WHITESTEIN
Technologies

Living Systems® Process Suite

LSPS Tutorials

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 series of complete tutorials that focus on different goals you might want to achieve in your models:

- Forms Tutorials and UI Forms Tutorials to help you create the GUI you require
- Process Tutorials with design patterns for your business processes
- Data Model Tutorials with solutions for your data models and related issues
- Other Tutorials

Chapter 2

Forms Tutorials

2.1 Chart (forms)

You can download an example implementation here. To import it to your workspace, go to **Import** > **Existing Projects into Workspace**; select **Select archive file** and locate the *charts.zip* file and select **ChartTutorial**.

2.1.1 Creating a Donut Chart

A donut chart is a special case of the pie chart: the pie slice series have the inner size plotting property larger than 0 and a custom size. Therefore, to create a donut chart, we will start from the pie chart.

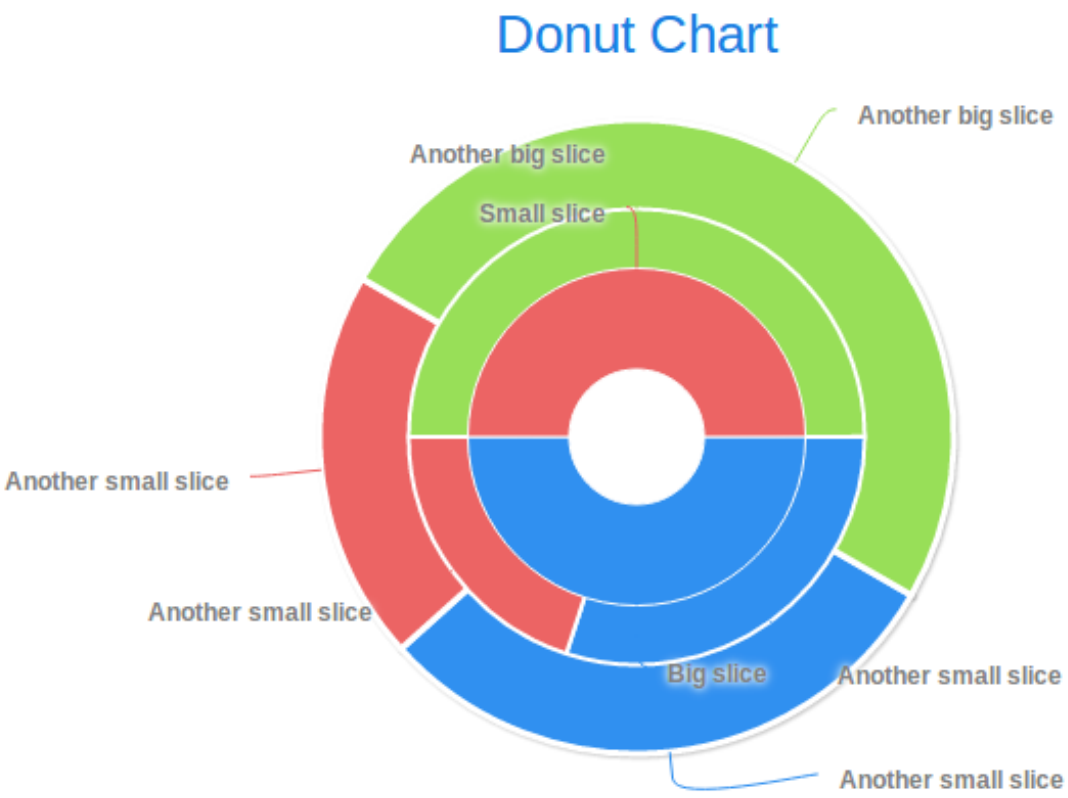
To create a donut chart, do the following:

1. Insert a pie chart component into the form.
2. Define the data series: you can do so either in the *Pie slice series* property in the Properties view of the Pie Chart or using the *addPieSliceSeries()* method.

```
def PieSliceSeries innerSeries := new PieSliceSeries("Inner Slice Series", [
       new forms::PieSlice("Big slice", 10),
       new forms::PieSlice("Small slice", 10)
     ]
  );
def PieSliceSeries middleSeries := new PieSliceSeries("Middle Slice Series", [
       new forms::PieSlice("Another small slice", 30),
       new forms::PieSlice("Another small slice", 20),
       new forms::PieSlice("Another big slice", 50)
     ]
  );
def PieSliceSeries outerSeries := new PieSliceSeries("Outer Slice Series", [
       new forms::PieSlice("Another small slice", 30),
       new forms::PieSlice("Another small slice", 20),
       new forms::PieSlice("Another big slice", 50)
    ]
  );
//The dataseries as rendered in the order (z-index) as in the returned list
//(outerSeries is the lowest):
[ outerSeries, middleSeries, innerSeries ]
```

3. Define the required inner size and total size in the plotting options for each series.

```
....
def PlotOptionsPie middleSeriesPlotting := new PlotOptionsPie(
  //inner border at the perimeter of the chart:
  innerSize -> new AttributeSize("20%"),
  size -> new AttributeSize("60%"),
  startAngle -> 90, borderWidth -> 2);
~
innerSeries.plotOptions := middleSeriesPlotting;
[ outerSeries, middleSeries, innerSeries ]
```



2.1.2 Creating a Bar Chart

To render a data series as a bar chart, do the following:

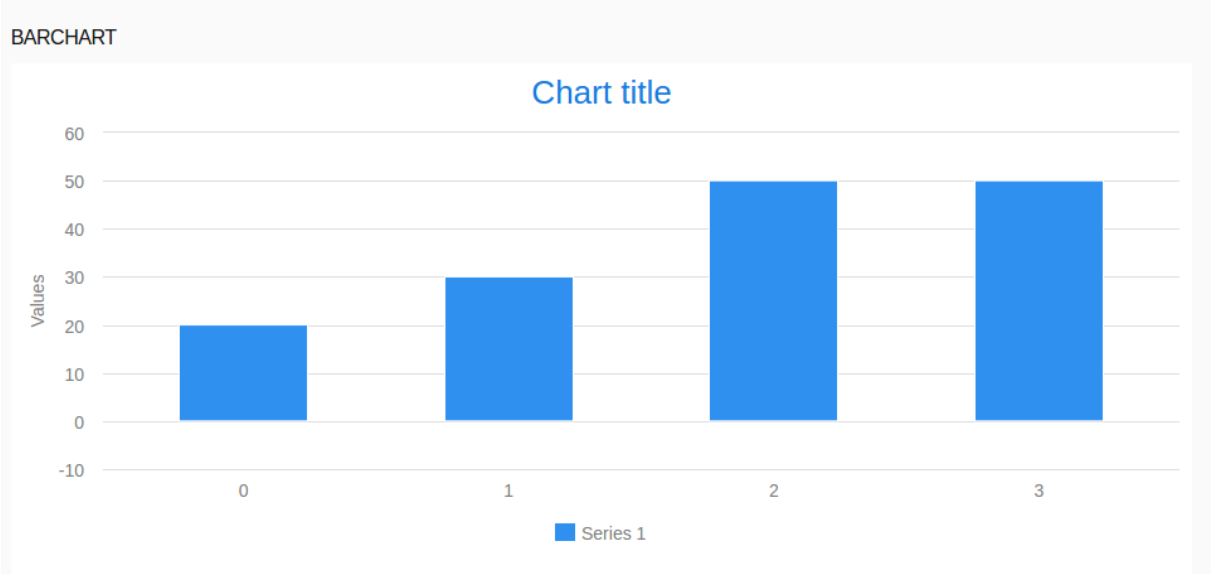
1. Insert the Cartesian Chart component into your form.
2. Define its data series in its Properties view.
3. Set the `plotOptions` property of the data series to `PlotOptionColumn`.

```
myDataSeries.plotOptions := new PlotOptionsColumn(
          dataLabels -> new DataLabels(formatter -> "this.y"),
          color -> new Color(0, 255, 150));
```

Note that *CategoryDataSeries* are plotted as bar charts by default.

Example TimeDataSeries plotted as a bar chart

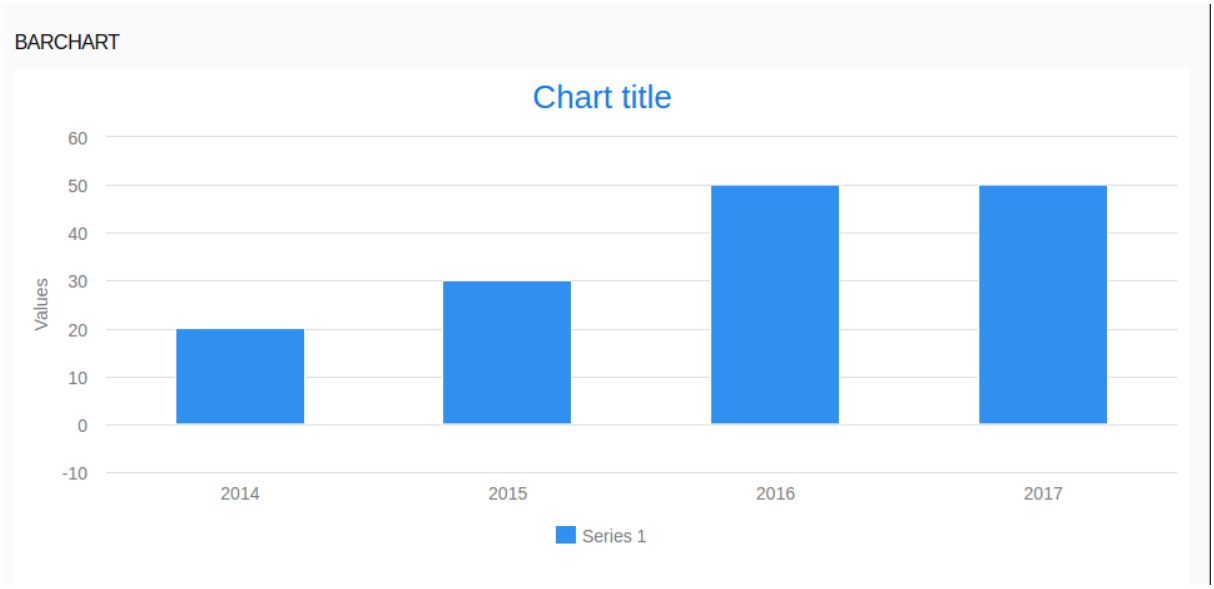
```
def TimeDataSeries tds := new TimeDataSeries([
   new TimeDataSeriesItem(now(), 1, 3),
   new TimeDataSeriesItem(now()+seconds(3), -1, 5),
   new TimeDataSeriesItem(now()+seconds(5), 2, 7)]
);
tds.plotOptions := new PlotOptionsColumn(range -> true);
[tds]
```



2.1.2.1 Setting the Category as X Axis Values

To use the category name on the x axis as values, set the x axis to an axis with `category -> []`. You can do so directly in the X Axis field on the Detail tab of the Properties view or anywhere by calling the *addXAxis()* method of the chart.

```
c.addXAxis( new Axis(categories -> []))
```

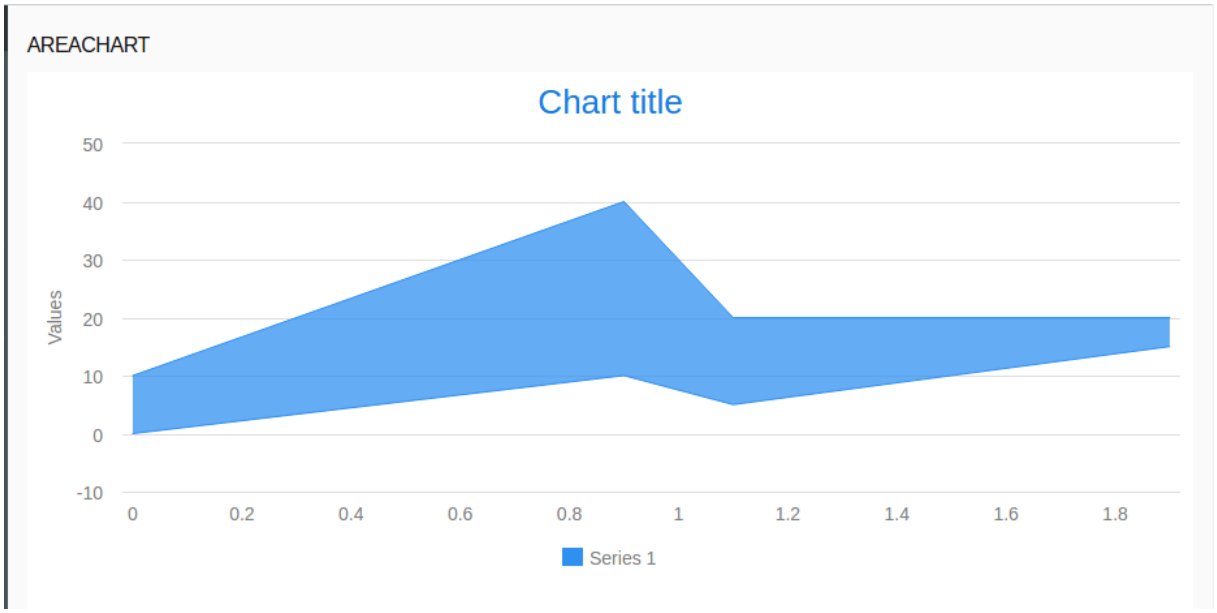


If you need to override some of the category names, define the names in the categories list of the chart; for example, `c.addXAxis( new Axis(categories -> ["Year 2014", null , "Year 2016"]))` overrides the category names of the first and third item.

2.1.3 Creating an Area Chart

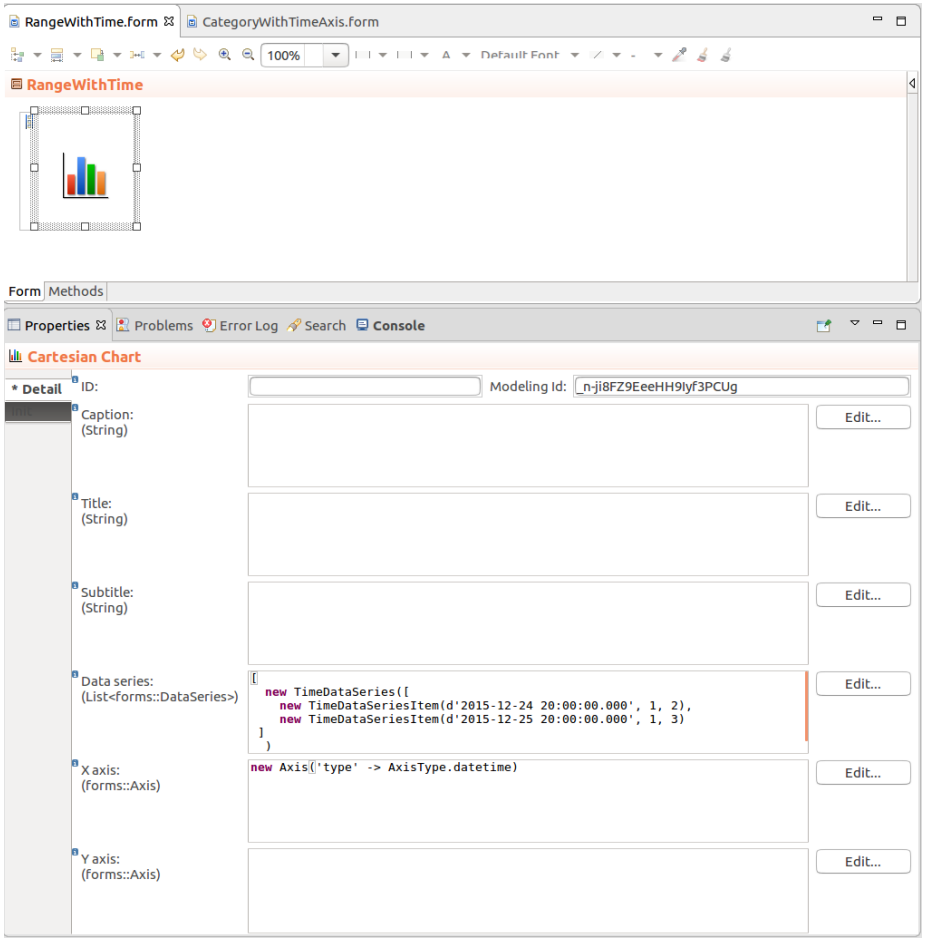
To render a data series as an area chart in the Cartesian Chart, set its plot options to PlotOptionsLineArea.

```
def CategoryDataSeries cds := new CategoryDataSeries(
  [
     new CategoryDataSeriesItem("first", 1, 4),
     new CategoryDataSeriesItem("second", -1, 2),
     new CategoryDataSeriesItem("third", 0, 3)
  ]
);
cds.plotOptions := new PlotOptionsLineArea(spline -> true, range -> true);
[ cds ]
```



2.1.3.1 Creating an Area Chart with Time X Axis

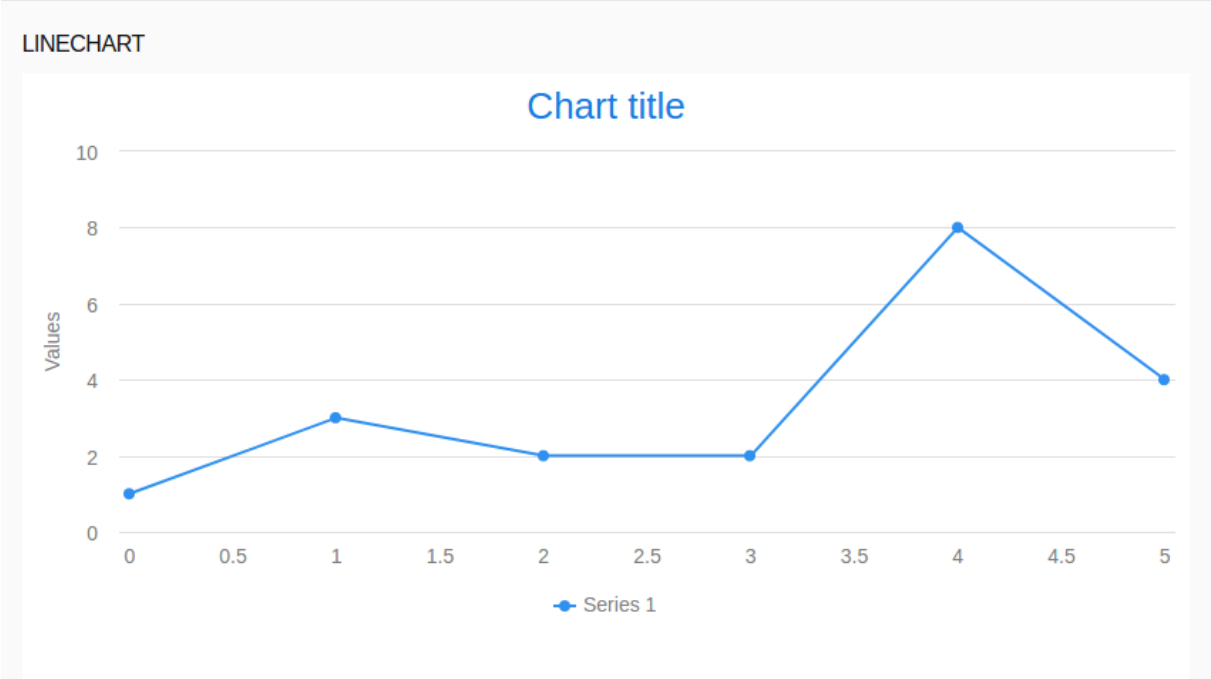
To create an area chart that will have time axis, use the TimeDataSeries in the Cartesian Chart and set the type of the x axis to `AxisType.datetime`.



2.1.4 Creating a Line Chart

By default, *ListDataSeries* are rendered as line charts. If you want to plot another series type as a line chart, set its plot options to *PlotOptionsLine*.

```
cds.plotOptions := new PlotOptionsLine(spline -> true);
[ cds ]
```



2.2 Validating a Record from a Form

In this tutorial, you will:

- create a page that will add an entry to the database and log the event,
- validate values of a record defined as user input in a form, and
- start a process when a user performs some action on a page.

Requirements:

- Create an order based on user input.
- Make sure the user enters all data in the correct format.
- Make sure the order is persisted only after the user submits it.

1. **Create a structure with an executable *order-placing* module:**
   (a) Open the *Modeling* perspective.
   (b) Go to File -> New -> GO-BPMN Project.
   (c) In the pop-up enter the project name *OrderProcessing* and click Next.
   (d) In the *Module name* field, enter `order-placing` and click **Next**.
   (e) Click **OK** and **Finish**.
2. **Create the data definition with the *Order* shared record with the following fields:**
   - `item` of type String
   - `price` of type Decimal
3. **Create the form with the content for the Order page:**

(a) Create a form definition; make sure to select **Use FormComponent-based UI**.

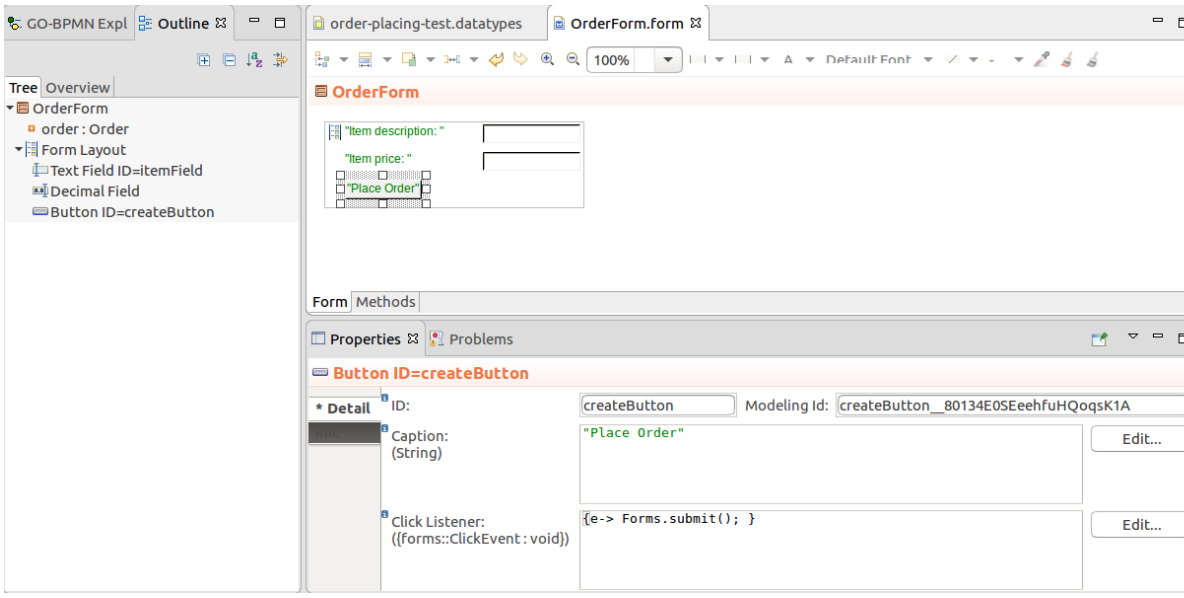
(b) Create a form variable `order` of the type `Order` for the data of the new order.

(c) On the *Methods* tab, define a form constructor that initializes the variable.

```
public OrderForm(){
 order := new Order();
}
```

(d) Insert the following components as displayed below and define the components' properties in their Properties views:

- Form Layout
- Text Field with the properties:
  - ID: `itemField`
  - Caption: `"Item:"`
  - Binding: Reference to the field of the order variable `&order.item`
- Decimal Field with the properties:
  - ID: `priceField`
  - Caption: `"Price:"`
  - Binding: Reference to field of the order variable `&order.price`
- Button:
  - ID: `createButton`
  - Caption: `"Place Order"`
  - Click Listener: Submit on click `{e-> Forms.submit(); }`



4. **Create a document definition with the Order document:** set the UIDefinition to return your form:

```
new OrderForm();
```

At this point, you have a runnable model: if you run it, you will realize that the order entry in the database is created at the moment the model instance is created, that is, at the moment you open the Document and the *order* variable is instantiated. However, you want to create the entry only after the order data is validated so no bogus data is persisted in the database. This will allow you to put the cancelling mechanism in place: you will be able to leave the document without having persisted any data.

This mechanism can be implemented using *change proxies*: `Change proxies` hold preliminary versions of shared Records: the values of change proxies are not reflected in the database. Change proxies exist in sets called proxy sets. To *apply values of proxies* and create the actual shared record instances, you *merge their proxy set*.

You can create a change proxy not only for an existing instance of a shared record but also for a shared record type; the shared record instance does not exist when you are creating the proxy. It is *created* when its proxy set is merged.

1. **Make the *order* a proxy:**

   (a) Create a form variable *recProxySet* of type *RecordProxySet* and initialize it from the form constructor before the *order* variable.

   ```
   recProxySet := createProxySet(null);
   ```

   (b) Also from the form constructor, adjust the initialization of the *order* variable to be a change proxy over the order shared-record type:

   ```
   order := recProxySet.proxy(Order);
   ```

   (c) In the *Click Listener* expression of the *createButton*, add the merge of the proxy set with the proxy object.

   ```
   { e ->
        recProxySet.merge(false);
        Forms.submit()
   }
   ```

   (d) Add a **Cancel** button that will navigate away from the screen, for example,

   ```
   { e -> Forms.navigateTo(new UrlNavigation(openNewTab -> false, url -> ""))}
   ```

2. **Validate the *order* data:**

   (a) Create a constraint definition and define the constraints for the fields of the Order record.

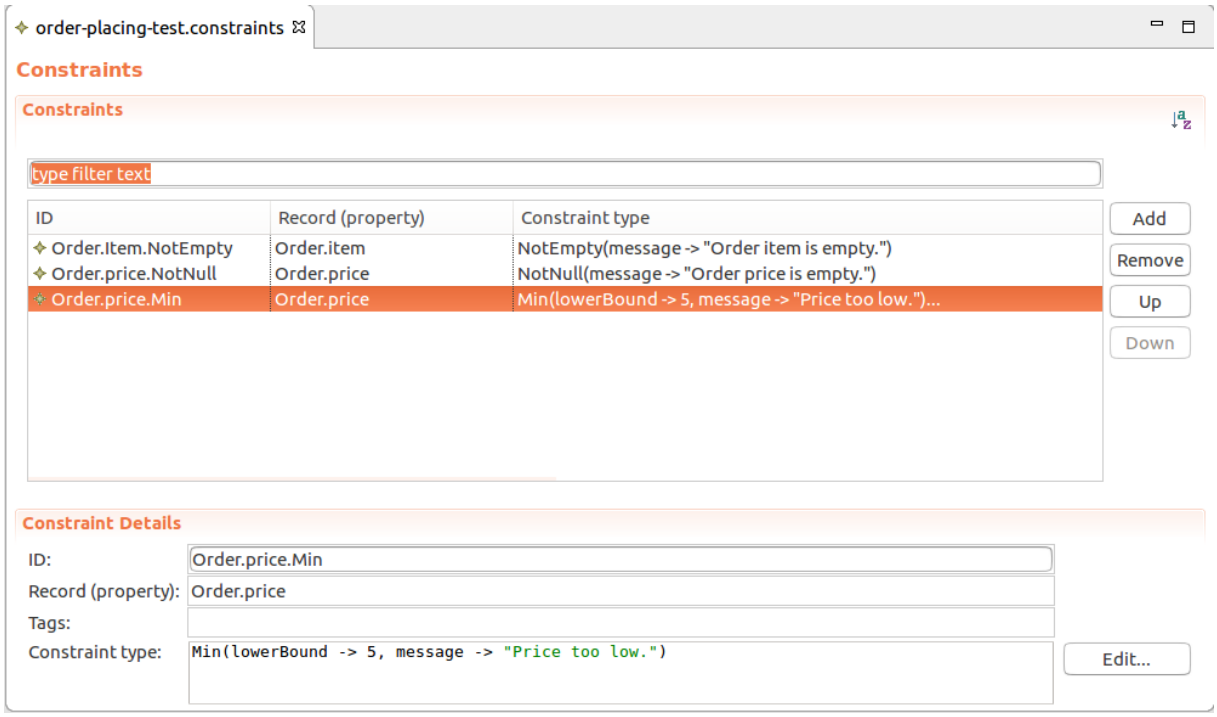


Figure 2.1 Constraints for the Order Record

   (b) To check if the input meets the defined constraints, trigger validation of the order on click of the createButton: in the Click Listener expression, call the `validate()` function on the order variable and handle the returned error messages:

   ```
   { e ->
    def List<ConstraintViolation> errors :=  validate(order, null, null, null);
     if errors.isEmpty() then
       recProxySet.merge(false);
       Forms.submit();
     else
   ```

```
        //Displays the errors on the createButton
        //if the user never enters any value in the item
        //and price field:
        showDataErrorMessages(errors, createButton)
      end
    }
```

(c) Now the messages from constraints are all displayed on the createButton; Enable displaying of the messages on the respective input components by calling `c.inferValidator(null)` on the input fields.

3. Upload the module and test the document:

   (a) Make sure the server is running.

   (b) Right-click the module and go to Upload As -> Model

   (c) Go to `http://localhost:8080/lsps-application` and log in.

   (d) Click **Documents** in the menu on the left.

   (e) Test the Order page.

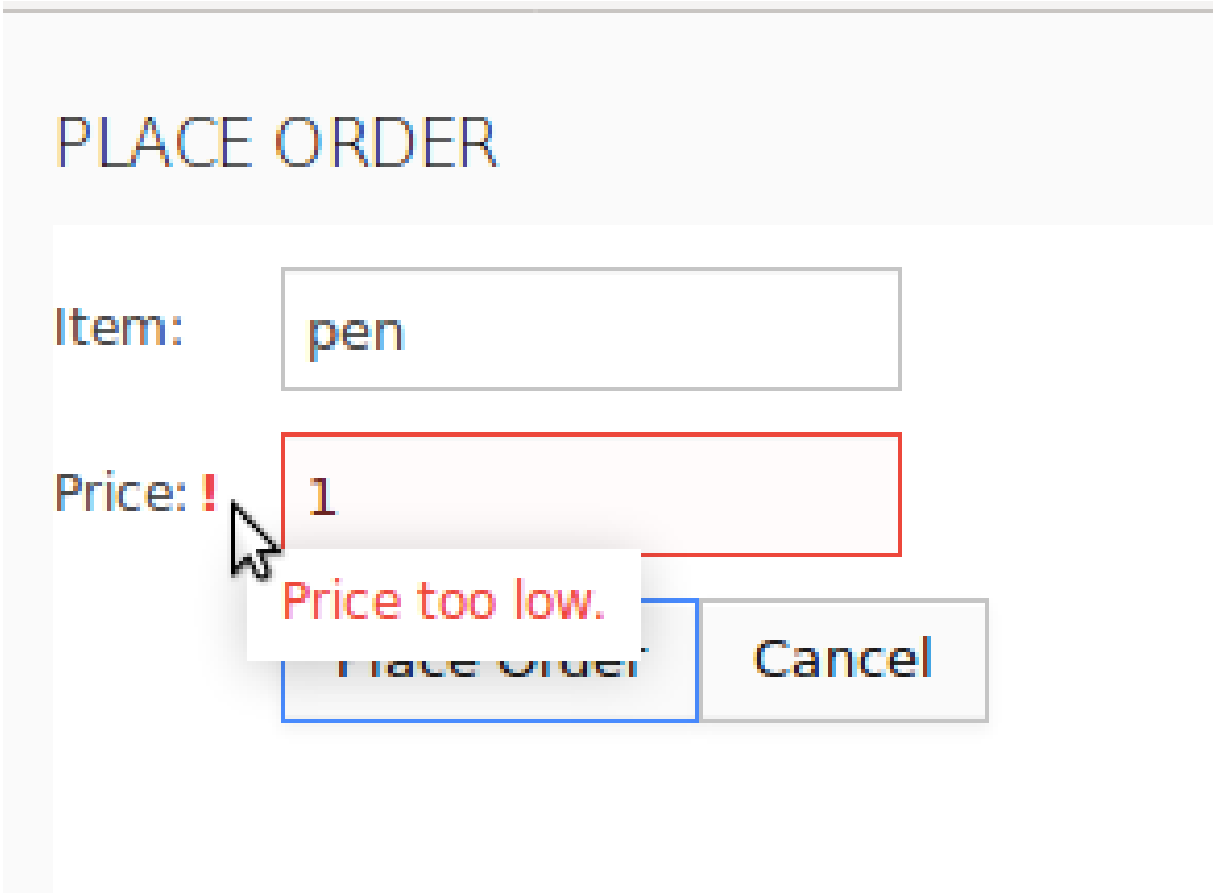


Figure 2.2 Order page

2.2.1 Log Process

You will now extend the *Order* page to instantiate a process that will log a message when the user places an order.

Note that you could simply call the `log()` function from the UI definition when the user performs some action, too. However, for demonstration purposes, you will run a BPMN Process, which could potentially execute other actions.

1. **Create the logging process:**

   (a) Create a *logging* module with a BPMN process.

   (b) In the graphical editor with the process file, right-click into empty space on the canvas and under **New** select the None Start Event.

   (c) Drag the quicklinker icon next to the None Start Event to a spot where you want to insert the next process element, the Log task.

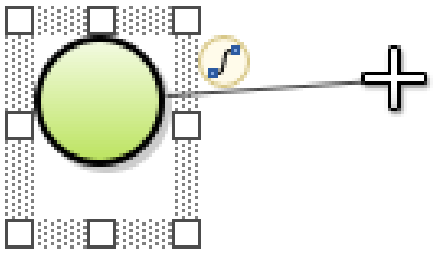


Figure 2.3 Dragging quicklinker

   (d) In the context menu, select Task and then Log task.

   (e) On the *Parameters* tab in the Properties view of the task, define the message that should be logged and its message level.

   (f) Connect the task to a Simple End Event.

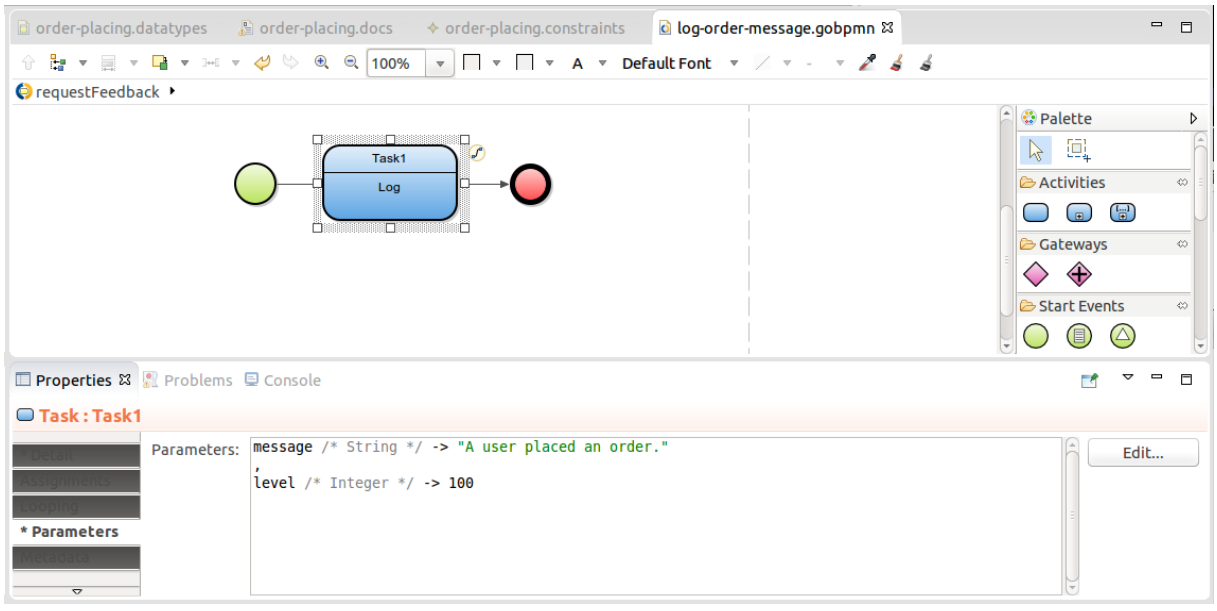


Figure 2.4 Finished process

2. **Instantiate the logging model from the *createButton*:** To the *Click Listener* of the createButton, add the *createModelInstance()* function call Note that you can pass a process entity from the call if the process needs to work with a shared record from the document, in our case the *order*.

```
{ e ->
def List<ConstraintViolation> errors :=  validate(order, null, null, null);
 if errors.isEmpty() then
  //when the form is valid, the shared record instance is created based on the proxy Order o
   recProxySet.merge(false);
   //creates a model instance of the order-placing module
   //which instantiates the log Process:
   createModelInstance(true, getModel("logging", "1.0"), order, null);
   Forms.submit();
else
   showDataErrorMessages(errors, orderButton)
 end;
```

3. Save the definitions and upload the modules.

4. To upload the *logging* module automatically with the *order-placing* module, import it to the order-placing module.

5. Go to the application and create an order from the document.

Let's check that the logging model with the process was instantiated:

1. Back in PDS, switch to the Management perspective.

2. Refresh the Models view: It now contains an entry of the logging model instance.

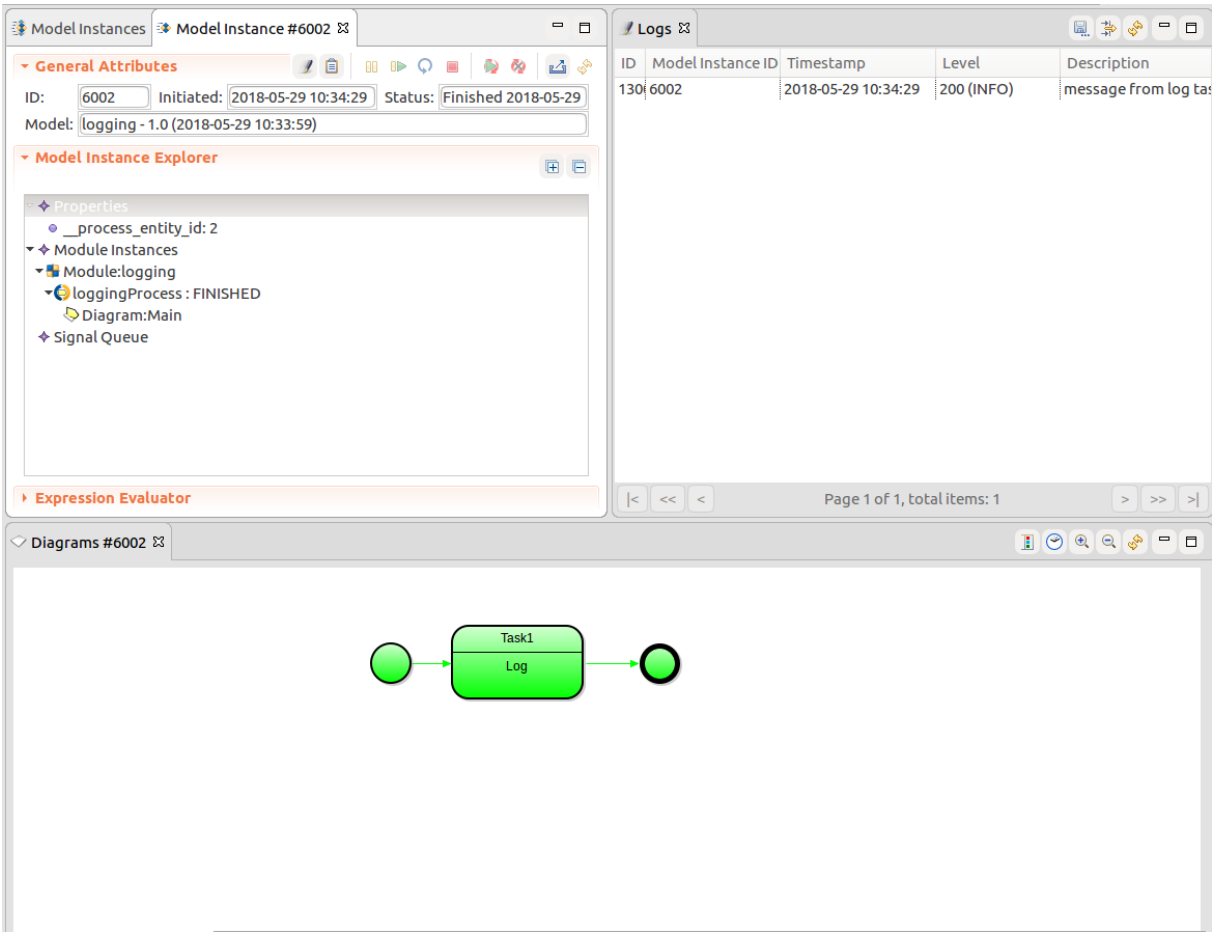


Figure 2.5 Model Instance details and live diagram\; note the process entity property in the properties tree node.

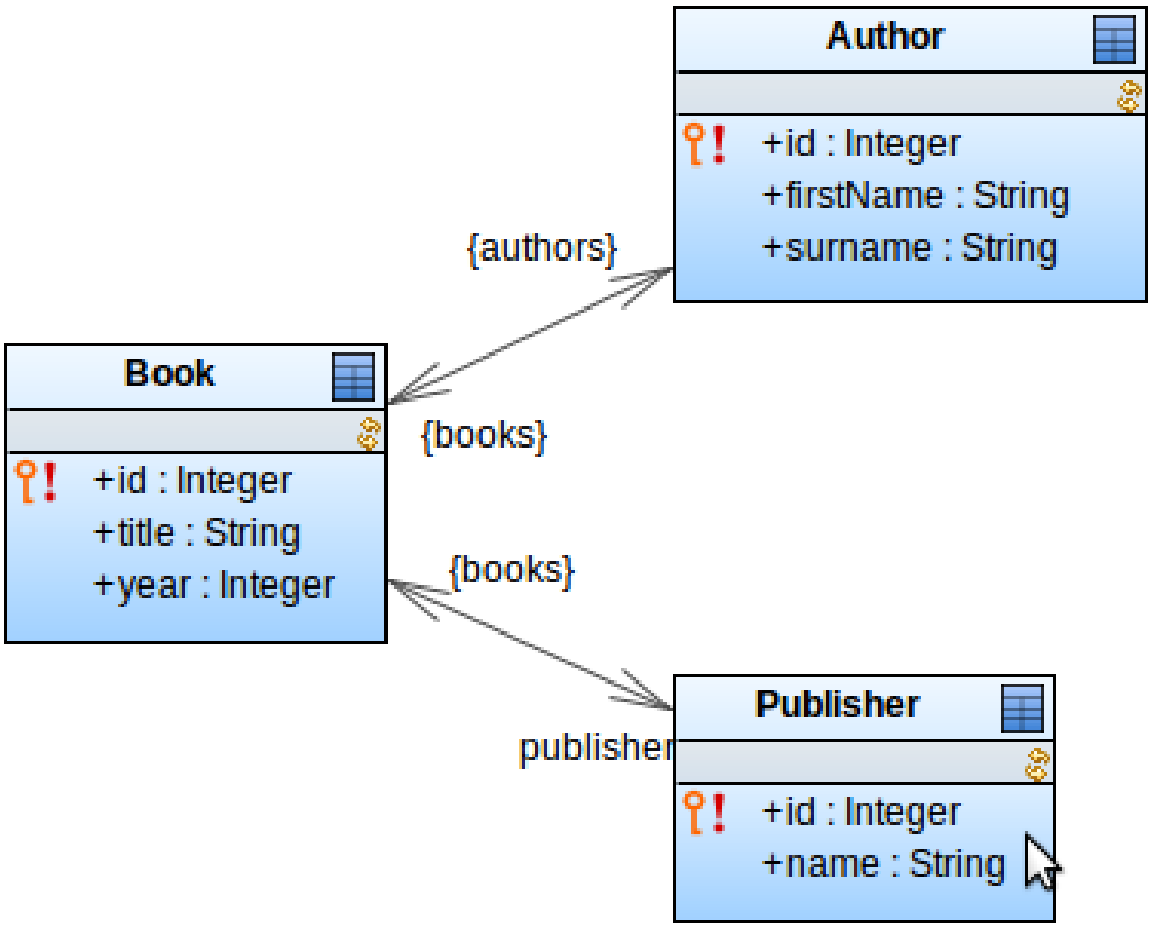
2.3 CRUD Grid

We will create a document with an editable overview of persisted entities: the user will be able to switch the entity type displayed in the grid, edit and delete any entry.

2.3.1 Creating Database Data

First, prepare the persisted data that will be displayed in the document:

1. Create a data type hierarchy of shared Records Book, Author, and Publisher. Create relationships between Book and Author, and Book and Publisher as depicted below.



2. Initialize database data, for example:

   (a) Create a process definition.

   (b) In the process, create a workflow that will be executed: for example, a None Start Event with an outgoing Flow to a Simple End Event.

   (c) On the Assignment tab of the Flow, define an expression that will initialize the database data, for example:

```
 def Author heller := new Author(firstName -> "Joseph", surname -> "Heller");
def Author vonnegut := new Author(firstName -> "Kurt", surname -> "Vonnegut");
def Author kerouac := new Author(firstName -> "Jack", surname -> "Kerouac");
~
def Publisher sas := new Publisher(name -> "Simon & Schuster");
def Publisher p := new Publisher(name -> "Putnam");
def Publisher dp := new Publisher(name -> "Delacorte Press");
def Publisher vp := new Publisher(name -> "Viking Press");
 
new Book(year -> 1961, title -> "Catch 22", authors -> {heller}, publisher -> sas);
new Book(year -> 1988, title -> "Picture This", authors -> {heller}, publisher -> p);
new Book(year -> 1973, title -> "Breakfast of Champions", authors -> {vonnegut}, publishe
new Book(year -> 1969, title -> "Slaughterhouse-Five", authors -> {vonnegut}, publisher -
new Book(year -> 1957, title -> "On the Road", authors -> {kerouac}, publisher -> vp);
```

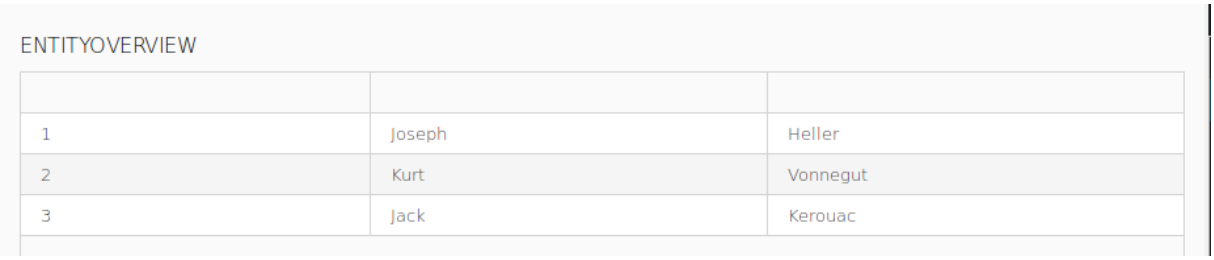
3. Run the module.

   The quickest way to test your models is to do the development testing on the PDS Embedded Server: click  to start it and connect PDS to the server, and then right-click the module and go to **Run As** > **Model** to upload the module and create its model instance.

2.3.2 Creating the Form

You will create a Grid over the shared Records that will display values of the Record fields:

1. Create a form definition.
2. In the form, insert the Vertical Layout and Grid component.
3. Define the Grid component properties:
   - Define its name as `EntityGrid`
   - Set its data source to *Type* and the value to `Author`
   - Select the *Editor enabled* flag in the Editing property.
   - On the Grid's Init tab, enable filtering by calling `c.setFiltrable(true)`
4. For each Author property, insert a Grid Columns into the Grid and set the Value Provider to *Property path* and insert the property path: you will insert columns for the `Author.id`, `Author.firstName`, and `Author.surname` property paths. Also select the **Editable** flag on each column.
5. If you have not done so yet, run PDS Embedded Server by clicking  , right-click the form and go to *Run As* > *Form Preview*: this will open a preview of the form in your browser.



   Note if you click a row, you can edit the entries: edits are reflected on the database when you save the edits or press Enter.

   The Grid is static and it will always display only Authors. However, the user should be allowed to change the type of entity displayed in the grid. When they select an entity type, that is, author, book, or publisher, the content of the Grid need to be updated. And not only the content: the grid columns have to be update so that the columns with the correct data are displayed. To achieve this, you will use the dynamic features of the forms.

6. First, let us externalize the setting of the displayed entity type:
   (a) Create a form variable `currentEntity` of type `Type<Record>` that will hold the entity the user selects.
   (b) On the Grid, set value of the data source to `currentEntity`.
   (c) Run preview of the form: right-click the form and go to *Run As* > *Form Preview*. The preview will fail with a runtime exception since the `currentEntity` variable is `null`.

   The first solution that comes to mind is to initialize the variable from a component higher in the hierarchy, in our case, the vertical layout component, during form initialization: However, this will result in the same exception because these expressions are executed after the form tree is initialized. You can check this in the form expression (right-click into the form and select **Display Widget Expression**): You need to initialize the variable sooner. You can do so in the form constructor:
   (d) Open the methods file of the form: the file is created automatically along with the form file and bears the same name.
   (e) Define a new constructor with the variable initialization:

```
 public EntityOverview(){
  currentEntity := type(Author);
}
```

7. Next adapt the columns and their value providers according to the `currentEntity` record. Do this dynamically; do not insert individual columns into the form (each entity has requires its own columns):

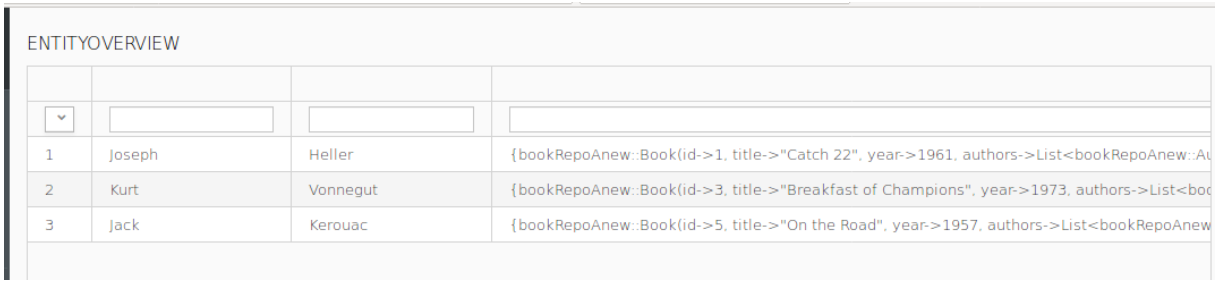
   (a) Delete the Grid Columns in the Grid.

   (b) On the Init tab of the Grid, define how to add the columns:

```
//get a list of properties of the entity in the currentEntity variable:
def List<Property> properties := currentEntity.getProperties();

foreach Property p in properties do
   c.addColumn(new PropertyPathValueProvider(p));
end
```

   (c) Run preview of the form: right-click the form and go to *Run As* > *Form Preview*.



   The next problem is how to deal with display columns for properties on related Records. Let us filter properties of these complex types: adjust the Init expression on the Grid as follows:

```
   def List<Property> properties := currentEntity.getProperties();
~
  foreach Property p in properties do
  //exclude properties with Records or Collections:
     if !(
          p.getPropertyType().isSubtypeOf(type(Record)) ||
          p.getPropertyType().isSubtypeOf(type(Collection<Object>))
          )
     then
        c.addColumn(new PropertyPathValueProvider(p), null,
            //Boolean sortable:
            null,
            // Boolean editable:
            true,
            // Editor editor:
            null);
     end
  end
```

8. Allow the user to change the value of the currentEntity:

   (a) Add a local variable `options` of type `Map<Object, String>` and initialize it from the constructor:

```
EntityOverview {
   public entityOverview(){
     currentEntity :=  type(Author);
     //added initialization of options:
     options := [Author -> "Author", Book -> "Book", Publisher -> "Publisher"];
   }
}
```

   (b) Add a Single Select List component above the Grid with the following properties:

   - *Binding* to Reference and its value to `&currentEntity`

- *Options* to Map and its value to `options`

(c) On the Init tab of the Single Select List component, define the action when the user selects an entity:

```
c.setOnChangeListener({ e ->
  //remove all columns:
  foreach forms::GridColumn c in EntityGrid.getColumns() do
    c.remove()
  end;
  //update the type data source of the grid:
  EntityGrid.setDataSource(new forms::TypeDataSource(currentEntity));
  //get list of properties of the entity record:
  def List<Property> properties := currentEntity.getProperties();
  //create colums for properties in the grid:
  foreach Property p in properties do
    if !(p.getPropertyType().isSubtypeOf(type(Record)) or
       p.getPropertyType().isSubtypeOf(type(Collection<Object>)))
    then
       EntityGrid.addColumn(new PropertyPathValueProvider(p));
    end
  end;
} );
c.setNullSelectionAllowed(false);
```

9. Since a part of the code runs when the user selects an option in the Single Select List and a part of the code that loads the Grid when initialized are identical, extract the code to a method. The concept is quite generic so you can define it as an extension method of the grid:

```
  @ExtensionMethod ~
public void setEntityColumns(Grid g, Type<Record> currentEntity) {
~
  g.setDataSource(new forms::TypeDataSource(currentEntity));
~
  def List<Property> properties := currentEntity.getProperties();
~
  properties.compact().collect(
    { p ->
       if !(p.getPropertyType().isSubtypeOf(type(Record)) or
            p.getPropertyType().isSubtypeOf(type(Collection<Object>)))
       then
         g.addColumn(new PropertyPathValueProvider(p), null,
            //Boolean sortable:
            false,
            // Boolean editable:
            true,
            // Editor editor:
            null
          )
       end
    }
  );
~
}
```

Adapt the assembly of columns on the grid and on the single-select list to `EntityGrid.setEntity↩Columns(currentEntity)`.

10. Add the column with the Delete button: in the *setEntityColumns* extension method, add the call `g.add↩ButtonColumn("Delete", { e:Record -> deleteRecords({e})});` to the end. If you have not defined the method, add the call to the code that creates the columns in the Single-Select component and to the Init code of the Grid.

There is one more issue to take care of and that is the headers of columns. Normally, to display a name of a Record or its property, you set the caption expression on each component with the `setHeader()` call. As this can be

pretty tedious, you can use labels to pass the caption expression. Labels are defined on records and fields and intended to hold their user-friendly name:

1. Set the labels on Field of the Author, Book, and Publisher Records.
2. Set the column header to the label value: if you defined the *setEntityColumns()* extension method then you need to add a `setHeader(core::getLabel(p))` call to the generated columns. If you have not, you will need to add it to the code that creates the columns in the Single-Select component and to the Init code of the Grid.

```
  @ExtensionMethod ~
public void setEntityColumns(Grid g, Type<Record> currentEntity) {
~
  g.setDataSource(new forms::TypeDataSource(currentEntity));
~
  def List<Property> properties := currentEntity.getProperties();
~
  properties.compact().collect(
    { p ->
       if !(p.getPropertyType().isSubtypeOf(type(Record)) or
            p.getPropertyType().isSubtypeOf(type(Collection<Object>)))
       then
         g.addColumn(new PropertyPathValueProvider(p), null,
            //Boolean sortable:
            false,
            // Boolean editable:
            true,
            // Editor editor:
            null
            )
            //adding header to each column:
            .setHeader(core::getLabel(p));
          end
      }
    );
   g.addButtonColumn("Delete", { e:Record -> deleteRecords({e}); g.refresh()});
}
```

3. Run preview of the form.

2.3.3 Adjusting Presentation

In the preview, you can spot that the Single-Select List and the Grid have empty space below: their size does not get adapted to their content.

To fix this, set the number of rows to the number of displayed items:

- on the Single Select List, set the number of rows to the number of displayed options:

```
c.setRowCount(options.size());
```

- on the Grid, set the number of rows to the number of data-source entries: You will need to get the number of entries for the selected Record and set this as the height of the Grid on initialization and whenever the user changes the displayed entity, that is in the `setEntityColumns()` method:

```
    ...
      );
      g.addButtonColumn("Delete", { e:Record ->
        deleteRecords({e});
        g.setHeightByRows(countAll(currentEntity));
        g.refresh()}
      );
      g.setHeightByRows(countAll(currentEntity));
  }
  ...
```

If you need to adjust the presentation further, such as, adding margin, consider using CSS or JavaScript.

2.3.4 Creating the Document

The final step is to create a page with the form: a page is represented by a document definition. When you upload a document definition, it is included in the list of documents, which are accessible from the Application User Interface. For more information on documents, refer to Documents-related documentation.

> **Note:** You can create fully customized page in Java directly as well. Refer to instructions on how to create a custom view.

To define a new document, do the following:

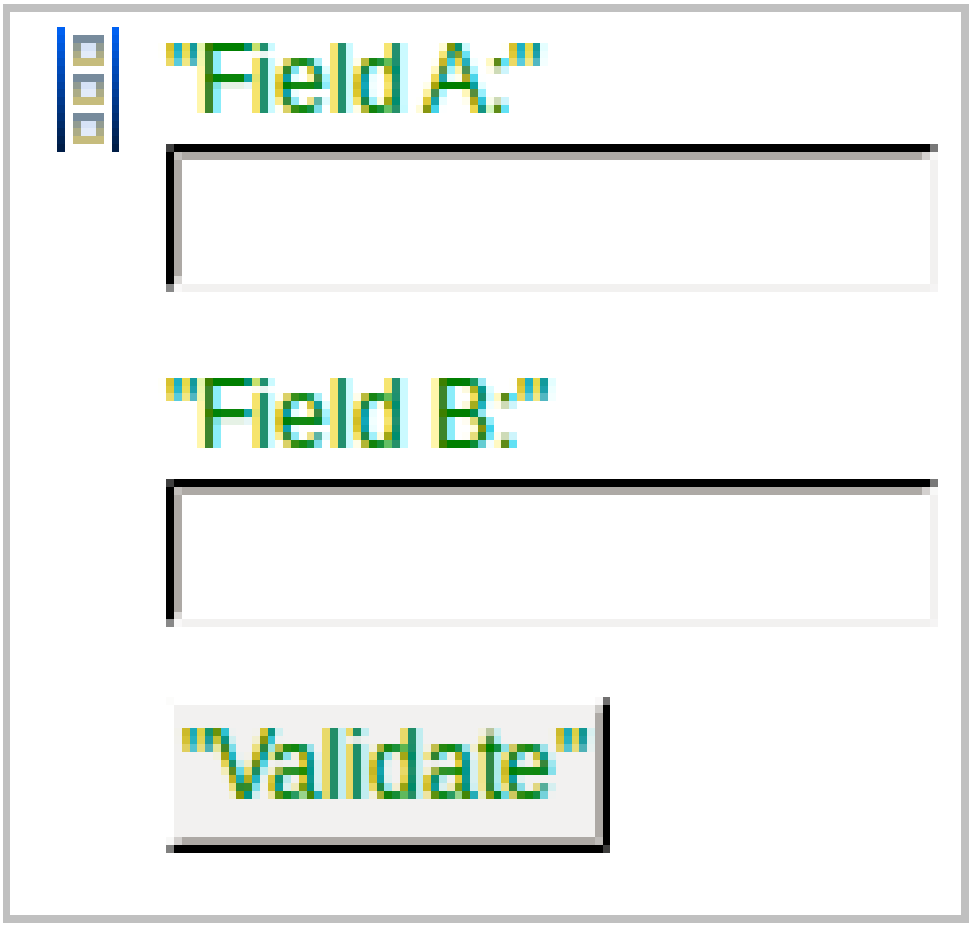
1. Create a document definition file:
   (a) Right-click your module.
   (b) In the context menu, go to New > Document Definition
   (c) In the New Document Definition dialog, define the definition file properties: check its location and modify its name.
2. Open the document definition file.
3. In the Documents area of the Document Editor, click **Add**.
4. In the right part, define the properties of the document:
   - **Name**: `entityOverviewDoc`
   - **Title**: `Entity Overview`
   - **UI definition**: `new EntityOverview()`
5. Upload your Module and check the Document on the Documents tab of the Application User Interface.

2.4 Validation of Multiple Components

Required result:

A forms::form component becomes invalid as part of front-end validation when some components hold a certain combination of values: in the example, a Text Field will be valid only if another Text Field contains a correct value and if the combination of the values of the fields is valid.

1. Create a form with two Text Fields.



2. Set field IDs, for example, to `a` and `b`.

3. Define a method on the form that adds error messages to components when they contain invalid values:

```
//rules for validation of the fields:
private void validateGroup(){
  //error message for field a:
  def String error1 := (a.getValue() == "1" and b.getValue()== "1") ? "Values must not equal
  //error message for field b:
  def String error2 := (b.getValue()== "3") ? "b value must not equal 3." : null;
  def String all := joinErrors(error1, error2);
  //setting the errors as custom error messages on a:
  if !all.isBlank() then
     a.setCustomErrorMessage(all);
  else
     a.setCustomErrorMessage(null)
  end
}
//concatenate errors from components:
private String joinErrors(String... errors) {
   def String concatenated := join(errors, "<br>");
   concatenated.isEmpty() ? null : concatenated;
}
```

4. Call the method whenever a value is changed on either of the fields.

```
 //Init on text fields:
c.setOnChangeListener({  e ->
   validateGroup()
})
```

2.5 Editing Data in a Popup with Conflict Check

Required result: The user accesses a Grid with entries of a shared Record type via a document. When they click the **Edit** column in a row, a Popup with editable data of the row is displayed. They can either save the changes or drop the changes. If someone else has edited the applicant in the meantime, log a notification.

The Popup form is reusable: the same form is used on two occasions: when creating a new applicant and when editing an existing applicant:

We will use the *Applicant* shared record displayed below.

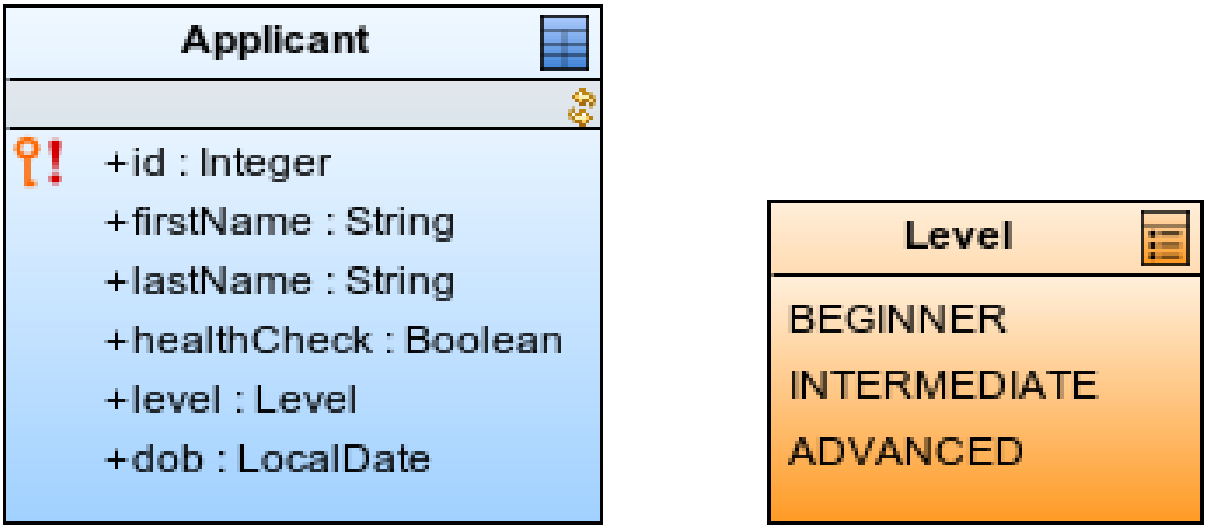
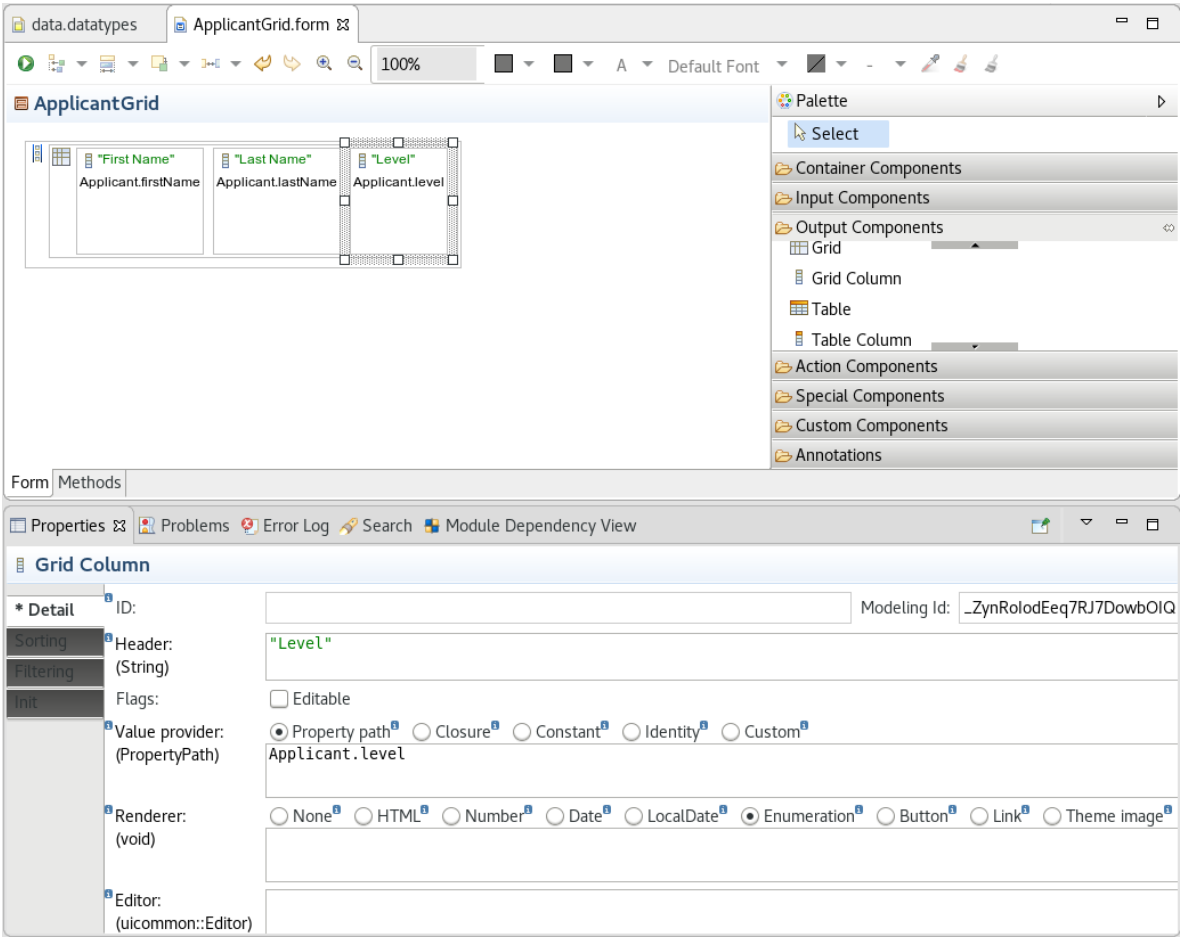


Figure 2.6 Applicant record with the Level enumeration used in the Applicant Record field

2.5.1 Displaying the Applicant List

Create a form with the list of applicants:

1. Create the form `ApplicantGrid`.
2. Design the form:
   (a) Insert a Vertical Layout.
   (b) Insert a Grid.
3. Set the Grid properties:
   (a) Set the ID to `applicantListGrid`.
   (b) Set the data source to **Type** and its value to the record type `Applicant`.
4. Insert the Grid Column for applicant properties:
   - Set the *Value Provider* to *Property Paths*.
   - Set the values of value providers to the Applicant fields: `Applicant.firstName`, `Applicant.lastName`, and `Applicant.level`.
   - For the *Applicant.level*, set the renderer to *Enumeration*.



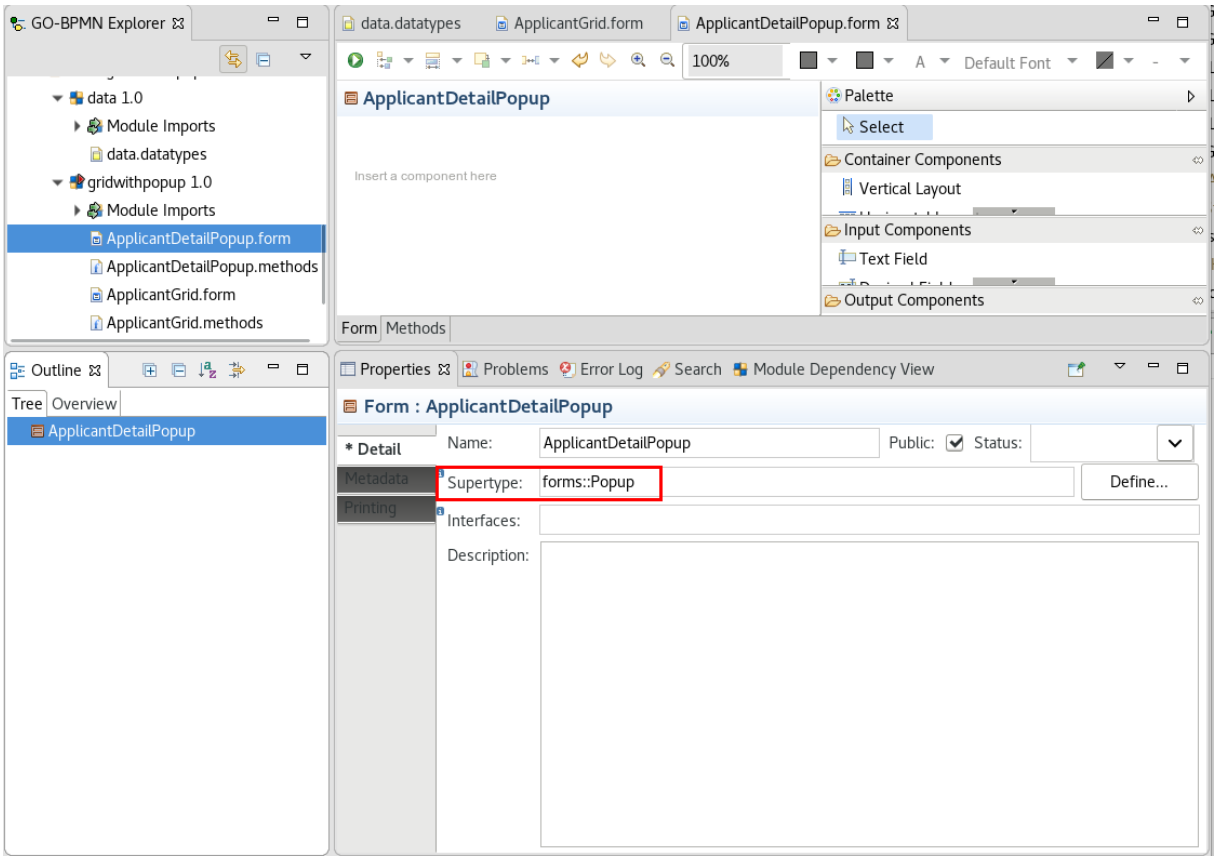
2.5.2 Displaying and Editing Applicant Details

To display the applicant data in a popup and to create a new applicant, you will create not only the form for the popup but also an object that will hold a copy of the applicant data. This will allow the user to close the popup without saving any changes they might have made. Were you to use the applicant object directly in the popup, the changes in the form would be applied immediately on the object.

Create the form for the details:

1. Create a form definition `ApplicantDetailPopup`.

2. In the Outline view, select the form root component and, in its Properties view, set its type to `forms::Popup` and make sure it is public.



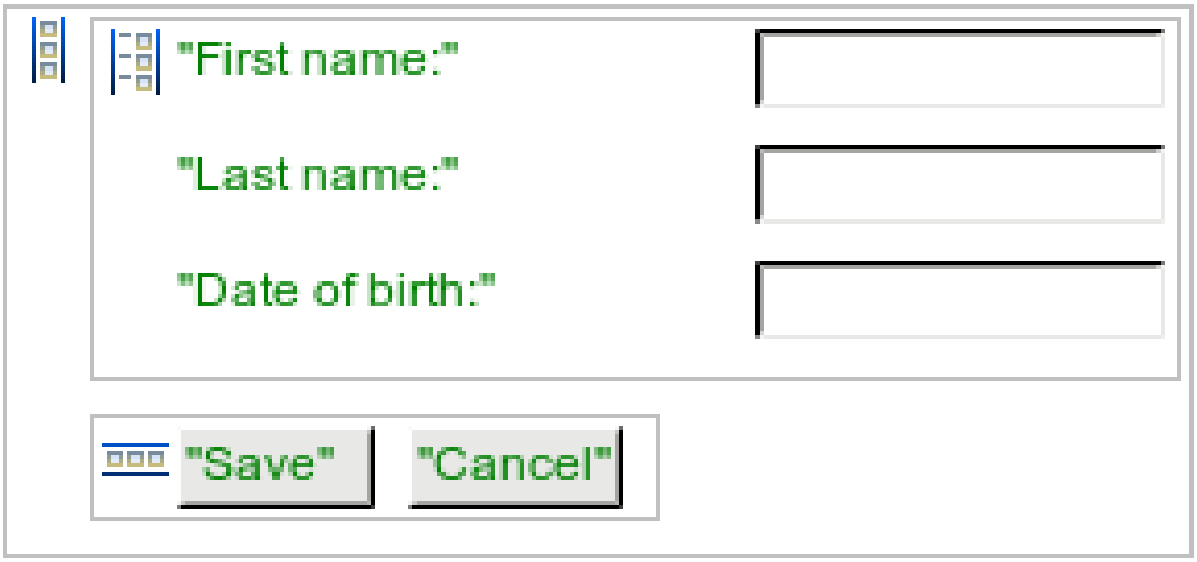
3. Create the form variable for the proxy mechanism, the *applicantProxySet* form variable of type *RecordProxy↩ Set*.

4. Create the *applicant* form variable of type *Applicant*, which will hold the data of the new or edited applicant: right-click the root node in the Outline view and select **New** > **Variable**; set its name to `applicant` and type to `Applicant`.

5. Define a parametric form constructors in the methods file of the form: Pass it the applicant parameter and store it in its proxy form variable so the popup can work with a proxy copy of a particular applicant:

```
public ApplicantDetailPopup(Applicant applicant){
  applicantProxySet := createProxySet(null);
  //change proxy of the applicant object is assigned
  //so that changes on the applicant are stored only after the user clicks Save:
  this.applicant := applicantProxySet.proxy(applicant)
}
```

6. Design the form tree: keep in mind it represents the content of a popup; bind the input fields to the application variable as appropriate.



7. Define the click listener expression on the **Save** button:

```
{ click:ClickEvent ->
      //apply the changes if the record has not been changed in the meantime;
      try applicantProxySet.merge(true)
      catch "com.whitestein.lsps.common.OptimisticLockException" ->
          //log a message if the record has been changed:
          log("failed to merge changes", 100);
          notify(
           caption -> "Conflict on merge",
           description -> "Your changes could not be saved: the data was changed."
          );
      end;
      //close the popup:
      this.setVisible(false)
}
```

8. Define the click listener expression on the **Cancel** button:

```
{ click:ClickEvent ->
      this.setVisible(false)
}
```

Now you need to display the data of an applicant in the popup when the user clicks an Edit button:

1. Insert a Grid Column that will render the **Edit** button that will open the public Popup with the row data:
2. Set the column details:
   (a) Set *Value Provider* to *Constant* with the value `"Edit"`.
   (b) Set *Renderer* to *Button*.
   (c) Below define the button action so that it creates and displays the popup with the data of the edited applicant:

```
{ clickedApplicant:Applicant ->
    //create the popup with details:
    def ApplicantDetailPopup appDetailPopup := new ApplicantDetailPopup(clickedApplicant)
    ~
    //display the popup:
    appDetailPopup.setVisible(true);
    ~
    //set listener on the popup, so the grid with applicants is updated when the popup cl
    appDetailPopup.setPopupCloseListener({ e->applicantListGrid.refresh()});
}
```

2.5.3 Creating a New Applicant

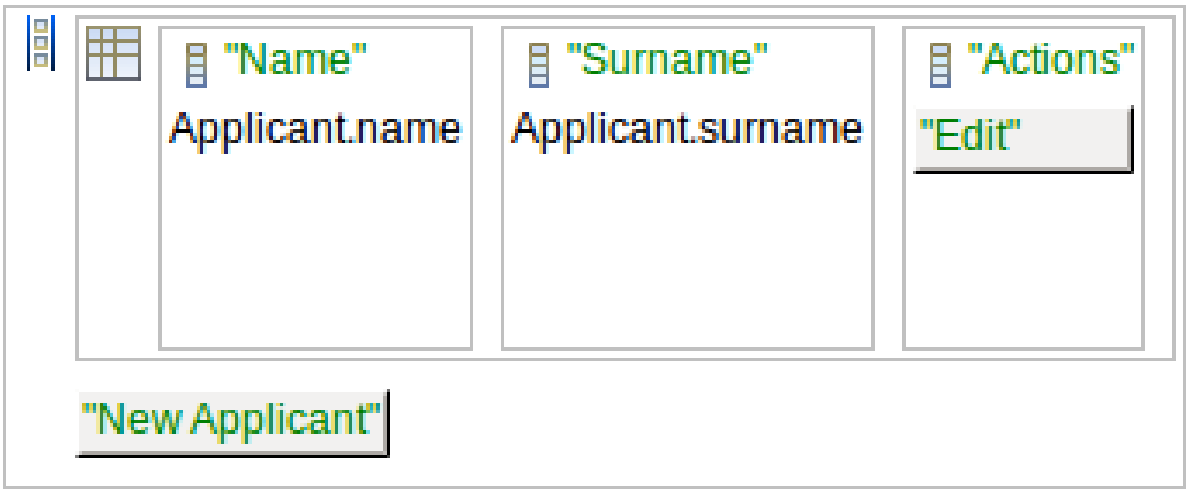
To allow the user to create a new applicant from the form, do the following:

1. Add to the methods of the *ApplicantDetailPopup* form, a non-parametric constructor that initializes the applicant variable to a proxy of the *Applicant* type:

```
public ApplicantDetailPopup(){
  //proxy set init:
  applicantProxySet := createProxySet(null);
  //change proxy diretly over the Applicant type:
  applicant := applicantProxySet.proxy(Applicant)
}
```

2. In the *ApplicantGrid* form, insert a *New Applicant* button.
3. Define its click action so it creates and displays the public popup using the non-parameteric constructor:

```
{ click:ClickEvent ->
    //creates the public popup with the non-parametric constructor:
    def ApplicantDetailPopup appDetailsPopup := new ApplicantDetailsPopup();
    appDetailPopup.setVisible(true);
    //refreshes the grid so it contains the new applicant:
    appDetailPopup.setPopupCloseListener({ e->applicantListGrid.refresh()});
}
```



Now you can use the **ApplicantGrid** form in documents or user tasks as their UIDefinition.

You can download the tutorial example here.

2.6 Filter over Grid and Table with a Custom Data Source

Note: This tutorial uses the `forms` module as its form implementation.

Required outcome: A form with a grid and table that use a custom data source and support filtering.

Note: You can download the tutorial model here: once you have imported the archive, you can test the model by running the *filterGrid* module.

2.6.1 Implementing a Custom Data Source

1. Make sure the record of the custom data source implements the `forms::DataSource` interface.
2. Add fields to the record for the filters as set on the Columns.

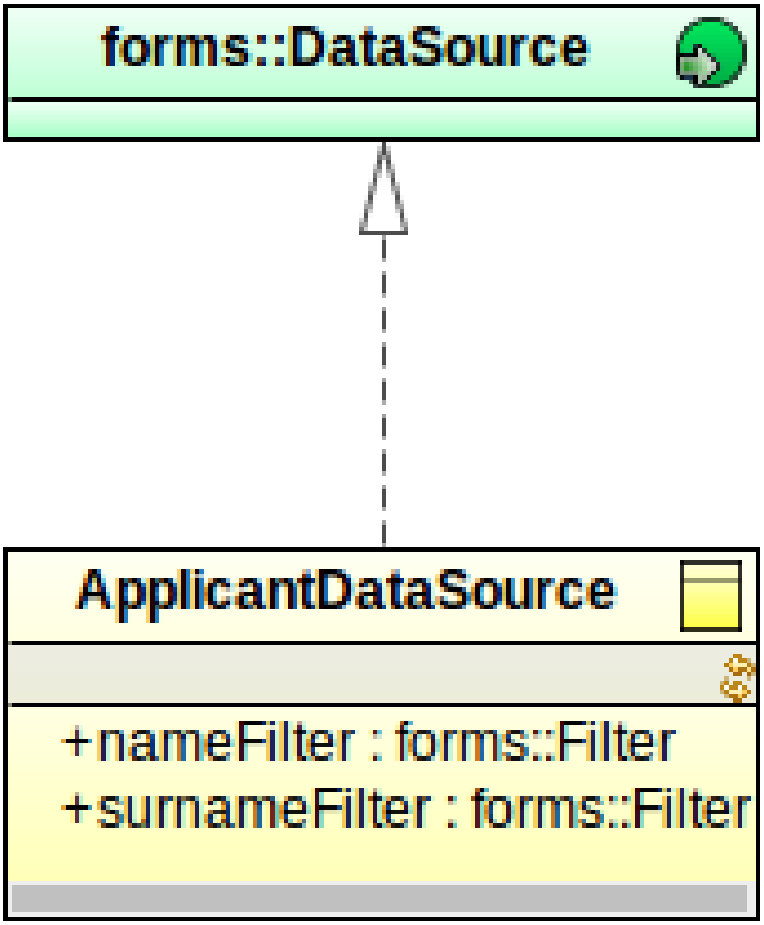


Figure 2.7 Custom data source record

3. Adapt the data source methods so that the getCount() and getData() methods handle the filtering.
   The filters are passed as input parameters to the methods.

Example data source methods

```
ApplicantDataSource {
  ~
  public Integer getCount(Collection<forms::Filter> filters){
  ~
       def String firstnameFilterSubstring := getFilterValue("firstName", filters);
       def String surnameFilterSubstring := getFilterValue("surname", filters);
  ~
     //count query that filters the results:
     getApplicants_count(firstnameFilterSubstring, surnameFilterSubstring);
    }
  ~
  public List<Object> getData(Integer startIndex*, Integer count*, Collection<forms::Filter> filte
  ~
       def String firstnameFilterSubstring := getFilterValue("firstName", filters);
```

```
        def String surnameFilterSubstring := getFilterValue("surname", filters);
  ~
        //query that gets results and applies filters:
        getApplicants(firstnameFilterSubstring, surnameFilterSubstring);
    }
  ~
    private String getFilterValue (String filterParameterName, Collection<forms::Filter> filters){
        //get first filter with matching name:
        def forms::Filter firstMatchingFilter := getFirst(filters, { f -> f.id == filterParameterNa
        // get search substring in filters:
        def String filterSubstring := firstMatchingFilter == null ? null : (firstMatchingFilter as
        filterSubstring
    }
  ~
  public Boolean supportsFilter(forms::Filter filter*){
    if
      filter.id == "firstName" || filter.id == "surname" then true;
    else
      false
    end
  }
  ~
  public Boolean supportsSort(Sort sort*){
    false
    }
  public String toString() {
    #"ApplicantDataSource"
  }
}
```

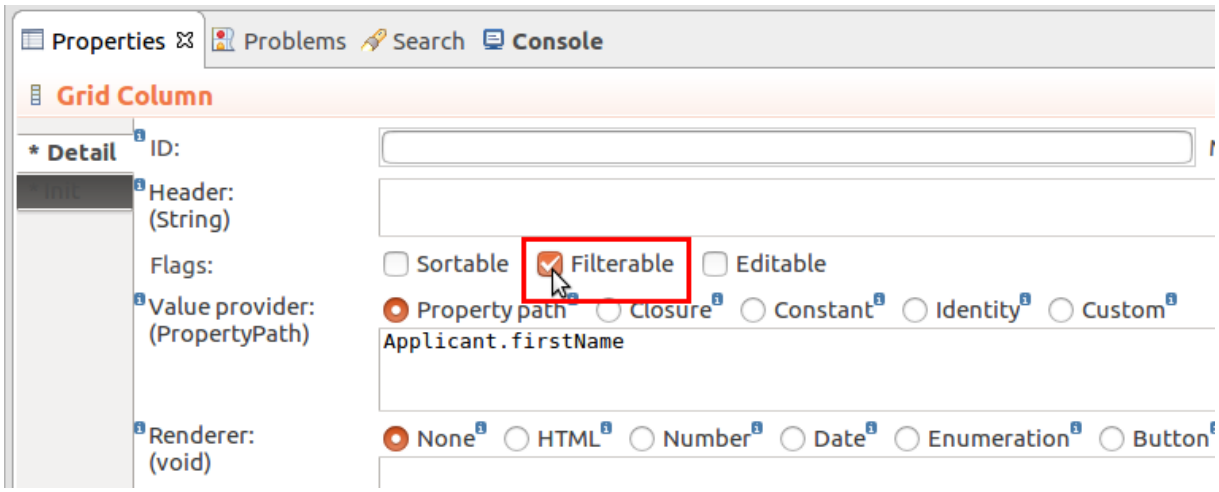
2.6.2 Creating the Form

1. On the Grid, set the data source to **Custom** with the value of the data source instance.

   ```
   new ApplicantDataSource()
   ```

2. On the columns of the respective component, do the following:

   (a) Enable filtering.



   (b) Configure the filter ID on the column:

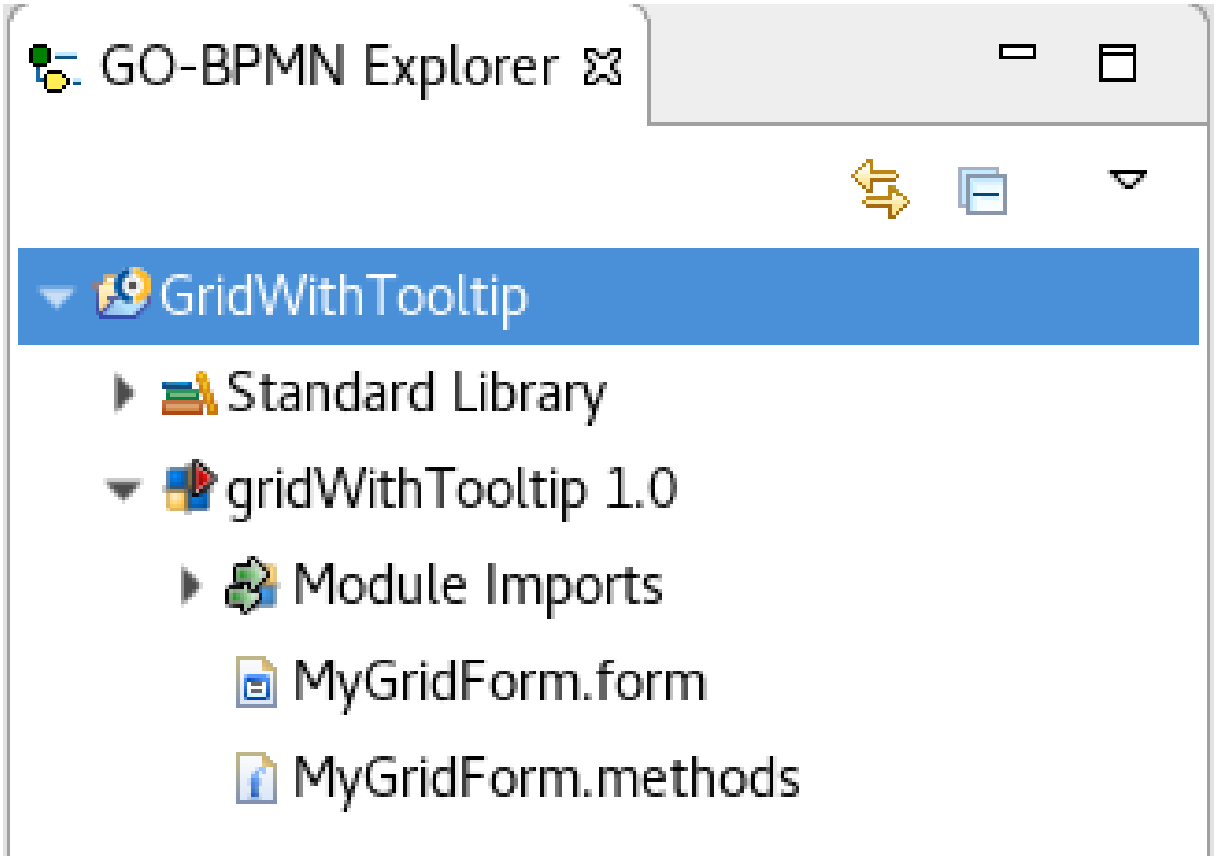
   - For a Grid column, set on the *Init* tab as `c.setFilterConfig(new Filter↩ Config(filterId -> "FILTERNAME"))`.
   - For a Table column, set on the *Filtering* tab as `new FilterConfig(filterId -> "FI↩ LTERNAME")`.

2.7 Icon in a forms::Grid

Required result: A Grid Column displays an icon in the given row.

	1
	2
	3
	4
	5

1. Create a GO-BPMN project with a module and a forms definition.



2. In the forms definition, insert a Grid and define its data source, for example, collection `{1..10}`.
3. Insert a Grid Column into the Grid, which will hold an icon.

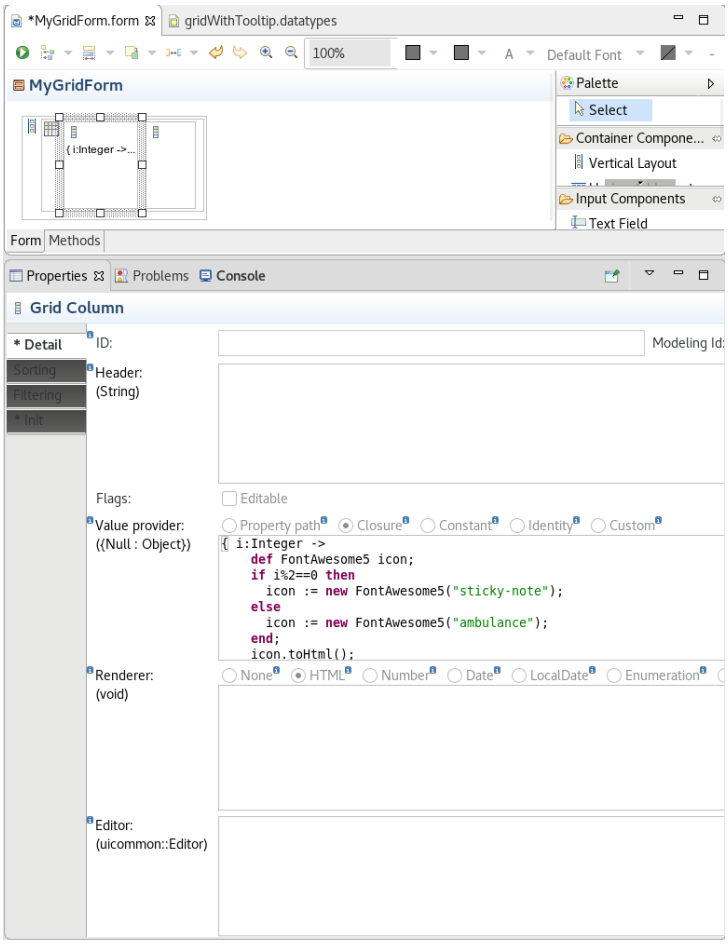
4. On the column, set type of the value provider to `Closure` and define its value as a closure that returns the icon, for example:

```
{ i:Integer ->
 new FontAwesome5("ambulance").toHtml();
}
```

5. Set the renderer to *HTML*.

You can modify the closure so the icon changes depending on circumstances:

```
 //i is the row object: we set the data source to a collection
//of Integers; therefore, it is an Integer:
{ i:Integer ->
  def FontAwesome5 icon;
  if i%2==0 then
    icon := new FontAwesome5("sticky-note");
  else
    icon := new FontAwesome5("ambulance");
  end;
  icon.toHtml();
}
```



Chapter 3

UI Forms Tutorials

3.1 Editable Table

Required result:

- ui::Table with columns with editable values.
- One of the columns contains a drop-down list with the possible options. The options are based on an enumeration.
- The table values are persisted when you click the **Submit** button.

Figure 3.1 Resulting form

To create a document or a to-do with such a table, you need to do the following:

1. Create the data type model with a shared record for the persisted entity and the enumeration.

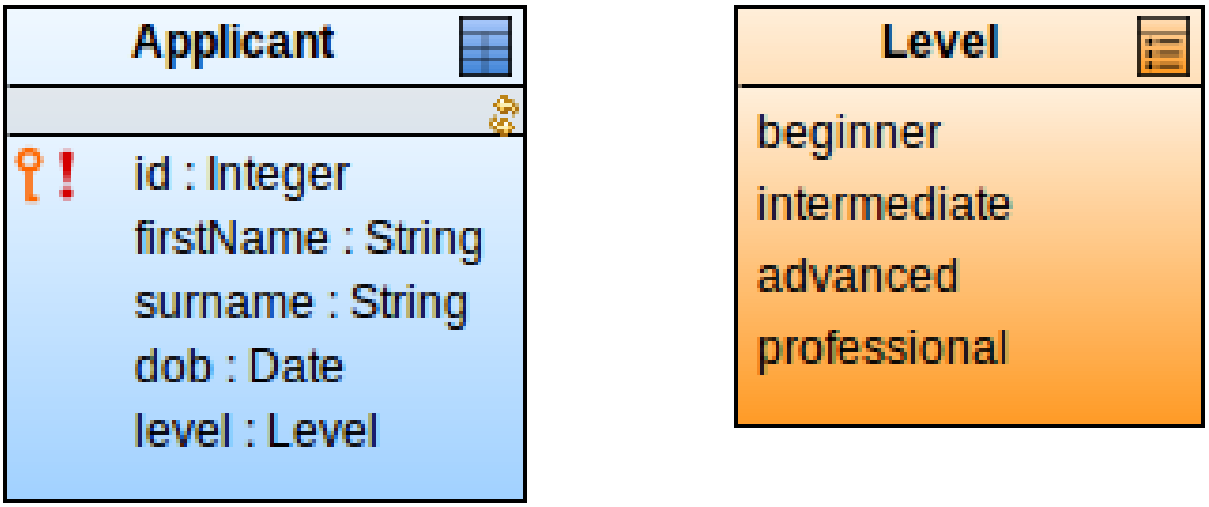


Figure 3.2 The underlying data type hierarchy

2. Create the form definition.
   (a) Create a form variable *applicant* of the shared record type *Applicant*: The table will use the variable as its iterator.
   (b) In the form, add the Table component and define its properties on the Detail tab:
      i. Set **Data Iterator** as the reference to the form variable.
      ii. Set Data Kind.
      iii. Define the Data expression.

      In this pattern, we assume you are using the Data Kind set to Data with the Data expression defined as a closure with two input parameters:`{x, y -> getAllApplicants()}`
   (c) In the table component, insert the Table Column components.
   (d) In the columns, add the Input components.

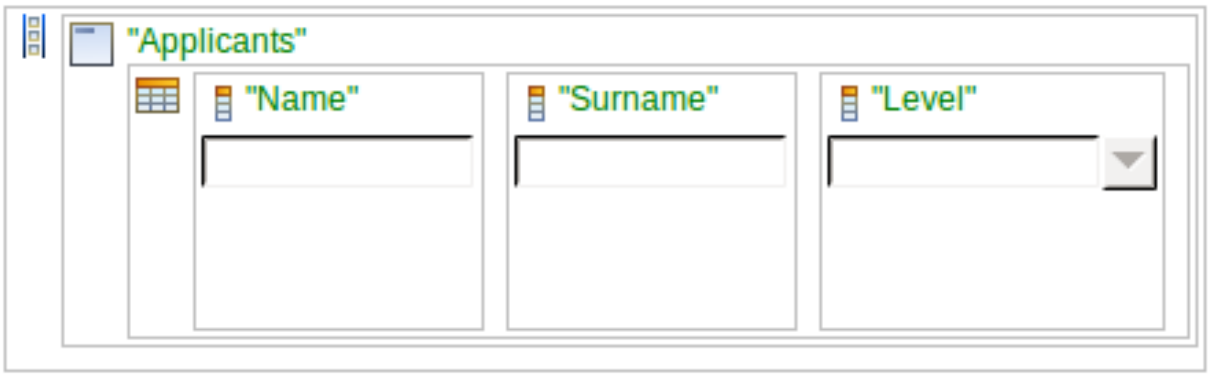


Figure 3.3 Asset table with columns with two text boxes and one combo box

In the example, we inserted two Text Boxes and one Combo Box:

i. On Text Boxes, define the binding to the reference to the iterator fields, for example, `&applicant.surname`.
ii. On the Combo Box component, define the binding to the iterator field and the options to be displayed in the drop-down area.

To bind options to the enumeration, convert the enumeration literals to options. You can do so using the `collect()` and `literalToName()` functions.

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

(e) Define the Submit button:

i. Insert the Button component into the form.

ii. Create ActionListener on it.

iii. On the listener properties, select the Submit action on the Actions tab.

(f) Optionally, set the text that should be displayed in the table if it contains no entries: on the Presentation Hint tab of the table properties, add the `no-data-message` hint.

3. Create a document or a process with a to-do that uses the form.

3.2 Table with Derived Values

Required result: A table with a column with a value derived from another column value: One column value is persisted; the derived value is transient. The column values depend on each other and adapt to each other when either is changed.

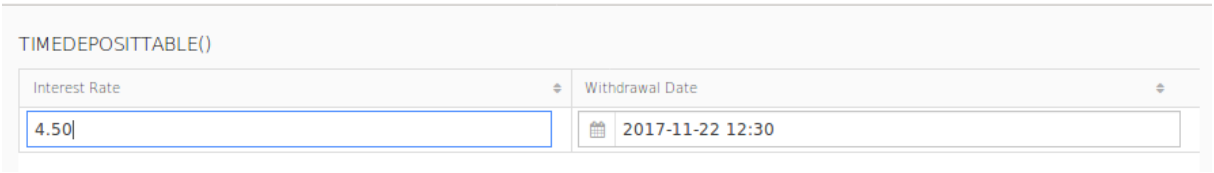


Figure 3.4 When you change Interest Rate, the Withdrawal Date changes. Withdrawal Date is not persisted.

1. Create the underlying data type hierarchy with the *base shared record* and a *non-shared record with fields for the derived values*:

(a) Create or import the base shared record.

(b) Create a record with the derived field.

(c) Define an association between the records: the derived record is the target of the relationship.

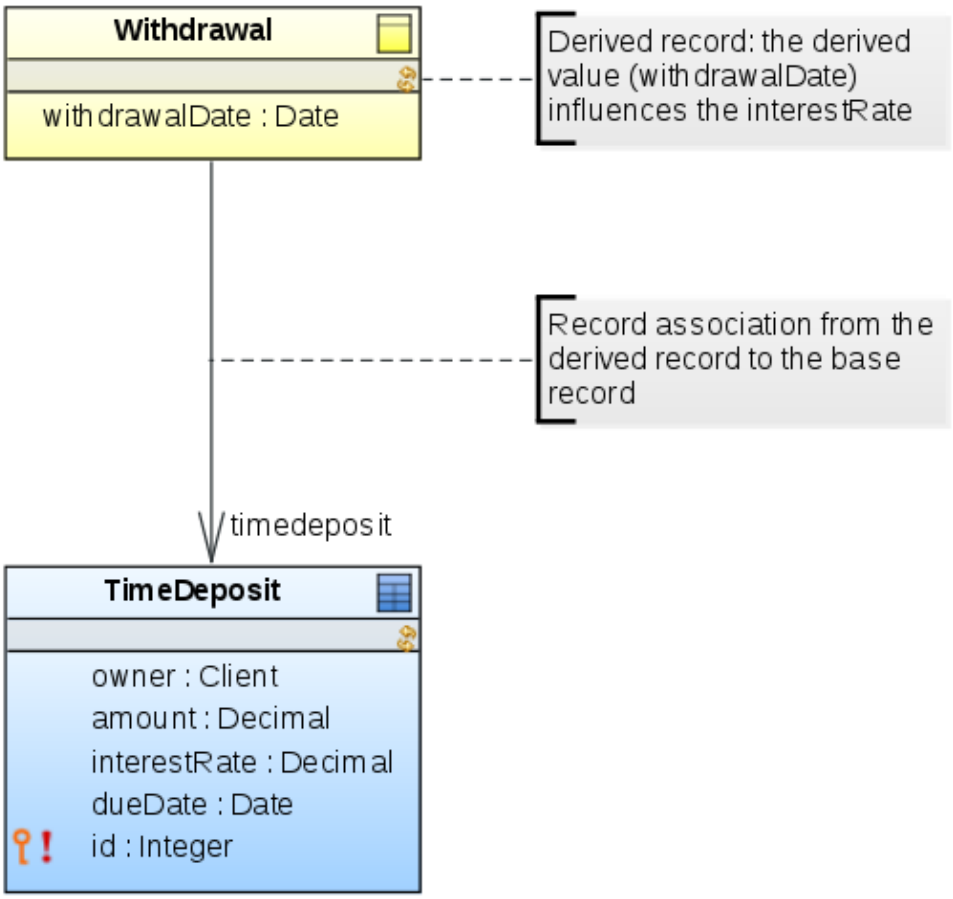


Figure 3.5 Base shared record TimeDeposit associated with the derived non-shared record Withdrawal

Important: In such scenarios, you **cannot use the supertyping mechanism** since a shared record is involved:

- If you used a derived non-shared record that is the supertype of the base shared record, the derived record would include the fields of the base shared record but the shared record itself could not be recovered efficiently.
- If you used a derived non-shared record that is the supertype of the base shared record, If you decided to define the base shared record as the supertype of the wrapper non-shared record, whenever you decide to refresh the table with the record data, new shared record instances would be created and written in the database.

2. Create the form definition.

   (a) Create a local variable of the derived record type.

   The variable will serve as the iterator variable for the table.

   (b) Create a local variable of the collection type with the derived records (for example, List<Withdrawal>), and initialize it so it holds the available Withdrawal object, for example, with the `collect()` function.

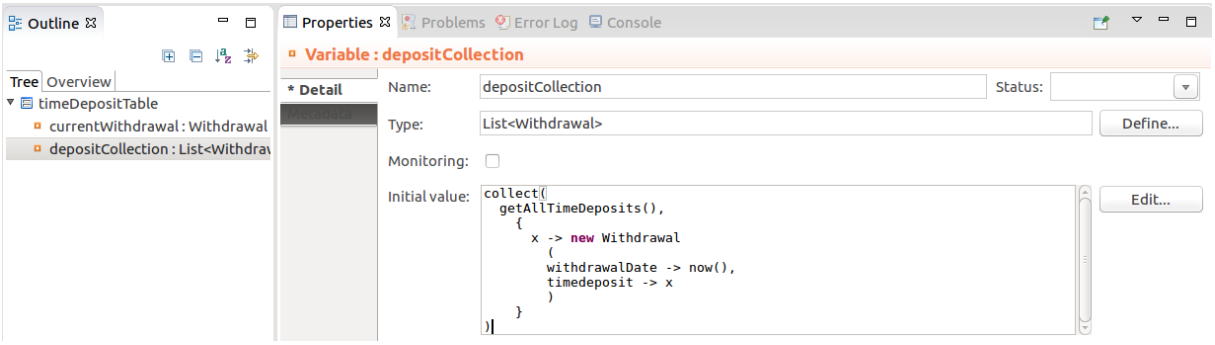
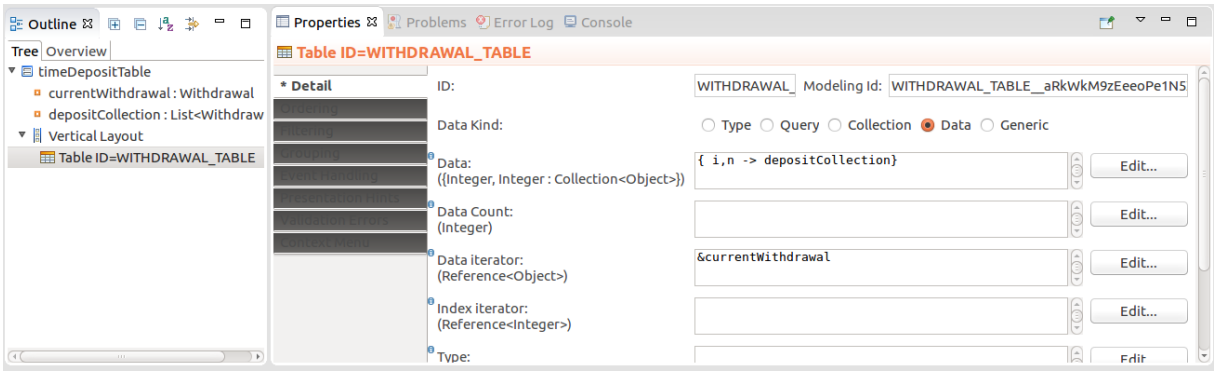


Figure 3.6 The collection form variable with the initial value

(c) In the form, insert the Table component and define its properties:

- *Data Kind* as **Data**
- *Data* as a closure that returns the local variable with data.
- *Data Iterator* as reference to the iterator variable



(d) In the table component, insert Table Column components and input components as their child components: define their ID and the binding of the input components to the respective field of the iteration variable (in the example, `&currentWithdrawal.timedeposit.interestRate` and `&currentWithdrawal.withdrawalDate`).

(e) On each input component define the following:

i. Create ValueChangeListeners: as the component to refresh, define the other input component and as Handle expression, define the new value of the iterator field, for example, using a function. Do not define the column as the component to be refreshed. Columns do not support the refresh action.

```
currentWithdrawal.withdrawalDate:= countWithdrawalDate(currentWithdrawal.timedeposit.interest)
```

ii. Set the Immediate property to `true` otherwise change of a value will not trigger change of the other value: the change would take place only after another event triggers the processing.

When set to `true`, the value changes are processed whenever the user clicks out of the input component or presses Enter.

3. Create a document or a process with a to-do that uses the form.

3.3 Calendar with Adding Entries Functionality

Required result: Form with a calendar into which you can add entries by selecting days in the calendar: entries details are defined in a pop-up dialog.

Do the following:

1. Create or import the shared record for your calendar entries.

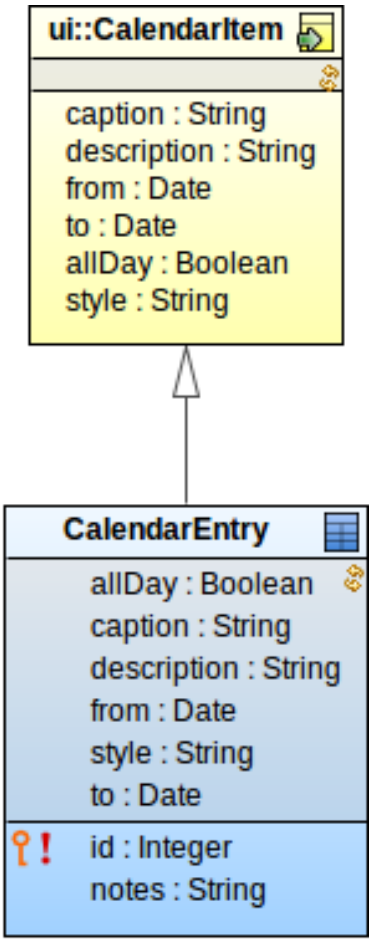


Figure 3.7 Shared record for calendar entries derived from the CalendarItem record

2. Create a form definition, open it and insert a Vertical Layout component.

3. Create a local variable of the calendar entry type.

   The variable will hold the data about a new calendar entry. For the example above, the variable will be of the `CalendarEntry` type.

4. Create the calendar:

   (a) Insert the Calendar component into the vertical layout.

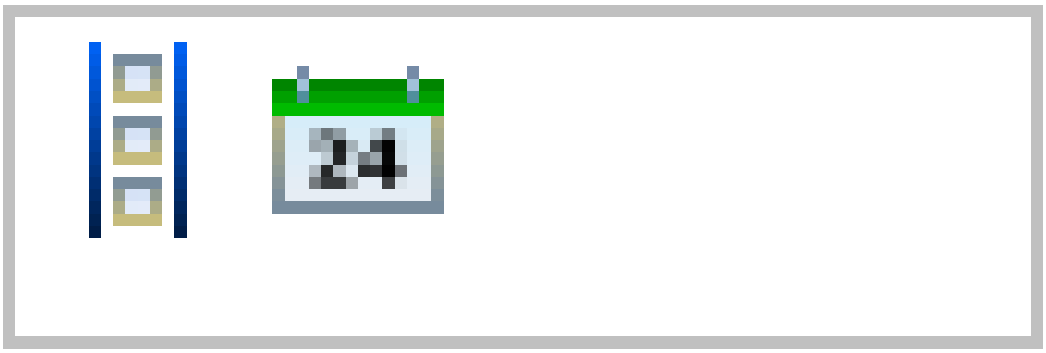


Figure 3.8 Vertical layout with calendar component

(b) Define the properties of the calendar:

- **Data**: closure that returns all calendar entries (The closure is called on calendar initialization and refresh: After you add a new calendar entry to the database, the calendar needs to be refreshed so as to load and render the new calendar entry.)

```
{ a, b -> (toSet(findAll(type(CustomCalendarItem)))) }
```

- **To item**: transformation of the data object to CalendarItem so the Calendar component knows how to display them; in this case, transformation of the *CalendarEntry* to *ui::CalendarItem*.

```
{ calItem:CalendarEntry -> new CalendarItem(caption -> calItem.caption, description -> calItem.description,
       from -> calItem.from, to -> calItem.to, allDay -> calItem.allDay, style -> calItem.style)}
```

5. Create the popup:

(a) In the form, insert the pop-up component and define its properties:

- **ID**: although component ID is not required, you will need it when displaying the pop-up (on button click, the visibility variable will be set to true the pop-up component will be refreshed so as to have it rendered).
- **Visible**: enter a name of a Boolean variable that holds the visibility of the popup.
  You can define a Boolean form variable; make sure to set its initial value to `false`.

(b) Nest the pop-up component in a *View Model*: right-click the popup and selects **Insert Parent** > **View Model**. Define its ID.

> **Note:** The view model component isolates the data in the pop-up component from the data in the form context: it creates an evaluation context over the screen context. You will initialize the calendar entry variable when the pop-up is displayed and get the dates the user selects in the pop-up, all this will take place in the new evaluation context.
> If you don't nest the pop-up in a view model component, the initialization of the variable will create a shared record with incomplete data in the screen context. When nested in the view model, the data is written into the screen context only after it is submitted or persisted by a listener.

6. Create the content of the popup: insert the *Form Layout* component and into it input components so the user to provide the other details for the *CalendarEntry*. Make sure the input components are bound to the correct field of the `CalendarEntry` variable.

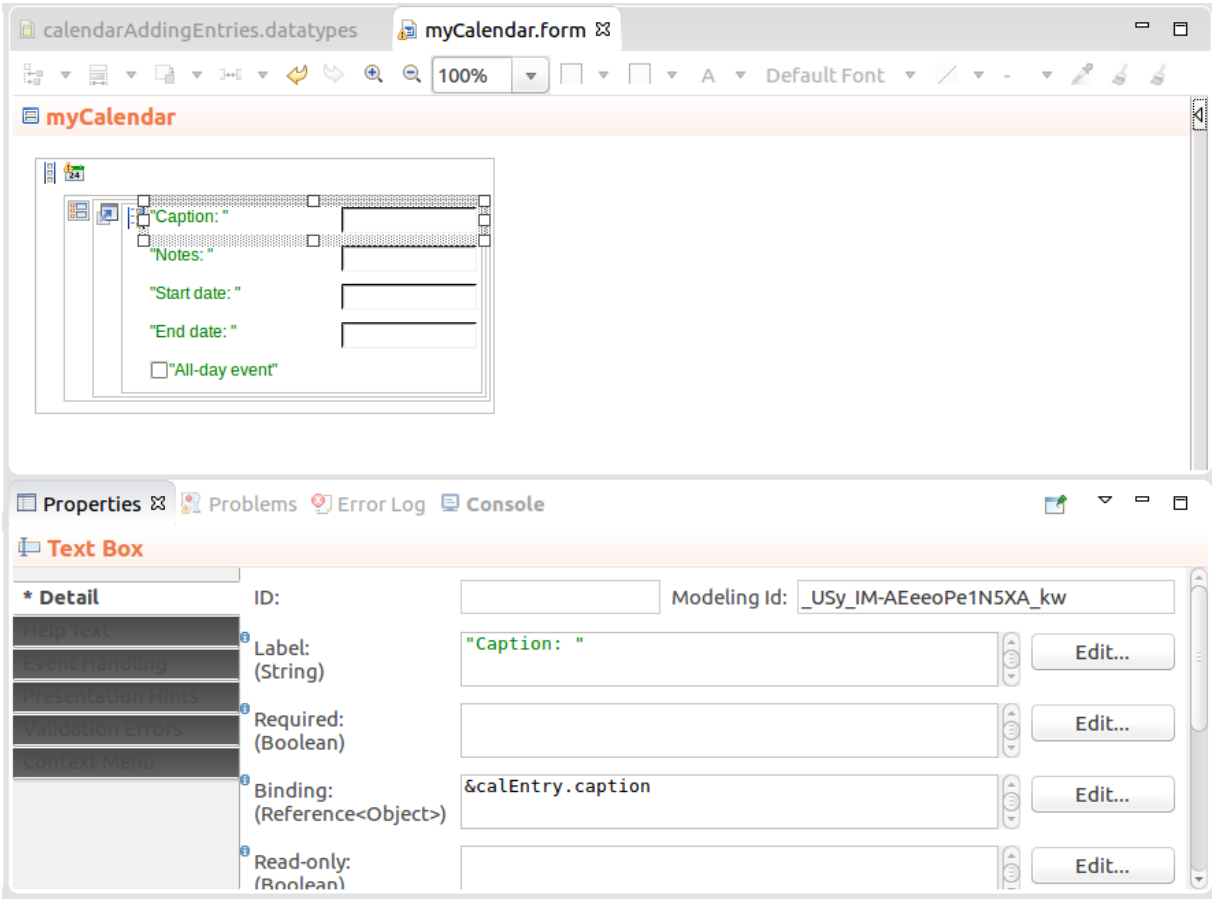


Figure 3.9 Calendar form

7. On the calendar component, create a *CalendarCreateListener* that will display the popup with the selected dates, when the user selects a time period by clicking and dragging:

   - Set its visibility to true and refresh it:
     - On the Basic tab, enter the **pop-up ID as the Refresh components** value.
     - On the Basic tab, define the **handle expression** so it sets the variable with the **pop-up visibility to true**.
   - Initialize calendar entry with the clicked dates: on the Basic tab, in the Handle expression, extract the dates from the event into the *CalendarEntry* variable:

```
calEntry:=new CalendarEntry(
        from -> _event.from,
        to -> _event.to
        )
```

8. Define the submit button in the pop-up that will persist the provided data and close the pop-up:

   - In the pop-up component of the form, insert the button component and define its properties.
   - Create the ActionListener on the button with the following:
     - Handle expression hides the pop-up.
     - Refresh the pop-up and the calendar.
     - Merge the view model (On the *Advanced* tab, enter the ID of the view model in the **Merge view model components** property)
     - Persist to save the new event in the database so it is picked up by the `findAll()` call on calendar refresh.

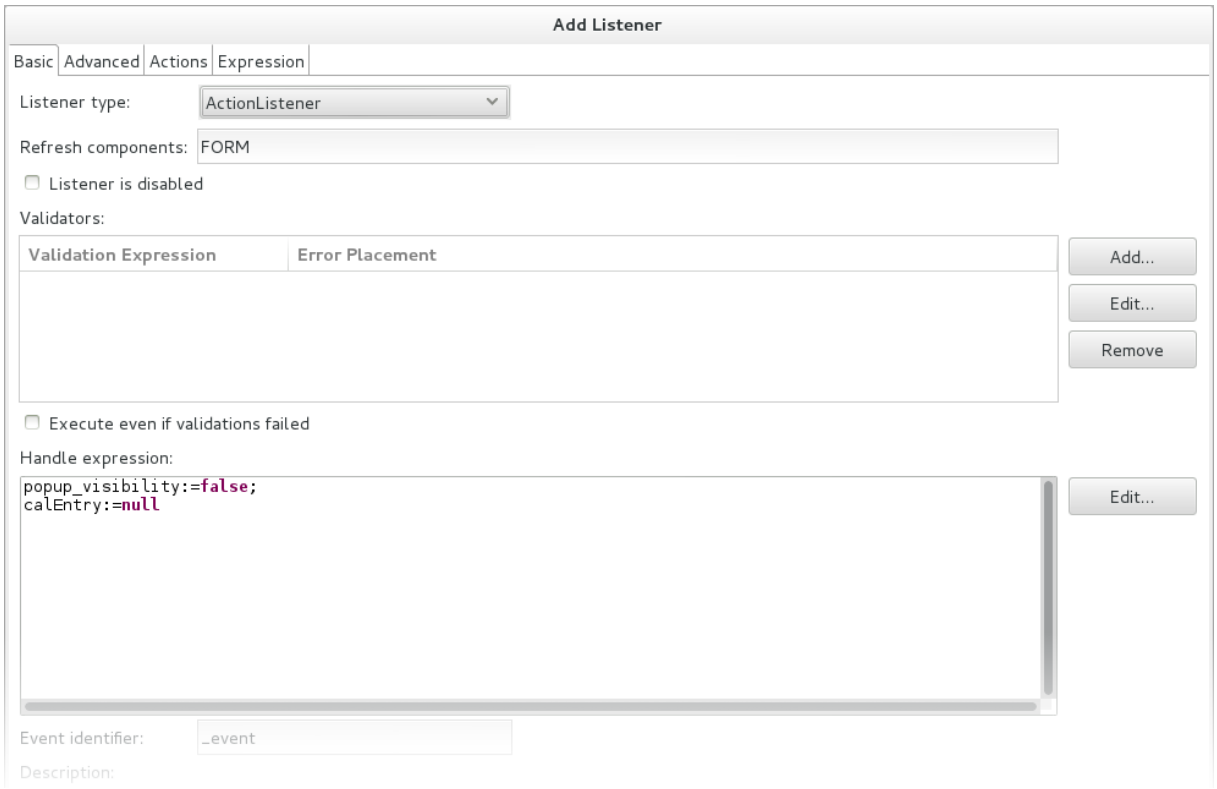
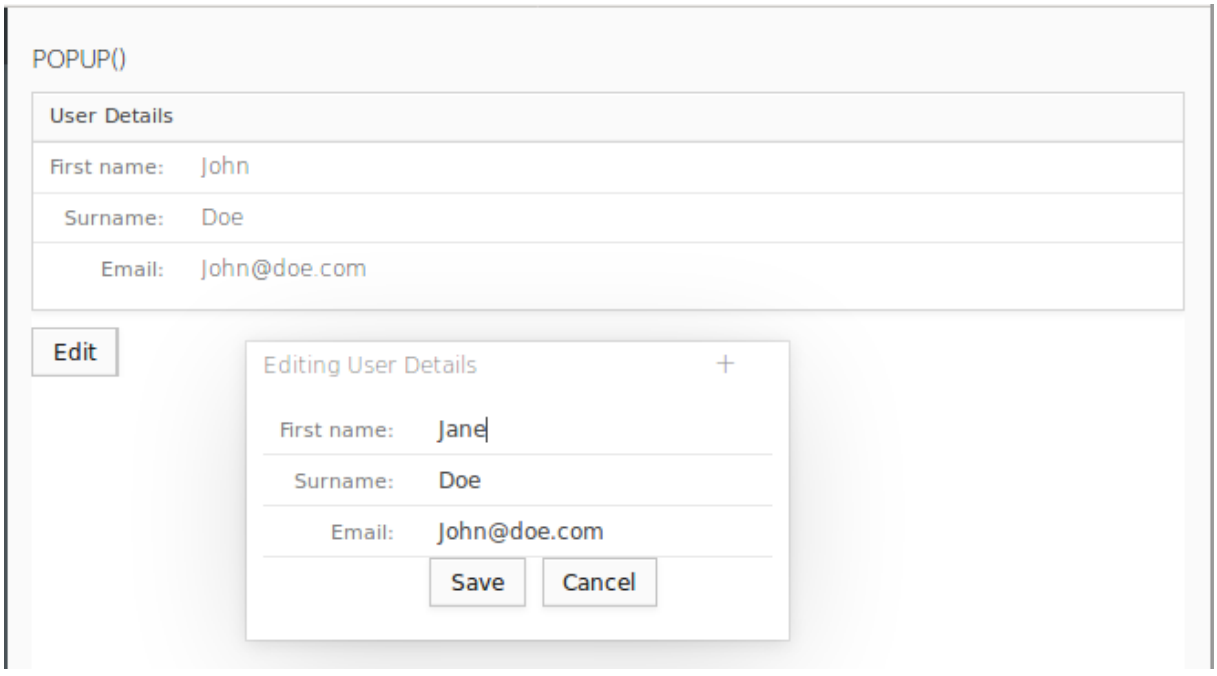


Figure 3.10 Listener on the submit button

3.4 Pop-up with Save and Cancel Buttons

Required result: When you click a button in your form, a popup where you can edit the form data is displayed. The pop-up contains an Save and a Cancel button. When you click the Save button, the pop-up closes and the data in the form contains the new data. When you click Cancel, the data in the pop-up is discarded and the pop-up closes.



To create a pop-up window with a Save and Cancel button, do the following:

1. Open the form with the data you want to edit in the popup.

   In the example, the data already exists and is stored in a form variable. If you want to create new data from the popup, make sure to initialize the data in the View Model we create in the next step.

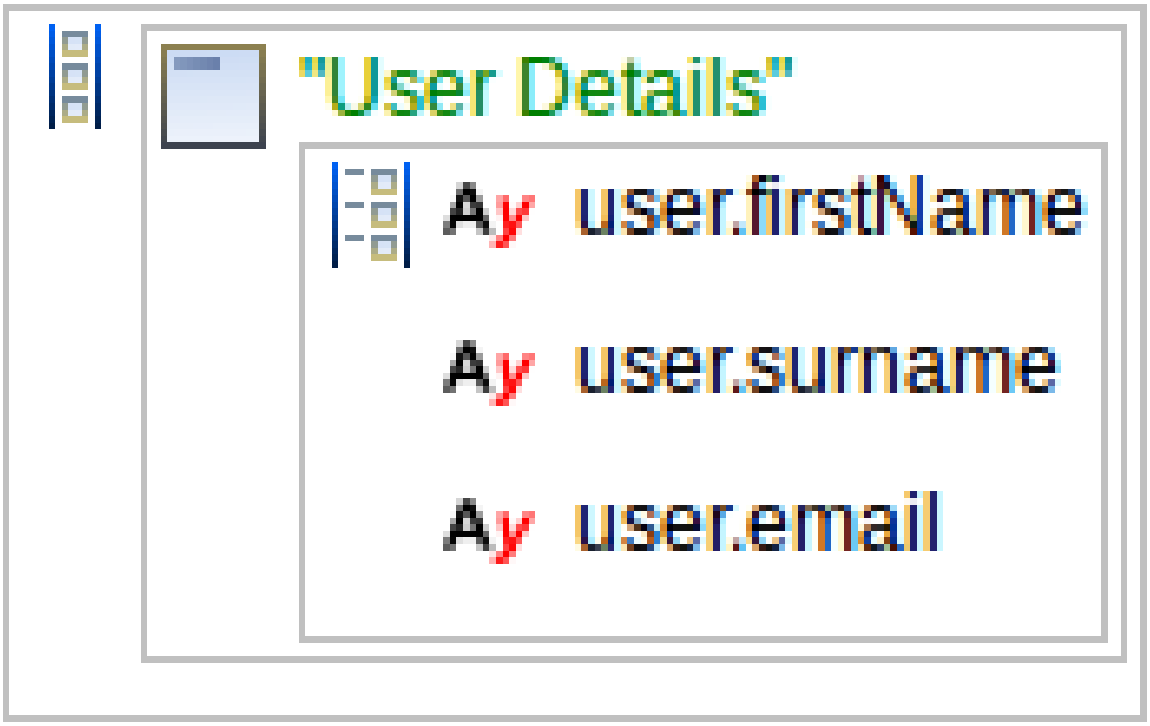


Figure 3.11 Form with user details

2. Insert the View Model component into the form and define its ID.

   The view model creates a new context for its child components. It holds the differences to the form context. This will allow us to discard or save the differences in a single step: we will either merge the view-model context or discard it (for more details on how it works, refer to `view model`).

3. Insert the Popup component into the View Model component.

   If you plan to create a complex component tree in the popup, consider using the `dynamic popup` to prevent performance issues: the dynamic popup is created only when the popup is requested, while the modeled popup is created when the form is initialized, which can be time consuming.

4. Define the popup behavior:

   (a) Create a Boolean form variable with the initial value to `false` and set the variable as value in the **Visible** property of the popup.

   (b) Create the logic that will open the popup, for example, insert a Button with an ActionListener that sets the visibility variable to true and refreshes the popup.

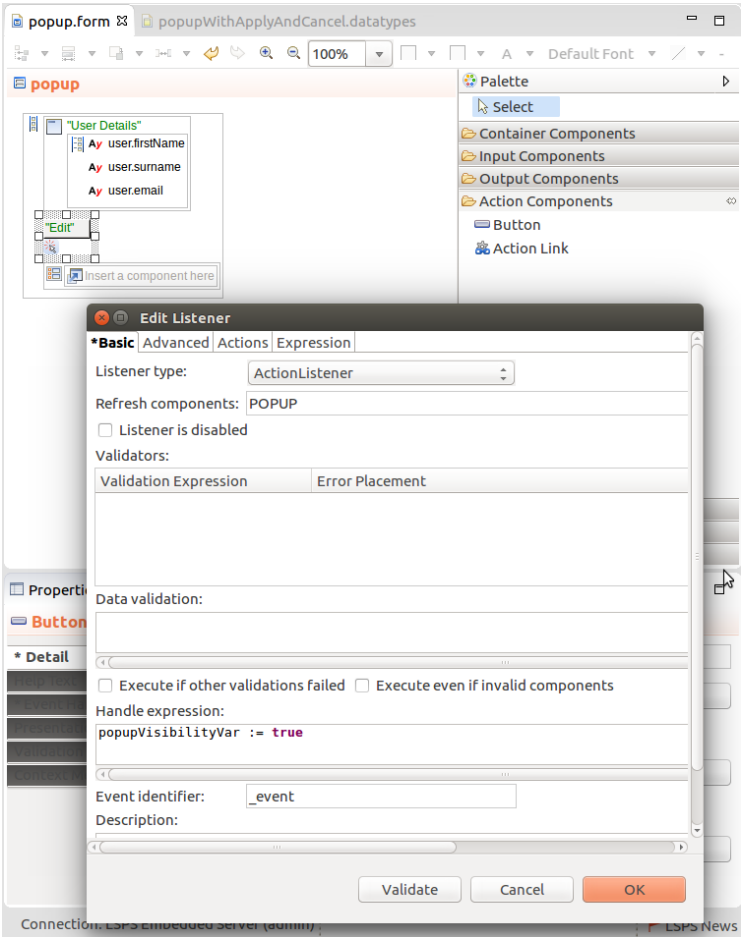


Figure 3.12 Setting Popup visibility for the ∗Save∗ button click

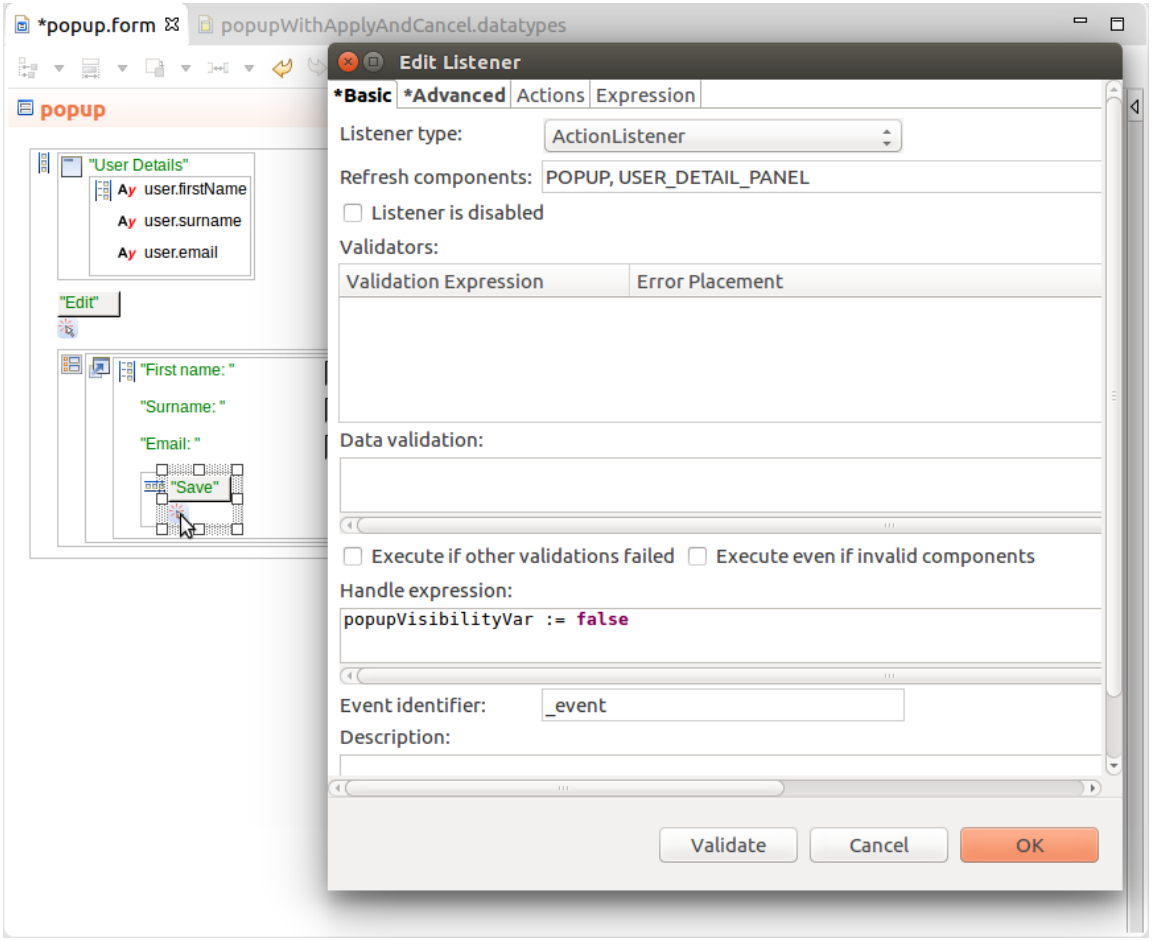
5. Create the popup content:

   (a) Insert a layout component and input components into the Popup component.

   (b) Bind input components to the local variable and define the labels.

   (c) Insert the Button component for the **Save** button and attach to it an ActionListener that will execute the following:

   - Merge the changed data to the form context: On the *Advanced* tab in the **Merge view model components** property, insert the ID of your view model.
   - Close the popup: on the Basic tab in the Handle expression, set the popup visibility to `false` and in the Refresh components, insert the ID of the popup component.
   - Refresh the data in the form (outside of the view model): in the Refresh components, insert the IDs of the components.



(d) Insert the Button component for the **Cancel** button and attach to it an ActionListener that will execute the following:

- Close the popup: on the Basic tab in the Handle expression, set the popup visibility to `false` and in the Refresh components, insert the ID of the popup component.
- Discard the changes in the View Model: On the Advanced tab in the *Clear view model components* property, enter the name of your view model.
- Set the listener to execute in the form context: On the Advanced tab, set the *Execution context* property to **Top level**.

If left set to default, the listener would execute in the execution context created by the view model. Since we are discarding the data from the view model, the visibility setting would be discarded as well and the popup would remain open.

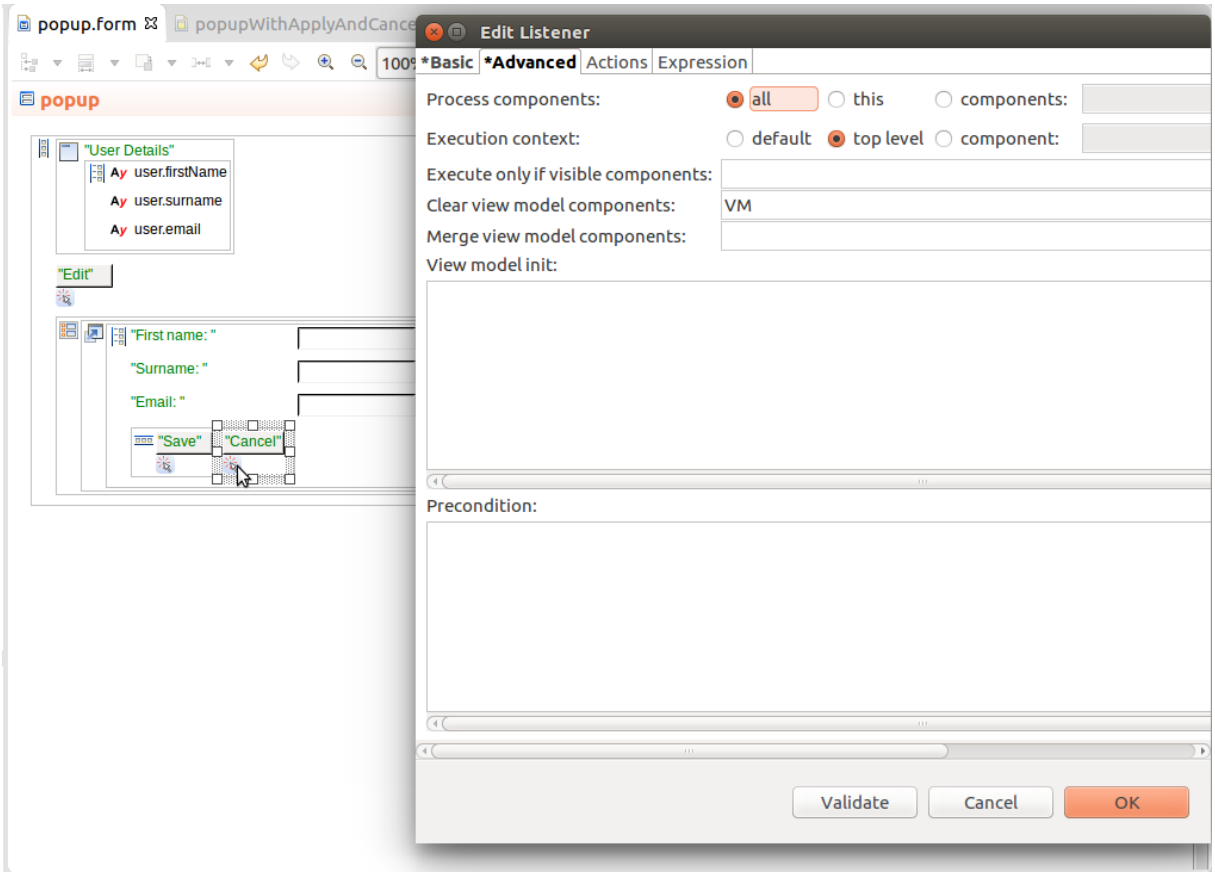


Figure 3.13 Setting cancel as View Model action for the Cancel button click

6. Run the Form Preview and check the functionality.

Chapter 4

Process Tutorials

- The pattern of restartable processes allows you to design model instances that can be restarted at any point and acquire the same execution status.
- The pattern of agile processes allows you to design processes in which you can skip an Activity or a flow, and switch between Activities or flows as required without breaking your data.
- To create a model instance over a record with your business data can be useful in the cases when you allow your users to create entities for documents: The user creates an shared record from a document, for example, an order, and on submit a new model instance takes care of further actions over the entity, such as processing the order.
- To monitor the start of a model instance can help you to make sure that restartable model instances are in the correct status upon restart: The user adapts a restartable model so that the starting is monitored from the correct point and checks the starting on runtime from Management Console or Management Perspective.

4.1 Restartable Processes with Start Monitoring

Restartable processes refer to business models that are designed in such a way that they can be restarted at any point of their execution without losing or corrupting any data or the model instance status after restart: when restarted, such model instances skip through the activities that were already performed until they reach the status from before the restart. To design such models, you need to make sure that the models rely only on persisted data; the model instance itself must not at any point rely on any data that gets lost when it finishes.

For example, if you interrupt an order-dispatch process at a moment when the order is ready for dispatch, on the restart of the model instance, the process omits the invoicing and payment activities based on the order status and proceeds directly to the dispatch activity.

This design pattern is especially useful when you want to be able to `update the underlying models or application`. With restartable models you simply

- `finish the running model instances`;
- upload the new version of the model;
- start up the model instances again.

In the sections below you will learn how to:

- Design models that rely on the status for their execution station on persisted business data.
- Design an orchestrating model, that will start model instances over the business data anew. It will follow the start of the model instances so you can check if the starting finished successfully.

4.1.1 Designing a Restartable Process

We start from a simple flow that runs over a piece of business data stored in the process variable. The flow has two *User* activities which signal that the status of the business data changes or is about to change.

In the todo generated by the *User* activities the user simply submits their Todo.

At this point, the status of the business data is updated to a new status in the User Task assignment. Like this, the status is preserved, even if the model instance is terminated.

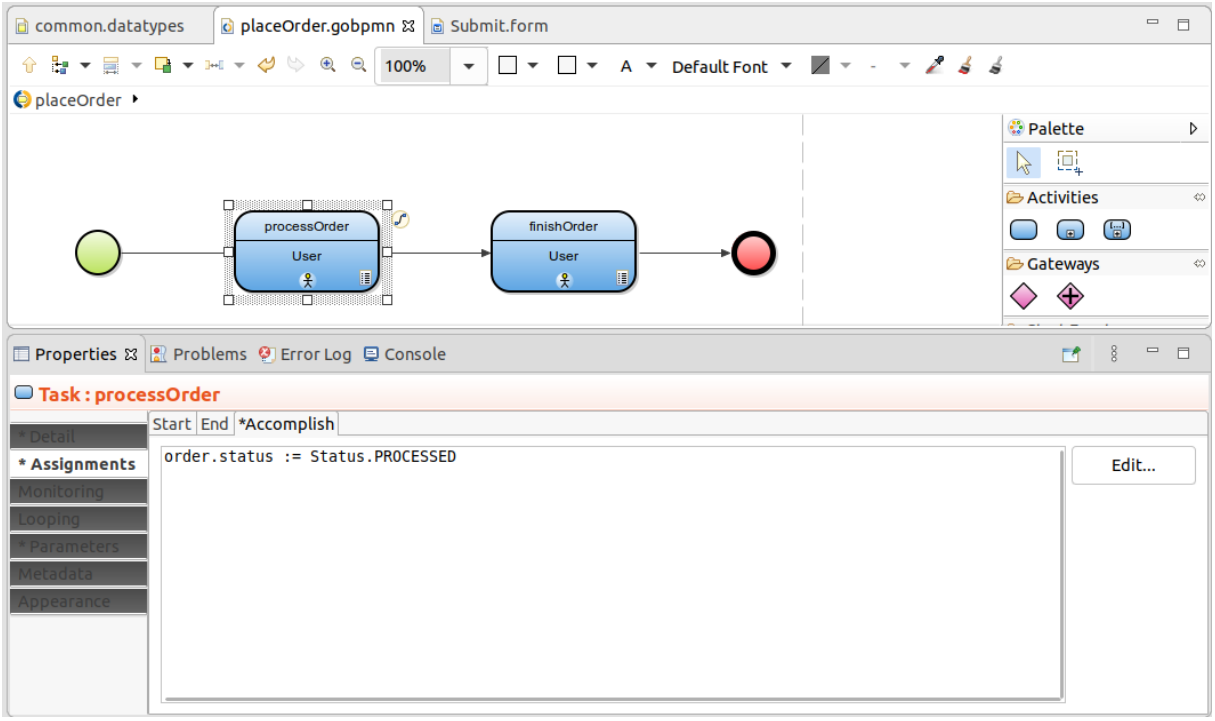


Figure 4.1 Assignment in the User Task that sets the status of the order variable when the user submits the todo

Therefore to make a model instance restartable, change your model as follows:

1. Persist all data relevant to the flow logic: Store it in Shared Records or their fields.

   In the example, we must change the type of the *order* process variable from a common record to a shared record: every model instance runs for a particular order. The status of the order process is determined by the *status* field of the *order* variable.

2. Adapt the way you acquire the business data so that the process gets the correct data when started.

   In the example, this means to initialize the *order* process variable to one of the following:

   - new *Order* shared record: no business data exists yet; the order is just being placed.
   - existing order: the model instance runs over an existing order; the order is passed as a parameter to the model instance.

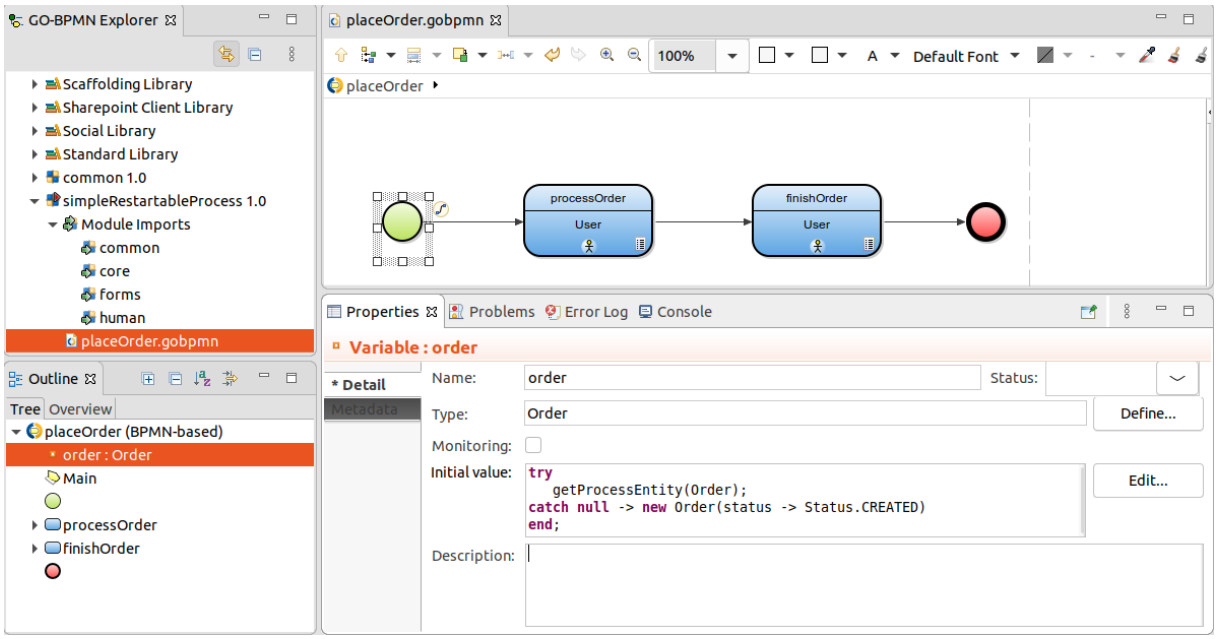


Figure 4.2 Initialization of the order process variable

3. Design the mechanism that skips over to the latest possible flow element when the persisted business data has some value, in the example, the *order* status value decides whether the default flow or the condition flow of the gateway is taken to the next User Task:

   (a) Add an exclusive gateway before any element that changes the status of your business data.

   (b) Define the flow that enters the element as default.

   (c) Design another flow that will go around the element.

   (d) Add the required condition on the non-default flow of the gateway.

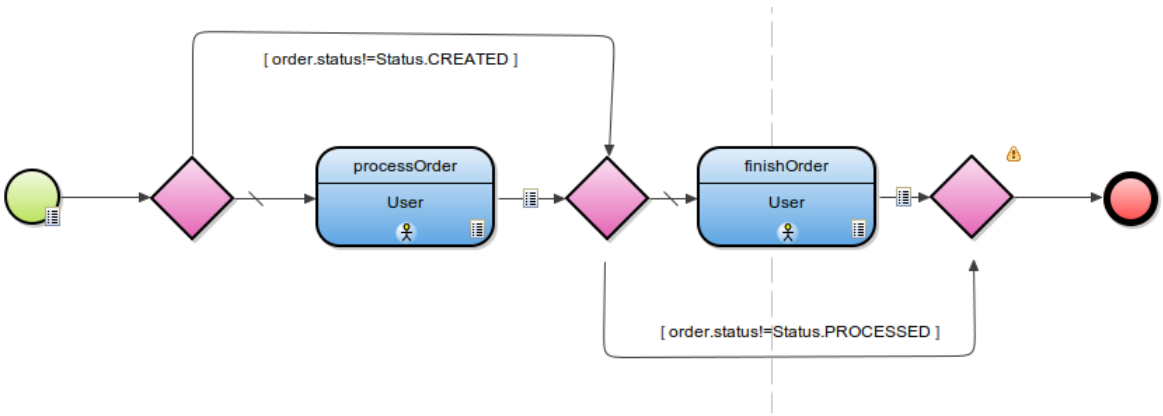


Figure 4.3 Restartable mechanism built around a simple flow

4. Define the orchestrating model which will run a model instance of the order process over each *Order* instance.

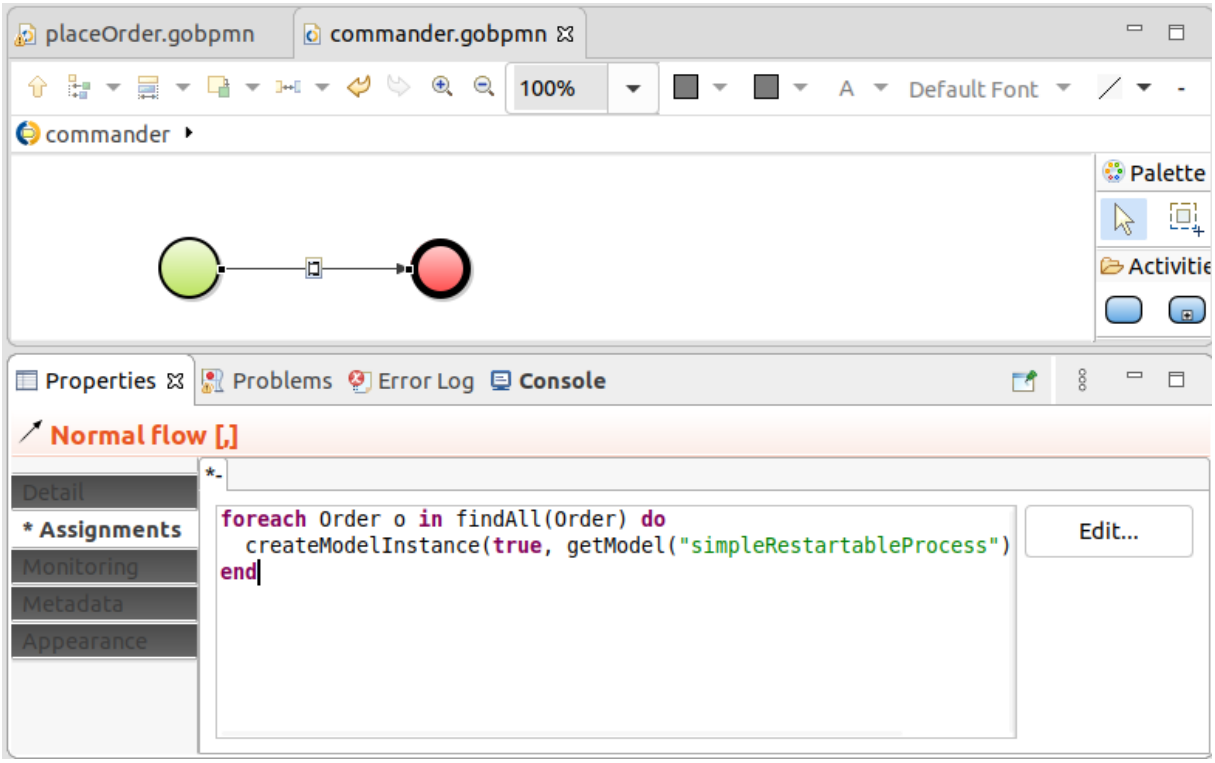


Figure 4.4 Process that runs a model instance over each order record

5. Insert the mechanism that will monitor that the starting of the order model instances and the orchestrating model instance succeeded:

   (a) In the orchestrating process, call `watchStarting()` on the started model instances. This will activate the monitoring of its start.

```
foreach Order o in findAll(Order) do
 def ModelInstance i := createModelInstance(true, getModel("simpleRestartableProcess"),
 watchStarting(i);
end
```

   (b) Add the `clearApplicationRestartData()` call before the `createModelInstance()` calls: this will remove the data about the previous model-instance starting.

   (c) Call `watchStarting(thisModelInstance())` after the `clearApplicationRestartData()` call on the orchestrating model as well; like this you can check if the starting of this instance was successful. Call the function from, for example, the Start assignment of the process. The location of the call does not really matter, however, it must be called from within the first transaction of the model.

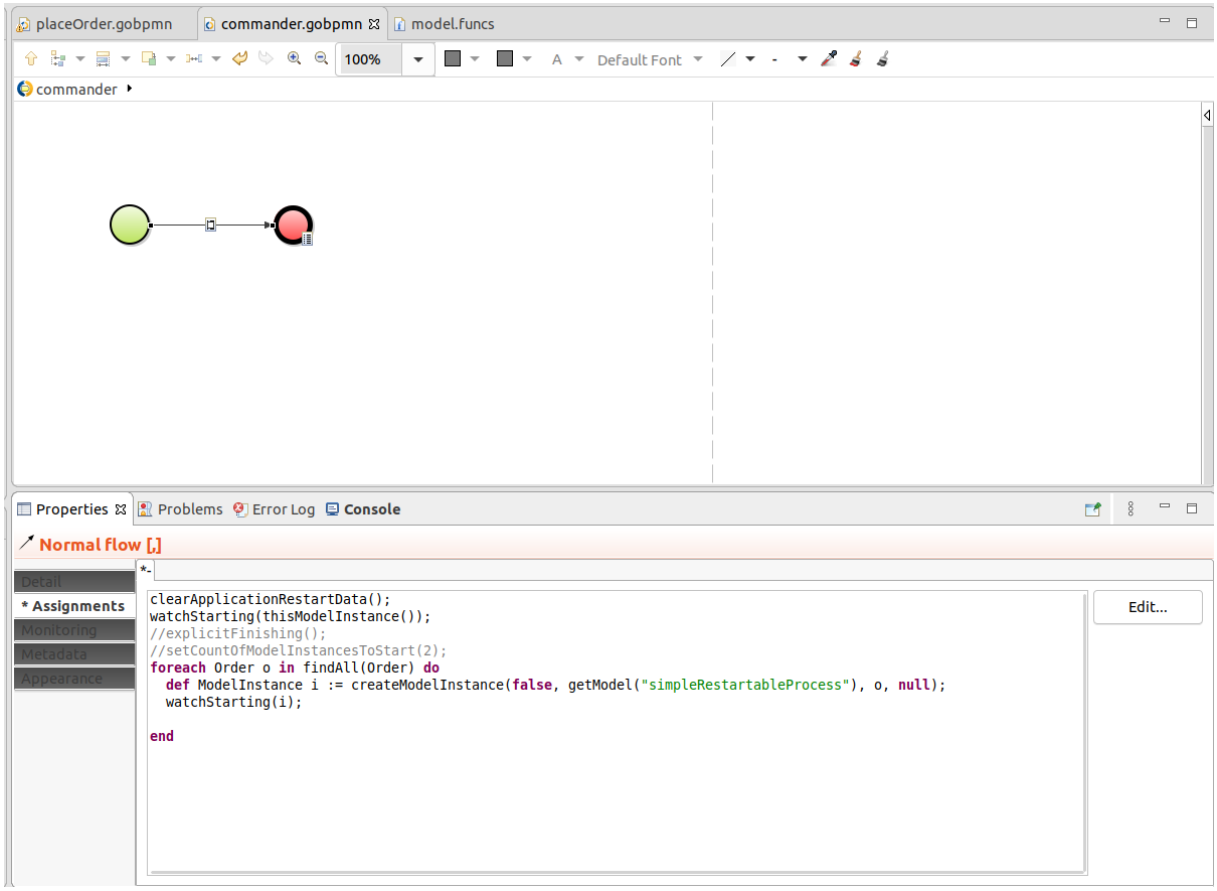


Figure 4.5 Orchestarting process with the start-watching calls on the Start Event

This is a very simple process and in more complex processes, the designing of the skipping might become tedious if not impossible. Consider extracting the skippable elements that change the status of your data to separate processes and create a process that will call them in Reusable Subprocess activities. For a full example, refer to the agile pattern.

To test the model:

1. Run a few instances of the business model and submit some of the todos to produce some testing data.
2. Finish all running model instances.
3. Run the orchestrating model instances.
4. In Management Console or Management perspective, go to *Application Restart* and inspect the start monitoring data.

4.2 Agile Processes

Required result: A model that is restartable and allows the user to move between tasks arbitrarily:

- When the adminstrator restarts the model instance, the model instance will recover its original progress based on persisted data.

- When the user can move to any task in the workflow.

  Note: You can download an example implementation here. To import the model, do the following:

  1. Create a GO-BPMN project.
  2. Right-click the project and select *Import* > *Archive file*.
  3. Enter the path to the zip file into the *From archive file* field.
  4. Click **Finish**.

Patterns of agile mechanisms solve the following:

- **Set the correct execution state after restart**: on restart, the process omits activities that were already performed.

  For example, if you interrupted an order-dispatch process at a moment when the order is ready to be dispatched, on restart, the process omits the invoicing and payment activities and proceeds to the dispatch activity.

  This also allows you to update the underlying model easily: you stop your model instances, upload a new version of the model, and resume the stopped model instance according to the new model. The new model instance get into the same or equivalent execution status as the original model instance on resume.

- **Skip arbitrarily through activities**: skipping is used to implement such features as breadcrumb navigation; the user can switch between activities freely. On switch, the process deactivates the current activity and activates another.

The pattern is as follows:

- Each "skippable" flow sequence is implemented as a process or a task type: the sequence has the *activity reflection type* enabled so it can be triggered by the Execute task.
- The sequence is executed by an Execute task of a wrapper process, which wraps the Execute task in the skipping mechanism.
- The wrapper process takes a parameter with the current step: if the current step does not correspond to the required step, the Execute task is not executed.
- The wrapper process is called as a subprocess from an orchestrating main process.
- If the *skippable* flow sequence signalizes that it should be deactivated, the Executable task handles the signalization, deactivates the wrapper process and activates another wrapper process.

4.2.1 Base

We will create a process that will represents one step of our process: The step will be then used multiple times in a main process in a series of Reusable Subprocesses. We will design the ommitting and skipping mechanisms in the step process.

1. Design the subprocess:
   (a) Create a BPMN-based process called *step*.
   (b) Unselect the *Instantiate Automatically* option in the process properties: If you leave the option selected, a bogus instances of the step process would be created everytime we instantiate the main process. Also, it would not be possible to define parameters for the process.

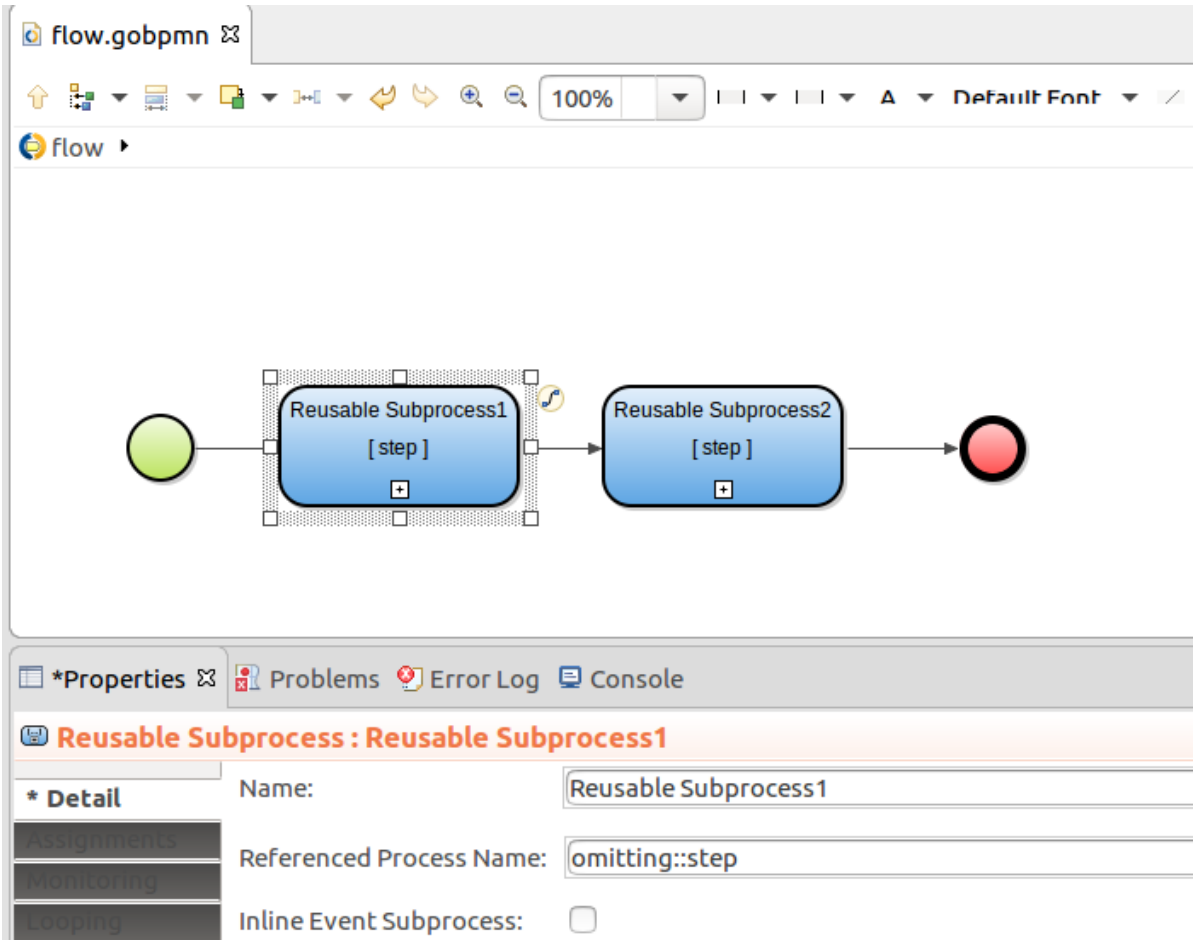
(c) Define the process parameters: these should provide data for the activity of the step.

For example, use the User task and design for it a form with a submit button.

2. Design the coordinating parent process:

(a) Create process definition with a BPMN-based process; name it *flow*.

(b) In the *flow* process, create a flow of Reusable Processes with the *step* process.



3. You now have a working model: run it and check that the process instance has one step sub-Processes running at a time and that the step sub-process is not instantiated as its own process instance.

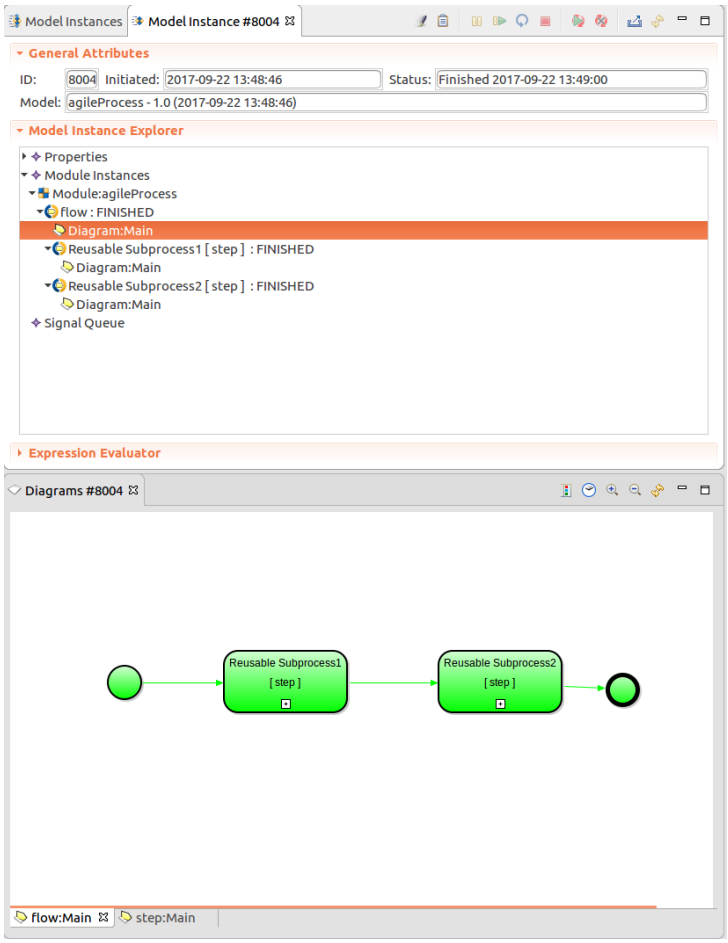


Figure 4.6 Model instance details of a successful run

4.2.2 Skipping

The skip mechanism allows you to restart your model instances at any point without having to worry about data consistency or flow interruptions.

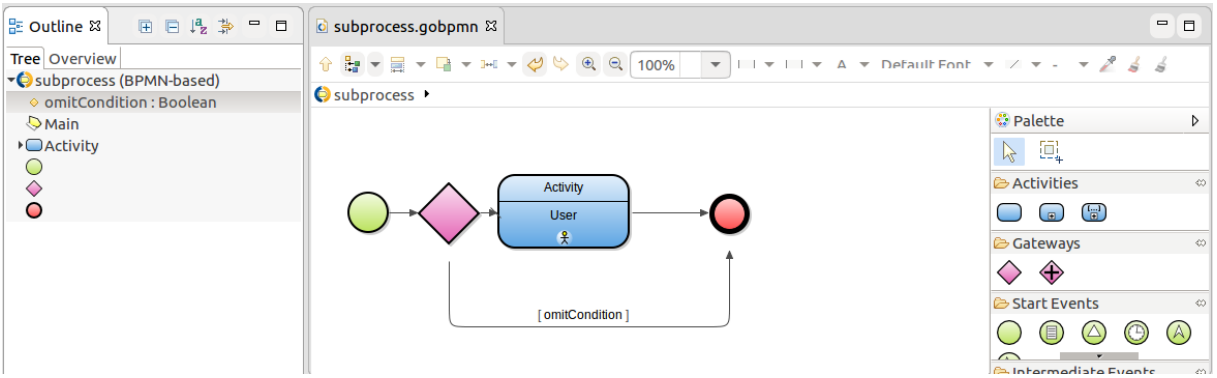
To implement it, we will make the *flow* process send data on whether the *step* process needs to execute its Activity to the *step* process. The value of the data should depend on persisted data so it remains unchanged in case of model instance restart. However, we will use a global variable to store the data to keep it simple.

Goal: Create the skipping mechanism inside *step* around its Task and make the *flow* send the *omitCondition* value to *step*.

1. In the *step* process, design the evaluation of the condition:

   (a) Define the `omitCondition` parameter of type Boolean: the parameter will be sent by the parent process.

   (b) Around the step task, design the workflow that will avoid the Activity when the `omitCondition` will be true.



(c) Make the flow pointing to the activity the *default* flow.

2. Create a global Integer variable *lastSuccessfulStep* with initial value 0: it represents the last successfully executed *step* process.

3. In the *flow* process, pass the condition based on the last successful step as the *omitCondition* argument to each Reusable Process.

   - on the first Reusable Subprocess `omitCondition -> lastSuccessfulStep >= 1` (When the last successful step equals or is larger than 1, the condition is `true`.)

   - on the second Reusable Subprocess `omitCondition -> lastSuccessfulStep >= 2` (When the last successful step equals or is larger than 2, the condition is `true`.)

4. Test the process, set the initial value of *lastSuccessfulStep* to a value (0, 1, and 2) and run a model instance with the value. Check the behavior of the subprocesses: make sure the skipping works as expected.

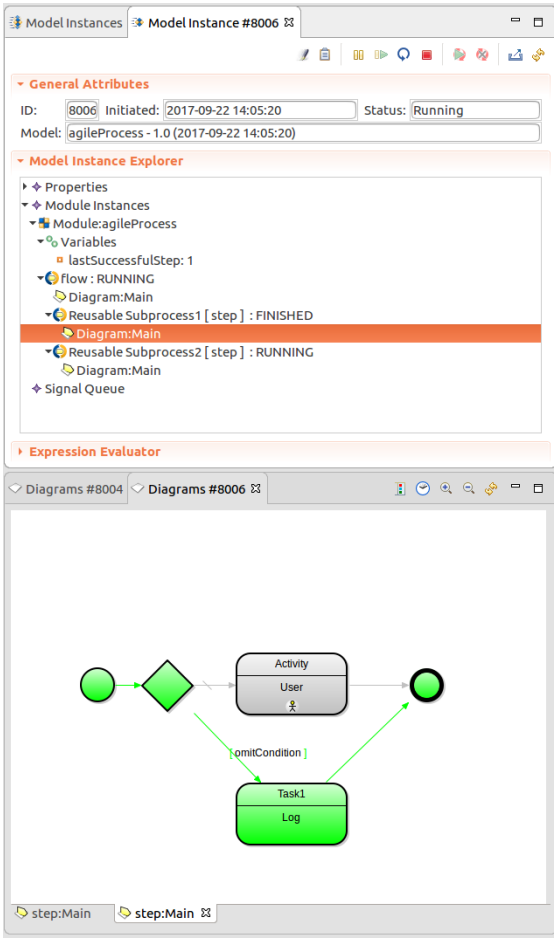


Figure 4.7 Run with lastSuccessfulStep set to 1. The subprocess that used the omitting flow displayed below.

Points to Consider

- While here you are omitting the same task over and over again, in real world scenarios, you will omit different types of Activities:

  Change the type of the User task in the *step* sub-process to the *Executable* task type and pass the user task in the *activity* parameter of the sub-process along with the condition parameter.

- To evaluate the conditions for omitting, do not use global variables: use data persisted in the database.

4.2.3 Deactivation

The deactivation mechanism terminates the current sub-process instance under specific circumstances. It could be either when it receives a signal or when a condition becomes true.

We will use a Signal: To the Task in the *step* process, we will attach an interrupting Catch Signal Intermediate Event. It will wait for the Task to throw a Signal: when this happens, the Signal Intermediate Event will be activated and the Task deactivated. The execution will take the outgoing flow of the event, which will enter a No Exit End Event. The end event will terminate the sub-process instance without letting the subprocess produce a token: this will prevent the parent process from continuing its execution.

To design the deactivation mechanism, do the following:

1. Adapt the Task in the *step* process so it throws a Signal; for example, if you are using a User task, add to its form a *Deactivate* button, which calls `sendSignal(false, {thisModelInstance()}, "deactivate")` when clicked.

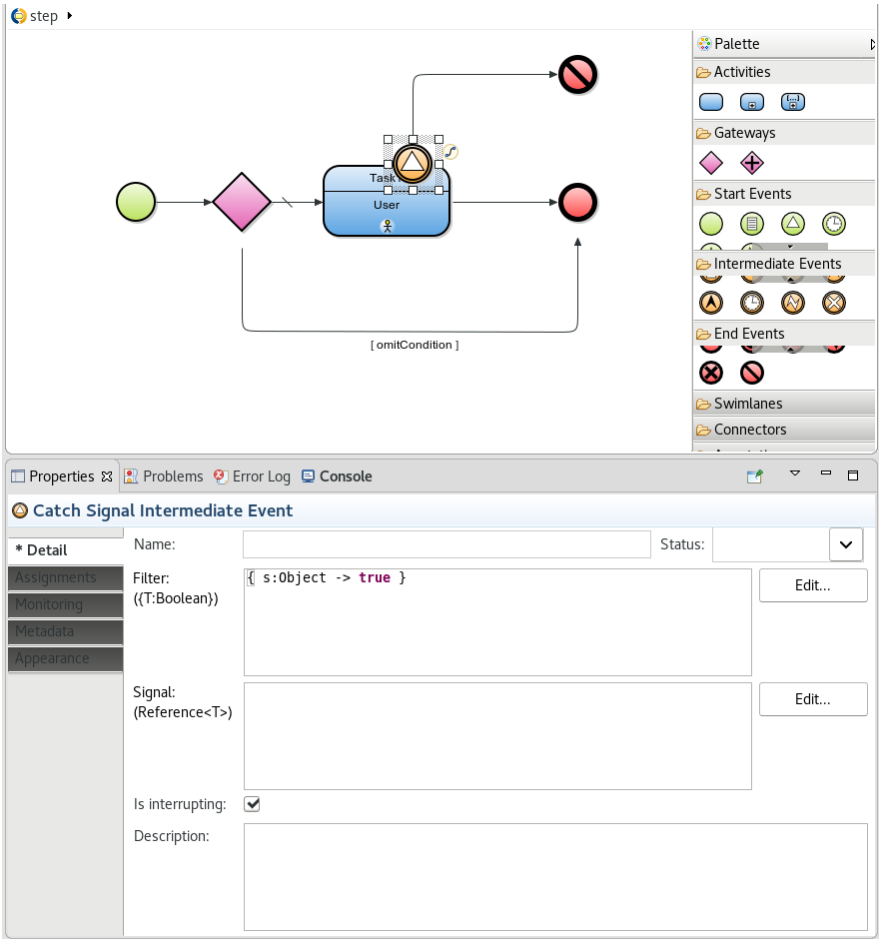
2. Adapt the *step* so it finishes when it receives a Signal:

   (a) Add an *interrupting* Intermediate Catch Signal Event to the boundary of the Task.

   (b) Set the filter to `{ r:Object -> true}`) so it catches any Signal.

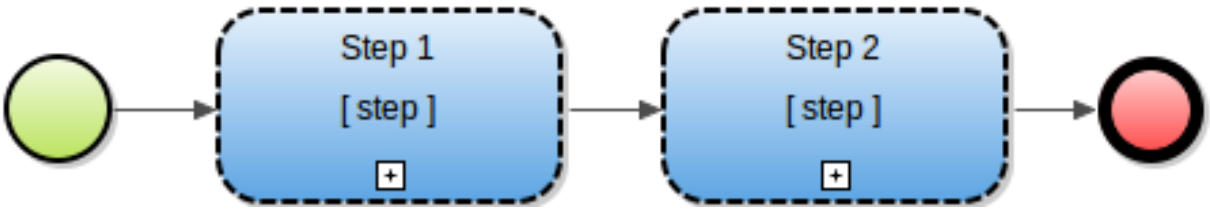
   (c) Connect the event to a No Exit End Event:

   The No Exit End event ends the execution flow of the reusable sub-process just like Simple End Event. Unlike Simple End Event, it prevents the execution to continue out of the Subprocess: The Subprocess does not produce a token.



3. In the *flow* process, change the Reusable Subprocesses to Inline Event Subprocess.

   Inline Event Subprocesses are considered a part of the parent process, that is, the *flow* process. They are instantiated as process instances while non-inline-event subprocesses create subprocess instances. Only Inline Event Subprocesses can finish with a No Exit Event.



4. Run the model and deactivate it in one of the steps.

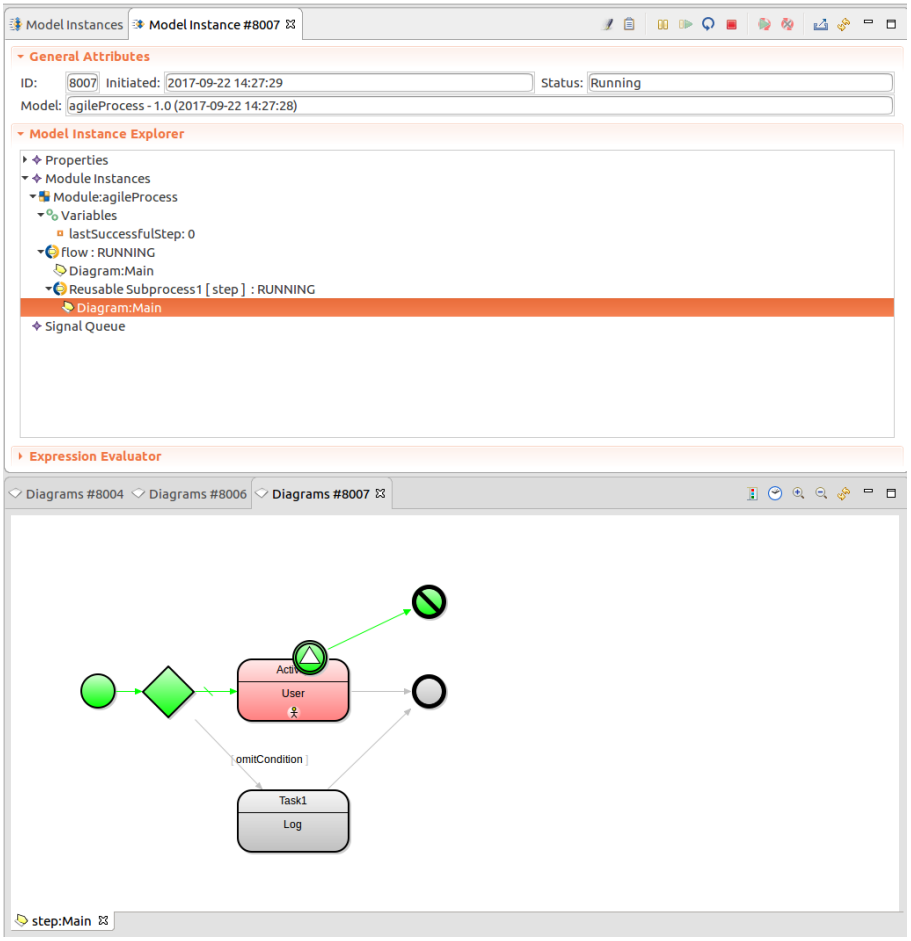


Figure 4.8 Model instance deactivated in the first subprocess. The diagram with the activity deactivated by the Catch Signal Event is below.

4.2.3.1 Summary

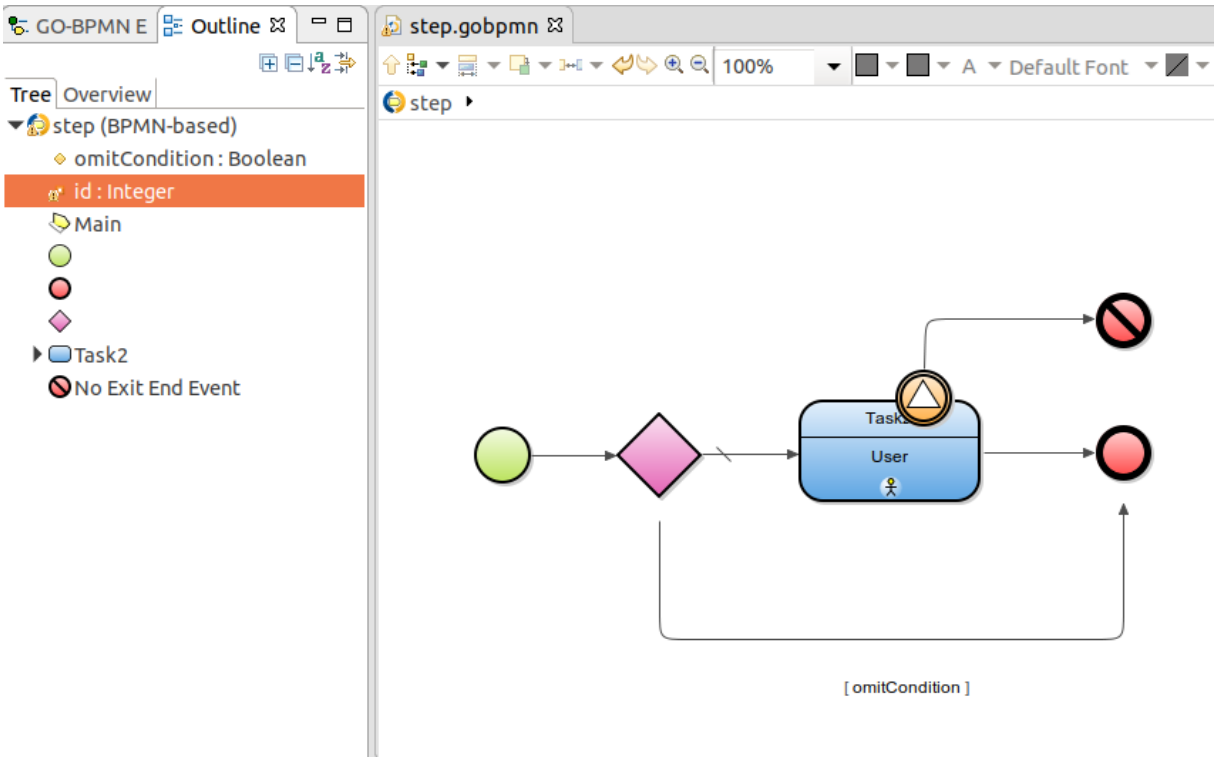
You have implemented the following behavior:

1. The User Task or another Activity of the *step* process can throw a Signal.
2. When this happens, the Catch Signal Intermediate Event catches the signal and terminates the Activity.
3. The outgoing flow of the Catch Signal Intermediate Event is taken.
4. The No Exit Event end is executed: The *step* subprocess instance finishes.
5. The *flow* process instance finishes since no sub-process instance is running.
6. The model instance is deactivated.

4.2.4 Activation

The model can now omit already performed Activities and deactivate their *step* processes. Now you need to let the *flow* process activate the correct step when another step is deactivated. You will pass the information to the *step* process as a parameter in the deactivation Signal:

1. Add the `id` Integer parameter to *step* so you can identify the step to activate. The parameter will be populated by the *flow* process depending on the position of the Reusable Process in the flow and hold the step that should be activated upon deactivation.



2. Add the information on which step should be activated upon deactivation to the deactivation Signal you are sending from your Task:

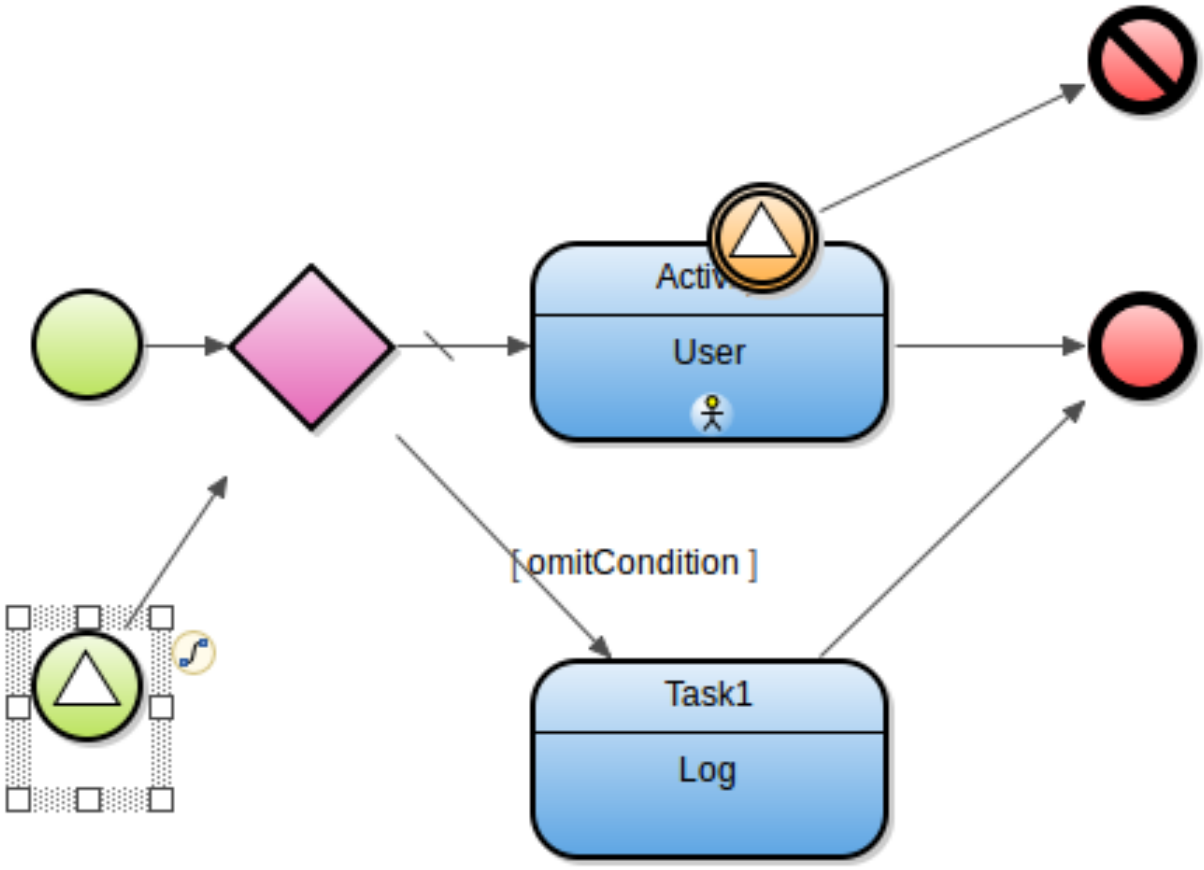
```
sendSignal(false, {thisModelInstance()}, goto)`
```

   If you are using a User Task as your activity, rename the **Deactivate** button to **Go To Activity** and navigate away from the to-do, when the user clicks it so the user is not stuck on a page with the to-do of the deactivated task.

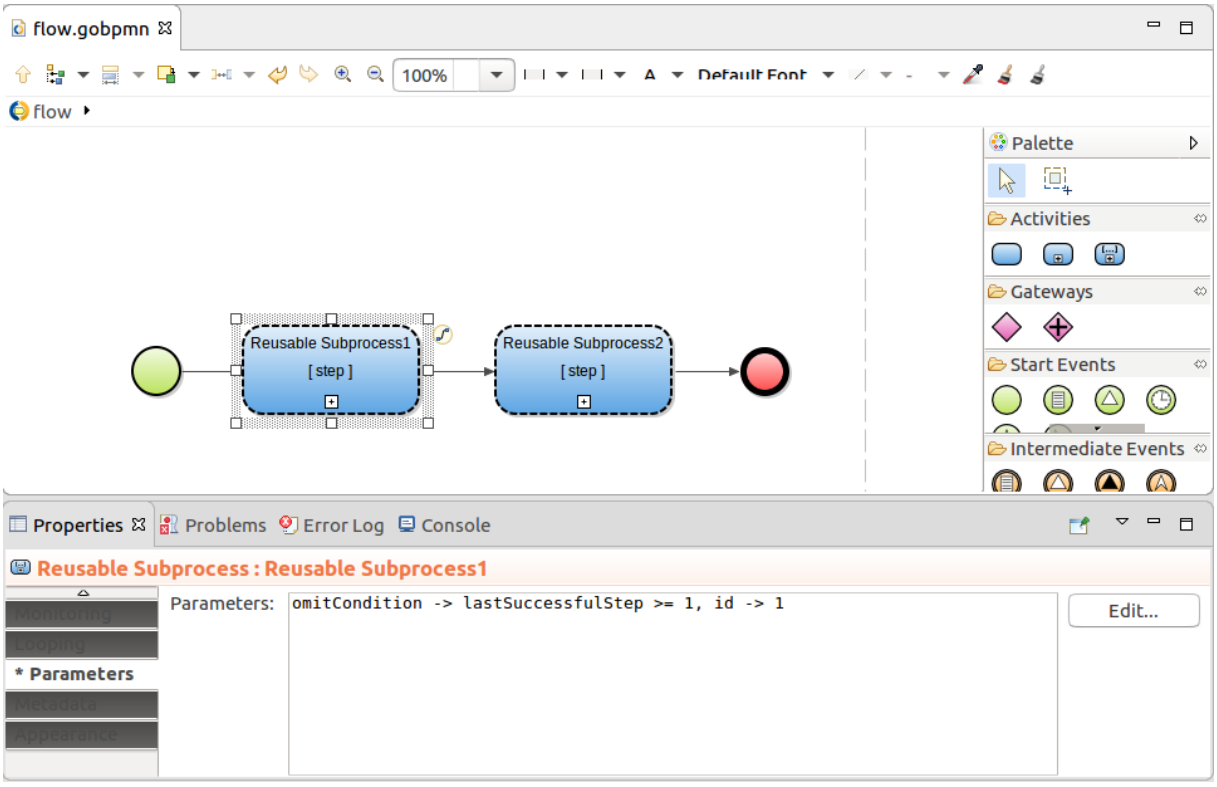
3. Still in the *step* process, add the Signal Start Event

   Since *step* is an Inline Event Subprocess, all Signal Start Events in all *steps* will be listening for a Signal and can be activated when a Signal is received.

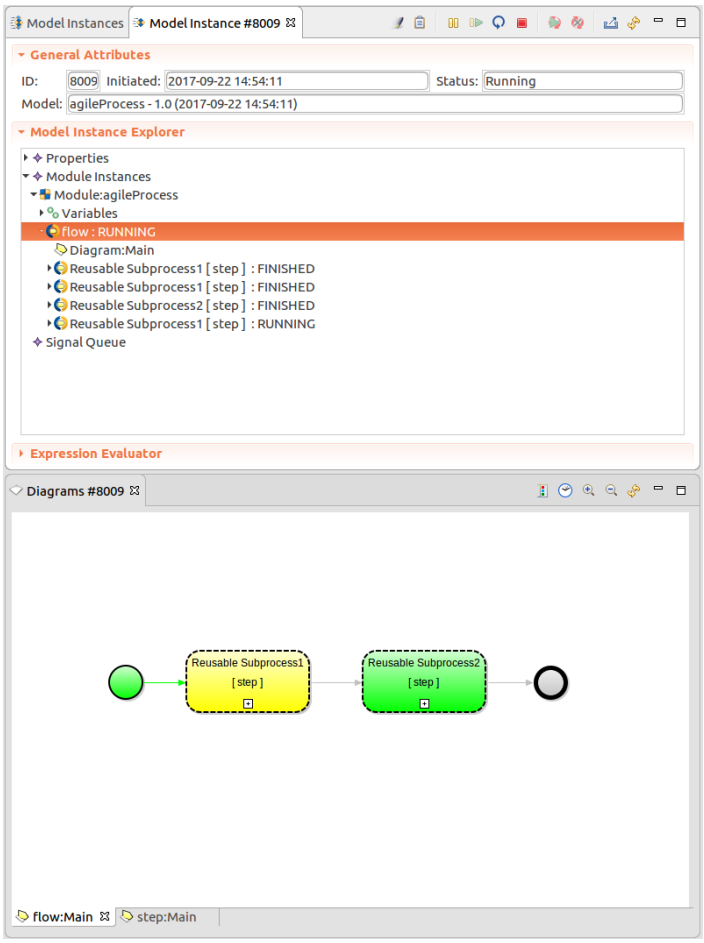
4. Define the Signal Start Event filter so the *step* process is activated only if its id parameter set by *flow* matches the `goto` parameter sent in the Signal: `{ activateStep:Integer -> activateStep == id }`



5. In the *flow* process, add the *id* parameter value to the reusable sub-processes and the *goto* parameter.



6. Run the model.



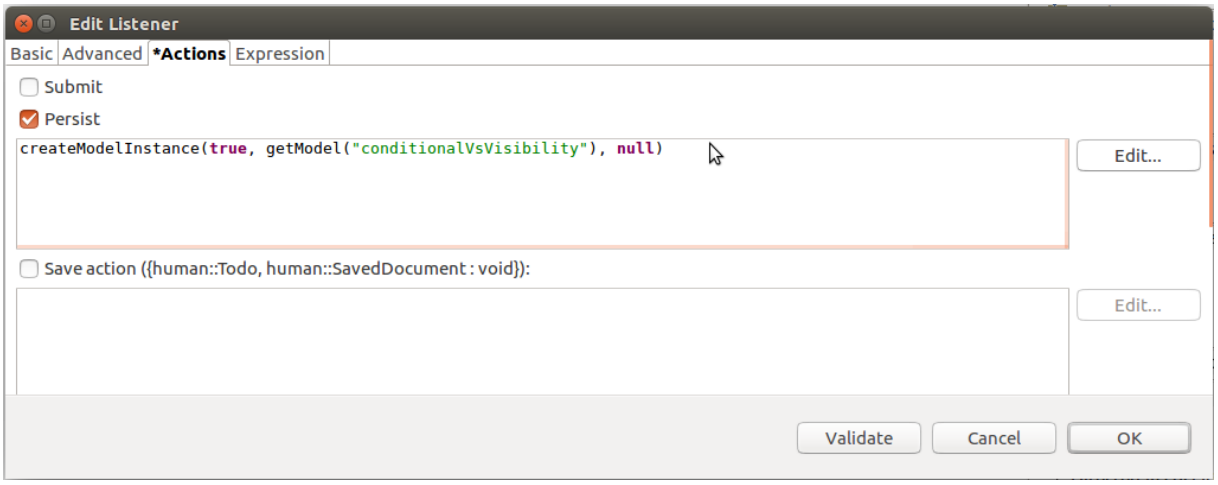
Points to Consider

- Typically, you will skip to a step based on input provided by a user in a to-do: in such a case, you will need to adapt the respective form so it passes the go-to data: a listener could send the signal with a *goto* value from an input field.

4.3 Creating a Model Instance from Document and Navigating to its To-Do on Submit

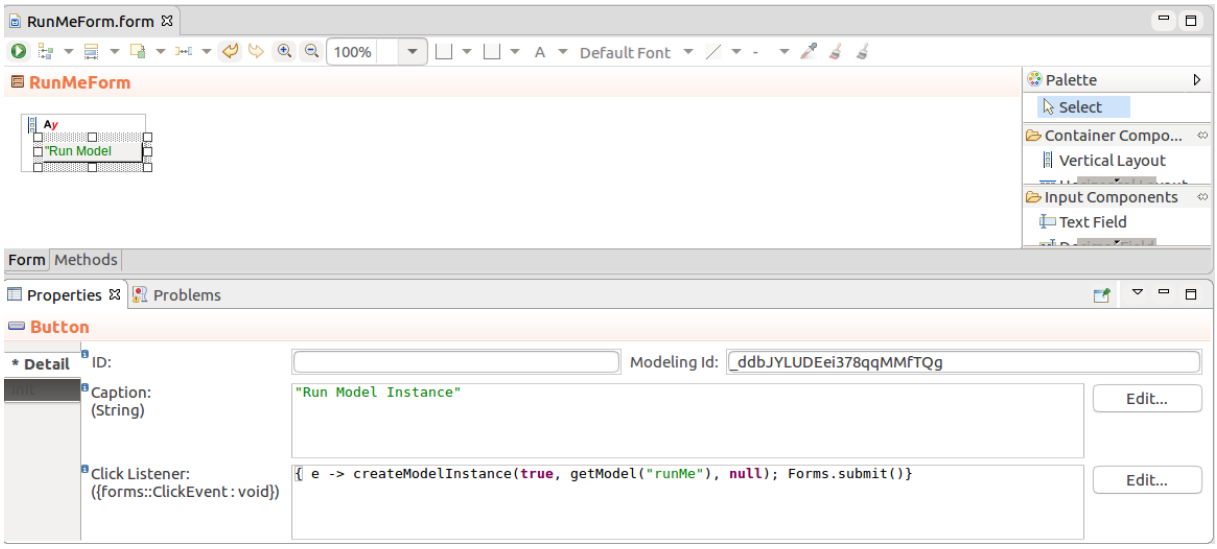
To create a model instance from a document and then navigate to one of its To-dos, do the following:

1. Open the form of your document:
   - For ui-module forms, define a listener that will create the model instance as follows:
     (a) Attach a listener of the required type to a component.
     (b) Create the model instance in its persist action, for example, `createModelInstance(true, getModel("myModelName"), null)`
     (c) Define a listener with the Submit action (it represents the moment when you want to submit the data and navigate away from the document).



- For forms-module forms, define the following expression on the respective component listener (typically the click listener of an action component, such as a Button):

```
{ e -> createModelInstance(true, getModel("runMe"), null); Forms.submit()}
```



2. Optionally, define the Navigate property in the document definition so the document navigates to a to-do generated by the model instance when submitted:

```
\navigates to the first to-do generated by the document:
{todos:Set<Todo> -> new TodoNavigation(todo -> todos[0], openAsReadOnly -> false)}
```

4.4 Monitor the Start of Model Instances

Monitoring the start of model instances allows you to check if a model instance has successfully finished its start sequence, which is by default the first `model transaction`.

Monitoring the start of model instances is typically useful when starting restartable model instances. Restartable model instances are based on a pattern that makes sure that a model instance returns to the correct status after if finished prematurely.

Such models rely on persisted business data for their status: If a piece of business data has been already handled by the given flow element or has been otherwise processed so that the element does not need to be executed, the element is skipped. More details on the pattern are available in the tutorial on restartable models:

When finishing and subsequently starting restartable model instances with the monitoring mechanism in place, you can check when and if the model instances have reached the status they were in before restart: You can start and check the status of the model instances manually, however, it is recommended to design an orchestrating model that will start and monitor the starting of model instances over your business data.

4.4.1 Monitoring the Start of Model Instances

To implement the monitoring of model instance starting:

1. Create a model with a process that will orchestrate the starting:
   (a) Monitor starting of the orchestrating model: call `clearApplicationRestartData()` to clear the data from a previous start-monitoring run and `watchStarting(thisModelInstance())`, for example, from the Start assignment of the process.
   (b) Make the process create model instances of the restartable model and pass it the relevant business data.
   (c) Mark the place from where to start monitoring the model instances by calling `watchStarting(<ModelInstanceToFollow>)`: At this point, the starting status of the model instance will be set to `IN_PROGRESS`.

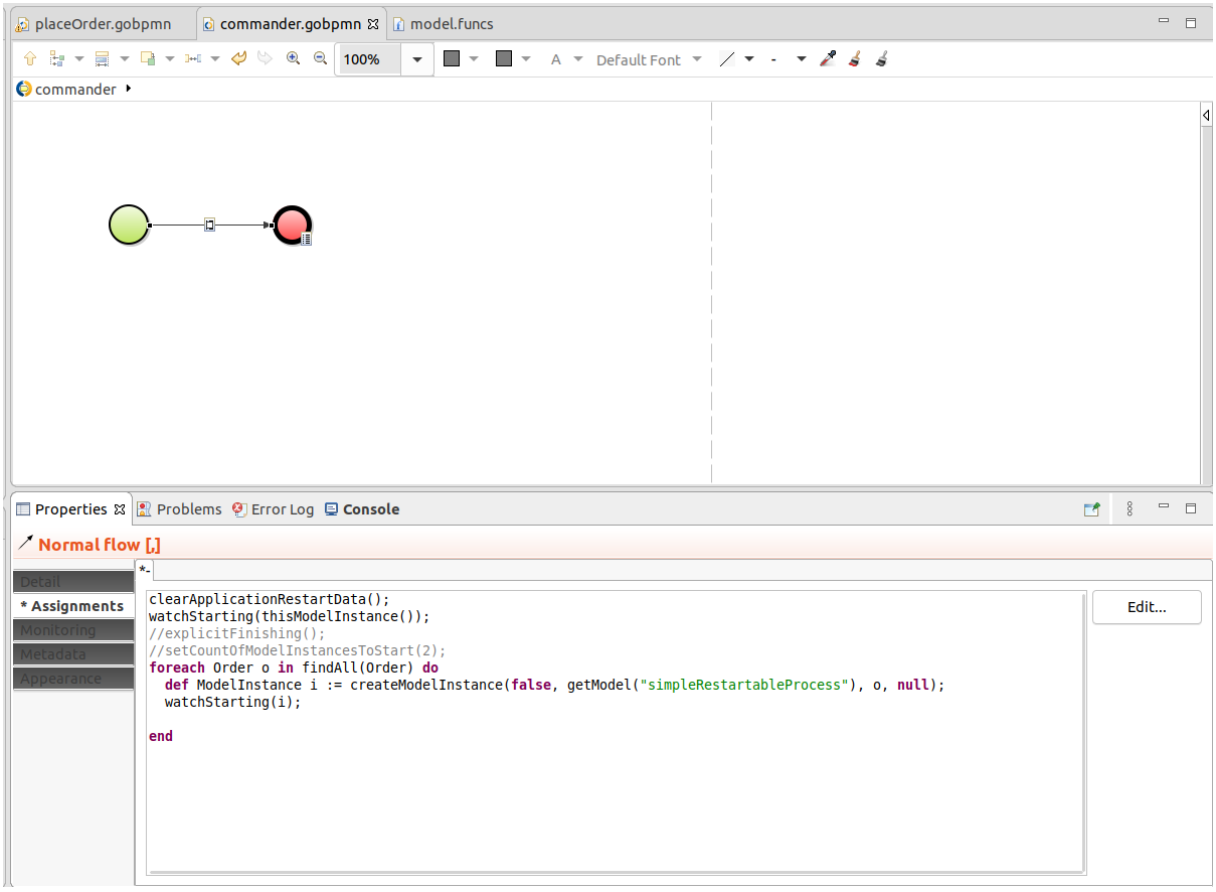


Figure 4.9 Orchestarting process with the start-watching calls on the Start Event

2. Adapt the restartable models:
   (a) Optionally, define where the start sequence finishes: By default, this is when the first transaction of the monitored model instance has been executed successfully: the starting status of the model instance is set to `FINISHED`. To finish the start sequence later, define the point when to finish manually.

4.4.2 Defining Finish of the Start Sequence

To define, when the starting of a model instance finishes explicitly do the following:

1. Along with the `watchStarting()` call, call `explicitFinishing()` to prevent the watching from finishing after the first model transaction.
2. Mark the finish of the start sequence with the `modelInstanceStartSucceeded()` call.

With the `explicitFinishing()` in place, the starting status of the model instance will be `IN_PROGRESS_EXPLICIT_FINISH`.

4.4.3 Defining Number of Expected Model Instances

You can define the number of expected model instances that finish their Start with the `setCountOfModelInstacesToStart()` call: the number is displayed in the Application restart views.

4.4.4 Checking the Start Progress of Model Instances

You can follow the status of starting of model instances in the *Restart Application* view in Management Console or in the Management perspective. Also, you can use the `getStartStatus(ModelInstance modelIntance)` and `getStartStatuses(List<ModelInstance> listOfModelIntances)` and `getRestartInfo()` functions of the Standard Library or the Command Line tool with the `appRestartInfo` and `appRestartInfoExport` commands.

If the status of a monitored model instance is `FAILED`, the start sequence of the model instance failed with an unhandled exception.

Note that the orchestrating model instance might fail as well: this will depend on whether the exception occurred within *its* start sequence.

Chapter 5

Data Model Tutorials

5.1 Creating Custom To-Do List

> **Important:** To complete this tutorial, you need the Enterprise Edition of PDS and the LSPS Maven Repository installed.

Typically, the default To-Do list will not cut it in the real world of business and you will require custom business-related data for a to-do while retaining the related mechanisms, such as, priority of the todo, its allocation, locking, annotations, delegation, substitution, etc.

Since the Todo is represented by the `Todo` record which is a *system* record, you cannot simply add a field to it. However, you can create a new Record related to the Todo Record and add the business data to this Record: in this tutorial, we create the `TodoItem` record with an additional field and a relationship to the `Todo` record:

- To create instances of TodoItem on runtime, we will create the record in the `issueAction` parameter of the User tasks.
- To query the to-do information of a TodoItem record, we will use joins from the TodoItem to the Todo.

> **Note:** The pattern of records related to system records can be applied analogously to other system records, for example, to extend the data held by `Person`.

5.1.1 Creating the Data Model

Before you start, create a project:

1. Open the *Modeling* perspective in your PDS.
2. Go to File -> New -> GO-BPMN Project.
3. In the pop-up enter the project name `custom_todo_list_model` and click **Finish**.

Since *Todo* is a system record and system records cannot be extended directly, you need to create a record that will represent our todo item with the business data and is related to the Todo system record:

1. Create a module that will hold the data hierarchy:

   (a) Go to File -> New -> GO-BPMN Module.

   (b) In the popup, do the following:

   - Select the `custom_todo_list_model` project.
   - In the *Module name* field, enter `custom_todo_list_data`.
   - Unselect the *executable module* option since this module is intended as a module import and never be instantiated by itself.

   (c) Click **Finish**.

2. Create a data type definition: right-click the module and go to New -> Data Type Definition.

3. Create the shared record with the business data, `TodoItem`:

   (a) Right-click the canvas in the graphical editor and go to **New** > **Shared Record**.

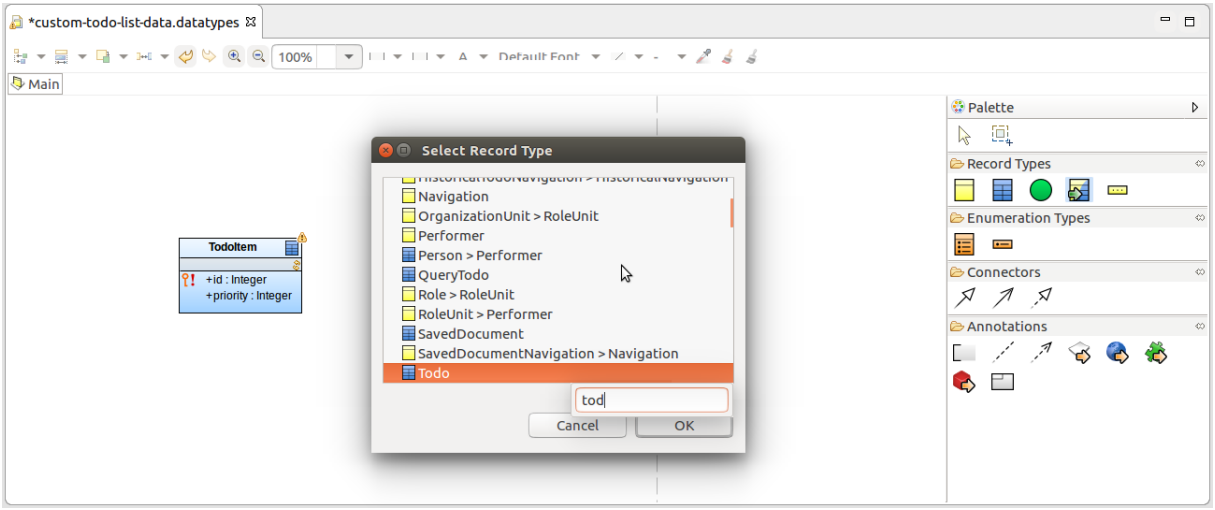
   (b) Enter the record name `TodoItem`.

   (c) Insert the field `priority` of type Integer into the `TodoItem` record.

4. Establish a relationship to the Todo record:

   (a) Right-click the canvas in the graphical editor and go to **New** > **Record Import**.

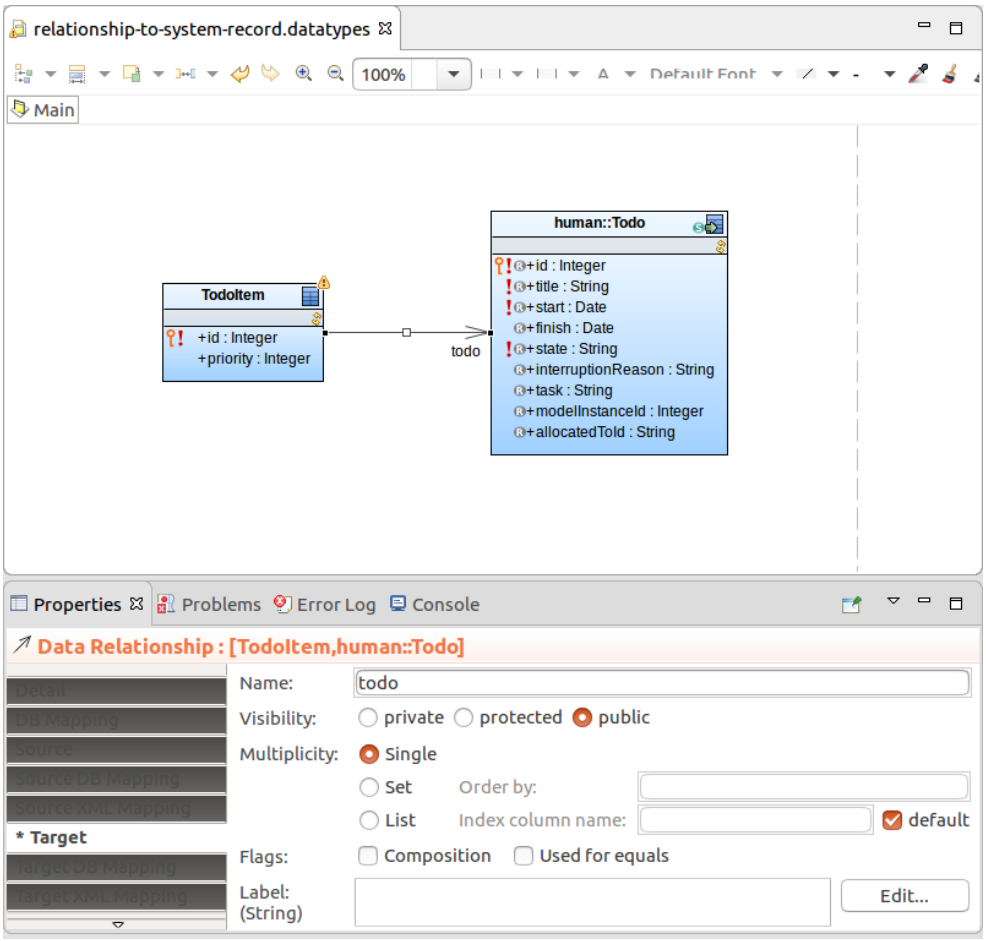
   (b) In the *human* module, select Todo (alternatively start typing `todo`) and click **OK**.



   (c) To create a relationship between TodoItem to Todo, drag the quicklinker from TodoItem to Todo.

   (d) Select the relationship and set the properties of the Todo end in the Properties view:

   - Name: `todo`
   - Multiplicity: `Single` (one TodoItem relates to one Todo)



Note: To display the fields and methods of the imported Todo record, right-click the record and under *Compartments* select the required items.

5.1.2 Creating the Todo Items

Todos are created when a User Task of a process instance is executed: to create the todo item related to the todo, create it in the `issueAction` closure of the User Task: *issueAction* is executed right after a todo is created and has the todo created by the User Task as its input parameter.

Let's create a process that will create a Todo and its TodoItem:

1. Create the module that will hold the process:
   (a) Right-click your project and go to New -> GO-BPMN module.
   (b) In the *module name* field, enter `custom_todo_list_process` and click **Finish**.
   (c) Import the **custom_todo_list_data** module (double-click the module Imports node in the *custom_todo_list_process* module).
2. Create the process definition and design the process:
   (a) Right-click the *custom_todo_list_process* module and go to New -> Process Definition.
   (b) Enter the name `CreateTodoItem` and select the **BPMN-based process** option.
   (c) In the process, create a local variable `newTodoItem` of the *TodoItem* type (in the Outline view of the process definition, right-click the root node and select **New** > **Variable**). It will hold the new todo item, so we can pass it to the todo form where we will edit its priority field.
   (d) Design a process flow with a User task.

(e) In the Properties of the User task, define the parameters of the task: in the `issueAction` parameter, create the TodoItem record over the to-do:

```
title /* String */ -> "Dummy Submit for Guest",
performers /* Set<Performer> */ -> {getPerson("guest")},
uiDefinition /* UIDefinition */ -> new SubmitForm(newTodoItem) ,
issueAction /* {Todo:void} */ -> { t:Todo -> newTodoItem := new TodoItem(todo -> t)}
```

Note that the `SubmitForm` does not exist yet; we will create it in the next step.

Note: If you attempted to change the value of the related to-do directly, for example, on a flow assignment with `newTodoItem.todo.title := "";`, you will get a validation error since *Todo* is a system record and fields of system records cannot be accessed directly.

5.1.2.1 Creating the Form for the To-Do

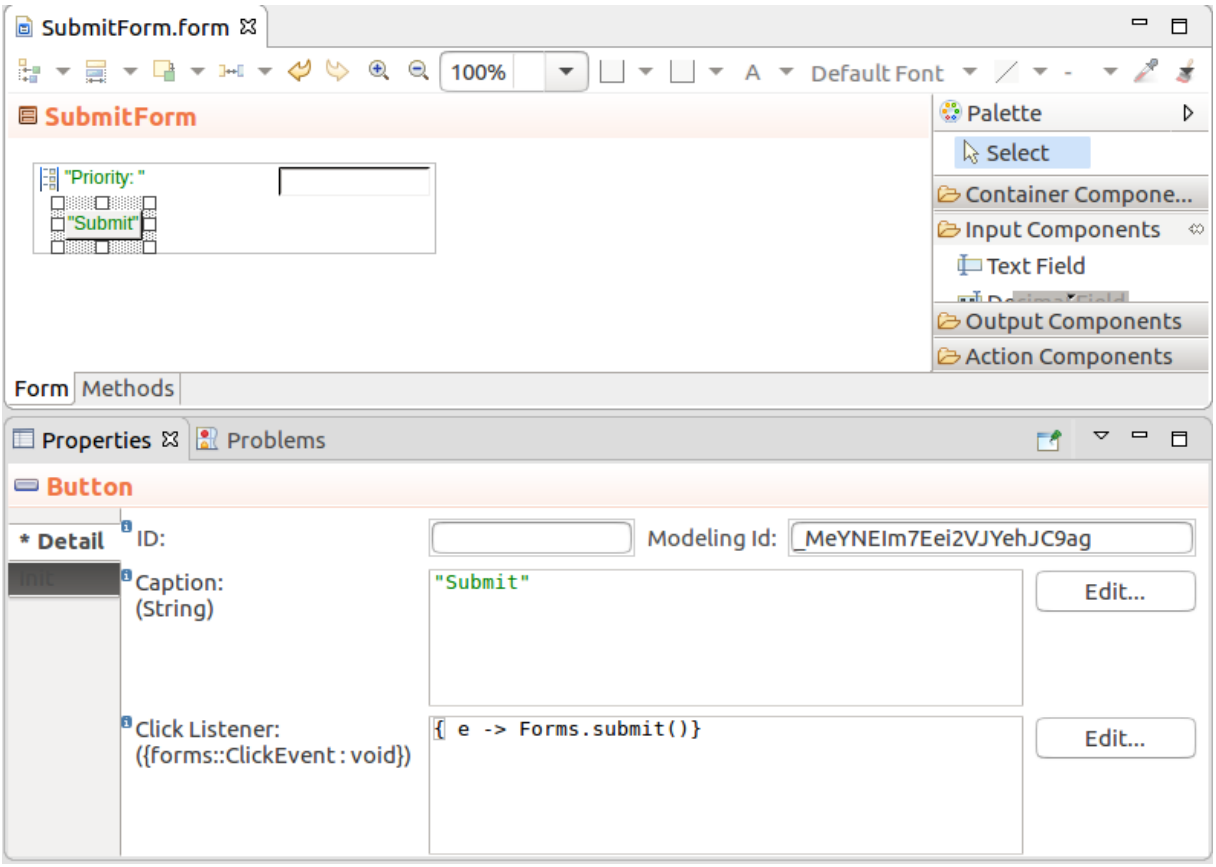
Create the form that will be used to gather the priority data for the TodoItem and submit the todo:

1. Right-click the `custom_todo_list_process` module and go to New -> Form Definition.
2. Enter `SubmitForm` as the name of your form and make sure the **Use FormComponent-based UI** is selected.

   Note: The **Use FormComponent-based UI** setting defines the module of the Standard Library that is used to create the form: When the option is selected, the `forms module` is used. Such forms are created more like in Vaadin and are a more powerful solution. When not selected the `ui module` is used. forms based on the ui module are used. Such forms are event driven and oriented on users without programming skills.
3. Create the form variable *newTodoItemVar* of type `TodoItem` (in the Outline view of the form definition, right-click the root node and select **New** > **Variable**).
4. Create a parametric constructor for the form that initialize the variable to the parameter value (open the form and display the Methods tab):

```
public SubmitForm(TodoItem newTodoItem){
   newTodoItemVar := newTodoItem
}
```

5. Back on the Form tab, insert the following components and define their properties in their *Properties* view:
   - Form Layout
   - Decimal Field with properties:
     - Caption: `"Priority:"`
     - Binding: set to Reference with the value `&newTodoItemVar.priority`
   - Button:
     - Text: `"Submit"`
     - Click listener: `{ e -> Forms.submit()}`



If you run the model now and the process executes the User task, and creates a todo along with its todo item. You can set the priority in the Application User Interface in the to-do.

5.1.3 Creating a List of Todo Items

Now we will create a page that will display the list of the todo items that have not been submitted yet (their todo is alive) and are assigned to the current user.

First, create a query that will retrieve todo items:

1. Right-click the *custom_todo_list_data* module and go to New -> Query Definition.
2. In the query editor, click Add.
3. On the right, define the query name as `getTodoItems`, set `TodoItem` as the record type, and set an iterator name, for example `ti`.
4. At this stage, the query returns all TodoItems. Restrict it so it returns only those todo items that are related to a LIVE todo:
   (a) Join the system todo table: select the **Join Todo List**.
   (b) Define the iterator for the returned todos in the Query Todo Iterator, for example `t`.
   (c) In the Todo List Criteria, define an expression that filters the todos from the joined todo list:

```
//returns todos of the current person:
new TodoListCriteria(person -> getCurrentPerson(),
 //exclude interrupted, accomplished, suspended todos:
 includeAllStates -> false,
 //exclude rejected todos:
 includeRejected -> false,
 //exclude to-dos allocated by other persons:
 includeAllocatedByOthers -> false,
 //exclude to-dos of substitutes:
 includeSubstituted -> false)
```

5. Now, the query returns all todo items related to a to-do of the current person. However, only the todo with the matching id should be returned. Define the condition in the **Condition** property:

```
ti.todo.id == t.id
```

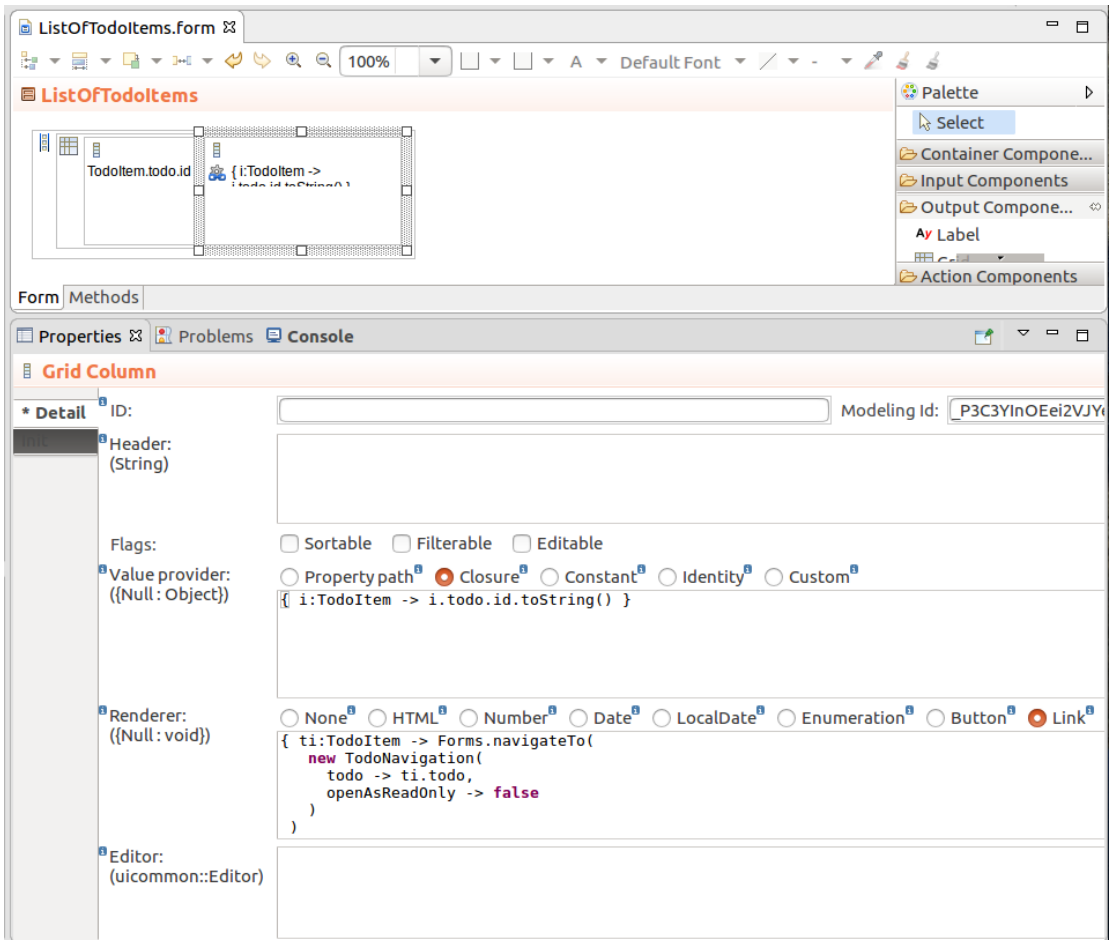
The query is ready and you can create a document with a form that will display the todo items:

1. Create the `custom_todo_list_ui` non-executable module that will hold the document.
2. Import the **custom_todo_list_data** module.
3. Create the document that represents the page with the todo items: Right-click the *custom_todo_list_ui* module and go to New -> Document Definition. Create a document with the properties:
   - **Name:** `todoItemsList`
   - **Title:** "`My Todo Items`"
   - **UI definition:** `new ListOfTodoItems()`
4. The UI definition does not exist yet, let us create it:
   (a) Right-click the *custom_todo_list_ui* module and go to New -> Form Definition.
   (b) Set the form name to `ListOfTodoItems` and click **Finish**.
5. In the editor with the form, insert a Vertical Layout.
6. Into the layout, insert the Grid component and define its properties:
   (a) Set **Data Source** to *Query* and `getTodoItems()` as its value.
   (b) Create Grid Columns with the value provider set to Property path with the respective custom todo item properties, for example, `TodoItem.todo.id`, `TodoItem.priority`.
7. Create a Grid Column that will contain a link which opens the task item:
   (a) Set Value Provider to *Closure* and define the closure that returns the link content below (You need to use the Closure type since a property path of type Integer cannot use the renderer Link):

```
{ i:TodoItem -> i.todo.id.toString() }
```

   (b) Set the Renderer to **Link**.
   (c) Below define the navigation of the link:

```
{ ti:TodoItem -> Forms.navigateTo(
  new TodoNavigation(
    todo -> ti.todo,
    openAsReadOnly -> false
    )
  )
}
```



If you haven't done so yet, now is the time to test the modules:

8. Run PDS Embedded Server by clicking the *Start Embedded Server* button  .
9. Generate todos: right-click the *custom_todo_list_process* module and go to **Run As** > **Model**.
10. Upload the document with the todo items list: right-click the *custom_todo_list_ui* module and go to **Upload As** > **Model**.
11. Create the `guest` user with all security roles.
12. Open your browser and go to `http://localhost:8080/lsps-application`, log in as the guest user
13. Go to Documents and click *My Todo Items*.
14. Click a link to navigate to the todo.
15. Stop PDS Embedded Server by clicking the *Stop Embedded Server* button  .

5.1.4 Removing the To-Do Navigation from the Menu

Now this is all nice and neat but the user can still access the default To-do List page: so we need to substitute the To-Do List in the Application User Interface with our to-do item list. To do this, we need to modify the Application User Interface itself.

First, generate LSPS Application:

1. Go to **File** > **New** > **Other**
2. In the popup dialog, select LSPS Application and click Next.
3. In the updated popup, enter the maven artifact details and click **Finish**.

For more details about the sources, refer to the `custom-application documentation`.

Let us remove the default To-Do List item and add our list item to the navigation menu:

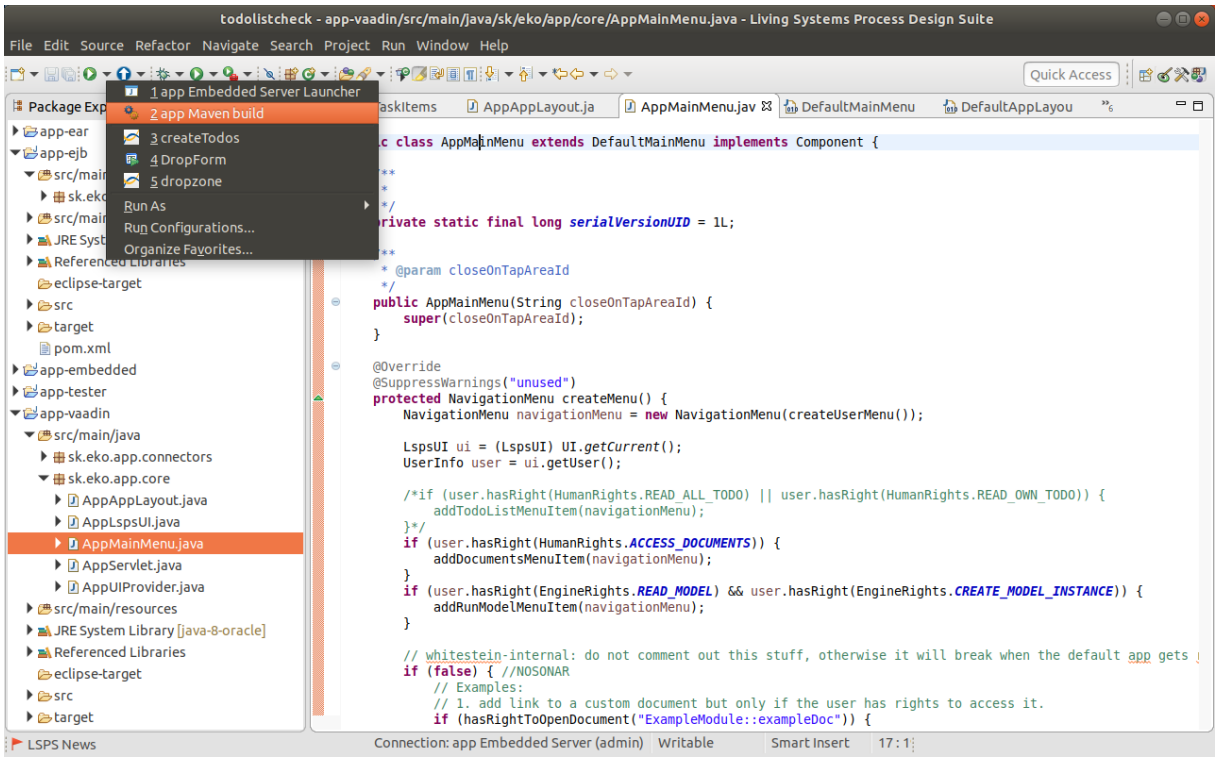
1. Since this is Java code, switch to the Java perspective.
2. Remove the To-Do List item:
   (a) Create a custom main menu:
      i. Create and open the `<YOUR_APP>.vaadin.core.AppAppLayout.java` class.
      ii. Modify the `createMainMenu()` method so it returns you custom MainMenu implementation.

```
@Override
  protected Component createMainMenu() {
    return new AppMainMenu(CONTENT_AREA_ID);
  }
```

   (b) Create the custom main menu:
      i. Create the AppMainMenu class that extends DefaultMainMenu.
      ii. Add the constructor from the DefaultMainMenu (there is a validation warning that the constructor is missing).
      iii. Copy and override the DefaultMainMenu's `createMenu()` method.
      iv. Remove the *if* statement with navigation item menu of the todo list:

```
@Override
@SuppressWarnings("unused")
  protected void createMenu() {
    /*if (ui.getUser().hasRight(HumanRights.READ_ALL_TODO) || ui.getUser().hasRight(Hu
      todoMenuItem = navigationMenu.addViewMenuItem(TodoListView.TITLE, TodoListView.
    }*/
    ...
```

   (c) Build the application: open the run configuration drop-down menu and click the build launcher.



(d) Run **SDK Embedded Server**: open the run configuration drop-down menu and click the embedded server launcher for you application.

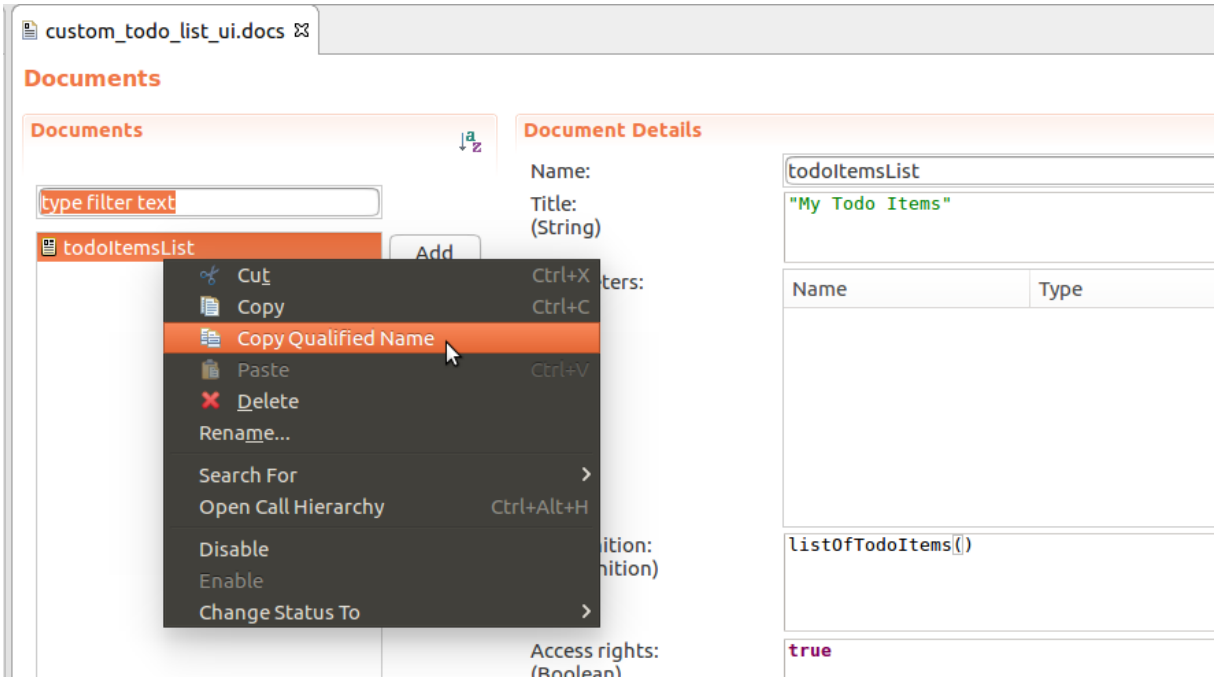
> **Important:** Note that this is a different server from PDS Embedded Server we used previously. PDS Embedded Server is generated on its first launch in the given workspace with the LSPS Application deployed. SDK Embedded Server is generated when you generate your application in the lsps-embedded-server project and has your application hot-deployed.

(e) Open the browser and check that the To-Do List item is no longer available in the navigation menu.

5.1.5 Adding the To-Do Items Navigation to the Menu

Now add the navigation item to the custom to-do list:

1. Copy the fully qualified name of the module with the document to the clipboard: right-click the document definition and select *Copy Qualified Name*.



2. Add a *addDocumentItem* call with the document to the `createMenu()` of `AppMainMenu.java`: paste the qualified name of the document to prevent typos.

```
//navigation item to the todo item document:
   if (hasRightToOpenDocument("custom_todo_list_ui::todoItemsList")) {
     navigationMenu.addDocumentItem("Todo items", "custom_todo_list_ui::todoItemsList", null
   }
```

3. Restart the server in debug mode.

4. From the Modeling perspective, upload the `custom_todo_list_ui` module and run the `custom_todo_list_model` module to create some todo items.

5. Check the menu in the browser.

5.1.6 Localizing the Name of a Menu Item

The document menu item is now in the navigation menu but its label contains the `???` characters: these signalize that the system failed to find the localization string. Let's create the string:

1. Use a localization key as the titleKey parameter in the `addDocumentItem()` call: `navigationMenu.addDocumentItem("nav.todoitems", "custom_todo_list_ui::listOfTodoItems", null, FontAwesome.ADN, null, null);`
2. Open *localization.properties* with the default localizations in the `com.whitestein.lsps.vaadin.webapp` package (`<YOUR_APP>-vaadin` project).
3. Add the localization key:

```
# navigation menu items
nav.todoitems = To-do List
nav.documents = Documents
nav.runProcess = Run Model
# This is the new key:
nav.itemsList = Todo Items
```

4. Restart the server.

5.1.7 Excluding the Todo Items Document from Documents

Right now the document with the todo items is accessible not only from the dedicated navigation menu but it is also available as a document on the Documents page. To remove it, do the following:

1. Create and use your custom AppAppNavigator:
   (a) In the AppLspsUI.createNavigator() method, call your navigator:

```
@Override
protected void createNavigator(ViewDisplay display) {
  Navigator navigator = new AppAppNavigator(getUI(), display);
}
```

   (b) Create the AppAppNavigatorClass that extends the DefaultAppNavigator class:
      i. Create the inherited constructor:

```
public AppAppNavigator(UI ui, ViewDisplay display) {
   super(ui, display);
 }
```

      ii. Override `documentsViewClass()` to return your document view class:

```
@Override
  protected Class<? extends AppView> documentsViewClass() {
    return AppDocumentsView.class;
  }
```

2. Create your implementation of the `DocumentsView` class (copy the content of the class to AppDocumentsView and adjust the `load()` method so it excludes the todo list document:

```
//added property for excluded documents:
private static final String EXCLUDED_DOCUMENT = "custom_todo_list_ui::todoItemsList";
...
~
 private void load() {
  try {
    //Add documents; renamed original documents to allDocuments:
    List<DocumentInfo> documents = new ArrayList<>();
    List<DocumentInfo> allDocuments = genericDocumentService.getNonParametricDocuments();
```

```
~
      //checking in the list of all documents for the excluded documents:
      for (DocumentInfo document : allDocuments) {
        //Added this if to excludes the document from the table:
        if (!EXCLUDED_DOCUMENT.equals(document.getId())) {
          documents.add(document);
        }
      }
      this.documents.clear();
      this.documents.addAll(documents);
    } catch (ErrorException e) {
      getUI().getAppErrorHandler().showErrorMessage("app.unknownErrorOccurred", e);
    }
  }
```

3. Adapt the calculation of `documentBadge` on `documentMenuItem` (the blue icon with the number of available documents).
4. Override the getDocumentBadge() method so it returns your local *documentBadge* variable.

Simple example of badge calculation

```
//constant with the document name:
private static final String DOCUMENT_IN_MENU = "custom_todo_list_ui::todoItemsList";
~
  @Override
  protected void calculateBadges() {
    todoBadge = (int) todoService.getTodoCount(new TodoListCriteria());
    try {
       List<DocumentInfo> nonParametricDocuments = documentService.getNonParametricDocuments();
            //exclude of the document from the count:
            documentBadge = 0;
            for (DocumentInfo documentInfo : nonParametricDocuments) {
              if (!DOCUMENT_IN_MENU.equals(documentInfo.getId())) {
                documentBadge++;
              }
            }
    } catch (ErrorException e) {
      throw new RuntimeException(e);
    }
    runModelBadge = (int) modelManagementService.getExecutableModulesCount();
  }
  @Override
  protected int getDocumentBadge() {
    return documentBadge;
  }
```

5.2 Validating a Related Record

To validate related records of a record (cascade validation), do the following:

1. Define the constraints for the related record.
2. Define the constraint that will trigger the validation of the related records:
   - If the relationship is *to-one*:
     (a) Set the record property to the relevant property path, for example, `Book.author`.
     (b) Set the constraint type to `RecordValidity`.

- If the relationship is *to-many*:
  (a) Set the record property to the relevant property path, for example, `Author.books`.
  (b) Set the constraint type to `RecordCollectionValidity`.

      Important: Make sure the validation does not result in infinite recursive validation (the relationship record does not validate the parent record).

3. Call the `validate()` function on the main record.

Example: Underlying data model Constraints

`def Author author := new Author(books -> {new Book(title -> null)}, name -> null); validate (author, null, null, null);` Since the *Author.books.RecordCollectionValidity* constraint defines `RecordCollectionValidity`, the validate() checks also the created book constraints and violations on the constraints *Author.name.NotNull* and *Book.title.NotNull* are returned.

Chapter 6

Other Tutorials

- Model Update Examples
- Editable Decision Table

6.1 Model Update Examples

This series of simple tutorials demonstrate how to update the model of running model instances.

Mind that updating models can be very complex: consider using another approach such as agile processes pattern to avoid the need for model update altogether.

Generally, model update is performed in as follows:

1. Open the PDS and connect to the LSPS Server.
2. Import the original model into your workspace.
3. Import or create the target model into your workspace.
4. Define rules for the model update in the model-update definition.
5. Run the model update with the model-update definition.
6. Unload the modules that are no longer used.

> **Note:** If you are updating Java implementations as well (this is the case when updating to a newer standard library or to a custom LSPS Application with new `custom java implementations`, consider suspending the model instances that use the resources that are modified. Then you can redeploy the LSPS Application EAR. Note that if you want to run according to both, the original and target models, your implementations *must* be backward compatible (in such a case it is not necessary to suspend the pertinent model instances).

This section contains examples of simple model updates with the following modifications in the target models:

- variable value
- task parameter change
- event change
- data type change

6.1.1 Updating a Variable Value

Required action: Update a model instance so that a variable value changes to a value derived from its original value.

In the example update, we will introduce the following changes on global variables:

- A variable will be removed.
- A variable will have its value modified to a value derived from the removed variable.

1. *Design the source model* with a process definition and a variable definition:
   (a) Create a module with a global variable definition with the following variables:
      - `varSet` of type `Set<String>` with the initial value `{"old value 1", "old value 2"}`
      - `varString` of type String with the initial value `"42"`
   (b) Create a process definition with a None Start Event, a Conditional Intermediate Event, and a Simple End Event.
   (c) Set the Condition parameter of the Conditional Intermediate Event to `false` to keep the model instance running so it can be updated.

Figure 6.1 Process

2. *Design the target model:*
   (a) Copy and paste the old module.
   (b) Modify the global variables
      - Modify `varSet` to have the initial value `{"new value 1", "new value 2"}`
      - Delete `varString`
      - Create `varInt` of type Integer with the initial value `1`.
3. *Create the `.muc` file*:
   (a) Right-click the parent project, go to New > Model Update Configuration and follow the instructions.
   (b) Open the Variables page in the newly opened editor with your muc file
   (c) Adjust the mapping if necessary: Map the new variable definition file to the old variable definition file and `varInt` to `varString`. Mapping of `VarSet` should be recognized automatically after the variable definition file is mapped.
   (d) Define the transformation expressions on the new variables:
      - `varSet`: `{"transformation value"}`
      - `varInt`: `toInteger(toString(old("varString")))`

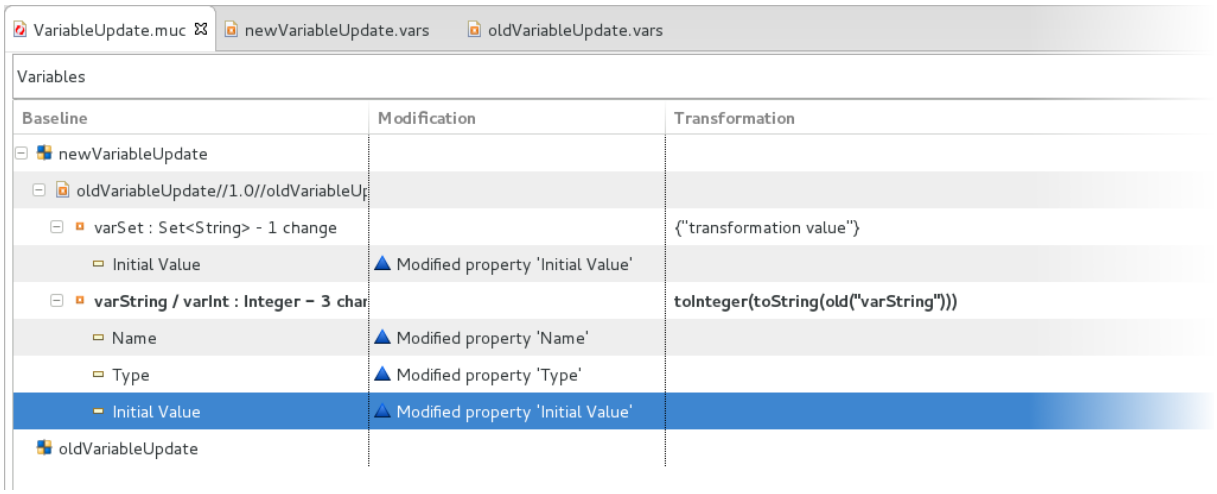


Figure 6.2 The muc file with variables and their transformation expressions

When you perform the model update, the system does the following:

1. First attempts to transform variable values according to their transformation expression.
2. If the expression does not exist, the system performs the transformation defined for the variable data type.
3. If neither the variable transformation expression nor the data type transformation exist, the variable is initialized. This typically applies to variables that were added in the new model.

   Note: When updating **local variables** of processes, sub-processes, and tasks, the update is determined by the update strategy of the parent element:

   - If the strategy of the parent element is continue, the parent context is preserved. The execution continues in the old transformed context: Its local variables are transformed as defined by their transformation expression.
   - If the strategy of the parent element is restart, the parent context is dropped and a new context is created: any local variables are discarded and new variables are initialized. The transformation expression on the variables is not applied.

To upload the resources and perform the model update, do the following:

1. Make sure your server with the Execution Engine, possibly the PDS or SDK Embedded Server, is running and your PDS is connected to it.
2. Upload the model to the server and create its model instance: In the GO-BPMN Explorer, right-click the source module and go to Run As > Model.
3. Upload the target model to the server: In the GO-BPMN Explorer right-click the module and go to Upload As > Model.
4. Switch to the Management perspective.
5. Refresh the Module Management and Model Instances view and check that both models are uploaded and the source model is instantiated.

6. In the Model Instances view, open the detail of the source-model instance and check the values of the global variables.

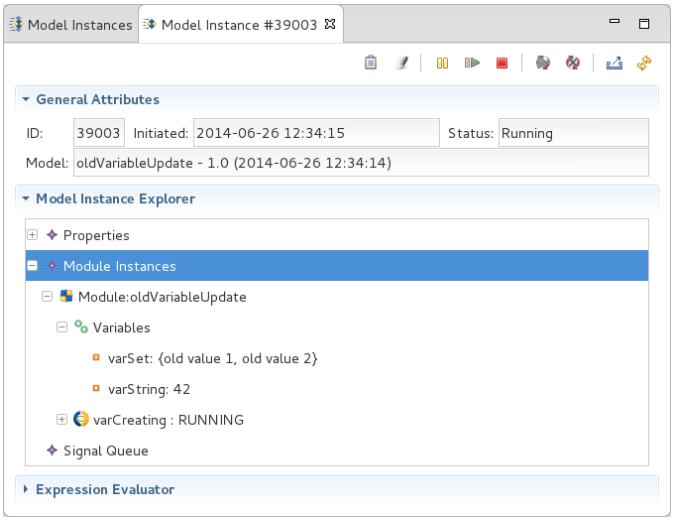


Figure 6.3 Detail of the old model with old global variables

7. Perform the update:

   (a) In the Model Instances view, click the Model Update ( ).

   (b) In the Model Update dialog window, provide the path to your muc file in the Configuration file field and click Next.

   (c) In the refreshed dialog, check that the model instance is listed and selected in the **Filtered Model Instances** section and click Next. Check the summary of the model update and click Finish.

   (d) Refresh the Model Instances view: The model instance should be in the Updated status.

   (e) Display the detail of the model instance.

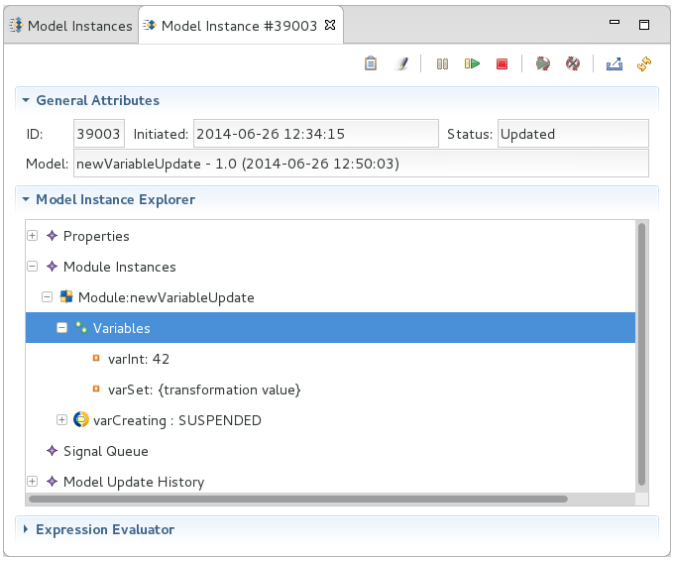


Figure 6.4 Detail View of the Updated Model Instance with New Global Variables

Note that the variables hold now the values defined by their transformation expression.

6.1.2 Updating a Task Parameter

Required action: Perform model update to a new model with modified task parameters and have a post process log the information about the update.

For this scenario, you should consider whether the task can be instantiated at the moment the model update is started or when the model instance is started after update:

- A task cannot be instantiated if it is **atomic** since it cannot be holding the token at that moment. Such tasks include the Log, Assign, Lock Task, etc. Changes on such tasks are considered as **a removal of the old task and adding of a new task**.
- A task can be instantiated when its task type requires **asynchronous or multi-step execution**, or waits for an event. These are tasks that can hold an execution token and become a `transaction border`. Such task types include the User, HttpCall, Web Service Client, and Server Tasks of the Standard Library and possibly custom tasks.

  For these tasks, you need to **define their transformation strategy** so that if such a task is running at the moment when you pause the process before the model update, or it will be running after the model instance starts after model update, the task is handled according to the transformation strategy. The strategy can be either restart or continue:

  - If the strategy is set to restart, the task ignores its old context and restarts as a new task.
  - If the strategy is set to continue, the task continues in its old context.

Let us update the Performers and Form parameter of a User Task. We will change the following:

- **Perfomers**

  `{anyPerformer()}` will be changed in the new model to `{getPerson("admin")}`

- **Underlying Form**

  The form content will be changed in the new model.

We will consider the outcome of both the restart and continue strategy.

Proceed as follows:

1. *Design the old model* as a module with a process and define its form definition with arbitrary content.

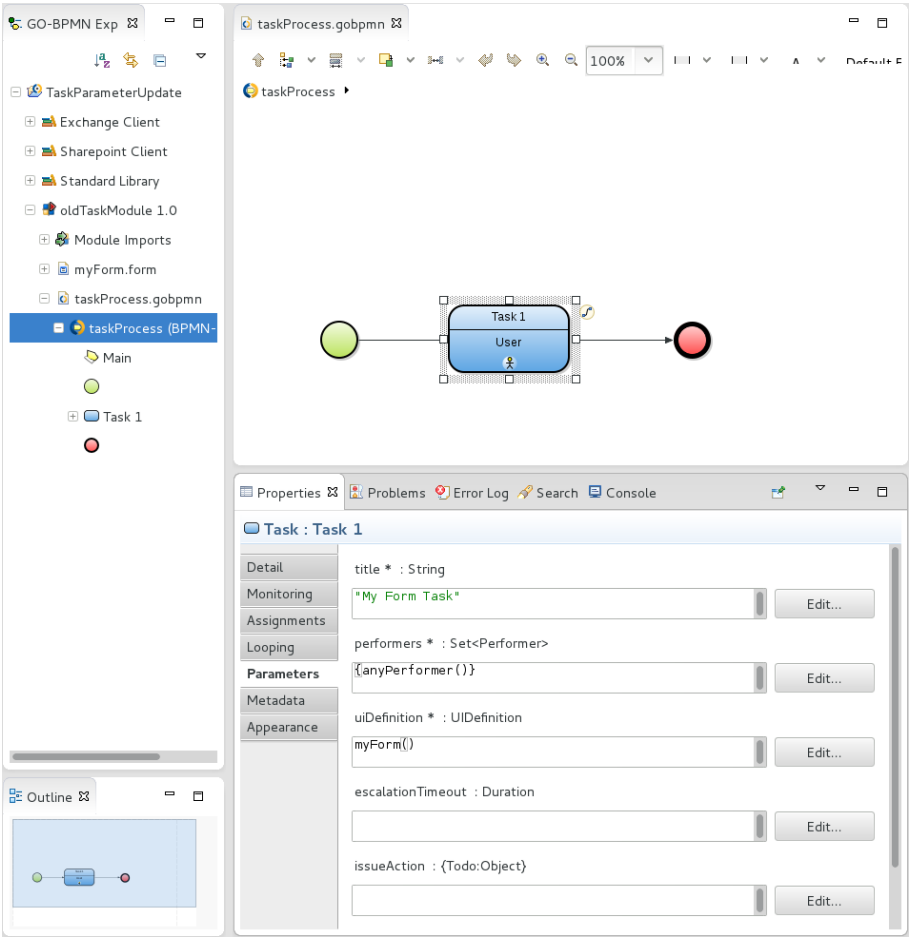


Figure 6.5 Old model

2. *Design the new model*: Copy and paste the old module and **modify the Performers parameter** of the User Task and modify the content of its **form** definition.

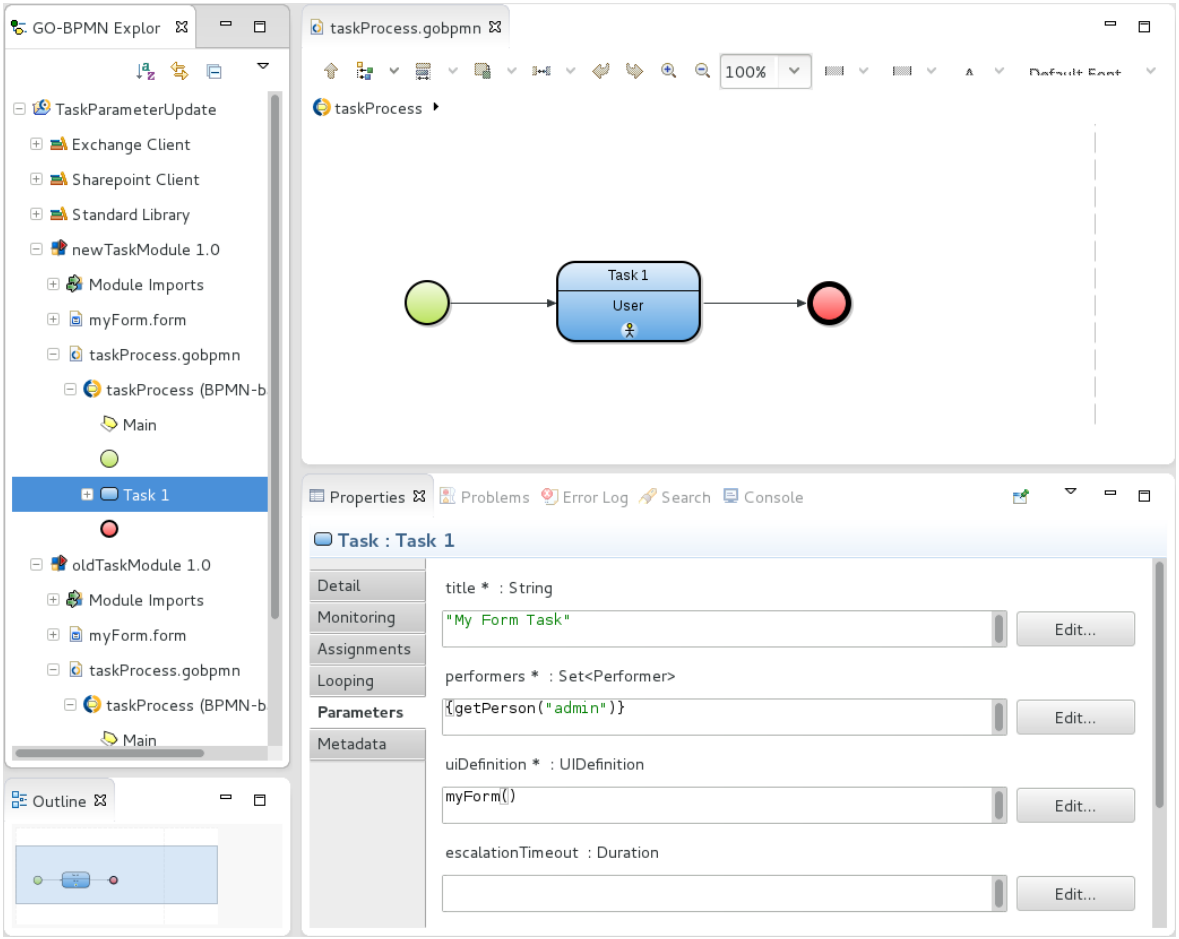


Figure 6.6 New model

3. *Create the* `.muc` *file*: Right-click the parent project, go to **New** > **Model Update Configuration** and follow the instructions.

4. Open the `.muc` file and on the Processes page locate the task parameter: The transformation strategy is set to Continue by default.

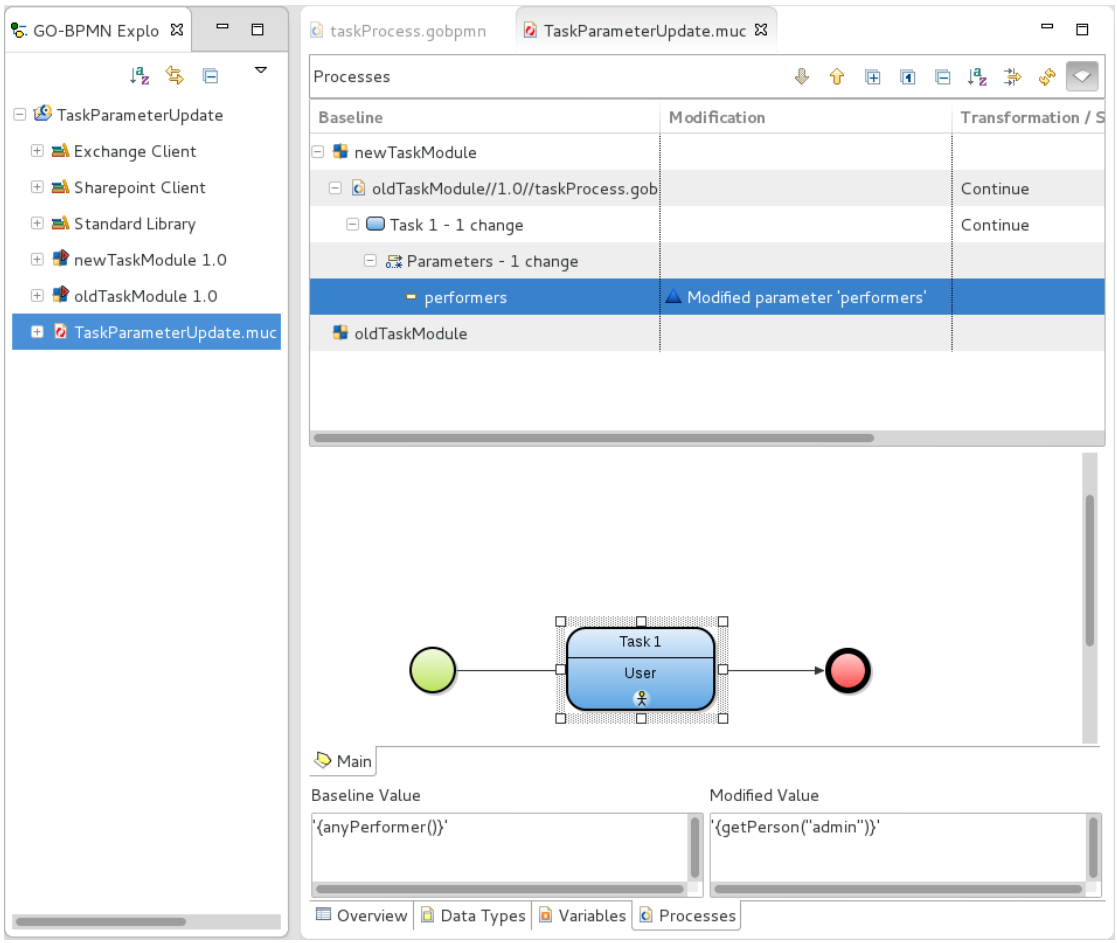


Figure 6.7 Model update configuration with the parameter change

Note that no changes on the form are detected since forms do not require any special handling on model update but are simply substituted with their new version.

5. Define a post process on the module that will log a message:

   (a) In the .muc file, right-click the new module and click Create Post-process.

      i. On the opened page, design the post process with a Log Task.

      ii. Define the message parameter of the Log Task.

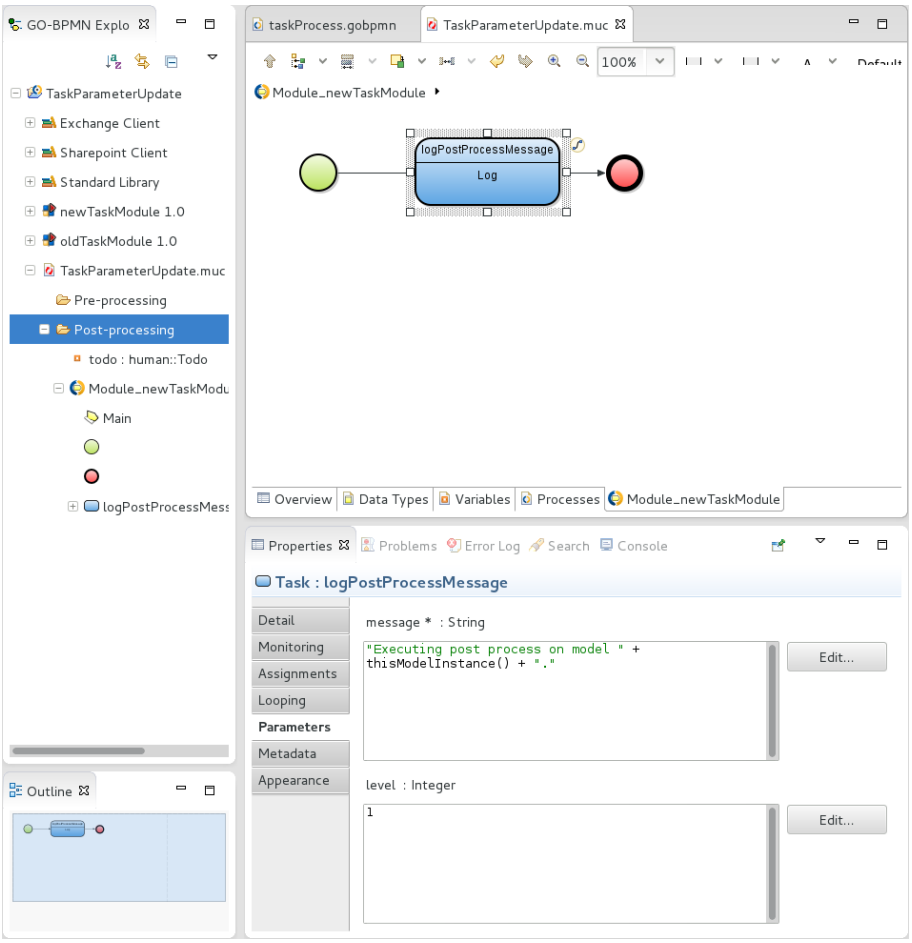


Figure 6.8 Post-Process

Make sure the transformation strategies on the module, process, and task of your muc file are set to Continue. This is the default transformation strategy.

To perform the model update, do the following:

1. Make sure your LSPS server, possibly the PDS or SDK Embedded Server, is running and your PDS is connected to it.

2. Upload the model to the server and create a model instance of the old model: In the GO-BPMN Explorer, right-click the old module and go to Run As > Model.

3. Upload the new model to the server: In the GO-BPMN Explorer right-click the new module and go to Upload As > Model.

4. Switch to the Management perspective.

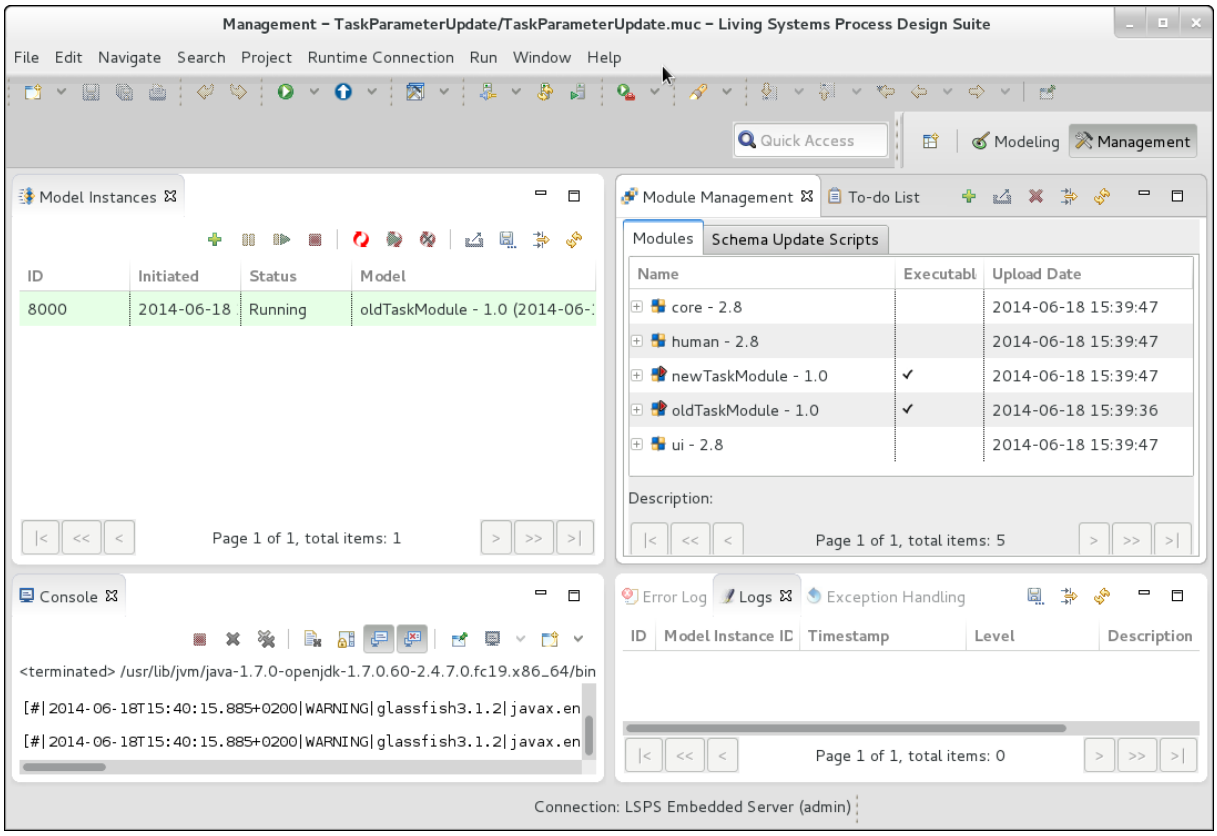


Figure 6.9 Management perspective with an instance of the old model

5. Refresh the Module Management and Model Instances view and check that both models are uploaded and the source model is instantiated.

6. Create a new user `guest` with all the security roles.

7. Go to the Application User Interface and lock the generated to-do:

   (a) Open a browser and go to `http://`<YOUR_SERVER_DOMAIN>/lsps-application/

   (b) Log in as the user guest.

   (c) Click TO-DO LIST.

   (d) Open the to-do, which was generated by the old model instance.

   (e) With the to-do content displayed, log out, so the guest user locks the to-do.

8. Back in the Management perspective, perform the update:

   (a) In the Model Instances view, click the Model Update ( ).

   (b) In the Model Update dialog window, provide the path to your muc file in the Configuration file field and click Next.

   (c) In the refreshed dialog, check that the model instance is listed and selected in the **Filtered Model Instances** section and click Next. Check the summary of the model update and click Finish.

   (d) Refresh the Model Instances view: The model instance should be in the Updated status. If the model instance is in the Pre-processes state, hit the Refresh button again.

   If the model instance is in the Pre-processes state, hit the Refresh button again. The model instance is still suspended: If you check the to-do list of the guest user, the to-do is not available since the user task is suspended.

(e) Check the Log view for the log message of the post-process.

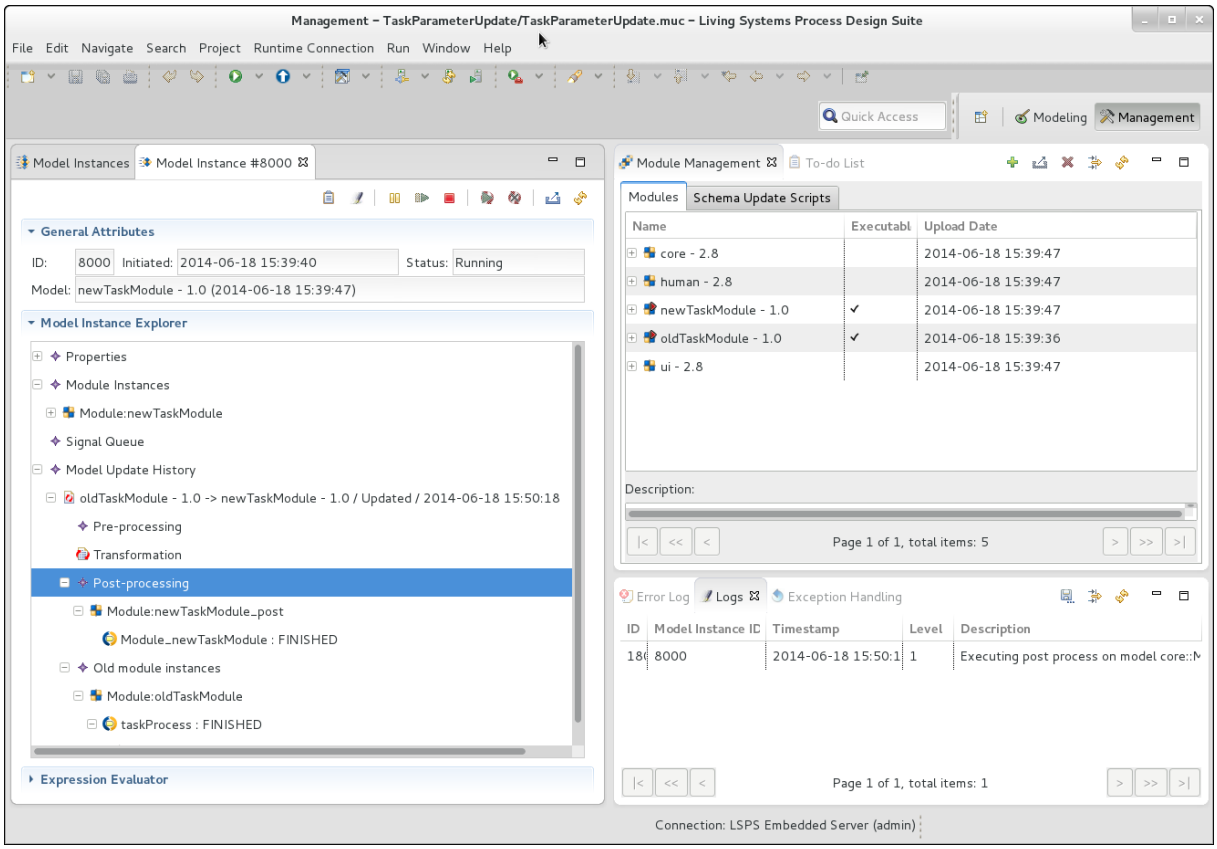


Figure 6.10 Log view with the post-process log message

(f) Select the model instance and click Resume ( ) button. The model instance becomes Running.

9. Go to the Application User Interface as the guest user. The to-do list of the guest user still contains the locked to-do in spite of the fact that the new model allows only the admin user as the to-do performer. However, its content already follows the form of the new model.

Set the transformation strategy on the `User Task` of your muc file to `Restart`. Leave the strategy on the parent process and module set to Continue and perform the model update anew. The to-do will be discarded.

> **Note:** If you set the strategy on the parent process and module to Restart, the entire process/module will be discarded on update and a new process/module will be instantiated.

6.1.3 Updating an Event Type

Required action: Update a model instance so that its None Start Event is changed to a Conditional Start Event and Timer Intermediate Event changes in a Conditional Intermediate Event.

A change of an event type does not allow to define any pre- or post-processing on the event, or a transformation expression since the change is detected as a removal of the old event and addition of the new event. If required, define model-update processes on the parent modules and process.

1. *Design the old model* with a process definition:

(a) Create a module with the old process.

(b) Create a process definition with a None Start Event, a Conditional Flow Event, and a Simple End Event.

(c) Set the Delay parameter of the Timer Intermediate Event, for example, to `new Duration(years -> 1)`.

2. *Design the new model:* copy and paste the old module, change the None Start Event to a Conditional Start Event and the Timer Intermediate Event to Conditional Intermediate Event, and set their condition, for example, to false.

3. *Create the .muc file:*

(a) Right-click the parent project, go to New > Model Update Configuration and follow the wizard.

(b) Open the `.muc` file and on the Process page and check the element mapping.

The change mapping might be incorrect as shown in Model Update Configuration with Incorrect Mapping: The `newUpdateEventType` Process is recognized as a new Process, while we want it to be mapped to the `oldUpdateEventType` Process.

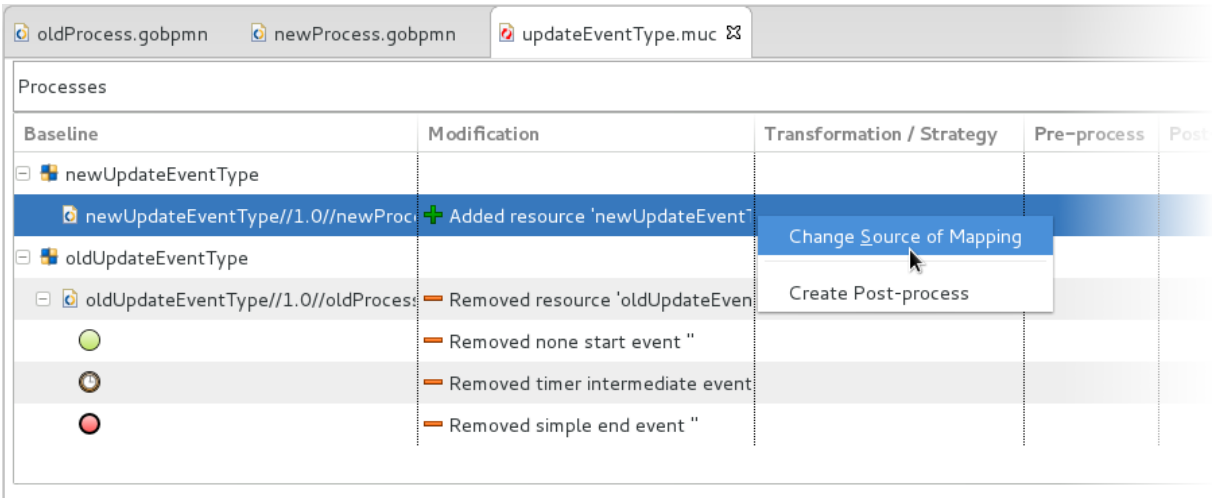


Figure 6.11 Model Update Configuration with Incorrect Mapping

(c) Rigth-click the element and adjust the mapping if needed.

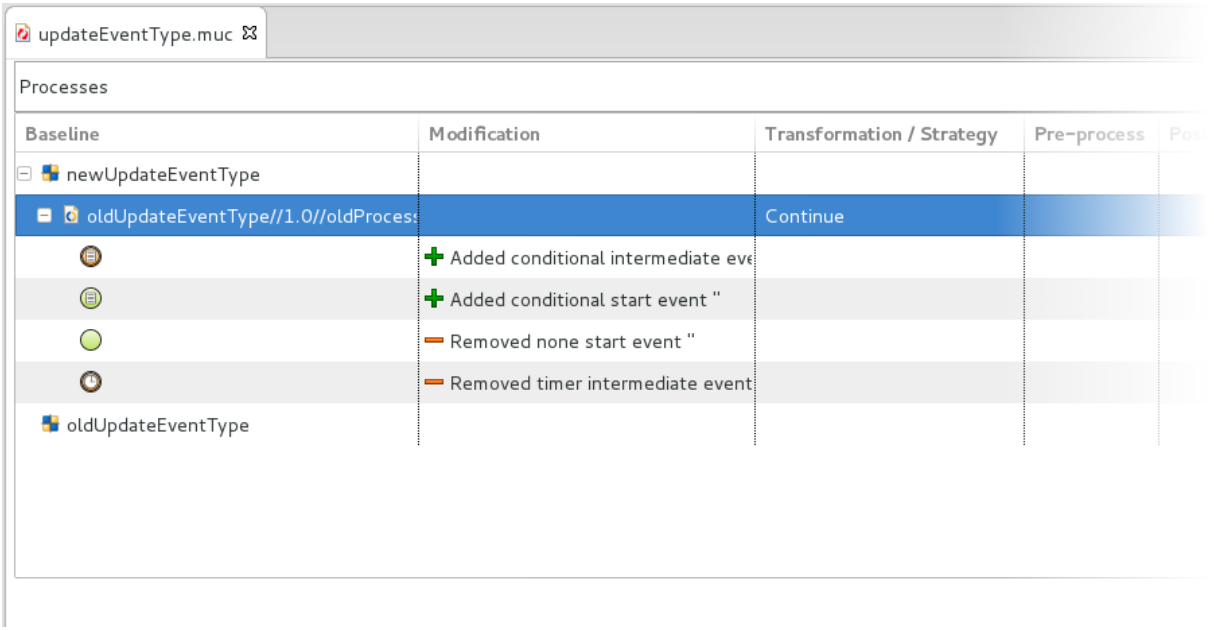


Figure 6.12 Model Update Configuration with Corrected Mapping

The transformation strategies on the Process is set to *Continue*. This is the default transformation strategy. If we used the *Restart* strategy, the process would be restarted on update if on the element at the given moment.

To perform the model update, do the following:

1. Make sure your server with the Execution Engine, possibly on the PDS or SDK Embedded Server, is running and your PDS is connected to it.
2. Upload the model to the server and create a model instance of the old model: In the GO-BPMN Explorer, right-click the old module and go to Run As > Model.
3. Upload the new model to the server: In the GO-BPMN Explorer right-click the new module and go to Upload As > Model.
4. Switch to the Management perspective.
5. Refresh the Module Management and Model Instances view and check that both models are uploaded and the source model is instantiated.
6. Switch to the Management perspective.
7. Refresh the Module Management and Model Instances view and check that the old model is instantiated and the new model uploaded.
8. In the Model Instances view, open the detail of your old model and check the execution diagram of the process.

Figure 6.13 Execution Diagram of the Old Model Instance

9. Perform the update:

   (a) In the Model Instances view, click the Model Update ( ) button.

   (b) In the Model Update dialog window, provide the path to your muc file in the Configuration file field and click Next.

   (c) In the refreshed dialog, do not apply any filtering, just click Next so that any available instances of the old model are updated (in this case, exactly one model instance is running).

   (d) Check the summary of the model update and click Finish.

   (e) Refresh the Model Instances view: The model instance should be in the Updated status.

   (f) Click the Continue ( ) button to trigger the execution of the updated model instance.

10. Check the execution diagram of the updated model instance.

Figure 6.14 Execution Diagram of the Updated Model Instance

Note that the execution remains on the new Conditional Event.

Now set the transformation strategies on the process to `Restart` so the Process is restarted on update if on the event. Perform model update as described above: The execution remains on the new Conditional Start Event since the process instance was restarted.

6.1.4 Updating a Data Type

Required action: Update a model instance with a changed data type of a record property.

> **Important:** It is not recommended to update shared records via model update since changes on shared records are reflected on the database. While it is safe to add a new field to a shared record and remove not-nullable fields using model update, modifications to fields, such as modification of their data types, might result in corrupted database schema or data loss. It is recommended to migrate the database directly, not via update of shared records.

When you are updating a model instance to a model with changed non-shared record types, all record instances will be updated according to the transformation expression.

We will update a model instance's data hierarchy as follows:

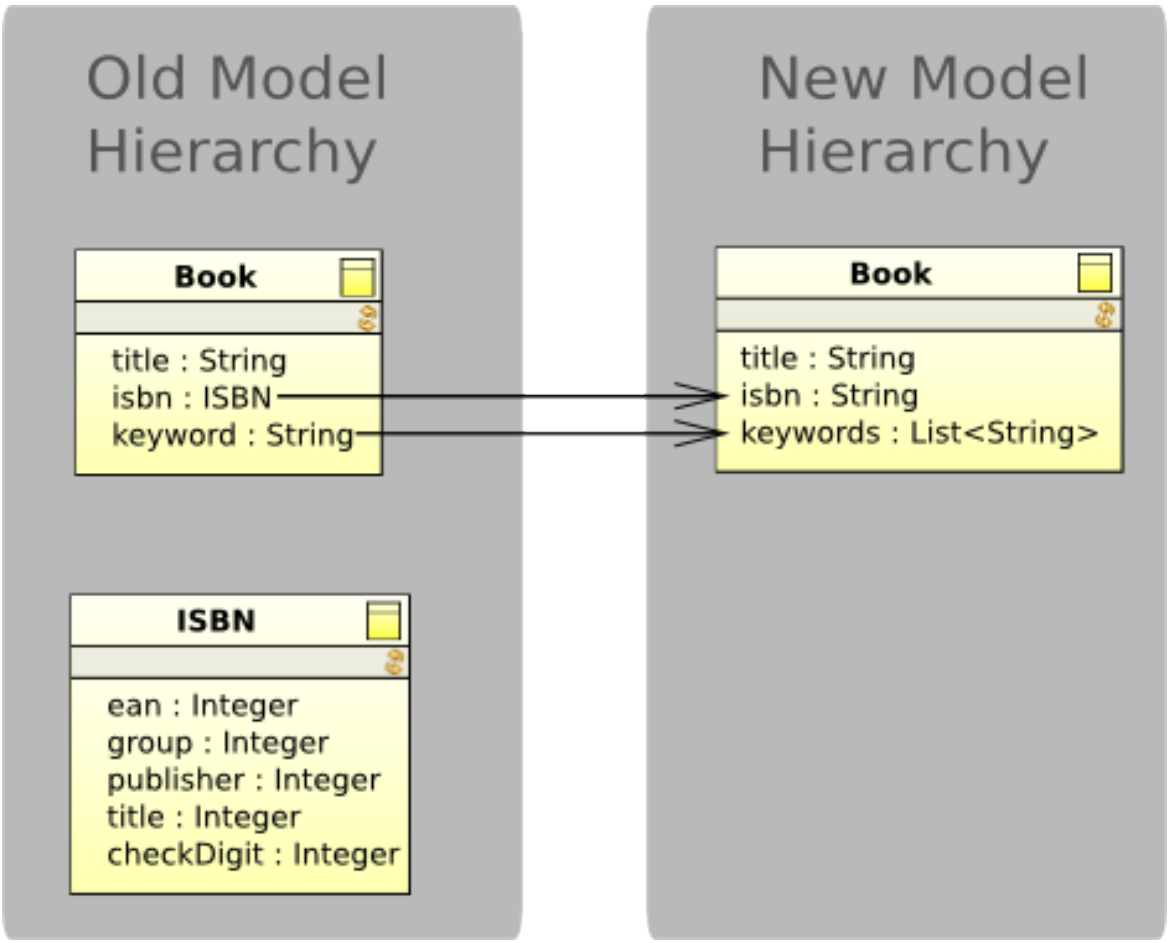


Figure 6.15 Old and new data type models

The data type update will involve the following changes:

- The ISBN record is removed: No further actions are required.
- The Book.isbn field is changed from the ISBN type to String: The new isbn field must concatenate and format the old instance of ISBN for the given Book instance.
- The keyword field is changed from a string to a list of strings and renamed to keywords: The new keywords field should import the old keyword string.

1. *Design the old model* with a process definition, variable definition, and data type definition as shown in Old model for data type update:
   (a) Create a module with the old data type hierarchy with the `Book` and `ISBN` records.
   (b) Create a global variable definition with a bookSet variable of type `Set<Book>` and a book variable of type `Book`.
   (c) Create a process definition with a None Start Event, a Conditional Flow Event, and a Simple End Event.
   (d) On the flow from the None Start Event define an assignment expression that creates three Book instances assigned to the bookSet global variable:

```
bookSet := {
  new Book(title -> "Brave New World",
           isbn -> new ISBN(ean -> 978, group -> 1, publisher -> 85399, title -> 393, che
           keyword -> "science fiction"),
```

```
        new Book(title -> "Catch-22",
                 isbn -> new ISBN(ean -> 978, group -> 1, publisher -> 4055, title -> 387, chec
                 keyword -> "army"),
        book := new Book(title -> "Brave New World",
                         isbn -> new ISBN(ean -> 978, group -> 1, publisher -> 85399, title ->
                         keyword -> "science fiction")
    }
```

(e) Set the Condition parameter on the Conditional Flow Event to `false`. The Conditional Flow Event will hold the execution so that the process creates the record instances and then remains running.

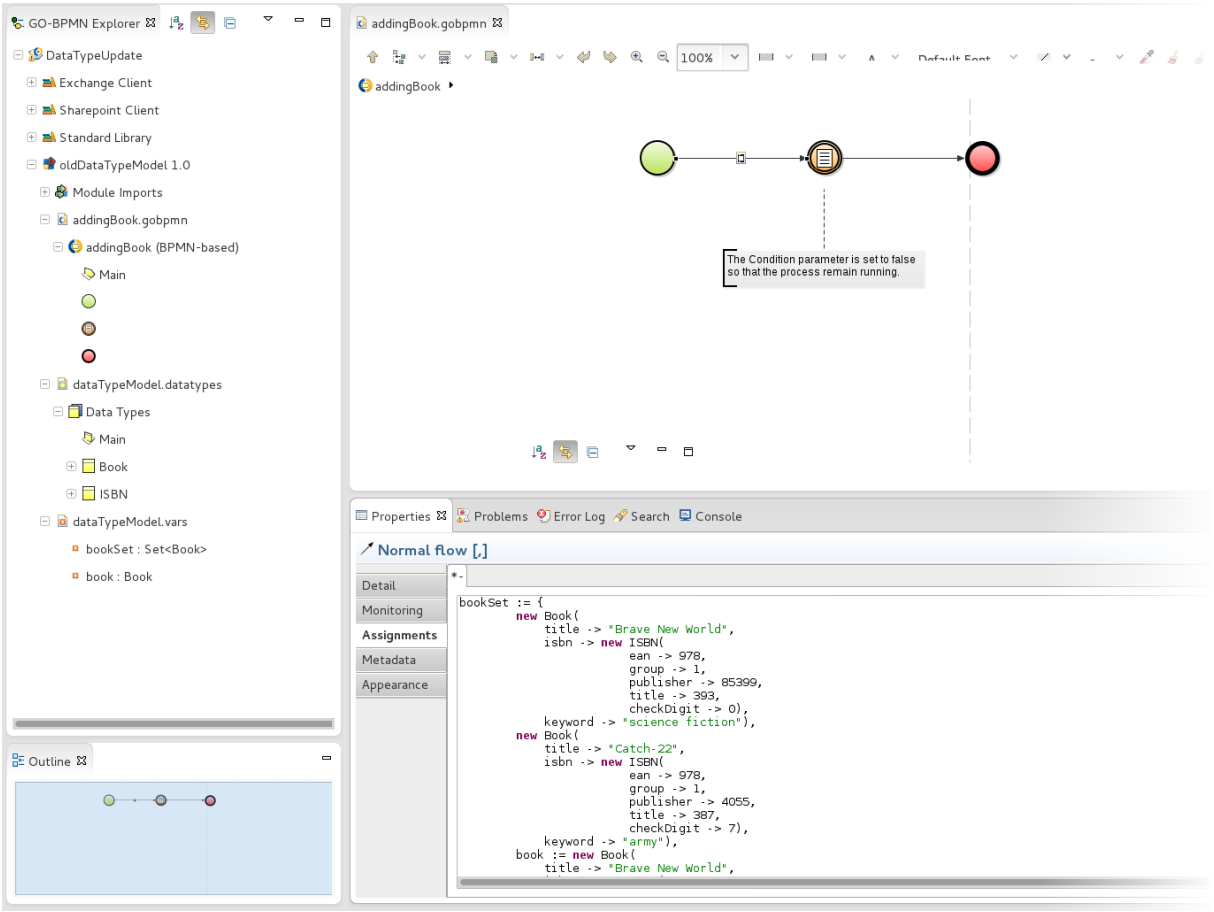


Figure 6.16 Old model

2. *Design the new model:* copy and paste the old model, modify the data type model, and remove the assignment expression on the flow.

3. *Create the .muc file*:

   (a) Right-click the parent project, go to New > Model Update Configuration and follow the wizard.

   (b) Open the .muc file and on the Data Types page and define the transformation expressions for the isbn record field and the new keywords field:

   - isbn:

     ```
     toString(old("isbn.ean")) + "-" +
     old("isbn.group") + "-" +
     old("isbn.publisher")  + "-" +
     old("isbn.title")  + "-" +
     old("isbn.checkDigit")
     ```

     This expression will take individual fields from the old record and concatenate them into a new hyphenated isbn value.

- keywords: `[old("keyword")]` This expression will take the keyword string and add it to a new list of keywords.

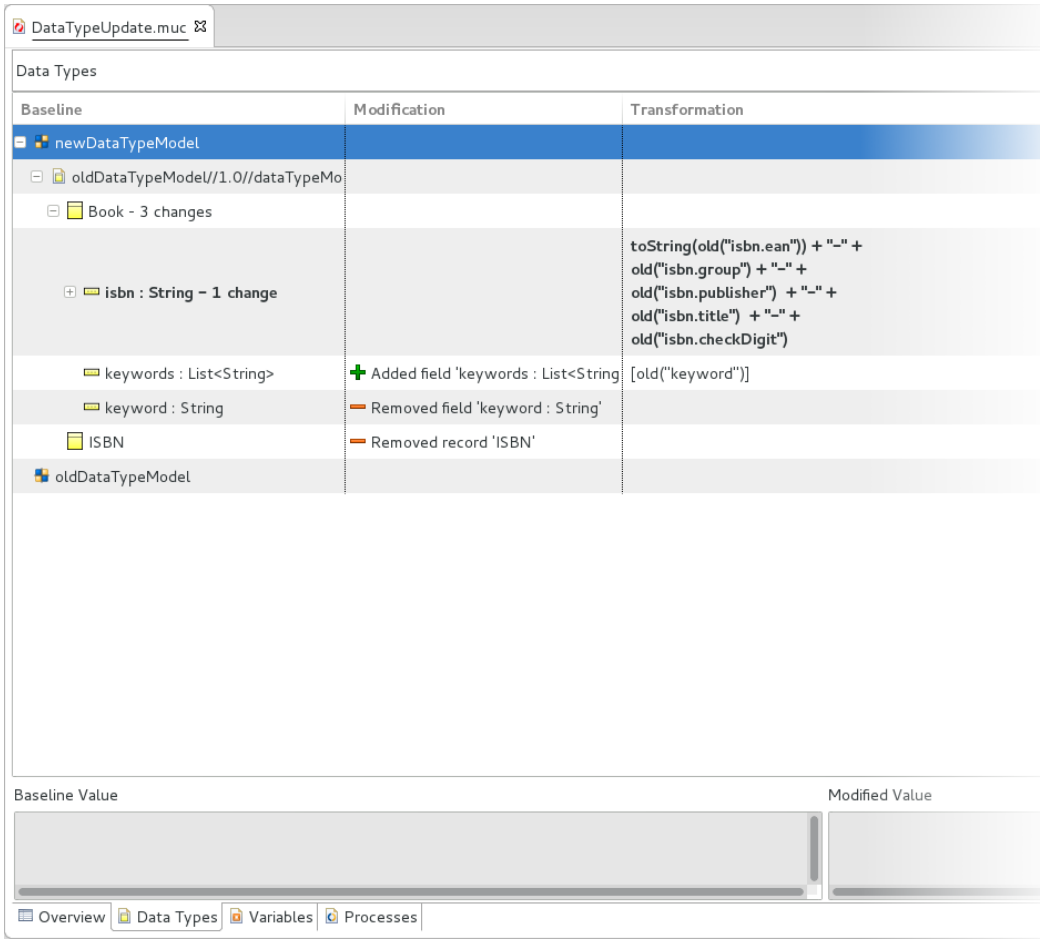


Figure 6.17 Model Update Configuration with the Data Type Changes

To perform the model update, do the following:

4. Make sure your server with the Execution Engine, possibly on the PDS or SDK Embedded Server, is running and your PDS is connected to it.

5. Upload the model to the server and create a model instance of the old model: In the GO-BPMN Explorer, right-click the old module and go to Run As > Model.

6. Upload the new model to the server: In the GO-BPMN Explorer right-click the new module and go to Upload As > Model.

7. Switch to the Management perspective.

8. Refresh the Module Management and Model Instances view and check that both models are uploaded and the source model is instantiated.

9. In the Model Instances view, open the detail of your old model and check the execution diagram of the process.

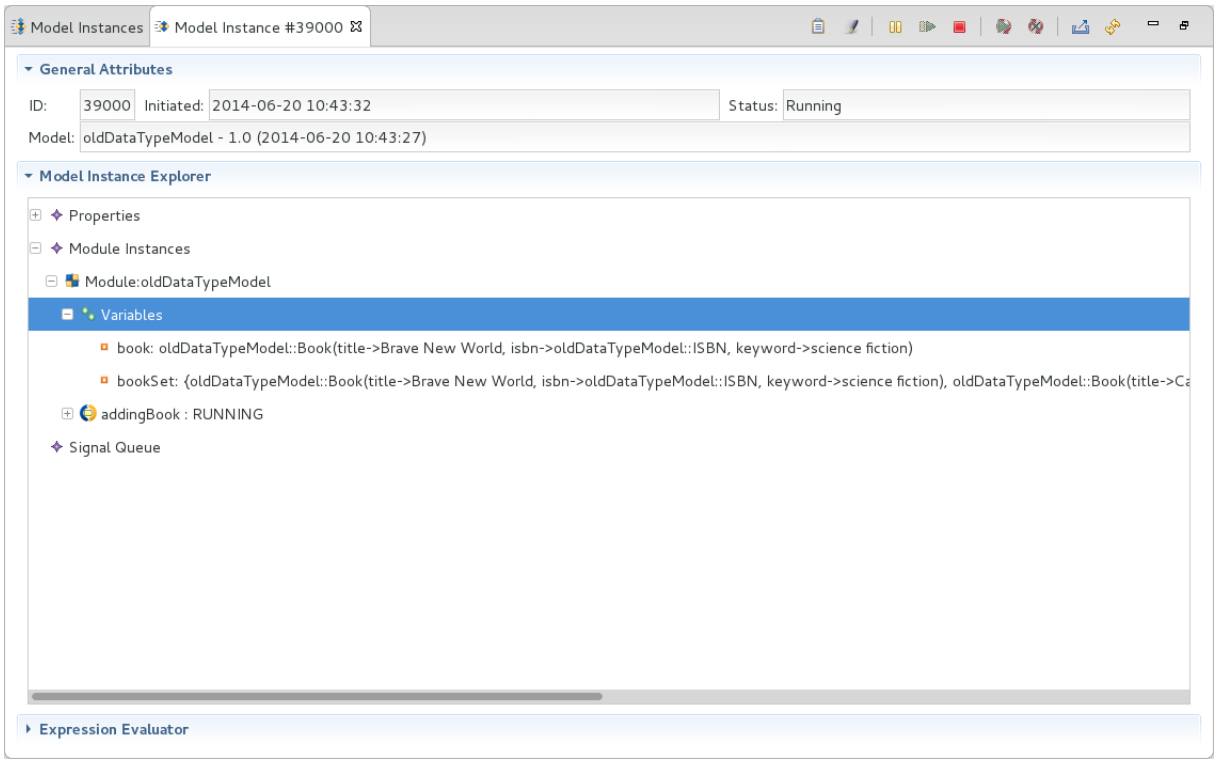


Figure 6.18 Detail of the Old Model with Old Global Variables

10. Perform the update:

    (a) In the Model Instances view, click the Model Update ( ) button.

    (b) In the Model Update dialog window, provide the path to your muc file in the Configuration file field and click Next.

    (c) In the refreshed dialog, do not apply any filtering, just click Next so that any available instances of the old model are updated (in this case, exactly one model instance is running).

    (d) Check the summary of the model update and click Finish.

    (e) Refresh the Model Instances view: The model instance should be in the Updated status.

    (f) Select the model instance and click Resume ( ). The model instance becomes Running.

11. Check the Log view for the log message of the post-process.

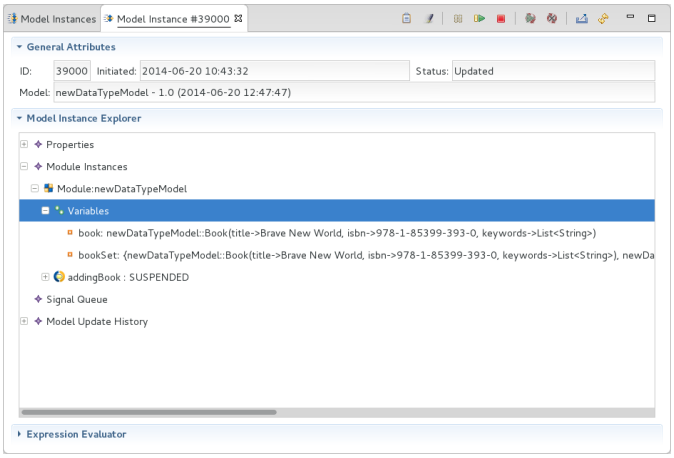


Figure 6.19 Detail View of the Updated Model Instance with New Global Variables

The variables hold values of the new data types: the record values were transformed according to the transformation expression defined for the data types.

6.2 Editable Decision Table

Required result: Decision table for resolution of employee position level based on their years of experience that is persisted and can be edited from the Process Application.

You can download an example implementation here. To import it to your workspace, create a GO-BPMN project; right-click the project and go to Import > Archive file; select only the `positionLevelResolver` directory, which is a module, and click **Finish**.

1. Import the `dmn` module into your module: In the *GO-BPMN Explorer*, right-click your module, click **Module Imports**, click **Add** and double-click dmn.
2. Design the decision table:
   (a) Create the decision table definition: right-click the module, go to **New** -> **Decision Table Definition**, enter the file name and click **Finish**.
   (b) Keep the default FIRST hit policy and LSPS as the expression language for the business rules.

   Note: While you can use SFEEL as the expression language for the business rules, it restricts the options, such as, using the resources of the Standard Library and accessing the contexts.
   (c) Define the input parameter with the number of years of experience:
      i. Rename the default `input` parameter to `yearsOfExperience`.
      ii. Set its data type to Integer.
   (d) Define the output with position level:
      i. Rename the default `output` parameter to `level`.
      ii. Set its data type to String.
      iii. Define its options as Strings of the possible level, such as, `"Associate Engineer"`, `"Engineer"`, `"Senior Engineer"`, `"Principal Engineer"`.
      iv. Add rules for different *yearsOfExperience* input: click **+** below the table and define the age ranges as expressions.

Now you need to create a form, and a document with the form so you can access it from the Process Application:

1. Create a Form with the Decision Table component:
   (a) Create a form definition: right-click the module, go to **New** -> **Form Definition**, enter the name `PositionDecisionTableForm` and click **Finish**.
   (b) Create local variable **positionLevelDecisionTableVar** of the type of your decision table: it will hold the decision table and initialize it to the stored decision table if such a table exists.
   (c) Initialize the **positionLevelDecisionTableVar**, for example, in the form constructor:

```
public PositionDecisionTableForm(){
  positionLevelDecisionTableVar := new positionLevelResolver(false);
  //if cannot load the decision table, create a new one and persist it:
  if positionLevelDecisionTableVar.load("storedPositionTable") == false then
     positionLevelDecisionTableVar := new positionLevelResolver(true);
     positionLevelDecisionTableVar.save("storedPositionTable")
  end
}
```

   (d) Design the form:
      i. Insert a **Vertical Layout** component.

ii. Insert a **Decision Table Editor** component into the layout and set its properties as follows:
   - Decision table: `positionLevelDecisionTableVar`
   - Rights: `[DecisionTableRights.CAN_EVERYTHING]`

iii. Insert a **Button** that will persist the changes on the decision table: this is performed by calling the `save(<ID>)` method on the table. The click listener closure could be set to something like `{ e -> positionLevelDecisionTableVar.save("storedPositionTable") }` Make sure you are saving the table under the same String ID as you are using when loading it in the form constructor.

(e) Create a document with the form as its UI definition.

2. Create the `PositionEvaluator` form definition for the years-of-experience input and evaluation with the following:

   - Local variable *positionLevelDecisionTable* of the type of your decision table.
     Initialize the variable from the constructor of the form to the persisted decision table, for example:

```
public PositionEvaluator(){

positionLevelDecisionTable := new positionLevelResolver(false);

  if (positionLevelDecisionTable.load("storedPositionTable") == false) then
    positionLevelDecisionTable := new positionLevelResolver(true);
  end;
}
```

   - A Decimal Field component for input of years of experience with the ID `yoe` and the binding set to a valid value
   - A Label component with the ID `result` that will display the position level
   - Button to request the level evaluation on click and set it as the value of the result field

```
//example click listener that returns the decision,
//writes it to the result field:
{ e ->
if (yoe.getValue()== null) then
   result.setValue("Enter an integer.")
else
   result.setValue(positionLevelDecisionTable.evaluate(yoe.getValue() as Integer))
end
}
```

   (a) Create a document with the UI definition set to `new PositionEvaluator()`

3. Run a LSPS Embedded Server and upload the module.

4. Test the documents:

   (a) Log in to the Process Application in your browser.

   (b) In the document with the position evaluator, enter the years-of-experience data and get the resulting level.

   (c) Now open the document with the decision table, and edit the values and add a new rule.

   (d) Open the evaluator document again and enter the years-of-experience data and get the updated resulting level.

Chapter 7

LSPS Application on a Local Server and Database

In this tutorial, you set up a MySQL database with LSPS tables, set up the WildFly server, deploy the LSPS Application to the WildFly server, and connect to the server from PDS. We assume you are on Linux.

> **Important:** This environment is not intended for production. For simplicity, resources are set up in the home directory and no security aspects are taken into consideration. More detailed deploy instructions are available in the `Deployment Guide`.

You will need the following:

- MySQL 8.0
- JDBC driver for MySQL
- WildFly 16
- lsps-runtime (requires licensed)

7.1 Setting up Local MySQL Database

1. Install MySQL: make sure to perform this as the administrator on Windows.
2. Log in as root user:

   ```
   mysql -u root -p
   ```

3. Create LSPS database and user:

   ```
   CREATE USER 'lsps' IDENTIFIED BY 'lsps';
   CREATE DATABASE lsps DEFAULT CHARACTER SET utf8mb4 DEFAULT COLLATE utf8mb4_unicode_ci;
   ALTER USER 'lsps'@'localhost' IDENTIFIED BY 'lsps'
   GRANT XA_RECOVER_ADMIN ON *.* TO 'lsps'@'%';
   SET GLOBAL log_bin_trust_function_creators = 1;
   quit
   ```

4. In the `[mysqld]` section of the mysqld.conf file, add `max_allowed_packet=512M`. On Windows, define this in your `C:\ProgramData\MySQL\MySQL Server 8.0\my.ini` in the mysql installation directory.

```
 $ cat /etc/mysql/mysql.conf.d/mysqld.cnf |grep -E "(max_allowed|default_time_zone)"
max_allowed_packet= 512M
default_time_zone=+01:00
```

5. Initialize the database with the `lsps-db-migration-lsps` tool from `lsps-runtime/cli-tools`.

```
java -jar lsps-db-migration-lsps-<VERSION>-full.jar --databaseUrl jdbc:mysql://localhost/lsps \
--user lsps --password lsps
```

Initialized Database

```
mysql> USE lsps;
Database changed
mysql> SHOW TABLES;
+--------------------------------+
| Tables_in_lsps                 |
+--------------------------------+
| LSPS_ACTIVE_USERS_TRACK        |
| LSPS_BINARY_DATA               |
| LSPS_BINARY_DATA_METADATAS     |
| LSPS_DASHBOARD_TABS            |
...
```

7.2 Setting up Local WildFly

1. Set up JAVA_HOME and add JAVA_HOME/bin to PATH.

```
export JAVA_HOME=/usr/lib/jvm/java-8-oracle
export PATH=$JAVA_HOME/bin:$PATH
```

2. Download WildFly and extract it.

```
~$ #we use home directory; on linux consider /opt
~$ unzip Downloads/wildfly-16.0.0.Final.zip
~$ mv wildfly-16.0.0.Final/ wildfly
```

3. Set up data source for the MySQL database:

   (a) Download and add the MySQL JDBC driver:

   ```
   ~$ mkdir -p wildfly/modules/com/mysql/main
   ~$ cp Downloads/mysql-connector-java-\<VERSION>.jar wildfly/modules/com/mysql/main/mysql-connector-java.jar
   ```

   (b) Configure the driver in *wildfly/modules/com/mysql/main/module.xml*.

   ```
   ~$ cat wildfly/modules/com/mysql/main/module.xml
   <module xmlns="urn:jboss:module:1.0" name="com.mysql">
     <resources>
        <resource-root path="mysql-connector-java.jar"/>
     </resources>
     <dependencies>
       <module name="javax.api"/>
       <module name="javax.transaction.api"/>
     </dependencies>
   </module>
   ```

   (c) Add the authentication jar:

   ```
   ~$ mkdir -p wildfly/modules/com/whitestein/lsps/security/main
   ~$ cp ~/Downloads/lsps-runtime/lsps-security-jboss-<VERSION>.jar
         wildfly/modules/com/whitestein/lsps/security/main/lsps-security-jboss.jar
   ```

   (d) Configure the authentication module:

```
cat wildfly/modules/com/whitestein/lsps/security/main/module.xml
<module xmlns="urn:jboss:module:1.0" name="com.whitestein.lsps.security">
  <resources>
    <resource-root path="lsps-security-jboss.jar"/>
  </resources>
  <dependencies>
    <module name="javax.api"/>
    <module name="javax.transaction.api"/>
    <module name="org.picketbox" />
  </dependencies>
</module>
```

4. Create the admin user for WildFly:

```
~$ ./wildfly/bin/add-user.sh -u admin -p admin
```

5. Set up profile configuration in `wildfly/standalone/configuration/standalone-full.xml` (The file configured as instructed below is available here:

   - Add LSPS_DS transaction datasource that connects to your lsps database with the driver:

```
<datasources>
<!-- ADDED: -->
<xa-datasource jndi-name="java:/jdbc/LSPS_DS" pool-name="LSPS_DS" enabled="true" use-java
     <driver>mysqlxa</driver>
     <xa-datasource-property name="URL">jdbc:mysql://localhost:3306/lsps?useUnicode=true&
     <security>
        <user-name>lsps</user-name>
        <password>lsps</password>
     </security>
     <transaction-isolation>TRANSACTION_READ_COMMITTED</transaction-isolation>
        <xa-pool>
          <min-pool-size>10</min-pool-size>
          <max-pool-size>20</max-pool-size>
          <prefill>true</prefill>
        </xa-pool>
      </xa-datasource>
      <drivers>
         <driver name="mysqlxa" module="com.mysql">
         <xa-datasource-class>com.mysql.cj.jdbc.MysqlXADataSource</xa-datasource-class>
      </driver>
    </drivers>
</xa-datasource>
```

   - Set the datasource of the default bindings of urn:jboss:domain:ee:4.0 to LSPS_DS

```
<default-bindings context-service="java:jboss/ee/concurrency/context/default" datasource=
```

   - Set up the security lspsRealm.

```
<subsystem xmlns="urn:jboss:domain:security:2.0">
    <security-domains>
        <security-domain name="lspsRealm" cache-type="default">
             <authentication>
                 <login-module code="com.whitestein.lsps.security.jboss.LSPSRealm" flag="
                     <module-option name="dsJndiName" value="/jdbc/LSPS_DS"/>
                 </login-module>
             </authentication>
         </security-domain>
```

   - Prolong the locking isolation on the web cache container:

```
    <cache-container name="web" default-cache="passivation" module="org.wildfly.clusterin
       <local-cache name="passivation">
       <locking isolation="REPEATABLE_READ" acquire-timeout="600000"/>
```

   - Add mail session LSPS_MAIL:

```
<mail-session name="lspsmail" jndi-name="java:jboss/mail/LSPS_MAIL">
    <smtp-server outbound-socket-binding-ref="mail-smtp"/>
</mail-session>
```

- Configure JMS:
  - Enable persistence on jms <subsystem xmlns="urn:jboss:domain:messaging-activemq↩:6.0">:

```
<subsystem xmlns="urn:jboss:domain:messaging-activemq:6.0">
<server name="default" persistence-enabled="true">
```

  - Add queue and topic:

```
<jms-queue name="LSPS_QUEUE" entries="java:jboss/jms/LSPS_QUEUE"/>
<jms-topic name="LSPS_TOPIC" entries="java:jboss/jms/LSPS_TOPIC"/>
```

6. Adjust JAVA_OPTS:

- On Linux, in *wildfly/bin/standalone.conf*:

```
~$ cat wildfly/bin/standalone.conf
if [ "x$JBOSS_MODULES_SYSTEM_PKGS" = "x" ]; then
   JBOSS_MODULES_SYSTEM_PKGS="org.jboss.byteman"
fi
if [ "x$JAVA_OPTS" = "x" ]; then
    JAVA_OPTS="-Xms64m -Xmx800M -XX:MetaspaceSize=96M -XX:MaxMetaspaceSize=512m -Djava.ne
    JAVA_OPTS="$JAVA_OPTS -Djboss.server.default.config=standalone-full.xml"
    JAVA_OPTS="$JAVA_OPTS -Dorg.eclipse.emf.ecore.EPackage.Registry.INSTANCE=org.eclipse.
    JAVA_OPTS="$JAVA_OPTS -Dorg.apache.el.parser.COERCE_TO_ZERO=false"
    JAVA_OPTS="$JAVA_OPTS -Dcom.whitestein.lsps.vaadin.ui.debug=true"
else
   echo "JAVA_OPTS already set in environment; overriding default settings with values: $
fi
```

- On Windows, add at the end of *wildfly/bin/standalone.conf.bat*:

```
set "JAVA_OPTS=-Xms64m -Xmx800M -XX:MetaspaceSize=96M -XX:MaxMetaspaceSize=512m -Djava.ne
rem # ADD THE FOLLOWING:
set "JAVA_OPTS=%JAVA_OPTS% -Djboss.server.default.config=standalone-full.xml"
set "JAVA_OPTS=%JAVA_OPTS% -Dorg.eclipse.emf.ecore.EPackage.Registry.INSTANCE=org.eclipse
set "JAVA_OPTS=%JAVA_OPTS% -Dorg.apache.el.parser.COERCE_TO_ZERO=false"
```

7. Deploy the ear with LSPS Application: here we deploy the default ear from lsps-runtime.

```
cp ~/Downloads/lsps-runtime/lsps-application-3.2.ear ~/wildfly/standalone/deployments/
```

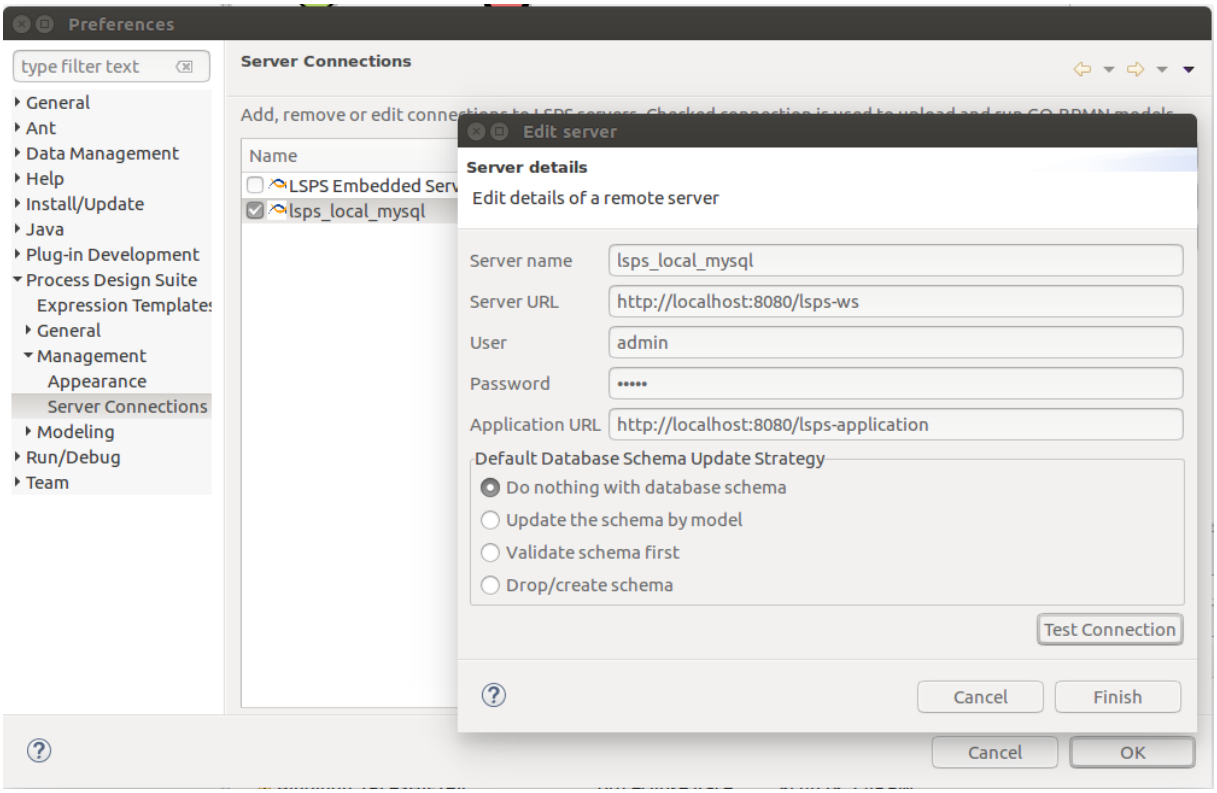
8. Start the server:

```
~/wildfly/bin$ ./standalone.sh
```

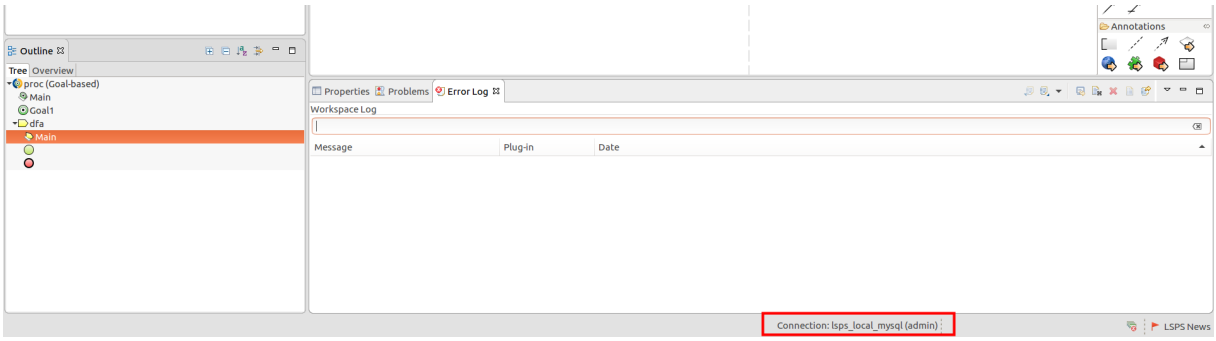
7.3 Connecting to Local WildFly from PDS

To connect your PDS to the server, do the following:

1. In the Modeling perspective of PDS, go to **Server Connection** > **Server Connection Settings**
2. In the dialog, click **Add**.
3. Enter the connection properties and test the connection.



4. Make sure the connection is selected in the Server Connections.
5. In the status bar, check that the connection is active.



Now you can use the `management perspective` to "communicate" with the LSPS Application on your server.