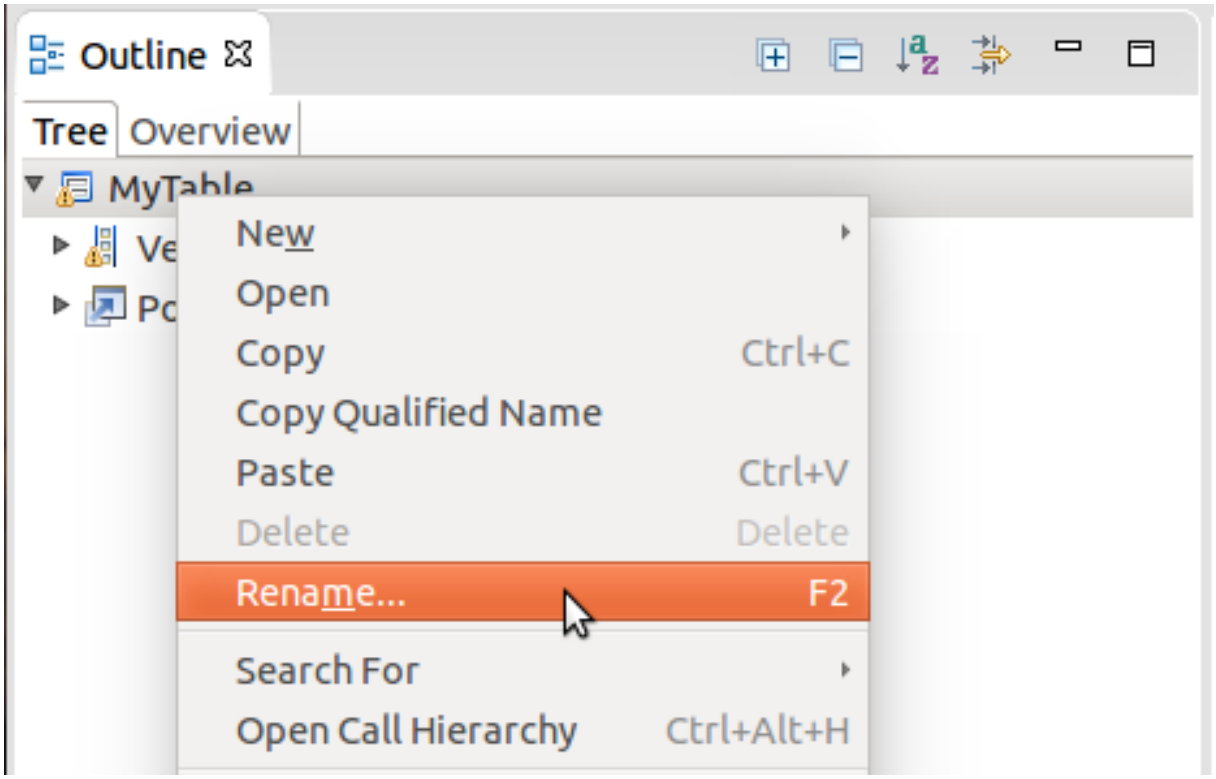


Figure 3.3 Renaming Form

Alternative, rename the form definition file.

3.3 Previewing a Form

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.

A form-preview configuration is created for the preview if it doesn't exist yet. Modify the preview configuration if you need to send input parameters to the form.

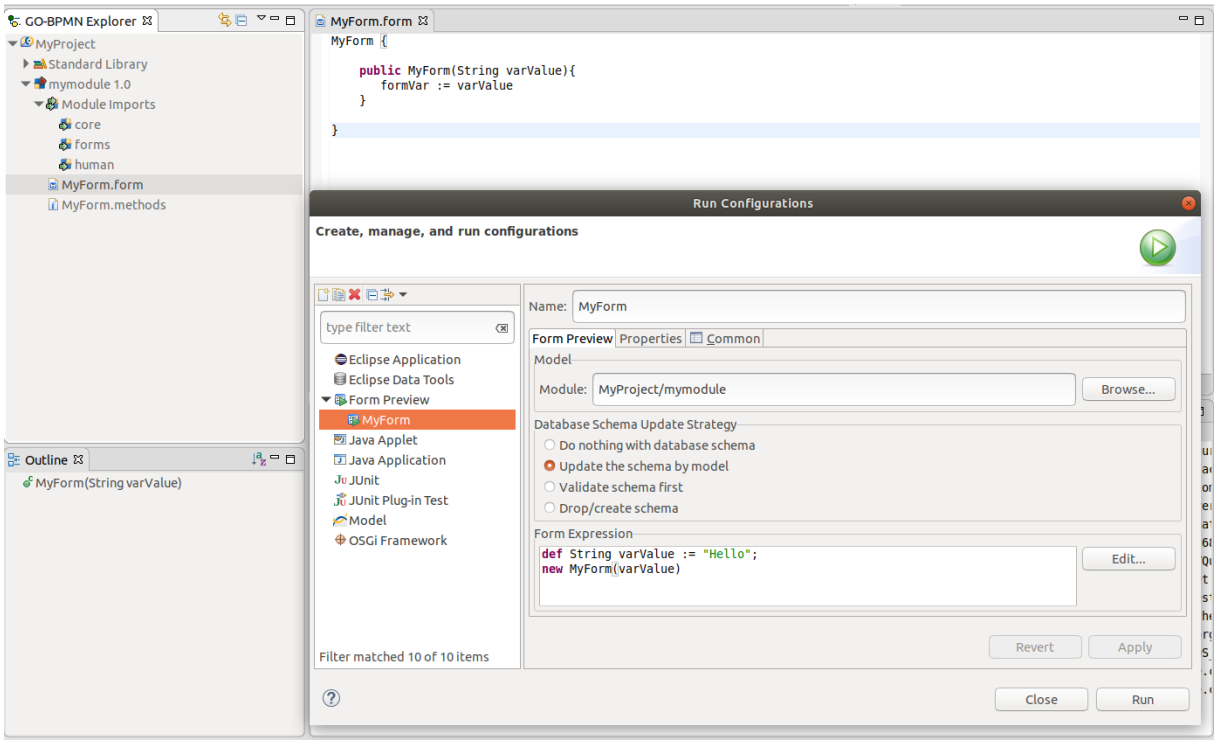


Figure 3.4 The form preview called with the argument Hello

When you request a form preview, the LSPS Server to which your PDS is connected, runs a persisted instance of the module with the form: it initializes global variables and form parameters, and creates the form context. Then your system browser with the form preview opens.

To delete the model instances created by form preview, open the *Model Instances* view of the *Management* perspective, right-click anywhere into the view and click *Remove All Form Preview Model Instances*.

> **Important:** Make sure to `disable this feature` in production environments.

Chapter 4

Form Design

A form is designed in the forms definition using the Forms editor. It comprises constructors, methods, local variable and a tree of form components, such as, input fields, radio lists, etc. To design the component tree, you insert the required form components into a canvas and define the properties of components in the Properties view.

On runtime, the form is resolved into a tree of components with the components based on the components in the definition: one component in the definition can result in multiple components in the form on runtime, for example, when you use a repeater, the component in the repeater results in multiple runtime components. Each component on runtime can be attached only to one parent: a component needs to be detached for its current parent before you can attach a component to another parent.

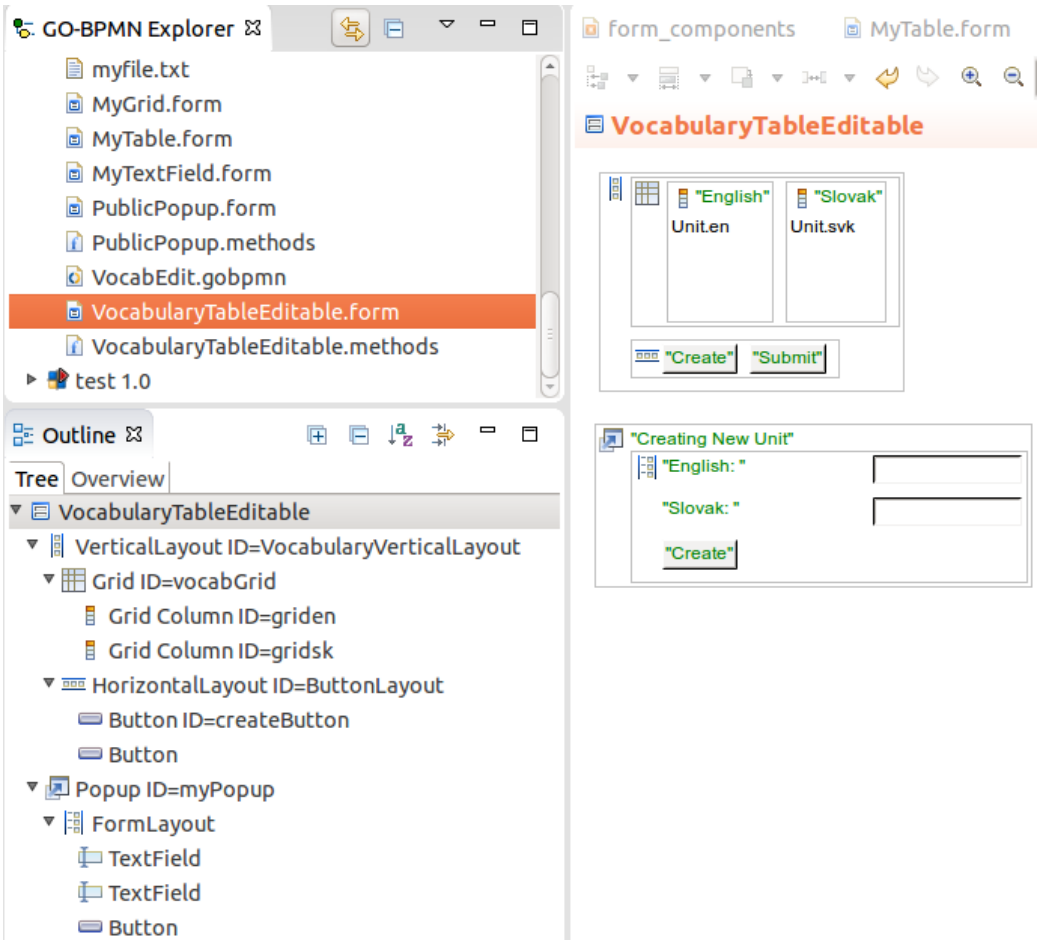


Figure 4.1 A forms::form definition with a Form

The form is instantiated by a *User Task* or a *Document* based on its *UI Definition* property. The property holds the constructor call that the task or document use to create their form. The constructor calls the `createWidget()` method, which does the following:

1. It calls the preCreateWidget() method as defined by the user is executed.
2. It creates the root component of the form tree, `FormWidget`.
3. It constructs the tree of form components as defined in the form definition.
4. It calls the *Init* expressions defined on the components of the tree.
5. It calls the postCreateWidget() method as defined by the user is executed.

Then form is then rendered.

The constructor call can be parametric so you can send input to your form when it is instantiated.

The *FormWidget* component cannot be changed; however, you can access it with the `getWidget()` method.

When the user interacts with the form, components of the tree produce event objects. These events trigger listener expressions: When an event occurs, its listener expression is performed; for example, when the user clicks a button, a ClickEvent is produced, and the Click Listener expression of the button is executed; when the user inserts a value, a ValueChangeEvent is produced, and the expression defined as the OnChangeEventListener of the component is executed.

The component tree can grow as the user works with the form since components can be added to the tree dynamically; for example, a new popup can be created every time the user clicks a button. To prevent storing of unnecessarily large trees, the server calls the garbage collector over the form: The collector prunes the component tree branches that are not referenced. The collector is called when the form is submitted or saved.

> **Important:** The example forms in this guide often work with data stored in variables. In production, such solutions are not intended for storage of extensive data amount and might result in out-of-memory errors. When acquiring data for forms use queries whenever the data is persisted in a database.

4.1 Defining a Parametric Constructor

To define a constructor for your Form, open the **methods file of your Form** and define the constructor as a new constructor method.

An example parametric constructor: The initialDate variable is defined as the form's local variable.

```
 DateField {
     public DateField(Date date) {
     initialDate := date;
   }
}
```

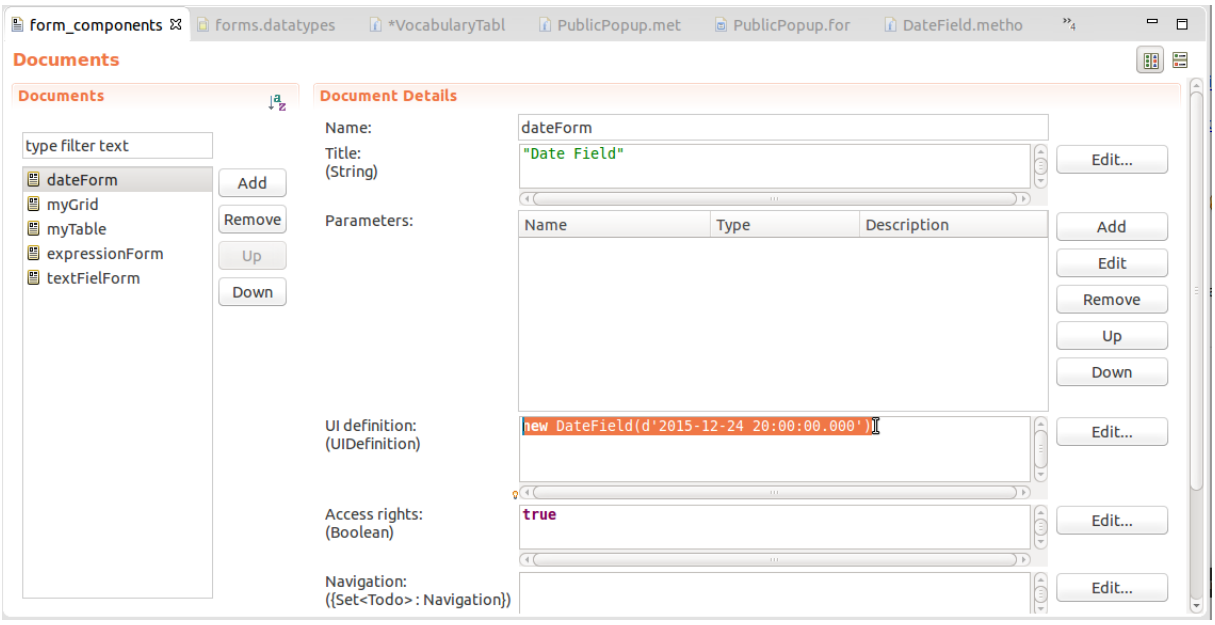


Figure 4.2 Calling a parametric form constructor from a document

4.2 Defining a Local Form Variable

To define a local form variable, do the following:

1. Make sure you have the form resource opened in the Form editor.
2. In the Outline view, right-click the form and go to **New** > **Variable**.
3. In the displayed Properties view, define the variable properties.
4. In the methods file of your Form, define the constructor that will initialize the value of the variables if applicable.

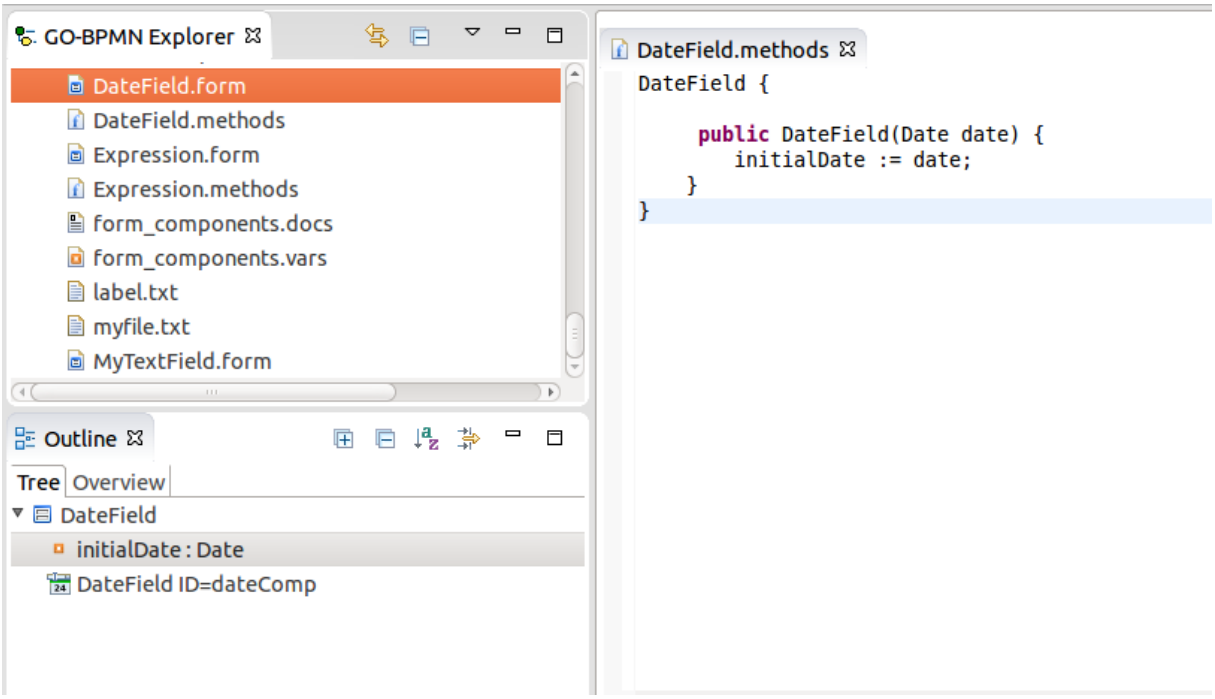


Figure 4.3 Parameterized form constructor that sets a local variable value

4.3 Creating a Form Hierarchy

> **Important:** Note that since both, the *ui* and *forms* implementations, can be used to return a UIDefinition to a Document or To-do; sometimes it can be unclear on runtime which implementation is used; typically if you use an expression that returns `human::UIDefinition`. The LSPS Server might not be able to detect which implementation you are using and return an exception. Therefore make sure *to remove either the ui or the forms module form your module imports* so the used implementation can be recognized correctly based on the imported module.

To create the component hierarchy in your form definition, do the following:

1. Open the form definition file (double-click it in the *GO-BPMN Explorer*).

2. Click the required component in the palette and then click to the respective position on the canvas. Alternatively right-click the position in the canvas, go to *Insert Component* and select the component from the context menu.

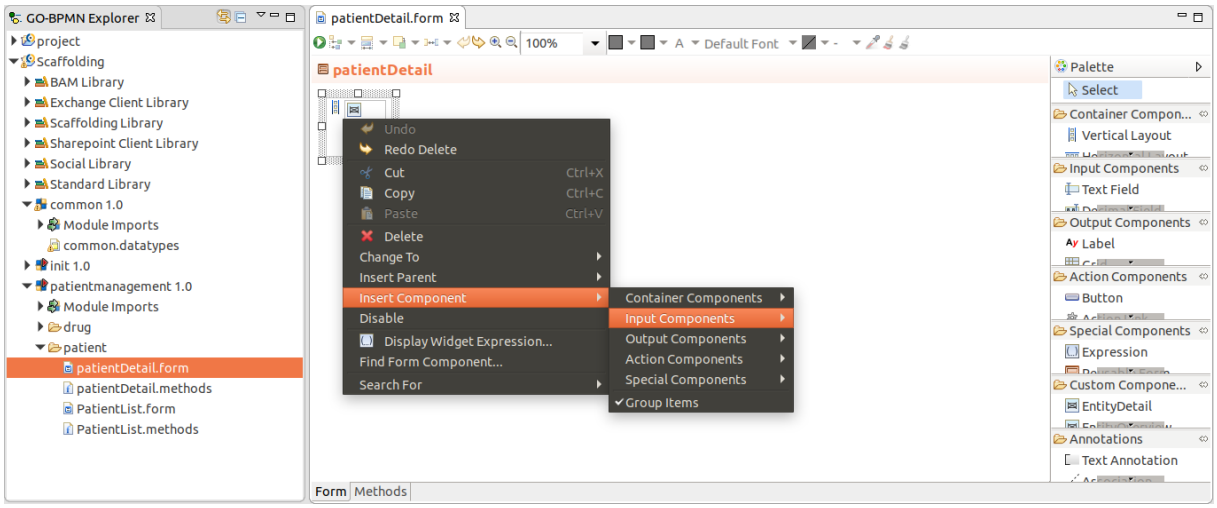


Figure 4.4 Inserting a component into the form from the context menu

3. Define the component properties in the Properties view on the Details tab; if you require other properties, such as, style properties and additional size properties, call the respective methods on your component on the Init tab as an expression.

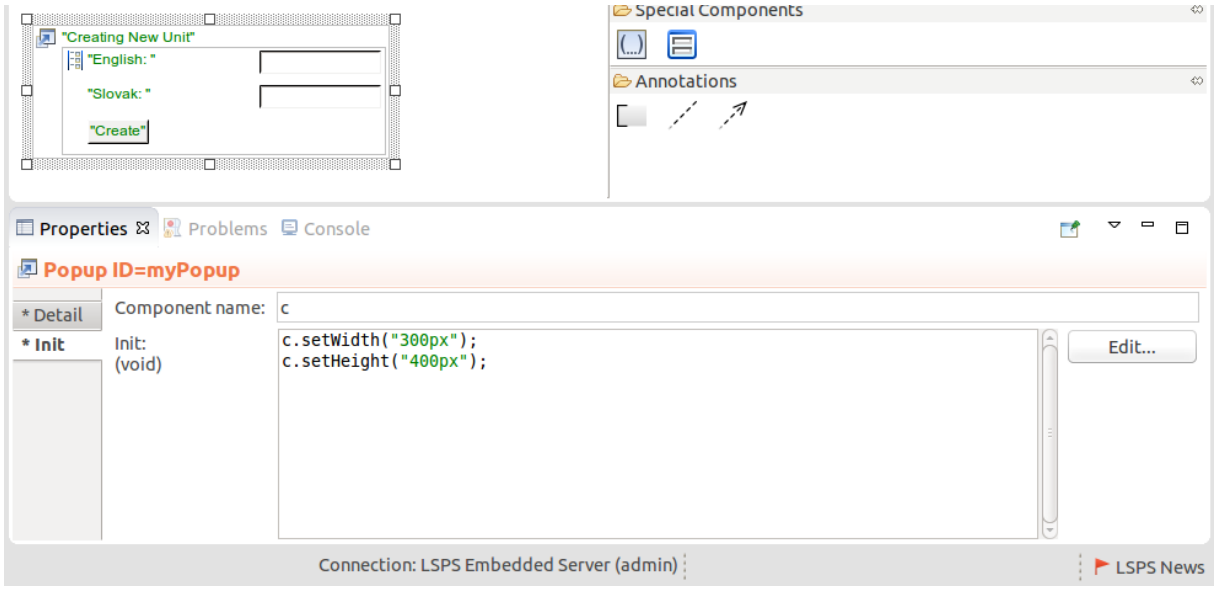


Figure 4.5 Init expression on a poopup component

This method can be used to reuse a Form.

4.4 Inserting a Parent Component

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.

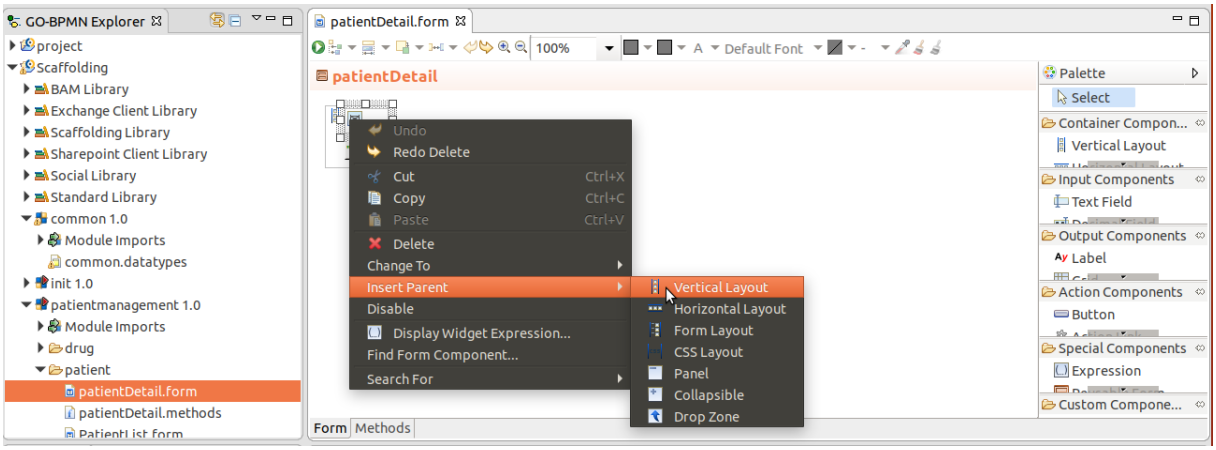


Figure 4.6 Wrapping component in a layout component

4.5 Deleting a Parent Component

To delete only the 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.

4.6 Searching for Form Components

If an error on a form component occurs during runtime, the server returns an error message with the modeling ID of the form component. While the modeling id is generated automatically, you can change it with `setModeling↩Id()`. To get the modeling id of a component, call the `getModelingId()` method.

To search for the form component that caused the error in the current workspace, do the following:

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

4.7 Displaying Form Source Code

Since forms as well as their components are defined as Records with methods, you can display the entire Form source in the Expression editor:

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

4.8 Enabling Error Reporting on Components

To enable reporting of runtime errors so that they contain modeling IDs of involved components, run your server with the `-Dcom.whitestein.lsps.vaadin.ui.debug=true` system property:

- If you are running PDS Embedded Server, go to **Server Connection** -> **Server Connection Settings**, select the connection and click **Edit**.

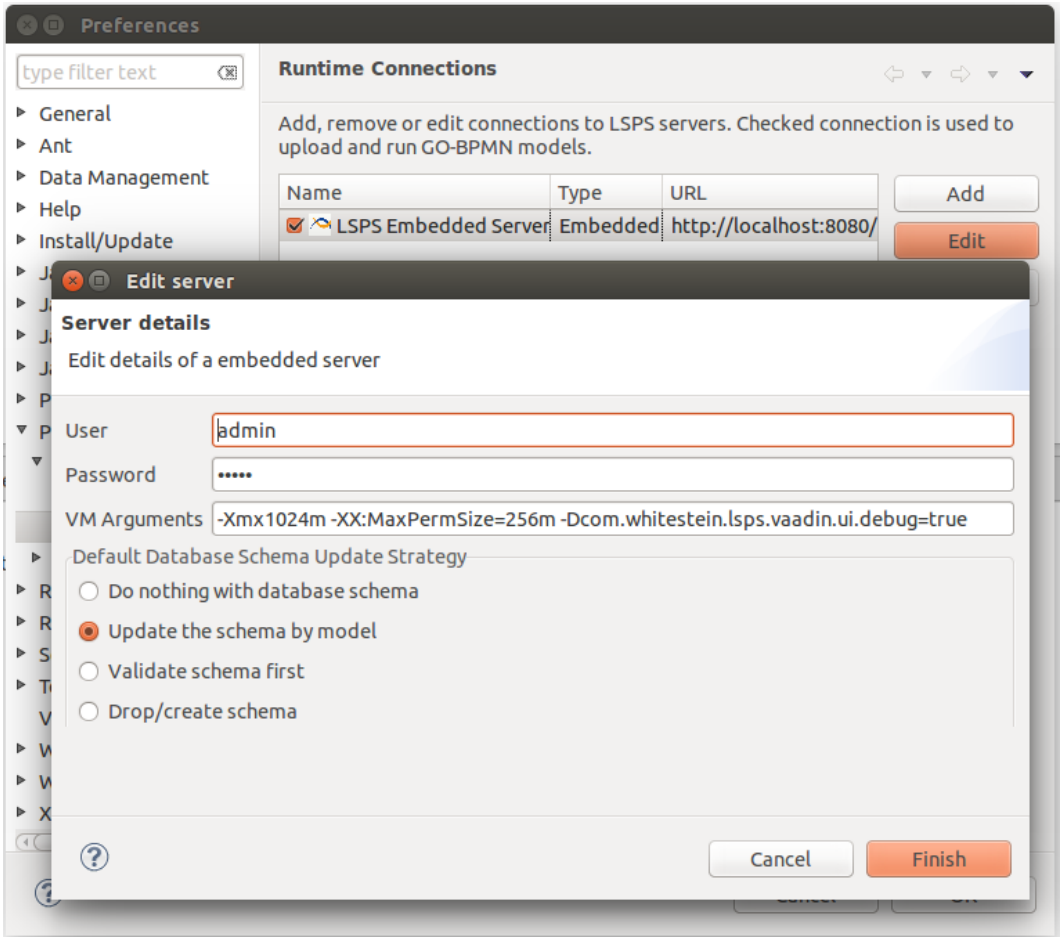
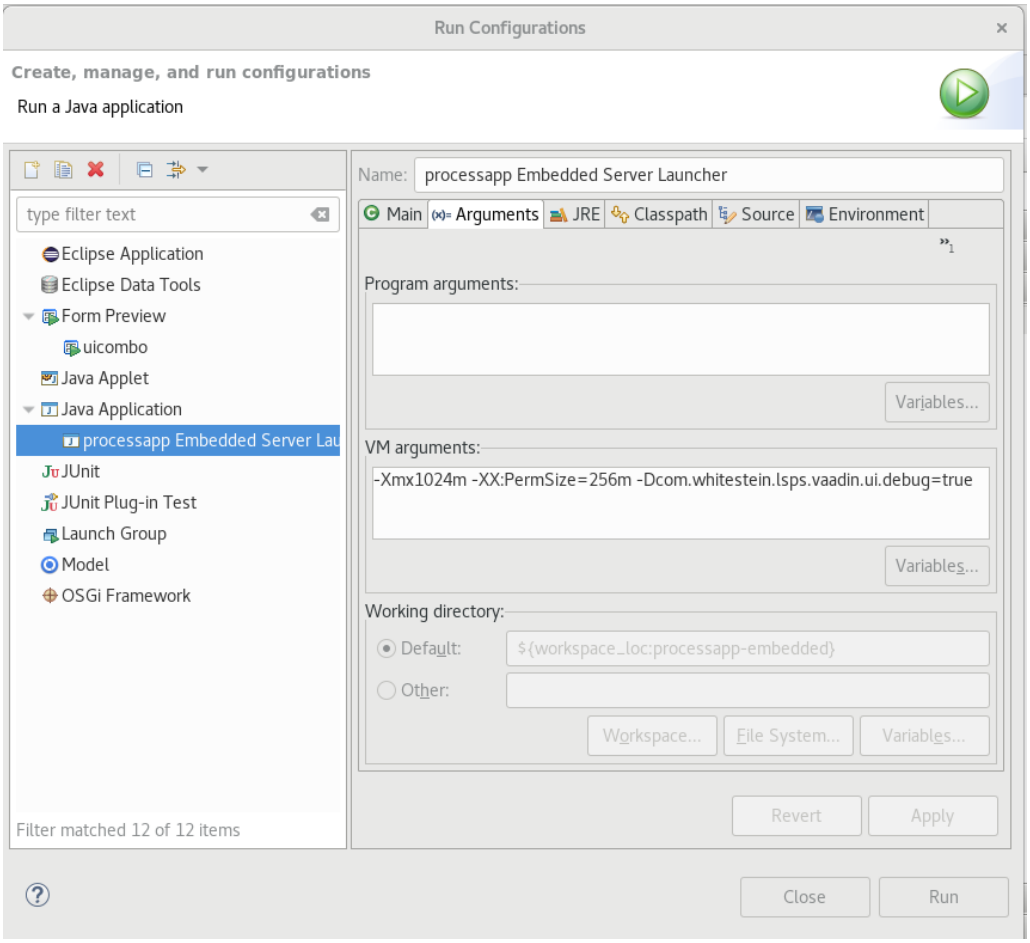


Figure 4.7 Enabling modeling IDs on form components

- If you are connecting to another PDS server or you are using SDK Embedded Server, switch to Java perspective; go to **Run** > **Run Configurations**; select the configuration under **Java Application** or **Remote Java Application** and add the property on the *Arguments* tab to *VM arguments*.



4.9 Troubleshooting a Form

When troubleshooting a form, consider the following:

- Examine the form expression: right-click the form in the form editor and select **Display Widget Expression**.
- Add breakpoints to expressions in the form and run it in `LSPS Debugger`.

Chapter 5

Actions on Forms

All forms allow you to perform the following actions:

- execute some logic before and after the form is initialized
- acquire device location
- acquire browser window size
- create transient versions of persisted data
- save the form
- submit the form
- navigate to another page

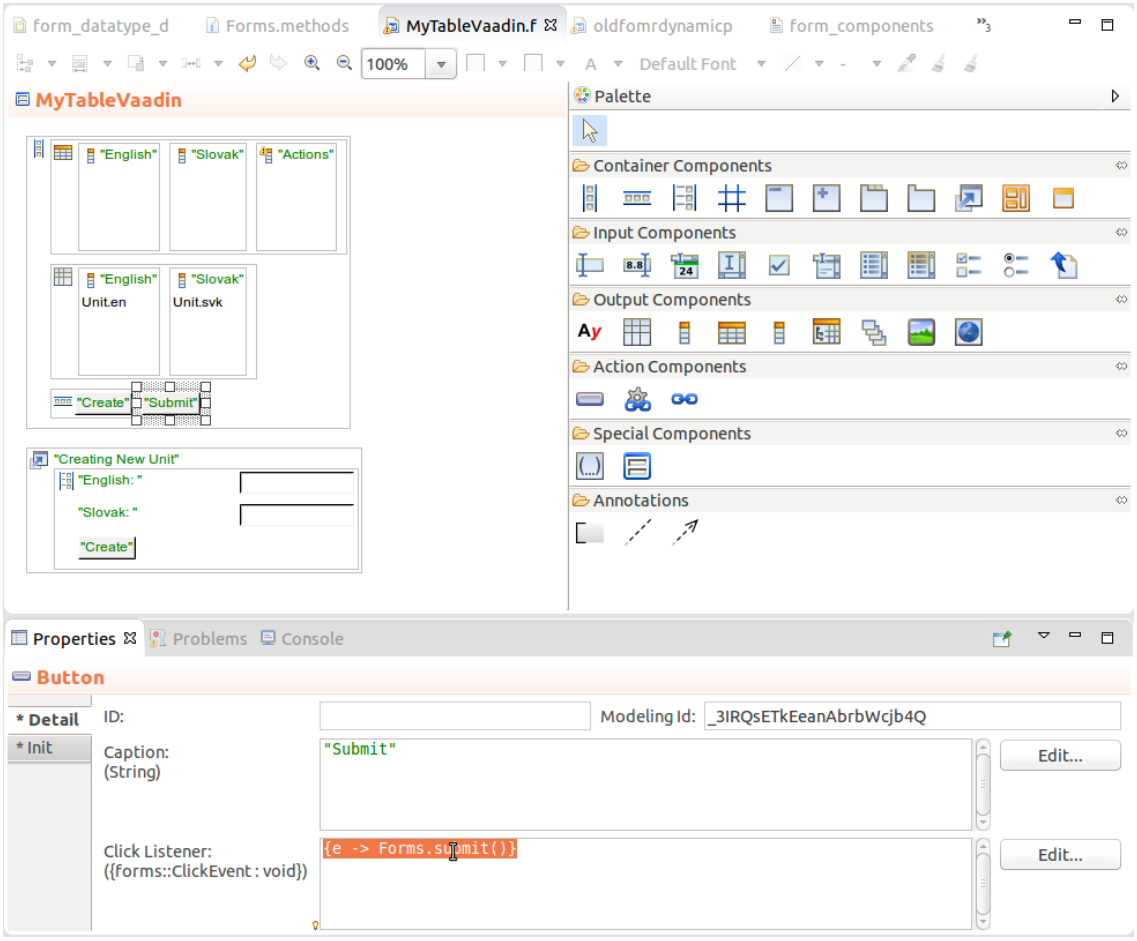


Figure 5.1 Submit call attached to button click listener

5.1 Executing Logic Before and After Form Initialization

To execute some logic, before or after the component tree of your form is initialized (to create a pre-init or post-init hook), add the required code to the `preCreateWidget()` or `postCreateWidget()` method in the methods file of the form.

Example content of the MyTable.methods file with methods for MyTable.form

```
 MyTable {
~
  private void preCreateWidget() {
     //running an asynchronous model instance that will notify
     //the respective users about that somebody opened the document and
     //is about to edit it:
     createModelInstance(false, getModel("notifications"), null);
  }
~
  private void postCreateWidget() {
      //get a list of form components of the hl layout component:
      componentList := hl.getComponents();
      //sets the value of a label to display the list of components:
      compListLabel.setValue(toString(componentList));
      //display a popup on form load:
      createMyPopup().show();
  }
}
```

5.2 Detecting Location

To detect geoposition of the device in your form, use the `Forms.detectLocation()` method.

The instructions for the client browser on how to provide the geographical position data is defined by the PositionOptions object that can be passed as parameter:

- **enablehighaccuracy**: position accuracy (if true, the GPS position is requested; the default value is `false`)
- **timeout**: time in milliseconds that can pass before the call fails

  On such failure, the call returns `GeolocatorError.Timeout`. The countdown starts when the user permits the detection. By default, no timeout is applied.
- **maximumAge**: maximum accepted age of a cached position

  The default value is 0, so that cached positions are not accepted.

If no PositionOptions is passed the default values are applied.

Example detectLocation() call

```
Forms.detectLocation(
      new forms::PositionOptions(
            enableHighAccuracy -> true,
            timeout -> 10000,
            maximumAge -> 1000), { ge ->
                forms::notify( caption-> "This is the position: "+ ge.position + " error: " + ge.
                }
);
```

5.3 Checking the Size of the Browser Window

To get the current size of the browser window in DIPs, run the *Forms.getBrowserWindowSize()* method.

5.4 Creating Transient Data in Forms

When editing a shared record in forms, the changes are applied instantly: when you change the value in a text field that is bound to a shared record property, the change is persisted when the field loses focus. Generally this is not the required behavior: you want to apply changes only on submit or when some other confirming action takes place.

To implement such a behavior, use `proxies` of your records in forms.

> **Important:** Proxy objects are not supported by forms created with the *ui* Module: these forms use `execution levels` created with the (../ui-vaadin/uispecialcomponents.html::viewmodel) component to isolate any changes instead.

When designing forms that make use of change proxies, consider creating and merging objects out of your form definition: This will keep your form definition easier to maintain and reuse since it will hold only presentation logic. You can do so by defining a parametric constructor that will take the logic-related arguments, such as, change proxies, and action handlers.

For examples, refer to Creating a Modal Popup and Creating a Public Popup.

5.5 Saving a Document or Todo

To save the Document or To-Do with your Form, call `Forms.save()`: the state of the Document or To-do will be saved as the internal state of the Document or To-do as part of its model instance. The method returns the shared record instances `human::SavedDocument` or `human::Todo` depending on whether it was created in a document or in a todo respectively. As part of the save action the form component tree is garbage collected by the LSPS garbage collector to prevent too big a component trees from being saved.

Note that while Saved To-dos remain accessible to the user even after they navigate away from the To-do page, you will probably need to provide a way how to access saved Documents: for example, you could create another document with a form that will contain a list of available saved Documents. Saved Documents can be accessed via the SavedDocuments shared record.

> **Important:** Saved documents and to-dos can be loaded only into the form there were saved from↩ : If you update the form, typically by uploading its new version as part of a new module, the saved documents and to-dos will not be compatible with the new form and attempts to load them into the form might fail.

> **Note:** The save action fails if the user attempts to save a Popup state.

5.5.1 Executing Logic on Restore of a Saved Document or To-Do

To execute some logic, when a saved to-do or a saved form are opened, define the `onLoad()` method in the methods file of the form.

Example onLoad() implementation

```
 MyForm {
~
  public MyForm() {
      status := "value set by the constructor";
  }
~
  protected void onLoad() {
      status := "value set when restored from a saved todo or document";
      getWidget().refresh();
  }
}
```

5.6 Submitting a To-Do or Document

To submit a to-do or document, call the static `Forms.submit()` method. Typically you will do so on button click: insert a *Button* component into the form; in its *Details* in the *Properties* view, define the call in the *Click Listener* property:

```
 {e -> Forms.submit()}
```

The method can define a *Navigation* parameter: on submit, the form will navigate to this location:

```
 {e -> if myForm.isValid() then Forms.submit(new UrlNavigation(url -> "www.whitestein.com", openNe
```

If the User Task or Document with the form defines their *Navigation* parameter, the submit navigation is ignored. If neither the User Task or Document or the *submit()* call define a Navigation, the system navigates to the home page of the application.

> **Note:** The *openNewTab* parameter is ignored in submit calls and the navigation is always performed in the same tab (otherwise, data of submitted todos and documents would remain accessible in the tabs).

5.7 Navigating from a To-Do or Document

To navigate to a target location from a form, use one of the navigate methods of the Forms record:

```
 def Button navigateButton := new Button("Navigate", {e -> Forms.navigateTo(new UrlNavigation(oper
//button that navigates to a new document that has a Navigation Factory function:
def Button navigateToDocumentButton := new Button("Navigate", {e -> Forms.navigateTo(getMyDocument
//button that navigates to the home page:
def Button navigateButton := new Button("Navigate", {e -> Forms.navigateTo(new UrlNavigation(url -
```

On navigation, the current `transaction` is committed: any action performed after the call takes place in a new transaction.

> **Note:** Navigation on submit for Documents and Todos can be defined directly on the Document and User Task definitions respectively as well. Such navigation overrides the navigation of *submit()* calls.

Chapter 6

Mobile Forms

LSPS provides support for creating mobile forms so that you can display the forms of LSPS applications in a specific way on a mobile device or display a mobile-friendly version of the form.

> **Note:** If you are creating a mobile LSPS application, do not aspire to substitute the desktop application: the mobile application should be an extension of the desktop application.
>
> Also make sure to follow the common guidelines for mobile applications: keep the forms simple and intuitive.

To create mobile versions of your forms, do the following:

- In your form, add the Resize Listener to detect the screen size on runtime: from the listener expression, call the `getBrowserWindowSize()` function to get the width and height of the device screen in DIP (density independent pixel) of the device.
- Define in the listener how the form components should behave:
  - For adaptive design you can simply hide components, or change their properties.
  - For mobile-dedicated version, consider embedding the form versions for different devices in a layout component and hide and display or hide the layouts from the Resize Listener expression based on the window size.

  Consider using the same binding for the input components in individual Form version to keep any data that the user provides in all form versions. This will allow the system to preserve the input in case a form component is hidden on device resize (the user rotates the device).

> **Important:** If the user provides invalid values in the form and the Form component will be hidden, the invalid values cannot transferred to the displayed form. Consider handling such situations in your form.

Chapter 7

Form Components

Form components constitute your form hierarchy: They are implemented as Records with Fields and methods and have the forms::Form Record as its super type with the form widget.

Every component allows a set of common component actions and can define the following of properties:

- **ID**: identifier of the component on design time

  ID allows you to reference the component from another component in the form while designing it.

- Modeling id: identifier of the component on runtime

  Modeling id is generated automatically and serves to identify form components on runtime.

- **Caption**: a String expression rendered as label

  If the caption references localization identifiers, these are resolved automatically to the target locale. Note that the caption is rendered by the parent layout component. Hence if you create a form with a sole component with a caption, the caption will not be rendered in the form. This is due to limits of the Vaadin framework.

- **Description**: tooltip displayed on hover

Component-specific properties can be set on the *Detail* tab in the Properties view or on the Init tab. Both become part of the form tree.

You will work with the following types of components:

- group components that can hold other components and generally define the form layout
- input and action components so the user can interact with the forms
- Expression component, so you can define your component in the Expression Language
- Reusable Form component so you can reuse an existing form

Note that you can `implement a custom form component` if required.

7.1 Binding of Components

Some form components hold a value: For example, a Text Area component holds a String value with its text. You can get the value with the `getValue()` call and set it with the `setValue()` call.

However, if you want to work with the value, you can bind the component to a variable or a variable field if the variable is of a Record type:

- When the value of the component changes, the value in the variable to the new value;
- When the component is loaded or refreshed, its value uses the value in its *Binding*.

The variable must be of the same type as the value in the component, in the case of the Text Area, it must be of type String.

> **Note:** If the user enters a value into a component, which cannot be stored in the target object, the component validation fails: In such a case, the value is not stored in the target object and an error message is displayed on the component.

7.1.1 Setting Component Binding

To set the target object of binding, either call the `setBinding()` method or set the *Binding* property in the Properties view.

> **Note:** By default, the binding is set to a default object which takes any value so you don't have to set the binding if not needed.

Depending on whether and when the binding value should be reloaded, you can use one of the following types:

- **Reference**: reloaded when the component is initialized or refreshed
- **Value**: loaded when the component is initialized (This is convenient if you want to set a fixed value to a component.)
- **Closure**: loaded when the component is initialized or refreshed
- **Custom**: custom implementation of the Binding interface (This allows you to apply custom logic to the binding before it is loaded.)

> **Example:** Let us consider a Form with three Label Fields (Label Fields are components that simply display the value of the Binding as a read-only value):
>
> - LabelA uses Reference Binding to the Date variable `&varTime` with its initial value set to the time returned by `now()`
> - LabelB uses Value Binding `now()`
> - LabelC uses Closure Binding `now()`
>
> All Labels will show the current date and time when the Form is loaded. If we add a Button component that will refresh the Label components on click, then on click:
>
> - LabelA will display the value stored in the `varTime` variable.
> - LabelB will keep the same time value.
> - LabelC will always update the time to the current time.

7.1.2 Getting Component Binding

To get the value set in the component target object, call `getValue()` on the component.

> **Note:** A component value is stored in the target object, only if it meets the formatting requirements, for example, if you enter a character into a Decimal Field, the value is ignored and the component displays an error. To work with such values, use the `getUserText()` method.

7.1.3 Clearing Component Binding

To clear the component binding, call `setBinding()` with the reference parameter set to `null`.

```
myTextField.setBinding(null as Reference)
```

7.1.4 Implementing Custom Binding

If you want to process binding data before they are used by a component, implement your custom Binding data type. This allows you, for example, to convert input values to other units.

1. Create the Binding data type that implements the forms::Binding interface.
   (a) Create data type definition
   (b) Import the *forms::Interface* type and extend it.

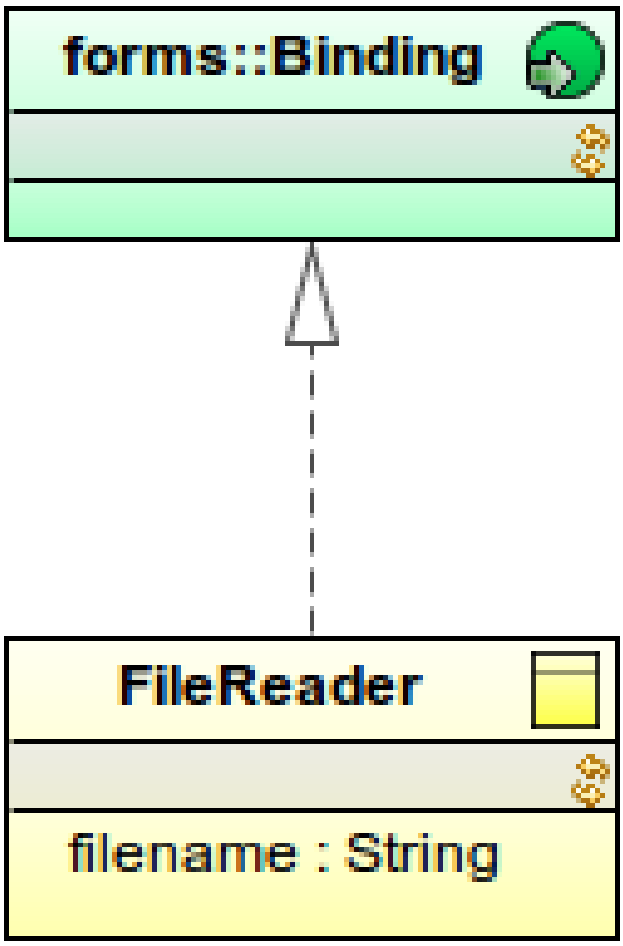


Figure 7.1 Custom binding record

2. Implement the interface methods:

   (a) Right-click your binding record and select *New Methods Definition*.

   (b) Open the newly created methods definition file and implement the following methods:

   - `get()` returns the component value
   - `set()` sets the component value
   - `getBinding()` returns the property of a RecordAndProperty object; the validations applicable for the property are used to validate the value (refer to the documentation on `constraints`); usually you want to return `null`;

```
FileReader {
  public Object get(){
      def File myfile := getResource("custom_binding", filename);
      fromBinaryToString(myfile.content);
    }
  public void set(Object value) {
      error("Cannot set target file! Readonly!")
  }
  public RecordAndProperty getBinding() {
    null
  }
}
```

Now you can use your binding implementation as Custom Binding in your form. Consider using and distributing your implementation as part of a custom Library.

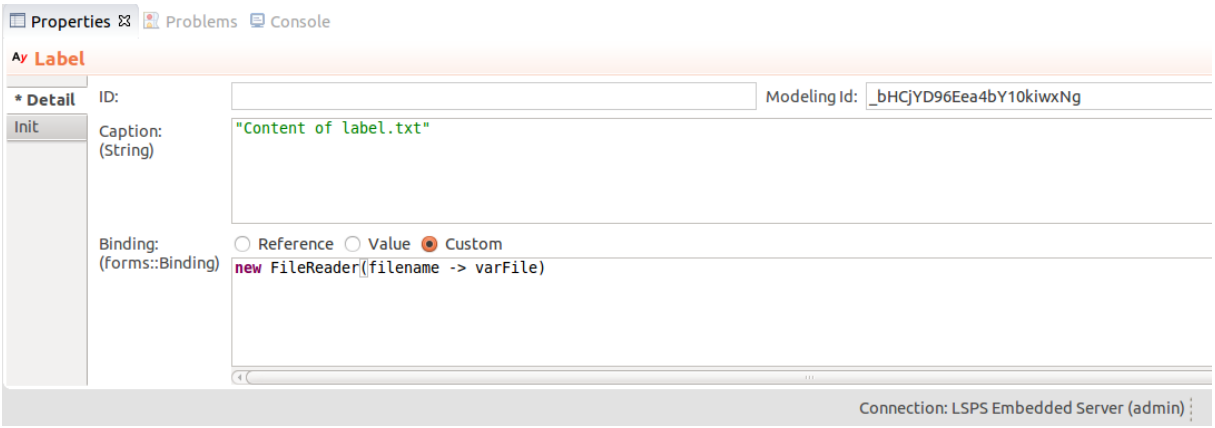


Figure 7.2 Custom binding on a Label component

7.1.5 Defining an Initial Value of a Component

To define the initial value for a component where the data comes in later, define a form variable with the initial values; and define the binding as a closure that provides the appropriate data:

For example `myFormVariable != null ? myFormVariable : null`

7.2 Validation of Components

When you insert a value into a form component, the form automatically checks if the value meets the basic format requirements, such as, whether you entered a decimal value in a Decimal Field

If you want to introduce further validation rules to a forms::form, you have the following options:

- Add validators to *input components* to validate input automatically.
- Set custom error messages on any form component: Handling of the message is fully under your control.
- If your components are bound to records, you can validate the input against the rules defined for the records.

7.2.1 Adding a Validator

To add validation rules and validate the values in an *input form component* when the value might have changed, add *validators* to the components.

The value is validated against its validators on every validation run:

- when the validator is added;
- when the value might have changed, for example, the component lost focus, its `setValue()` method is called, `isValid()` is called, the component is rendered, etc.

To set a validator, call the `addValidator()` method on the component. The method takes a closure as its input argument:

- The input of the closure is the value in the input component: on a Text Box, the input is its text; On a Decimal Field, the input is the decimal value. Mind that this value must be in the correct format.
- The closure returns the error message that is displayed on the component. If the closure returns `null`, the component is valid.

If you want to validate only after some action takes place, name your input component and add the validator in the appropriate listener expression.

Example expression that adds a validator to a Date Field

```
MyDateField.addValidator (
  { dateValue:Date ->
    if
      dateValue > date(2018, 3, 3)
    then
         null
    else
        "The date must occur later than March 3, 2018."
    end
  }
)
```

You can get your validation messages and messages caused by incorrect input format with the `get↩ ValidatorsMessages()` or `getValidatorsMessagesRecursive()` methods.

7.2.2 Validating Record Constraints in Forms

When acquiring data for a record from a form, generally the input components in your form are **bound to a record or record field**: the binding is set to type *Reference* and its value to `patient.ssn`. For such form components, you can check whether the input values meet the record constraints and display the constraint messages, called data-error messages.

To validate the value and display data-error messages in a form, do one of the following:

- To *validate the value automatically*, call `inferValidator()`: The method adds a validator to a component, which keeps checking the constraints of the bound Record Property and displays any returned constraint messages on the component.

  ```
  c.inferValidator(null)
  ```

  > **Note:** The `inferValidator()` creates a validator object for the component: The input value is assigned to the validator object only if it is in the correct format: If the users enters a string into a Date Field, the string does not make it into the validator object. Therefore, if you call `inferValidator()` after the form component has been assigned a value, then any subsequent invalid values are ignored and the first value is always restored.

- To *validate at a particular moment*, call `validate()` and `showDataErrorMessages()`: The *validate()* call returns a list of data-error messages and *showDataErrorMessages()* erases the currently displayed data-error messages and then displays the received data-error messages on the subtree of the second argument.

  ```
  { e ->  showDataErrorMessages(validate(currentPersonVariable, null, null, null), null)}
  ```

  Example on-click validation of the newUnit record instance in a Listener expression:

  ```
  { click ->
    def List<ConstraintViolation> cv := validate(newUnit, null, null, null);
    if cv.isEmpty()
      then
        //merge without optimistic-lock check:
        myProxySet.merge(false, newUnit);
        vocabTable.refresh();
      else
       def List<String> messages := collect(cv, {cvi:ConstraintViolation -> cvi.message});
       //The showDataErrorMessages() function displays violation messages
       //on the subtree of the component defined by the second argument.
       //The validation messages are displayed on the components that have the Binding
       //to the data that produced the constraint violation:
       showDataErrorMessages(cv, MyButtonComponent);
    end
  }
  ```

To get the data error messages of a form component, use the `getDataErrorMessages()` or `getDataErrorMessagesRecursive()` methods.

To clear the data error messages, use the `clearConstraintViolations()` function.

More information on constraints is available in the Modeling Guide.

7.2.3 Checking Component Validity

To check if a component and all its children are valid, use call the `isValid()` method. The method searches for any validators, including inferred validators, that return a error message. If at least one error message is found, the method returns `false`.

Custom error messages are ignored by the `isValid()` method.

7.2.4 Setting a Custom Error Message

To have full control over when an value in an is validated, use the custom error message of components: With the `setCustomErrorMessage()` method, you can set and display one arbitrary error message on any component.

To get the error messages of a form component, call the `getCustomErrorMessage()` method and to clear the error messages, call `setCustomErrorMessage(null)` on your component.

Since error messages are not produced by validators or constraint mechanism, they do not represent a validation breach: consequently, the `isValid()` call *IGNORES error messages*. They are simply messages attached to components.

```
{ e ->
  def String val := text.getValue() as String;
  if val.isBlank() or val.length() < 3  then
    //setting error message on a component:
    text.setCustomErrorMessage("Must be at least 3 characters.");
  else
    Forms.submit()
  end
}
```

7.2.5 Getting All Error Messages

To collect the custom error, data error, and validator messages on a form component, use the `getAllError↩Messages()` or `getAllErrorMessagesRecursive()` methods.

7.3 Actions on Components

7.3.1 Refreshing a Component

When you change a value not related to the underlying data source in your form, such as a caption, table and column width, etc. the component in the client browser is refreshed instantly.

However, if you change the related data, such as the data source, you need to call the `refresh()` method on the component to have the component rendered with the modified data. The `refresh()` call causes refresh of all child components as well: the binding is reloaded, the data source is refreshed, and component validation triggered.

You do not need to call `refresh()` if you use the `setValue()` calls to modify the component data since it is called automatically.

7.3.2 Setting the Visibility of a Form Component

A visible component is included in the form: by default, all components apart form the Popup are visible. If you set a component as not visible, it is excluded from the form hierarchy.

- `setVisible(Boolean visible*)`: set the component visibility
- `isVisible()`: returns the component visibility

7.3.3 Setting the Caption of a Form Component

- `getCaption()` returns the component caption
- `setCaption()` sets the component caption The setCaption() call localizes any localization identifiers automatically; hence there is no need to include the `localize()` call.

Important: The caption of the Text Field is rendered by the parent layout component. Hence if you create a form with a sole Text Field component, the form will be rendered without its caption text. This is due to limits of the Vaadin framework.

7.3.4 Adding an Icon to a Form Component

All form components can display a resource as their icon. You can set an icon on a component in the following ways:

- If the icon is a resource, use the `setIcon()` call:

```
c.setIcon(FileResource.fromModule("form_components", "action/img/person.png"));
//For font-awesome icons, use the FontAwesome Resource record:
c.setIcon(new FontAwesome("minus-square"))
```

- If the icon should be added in a flexible way without being a resource, use the `setStyleName()` call or a custom rule in a theme:

```
c.setStyleName("fa fa-sun");
```

Using style names allows you to externalize the icon content and defined additional rules.

```
 //adding the style name of the icon rule:
anycomponent.addStyleName("myIcon far");
// or anycomponent.addStyleName("myIcon fas");
//add the style class with additional datasource
//on the given component:
c.addStyleSheet(
    ".myIcon:before {" + #10 +
    "   content: '\f556';" + #10 +
    "}" + #10
  );
```

It is not possible to set the icon size: use bitmap files with the correct size or adjust the style with your CSS files.

Important: The icon of a component is rendered by the parent layout component. Hence if you set an icon on the root component, the icon is not be rendered. This is due to limits of the Vaadin framework.

7.3.4.1 Inserting a Vaadin Icon as HTML

You can insert a Vaadin icon into the following components that support the HTML mode:

- into the component caption of Action Link, Button, Label, and Link components
- into the content of the Label component

You can do so in two ways:

- with the `vaadinIcon()` call
- using the FontAwesome5 record in HTML mode

Note that the component must support the HTML content mode or have an HTML Renderer (Grid), and the mode or renderer must be enabled.

The list of icon names is returned by `allVaadinIconNames()` call.

Example Label component with a Vaadin icon returned by vaadinIcon():

```
 { e ->
  label.setContentModel(ContentMode.Html)
  label.setValue("This is an abacus icon: " +
  vaadinIcon("abacus"));
}
```

Example Label component with a Vaadin icon as the FontAwesome5 record in HTML:

```
 c.setContentMode(ContentMode.Html);
c.setCaptionMode(CaptionMode.Html);
def FontAwesome5 sun := new FontAwesome5("sun");
c.setValue(sun.solid().spin().sizeLarge().toHtml());
c.setCaption(sun.solid().spin().sizeLarge().toHtml());
```

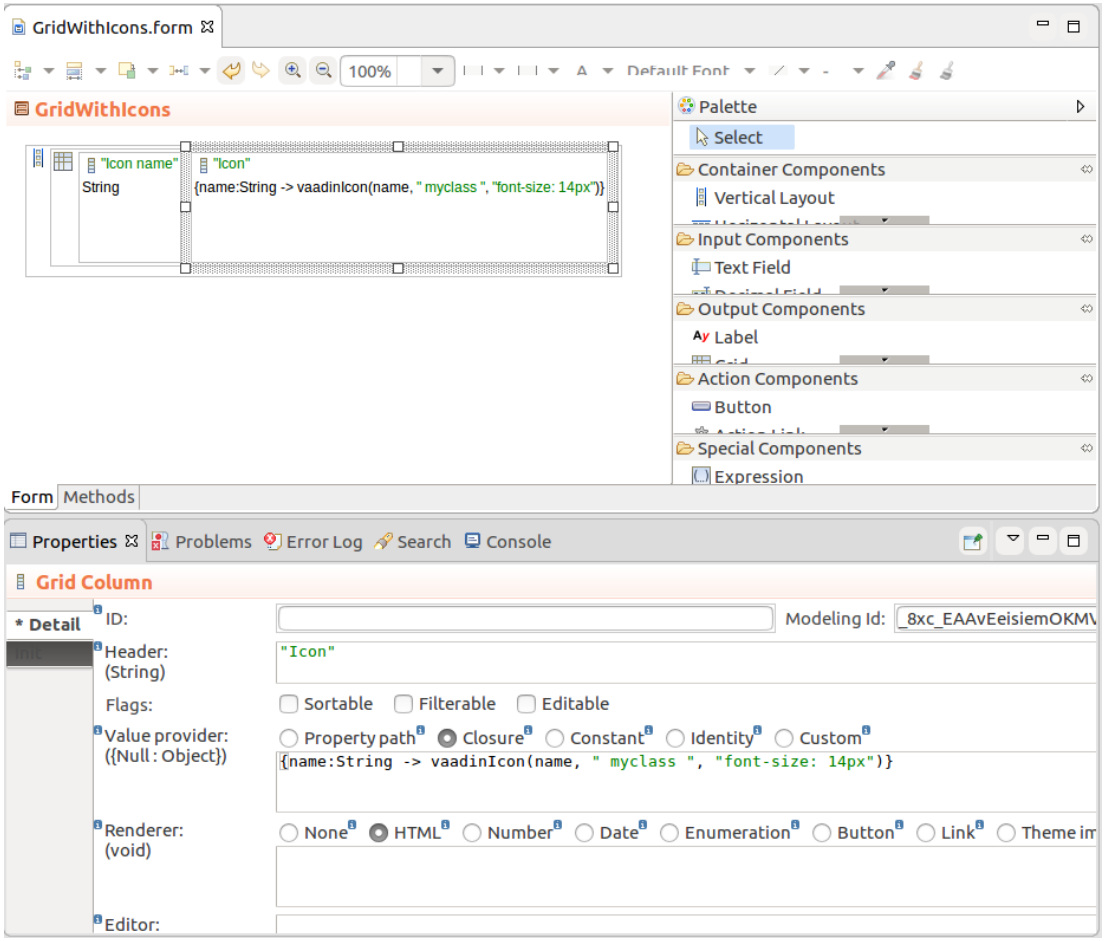


Figure 7.3 A Grid component with collection datasource allVaadinIconNames(); focused is the Column that renders the icons.

7.3.5 Setting a Component as Read-Only

When a component is in read-only mode, the user cannot change the value of the component. The mode does not influence other functionalities of components.

- setReadOnly(Boolean readOnly∗): sets a component as read-only or editable
- isReadOnly(): returns the current read-only status

7.3.6 Setting Component Size

Every component defines its default size settings (height and width) which you can change with the size-related methods `setWidth()`, `setHeight()`, and `setSize()`.

Size can be defined as follows:

- in percent of the size of the component slot: a component slot is space allocated to the component;

  Note: This space is restricted restricted by the parent layout component and can be adjusted with the Expand Ratio methods.

Consider two special values:

- `100%` referred to as *full* since it sets the component to fully fill its slot
- `null` referred to as *wrap* since it sets the component to take the minimum size required to accommodate its content.

- absolute size:
  - in pixels (px), picas (pc), points (pt), mm, cm, or inches (in)
  - font-relative units:
    - em: size in factors of the letter `m` size
    - ex: size in factors of the letter `x` size When you define absolute size of your component, make sure it does not overflow its component slot if this is not required. Such situations can lead to overlapping content.

7.3.6.1 Default Component Sizes

All components have their default sizes used if you do not specify their size explicitly:

- Width is set to *100%* or *full*: the width is set to the width of the parent component (identical to `100%`)
  The exception are all Input Components, Horizontal Layout, and Popup which wrap their content by default.
- Height is set to `null` or *wrap*.
  The only exception is the Map component that has a fixed height of 400px.

7.3.6.2 Setting Absolute Size

When setting component size, use the setter methods `setWidth()` and `setHeight()`: the methods take parameters in the format [number][units], such as 100%, 10em, 25px.

You can get the current size with the `getWidth()` and `getHeight()` methods.

7.3.6.3 Wrapping Content

To make a component wrap its content, call the respective setter method with the `null` parameter, for example, `c.setWidth(null)` or one of the `wrap` methods:

- setWidthWrap() or setWidth(null)
- setHeightWrap() or setHeight(null)

Note that if the size is set to *wrap*, it is considered *null*.

7.3.6.4 Filling Parent

To set the size of a component to fill its component slot, call a *full* method on the component.

- setWidthFull() or setWidth("100%")
- setHeightFull() or setHeight("100%")
- setSizeFull() combines both calls above

7.3.6.5 Setting Expand Ratio

Expand ratio sets the size of the component slot in proportion to the parent component size. Mind it does not set the size of the component itself.

Layout components allocate to their children *component slots* of equal sizes; the child components fill the slot as defined by their properties:

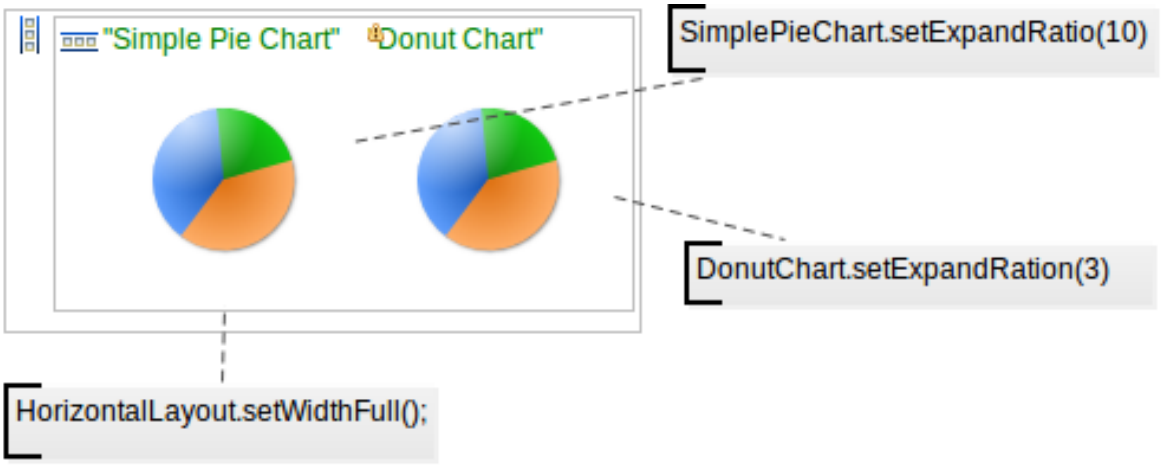
- They can *fill* the slot or a percentage of the slot.
- They *wrap* their content and the slot adapts to this minimal size.
- They use their absolute size.

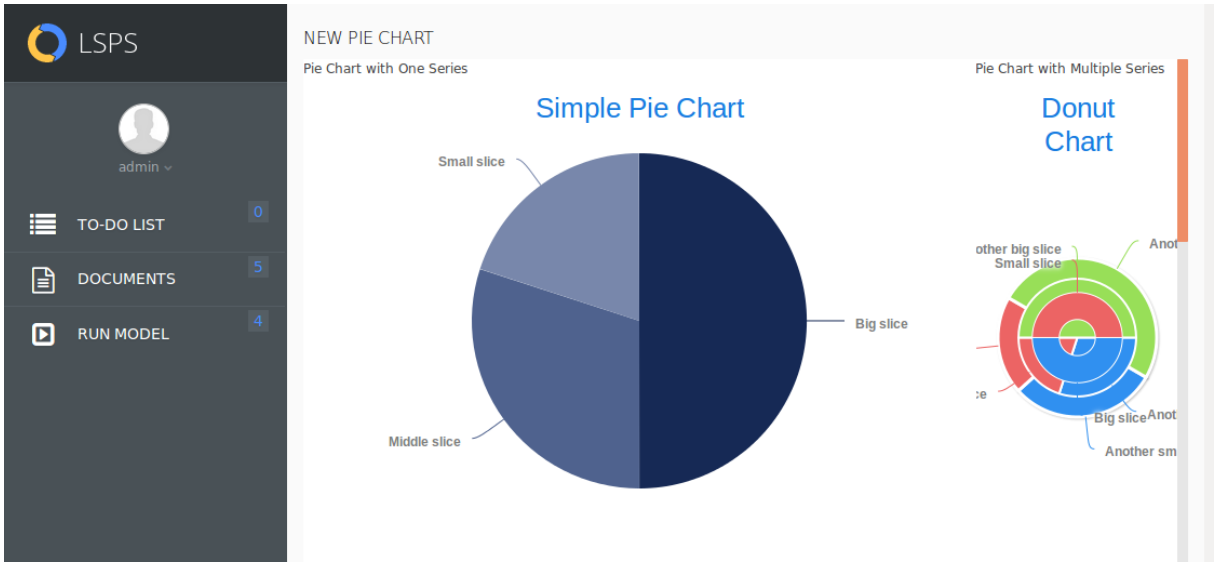
  Important: When you set absolute size on a component, note that this does not influence the size of the component slot and might result in content overlap.

The ratio expand applies only to child elements of **Horizontal**, **Vertical**, and **Grid** layouts since only these components allow you to change the slot size:

When the Layout component is calculating the component slot sizes, first it gives the required space to components that wrap their content (the slot adopts the size of the component); then it calculates the size of the remaining available space and distributes this space either equally or according to the defined expand ratio numbers.

- setExpandRatio(Decimal ratio)
- getExpandRatio()





7.3.6.6 Setting Alignment

You can align the component with its slot: the alignment impact depends on the size of the component: if the Component is as wide as its slot, bottom and top alignment will have no effect.

- setComponentAlignment(Alignment alignment∗)
- getComponentAlignment()

7.3.7 Add Style Name to a Form Component

> **Note:** You can only add style names to form components from a GO-BPMN model: To create custom SCSS style rules, `implement them in the style sheets of the Application User Interface`.

To add a style name, that is the style name to the class attribute, of a component, use one of the following methods:

- `addStyleName(selector1, selector2)` adds the selectors to the class attribute to the component. This call does not modify the classes the component already has. The call triggers the *RepaintRequestEvent* on the component which you can handle if necessary.

  By default the following selectors are available: `l-border-none`, `l-border-left`, `l-border-top`, `l-border-right`, `l-border-bottom`, `l-highlighted`, `l-emphasized`, `l-grayed`, `l-border`.

  > **Note:** For all your classes, under the hood, the system creates their copy with the name `v-<component_type>-<class_name>` so you can style the component either in the class `<class_name>` or `v-<component_type>-<class_name>` in your SCSS file.

- `setStyleName(selector1, selector2)` replaces the class attributes of a component with the values passed in the parameters.

You can work with style name using the following methods:

- `getStyleName()` returns the class attributes of the component.

  The method returns only the attributes that you added with the `setStyleName()` method: classes set by the system are ignored.

- `removeStyleName()` removes the class values passed in the parameters.

  You can remove only the class attributes that you created with the `setStyleName()` and `removeStyleName()` calls: classes set by the system cannot be removed.

For information on how to add your style and style rules, refer to Add Style Name to a Form Component.

7.3.8 Disabling a Component

A disabled component has the `disabled` style. Disabled components cannot be modified or updated in any way.

- isEnabled(): retuns the current enabled status
- setEnabled(Boolean enabled∗): sets the component to disabled or enabled

7.3.9 Setting a Tooltip

To display a tooltip, when the user hovers over a component, set the description of your component with the `setDescription()` method.

> **Important:** The description is handled as HTML; hence make sure no injection and XSS vulnerabilities can be exploited.

```
myLabel.setDescription({ r:MyRecord ->
            "Status of row object: " + m.status + #10 + m.id});
```

7.3.10 Displaying a Notification Dialog

To display a notification, use the `notify()` function.

The displayed notification can be of type:

- *Info*: immediately fades out with no delay
- *Warning*: disappears after 1.5 second
- *Error*: requires user click to disappear
- *Tray*: immediately fades out and is displayed in the bottom-right corner

```
MyTextField.setOnChangeListener(
  {e->
    notify(
      caption -> "Notification on Value Change", "" + MyTextField.getValue(),
      ntype -> NotificationType.Info)
  }
);
```

7.3.11 Defining a Context Menu

Each form component can define its context menu which is displayed when the user right-clicks the component or when the `showContextMenu()` method is called.

The menu contains a list of menu items, which trigger an action when clicked: the action is defined by their `onClick` parameter.

To define a context menu on a component, use its `setContextMenuItems()` method.

Example context menu with one item

```
c.setContextMenuItems(
  [
    new forms::MenuItem(
       caption -> "Show Details",
       onClick -> { ->
         showPatDetail(somePatient).setVisible(true);
       }
    )
  ]
)
```

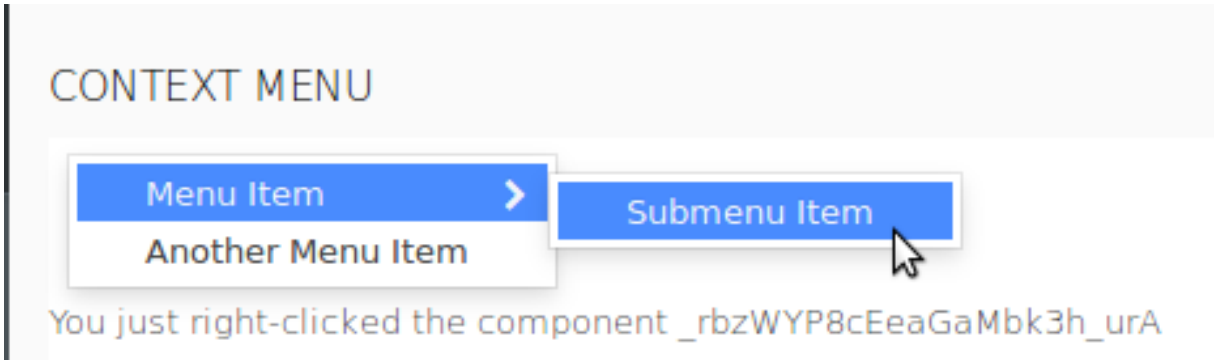
Example of displaying the context menu programatically

```
{ e:ClickEvent ->
  //the method has Integer parameters with x and y
  //coordinates of the context menu (0,0 is the upper left corner):
  somePatientText.showContextMenu( e.x, e.y)
}
```

Menu items in the context menu can have child menu items. Note that the *onClick* closure on a menu item with a submenu is ignored (only clicks on leaf menu items can be handled).

Example context menu

```
c.setContextMenuItems(
  [
    new forms::MenuItem(
       //This onClick closure is ignored
       //since the item has a submenu:
       onClick -> { ->
         c.setValue("You just clicked the Menu Item option");
         c.refresh();},
       subMenu -> [
         new forms::MenuItem(
           onClick -> { ->
             MyText.setValue("You just clicked the Submenu Item option");
             MyText.refresh()},
           caption -> "Submenu Item")],
           caption -> "Menu Item"),
         new forms::MenuItem(
           onClick -> { ->
             MyText.setValue("You just clicked the Menu Item option");
             MyText.refresh()},
           caption -> "Another Menu Item"
    )
  ]
)
```



7.3.12 Handling a Right-Click Event

To handle a right-click on a component and possibly build a context menu dynamically, use the `setContext↩ ClickHandler()` method.

```
c.setContextClickHandler(
  { x:ContextClickEvent ->
     eventText.setValue("You just right-clicked component " + x.source.modelingId);
  }
)
```

7.3.13 Defining a Value Provider

Value provider defines the content of the cell in the row. Its value is based on individual object return gradually from the Data Source; it is used to define what value a column of a Table, Tree Table, Grid or Combo components should display.

> **Important:** Value Provider must return values of a simple data type: If it returned Objects of another type, it would not be possible to filter and sort them.

You can define the following Value Provider types:

- *Property path*: path to a Property of the data source Record

```
Person.name
```

- *Closure*: a closure that returns the value for the cell with the Value Provider object as its input

  The Closure Value Provider is typically used for calculated fields (note that when calculating fields, you will need to refresh the entire Grid when one of the relevant values changes).

```
{ person -> person.firstname + person.surname}
```

- *Constant*: a constant value (for example, caption for an Edit Button; the on-click action is then defined in the Renderer)

- *Identity*: the row object type

```
Integer
```

- *Custom*: implementation of your custom value provider

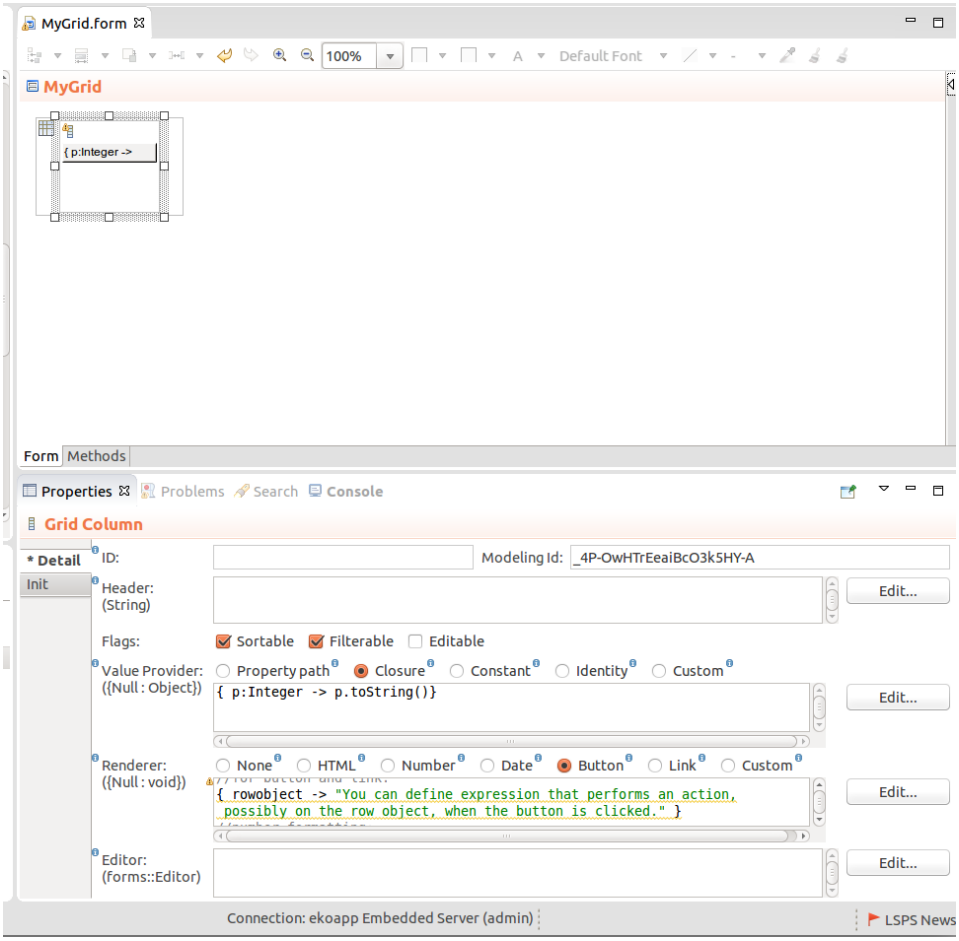


Figure 7.4 Form with a Grid Column: the Value Provider casts the Row object to a String; the String is then used as the label of the Button; On click, the Closure in the Renderer property is executed.

Chapter 8

Component Reference

8.1 Container Components

Container components can define one or multiple child components serve to organize the content.

8.1.1 Common Properties and Functionalities

On all container components, you can get do the following:

- get information about its child components:
  - the number of child components with the `getComponentCount()` method

    ```
    myLayout.getComponentCount();
    ```

  - the list of child components with the `getComponents()` method The method returns a list of components: the list items are returned as defined in their `toString()` method;

    ```
    myLayout.getComponents();
    ```

    When calling the `getComponents()` method, you will need to do so after the form was initialized: to do so, call the method in the `postCreateWidget()` method of the form. The call is pre-generated in the methods file of your form.

8.1.1.1 Setting Spacing and Margin

All container components can define their style with the following methods:

- setMargin(Boolean margin_enabled): sets a default margin of the container component

  To change the margin size, use the `properties file of the theme in the generated LSPS Application`.

- setSpacing(Boolean spacing_enabled): sets a default spacing between the child components

  It is not recommended to customize spacing since it might result in a inconsistent UI presentation; consider changing font size to meet your requirements.

8.1.1.2 Populating Container Components

You can populate your container component as follows:

- using a parametric constructor with the components as parameters, for example, `new FormLayout(textField, submitButton, navigateButton)`
- using the `addComponent` method of the component record, for example, `layout.addComponent(textField)`

8.1.2 Container Component Reference

8.1.2.1 Layout Components

Layout components can have multiple child components: by default the child components are allocated equal space within the layout, so if you have 3 child components in a layout, each will be allocated 33% of the available space. This space is referred to as the *component slot*. To change the size of component slots, use Expand Ratio on the child components. You can then call methods on the child elements to define their size in their slot.

8.1.2.1.1 Horizontal Layout (forms::HorizontalLayout)

The *Horizontal Layout* component ( ) arranges its children horizontally.

Mind that by default the layout wraps its components, so if the components overflow the layout and they are set to fill the parent, the overflowing content will not be visible. To remedy this situation, call `setWidthFull` on the layout component.

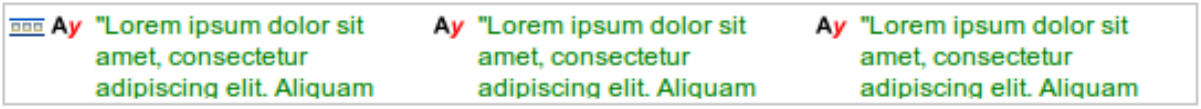


Figure 8.1 Form with multiple Horizontal Layouts

Expression that returns a Horizontal Layout with child components

```
def forms::HorizontalLayout hl := new forms::HorizontalLayout();
hl.addComponents(
    {
     new forms::Button("Confirm", {e -> Forms.submit()}),
     new forms::Button("Cancel", {e -> Forms.navigateTo(new AppNavigation(code -> "todoList"))})
    }
);
hl
```

8.1.2.1.2 Vertical Layout (forms::VerticalLayout)

The *Vertical Layout* component ( ) arranges its child components vertically.

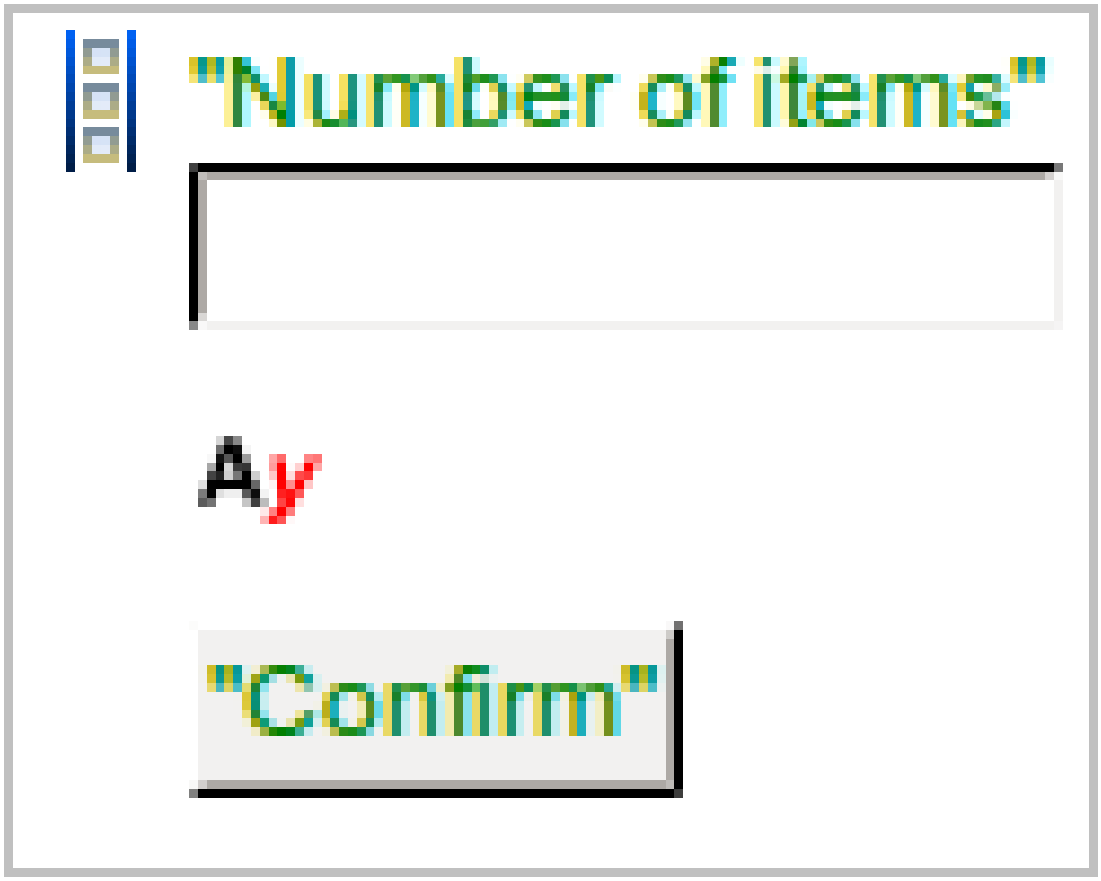


Figure 8.2 Form with Vertical Layouts

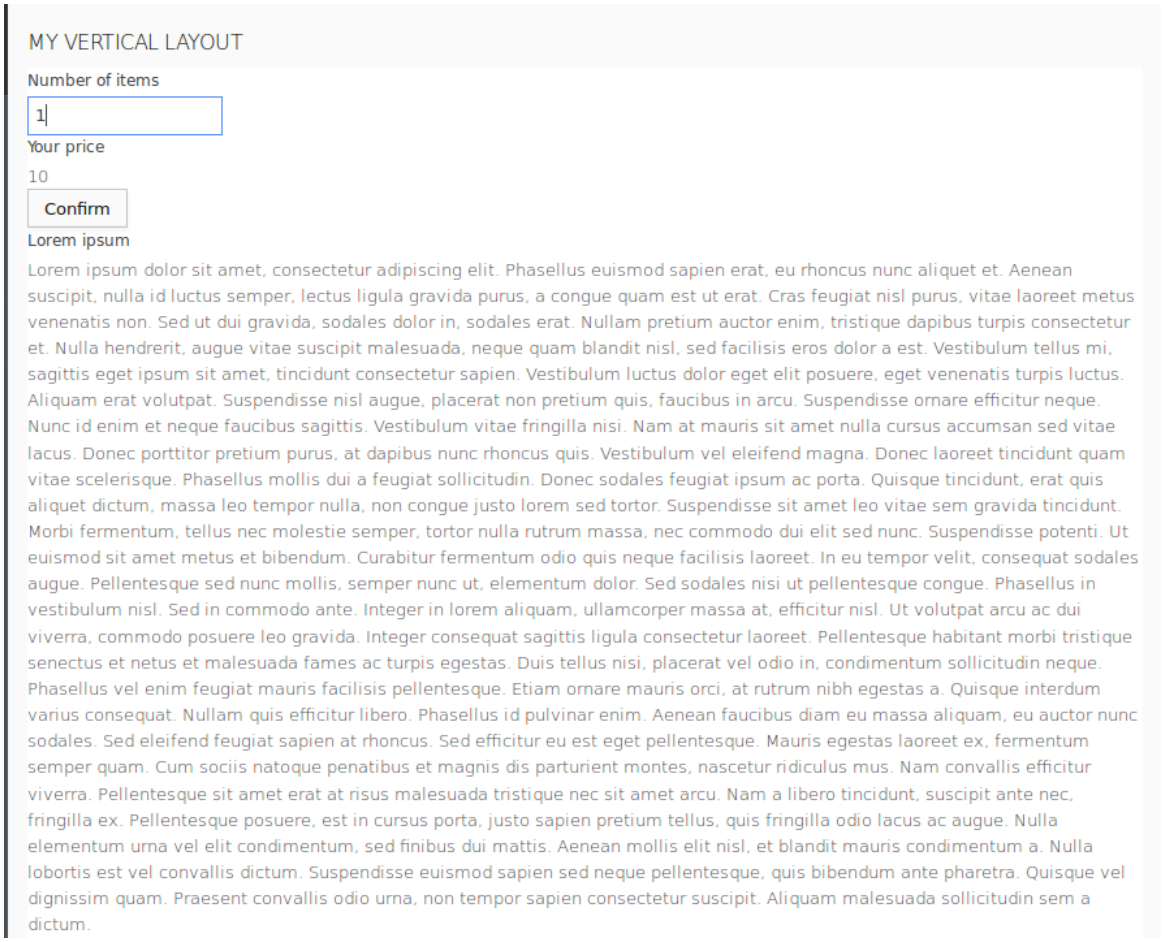


Figure 8.3 Rendered form with Vertical Layouts

```
def forms::VerticalLayout vl := new forms::VerticalLayout();
def forms::Table t := new forms::Table();
t.setDataSource(new TypeDataSource(User));
~
t.addColumn(new forms::TableColumn
      (valueProvider -> new PropertyPathValueProvider(User.name),
      sortable -> true, filtrable -> true,
      generator -> { username:String -> new Label(username)},
      modelingId -> "87jlkjh",
      data -> null));
t.addColumn(new forms::TableColumn
      (valueProvider -> new PropertyPathValueProvider(User.note),
      sortable -> true, filtrable -> true,
      generator -> { note:Object -> new Label(note)},
      modelingId -> "87j98098jh",
      data -> null));
~
vl.addComponents([t, new forms::HorizontalLayout(new forms::Button("Create", { click -> openMyPopu
vl;
```

8.1.2.1.3 Grid Layout (forms::GridLayout)

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 Layout component.

Important: We recommend to use Grid Layout only if it is not possible to solve the layouting of your form with the Horizontal and Vertical Layouts easily. Also, make sure not to use nested Grid Layouts since this may cause performance issues.

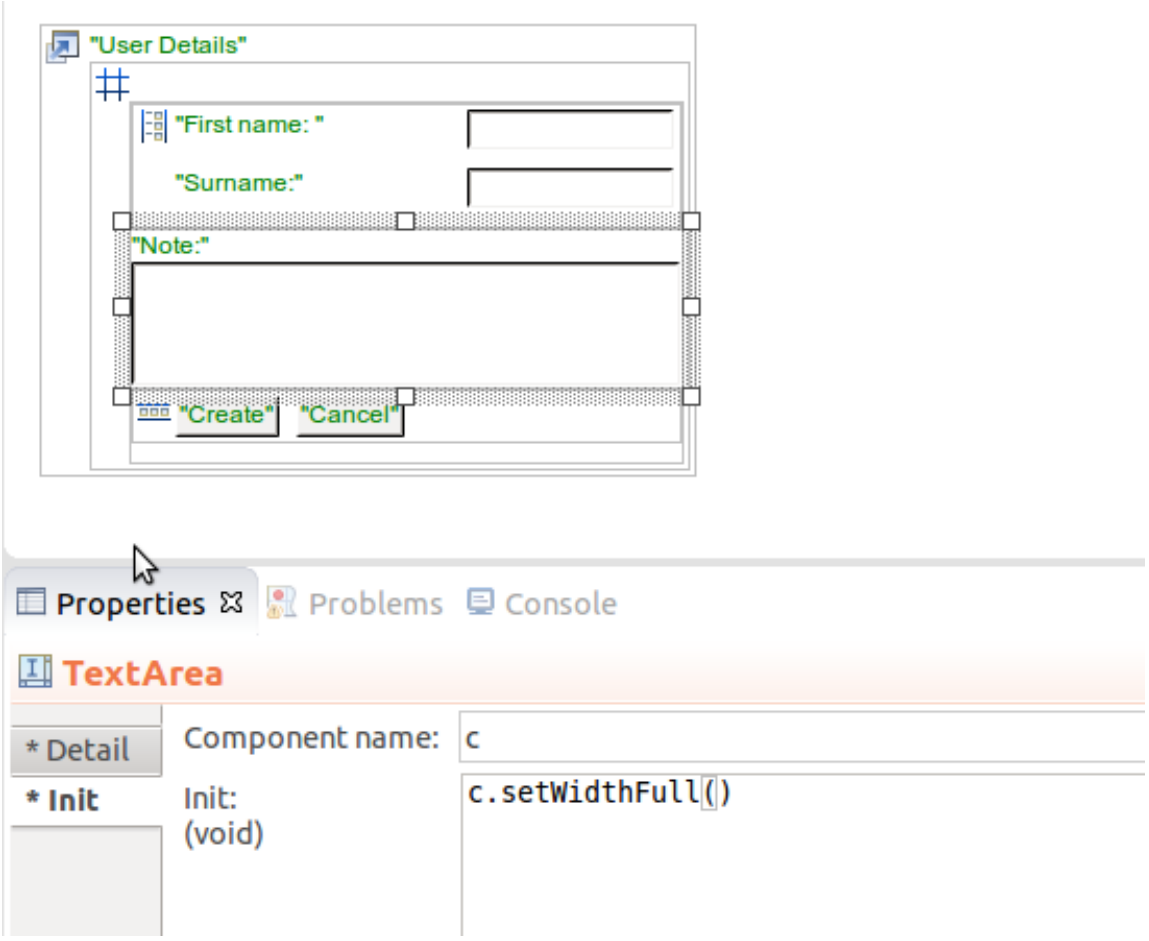


Figure 8.4 Form with Grid Layout: Note that the Popup has the width set to a value and the Text Area has its width set to full. The Text Area will always adapt to fill the Grid Layout while the input fields are implicitly set to wrap their content and will retain the size regardless of the Popup size.

Expression that returns a Grid Layout component

```
def GridLayout gl := new GridLayout();
gl.addComponentToGrid(new Label("row1,column1"), 1, 1);
gl.addComponentToGrid(new Label("row2,column1"), 2, 1);
gl.addComponentToGrid(new Label("row1,column2"), 1, 2);
gl.addComponentToGrid(new Label("row2,column2"), 2, 2);
gl.setRowExpandRatio(0,7);
gl.setColumnExpandRatio(2,2);
gl
```

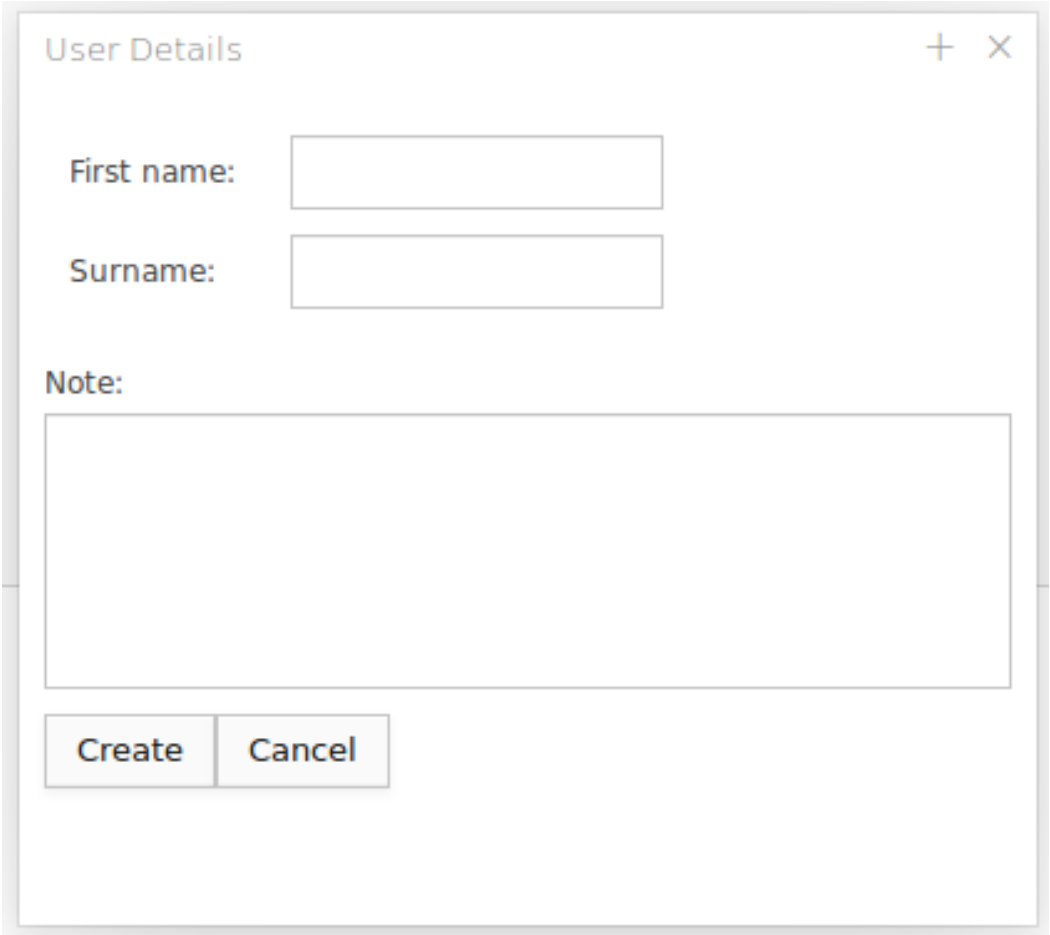


Figure 8.5 Form with Grid Layout

8.1.2.1.4 CSS Layout (forms::CssLayout)

The **CSS Layout** component () is rendered as a `<div>` element that wraps the inserted child components. To set its style, use the `setStyleName()` method.

```
def CssLayout cl := new CssLayout();
cl.addComponent(new TextField("Name:", &name))
```

8.1.2.1.5 Form Layout (forms::FormLayout)

The *Form Layout* component () arranges child components horizontally with their captions displayed on the left.

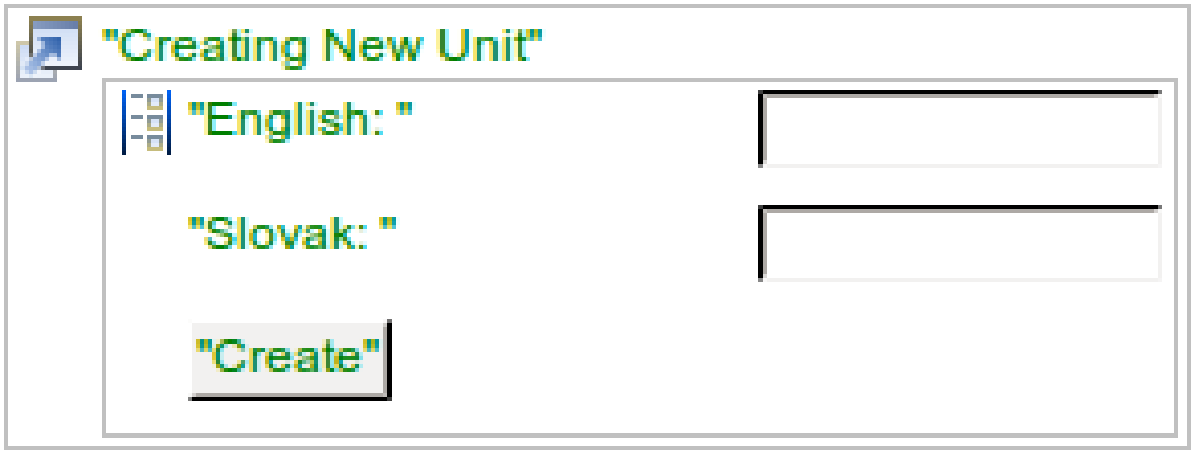


Figure 8.6 Form with Form Layouts

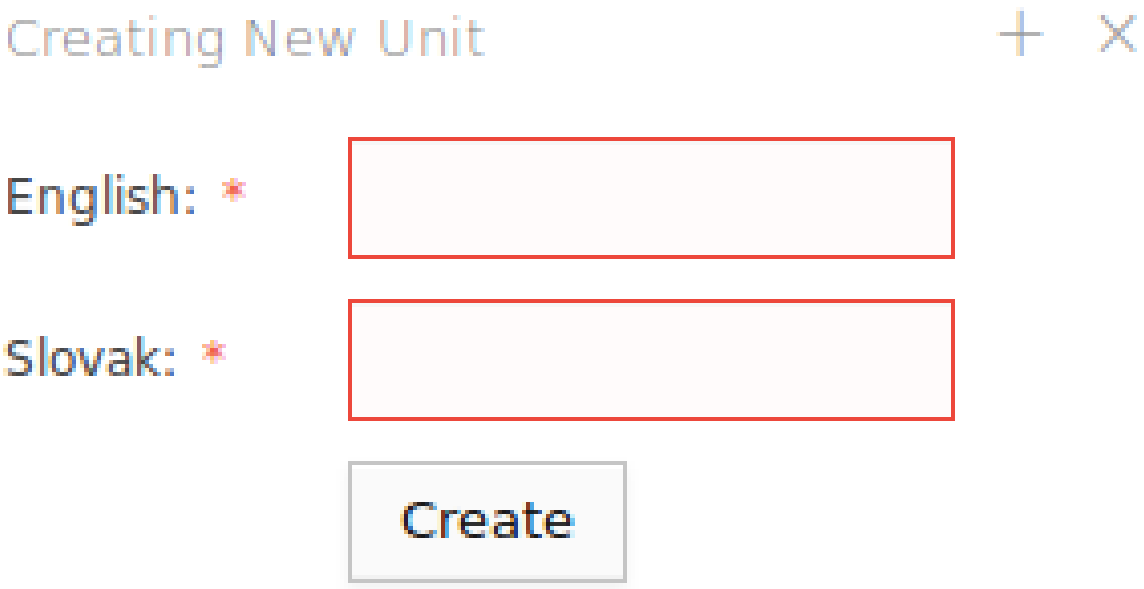


Figure 8.7 Form with Form Layouts

Expression that returns a Form Layout with child components

```
def forms::FormLayout fl :=
    new forms::FormLayout(
            new TextField("Name:", &name),
            new Label("Please confirm:"),
            new forms::CheckBox("Do you agree with the conditions as stated in the consumer agreem
```

8.1.2.1.6 Panel (forms::Panel)

The *Panel* component ( ) adds a label and scroll to its child component automatically if applicable.

Note that to have the scrollbar displayed, the following must be true:

- The child component must not define its size as *100%* or *full* (There is no sense in scrolling a child that fills its parent fully since there is nothing to scroll).

- The Panel must not be set to wrap its content: this is the default size setting of the Panel component.

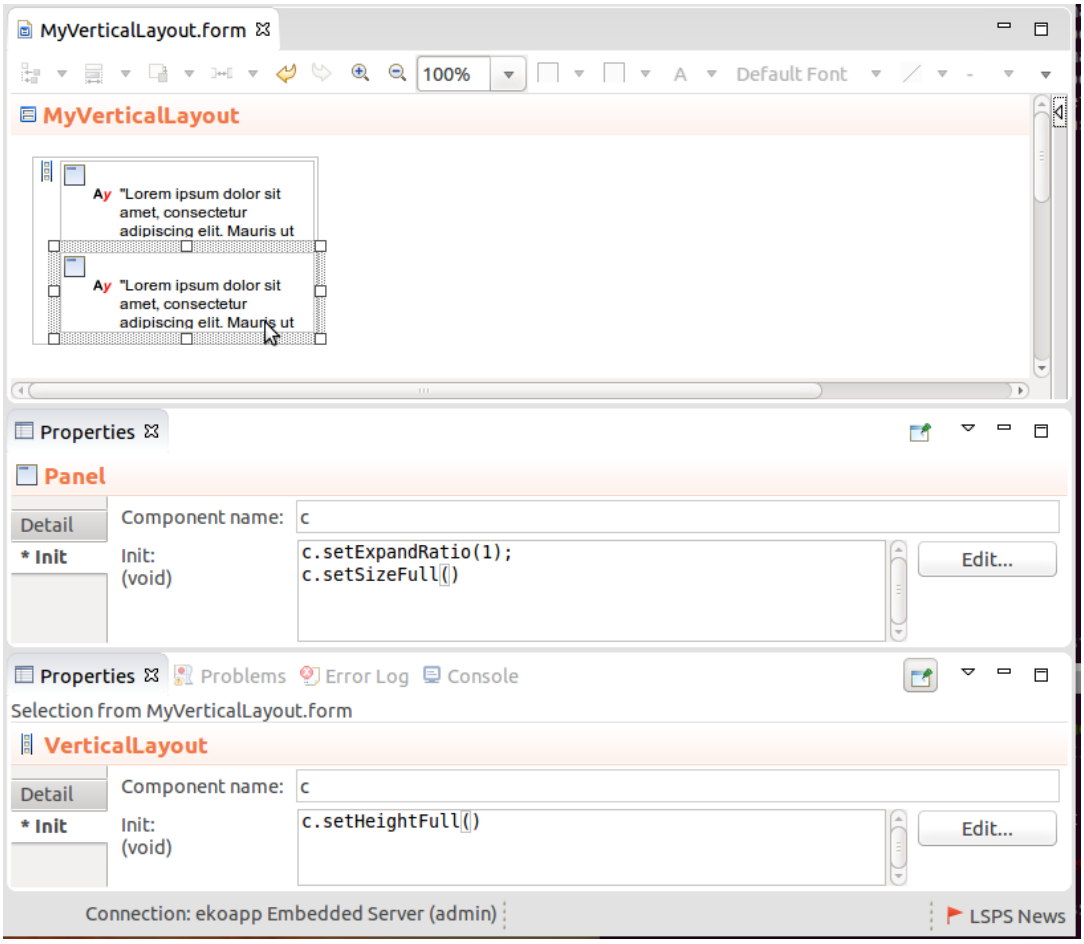


Figure 8.8 Vertical Layout has the size of the page and Panels fill the layout. The Labels wrap their content by default; hence the Label has its size that depends on its content; its Panel has the size determined by the Panel and Vertical Layout

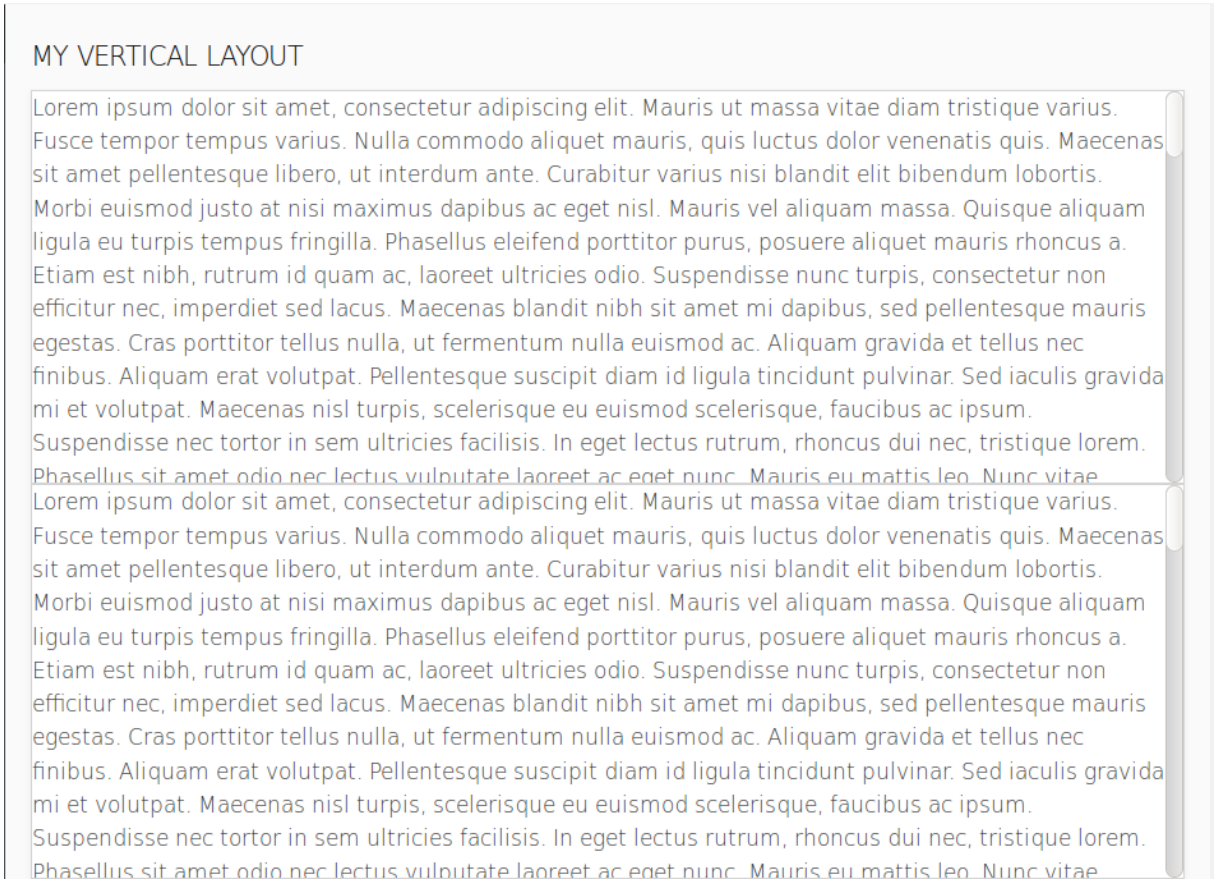


Figure 8.9 Form with Panels with scrollbars

Expression that returns a Panel component

```
def forms::Panel p := new forms::Panel("Lorem Ipsum", new Label(lorem));
p.setSizeFull();
p
```

8.1.2.1.6.1 Setting Scrollbar Position

To set the scrollbar position on a Panel component, call the `scrollToTop()` or `scrollToBottom()` method.

8.1.2.1.7 Collapsible

The *Collapsible* component () allows the user to collapse and expand its content.

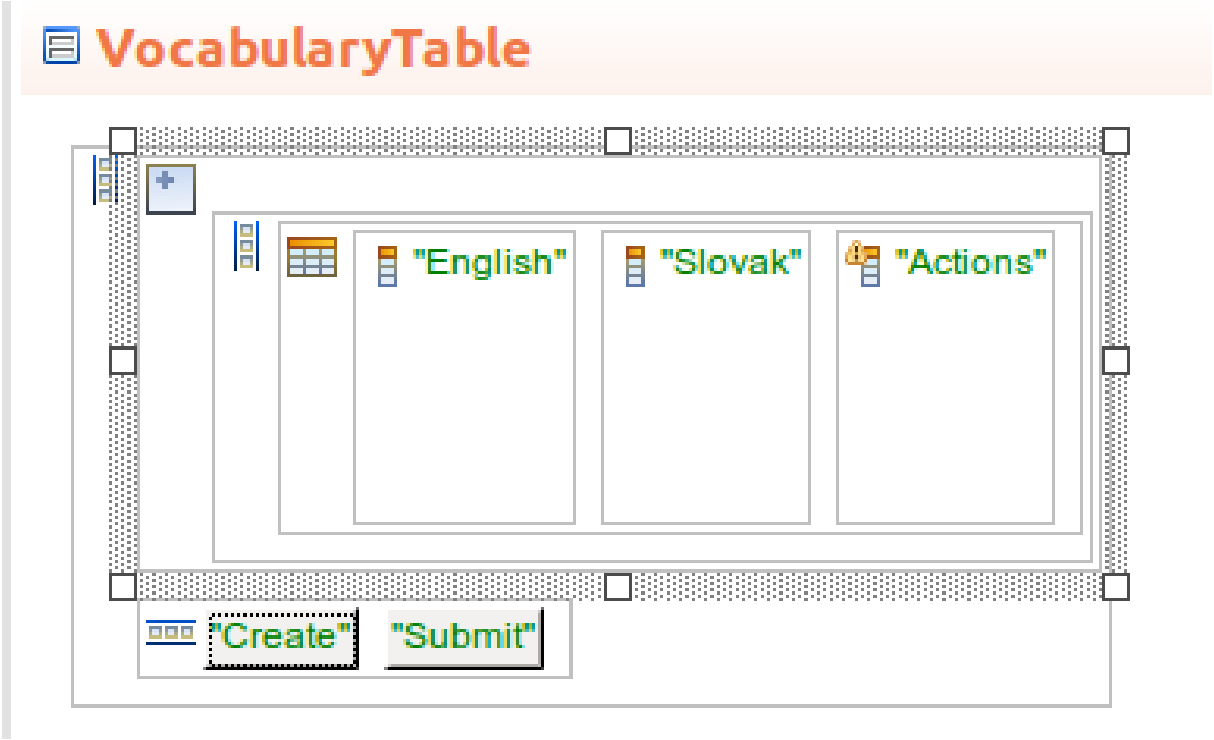


Figure 8.10 Form with a Collapsible component

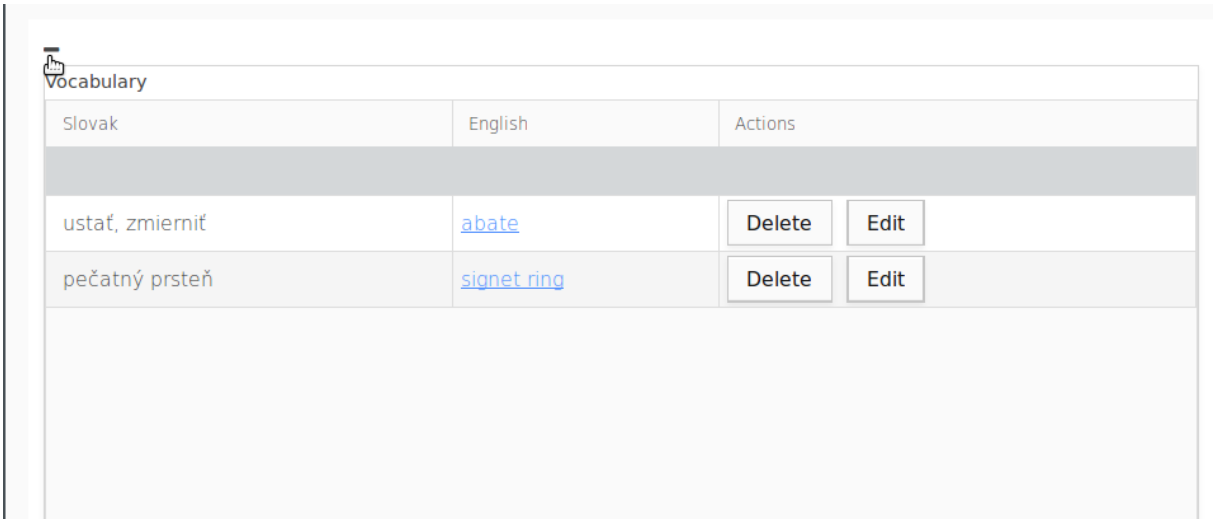


Figure 8.11 Form with a Collapsible rendered

Expression that returns a Collapsible component

```
def forms::Collapsible p := new forms::Collapsible("Lorem Ipsum", new Label(lorem));
p.setSizeFull();
p
```

8.1.2.2 Other

8.1.2.2.1 Tab Sheet (forms::TabSheet)

The *Tab Sheet* component () holds only *Tab* components, which can then contain further content.

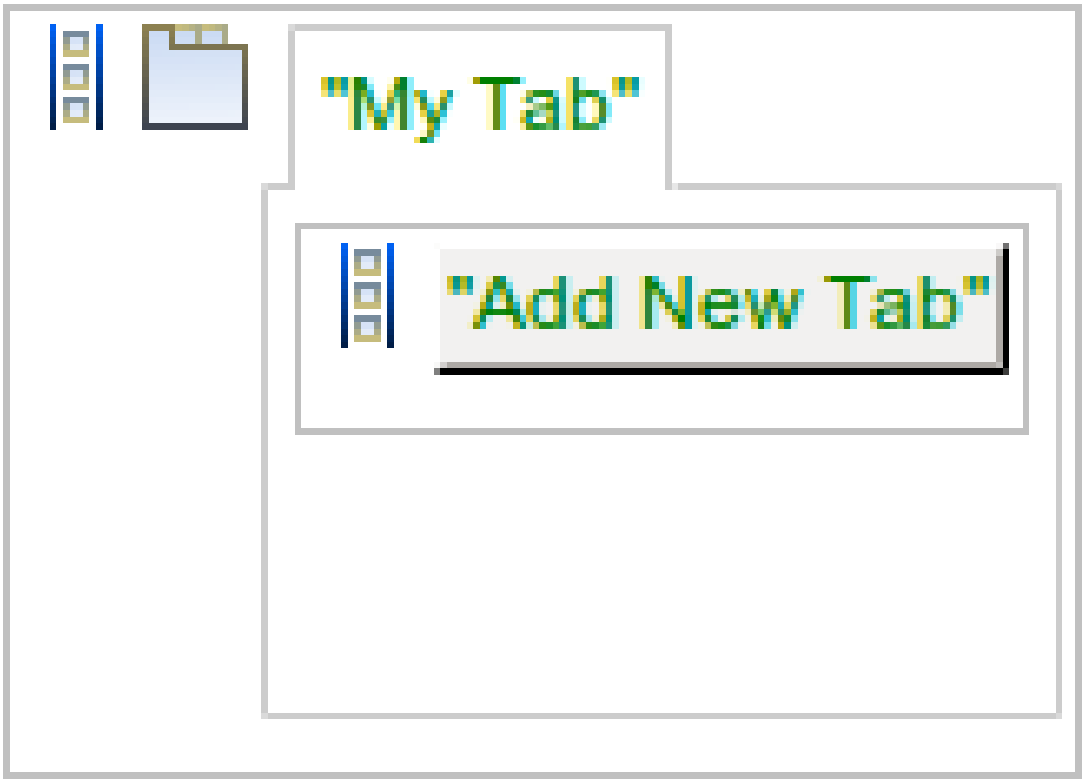


Figure 8.12 Form with Tab Sheet and Tab components

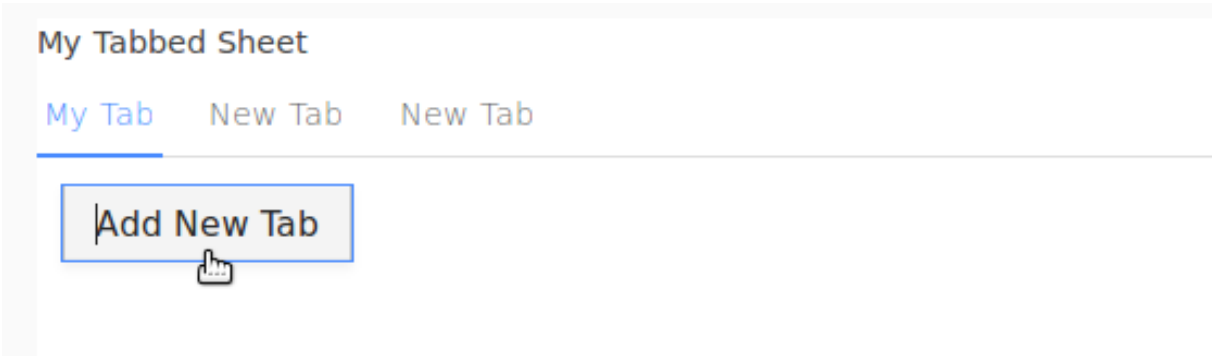


Figure 8.13 Tab Sheet component with Tabs

Related methods:

- getTabs(): returns the list of tabs of the Tab Sheet

- addTab(TabContent, TabName): adds a tab with the TabName and TabContent
- getSelectedTab(): returns the currently selected tab
- getTab(TabId): returns the Tab with the ID
- removeTab(Tab): removes the Tab

Expression that returns a tab sheet with one tab

```
def TabSheet ts := new TabSheet();
  ts.addTab(new Label("Label in a tab"), "tab name");
  //get the content of the first tab(Label):
  def Component tabRoot := ts.getTabs()[0].getContents();
  ...
ts
```

8.1.2.2.1.1 Adding a Tab

To add a new tab to a tab sheet, call the *addTab()* method of the tab sheet: the arguments represent the content of the tab, and that its caption and its component).

```
{ e -> MyTabSheet.addTab(new Label("Here is your new tab!"), "New Tab")}
```

8.1.2.2.1.2 Enabling Tab Closing

Tabs cannot be closed by default: to add the Close button to a Tab, call `setClosable(true)` on the Tab component.

8.1.2.2.1.3 Focusing a Tab

To select a particular Tab in a Tab Sheet, use the `setSelectedTab(<TAB_INDEX>)` call on the sheet: To get the index of a tab, call `getIndex()` on the Tab.

When you select a Tab in the client, the system throws a *TabChangeEvent*. You can listen for the event on the parent Tab Sheet: call the *setTabChangedListener()* method with the appropriate handling on the component.

Note that if you want to get the Tab component selected when the Form is loaded, you need to do so in the `post↩CreateWidget()` method:

```
MyForm {
~
  private void postCreateWidget() {
      setResultContent();
  }
~
    private forms::Tab getCurrentTab(){
       myTabbedSheet.getSelectedTab();
    }
~
    private void setResultContent(){
       resultTextField.setValue("You have selected the " + getCurrentTab() + " tab.");
    }
}
```

8.1.2.2.2 Accordion (forms::Accordion)

The *Accordion* component is just like Tab Sheet but with Tabs arranged vertically.

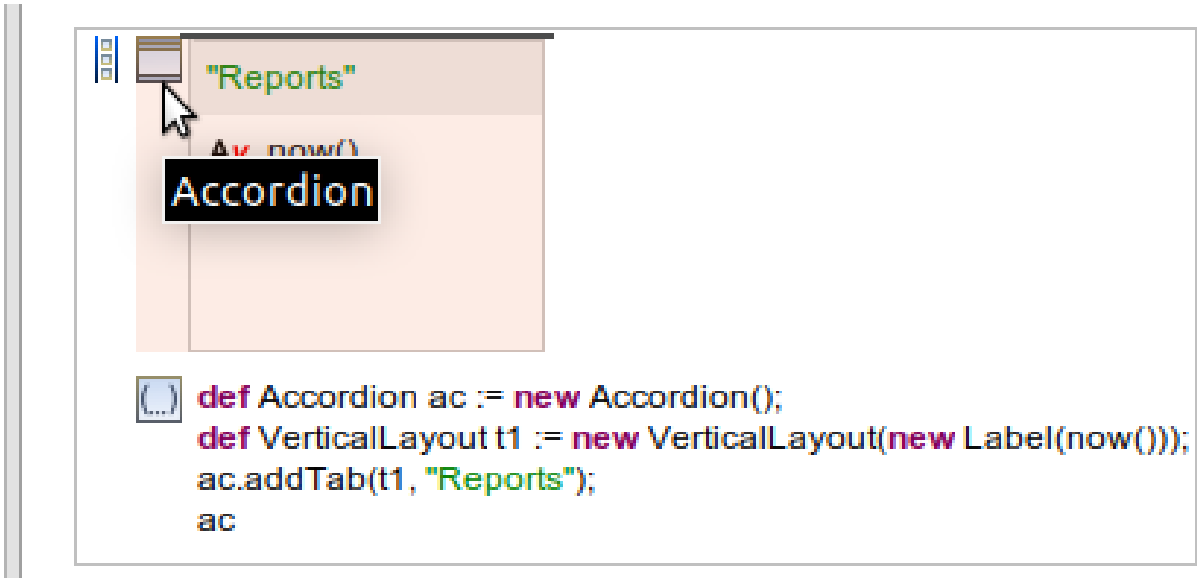


Figure 8.14 Inserting a Tab into an Accordion

Expression that returns an Accordion with one Tab

```
def Accordion ac := new Accordion();
def VerticalLayout t1 := new VerticalLayout(new Label(now()));
ac.addTab(t1, "Reports");
ac
```

Figure 8.15 Rendered Accordion

8.1.2.2.3 Dashboard (forms::GSDashboard)

The *Dashboard* component ( ) is rendered as the space for Dashboard Widget components. The widgets can be dragged-and-dropped, resized, minimized, and maximized. Whenever the user performs an action on a Widget, the system throws a *DashboardUpdateEvent*: the event contains information about the change and is processed as defined in the *onWidgetUpdate* property of the Dashboard.

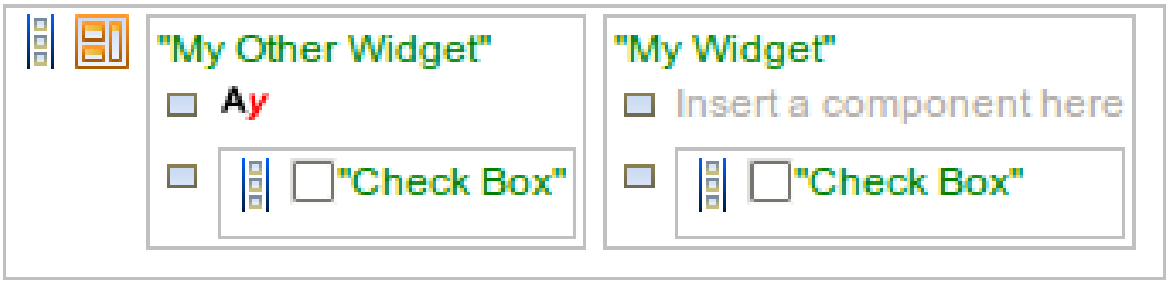


Figure 8.16 Form with a Dashboard component with Dashboard Widgets

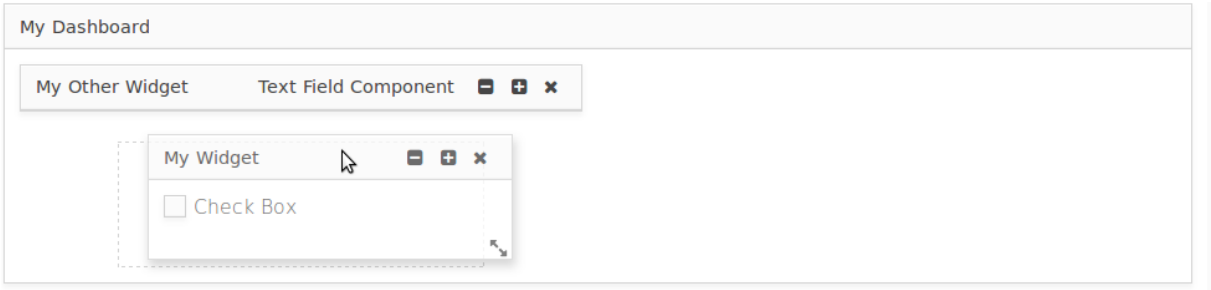


Figure 8.17 Form with a Dashboard component with Dashboard Widgets rendered in the Application User Interface (Widget being moved)

> **Note:** By default, the Dashboard width is set to fill its parent, while the height adapts to the space required by its widgets (the dashboard height wraps the content dynamically).

8.1.2.2.4 Dashboard Widget (forms::GSDashboardWidget)

The *Dashboard Widget* component ( ) is a draggable, resizable component of a Dashboard: In the Form, it must be the child of a Dashboard component.

The Widget position are defined relatively to the Dashboard area: The Dashboard area is divided in implicit 12 "columns" and an unrestricted number of rows with a set height. When you define the position, you define in respect to the columns and rows with *x* referencing one of the Dashboard columns and *y* referencing a Dashboard row.

The Widget size is also defined with respect to the Dashboard columns and rows: *width* is defined as the number of column the widget spans over and analogously, *height* the number of rows.

If you don't define the size or position of a Widget, it is calculated automatically. You can disable this by setting the *Autoposition* property to `false`. When Autoposition is set to *true*, the *x* and *y* coordinates are ignored. By default, `true` is assumed. When Autoposition is set to *true*, the *x* and *y* coordinates are ignored.

8.1.2.2.4.1 Adding a Dashboard Widget

You can add a Widget to a Dashboard in the following ways:

- To add to a Dashboard a widget which is displayed when the Dashboard is loaded, insert it into your Form from the palette in the Form Editor.
- To add a Dashboard Widget dynamically when an even occurs, use the `addWidget()` method of the Dashboard.

Adding a Dashboard Widget to a Dashboard on an event reception

```
myDashboard.addWidget(
    new Label("This is the content of my widget"),
    new Rect(x -> 2, y -> 2, width -> 3, height -> 4),
    true);
```

Apart from its content and generic component properties, a Widget can define the content of its toolbar, which is displayed next to its caption:

```
def GSDashboardWidget mw := MyDashboard.addWidget(
    new Label("This is the content of my widget"),
    new Rect(x -> 2, y -> 2, width -> 3, height -> 4),
    true);
    mw.setCaption("Caption");
    mw.setToolbar(new Label("Toolbar label"));
```

8.1.2.2.4.2 Removing a Dashboard Widget

To remove a widget from your Dashboard, call the *removeWidget()* method of the Dashboard.

8.1.2.2.4.3 Handling Changes on Dashboard Widgets

When the user changes the position of a Widget, resizes it, or closes it, the system throws a *DashboardUpdate↩ Event*: The event holds information on the Widget that caused the event, the new position of the widget, and the new state of the widget (`Minimized`, `Restored` in its widget size, `Maximized`, or `Closed`).

You can define how the Dashboard processes such events in its *onWidgetUpdate()* property: The property is of the Closure type with the event as its input. Define handling of the event in its body.

Example of handling a change of widget state

```
c.onWidgetUpdate := {
    e:DashboardUpdateEvent->
    if e.widget == myWidget then
       switch e.newState
         case WidgetState.Minimized -> myWidget.setCaption("Minimized");
         case WidgetState.Restored -> myWidget.setCaption("Restored");
       end;
    end;
}
```

8.1.2.2.4.4 Restoring Closed Dashboard Widgets

When you close a widget, the widget is discarded: if you want to re-display it, you need re-create it. Since when a widget is closed, the system throws a DashboardUpdateEvent with the relevant information (the Widget that caused the event, the new position of the widget, and the new state of the widget), you can handle such an event on your Dashboard.

Example of handling a closed widget by displaying a button that recreates the widget when clicked

```
c.onWidgetUpdate := {
    e:DashboardUpdateEvent->
    //checking if the event comes from the correct widget and whether it was caused by close:
    if (e.widget == myWidget && e.newState == WidgetState.Closed)
    then
      //adding a Button that will recreate the widget:
      myVerticalLayout.addComponents(
      [new forms::Button("Add My Widget", { e ->
      MyDashboard.addWidget(
         new forms::CheckBox("Check Box", &myBooleanVar), new forms::Rect(width -> 6, height -> 2)
      })]
      );
    end
}
```

8.1.2.2.5 Popup (forms::Popup)

Popup is rendered as a floating dialog box with all the usual features and actions, such as drag, close, resize, etc.

You can design a popup in two ways:

- as a private Popup: these are defined as part of the particular form and cannot be reused.
- as a public Popup: these are defined in their own form definition. While complete forms have their supertype set to `forms:Forms`, public popup forms have their supertype set to the type `forms::Popup` so they can be inserted into another Form tree.

Popups can be modal; each popup can define its popup variables.

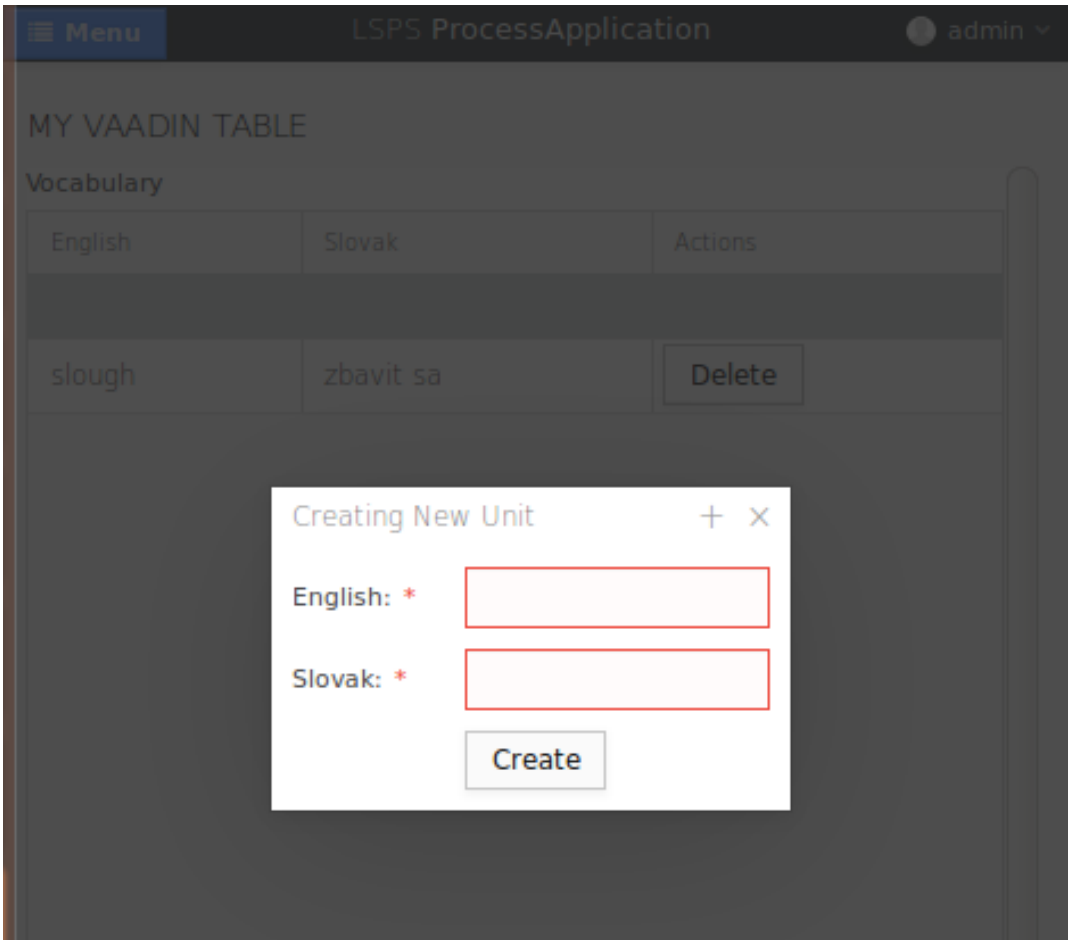


Figure 8.18 Modal popup

8.1.2.2.5.1 Designing a Private Popup

Private Popups can be instantiated only within their parent form and are also designed as its part; They cannot be reused in other forms.

To create a private Popup, do the following:

1. Insert the Popup component into your Form outside of your Form tree.

2. In its Properties view, enter the signature of the method that will return the popup, for example, `edit↩PatientInPopup(Patient p)`. *Do not implement the method*: it is automatically added to the popup factory.

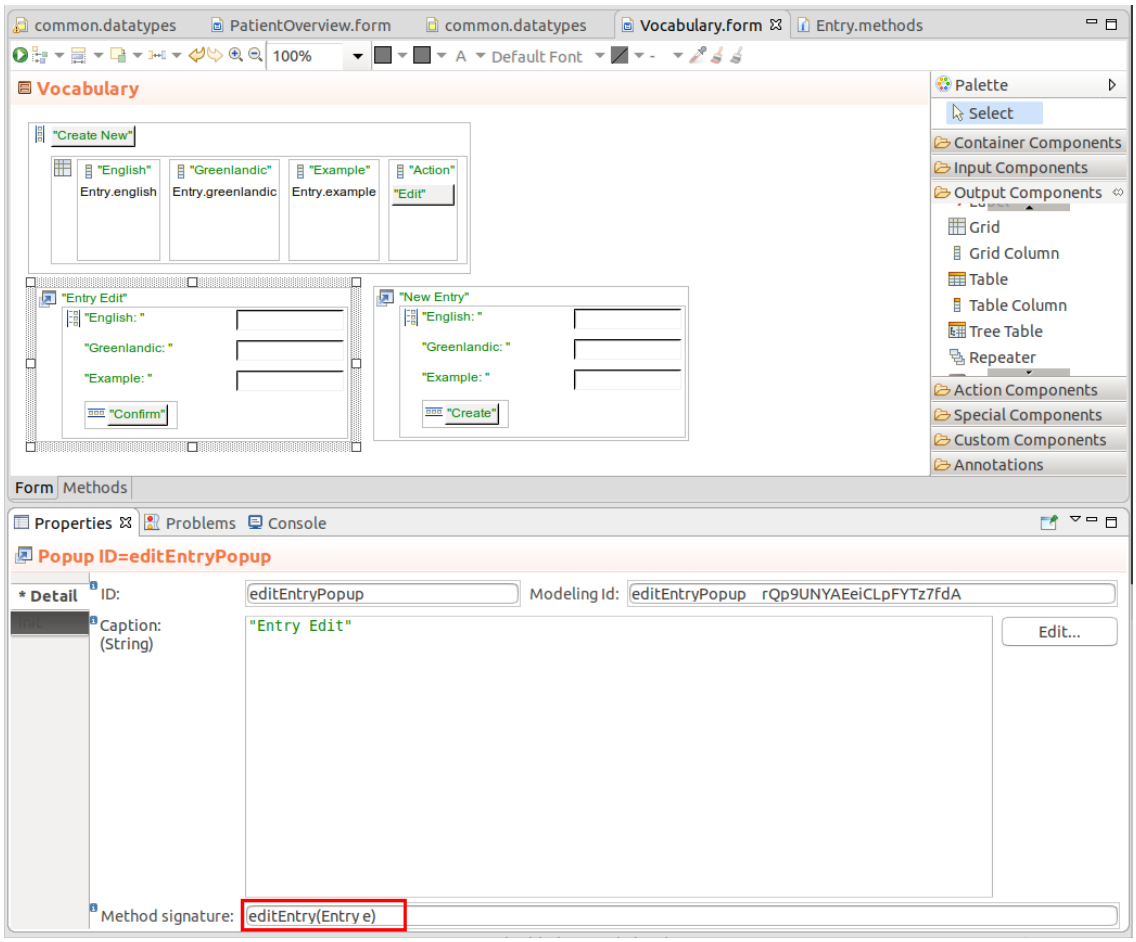


Figure 8.19 Popup method signature with an input parameter

3. Insert the child components into the popup and define their behavior: if you are passing an input argument to the popup method, you can access the input from any child component. Also note that you can define local variables on a popup: right-click the popup in the Outline view to do so.

4. Call the popup method with the method you defined in the **Method signature** field from the form to create the Popup object.

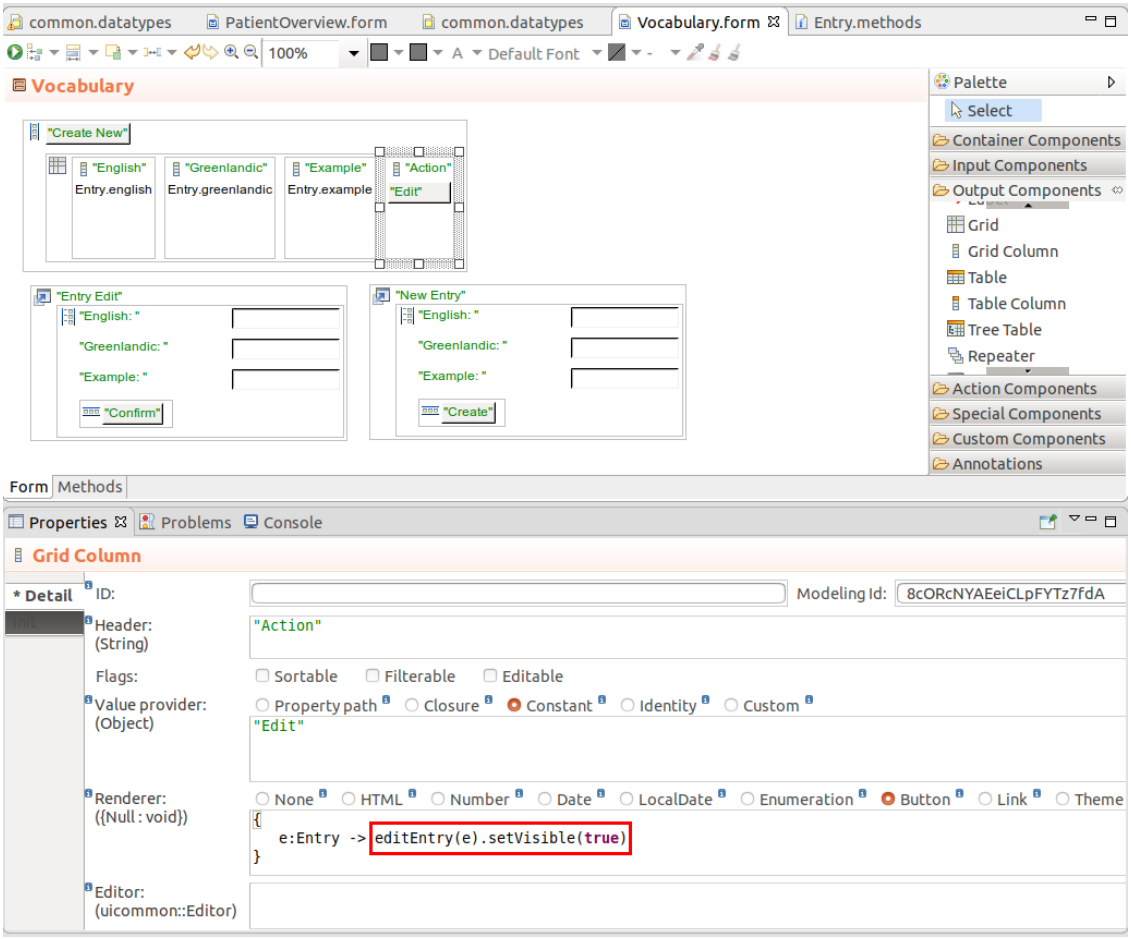


Figure 8.20 Displaying a popup on Grid-row click

Note that a Popup is invisible by default: set the Popup visibility to `true` to display it; you can use the `show()` or `setVisible(true)` call to do so.

5. Close a Popup: from a popup component, set its visibility to `false` or call its `hide()` method. the Popup is queued for garbage collection unless referenced somewhere in the form.

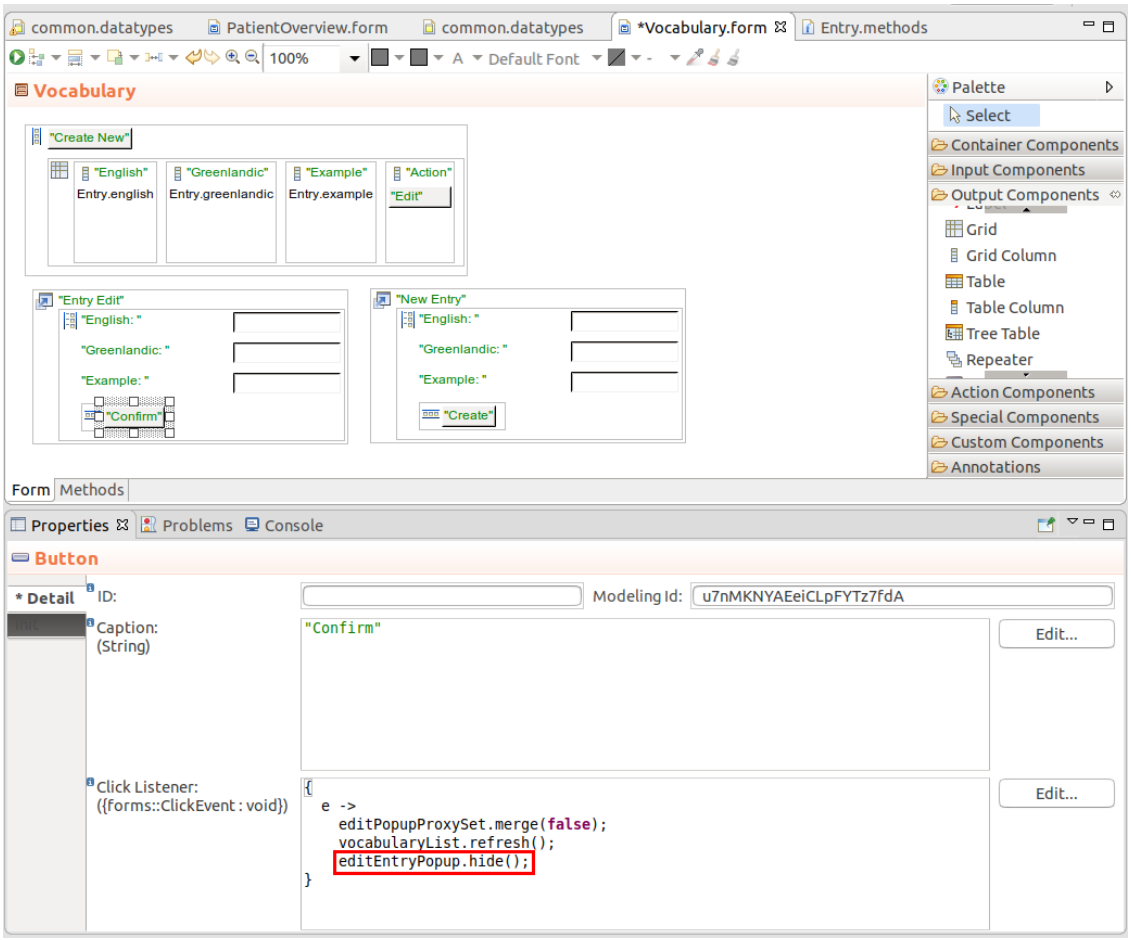


Figure 8.21 Hiding a popup on button click

You can find a complete example of a private Popup with component refresh in the section Quickstart.

8.1.2.2.5.2 Reusing a Popup: Designing a Public Popup

Public popups, unlike private popups, can be used by multiple forms: public popups are snippets with the Popup component that you can insert into another form.

To create a public popup, do the following:

1. Create a form definition.

2. In the Outline view of the form, double-click the root form component and in the Properties view, set forms↩ ::Popup as the Supertype.

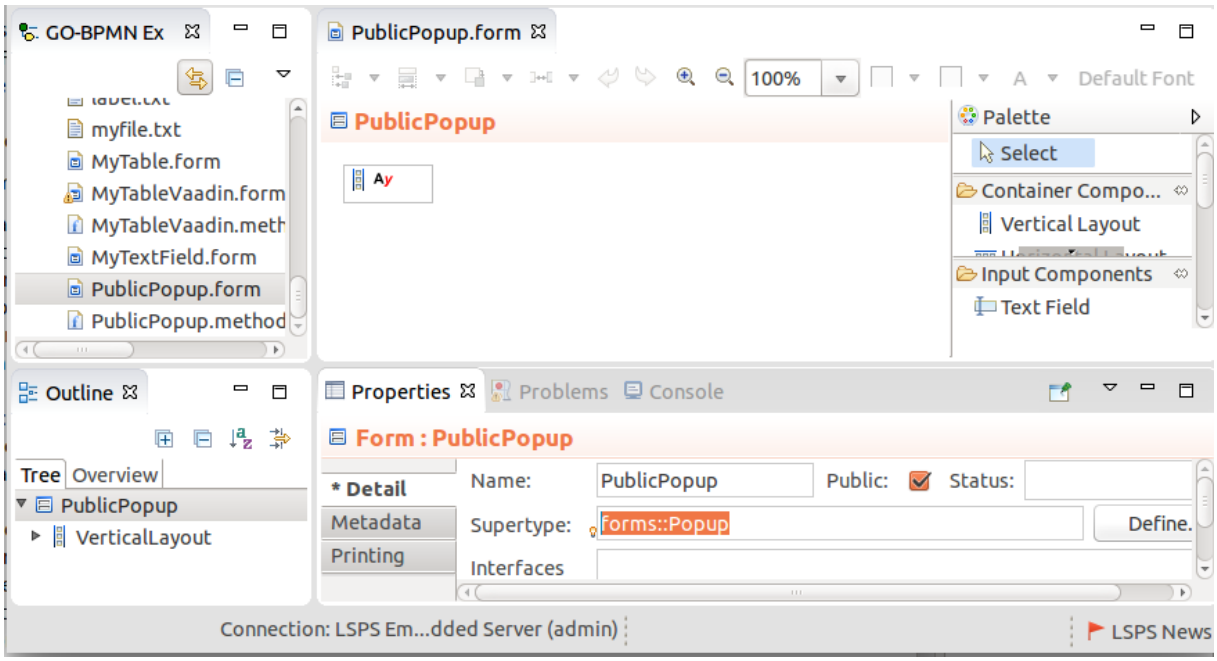


Figure 8.22 Public Popup definition

3. Consider defining a dedicated constructor to set the popup properties and to define parameters of the public popup from the form.

```
//Constructor that takes a parameter and assigns it to a form variable of the public popup
//The popup variable is then used as the binding of its components
public PublicPopup (Unit u) {
  editedUnit := proxy(u);
}
```

4. To create the public popup in a form, call the popup constructor, for example, on a button click:

```
{ click -> new PublicPopup(new Unit())}
```

5. To display and hide the public popup:
   (a) From within the popup, call `show()` in the constructor and `hide()` when required; for example on a close button.
   (b) From the calling form, create a form variable with the popup and call `show()` and `hide()` on the variable when required.

6. To update the underlying form when the Popup is closed, set the close listener on the popup, for example:

```
{ e ->
   def PublicPopup popup := new PublicPopup(u);
   popup.show();
   popup.setPopupCloseListener(
      { e -> MyFormWithPublicPopup.refresh()};
   );
}
```

When creating a public popup, carefully consider:

- *which logic to encapsulate in the popup*;

  For example, if you are creating a popup with user details, the popup might be used by forms to create a new user as well as to edit an existing user: in such a case, you could define a popup parameter that passes the edited user. If no user is passed, the popup constructor creates a new proxy user object; otherwise, it creates a proxy over the passed user object.

- ∗whether to display and hide the popup from within the popup or from the outside form;

  In the example with the user edit and create popup described above, keep the displaying and hiding of the popup inside the popup since it is governed by events within it: display the popup from its constructor; hide it from a button in the popup.

Refer to the tutorial on `editing Grid Data in a popup` for an example of a public popup.

8.1.2.2.5.3 Creating a Popup Variable

To create a variable on a public Popup, do the following:

1. Open the form definition of your Popup.
2. Right-click the root Popup component in the Outline view.
3. Go to **New** > **Variable**
4. In the Properties of the variable, define its name, type, and initial value.

To create a variable on a private Popup, do the following:

1. Right-click the Popup component in the Outline view.
2. Go to **New** > **Variable**
3. In the Properties of the variable, define its name, type, and initial value.

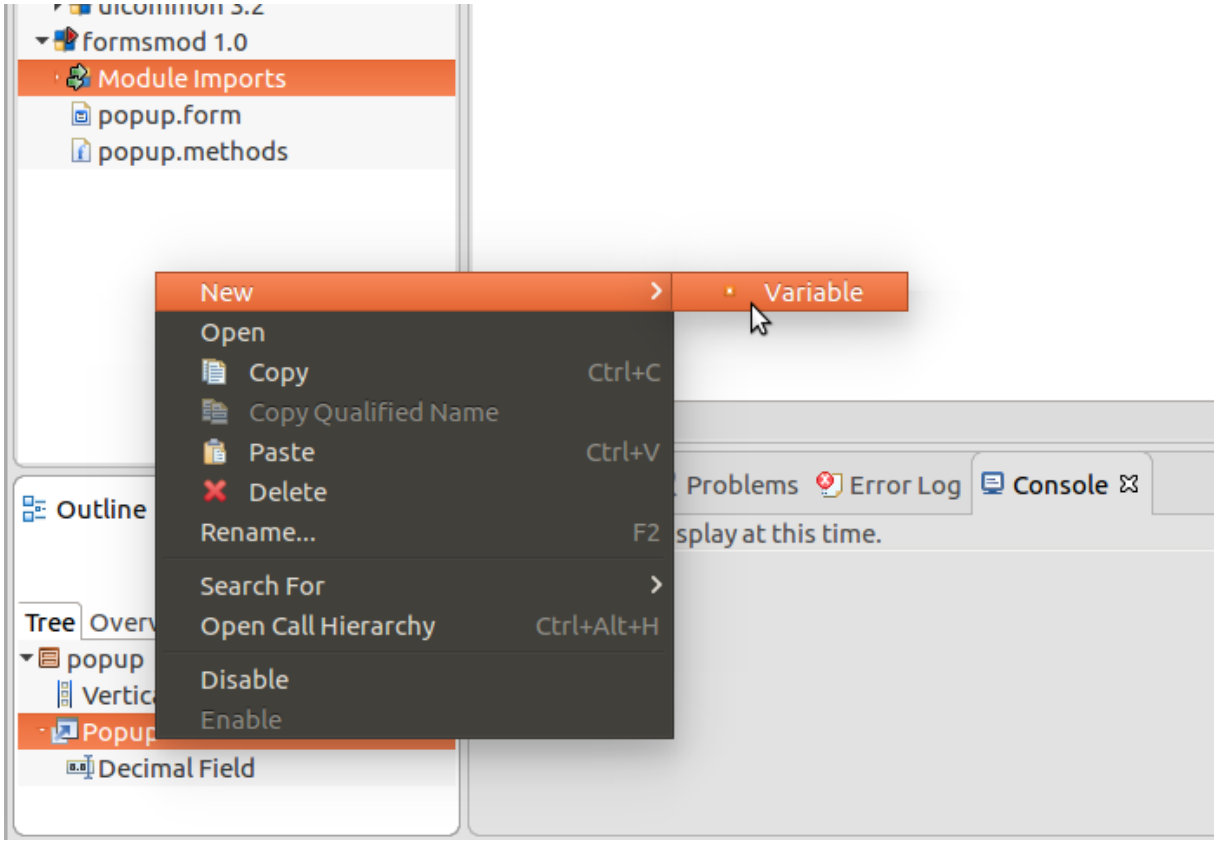


Figure 8.23 Creating instance variable of a popup

8.1.2.2.5.4 Creating a Modal Popup

To make a popup modal, set its modality with the `showModal()` method; for example, `{e -> createPopup(editedEntity, true).showModal()}`

8.1.2.2.5.5 Closing a Popup

To close a Popup, set its visibility to false; you can do so with the `hide()` or `setVisible(false)` calls.

To perform an action when a Popup is closed, use the `setPopupCloseListener()` method, for example:

```
c.setPopupCloseListener({ onCloseEvent -> debugLog({ -> "Popup closed"}, 200)})
```

8.1.2.2.6 Drop Zone (forms::DropZone)

The *Drop Zone* component  allows you to drag-and-drop one or multiple files into the browser to upload it.

It renders as an area where files from desktop can be dropped where the wait it is actually rendered is defined by its child component.

The Drop Zone component defines the action on file upload in its upload-result listener closure. The closure has two input parameters: the first one with the list of uploaded files and the second one with a list of error messages:

- When the upload succeeds, the List<File> parameter holds the uploaded files and the List<String> parameter message is null. Note that if you dropped into the zone multiple files, you will receive all the files.
- When the upload fails, the file parameter is null and the message is an error string.

You can restrict the number of files the user can insert into a drop zone in a single drag-and-Drop action with `setMinFiles()` and `setMaxFiles()` method calls.

If you insert a Drop Zone component into another Drop Zone component, the "highest parent" Drop Zone overrides its children.

While the files are being uploaded, the spinner is displayed and the application ignores user input.

> **Note:** If the user drags-and-drops a file into the form in a browser but misses the drop zone the browser takes over and handles the drop, that is, opens the file. To prevent this behavior, consider using the Drop Zone component as the root component or at least as hight up in the component hierarchy as possible.

Expression that returns a Drop Zone

```
def DropZone dz := new DropZone();
dz.setUploadResultListener(
    {
    uf:List<File>, errorMessage:List<String> ->
          uploadedFiles := addAll(uploadedFiles, uf);
          //uploadedFile is initalized to []
    }
  );
dz.setMaxFiles(2);
//enabling the dropzone:
dz.setDropEnabled(true);
dz
```

8.1.2.2.6.1 Displaying Dropped Files

To visualize the dropped files, for example, display a preview of image files, when they are dropped and uploaded, set the uploaded files as the content of the Drop Zone or of its child component.

Upload result listener that displays thumbnails of uploaded files in the given drop

The current batch is added to a new vertical layout, called imagesUplodedInThisBatch in the drop zone.

```
{
  uploadedFiles:List<File>, errorMessage:List<String> ->
    allUploadedFiles := allUploadedFiles.addAll(uploadedFiles);
    //allUploadedFiles is form variable and initialized in the form constructor.
    if(!allUploadedFiles.isEmpty()) then
      def List<forms::Image> thumbnails := collect(allUploadedFiles, { f:File ->
                  def forms::Image image := new forms::Image(new FileResource(f));
                  image.setWidth("100px");
                  image.setHeight("100px");
                  image}
       );
      def VerticalLayout thumbnailRoot := new VerticalLayout();
      thumbnailRoot.addComponents(thumbnails);
      mydropzone.setContent(thumbnailRoot);
    else
      notify("Failed upload", "Upload of the file failed.", NotificationType.Warning, null, 1000,
    end;
}
```

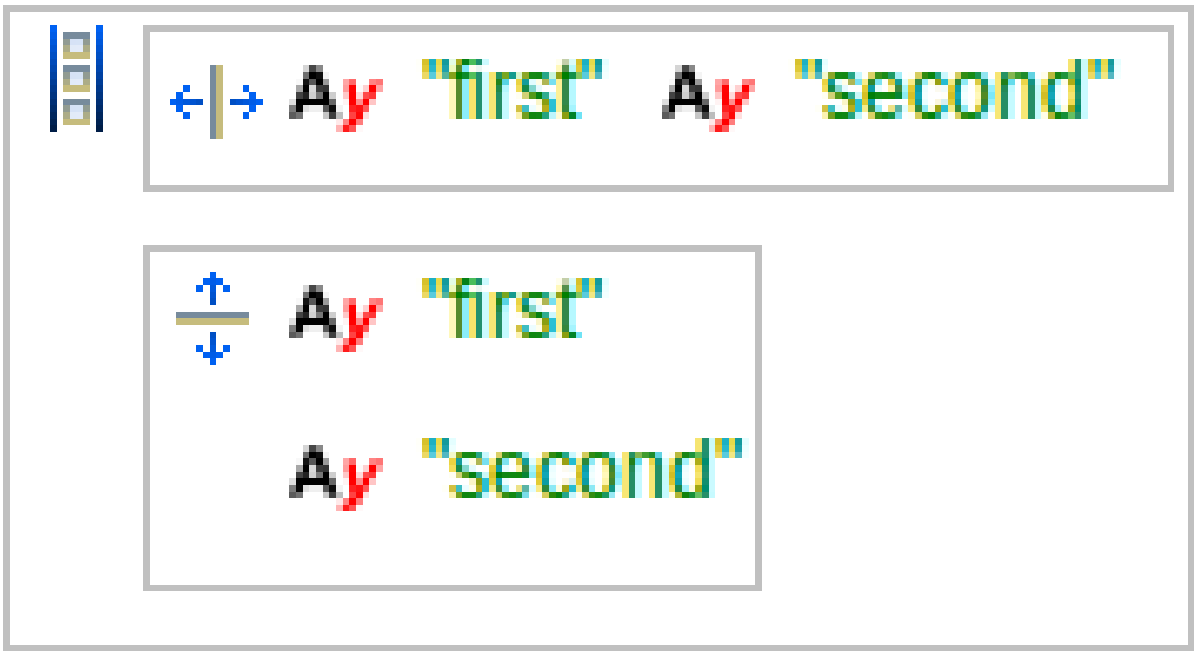
Upload result listener that displays thumbnails all uploaded files: The batch is added to a vertical layout, called allImagesUploaded, that is out of the drop zone.

```
{
  uploadedFiles:List<File>, errorMessage:List<String> ->
    if(!uploadedFiles.isEmpty()) then
~
      def List<forms::Image> fileImagesForAllUploadedImages := collect(uploadedFiles, { f:File ->
                  def forms::Image image := new forms::Image(new FileResource(f));
                  image.setWidth("100px");
                  image.setHeight("100px");
                  image}
       );
      allImagesUploaded.addComponents(fileImagesForAllUploadedImages);
~
    else
      notify("Failed upload", "Upload of the file failed.", NotificationType.Warning, null, 1000,
    end;
}
```

8.1.2.2.7 Split Panels (forms::HorizontalSplitPanel and forms::VerticalSplitPanel)

Split panels hold 2 child components and render a draggable split bar between them so that the space is divided into 2 areas.

You can use the *Vertical Split Panel* or *Horizontal Split Panel*: insert the component into your form and then insert the first and the second component: the split will be rendered between the two automatically.



Expression that returns a Horizontal Split Panel

```
def HorizontalSplitPanel hsp := new HorizontalSplitPanel();
hsp.setFirstComponent(new Label("First Label"));
hsp.setSecondComponent(new Label("Second Label"));
hsp
```

Expression that returns a Vertical Split Panel

```
def VerticalSplitPanel vsp := new VerticalSplitPanel();
vsp.setFirstComponent(new Label("First Label"));
vsp.setSecondComponent(new Label("Second Label"));
vsp
```

You can set the position of the split with the `setSplitPosition()` method: `mySplitPanel.setSplit↩Position(75, SizeUnit.Percentage)` results in the split at 3/4 of the Split Panel area relative to the first component. If you want to use the second component instead, call the method with the additional `true` parameter: `mySplitPanel.setSplitPosition(75, SizeUnit.Percentage, true)`. To the position of the split, call its `isLocked()` method.

To set the range within which the user can drag the split, use the `setMinSplitPosition()` method and `setMaxSplitPosition()` moethods.

```
//Split Panel with restricted split range:
c.setMinSplitPosition(10, SizeUnit.Percentage);
c.setMaxSplitPosition(40, SizeUnit.Percentage);
```

To perform an action when the position of the split is changed, call the `setSplitPositionChanged↩Listener({SplitPositionChanged:void})` method.

8.2 Input Components

Input components serve to acquire input from the user. Depending on their type, they hold a certain value, which is stored in their binding; for example, an input field holds a string, so the binding must be able to hold a string value.

When the user enters an input or when they change the focus of the cursor, depending on whether the component is immediate, the component checks if the input value is correct; for example, whether the value in a Decimal Field contains a number. If the value is incorrect, the field is marked as invalid and no further processing takes place. If the value is correct the binding value is updated and the component produces a value-change event, *ValueChangeEvent*: when the event is produced, the closures defined by the change listeners of the component are executed (see Performing Action on Input in an Input Component).

8.2.1 Common Properties and Functionalities

- *Binding*: the relevant value and target slot; for a Text Field, binding represents the displayed and stored value; for Tree Table this is the currently selected node, etc.
- *Required*: if checked, the component is marked with the ∗ symbol to indicate that the user must enter a value into the component; no validation logic is provided.
- *Immediate*: if the changes on the components are to be produced immediately or on focus change
- *Validation*: you can add validation rules for the input values with the `addValidator()` call

8.2.2 Getting the User Input from an Input Component

To get the value as entered by the user, that is, even if it is not valid, call `getUserText()`. The return value is always a String.

> **Note:** To acquire the value in the target object of the binding that is the most recent valid value, use `getValue()`.

8.2.3 Performing Action on Input in an Input Component

An *input component* produces a *value change event* when its input value changes. When the event is produced, the following is performed:

- The action defined by the `setOnChangeListener()` method
- The action defined by the `setOnValidValue()` method if the value of the component is valid, that is, none of the validators (inferred, explicitly added, or validator of the format) returns a message (`isValid()` returns `true`); it is basically a shorthand for

```
if myComponent.isValid() then
...
end
```

Example

```
c.setNumberFormat("#km");
c.setOnValidValue({ e ->
  def String userText := c.getUserText();
  userText := userText.replaceAll("km", "");
  metersField.setValue(userText.toDecimal()*1000);
});
```

If you need to work with previous values of an input component, you can acquire these from the `oldValue` property of the ValueChangeEvent.

```
c.setOnChangeListener({e:ValueChangeEvent-> oldValue.setValue(e.oldValue);
newValue.setValue(c.getValue())})
```

8.2.4 Input Components Reference

Input components hold a value and allow the user to enter a value: when this happens, the components executes the action defined by its *value change listener*. The input values are stored in the target object defined by their binding.

8.2.4.1 Text Components

Text components, that is, Text Field, Text Area and Password Field, allow the user to enter free-text input with the following additional options:

- restriction of the number of characters using the *setMaxLength()* methods The user will not be able to enter an input longer than the set limit.

```
c.setMaxLength(20)
```

- display a hint in the component before the user enters their input using the *setHint()* method

```
c.setHint("Enter your notes here.");
```

- automatically expand the area vertically to wrap the input text with the `setExpandable(true)` method call

8.2.4.1.1 Text Field (forms::TextField)

The *Text Field* component ( ) is rendered as a single-line text field that allows entering text input.

> **Important:** The caption of the Text Field is rendered by the parent layout component. Hence if you create a form with a sole Text Field component, the form will be rendered without its caption text. This is due to limits of the Vaadin framework.

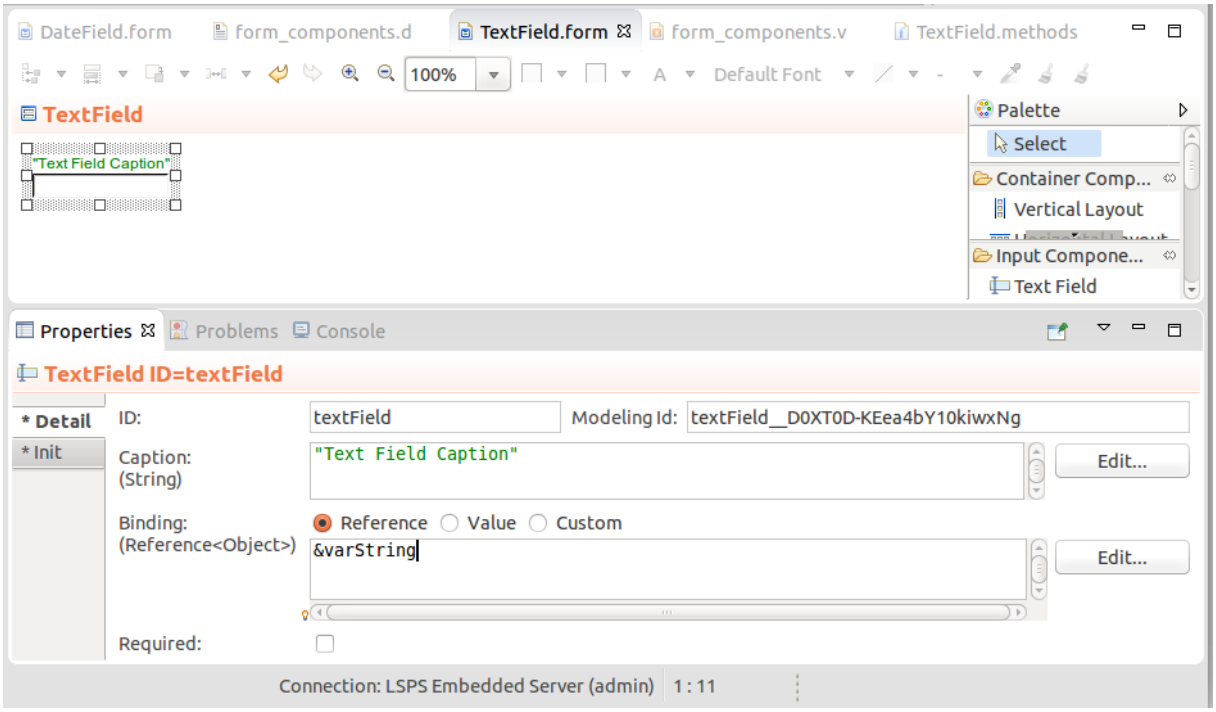


Figure 8.24 Form with a Text Field

Expression that returns a TextField:

```
textField := new forms::TextField("Text Field Caption", new forms::ReferenceBinding(&varString));
textField.setValue("Binding value in the reference string object");
textField.setMaxLength(10);
textField
```

Figure 8.25 Text Field rendered in the default theme

8.2.4.1.2 Text Area (forms::TextArea)

The *Text Area* component () is rendered as a multi-line free-text field. It is expandable by default: its length adapts to accommodate user input.

If you want to set the height, that is, the number of lines in the text area:

- If the Text Area is expandable (by default), use `setMaxRows()` and `setMinRows()`.
- If the Text Area is not expandable, use the `setRows(Integer rows)` method.

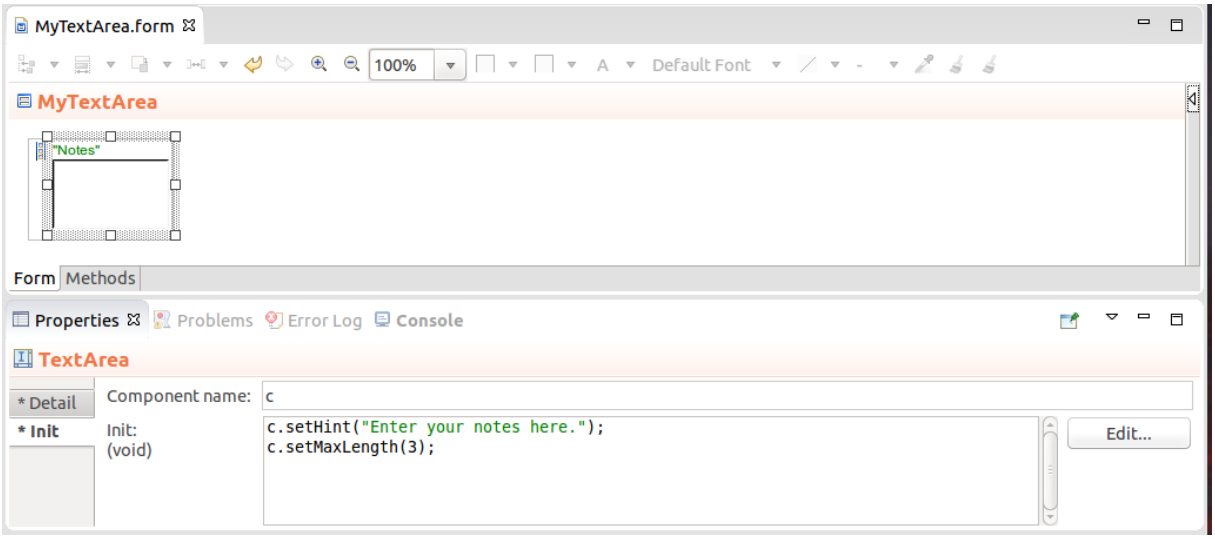


Figure 8.26 Form with a Text Area

Important: The caption of the Text Field is rendered by the parent layout component. Hence if you create a form with a sole Text Area component, the form will be rendered without its caption text. This is due to limits of the Vaadin framework.

Expression that returns a Text Area

```
myTextArea.setHeight("4")
```

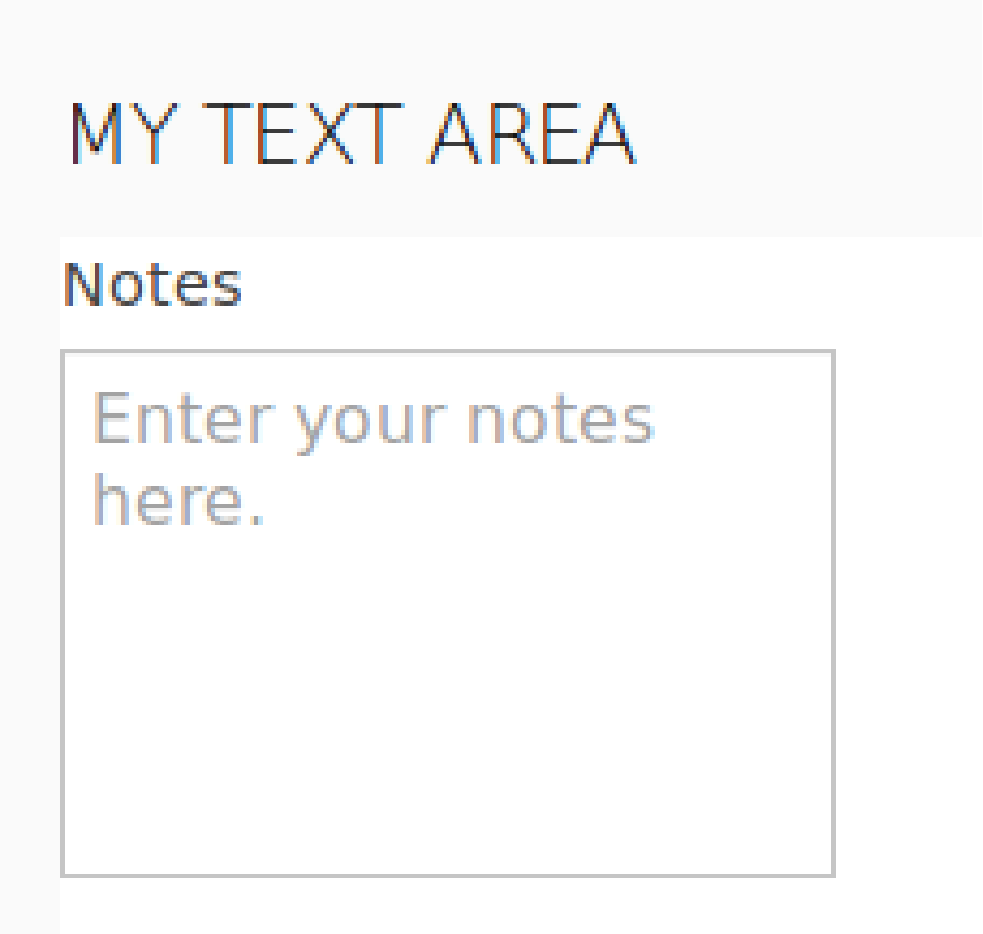
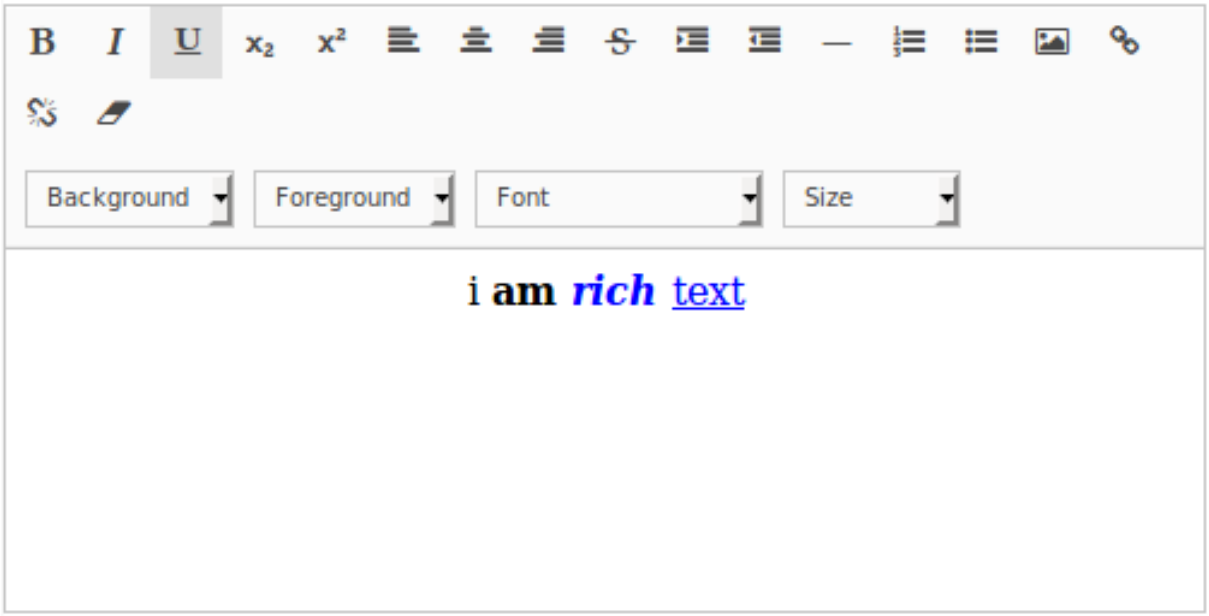


Figure 8.27 Text area rendered

8.2.4.1.3 Rich Text Area (forms::RichTextArea)

The Rich Text Area component ( ) enables the user to enter formatted text. You can also set the area to hold a formatted text from your form: use HTML formatting:

```
MyRichText.setValue("i <b> am <font color=""blue""> <i>rich </i> </font> </b> <font color=""blue"">
       <u>text</u> </font>")
```



8.2.4.1.4 Password Field (forms::PasswordField)

The *Password Field* component () is a single-line text field that hides the input.

When implementing password fields the component only prevents the displaying of the input. To secure the provided data, make sure the form uses the HTTPS protocol.

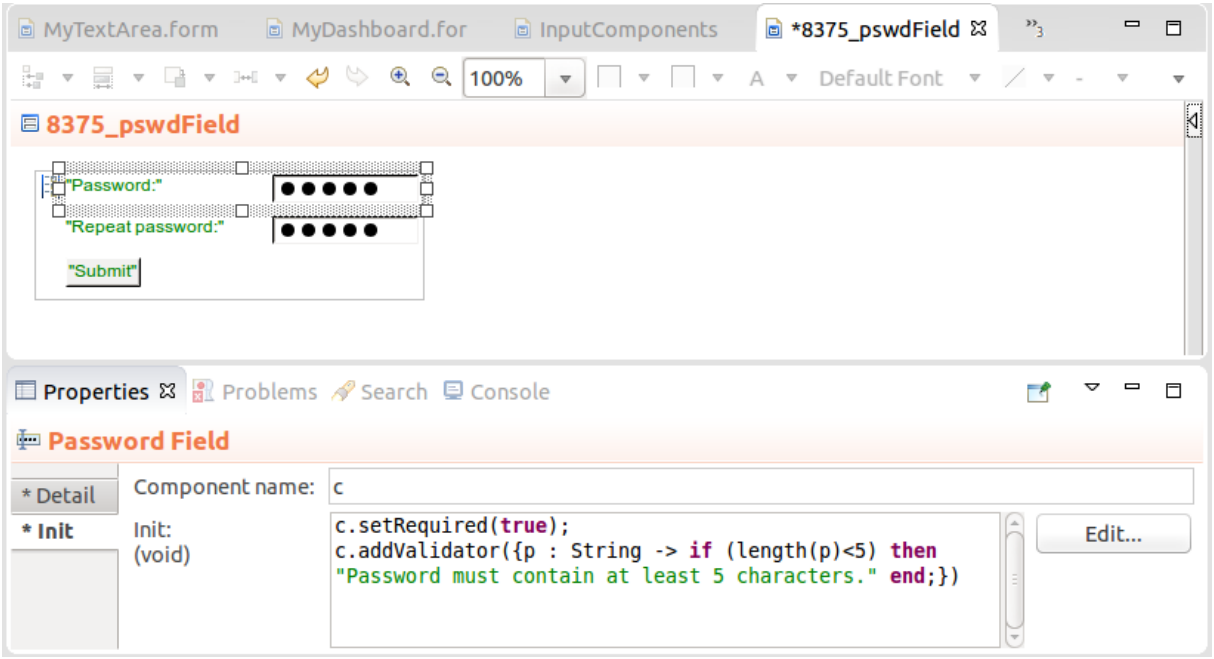


Figure 8.28 Form with a Password Field

Important: The caption of the Password Field is rendered by the parent layout component. Hence if you create a form with a sole Password Field component, the form will be rendered without its caption text. This is due to limits of the Vaadin framework.

Expression that returns a Password Field

```
def PasswordField p := new PasswordField("Password:", &password);
p.focus();
p.isRequired();
p
```

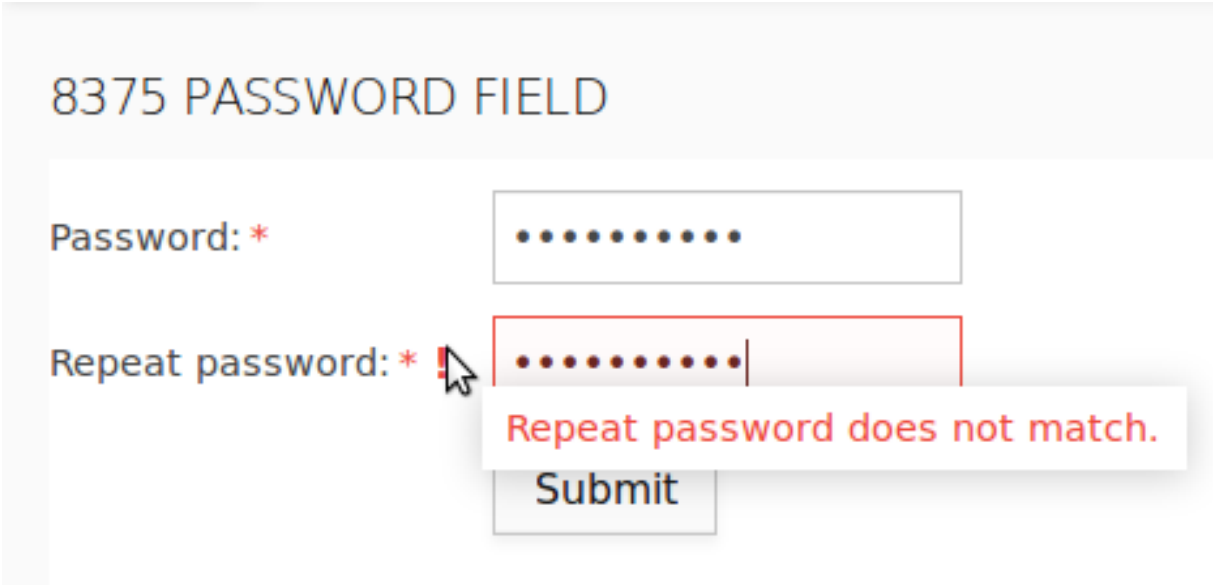


Figure 8.29 Password Fields rendered

8.2.4.2 Decimal Field (forms::DecimalField)

The *Decimal Field* component () is rendered as a single-line field that allows the user to enter a decimal value. The format of the decimal value is by default set to user's locale: You can refine the allowed format with the setNumberFormat method.

> **Important:** The caption of the field is rendered by the parent layout component. Hence if you create a form with a sole Text Field component, the form will be rendered without its caption text. This is due to limits of the Vaadin framework.

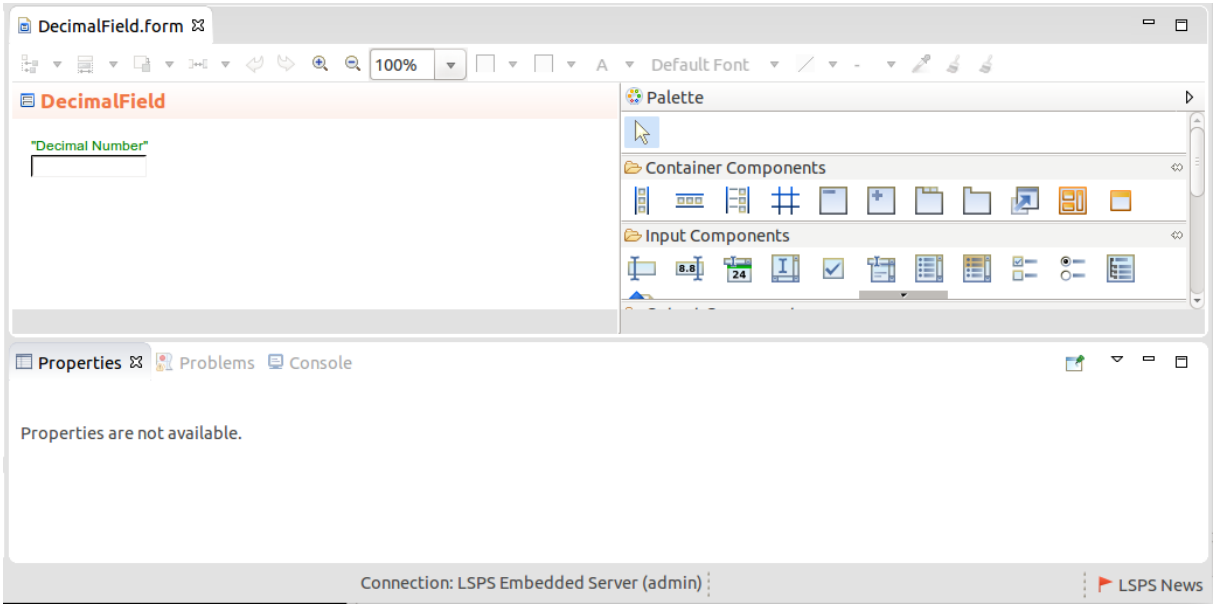


Figure 8.30 Form with a Decimal Field

Expression that returns a DecimalField

```
new forms::DecimalField("Value", &varValue);
```

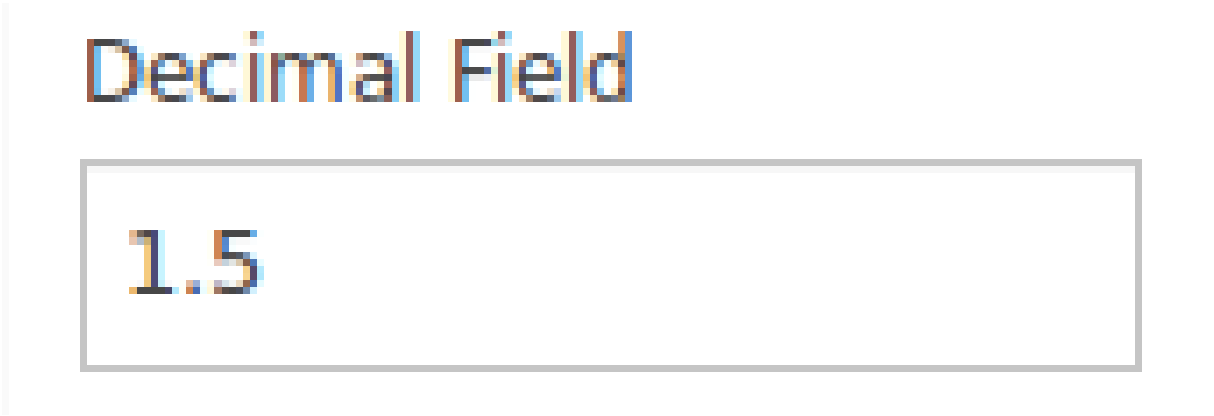


Figure 8.31 Decimal Field rendered in the default theme

Figure 8.32 Decimal Field with an invalid value

8.2.4.2.1 Defining Decimal Format

To define the format of the decimal value in a Decimal Field, use `setNumberFormat()`:

```
def forms::DecimalField df := new forms::DecimalField("Value", &value);
df.setNumberFormat("#,##0.00;(#,##0.00)");
df.setOnChangeListener({
   changeevent ->
      def Integer rounded := roundUp((df.getValue() as Decimal));
      value := rounded;
      df.setValue(rounded)
    }
);
```

8.2.4.3 Date Field and Local Date Field

The *Date Field* and *Local Date Field*  components are rendered as a single-line text field that allows the user to enter a date either using the calendar picker or by typing the date in one of the allowed date formats.

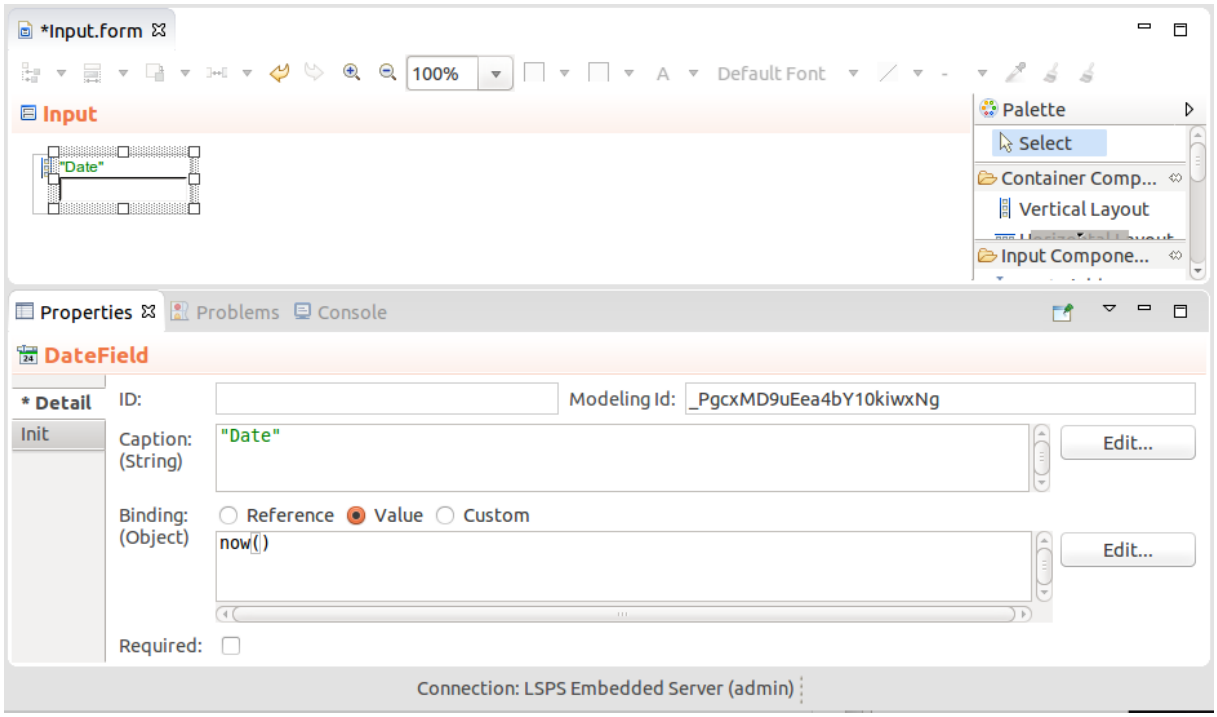


Figure 8.33 Form with a Text Field

Expression that returns a Date Field

```
def Date meetingTime :=  now();
def DateField myDateField := new DateField("Meeting", &meetingTime);
myDateField.setDateFormat("yyyy-MM-dd' at 'HH:mm:ss z");
myDateField.addValidator(
  {
    dv:Date -> if (dv > date(2016,3,3))
```

```
    then
      null
    else
      "The meeting must occur later than March 2, 2016."
    end
  }
);
myDateField
```

Expression that returns a Local Date Field

```
def LocalDate meetingTime :=  now().toLocalDate();
def LocalDateField myLocalDateField := new LocalDateField("Meeting", &meetingTime);
myLocalDateField.setDateFormat("yyyy-MM-dd");
myLocalDateField.addValidator(
  {
    ldv:LocalDate -> if (ldv > date(2016,3,3).toLocalDate())
    then
      null
    else
      "meeting too soon"
    end
  }
);
myLocalDateField
```

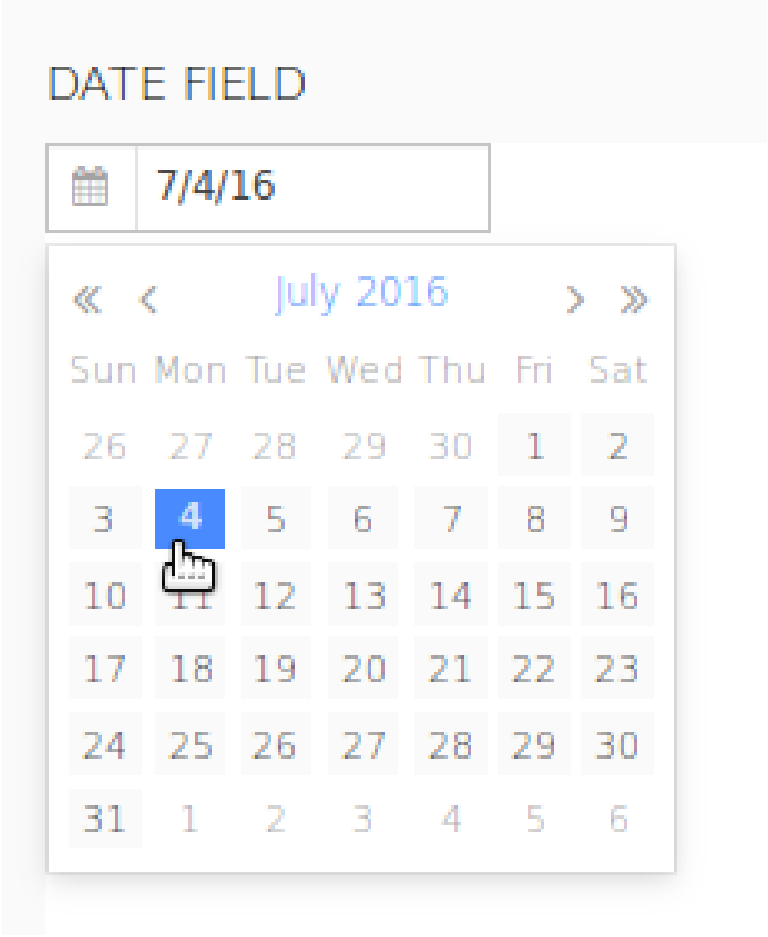


Figure 8.34 Date field with the default date format and with the date picker expanded

8.2.4.3.1 Defining Allowed Date Format on a Date Field and Local Date Field

To define the format of the date the user can enter into the Date Field or Local Date Field, call `setDate↩Format()` or `setDateFormats`. The format parameter is defined as a String with the possible values of the java SimpleDateFormat (refer to `http://docs.oracle.com/javase/6/docs/api/java/text/↩SimpleDateFormat.html`).

To use the date format of the LSPS Application, set the format either to `date` to use the `dateFormat` or to `dateTimeFormat` to use the default date-and-time format via the *LspsAppConnector.getApplicationMessage()*.

To set the error message displayed when the entered date does not match the required format with the method `setParseErrorMessage()`.

8.2.4.4 Check Box (forms::CheckBox)

The *Check Box* component (☑) is a simple check box with Boolean Binding: its caption renders next to the check box.

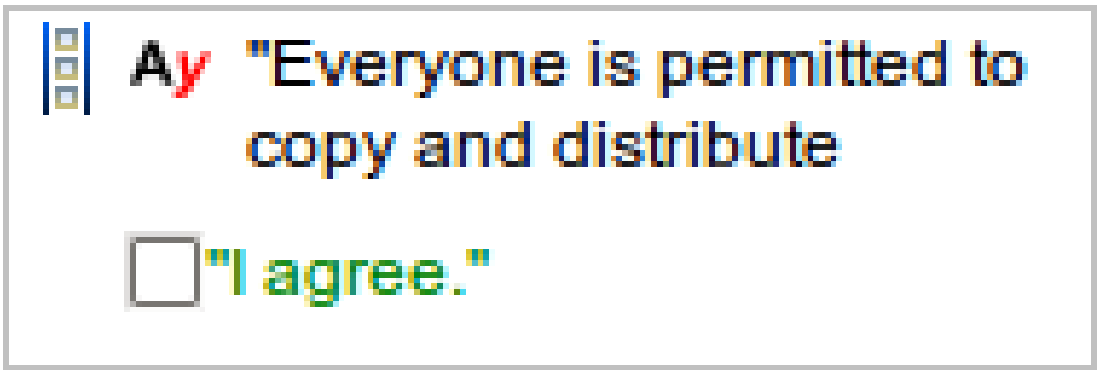


Figure 8.35 Form with a Check Box

Expression that returns a CheckBox

```
new forms::CheckBox("I agree", &varAgree)
```

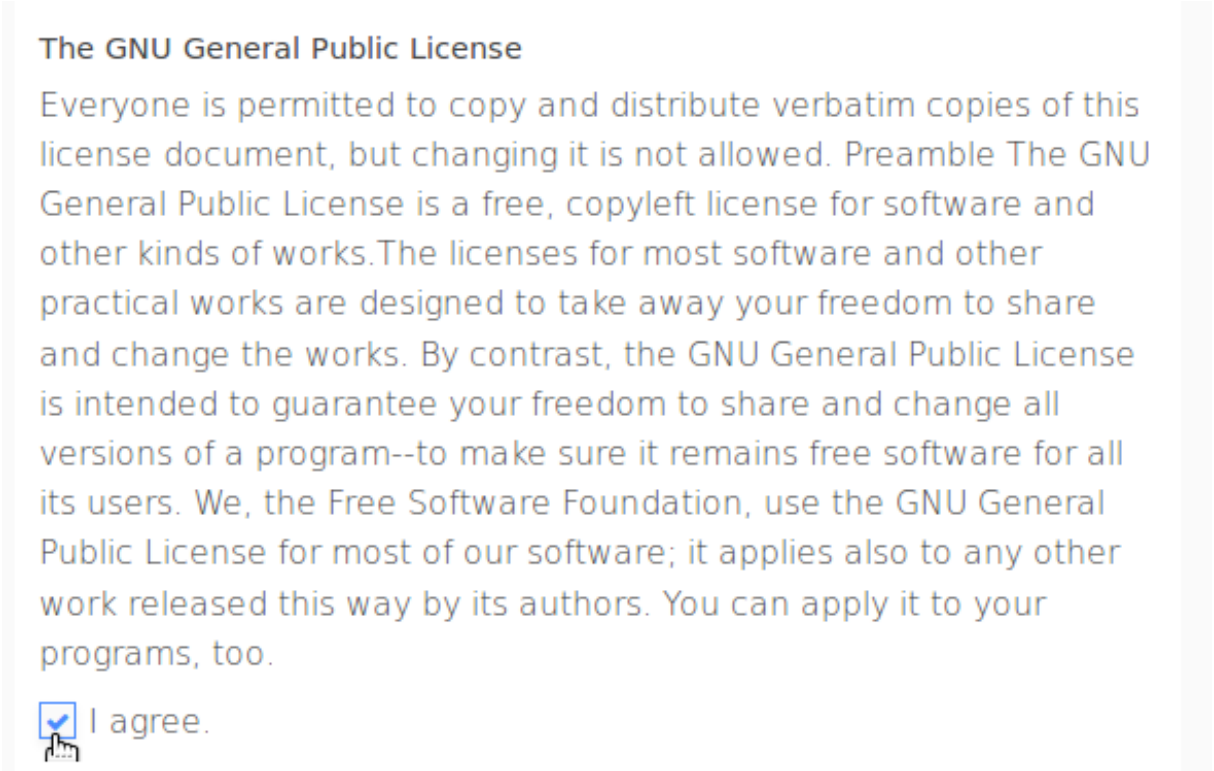


Figure 8.36 Check Box rendered in the default theme

The Check Box component produces the ValueChangeEvent when its check box is selected or unselected: to handle such a change, add a Change Listener to the component.

Example Change Listener on a Check Box component

```
confirmationCheck.setOnChangeListener(
   { e ->
      if confirmationCheck.getValue() = true
        then c.setEnabled(true)
      end
    }
)
```

8.2.4.5 Select Components

The *Select* components provide the user options which they can select and include such component as combo box, radio button list, etc.

8.2.4.5.1 Removing the No-Value Option in Select Components

By default, the select components contain an option with no value the user can select: to disable this `no-value` option, call *setNullSelectionAllowed(false)* on the component.

8.2.4.5.2 Setting the Select Options

You can define the options available in a select component either in the Properties view or with the component methods:

- The options defined in the *Options property of the component* are loaded when the form is loaded.
  - *Map*: map of values and labels; for example, `map(findAll(Region), { region:Region -> region.name })`
  - *Objects*: collection of objects
    The labels are calculated by calling the `toString()` function over the objects, for example, `findAll(Region)`.
  - *Select items*: collection of options defined as SelectItem objects; for example,

```
collect(findAll(Region),
{ r:Region -> new SelectItem(value -> r, label -> r.name)}
)
```

- The options set with the `setOptionsDataSouce()` or `setOptions()` method load the options from anywhere in the form and you can load it at the moment when the component is created. Options loaded in this way are lazy-loaded: they are loaded at the moment they are displayed.

  Setting options with method calls

```
//various calls that set the options in a combo box:
myCombo.setOptionsDataSource(new TypeDataSource(User), new PropertyPathValueProvider(User.name));
myCombo.setOptions(allValues, new IdentityValueProvider(String));
myCombo.setOptions({ 1, 2, 3 })
```

  Important: If you set the options of a component in both ways, that is, using the Options property and the setOptionsDataSouce() or `setOptions()` call, the one called later takes precedence: to identify the order of the method calls, right-click the form, select **Display Widget Expression**, and examine the code.

8.2.4.5.3 Combo Components

8.2.4.5.3.1 Combo Box (forms::ComboBox)

The *Combo Box* component () renders as a single-line text field with a drop-down list of options: The user can select one option in the drop-down menu.

Options can be defined as a Map or a Collection, or an OptionsDataSource. To define an action when the user selects or unselects an option, use the `setOnChangeListener()` method.

8.2.4.5.3.2 Search Combo Box (forms::SearchComboBox)

The *Search Combo Box* component () renders as a single-line text field with a drop-down list of options: The user can select one option in the drop-down menu and the options are filtered out once the user has entered the minimum amount of characters (3 by default).

The value set in the filter is passed as a SubstringFilter to the *Options* data source: if you are using a custom data source, make sure that the data source processes the filter value correctly.

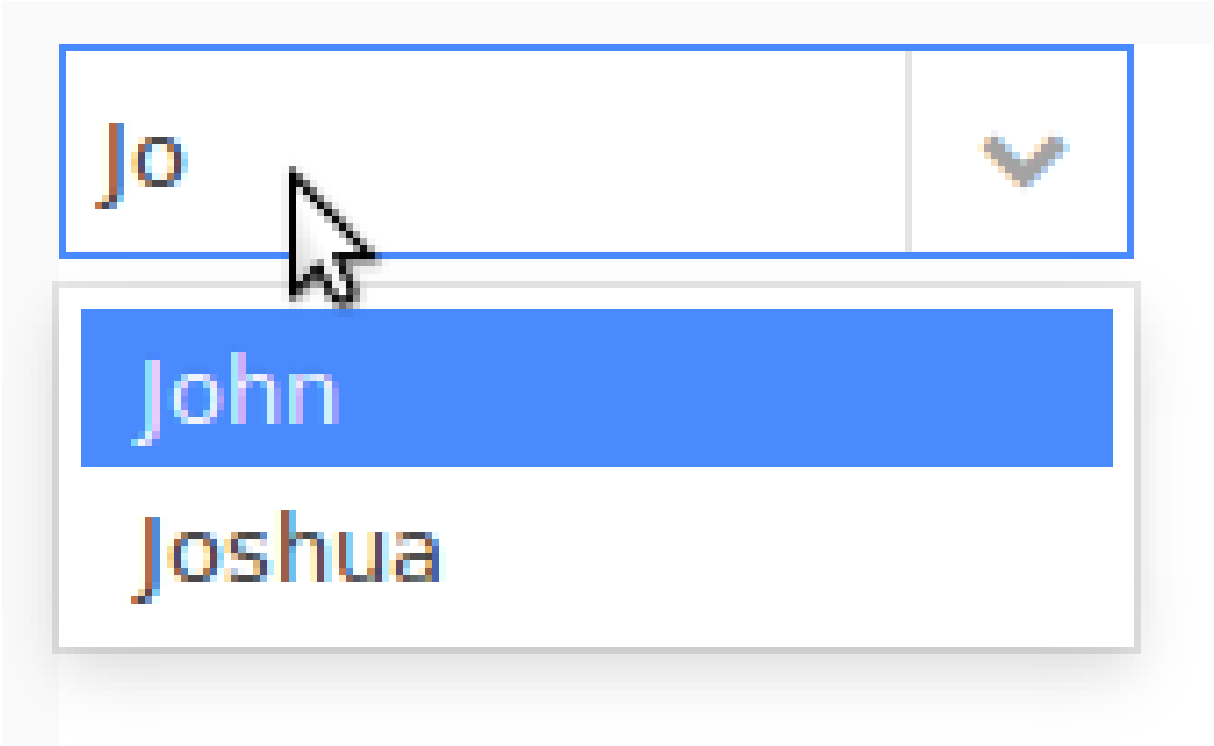
```
//Example getData() method of a custom data source (record implements forms::DataSource):
  public List<Object> getData(Integer startIndex*, Integer count*, Collection<forms::Filter> filte
     if filters.isEmpty() then
        getAll(startIndex, count)
     else
        //value in the filter:
        def String currentFilter := cast(filters[0], SubstringFilter).value;
        getMyDataFromMyDS(currentFilter)
     end
  }
```

Options can be defined as a Map or a Collection of objects or SelectItems, or an OptionsDataSource.

To define an action when the user selects an option, use the `setOnChangeListener()` method.

In addition, you can set the following:

- minimum number of characters for the search filter to apply with the `setMinimalFilterLength()` method
- number of options displayed in the drop-down with the `setPageLength()` method
- time to wait after user input before searching for applicable options with the `setFilterTimeout()` method



8.2.4.5.3.3 Support for Entering a Custom Value

To allow the user to enter a value that is not available in the options in a combo box component, use the `setNew↩ ItemHandler()` method.

```
c.setNewItemHandler(
    {
      newValue:String ->
      newValue
    }
)
```

Note that the closure has void as return value, the user will not be able to enter a custom value:

```
c.setNewItemHandler(
    {
      newValue:String ->
      //The closure returns void so the user will be able to enter only one the option values:
      debugLog({ -> "User entered a new value: " + newValue}, INFO_LEVEL);
    }
)
```

If you enable this feature on a combo component, typically you will need to add the new option to the options of the combo component: use the `setOptions()` method to implement this behavior. Also, if you working with options that are persisted while new options are not persisted, consider defining the ValueProvider parameter↩ : ValueProvider defines how to display the options and that possibly including your new option. For example, you could handle a new option as follows:

```
c.setNewItemHandler(
    {
      newValue:String ->
      options := add(options, new SelectItem(label -> newValue, value -> newValue));
      c.setOptions(options);
      //the value returned when you request value of the combo component:
      newValue
    }
);
```

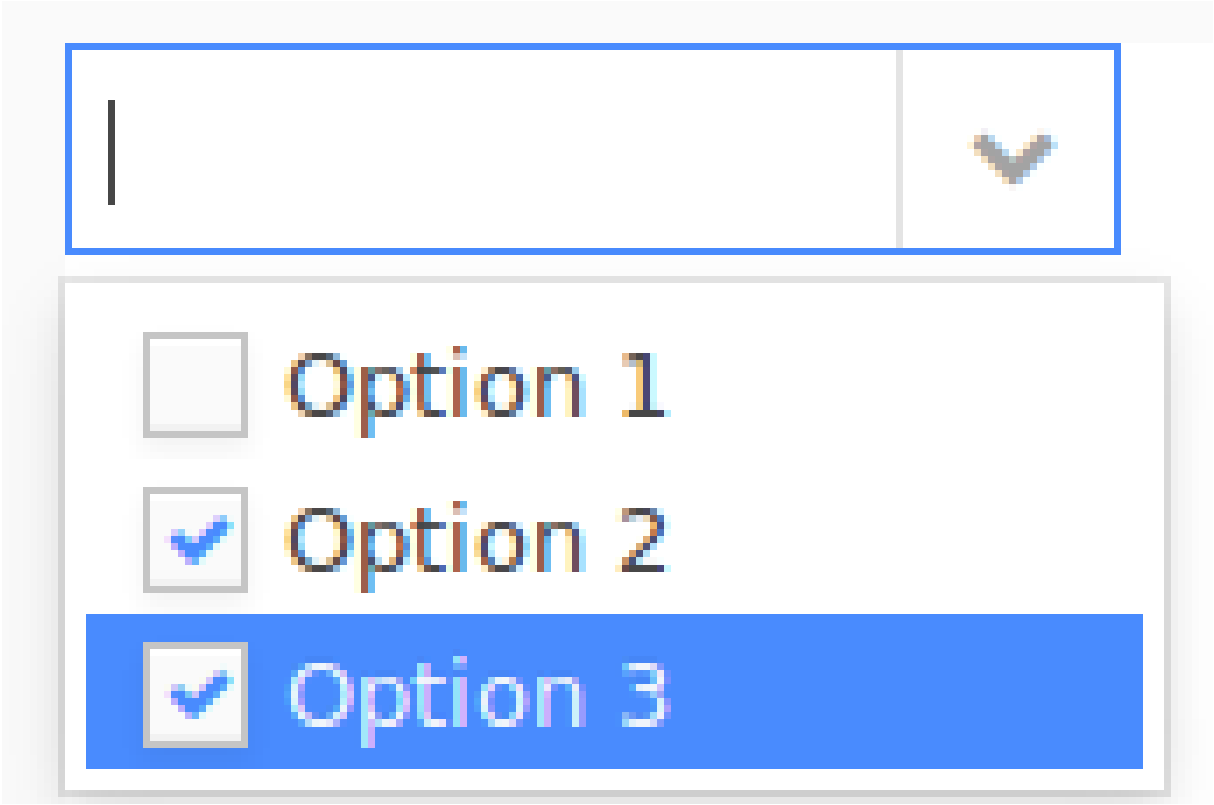
To modify the added options, you can preprocess the options in the setOptions() method with a value provider:

```
c.setNewItemHandler(
    {
      newValue:String ->
      options := add(options, new SelectItem(label -> newValue, value -> newValue));
      c.setOptions(options, new ClosureValueProvider({ newOption:String -> newOption}));
      //the value returned when you request value of the combo component:
      newValue
    }
);
```

8.2.4.5.4 Multi-Select Combo Box (forms::MultiSelectComboBox)

The *Multi-Select Combo Box* component () renders as a single-line text field with a drop-down list of options. The user can select multiple options in the drop-down menu.

Options can be defined as a Map or a Collection, or an OptionsDataSource. To define an action when the user selects or unselects a combo box option, use the `setOnChangeListener()` method.



8.2.4.5.5 Single-Select List (forms::SingleSelectList)

The *Single-Select List* component ( ) displays a list of options from which the user can select one option. You can set a limit to the number of displayed options using *setRowCount(Integer)*: if more options are available that the set limit, the list of options becomes scrollable.

To define the action performed when select or unselect of an item, use the `setOnChangeListener()` method.

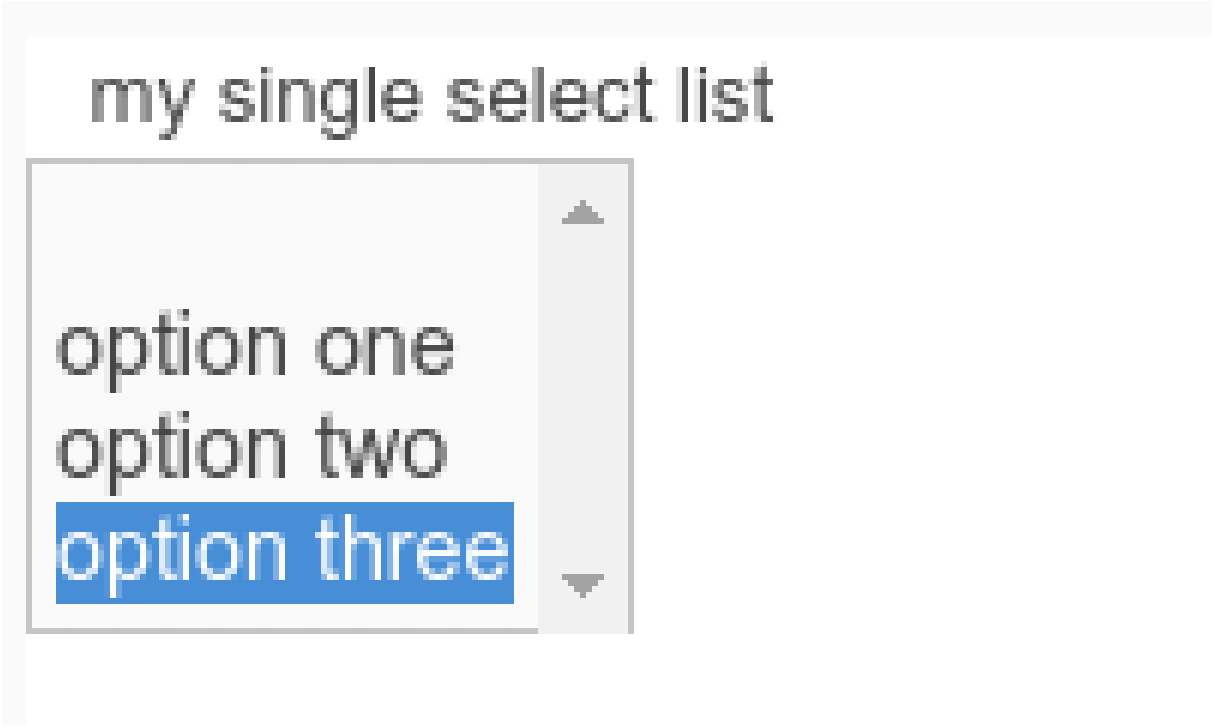


Figure 8.37 Single Select List with an option selected

Example expression that returns a single select list

```
def String selectedValue;
def SingleSelectList singleSelect := new SingleSelectList("My Single Select", &selectedValue);
singleSelect.setOptions([ "value" -> "displayed key", "another value" -> "another displayed value" ]);
singleSelect.setOnChangeListener( {_ -> myText.setValue(singleSelect.getValue())});
singleSelect.setRowsCount(2);
singleSelect.setNullSelectionAllowed(false);
singleSelect;
```

8.2.4.5.6 Multi-Select List (forms::MultiSelectList)

The *Multi-Select List* component ( ) displays a list of options from which the user can select multiple options.

It can define the number of displayed Options using *setRow(Integer)*: if more options are available that the set limit, the list of options becomes scrollable.

To define an action when the user selects or unselects an item, use the `setOnChangeListener()` method.

Example expression that returns a multi select list

```
def MultiSelectList msl:= new MultiSelectList("", &selectedItems);
msl.setOptions([ "value1" -> "displayed option 1", "value2" -> "displayed option 2"]);
msl.setOnChangeListener( { _-> selection.refresh() });
msl.setNullSelectionAllowed(false);
msl.setRows(2);
msl;
```

8.2.4.5.7 Check Box List (forms::CheckBoxList)

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. To define an action when the user selects or unselects an option in the list, use the `setOnChangeListener()` method.

Example expression that returns a check box list

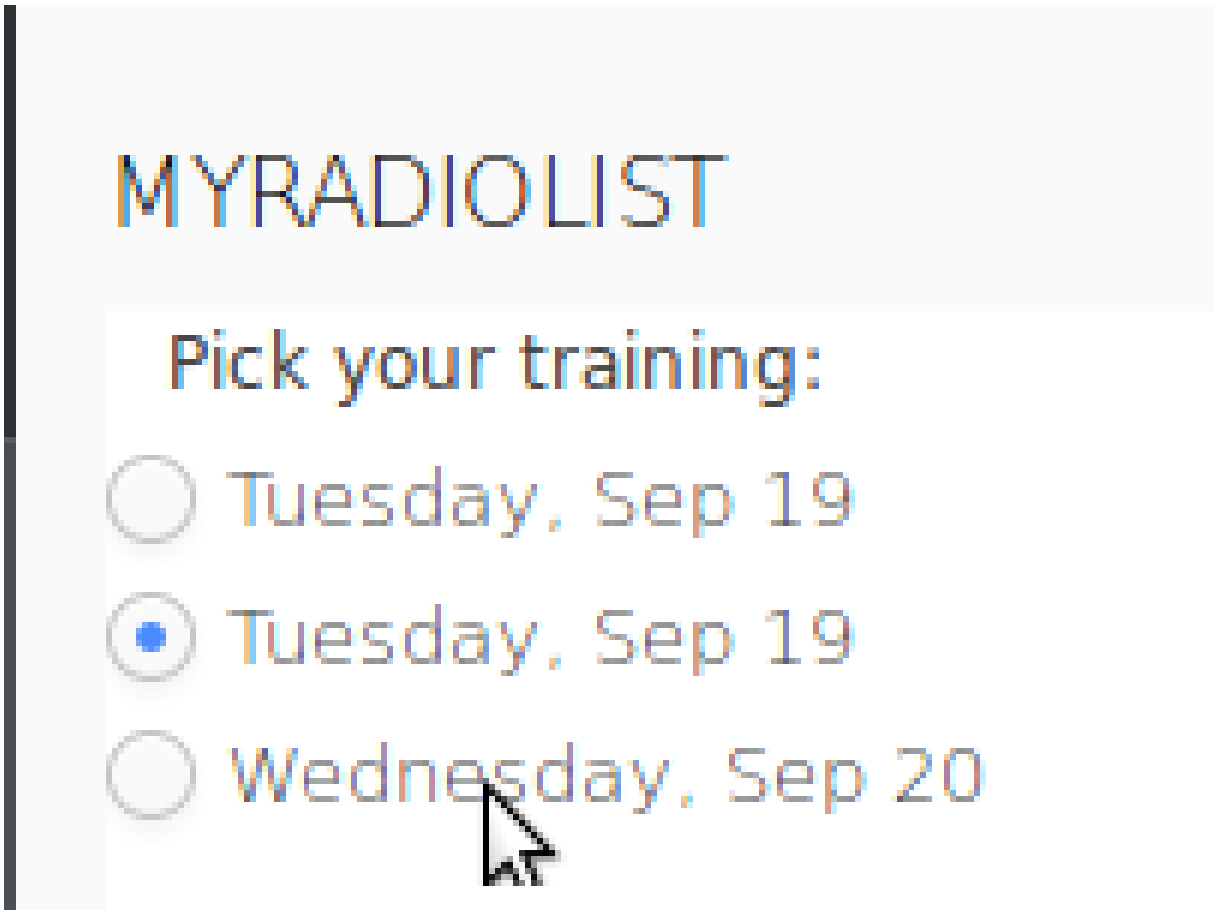
```
def CheckBoxList cbl:= new CheckBoxList("", &selectedItems);
cbl.setOptions([ "value1" -> "displayed option 1", "value2" -> "displayed option 2"]);
cbl.setOnChangeListener( { _-> selection.refresh() });
cbl.setNullSelectionAllowed(true);
cbl;
```

8.2.4.5.8 Radio Button List (forms::RadioButtonList)

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. To define an action when the user selects or unselects an option in the list, use the `setOnChangeListener()` method.

Example expression that returns a radio button list

```
def RadioButtonList rbl:= new RadioButtonList("");
rbl.setOptions([ "value1" -> "displayed option 1", "value2" -> "displayed option 2"]);
rbl.setBinding(&selectedItem);
rbl.setOnChangeListener( { _-> select.refresh() });
rbl.setNullSelectionAllowed(true);
rbl;
```

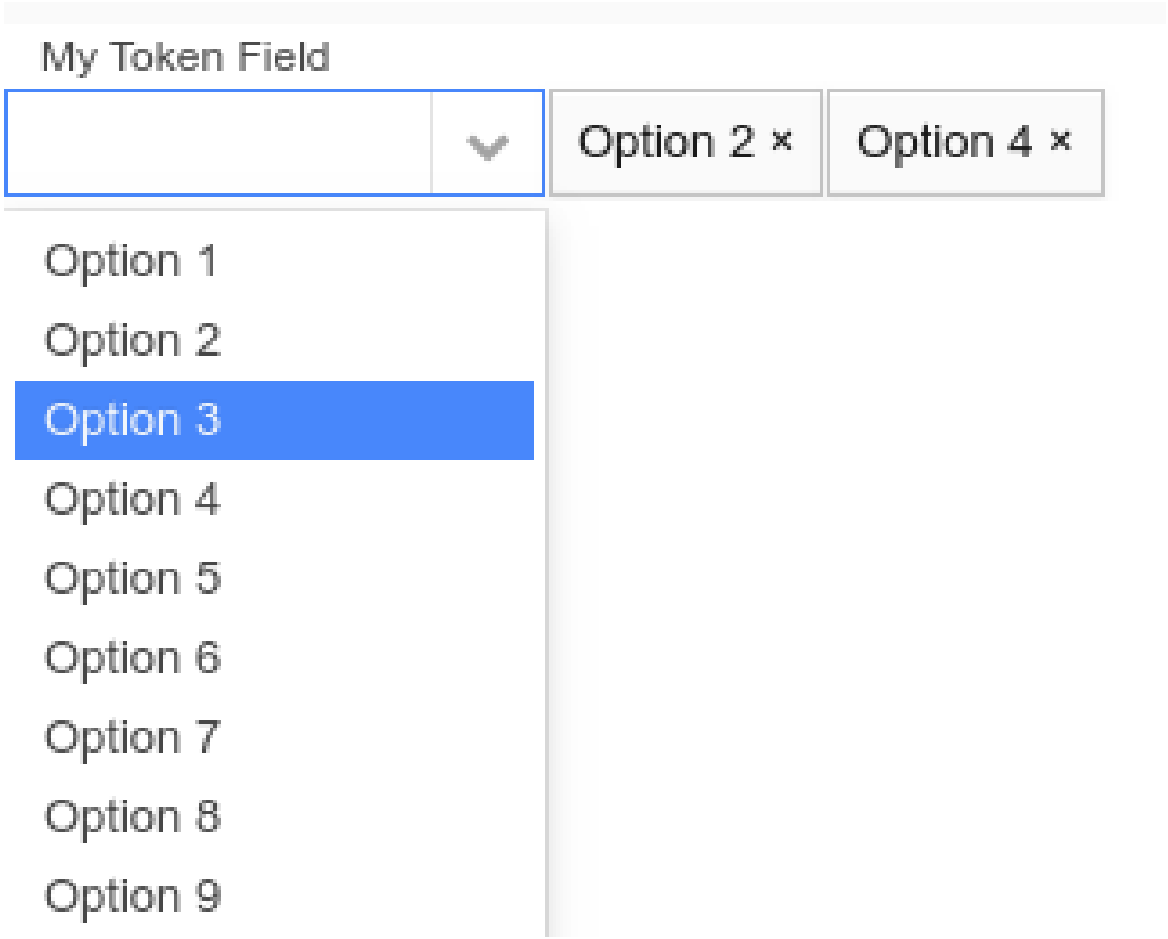


8.2.4.5.9 Token Field (forms::TokenField)

The Token Field component () serves for selection of multiple options. It is rendered as a combo box by default with the options displayed based on users input in its drop-down box: the component searches for the options with labels that contain the entered characters. The user then selects the option by clicking the proposed option or entering the entire values and hitting Enter. They can remove the selected option by clicking the X icon in the option. 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 calling its `addStyleName("tokentextfield")` method.

By default, selected Tokens are no longer proposed in the drop-down menu. You can deactivate and activate this behavior with the `setExcludeSelectedTokens(Boolean excludeSelectedTokens)` method and check the current setting with the `isExcludeSelectedTokens()` method.



Example expression that returns a token field

```
def TokenField tf:= new TokenField("");
tf.setOptions([ "value1" -> "displayed option 1", "value2" -> "displayed option 2"]);
tf.setBinding(&selectedItems);
//tf.setOnChangeListener( { _-> select.refresh() });
tf;
```

8.2.4.6 Tree (forms::Tree)

> **Important:** In the Vaadin 8 implementation, this component is not supported. For more information, refer to information on Vaadin 8 upgrade.

The Tree () component serves as a picker of one item from an expandable tree hierarchy.

The tree hierarchy is defined by the data source and what is displayed as node labels for the tree objects is defined by the caption provider.

- Data source can be of the following types:
  - ClosureTreeDataSource: the input object of the closure is the clicked node and it returns a list of objects that represent the data source;
    Example ClosureTreeDataSource:

```
new ClosureTreeDataSource(
  { node:Object ->
       def List<Object> result;
       if node == null then
         result := getAllApplicants();
       else
         if (node.getType() == Applicant and (node as Applicant).address != null) then
           result := [(node as Applicant).address];
         else
           result := [];
         end;
       end;
     result;
  }
)
```

  - SimpleTwoLevelTreeDS: two-level tree
    Example SimpleTwoLevelTreeDS data source:

```
    def SimpleTwoLevelTreeDS tree := new SimpleTwoLevelTreeDS(
       [ "eva" -> ["climbing", "painting"], "juraj" -> ["reposing", "windsurfing"] ]
    )
```

    The tree identifies the node based on designated IDs. Therefore an object can appear multiple times in a tree. The `select(<VALUE>)` call on the tree, selects the first occurrence of the object.

- Caption provider can be of the following types:
  - IdentityValueProvider: the current row
  - ConstantValueProvider: node labels are constant values

```
new ConstantValueProvider("Constant node label")
```

  - ClosureValueProvider: closure that returns the labels for every node type including null value

```
new ClosureValueProvider(
  { x:Object ->
    if typeOf(x) == type(Applicant) then
       (x as Applicant).name + (x as Applicant).surname;
    else
      if x != null then
       (x as Address).country;
      else
        null
      end
    end
  }
)
```

Where the selected value is stored is defined by the *Binding* property. To set and get the selected value programatically, use the setValue() and getValue() methods.

Setting a selected node in a Tree component

```
//Data source:
//def SimpleTwoLevelTreeDS tree := new SimpleTwoLevelTreeDS(
//      [ "eva" -> [climbVar, "painting"], "juraj" -> [ "windsurfing"] ]
//   )
//setting climbVar as the selected value in the tree:
c.setValue(climbVar);
```

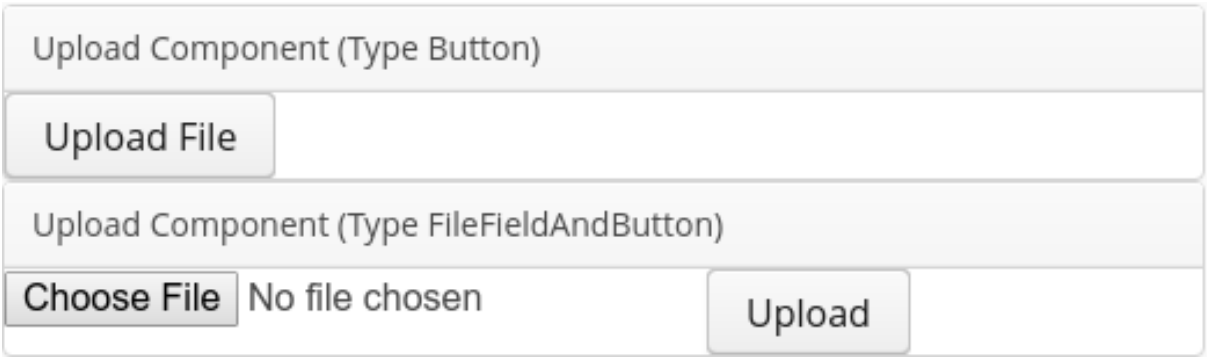
8.2.4.7 Upload (forms::Upload)

The *Upload* component  allows the user to upload a file to the server. Depending on its *field style* setting, it renders either as an upload button or as an input field for a file, a **Browse** and an upload button; Note that a file selected for download is not saved when calling `Forms.save()`.

You can define the action performed when the file has been uploaded with the `setUploadResultListener()` method.

Example expression that returns an Upload component

```
 def Upload up := new Upload();
//only presets the file type in the Browse popup:
up.setAcceptedMimeTypes("text/plain");
//render the upload button AND a "Choose File" button:
up.setFieldStyle(UploadFieldStyle.FileFieldAndButton);
//perform this action when the file is uploaded:
up.setUploadResultListener(
   { f:File, m:String ->
      if f==null then
      uploadOutcomeMessage.setValue("upload failed")
      else
        uploadedFile := f;
        uploadOutcomeMessage.setValue("upload success");
      end
   }
);
up
```



8.2.4.8 Multi-File Upload (forms::MultiFileUpload)

The *Multi-File Upload* component  allows the user to upload multiple files at once.

You can define the action performed when the file has been uploaded with the `setAllFilesUploaded↩Listener()` method.

Example expression that returns an Upload component

```
 def MultiFileUpload mfu := new MultiFileUpload();
mfu.setAllFilesUploadedListener(
   { e:AllFileUploadsDone ->
      if e.files==null then
          uploadOutcomeMessage.setValue("upload failed")
      else
          uploadOutcomeMessage.setValue("upload success");
      end
   }
);
mfu
```

8.2.4.9 Decision Table Editor

The *Decision Table Editor* component  allows the user to view and modify a decision table on runtime from the Process Application.

You can save the underlying decision table object with the `save()` method call and load it with the `load()` call, possibly called from a listener. Like this, your users can promptly adapt the rules used by already running model instances and adjust the execution in steps when the decision table is used to evaluate a rule.

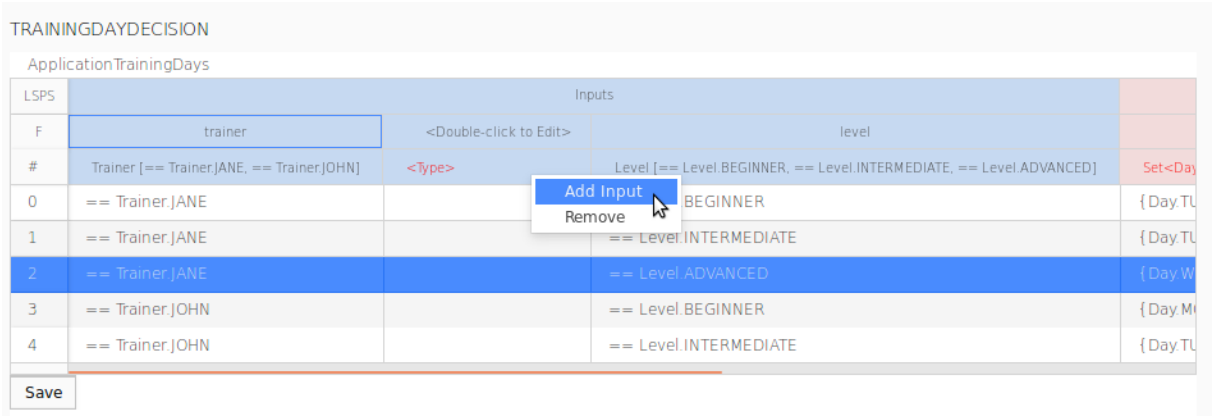
> **Important:** The *Decision Table Editor* component is part of the `dmn` module of the Standard Library unlike the other components, which are provided by the `forms` module. Make sure to import the `dmn` module into your module before you use the component.

To add a decision table editor to a form, do the following:

1. Insert the Decision Table Editor component into the form.
2. Set the component Properties:
   - Decision Table: the instance of the decision table it will use
   - Rights: the rights of the Process Application user over the decision table
     By default, decision tables are read-only. To allow the user to work with the component freely, set Rights to `[DecisionTableRights.CAN_EVERYTHING]`. If the table is not read-only, make sure to save the changes to the table object with the `<DecisionTableObject>.save(<StringI↩DOfTheTableObject>)` method. Consider to attempt to load the decision table object, in case it already exists, as part of the form initialization, for example, in its constructor:

```
public TrainingDaysForm() {
    //the form has the atd local variable of the type of the TrainingDays decision ta
     atd := new TrainingDays(false);
     //try to load it; if it is not loaded, create it and save it:
     if (atd.load("TrainingDaysDecisionTable") == false) then
       atd := new TrainingDays(true);
       atd.save("TrainingDaysDecisionTable");
     end;
}
```

When rendered in the Process Application, an editable Decision Table Editor is edited similarly to its definition. Note that if you want to edit the rule description, single-click the description cell and the double-click the area displayed below.



On information on how to design decision tables and use the Decision Table Editor, refer to the Decision Table chapter in the Modeling documentation or the tutorial on decision tables.

8.2.4.10 Slider (forms::Slider)

The Slider component serves to take a single input value of type Decimal from a value range. It is rendered as a bar with movable handle, with the bar representing the range and the handle the input value.

The bar range is set with the `setMin()` and `setMax()` methods. If unset, minimum is set to `0` and maximum to `100`. Use `getMin()` and `getMax()` to get the set values.

The number of decimal places that the value is set to is set with the `setDecimalPlaces()` method.

The slider can be rendered horizontally or vertically: set the orientation with the `setOrientation()` method; for example, `c.setOrientation(Orientation.Vertical)`

Example expression that returns a Slider component

```
def Slider sl := new Slider("Value Select");
sl.setBinding(&decVar);
sl.setMin(0);
sl.setMax(10);
sl.setDecimalPlaces(2);
sl.setOrientation(Orientation.Vertical);
sl.setOnChangeListener( { _->select.refresh()});
sl
```

8.3 Output Components

Output components serve to display data defined by their binding properties:

- Label (forms::Label)
- Data Source Components
- Charts
- Other Output Components

8.3.1 Label (forms::Label)

The *Label* component () displays read-only text of arbitrary length.

To define how the label content is rendered, call the *setContentMode()* method with the ContentMode enumeration literal. The literal can be set to:

- HTML renders HTML code
- preformatted preserves newlines
- or plain text.

Important: When using the *HTML* content mode make sure no injection and XSS vulnerabilities can be exploited.

```
Ay &varTime

Ay now()

Ay now().toString()

"Refresh"

(...) def Label l := new Label();
      l.setBinding(new ClosureBinding({->"The current <b>time</b> is " + now()}));
      l.setContentHtml(true);
      l.setCaption("Clock");
      l;
```

Figure 8.38 Form with a Label

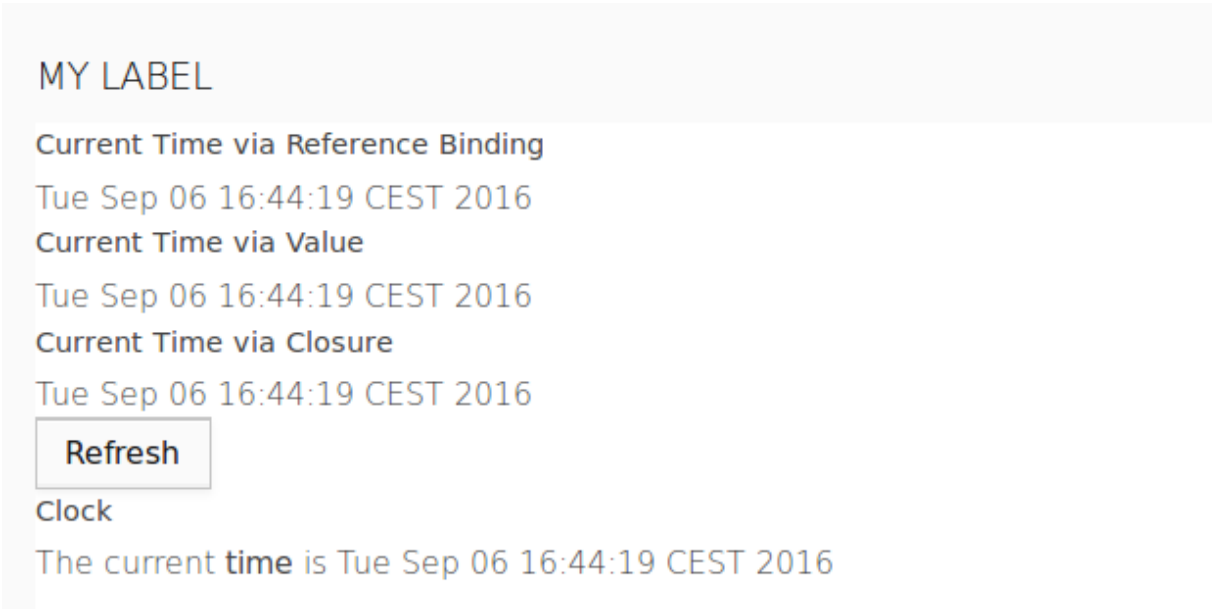
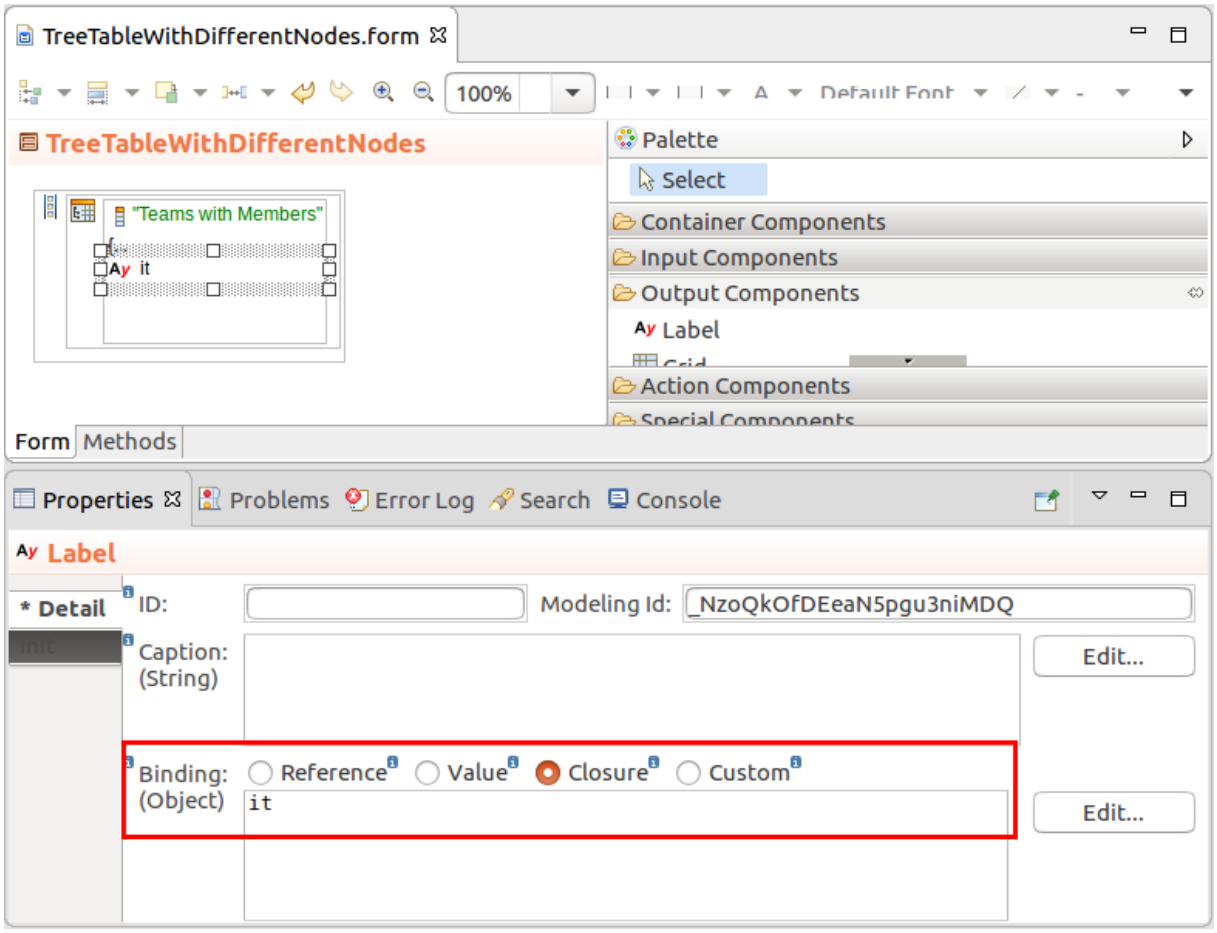


Figure 8.39 Form with Labels above rendered

Expression that returns a Label

```
def Label l := new Label();
l.setBinding(new ClosureBinding({->"The current <b>time</b> is " + now()}));
l.setContentMode(ContentMode.Html);
l.setCaption("Clock");
l;
```

> **Note:** When defining closure binding in the Properties view, define only the body of the closure as the property value.



8.3.2 Data Source Components

Data Source components operate over collections of items, such as, Lists of Records, Shared Records, Set of Strings, etc. and include Table (forms::Table), Grid (forms::Grid), Tree Table (forms::TreeTable), and Repeater (forms::Repeater)

They allow you to visualize data items of such collections in a regular pattern. How the data is obtained is defined by the *data source* of the component.

With the exception of the Tree Table data source, which requires a data source of types specifically required by the component, the data sources can be defined as:

- **Type**: shared Record type (all instances of the shared Record are obtained and used as the data source)

- **Query**: `query call`

  Note: If the query uses paging, the paging is ignored and the paging of the component is used instead.

- **Collection**: collection of data items The Collection is calculated only when the form is initialized and remains unchanged on refresh: To reload the data of a collection data source, use the `setDataSource()` method on the component.

  Note: When applying filtering and sorting on large collections, sorting and filtering actions might cause performance issues: consider using other data sources such as shared Type data sources for large data sets.

- **Custom**: custom data source which implements the *forms::DataSource* interface. The interface requires implementation of the following methods:

  - `getCount()` returns the number of displayed entries; note that you need to take the filters applied on the data into account. The filters are passed as input arguments.
  - `getData()` returns the data for the table; note that depending on your requirements, the method will need to filter, sort, and page data;
  - `supportFilter()` called when the user enters the filter value; The ID of the filters in individual columns is set in the FilterConfig: `new FilterConfig(filterId -> "nameColumn↩Filter")`
  - `supportSort()` called when the user sorts the sort value;

The data items are then used by the *component items*, for example, rows, as their sources of data. Data items are loaded lazily to allow reasonable performance when working with large data sets.

8.3.2.1 Table (forms::Table)

Important: The Table component is deprecated. Please, use the Grid component.

Important: In the Vaadin 8 implementation, this component is not supported. For more information, refer to information on Vaadin 8 upgrade.

The *Table* () component displays the data of the data source in a Table with every data item in a single row.

Note that Table columns can contain other components: while this option provides a high level of flexibility, Tables with many components in their columns might perform poorly. If you are displaying a lot of data in a Table, consider using Grid.

Tables support sorting and filtering out-of-the-box.

8.3.2.1.1 Designing a Table

To design a Table, do the following:

1. Insert Table into your Form.
2. Define the data-source type and the expression that returns it.
3. Insert the *Table Column* component () into Table and define its properties:

(a) Select the type of the value provider and its value: it returns the content of the cell based on the object of the row (Identity will serve the unchanged data object).

(b) Set the iterator type to the type of the value provider:

The iterator type must be always identical with the type of the Value Provider object: the user needs to set the iterator type explicitly since it is not always possible to infer the type of Value Provider object (for example, if the Value Provider is defined by a closure).

(c) Insert the form components into Table Column and define their properties.

(d) Insert components into the column: they will use the table iterator (*it* by default) to work with the iterated object).

4. Insert further columns and define their properties.

8.3.2.1.2 Designing a Table Column

To create a column in your Table, do the following:

1. Insert the *Table Column* component () into Table.
2. Select the type of the value provider, which will prepare the data item (Identity will serve the unchanged data object).
3. Set the iterator type to the type of the value provider:

   The iterator type must be always identical with the type of the Value Provider object: the user needs to set the iterator type explicitly since it is not always possible to infer the type of Value Provider object (for example, if the Value Provider is defined by a closure).
4. Insert the form components into Table Column and define their properties.
5. Insert components into the column: they will use the table iterator (*it* by default) to work with the iterated object).

Alternatively, you can add a column programatically with the `addColumn()` method of Table. This allows you to add columns dynamically on runtime, for example, on button click.

```
{e -> Vocab.addColumn(
   new forms::TableColumn(
       data -> null,
       modelingId -> null,
       filtrable -> false,
       generator -> null,
       sortable -> false,
       valueProvider -> new PropertyPathValueProvider(Unit.svk))
   )
}
```

8.3.2.1.2.1 Collapsing a Table Column

Columns of tabular components can be collapsed from the front-end by the user as well as programatically. You can also enable or disable the feature.

By default, Columns are collapsible so to collapse or expand a Column, call the `setCollapsed(Boolean)` method on the Column.

To disable collapsibility of a Column, call the `setCollapsible(false)` method on the Column.

8.3.2.1.2.2 Ordering Table Columns

To change the order of table columns, call the `setColumnOrder()` method with the column ordered as required.

```
//restoring column order from an listener:
{ _ ->
  def List<TableColumn> allColumns := myTable.getColumns();
  myTable.setColumnOrder(
       allColumns.sort({
       a:TableColumn, b:TableColumn ->
         switch a.getHeader() > b.getHeader()
           case true -> 1
           case false -> -1
           default -> -1
         end
       })
  );
}
```

8.3.2.1.2.3 Drag-and-Drop of Table Columns

To allow drag-and-drop of table columns, use the `setColumnReorderingAllowed(true)` table call.

8.3.2.1.3 Sorting a Table

Sorting is defined per Table Column: the data is sorted depending on which column the user calls the sorting. To enable it, do the following:

1. Open the column properties and select the Sorting tab.
2. Select the sortable flag on the Sorting tab.
3. If you do not want to use the column provider for sorting, define the sorting value provider below.

Alternatively, you can call the `setSorted()` method on the table column.

You can check the status of the column sorting with the `isSorted()` call.

> **Important:** When applying sorting on large collections (the Data Source is set to Collection), sorting actions might cause performance issues: consider using other data source types, such as shared records.

8.3.2.1.4 Filtering a Table

Table filtering is defined per Table Column: the data is filtered depending on which column the user enters the filter. Note that only filtering of Boolean, String, Integer, Decimal, Date, or Enumeration values is supported by the table filter.

> **Important:** Filters on Columns of Tables support only the Boolean, String, Integer, Decimal, Date, and Enumeration data types.

To enable filtering on a Table Column, do the following in the Properties view of Table Column:

1. On the Detail tab, define the *iterator* data type at the bottom of the Properties view to the type returned by the value provider.
2. On the Filtering tab, define the following:
   (a) Select the **Filterable** option.
   (b) Define *Filter value provider* so it returns the data for the filter:
   - *None*: the column value provider is used
     Make sure that the provider expression on the **Detail** tab is set to return the same data type as the iterator data type.

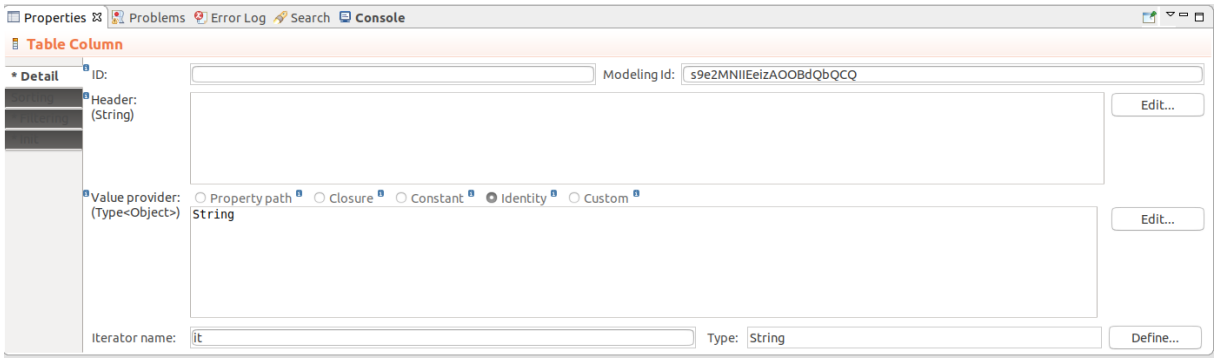


Figure 8.40 Setting of column value provider for None Filter value provider

- *Property path*: the property path to the property of the table object, for example, `Patient.diagnosis.code`.
- *Closure*: a closure that returns the value the filter receives, for example, `{ p:Patient -> p.reports.size() }` will add an integer filter for the number of reports of the Patients in the table.
- *Identity*: the identity passed from the row
  Make sure to enter the type of the identity below, such as, String, Integer, Decimal, Date, etc.

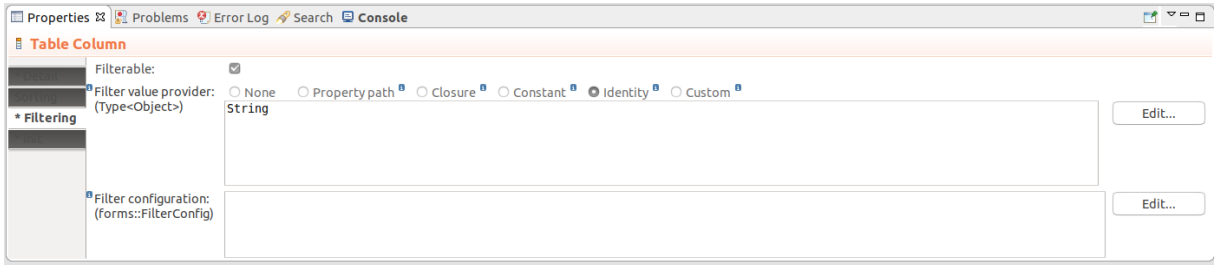


Figure 8.41 Identity filter setting

- *Custom*: custom implementation of ValueProvider

(c) Optionally define the filter configuration: The configuration allows you to further specify the filter properties: You can set it also with the `setFilterConfig()` method. Also if you are designing a `table with a custom Data Source`, it will allow you to identify the filters with set values using the filter id.

- *OptionsFilterConfig*: the user will be able to select one or multiple OptionsFilterConfig from an option set, which will be interpreted as the values and applied by the filter.

```
new OptionsFilterConfig(multiselect -> true, options -> [
  new SelectItem(label -> "label of 1", value -> 1),
  new SelectItem(label -> "label of 2", value -> 2)
])
```

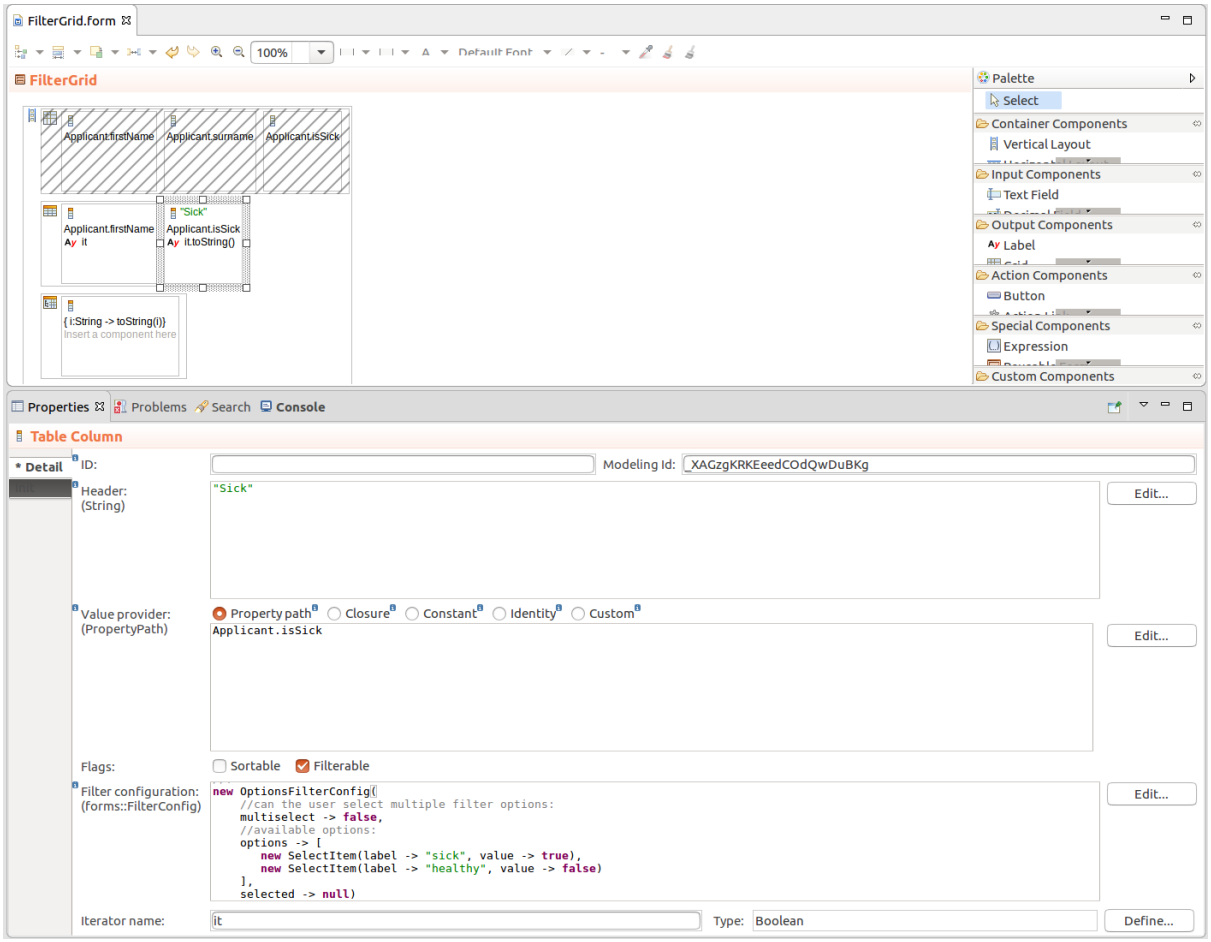
- *NumericFilterConfig*: the user will define a number range

```
// numeric filter with default values:
new NumericFilterConfig(equal ->50, moreThan -> 10, lessThan -> 90)
```

- *DateFilterConfig* so the user can define a date range to use for filtering and the format of selected dates

```
new DateFilterConfig(
  resolution -> uicommon::DateTimeResolution.Month,
  formatPattern -> "YYYY MMMM",
  lessThan -> now()+years(3),
  moreThan -> now()-years(3)
)
```

- *RexExpFilterConfig* to define the default regex pattern for filtering
- *SubStringFilterConfig* to define the default substring pattern for filtering
- *BooleanFilterConfig* to define properties of filter for Boolean values



3. Make sure the table is filterable: the flag is set with the `setFilteringEnabled(true)` method and is enabled by default.

Important: When applying filtering on large Collections (the Data Source is set to Collection), sorting and filtering actions might cause performance issues: consider using other data sources such as shared Type data sources.

8.3.2.1.4.1 Enabling and Disabling Filtering on Tables

To enable or disable filtering on Tables dynamically, use the `setFilteringEnabled(Boolean)` call on the component: The method hides or displays the filtering row.

8.3.2.1.5 Selecting Table Rows

To enable row selection on a Table, Grid, or Tree Table, set the table as selectable: `MyTable.isSelectable()` or `MyTable.setSelectable(true)`. You can then use the `setSelectionChangeListener()` to listen for user Selection and acquire the selected row with use the `getSelection()` method.

```
c.setSelectable(true);
c.setSelectionChangeListener({ e -> selected := c.getSelection().toString(); SelectedLabel.refresh()})
```

Use the `select()` method to select a row programatically.

Example table-row select that selects the user who was created first in the table

```
myUserTable.select(
    myUserTable.getDataSource().getData(
       startIndex -> 0,
       count -> 0,
       filters -> null,
       sortSpecs -> {new Sort(User.created, false)}).getFirst()
);
```

8.3.2.1.6 Getting Table Cell Content

To get the components inside a table cell, use the `TableColumn.getComponentInRow()` method.

Example usage of the getComponentInRow method

```
def FormComponent cellContent := myTableColumn.getComponentInRow(it);
cellContent.refresh();
```

8.3.2.1.7 Setting Batch Load Size of Table Data

To define the size of the data batch of a table, use the `setPageLength()` method: the data is loaded in these batches as you scroll down the table.

```
c.setPageLength(3)
```

If Table height is not set or is set to wrap, it adopt the page length height.

If you want to always adapt the height of Table to its content, you can query the data source of Table for its size, for example:

```
c.setPageLength(c.getDataSource().getCount([]))
```

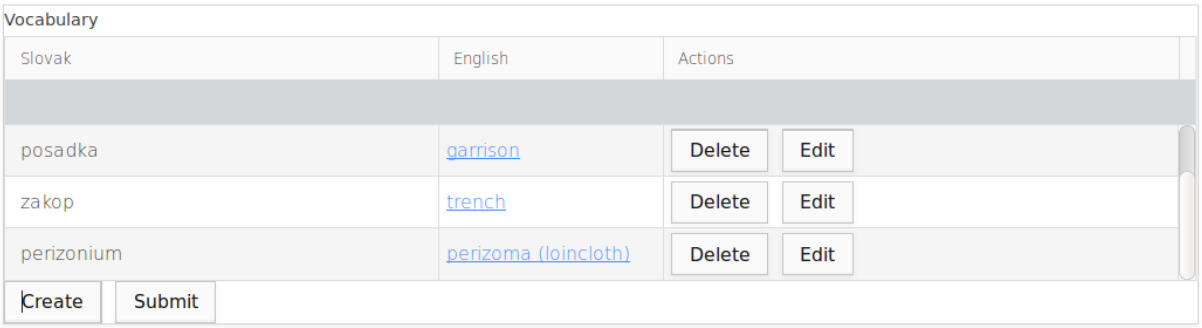


Figure 8.42 Grid with a scroll (the height is set to 3 rows and the data source returns 15 row objects

8.3.2.1.8 Setting No-Data Message

To set the message that is displayed in Table when it is empty, call the `setNoDataMessage(<String>)` method.

8.3.2.1.9 Setting CSS Style on Table Rows

To add a style to table rows, call the `setRowStyleGenerator()` method on your table: the method had a closure input parameter with the row object as its input and the CSS class that is added to the row /

element as its return value.

Note that the css class is automatically prefixed with `v-table-row-` unlike in grid where the class name is passed as is.

```
//"label" will become "v-table-row-":
c.setRowStyleGenerator({a:Applicant -> "label"});
```

To remove the previously added styles, set the closure to return `null`, for example, `table.setRowStyle↩Generator({a:Applicant -> null});`.

8.3.2.1.10 Defining Content Align on a Table Column

To define horizontal alignment in Table Columns, call the `setAlignment()` method on the column.

8.3.2.2 Grid (forms::Grid)

The *Grid* component displays single-line text data in a tabular layout and enables editing of persistent data and row selection.

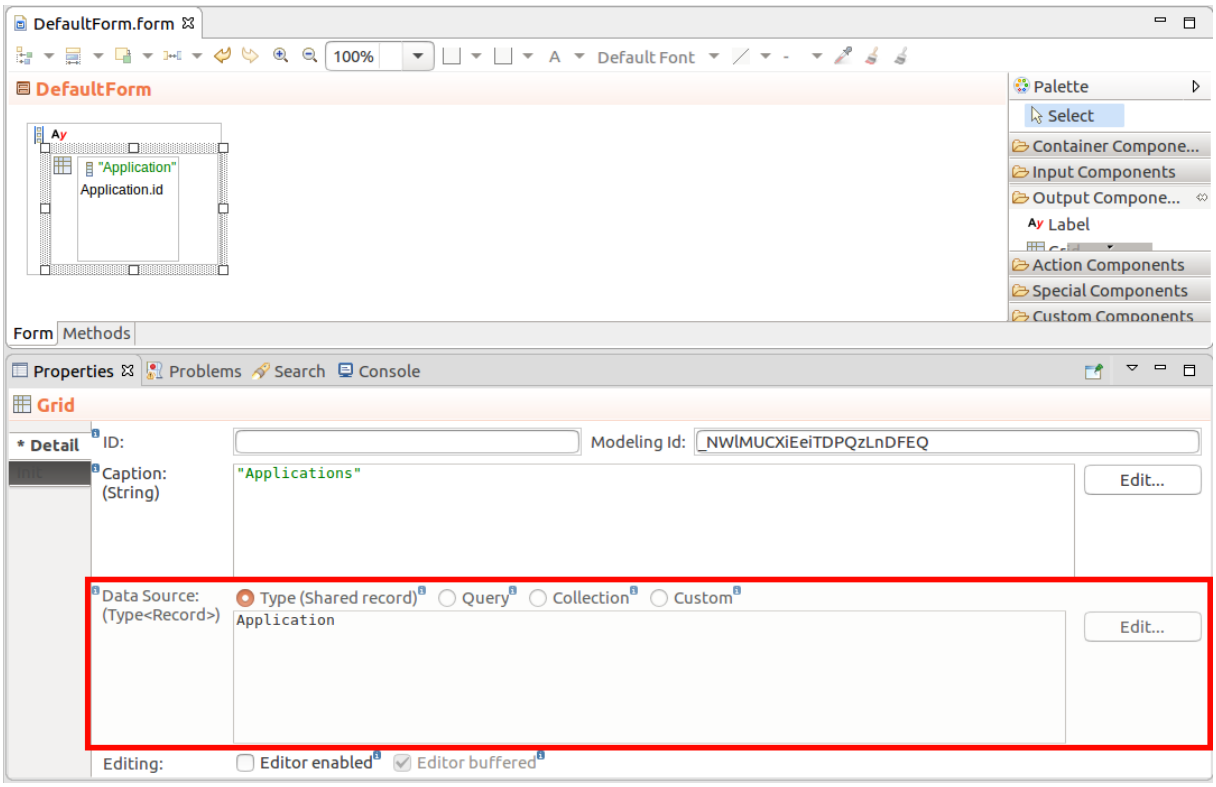
The data is lazy-loaded, which prevents potential performance issues when the grid is loaded.

Grid cells cannot hold further components. However, you can modify how the column content is rendered and render it, for example, as a button or a link, using renderers.

8.3.2.2.1 Designing a Grid

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

1. Insert the Grid component into your Form.
2. Define the data-source type and the expression that returns it.



3. Insert the Grid Column components and define their properties ( ) into the Grid. You can insert grid columns dynamically with the `addColumn()` method.

8.3.2.2.2 Creating a Grid Column

To create a column in your Grid, do the following:

1. Insert the *Grid Column* component ( ) into the Grid.
2. Select the type of the value provider, which return the column item:
   - `Property path`: path to the property of the record used as the Grid's data source, for example, `Person.id`
   - `Closure`: closure that returns the column item with the grid data object as its input argument, for example, `{ s:SavedDocument -> toString(s.id)}`
     Note that the renderer of the column (defined below) has the input argument of the closure as its input.
   - `Identity`: unchanged row object
   - `Custom`: custom implementation of *forms::ValueProvider*
3. Define the renderer (determines how the value from the value provider is rendered):
   - **None** renders the value as returned by the `toString()` call.
   - **HTML** renders a String value as HTML.
   - **Number** renders the value in the format defined below.
     Define the format as a String following the `DecimalFormat Java formatting rules`, for example, `"0000.00000"`

- **Date** renders the value in the format defined below.

  Define the format as a String following the SimpleDateFormat Java formatting rules in the text area below, for example, `"EEE, d MMM yyyy HH:mm:ss"` results in formatting `Wed, 7 Sep 2016 14:33:00`

- **LocalDate**: renders the Value Provider value in the defined format

  Define the format as a String following the SimpleDateFormat Java formatting rules in the text area below, for example, `"EEE, d MMM yyyy"` will result in formatting like `Wed, 7 Sep 2016`

- **Enumeration** renders an Enumeration value as a String.

- **Button** renders the value as a Button.

  The click action is defined as a closure with the value-provider object as its input parameter below.

```
{ clickRowObject:String ->
    varString := "The user clicked: " + clickRowObject.toString();
    MyEditableGrid.refresh()}
```

- **Link** renders the value provided by the Value Provider as a Link

  The click action of the link is defined as a closure in the field below. The input of the closure depends on the type of the value provider; for Property path, Constant, and Identity, the input is the return value of the closure; for Closure, the input is the row object:

```
{  clickRowObject:String ->
    varString := "The user clicked: " + clickRowObject.toString();
    MyEditableGrid.refresh()}
```

- **Theme image** considers the value to be the path to a Vaadin ThemeResource image and renders the image. 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/VAADIN/themes/lsps-blue", so the String path could be "favicon.png". The on-click action is defined as a closure with the Value Provider object as its input parameter below.

- **External image** considers the value to be the URL to an image and renders the image. The Value Provider must return a String with the URL to the image. For example, *external image renderer* could be used if for a Closure Value Provider set to `{a:Applicant -> "www.acme.com/images/" + a.pictureName}`.

- **Component** considers the value to be a form component.

  > **Important:** The component with its subtree components is generated for each column cell. In the case the component tree is complex, this can result in performance issues; Therefore, using the Component renderer is discouraged and other renderers should be used preferably whenever possible. It is provided primarily to allow full migration of Tables to Grids.

- **Custom** uses a custom renderer that implements the *forms::Renderer* interface (For further instructions, refer to the developer documentation.

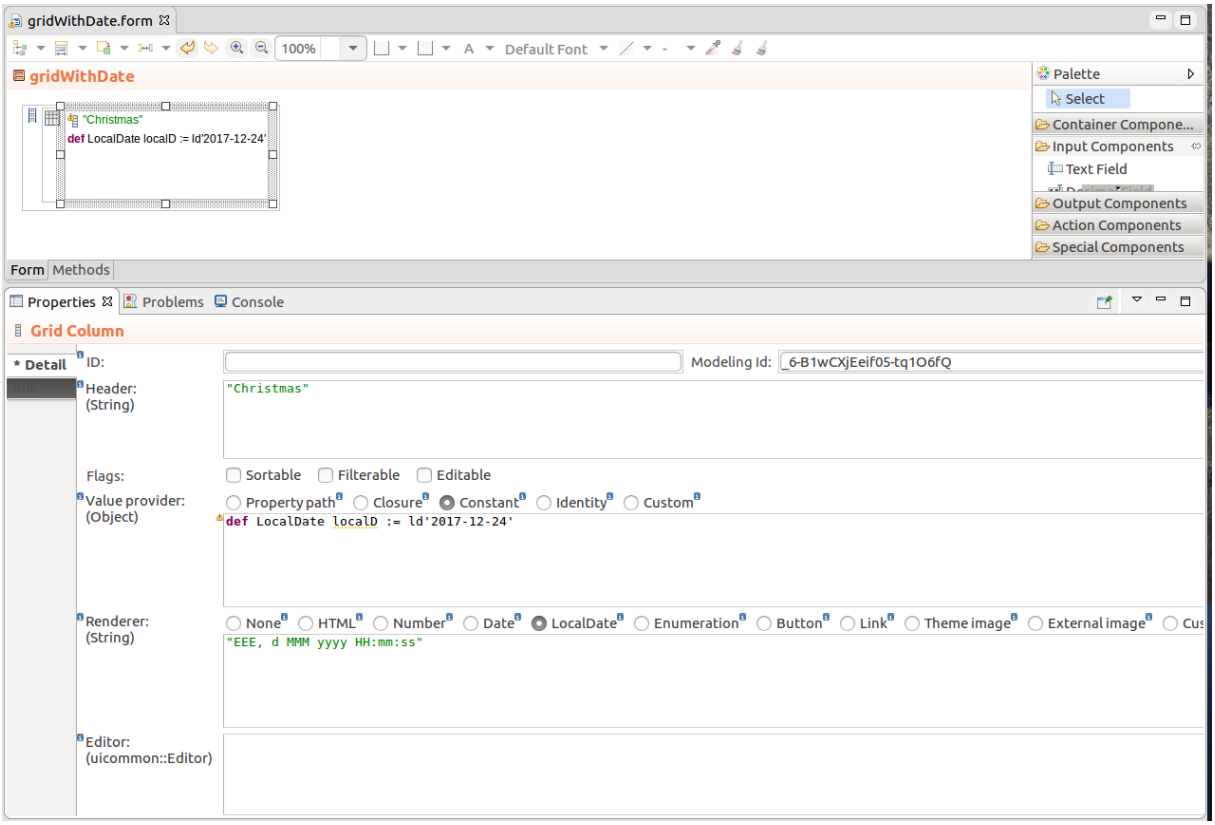


Figure 8.43 Theme image renderer

8.3.2.2.1 Creating a Grid Column Dynamically

To add a column dynamically on runtime, for example, when the user clicks a button, call the `addColumn()` method on the parent component.

```
{e -> Vocab.addColumn(
    new forms::TableColumn(
        data -> null,
        modelingId -> null,
        filtrable -> false,
        generator -> null,
        sortable -> false,
        valueProvider -> new PropertyPathValueProvider(Unit.svk))
    )
}
```

8.3.2.2.2 Hiding a Grid Column

You can hide columns of tabular components can be collapsed from the front-end by the user as well as programatically. You can also enable or disable the feature.

By default, Columns can be displayed:

- To disable hiding of a Column, call the `setHiddable(false)` method on the Column.
- To hide and show a Grid Column, call the `setHidden()` method on the Column.

8.3.2.2.2.3 Ordering Grid Columns

To change the order of grid columns, call the `setColumnOrder()` method on the grid column ordered as required.

```
//restoring column order from an listener:
{ _ ->
  def List<GridColumn> allColumns := myGrid.getColumns();
  myGrid.setColumnOrder(
       allColumns.sort({
       a:GridColumn, b:GridColumn ->
         switch a.getHeader() > b.getHeader()
           case true -> 1
           case false -> -1
           default -> -1
         end
       })
  );
}
```

8.3.2.2.2.4 Enabling Drag-and-Drop of Grid Columns

To allow drag-and-drop of grid columns, use the `setColumnReorderingAllowed(true)` call.

8.3.2.2.3 Sorting a Grid

Sorting is defined per Grid Column. To enable it, do the following:

1. Open the column properties and select the Detail tab.
2. Select the sortable flag.

Alternatively, you can sort based on a grid column with the `setSorted()` method. The method ignores the setting of the *Sortable* option of grid columns.

```
//sorting myColumn in descending order:
myColumn.setSorted(true, false);
```

You can check the status of the column sorting with the `isSorted()` and `isSortAscending()` call.

> **Important:** When applying sorting on large collections (the Data Source is set to Collection), sorting actions might cause performance issues: consider using other data sources such as shared Type data sources.

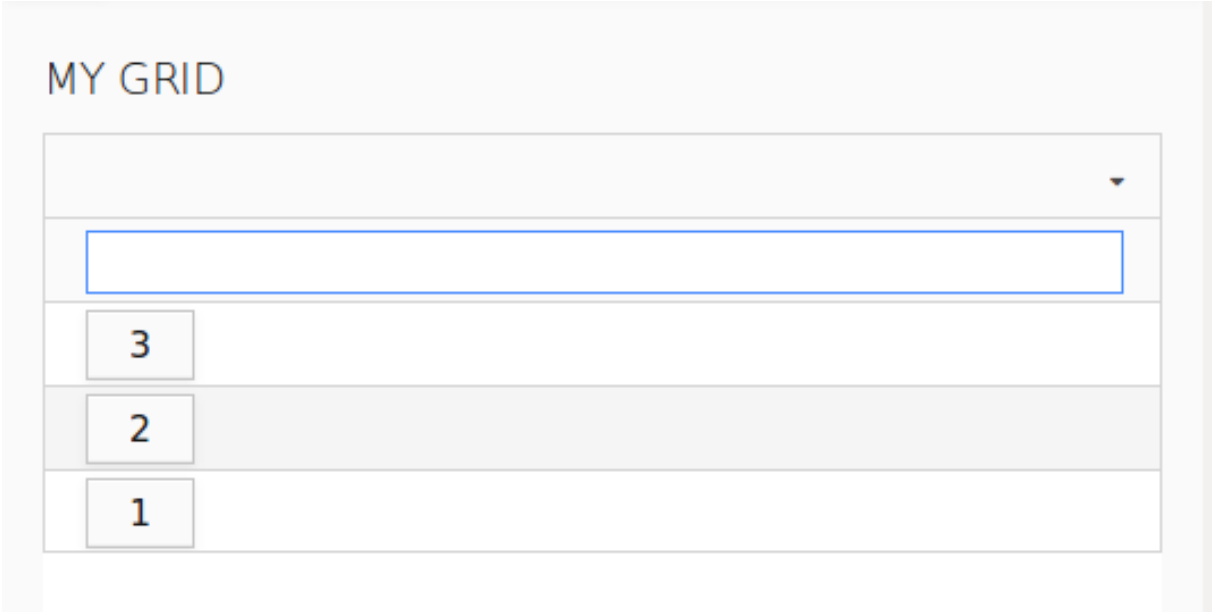


Figure 8.44 Filterable and sortable Grid Column

8.3.2.2.4 Filtering a Grid

Important: Filters on Columns of Grids support only the Boolean, String, Integer, Decimal, Date, and Enumeration data types.

To enable filtering on a Grid column, do the following:

1. Make sure the Grid is filterable: the flag is set with the `setFilterable(true)` method and is enabled by default.
2. On the respective column, set the filtering properties:
   (a) On the Detail tab, select the *Filterable* flag.
   (b) Optionally specify the filter properties with the `setFilterConfig()` method.
   - *OptionsFilterConfig*: the user will be able to select one or multiple OptionsFilterConfig from an option set, which will be interpreted as the values and applied by the filter.

```
def SelectItem adv :=  new SelectItem(label -> "Advanced", value -> Level.ADVANCED);
def SelectItem int :=  new SelectItem(label -> "Intermediate", value -> Level.INTERMEI
def SelectItem beg :=  new SelectItem(label -> "Beginner", value -> Level.BEGINNER);
~
c.setFilterConfig(
   new OptionsFilterConfig(
      multiselect -> true,
      options -> [
          adv, int, beg
      ],
      selected -> [adv, int]
      )
)
```

   - *NumericFilterConfig*: the user will define a number range

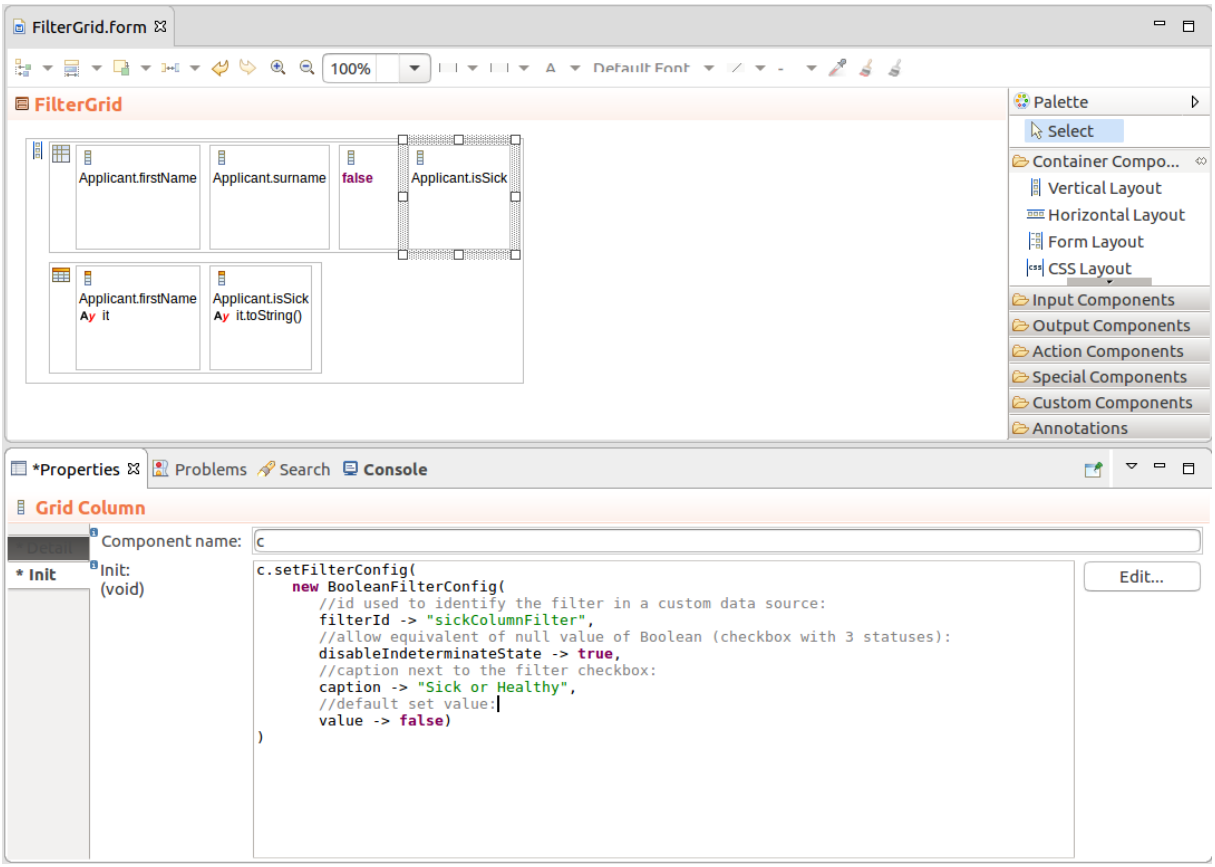
```
// numeric filter with default values:
new NumericFilterConfig(equal ->50, moreThan -> 10, lessThan -> 90)
```

   - *DateFilterConfig* so the user can define a date range to use for filtering and the format of selected dates

```
new DateFilterConfig(
  resolution -> uicommon::DateTimeResolution.Month, formatPattern -> "YYYY MMMM"
)
```

- *RexExpFilterConfig* to define the default regex pattern for filtering
- *SubStringFilterConfig* to define the default substring pattern for filtering
- *BooleanFilterConfig* to define properties of filter for Boolean values

Also if you are designing a `filterable grid with a custom data source`, the filter properties (its id) will allow you to identify the filters with set values using the filter id.



Important: When applying filtering on large Collections (the Data Source is set to Collection), sorting and filtering actions might cause performance issues: consider using other data sources such as shared Type data sources.

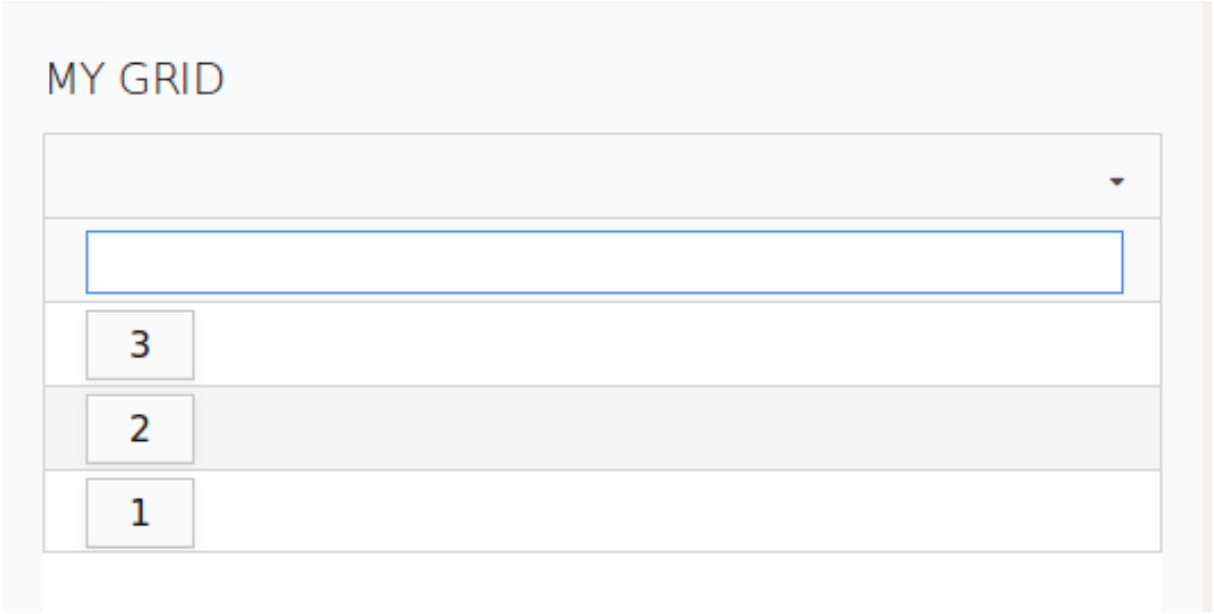


Figure 8.45 Filterable and sortable Grid Column

8.3.2.2.5 Working with a Component in a Grid Row

If you need to perform an action on a cell with a component in a grid row, use the `getComponentInRow()` method: the method returns the component in the cell of the grid column.

```
//Closure value provider expression of a grid column
//which refreshes the value of the row object in another Columns
//whenever it is changed (renderer is set to Component):
{ mr:MyRecord ->
   def TextField tf := new TextField("caption", &mr.field);
   tf.setOnChangeListener({ e ->  myc.getComponentInRow(mr).refresh();myc3.getComponentInRow(mr).r
   tf
}
```

```
{ e ->
  def Object rowObject := myg.getDataSource().getData(0, 1, null, null)[0];
  check.setValue(myGridColumn.getComponentInRow(rowObject))
}
```

8.3.2.2.6 Editing a Grid

Important: When enabling editing of Grid data, make sure to consider the following:

- Since Collections are immutable, you can enable editing only on a Grid with shared Records and Grid Columns with the Value Provider set to *Property path*.
- If the underlying associated shared Record Property returns `null`, the cell will remain empty and read-only.

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

1. Open the Properties view of the Grid component:

(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 value from a drop-down box

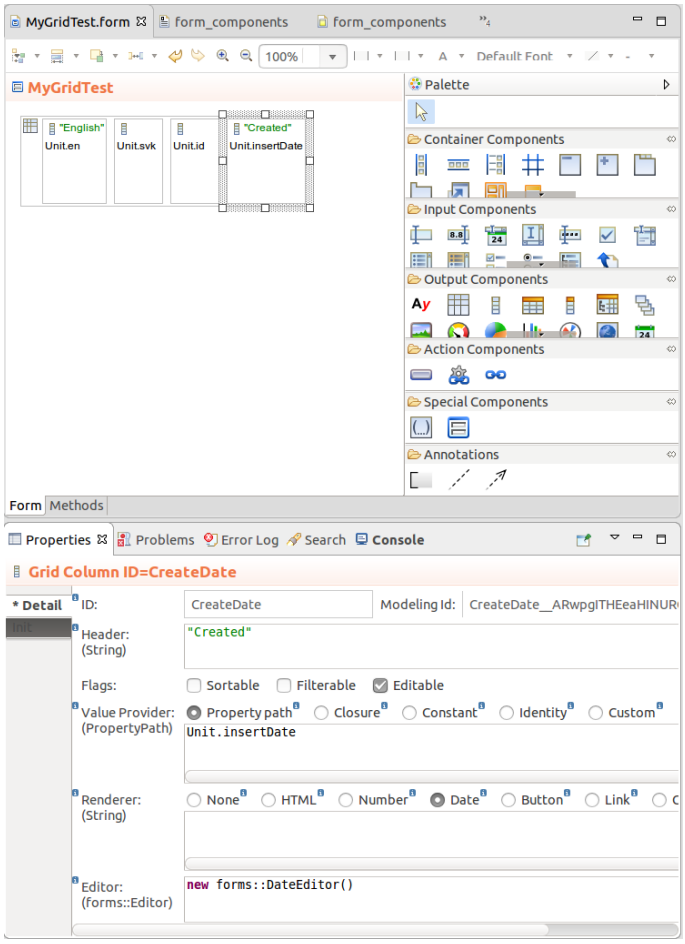


Figure 8.46 Editable Column with a date value

8.3.2.2.7 Enabling Row Selection in a Grid

To allow the user to select grid row and perform an action on row selection, do the following:

1. Make the Grid selectable using the `setSelectable(true)` call.

2. Set the selection mode with the `setSelectionMode(SelectionMode)` call:

- `SelectionMode.NONE`: selection disabled
- `SelectionMode.SINGLE`: single-row selection on click
- `SelectionMode.MULTI`: selection of multiple items using checkboxes in individual rows

  In the MULTI mode, the user can select multiple rows directly under each other by clicking the selection checkbox of the first row, holding the click, and dragging the mouse across the rows below.

3. To handle a selection, call the `setSelectionChangeListener()` method on the Grid component: its closure parameter defines how to handle the selection.

4. To select a Grid row, call the `select()` method.

```
 //on the Init tab of Properties view:
//enable selection:
c.setSelectable(true);
//set selection mode:
c.setSelectionMode(SelectionMode.MULTI);
//select rows with first two items from the datasource:
c.select(c.getDataSource().getData(0, 2, null, null));
~
//you can also select a particular item in a Collection
//datasource (in SINGLE selection mode):
//c.select(mycollection[0]);
~
//display selected items in an output component:
c.setSelectionChangeListener(
  {
    v:forms::ValueChangeEvent ->
      //set the content of an output component to the selected row:
      labelComponent.setValue(
        c.getSelection().toString());
  }
);
```

8.3.2.2.8 Enabling Selection of All Rows

To enable the *Select All* option on a Grid, set the Grid as selectable in MULTI selection mode and display the Select All checkbox:

```
c.setSelectable(true);
c.setSelectionMode(SelectionMode.MULTI);
c.setSelectAllCheckBoxVisible(true);
```

8.3.2.2.8.1 Enabling Selection of Multiple Rows in a Grid

To enable the *Select All* option on a Grid, set the Grid as selectable in MULTI selection mode and display the Select All checkbox:

```
c.setSelectable(true);
c.setSelectionMode(SelectionMode.MULTI);
c.setSelectAllCheckBoxVisible(true);
```

8.3.2.2.9 Displaying Detail Row in the Grid

To display a row with data in a Grid when the user clicks a row, do the following:

1. Create the detail form, which will be displayed in the detail row:
   (a) Create a form definition.
   (b) Create a variable that will hold the object with the detail data, for example, create a form variable of type Person.
   (c) Create a constructor that will have the detail object as its argument and store the object in the variable.

```
PersonDetail {
~
  public PersonDetail(Person p) {
      person := p;
  }
}
```

2. On the Grid, use the `setDetailsGenerator()` method to set the detail form to the grid rows.

```
c.setDetailsGenerator({ p:Person -> (new PersonDetail(p).getWidget()) });
```

3. Send the clicked item to the detail form and toggle the visibility of the detail row.

```
c.setItemClickListener({ event:ItemClickEvent ->
 def Object item := event.item;
 c.setDetailsVisible(item, !c.isDetailsVisible(item));
});
```

8.3.2.2.10 Setting Grid Height

By default, the Grid defines a fixed minimum height. You can change the height using the *setHeightByRows()*, *setHeightFull()*, or *setHeightWrap()* method.

```
c.setHeightByRows(3)
```

If the grid displays more row objects, the Grid height follows the height setting and a scrollbar is added:

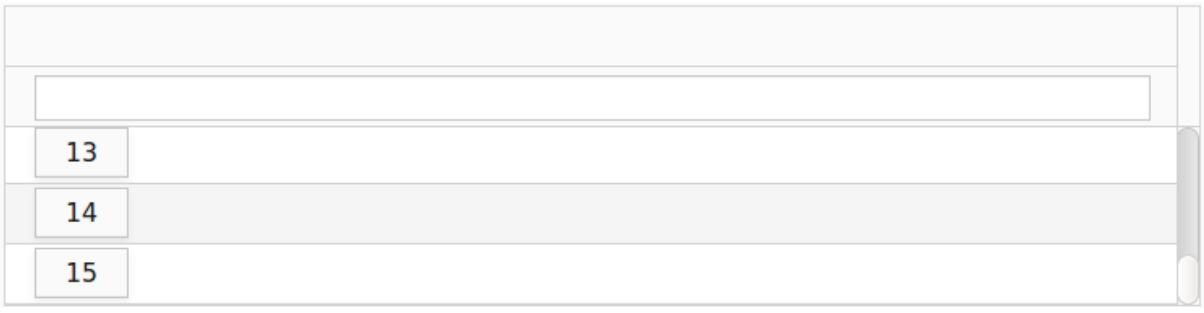


Figure 8.47 Grid with a scroll (the height is set to 3 rows and the data source returns 15 row objects

8.3.2.2.11 Wrapping Grid Rows

If you want to wrap the Grid around its content so as to adapt the height of the Grid to its content, query the data source size and adapt the height accordingly, for example:

```
c.setHeightByRows(c.getDataSource().getCount([]))
```

8.3.2.2.12 Displaying a Tooltip on a Grid

You can define a tooltip

- on the Grid

  ```
  myGrid.setDescription({ d:Date  -> "The <b>date</b> in the row:" + d}, ContentMode.Html)
  ```

- on the Grid Column

  A column tooltip takes precedence over the Grid tooltip.

  ```
  myGridColumn.setDescription({ rowObject:Object  -> "Column tooltip"})
  ```

- on the Grid Column header

  ```
  myGridColumn.setHeaderDescription(
     "Column header:" + #10 + myGridColumn.getHeader(),
     ContentMode.Preformatted)
  ```

  A column tooltip takes precedence over the Grid tooltip.

If the content-mode argument is not passed, all methods use the Preformatted content mode.

8.3.2.2.13 Setting Style Class on Grid Rows

To add a style class to grid rows, call the `setRowStyleGenerator()` method on your grid: the method has a closure parameter with the row object as its input and the CSS class that is added to the row /

element.

```
c.setRowStyleGenerator({a:Applicant -> "v-label"});
```

To remove the previously added styles, set the closure to return `null`, for example, `grid.setRowStyle↩ Generator({a:Applicant -> null});`.

8.3.2.2.14 Setting Style Class on Grid Columns

To add a style class to the cells of a grid column, call the `setCellStyleGenerator()` method on your grid column: the method has a closure parameter with the row object as its input and the class.

```
c.setRowStyleGenerator({a:Applicant -> "v-label"});
```

To remove the previously added styles, set the closure to return `null`, for example, `grid.setCellStyle↩ Generator({a:Applicant -> null});`.

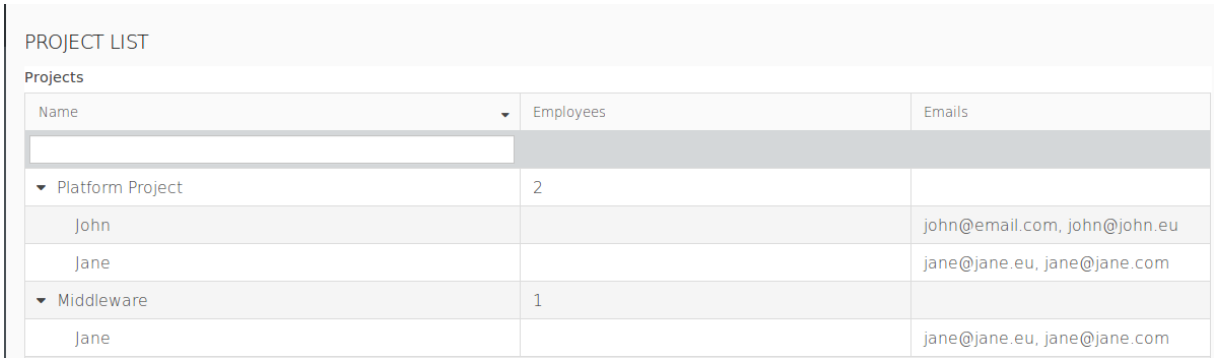
8.3.2.2.15 Defining Content Align on a Grid Column

To define horizontal alignment in Grid Columns, call the `setAlignment()` method on the column.

8.3.2.3 Tree Table (forms::TreeTable)

> **Important:** In the Vaadin 8 implementation, this component is not supported. For more information, refer to information on Vaadin 8 upgrade.

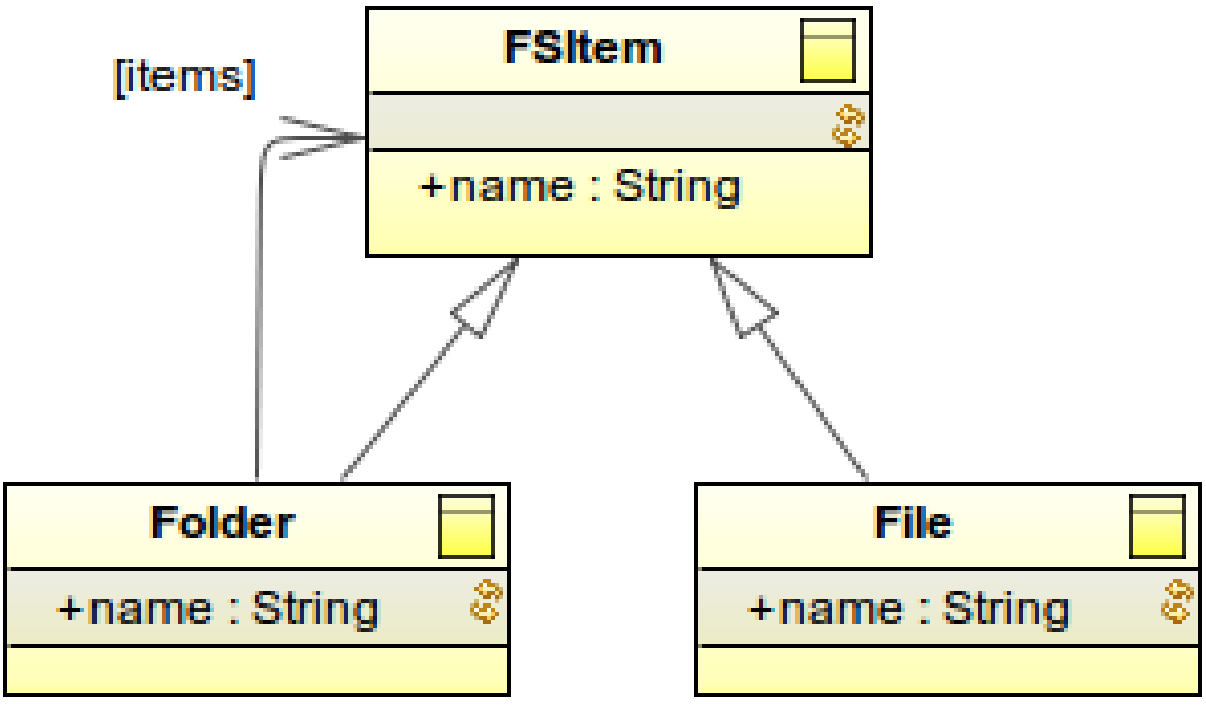
The **Tree Table** component allows you to display data in an expandable tree hierarchy. The nodes in the tree are identified by a special ID; Hence, one item can appear in multiple nodes of a tree.



8.3.2.3.1 Designing a Tree Table (forms::TreeTable)

To design a Tree Table in your form, do the following:

1. Insert the Tree Table component into the form.
2. In the Tree Table properties, define the data source of one of the following type:
   - **ClosureTreeDataSource**: the input object of the closure is the clicked node and it returns a list of objects that represent the data source;
     Example ClosureTreeDataSource



```
new ClosureTreeDataSource(
    {
    node ->
      //returns the list of root nodes on load (object stored in the home variable):
      if node == null then
        [home]
      else
        //what to return when the user clicks a node:
        //if the user clicked a Folder, return the list of its files and folders:
        if node.getType() == Folder then
          (node as Folder).items;
        //if the user clicked a file, do not return anything:
        else
          [];
        end;
      end;
    }
  )
```

- **SimpleTwoLevelTreeDS**: two-level tree
  Example SimpleTwoLevelTreeDS data source:

```
def SimpleTwoLevelTreeDS tree := new SimpleTwoLevelTreeDS(
        [ "eva" -> ["climbing", "painting"], "juraj" -> ["reposing", "windsurfing"] ]
    )
```

3. Insert *Table Column* components () into the Table.

   The column operates over the Data Source objects, so that on every column you can define what of the object is displayed.

4. Define the properties of each column:

   (a) Define the Value Provider that returns the content of the cell based on the row object.

   (b) Define the iterator type: the type is the type of the Value Provider

   The iterator type is always the same as the type of the Value Provider object: since it is not possible to infer the type of Value Provider object without possible inaccuracies (for example, if the Value Provider is defined by a closure), the user always needs to set the iterator type explicitly.

5. Insert the form components into the Table Column and define their properties. Use the Table iterator (*it* by default) to work with the iterated object).

8.3.2.3.1.1 Enabling and Disabling Filtering on a Tree Table (forms::TreeTable)

To enable or disable filtering on Tree Tables, use the `setFilteringEnabled(Boolean)` call on the component: The method hides or displays the filtering row.

> **Note:** When applying filtering and on large Collections, filtering actions might cause performance issues: consider using other data sources such as shared Type data sources for large data sets.

8.3.2.3.2 Defining a Tree Table with Different Node Types (forms::TreeTable)

To create a Tree Table with nodes that are of different data types, do the following:

1. Insert a Tree Table component into your Form definition.

2. Define the data source as a *SimpleTwoLevelTreeDS* or *ClosureTreeDataSource*.

   Check the type of the clicked node: typically you will cast the node to its type to be able to return the list of its child nodes. Note that on load, no node has been clicked yet, so the input object is *null*: make sure to handle this situation in the closure.

3. Insert the Table Column into the Tree Table and define the Value Provider type and the value provider expression below.

   Make sure that the Value Provider returns values of simple data types: Objects of another type cause a runtime exception since it is not possible to produce their filter and sort feature.

   - Closure Value Provider expression:

     ```
     {
        clickedNode:FSItem ->
        clickedNode.name;
     }
     ```

   - Property Path Value Provider expression:

     ```
     FSItem.name
     ```

8.3.2.3.3 Getting the Selected Item in a Tree Table (forms::TreeTable)

To allow the user to select an item in a tree table and obtain the selection, do the following:

1. Enable row selection on the tree table with the setSelectable() method:

   ```
   MyTreeTable.setSelectable(true)
   ```

2. Define the action when the user selects a row with the setSelectionChangeListener() method.

   ```
   MyTreeTable.setSelectionChangeListener( {e:forms::ValueChangeEvent ->
      //set the value in the selected row as caption of a Label:
      Selection.setCaption(
          toString(
             //e.source returns the TreeTable; getSelection the selected object:
             (e.source as forms::TreeTable).getSelection())
          )
       }
   )
   ```

8.3.2.3.4 Sorting a Tree Table (forms::TreeTable)

To enable sorting on a Grid or Table, select the sortable flag in the Sorting properties of the respective column: the user will be able to sort the displayed values by clicking the column header.

On a Table, if you do not want to use the column provider for sorting, define the sorting value provider below.

> **Important:** When applying sorting on large collections (the Data Source of the Grid or Table is set to Collection), sorting and filtering actions might cause performance issues: consider using other data sources such as shared Type data sources.

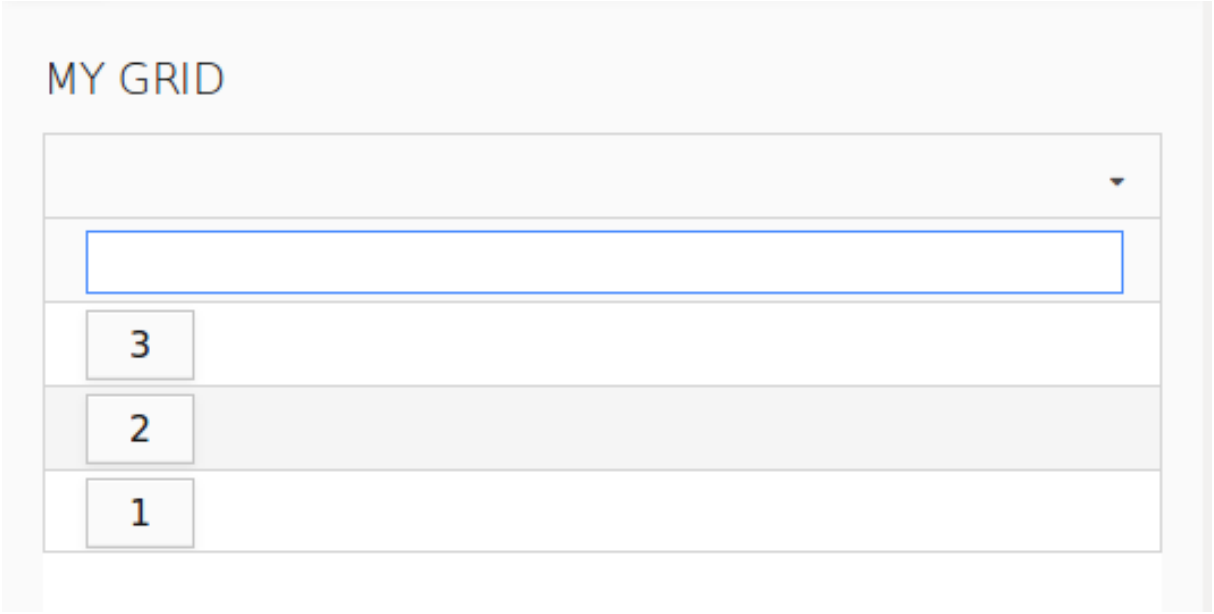


Figure 8.48 Filterable and sortable Grid Column

8.3.2.3.5 Filtering a Tree Table (forms::TreeTable)

Important: Filters on Columns of Tables and Tree Tables support only the Boolean, String, Integer, Decimal, Date, and Enumeration data types.

To enable filtering on a column of a Table, Tree Table, and Grid, do the following:

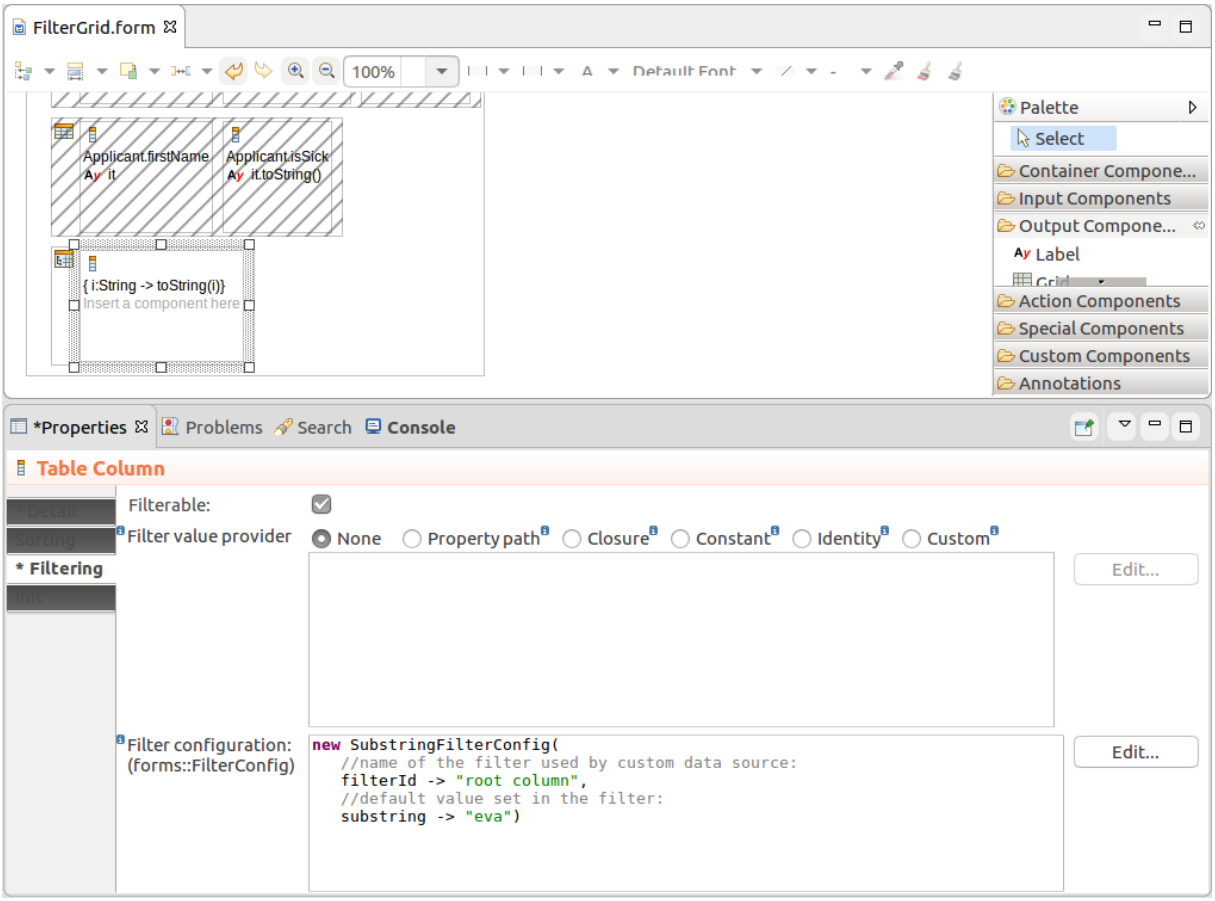
1. Make sure the table, tree table, or grid is filterable: the flag is set with the `setFilteringEnabled` method and is enabled by default.
2. On the column, set the properties:
   (a) Select the *filterable* flag.
   (b) Define the iterator data type at the bottom of the Properties view so the system can infer the filter it should use (Boolean, String, Integer, Decimal, Date, or Enumeration.
   (c) Make sure the Value Provider expression is set to return the same data type as the iterator data type.
   (d) Optionally define the filter configuration: The configuration allows you to further specify the filter properties: You can set it with the `setFilterConfig()` method.
   - *OptionsFilterConfig*: the user will be able to select one or multiple OptionsFilterConfig from an option set, which will be interpreted as the values and applied by the filter.

     ```
     new OptionsFilterConfig(multiselect -> true, options -> [
       new SelectItem(label -> "label of 1", value -> 1),
       new SelectItem(label -> "label of 2", value -> 2)
     ])
     ```
   - *NumericFilterConfig*: the user will define a number range

     ```
     // numeric filter with default values:
     new NumericFilterConfig(equal ->50, moreThan -> 10, lessThan -> 90)
     ```
   - *DateFilterConfig* so the user can define a date range to use for filtering and the format of selected dates

     ```
     new DateFilterConfig(
       resolution -> uicommon::DateTimeResolution.Month, formatPattern -> "YYYY MMMM"
     )
     ```

- *RexExpFilterConfig* to define the default regex pattern for filtering
- *SubStringFilterConfig* to define the default substring pattern for filtering
- *BooleanFilterConfig* to define properties of filter for Boolean values



Important: When applying filtering on large Collections (the Data Source is set to Collection), sorting and filtering actions might cause performance issues: consider using other data sources such as shared Type data sources.

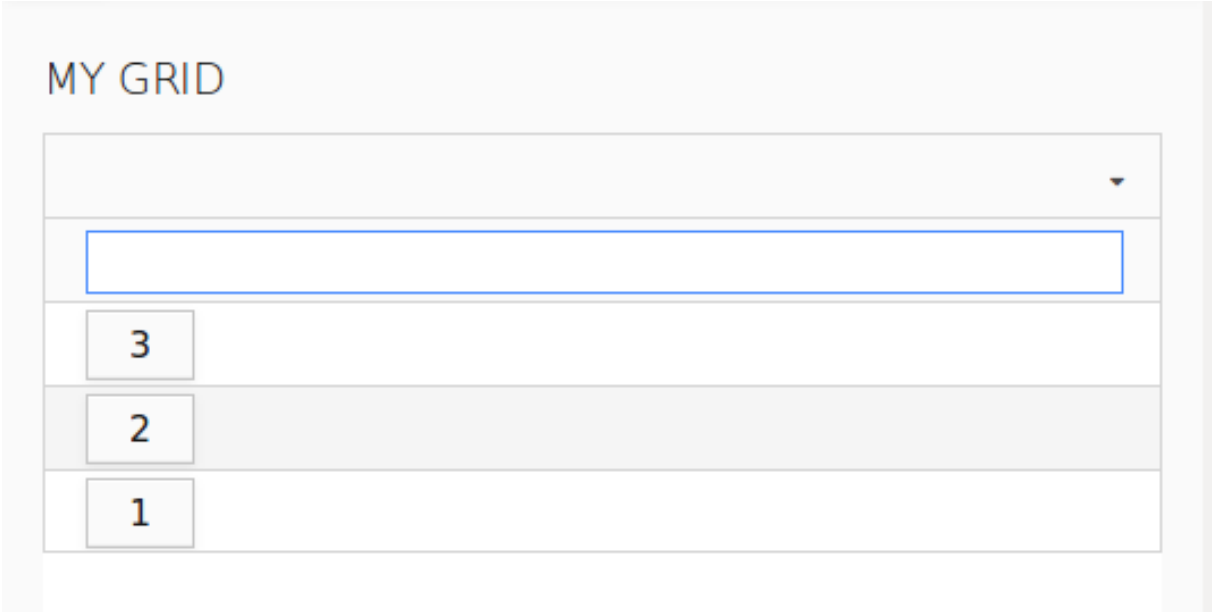


Figure 8.49 Filterable and sortable Grid Column

8.3.2.3.6 Enabling Filtering on Tree Tables (forms::TreeTable)

To enable or disable filtering on Tables and Tree Tables dynamically, use the `setFilteringEnabled(Boolean)` call on the component: The method hides or displays the filtering row.

8.3.2.3.7 Selecting Tree Table Row (forms::TreeTable)

To enable row selection on a Tree Table, set the tree table as selectable: `MyTable.isSelectable()` or `MyTable.setSelectable(true)`. You can then use the `setSelectionChangeListener()` to listen for user Selection and acquire the selected row with use the `getSelection()` method.

You can use the `select()` method to select a row from the Form.

```
c.setSelectable(true);
c.setSelectionChangeListener({ e -> selected := c.getSelection().toString(); SelectedLabel.refresh()})
```

8.3.2.3.8 Selecting Tree Table Rows (forms::TreeTable)

To enable row selection on a Tree Table set it as selectable: `MyTreeTable.isSelectable()` or `MyTreeTable.setSelectable(true)`. You can then use the `setSelectionChangeListener()` to listen for user Selection and acquire the selected row with use the `getSelection()` method.

You can use the `select()` method to select a row from the Form.

```
c.setSelectable(true);
c.setSelectionChangeListener({ e -> selected := c.getSelection().toString(); SelectedLabel.refresh()})
```

8.3.2.3.9 Collapsing a Column

Columns of tabular components can be collapsed from the front-end by the user as well as programatically. You can also enable or disable the feature.

By default, Columns can be collapsed:

- To disable collapsibility of a Column, call the following method:
  - The `setCollapsible(false)` method on the Column of a Tree Table or Table.
- To collapse or uncollapse a collapsible Column, call the following method:
  - The `setCollapsed(Boolean)` method on the Column of a Tree Table or Table.

8.3.2.3.10 Adding Column Dynamically

To add a Grid Column, Table Column, or a Tree Table Column to the parent component Dynamically, for example, on button click, call the `addColumn()` method on the parent component.

Adding a column on button click (Click Listener):

```
{e -> Vocab.addColumn(
   new forms::TableColumn(
       data -> null,
       modelingId -> null,
       filtrable -> false,
       generator -> null,
       sortable -> false,
       valueProvider -> new PropertyPathValueProvider(Unit.svk))
   )
}
```

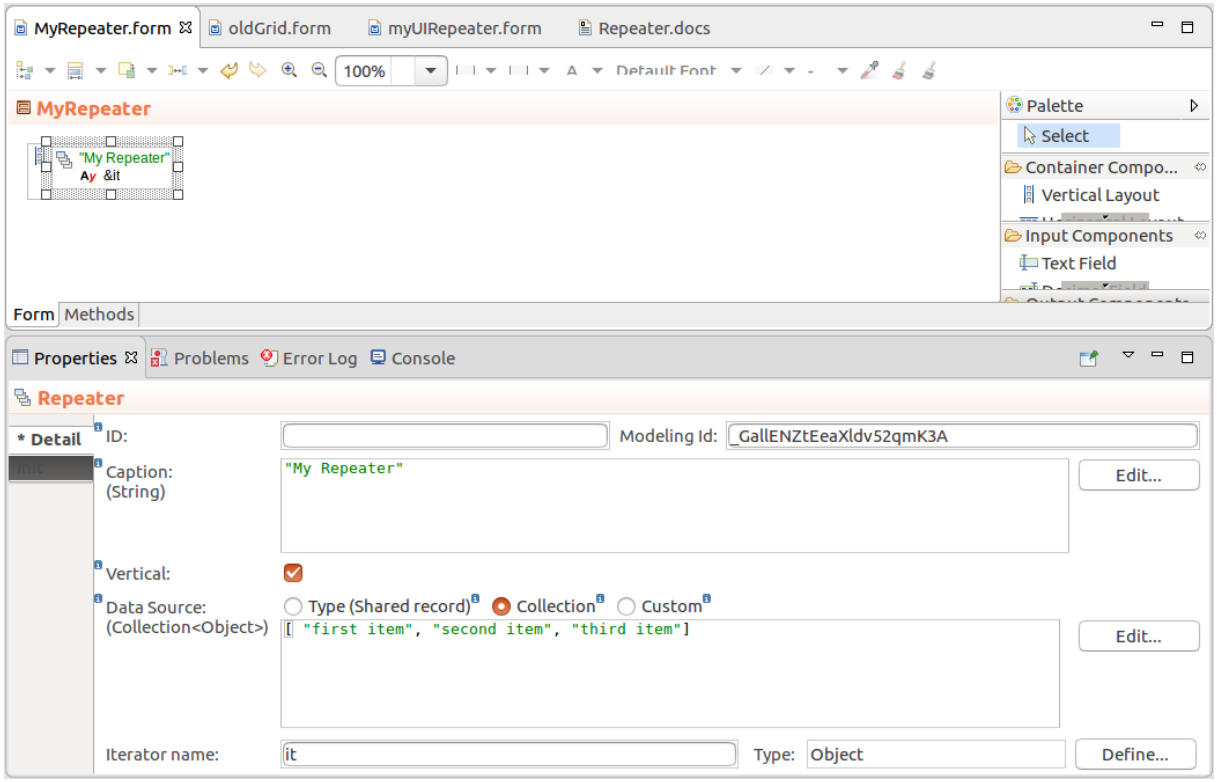
8.3.2.4 Repeater (forms::Repeater)

The *Repeater* component serves to display the same component repeatedly over a set of data defined by Repeater's data source. Note that unlike Tables, TreeTables, and Grids, Repeaters do not load their data source objects lazily.

8.3.2.4.1 Defining a Repeater

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

1. Insert the Repeater component into your Form.
2. In the Properties view of the Repeater, define its properties:
   (a) Select the Vertical option to display the repeated content vertically: leave blank to display the components horizontally.
   (b) Define the data source type and value.
   (c) If applicable, change the data type of the iterator in the *Type* field next to the iterator name.
3. Insert form components into the Repeater form: use the iterator name to work with the current data-source object.



8.3.2.4.2 Paging in Repeater

To implement paging in a Repeater, do the following:

1. Set the required page size on the Repeater with `setPageSize()`
2. Set the required start index on the Repeater with `setStartIndex()`.
3. Create a component, that will implement the next page functionality, for example, a Next button with a click listener:

```
{ e ->
  //set new start index when the user clicks previous
  myRepeater.setStartIndex(myRepeater.getStartIndex() + myRepeater.getPageSize());
  myRepeater.refresh();
}
```

4. Create a component, that will implement the previous page functionality, for example, a Next button with a click listener:

```
{ e ->
  //set new start index when the user clicks previous
  myRepeater.setStartIndex(myRepeater.getStartIndex() - myRepeater.getPageSize());
  myRepeater.refresh();
}
```

5. Create a form method that will disable and enable the next and previous page functionality when no more previous or next items are available:

```
private void refreshButtonEnabled() {
    nextButton.setEnabled( myRepeater.getStartIndex() + myRepeater.getPageSize() < repeaterCo
    prevButton.setEnabled( not (myRepeater.getStartIndex() - myRepeater.getPageSize() < 0) );
}
```

8.3.3 Charts

> **Important:** Chart components are based on Vaadin Charts, which are developed by a third party. Before designing your charts, make sure to obtain the respective licenses for Vaadin Charts.

Chart components render data as defined by their plotting options; hence you need to define these:

- *The data is defined as data series or data values* depending on the chart.
- *Plotting is defined in the plotting options* of each the chart or data series.

8.3.3.1 Generic Chart Features

All charts can be redrawn on request and trigger an action, when the user clicks a data point in the chart.

8.3.3.1.1 Redrawing Charts

To redraw a chart, call the `refresh()` method on the chart.

8.3.3.1.2 Performing Actions on Chart Click

To handle a click on a chart item, define the `onPointClick` property of the chart with its setter method `setOnPointClick()`. The property is a closure with the *ChartPointClickEvent* produced by the click as its input parameter.

Typically, you will define the action on the Init tab of the component so the clicks are processed from the moment the chart is displayed.

```
myChart.setOnPointClick(
  {
    click:ChartPointClickEvent ->
      clickedSeries.setCaption(click.series)
  }
);
```

If you need to pass further data about the data item, define `payload` on the data series items: The payload is passed as the *payload* property of the ChartPointClickEvent.

```
[
  new ListDataSeries(
    [
       new ListDataSeriesItem( 1 , "This is some payload."),
       new ListDataSeriesItem( 5 , "This is also some payload.")
    ]
  )
]
```

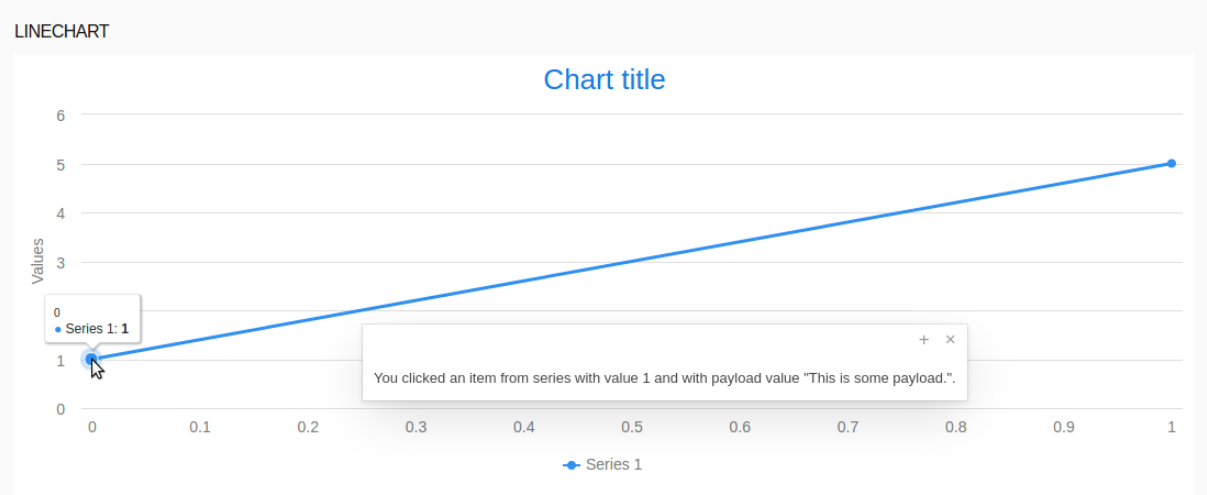


Figure 8.50 Item payload sent to a popup and displayed on click

8.3.3.2 Pie Chart (forms::PieChart)

The Pie Chart component is plotted as a pie chart; its data is defined in a list of `PieSliceSeries`. Each PieSliceSeries is plotted as its own pie with a set of PieSlices with the data. The pies are stack on each other in the order they are arranged in the data set. Mind that in such cases you will like need to adjust the plotting of the data series in order to make the pies visible.

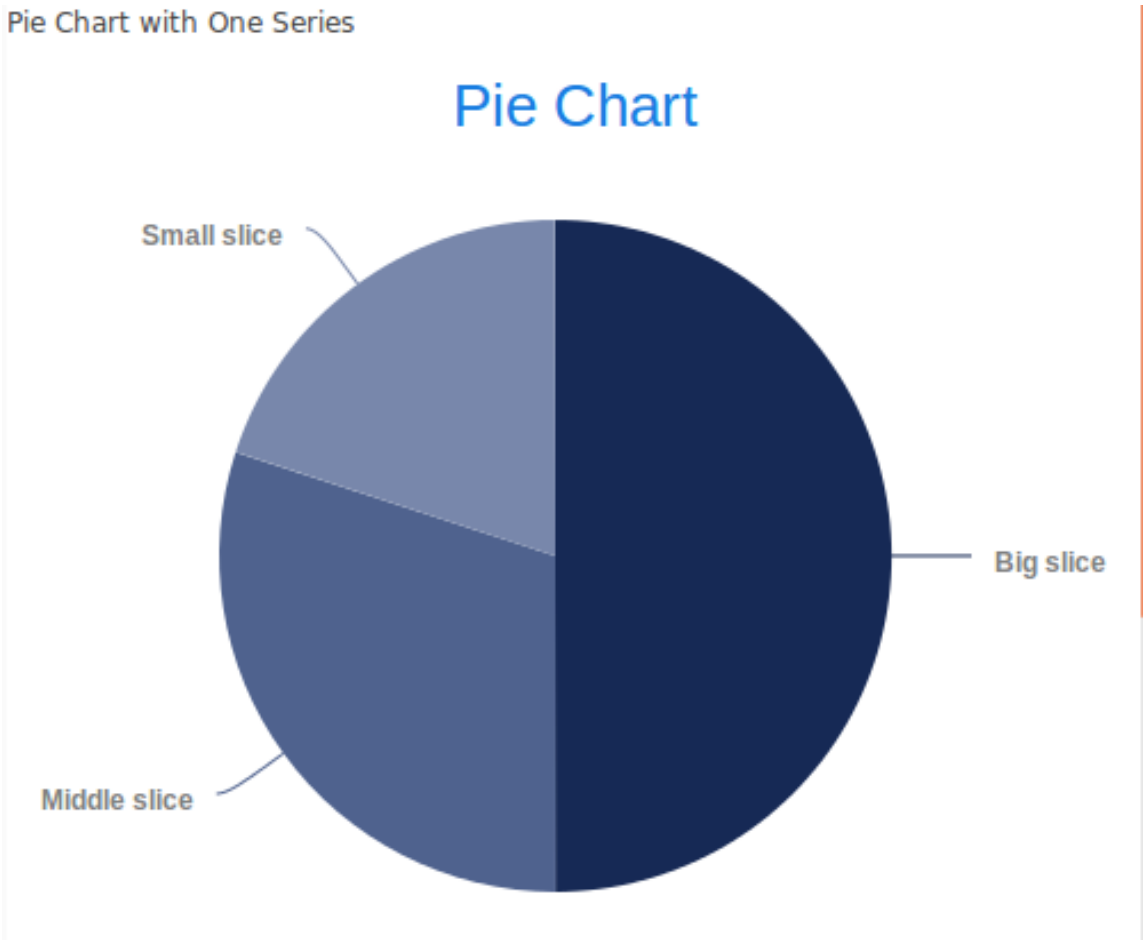
8.3.3.2.1 Creating a Pie Chart

To create a pie chart, do the following:

1. Insert the pie chart component into your form.
2. Define the pie data series:
   - either in the Pie slice series field of the Properties view
   - or using the *addPieSliceSeries()* method on a pie chart

   Data for a pie chart with a single series

```
def PieSliceSeries mySeries := new PieSliceSeries("My only series", [
        new forms::PieSlice("Big slice", 50, "Business data", new Color("#162955")),
        new forms::PieSlice("Middle slice", 30, "Business data", new Color("#4F628E")),
        new forms::PieSlice("Small slice", 20, "Business data", new Color("#7887ab"))
]);
[ mySeries ]
```



8.3.3.2.2 Adding a Pie Data Series on Action

To add a data series when the user or the system performs some action, run the `addPieSliceSeries()` method on the chart from the appropriate location, such as, Click Listener of a button.

```
{ e -> myAlreadyRenderedPeiChart.addPieSliceSeries(myPieSliceSeries)}
```

8.3.3.2.3 Plotting Pie Slice Series

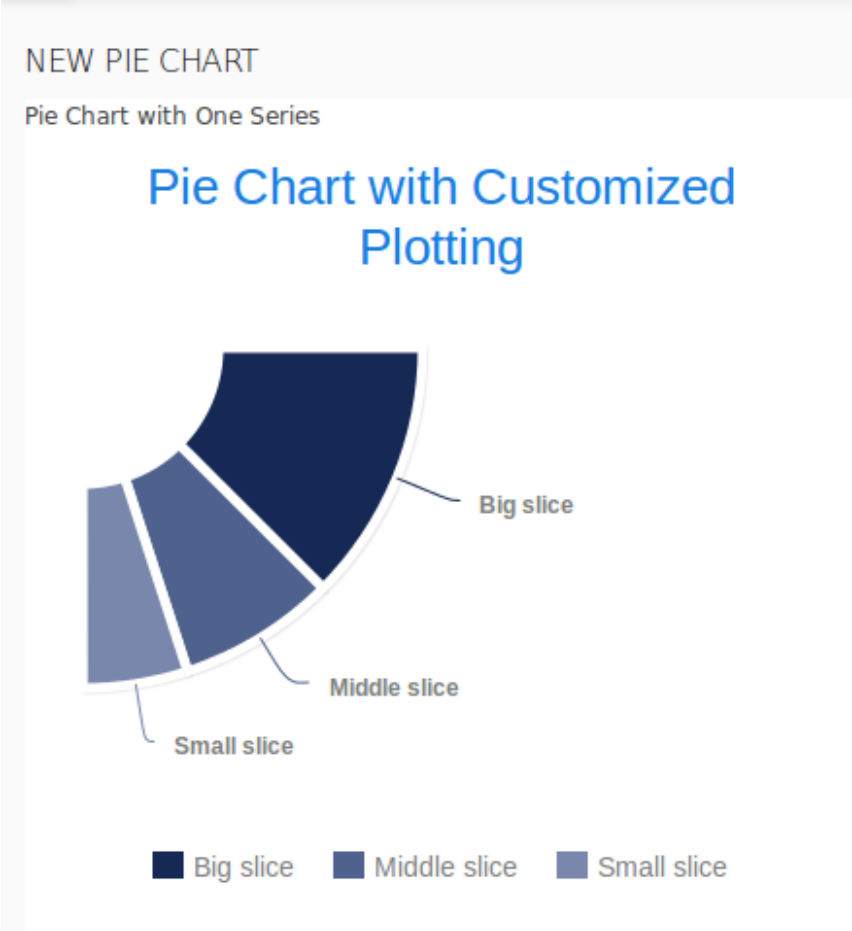
To define how individual pie slice series are presented, define their plotting options as a PlotOptionsPie object and assign it to the series.

In the PlotOptionsPie object, you can define the following properties:

- *innerSize*: distance of the inner border from the center (size of the "hole" in the donut chart)
  You can define it either in percent relative to the pie size or in pixels as Integer value.
- *dataLabels*: formatting of slice labels
- *startAngle*: rotation of the chart series (angle where the first slice starts)
  0 is the top; 90 is right;
- *size*: perimeter of the chart series
  You can define it either in percent relative to the plot area or in pixels as Integer value.

- *shadow*: shadow behind the chart series
- *center*: position of the chart center relative to left-upper corner of the component slot
- *endAngle*: degree that marks the end of the chart
  0 is the top; 90 is right; `null` and the default value result in *startAngle + 360*.
- *showInLegend*: whether the slices are included in the chart legend

```
def PlotOptionsPie mySeriesPlotting := new PlotOptionsPie(
                      //inner border is at 40% of the perimeter size:
                      innerSize -> new AttributeSize("40%"),
                      //rotation:
                      startAngle -> 90,
                      //chart series size:
                      size -> new AttributeSize(350),
                      shadow -> true,
                      borderWidth -> 5,
                      //position of the series:
                      center -> new AttributeCenter(0, 0),
                      endAngle -> 180,
                      showInLegend -> true);
mySeries.plotOptions := mySeriesPlotting;
```



8.3.3.2.4 Formatting Pie Slice Labels

To define formatting of data labels on data series of Pie Charts, define the dataLabels property on the pie data series as a PieDataLabels object.

Formatting expression can access the following variables:

- percentage: percentage the slice takes
- point: pie slice object
- series: pie series object
- total: total of the series
- *formatter*: label template
- *format*: label template that *overrides formatter*
  From *format*, the variables are available on point object, for example `{point.percentage}`.

```
dataLabels -> new PieDataLabels(
   //javascript code used as label:
   formatter -> "'Percentage: ' + this.percentage +
   '<br>Point name: '+ this.point.name +
   '<br>Series: ' + this.series.name +
   '<br>Total: ' + this.total +
   '<br>x: ' + this.x ",
   //formatter -> "'this.x'",
   //rotation in degrees counterclockwise:
   rotation -> 45,
   format -> "Name: {point.name} Percentage: {point.percentage} Y: {point.y}",
   enabled -> true,
   color -> new Color(8, 8, 8),
   //Distance of the data label from the slice edge; can be also negative:
   distance -> 150)
```

8.3.3.3 Cartesian Chart (forms::CartesianChart)

Cartesian Charts serve to plot data as line, bar, area, or bubble charts.

The charts can plot multiple data sets, called data series, with different plotting. Data series are groups of related data that represent one graph.

When creating a data series, pick the data series type depending on the data values:

- **ListDataSeries**: a sole decimal value
- **DecimalDataSeries**: a decimal and, a range or another decimal values
- **CategoryDataSeries**: a category, and ranges or sole decimal values
- **TimeDataSeries**: a point in time, and a sole decimal value or range

The data in a series is defined as a list of data-series items. Each type of data series can work only with data items of the respective type: CategoryDataSeries can contain only CategoryDataSeriesItems; DecimalDataSeries can contain only DecimalDataSeriesItems; ListDataSeries can contain only ListDataSeriesItems.

The data series items can hold ranges or values where applicable:

- **ListDataSeriesItems**: a sole decimal value, for example, `new ListDataSeriesItem(0.21);` the value is the y value and x is the index of the item in the list.
- **CategoryDataSeriesItems**:
  - a *decimal value* with an optional name; for example, `new CategoryDataSeriesItem("↩ Greenland", 56196)`

  - or a *range of values* with an optional name; for example, `new CategoryDataSeriesItem("↩ Greenland", 56196, 56239).`

- **DecimalDataSeriesItems** :
  - decimals with the *x and y coordinates*, for example, `new DecimalDataSeriesItem(2, 5)`
  - or a *range of decimal values*; for example,`new DecimalDataSeriesItem(7, 3, 8)`
- **TimeDataSeriesItems**:
  - date with a value, for example, `new TimeDataSeriesItem(now() + seconds(1), 7)`
  - or a *range and a date*; for example,`new TimeDataSeriesItem(now(), 1, 3)`

Each data series defines its plotting options; hence, each data series can be plotted differently from the others: override the plot options if you want to change the plotting.

Example data series

```
new CategoryDataSeries (
  [
    new CategoryDataSeriesItem("Greenland", 56196, 56239),
    new CategoryDataSeriesItem("China", 1388232693, 194202093),
    new CategoryDataSeriesItem("India", 1342512706, 1358354355),
    new CategoryDataSeriesItem("USA", 326474013, 328857273)
  ]
);
```

8.3.3.3.1 Adding a Cartesian Data Series on Action

To add a data series when the user or the system performs some action, run the `addDataSeries()` method on the chart from the appropriate location, such as, Click Listener of a button.

```
{ e -> myRenderedChart.addDataSeries(myCatesianSeries)}
```

8.3.3.3.2 Adding a Data Series Item on Action

To add a data series item to a data series when the user or the system performs some action, run the `addItem()` method on the data series from the appropriate location, such as, Click Listener of a button.

```
{ e -> myAlreadyRenderedDataSeries.addItem(new ListDataSeriesItem(5, "This is new item with value 5."))}
```

8.3.3.3.3 Changing the Current Data Series

To use another data series in a Cartesian chart, call the `setDataSeries()` method on the chart.

8.3.3.3.4 Changing Plotting of a Cartesian Chart Series

Each type of data series has default plotting which depends on the data in their items. Note that data series items can define their default plot options:

- ListDataSeriesItem: PlotOptionsLine
- CategoryDataSeriesItem: PlotOptionsColumn
- DecimalDataSeriesItem and TimeDataSeries: with range (low and high) PlotOptionsLineArea; with values PlotOptionsLine

If you are not happy with the default plotting, you can adjust plotting of individual series, by setting the `plotOptions` property of the data series.

Note that some plotting options apply only to some types of data series item: for example, if you are working with series items that have decimals values only, it is not possible to use the Bubble plotting since this plotting requires a range to define the size of the Bubble marker.

You can plot the series items as the following:

- `PlotOptionsColumn`: bar chart plotting

  It applies to items with a single value and CategoryDataSeriesItems with a range.
- `PlotOptionsBubble`: bubble chart plotting (requires range data series items; items with values are ignored)

  It applies to items with ranges.
- `PlotOptionsScatter`: scatter chart plotting (requires value data series items; items with ranges are ignored)

  It applies to items with values.
- `PlotOptionsLine`: line chart plotting

  It applies to items with values.
- `PlotOptionsLineArea`: line-area chart plotting (area below the line is filled)

  It applies to items with values.

Defining plotOptions property of a data series

```
myDataSeries.plotOptions := new PlotOptionsLineArea(spline -> true, range -> true)
```

8.3.3.3.4.1 Changing Plotting of Point Markers on Scatter, Line and Line Area Charts

To change the plotting of point markers on Scatter, Line and Line Area Charts, set the marker property of the plot options:

```
//changing marker plotting:
myDataSeries.plotOptions := new PlotOptionsScatter(marker -> Marker.diamond);
//"smoothing" out the graph line:
itemListDataSeries.plotOptions := new PlotOptionsLine(spline -> true);
//setting graph line color:
itemListDataSeries.plotOptions.color := new Color(0,0,0);
//to interrupt the flow of the graph, set the item where to interrupt to null:
def ListDataSeries dataSeries := new ListDataSeries([1, 2, 3, 2, null, 4, 6]);
//to stack bars of a bar chart on the y axis behind each other,
//set the plotOptions.stacked to true:
def forms::PlotOptionsColumn plotOptionsColumn := new forms::PlotOptionsColumn();
plotOptionsColumn.stacked := true;:
```

8.3.3.3.5 Polar Chart (forms::PolarChart)

Since Polar chart is a special case of the Cartesian chart, the uses the same types of data series just like in the Cartesian Chart and has the same plotting options.

8.3.3.3.5.1 Creating a Polar Chart

To create a polar chart, insert the component to your form and define its data series. Any other required properties will use their default values.

8.3.3.4 Gauge Chart (forms::GaugeChart)

The *Gauge Chart* component has a single decimal as its value.

To define an action to double-clicking the point, set the `onPointClick` field of the Gauge Chart with the setter method:

```
c.setOnPointClick({ e:ChartPointClickEvent -> doSomething()})
```

8.3.3.4.1 Creating a Gauge Chart

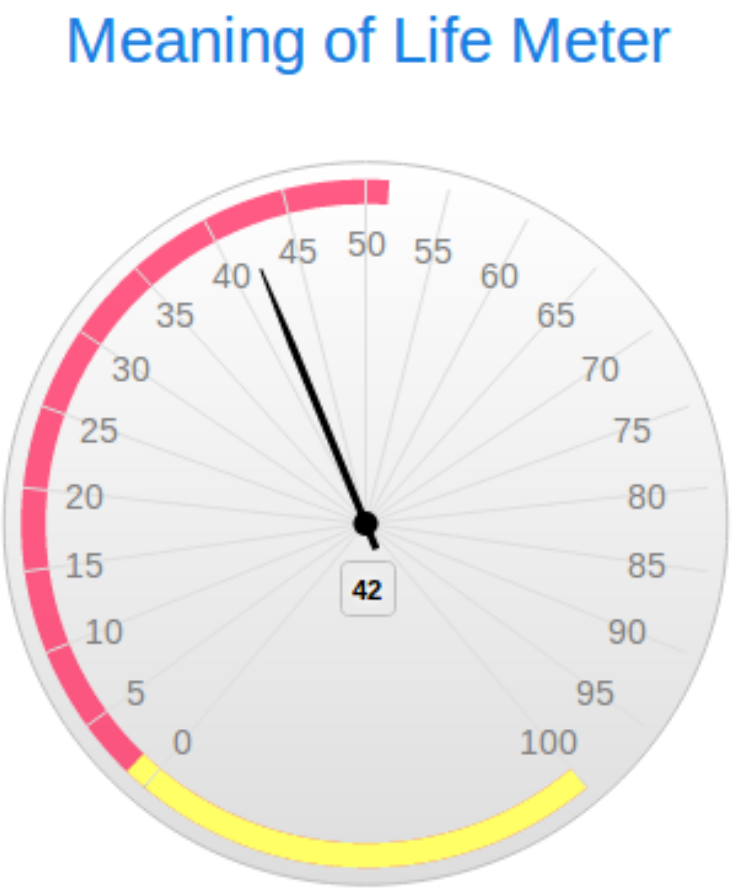
To create a gauge chart, do the following:

1. Insert the gauge chart component into your form.
2. Define the Properties:
   - *Value*: where the needle should point
   - *Value name*: value description displayed when hovering over the needle
   - *Y Axis*: axis

Note that ticks have by default 0 length and are hence not displayed.

Example axis definition

```
new Axis(
  min -> 0,
  max -> 100,
~
  tickLength -> 0,
  tickInterval -> 5,
  tickWidth -> 2,
  tickColor -> new Color(0, 100, 100, 1),
~
  plotBands -> [
    new forms::PlotBand(from -> 180, to -> 240, color -> new Color(255, 51, 102, 0.8)),
    new forms::PlotBand(from -> 130, to -> 180, color -> new Color(255, 255, 102, 1))
  ]
);
```



To remove the ticks, set `gridLineWidth` to 0.

8.3.3.5 Axes (forms::Axis)

Axes are rendered in some default way for each type of data series in your chart

8.3.3.5.1 Changing Axis Range

To change the range of the chart axis, set its `min` and `max` properties.

```
//new axis with its maximum and minimum set:
new Axis(
min -> 0,
max -> 240)
```

8.3.3.5.2 Axis Values

By default, both the axes display decimals as their values: This is intuitive for the y axis: after all, we are visualizing numbers, but might not be the required values for your x axis.

Note that the following values are used on the x axis by default:

- for ListDataSeriesItem and CategoryDataSeriesItem, the index of the item in the List
- for DateDataSeriesItem, the date in epoch milliseconds
- for DecimalDataSeriesItem, the date in epoch milliseconds

These properties can be overridden in the axis properties or with the `addXAxis()` or `addYAxis()` method of the chart.

8.3.3.5.2.1 Changing Axis Values

If you want to change the values on an axis, define a new axis in the axis properties or with the `addXAxis()` or `addYAxis()` method of the chart.

```
new Axis('type' -> AxisType.category, categories -> ["My first name", null, "My third name"])
```

If you want to change only the value types on the axis, it is enough to change the AxisType. For example, if you want to display Date object as dates on the axis, define a new axis of the datetime type: 'new Axis('type' -> AxisType.datetime)'.

8.3.3.5.2.2 Creating Multiple Axis

You can add multiple axes to you chart with the `addXAxis()` or `addYAxis()` method: by default such axes are rendered next to each other. You might want to render one of the Axis on the opposite side of the chart: to do so, set its `opposite` property to true.

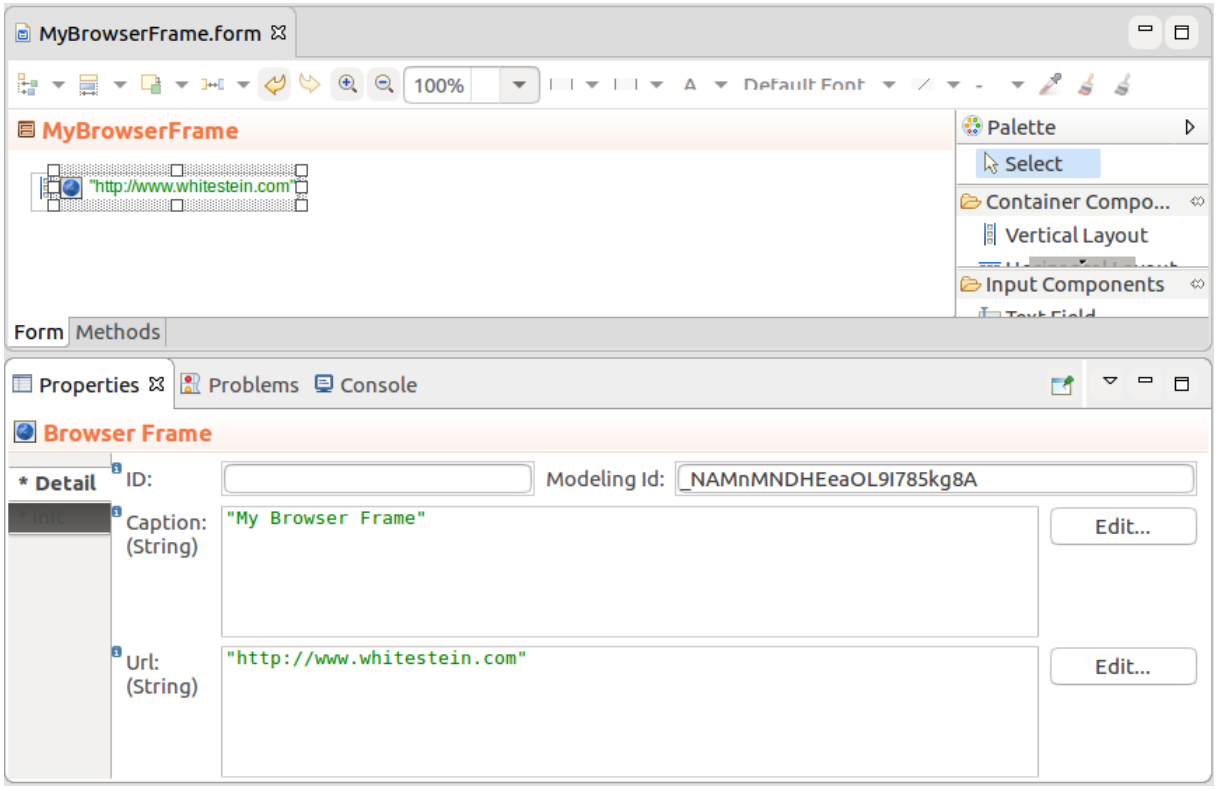
```
def forms::Axis yAxisTemperature := new forms::Axis();
yAxisTemperature.labels := new forms::ChartLabels(format -> "{value} °C");
yAxisTemperature.title := "Temperature";
yAxisTemperature.max := 20;
yAxisTemperature.min := -55;
chart.addYAxis(yAxisTemperature);

def forms::Axis yAxisPressure := new forms::Axis();
yAxisPressure.labels := new forms::ChartLabels(format -> "{value} kPa");
yAxisPressure.opposite := true;
yAxisPressure.title := "Pressure";
yAxisPressure.max := 102;
yAxisPressure.min := 20;
chart.addYAxis(yAxisPressure);
```

8.3.4 Other Output Components

8.3.4.1 Browser Frame (forms::BrowserFrame)

The *Browser Frame* renders as a view with the URL content. The target URL is defined in the **URL** property as a String, for example "`http://www.whitestein.com`".

To URL is loaded when the frame is initialized; to refresh the Browser Frame, call its `setURL()` method.



8.3.4.2 Image (forms::Image)

The *Image* () component renders an image which is defined as a `DownloadableResource` object of the image component.

The image objectj can be:

- `ThemeResource` which defines a file from the current CSS schema
- `FileResource` with a file uploaded with your modules
- `ExternalResource` with an external URL to an

8.3.4.2.1 Displaying an Image

To display an image in your form, do the following:

1. Insert the Image () component into your Form.
2. In the Properties view, define the icon source in the resource expression or with the `setSource()` method.

Note that the component does not check the type of the source. If you serve it a source of an incorrect type, it might result in a runtime exception or the content might not be displayed.

Expression that adds an Image located in the module to a Vertical Layout

```
 def Image image := new forms::Image();
image.setSource(new FileResource({ -> getResource("image description", "resource/picture.jpg")}));
~
myVerticalLayout.addComponent(image);
```

Expression that an Image component with an image located in the current theme

```
 def Image image := new forms::Image();
image.setSource(new ThemeResource("icons/auth_delegate.gif"));
image
```

8.3.4.3 Download (forms::Download)

The *Download* () component is rendered as a link or button, which downloads a resource when clicked. The resource can be a file uploaded with your modules, an external URL resource, or a file from the current application schema.

Downloads define the following type of data sources:

- **ExternalResource**: resource accessed via a URL
- **FileResource**: resource that holds a File object
- **ThemeResource**: resource from the current CSS theme

8.3.4.3.1 Defining a Download Button or Link

To create a Button or Link that starts a file download when clicked, do the following:

1. Insert the Download () component into your Form.
2. In the Properties view, define the resource expression.
3. In the Style property, define whether the component should be rendered as a link or as a button.

Expression that returns a Download component

```
new Download(
  "Download",
  new FileResource({ ->
    getResource("form_components", "output/myfile.txt")
  }),
  DownloadStyle.Link)
```

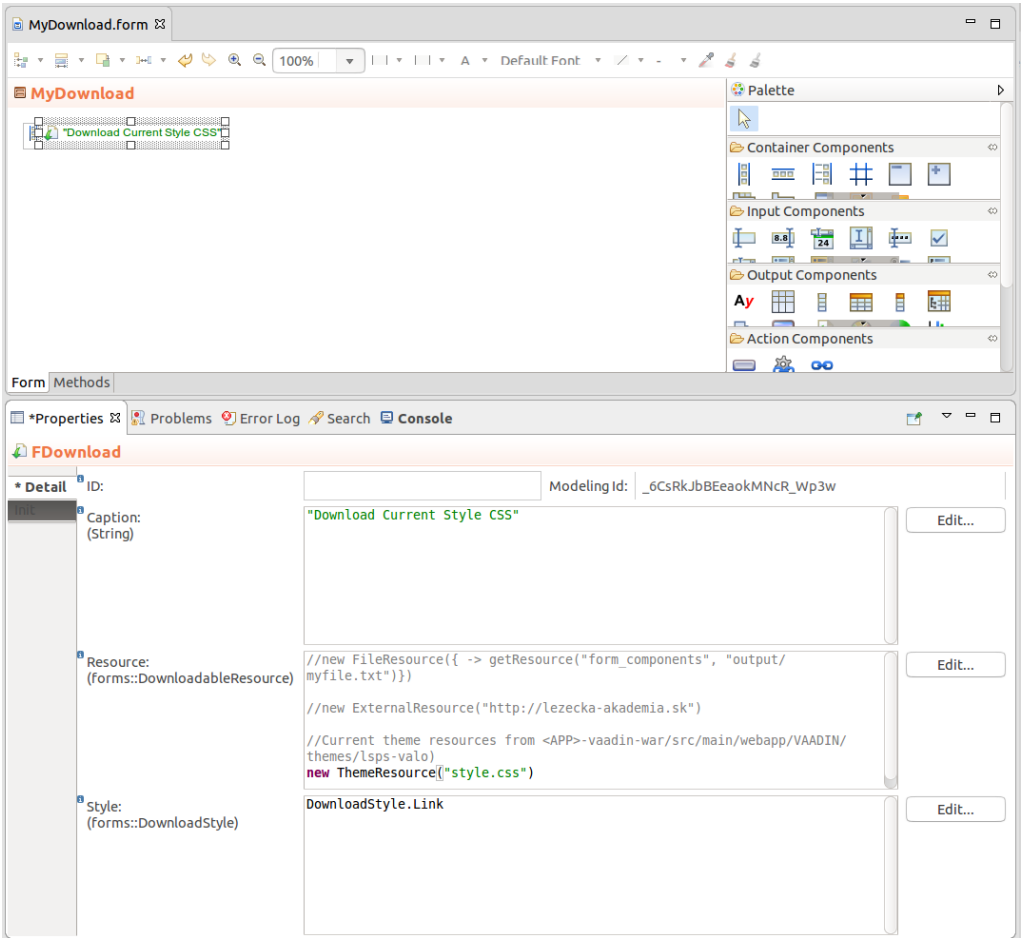


Figure 8.51 Form with a Download component

8.3.4.4 Calendar (forms::Calendar)

> **Important:** In the Vaadin 8 implementation, this component is not supported. For more information, refer to information on Vaadin 8 upgrade.

The *Calendar*  component serves to display and manage event data. The calendar API required that the event data be wrapped in CalendarItems.

Calendar can be rendered in the month, week, and day mode.

Calendar entries are defined as a list of CalendarItems returned by the calendar item provider, that can be either a ClosureCalendarItemProvider or a ListCalendarItemProvider.

```
new ClosureCalendarItemProvider(
  { from, to ->
    collect(
      //get data of event that occur within the displayed period:
      select(
        tripEvents,
        {e:TripEvent ->
            e.startDate <= to || e.endDate >= from
        }
```

```
      ),
      //Create CalendarItems over the data:
      {e:TripEvent ->
        new forms::CalendarItem(
          allDay -> true,
          data -> e,
          endDate -> e.endDate,
          caption -> e.description,
          startDate -> e.startDate)
      }
    )
  }
)
```

In the component, you can handle the following events:

- creating a new calendar item: the CreateEvent is thrown when the user clicks on a day field or drags the mouse pointer over multiple fields

```
TripCalendar.setCreateListener (
  { x:forms::CalendarCreateEvent ->
    //display popup for event creating (event is created on button click
    //in the popup and added to the business data):
    createPopup(x.from, x.to).show();
    TripCalendar.refresh()
  }
);
```

- editing a calendar item: the EditEvent is thrown when the user clicks an event text

```
TripCalendar.setEditListener({
x:forms::CalendarEditEvent -> createPopup().show();
TripCalendar.refresh()
});
```

- rescheduling a calendar item: the RescheduleEvent is thrown when the user drag-and-drops an existing item

```
TripCalendar.setRescheduleListener({
  x:forms::CalendarRescheduleEvent ->
  def TripEvent te := x.data as TripEvent;
  te.startDate := x.from;
  te.endDate := x.to;
  TripCalendar.refresh();
});
```

8.3.4.5 Map Display (forms::MapDisplay)

The *Map Display*  component renders as an OpenStreetMap with the defined center location, zoom, and one or multiple markers. You can work with map markers with the maps methods addMarker(), removeMarker(), on↩ MarkerClick(), and onMarkerDrag().

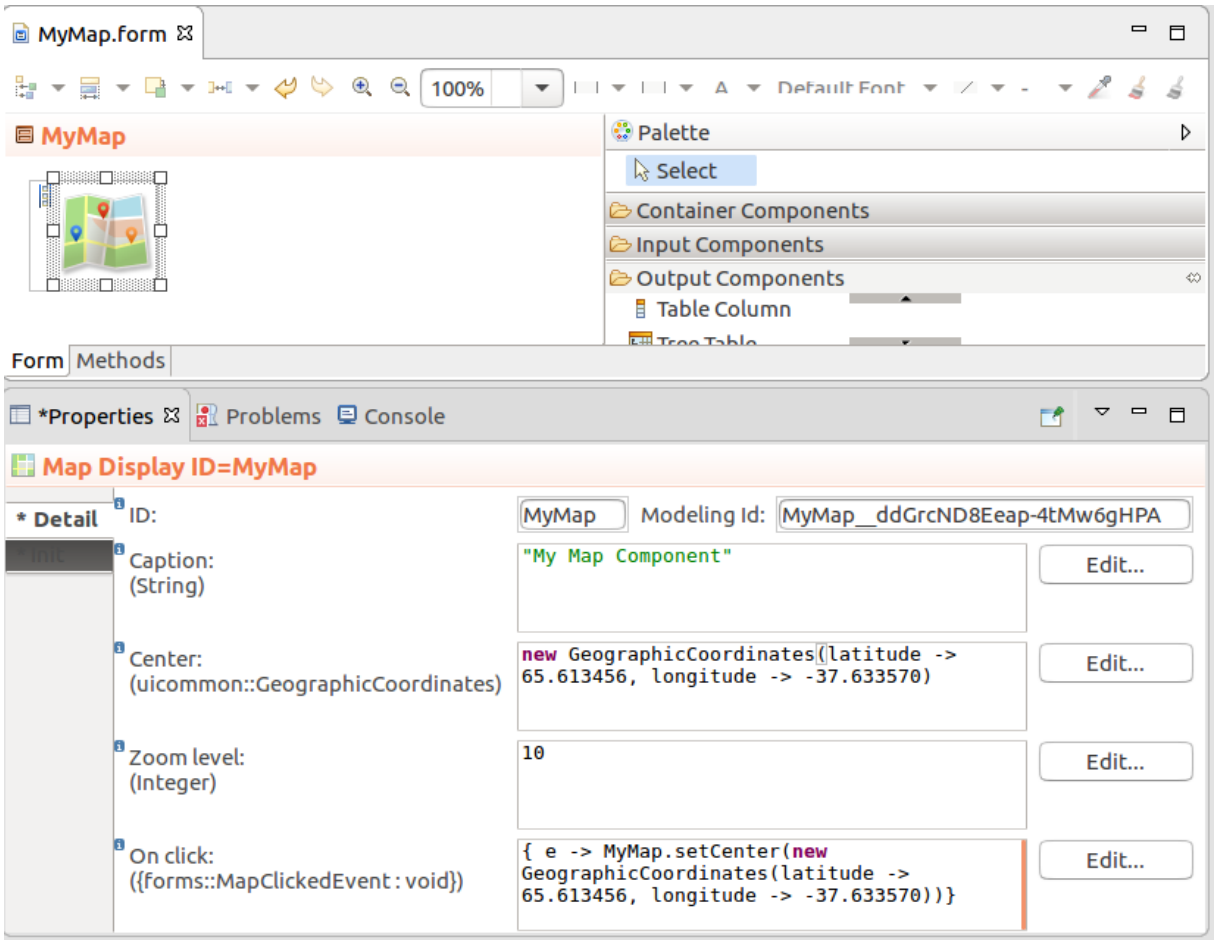
> **Note:** To work with the client GPS coordinates, use the Forms.detectLocation() method.

Map Properties

- **Center**: coordinates of the center of the rendered map

- **Zoom level**: default zoom on initialization and refresh defined as an Integer with value 0-18 (0 being the lowest zoom with the entire Earth displayed)

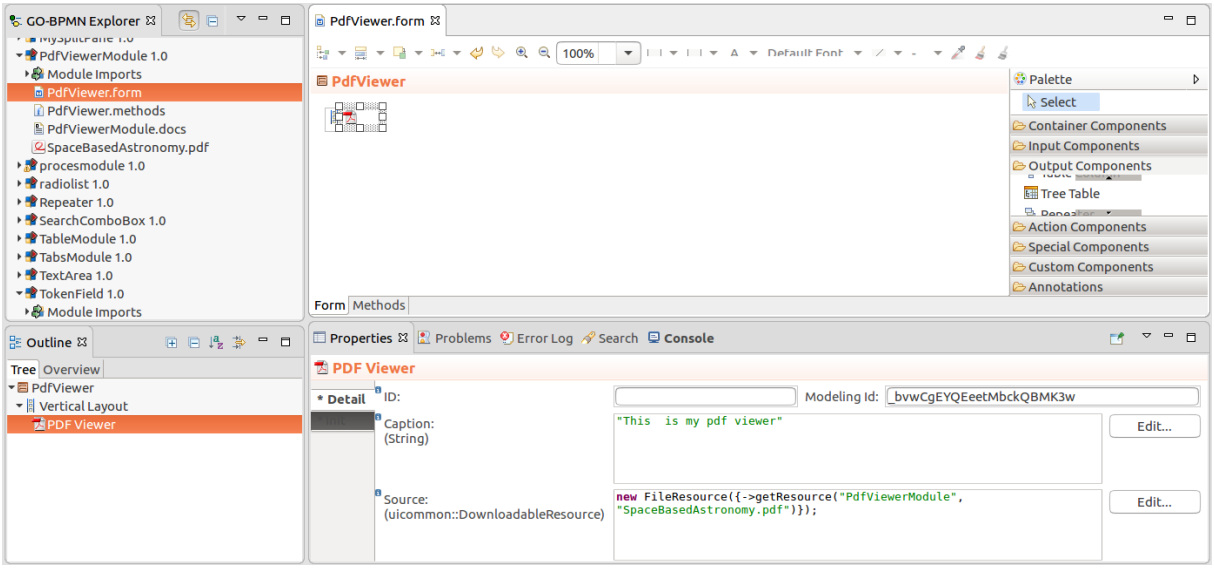
- **On click**: closure executed when the map is clicked



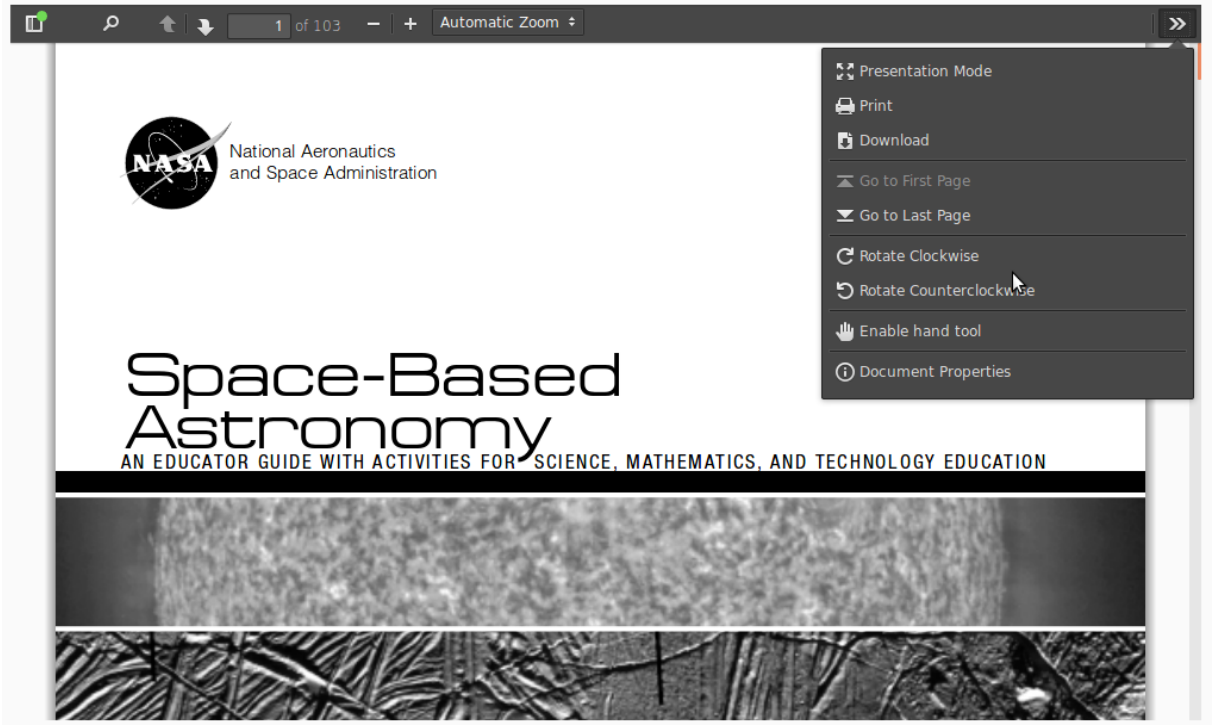
8.3.4.6 PDF Viewer (forms::PdfViewer)

The *PDF Viewer* component, *PDFViewer*,  displays a PDF defined as a FileResource; for example, `new FileResource({->getResource(thisModel().name, "MyDoc.pdf")})`.

When opening a PDF in your browser, the browser saves data about the currently open page, applied zoom, and sidebar position. By default, the PDF Viewer component applies this data when it is available: you can enable or disable this behavior with the `restoreLastPdfState()` method.



On the first load, the viewer might need a few seconds to load its resources.



8.4 Action Components

Action components are clickable components: on click they trigger an action:

- Button (forms::Button)
- Action Link (forms::ActionLink)
- Link (forms::Link)

8.4.1 Button (forms::Button)

The *Button* ( ) component renders as a button and typically defines an action taken when the button is clicked:

On click, the expression defined in *Click Listener* is executed.

A button click action can be also triggered with the shortcut set with the `setClickShortcut` method, for example, `mySaveButton.setClickShortcut(KeyCode.A, ModifierKey.CTRL)` to trigger action with CTRL+A. The shortcut can be removed with the `removeClickShortcut()` method.

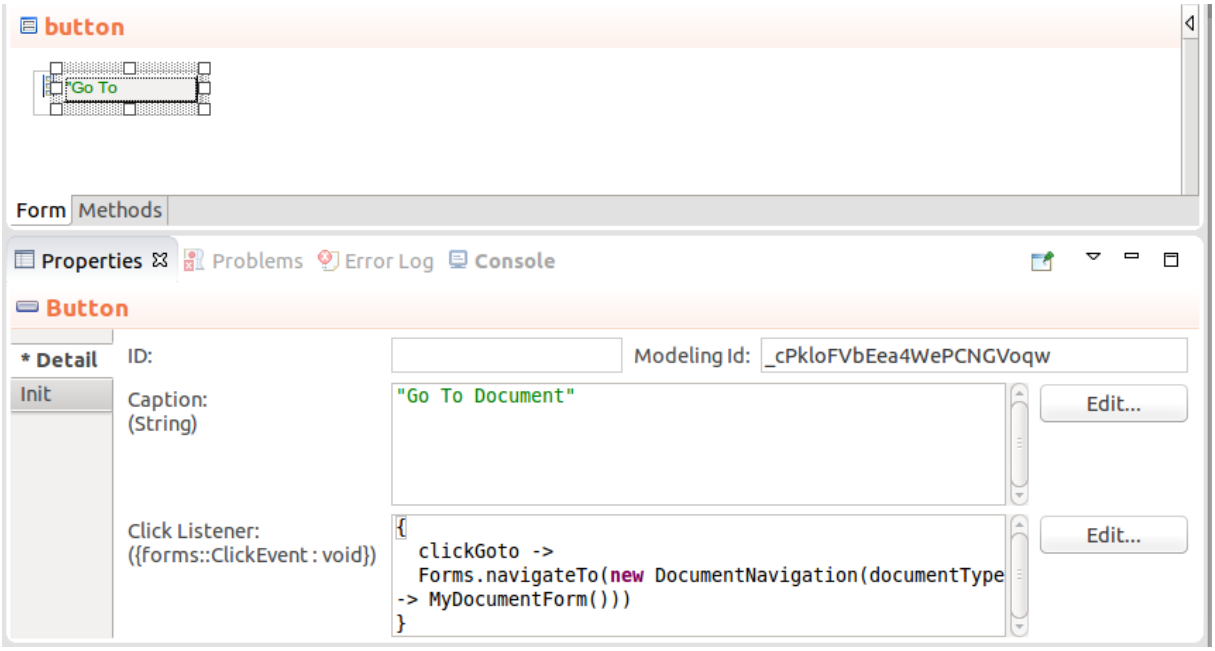


Figure 8.52 Form with a Button

Expression that returns a Button:

```
new forms::Button("Submit",
{ submitClick -> Forms.submit();
Forms.navigateTo(new DocumentNavigation(documentType -> MyForm()))})
```

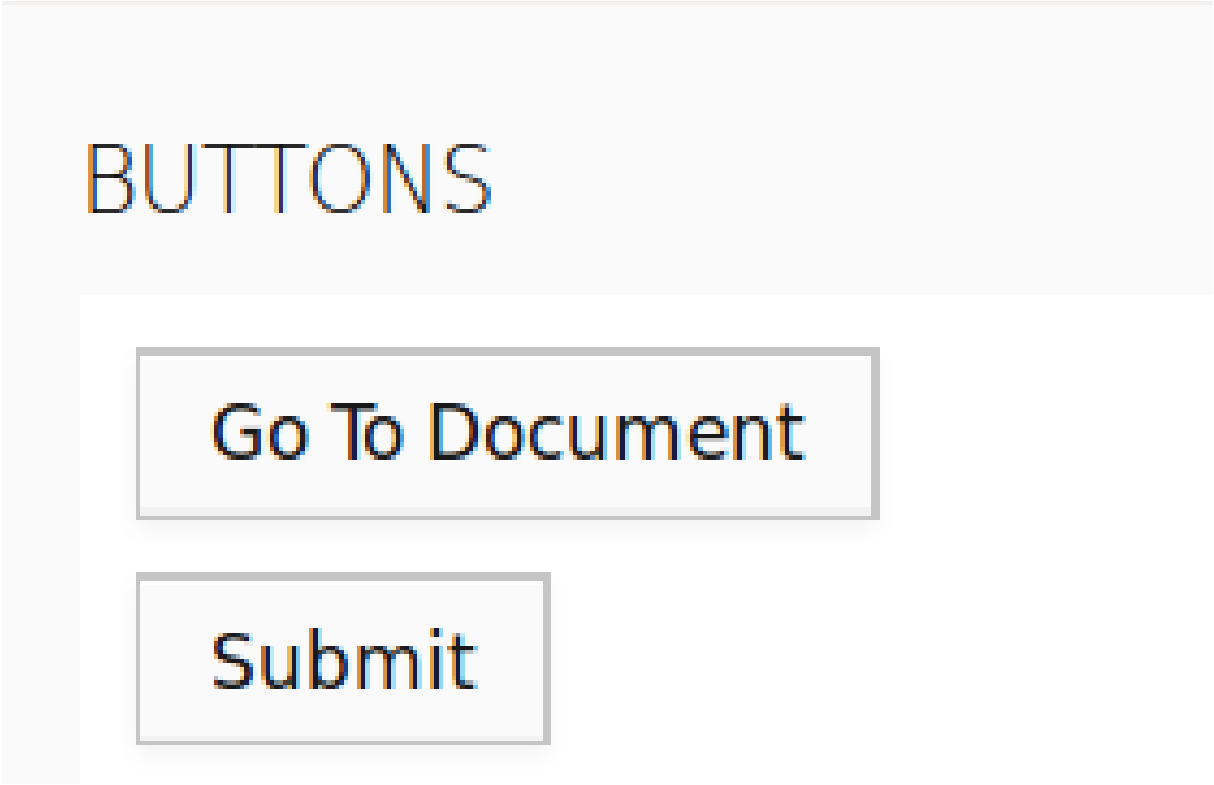


Figure 8.53 Button rendered in the default theme

8.4.2 Action Link (forms::ActionLink)

The *Action Link* component ( ) is a link that performs an action when clicked (it is a Button rendered as a link).

Action Link defined as an expression

```
 new ActionLink(
   "caption from constructor",
   { e -> Forms.navigateTo(new UrlNavigation(openNewTab -> true, url -> "https://www.whitestein.co
)
```

Figure 8.54 Action Links rendered

8.4.3 Link (forms::Link)

A *Link* component ( ) serves to navigate away from the page on click: it is rendered as <a> with the *href* attribute.

The advantage of a *Link* over an *Action Link* is that you can open a target location in a new tab or window. Also you can calculate the navigation link lazily in the Click Listener expression.

The component defines the following:

- **Navigation**: an expression that returns the target location as a Navigation object, for example, *DocumentNavigation*, *TodoNavigation*, *UrlNavigation*, etc.

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

Note: To create a link that will download a resource, use the Download component.

8.4.3.1 Opening a Link Resource in a New Window or Tab

To open the target location in a new tab or window, set the *OpensNewWindow* on the Link component to true by calling the *setOpensNewWindow(true)* method.

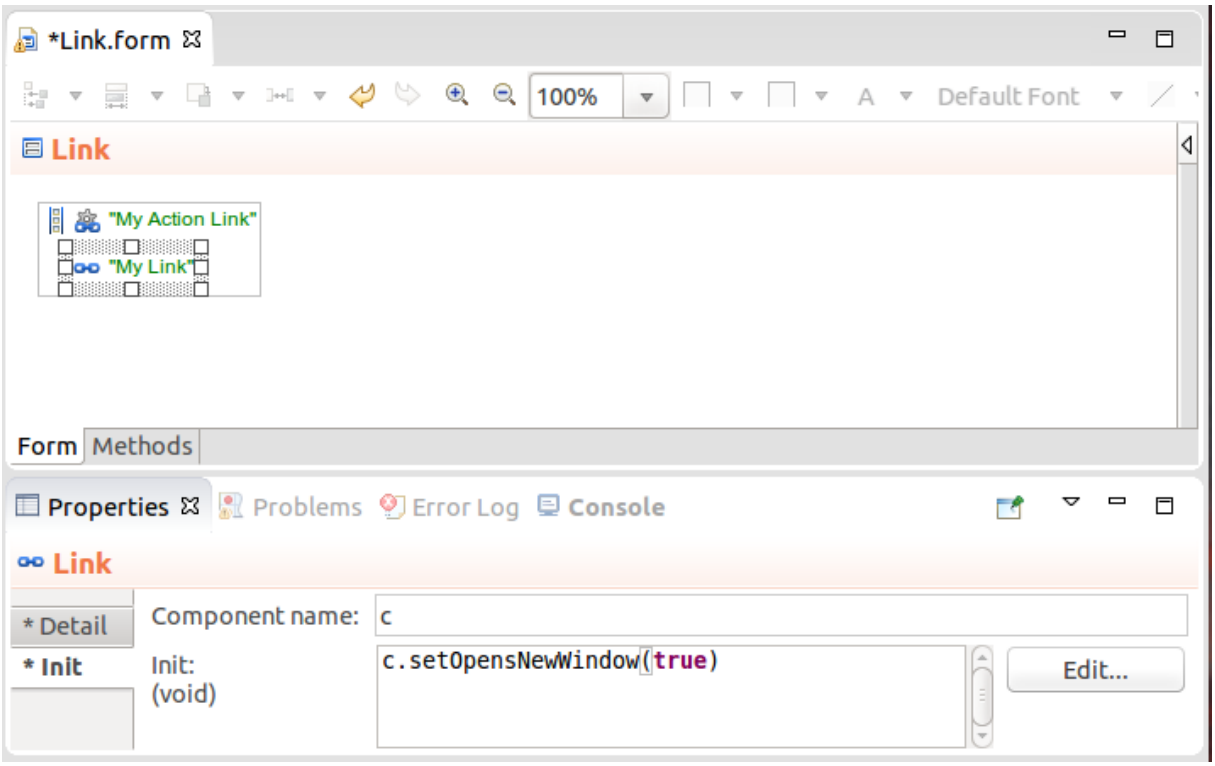


Figure 8.55 A Link component that opens a new tab or window

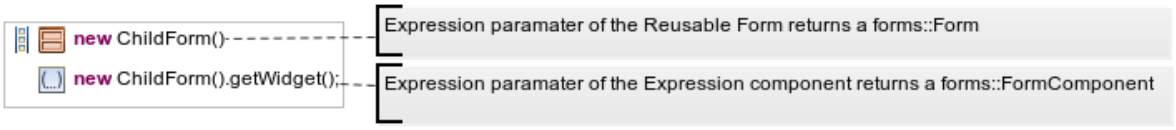
8.5 Special Components

8.5.1 Expression Component (forms)

The Expression Component serves to define a component as an expression of the Expression Language, be it an entire form tree or a single particular component. Note that the component per se is not represented by any record of the forms module since it always resolves to another component. As a result, the modeling ID of the Expression Component is not rendered in the form.

The *Expression* component allows the user to define a form component in the *Expression Language*: the component defines an expression that returns the required form component.

> **Note:** If the expression returns a forms::Form, consider using the Reusable Form so you don't have to issue `getWidget()` calls.



8.5.1.1 Creating a Form Component Programmatically

To create a component in the Expression Language, insert the Expression Component into the form and define the expression in the Expression property: the expression must return a single componet which is included in the form tree at the given location.

```
def TextField textField := new TextField("Text Field Caption", new ReferenceBinding(&varString));
~
textField.setImmediate(true);
textField.setValue("Binding value set in the reference string object.");
textField.addValidator({value:Object ->
  if length(value as String) != 4 then
    "The value must contain 4 characters";
  end;
});
~
def Button submitButton := new Button("Submit", {e -> Forms.submit()});
~
def Button navigateButton := new Button("Navigate", {e -> Forms.navigateTo(new UrlNavigation(url
~
new FormLayout(textField, submitButton, navigateButton);
```

Note that the root component of the expression tree adopts the modeling id of the Expression component; if you call `setModelingId()` on the root component in the expression, the setting is ignored. Also note that the modeling ID set on the Expression Component is not used in the rendered form.

8.5.1.2 Inserting Form into a Form Component Dynamically

To insert a form content into a component, you need to get the top component of the form definition, called the widget. You can obtain the widget with the `getWidget()` call of a form.

Changing the content of the documentInputLayout layout component on button click

```
myButton.setOnChangeListener({_ ->
myVerticalLayout.removeAllComponents();
  switch myButton.getValue()
    case common::DocumentType.ACCOUNT -> myVerticalLayout.addComponent(new accountForm().getWidget
    case common::DocumentType.LOAN -> myVerticalLayout.addComponent(new loanForm().getWidget());
    case common::DocumentType.MORTAGE -> myVerticalLayout.addComponent(new mortageForm().getWidget
  end
});
```

8.5.2 Reusable Form Component

The *Reusable Form* component serves to use an already defined Form in another Form. The Form created by the Reusable Form component is then rendered as part of the parent Form. The Form to be reused is defined by the Expression property of the *Reusable Form* component.

Example of Expression property on a Reusable Form component

```
new myCustomForm(currentBussinessData);
```

Chapter 9

Migration of Forms from ui to forms Implementation

Before migrating forms of the ui module to the forms module, consider the impact of the following changes:

- In migrated forms, the initialization of local variables is moved to the form constructor.
- All listeners with their properties are removed (any validation, handle, action properties are lost).
- If you are using the form in a document or a user task, the usage expressions are not migrated (they will still use the original form)

Migration creates a backup file of all forms so you can refer to it to identify data that was not migrated.

To migrate forms from ui to forms, do the following:

1. Prefix all data types from the *ui* module: This will prevent name clashes with the names in the *forms* module:
   (a) Go to **Project** > **Migrate LSPS 3.1 Projects**.
   (b) In the *Migration* dialog, select the projects and click **Next**.
   (c) Select **Add ui:: prefix** (unselect any other irrelevant options) and click **Finish**.
2. Import to your Modules the *forms* Module: In GO-BPMN Explorer, double-click Module Imports, click Add and double-click the forms Module in the Standard Library node.
3. In GO-BPMN Explorer, right-click either the form or the Module with the forms you want to convert and select **Convert to FormComponent**.
4. When converting all forms in a Module, define the names of the forms in the displayed dialog box.

9.1 Changes in Forms

Main Benefits

- **No fixed event-processing lifecycle**: the user specifies the behavior with actions when they see fit. Part of the behavior, such as, Submit and Refresh, is now defined in methods of the Forms record. Component specific actions, such as, validation, became methods of the Form components.

> **Note:** In *UI* forms, form actions are defined as callback expressions: the user must be aware of when the callbacks are used. This depends on the `event-processing lifecycle` and related mechanisms, which are relatively complex.

- **Simple changing of the tree component dynamically** (changes of the tree on runtime) since Form components are Records with methods (ui forms are functions)

Improvements on Components

- *Grid and Table* support row selection: the form can handle the row-selection event.
- *Popup* is now defined out of the form component tree and can be reused.
- *TextField*, *DecimalField* and *DateField* substitute the *ui::TextBox* component.
- *Chart* design has been improved significantly.
- *Pie Chart* can now be rendered as a Donut chart.
- *Charts* are now more flexible and dynamic: you can add data points on runtime without reloading the charts.

New Components

- Various new components, including *PasswordField*.

Removed Components

The approach eliminates the need for the following components:

- **ui::ViewModel**: the "undo/cancel" capability can be now achieved with `Record Change Proxies` in more controlled and transparent manner (the proxy mechanism is a generic modeling mechanism; an example is available in Quickstart.
- **ui::Container** and public-listener and registration-points mechanism: forms can define public methods to allow access from "outside".
- **ui::Conditional**: substituted with the *setVisible()* method of components.
- **ui::Geolocator**: replaced by the *Forms.detectLocation()* method.
- **ui::Message**: the user can now display the messages on failed validation easily in the entire form.

Missing Features

- Inserting a new value in the Combo-Box component
- Saving of documents and to-dos
- Support of Custom components in PDS (It is still possible to implement your component and create it from the Expression Component as a Record instance.)

Feature	forms	ui

Feature	forms	ui
General modeling approach	form behavior defined by expressions and their methods	behavior defined in GUI of the forms editor (listeners)
Properties	partial support by the GUI of the form editor; fully supported as expressions	support by the GUI of the form editor
Actions	partial support by the GUI of the form editor (if defined directly on the form component); fully supported as expression anywhere else	Predefined lifecycle with actions; user defines callbacks for the lifecycle (they must be aware of the lifecycle mechanism, which is relatively complex)
Values	Given either by binding to variable, data source or internal value of the component.	By binding to variable.
Iterating values (collections in UI)	By data providers.	By iterators which are usually local variables.

Chapter 10

Vaadin Upgrade

Forms make use of the Vaadin framework: by default, the LSPS Application uses Vaadin 7. Since 3.3.2082, it is possible to switch to Vaadin 8: mind that Vaadin 8 no longer supports Table and Table Tree, and you will need to tranform your Tables and Table Tree to another components, such as the Grid.

The process differs slightly on whether you have an application that was generated in an LSPS that supports Vaadin 8; that is, dependig on whether you generated your application in an LSPS version prior or after 3.3.2082.

Important: In the Vaadin 8 implementation, the following components are not supported:

- Calendar
- Table (use Grid)
- Tree
- TreeTable

10.1 Upgrading to Vaadin 8 from Before Vaadin 8 Support

To upgrade to Vaadin 8 your custom application which was generated prior to Vaadin 8 support was added, do the following:

1. Install the LSPS maven repository to the repository used by PDS.

   If you are using the system maven repository, it is typically located in your home directory. Run `cd ~/.m2/; unzip <DOWNLOADLOCATION>/lsps-repo<VERSION>.zip`

2. Open <applicationName>-vaadin/pom.xml

   (a) Exclude `lsps-human-vaadin-v7` from the *lsps-human-app* dependency.

```
<dependency>
 <groupId>com.whitestein.lsps.human-processes</groupId>
 <artifactId>lsps-human-app</artifactId>
 <!-- ADD:-->
 <exclusions>
    <exclusion>
      <groupId>com.whitestein.lsps.human-processes</groupId>
      <artifactId>lsps-human-vaadin-v7</artifactId>
    </exclusion>
 </exclusions>
</dependency>
```

(b) Add the following dependency as the last dependency:

```
<dependency>
       <groupId>com.whitestein.lsps.human-processes</groupId>
       <artifactId>lsps-human-vaadin-v8</artifactId>
</dependency>
```

3. Add the following dependency to the root pom in <applicationName>/pom.xml:

```
<dependency>
 <groupId>com.whitestein.lsps.human-processes</groupId>
 <artifactId>lsps-human-vaadin-v8</artifactId>
 <version>${lsps.version}</version>
</dependency>
```

4. Modify the following java classes in your <applicationName>-vaadin project:

   (a) Make `DefaultLspsAppConnector` in the *connectors* extend `LspsAppConnectorImplV8` instead of `LspsAppConnectorImpl`.

   (b) Change its `getFormsComponentFactory()` as follows:

```
@Override
 public FormComponentFactoryV8 getFormsComponentFactory() {
   return new LspsFormComponentFactory();
}
```

   (c) Make `LspsFormComponentFactory` extend `FormComponentFactoryV8` instead of `Form↩ ComponentFactory`.

   (d) Make `LspsUIComponentFactory` extend `UIComponentFactory_PureV8Impl` instead of `UIComponentFactoryImpl`.

   (e) In *AppLspsUI*, add `@Widgetset("com.whitestein.lsps.vaadin.widgets.Widget↩ Set")` annotation above the class declaration.

5. Rebuild your application.

10.2 Upgrading to Vaadin 8

To upgrade to Vaadin 8 your custom application which was generated after the Vaadin 8 support was added, do the following:

1. Install the LSPS maven repository to the repository used by PDS.

   If you are using the system maven repository, it is typically located in your home directory. Run `cd ~/.m2/; unzip <DOWNLOADLOCATION>/lsps-repo<VERSION>.zip`

2. Open <applicationName>-vaadin/pom.xml

   (a) Exclude `lsps-human-vaadin-v7` from the *lsps-human-app* dependency.

```
<dependency>
 <groupId>com.whitestein.lsps.human-processes</groupId>
 <artifactId>lsps-human-app</artifactId>
 <!-- ADD:-->
 <exclusions>
    <exclusion>
      <groupId>com.whitestein.lsps.human-processes</groupId>
      <artifactId>lsps-human-vaadin-v7</artifactId>
    </exclusion>
 </exclusions>
</dependency>
```

   (b) Uncomment the following dependency:

```
 <dependency>
        <groupId>com.whitestein.lsps.human-processes</groupId>
        <artifactId>lsps-human-vaadin-v8</artifactId>
    </dependency>
```

3. Add the following dependency to the root pom in <applicationName>/pom.xml:

```
<dependency>
  <groupId>com.whitestein.lsps.human-processes</groupId>
  <artifactId>lsps-human-vaadin-v8</artifactId>
  <version>${lsps.version}</version>
</dependency>
```

4. Modify the following in your <applicationName>-vaadin project:

   (a) In *AppLspsUI*, uncomment `@Widgetset("com.whitestein.lsps.vaadin.widgets.WidgetSet")` annotation above the class declaration.

   (b) Remove the *connectorsV7* package.

5. In *AppUiProvider.createLspsAppConnector()* comment out the "return new DefaultLspsAppConnectorV7(lspsUI)"; and uncomment `return new DefaultLspsAppConnector(lspsUI);`

6. Rebuild your application.