jBPM Form modeller - Getting Started guide

Version 6.1.0-SNAPSHOT

by *The JBoss Drools team* [http://www.jboss.org/drools/team.html]


Drools
Expert

Chapter 1. What is jBPM Form modeler

jBPM Form modeler is a form engine and editor that enables users to create forms to capture and display information during task execution, without needing any coding or template markup skills.

It provides a WYSIWYG environment to model forms that it's easy to use for less technical users.

Key features:

- Form Modeling WYSIWYG UI for forms
- Form autogeneration from data model / Java objects
- Data binding for Java objects
- Formula and expressions
- Customized forms layouts
- Forms embedding

The form modeler's user interfaces is aimed both at process analyst and developers for building and testing forms.

Developers or advanced used will also have some advanced features to customize form behavior and look&feel.

This guide intends to describe in a simple ways all the steps required to create a process with human tasks, generate and modify the forms for these tasks and execute them.

It will provide initial guidance to perform all initial steps, but it will not provide a full description of all available features.

Chapter 2. First steps to create a form driven process

This guide intends to describe in a simple ways all the steps required to create a process with human tasks, generate and modify the forms for these tasks and execute them. It will provide initial guidance to perform all initial steps, but it will not provide a full description of all available features.

Given that forms are going to be used in tasks, it's possible to generate forms automatically from process variables and task definitions. These forms can be later be modified by using the form editor. In runtime, forms will receive data from process variables, display it to the user and capture his input, and then finally updating process variables again with the new values.

The following example will show all the steps to follow to create a form for the 'Create order' task in the process below.

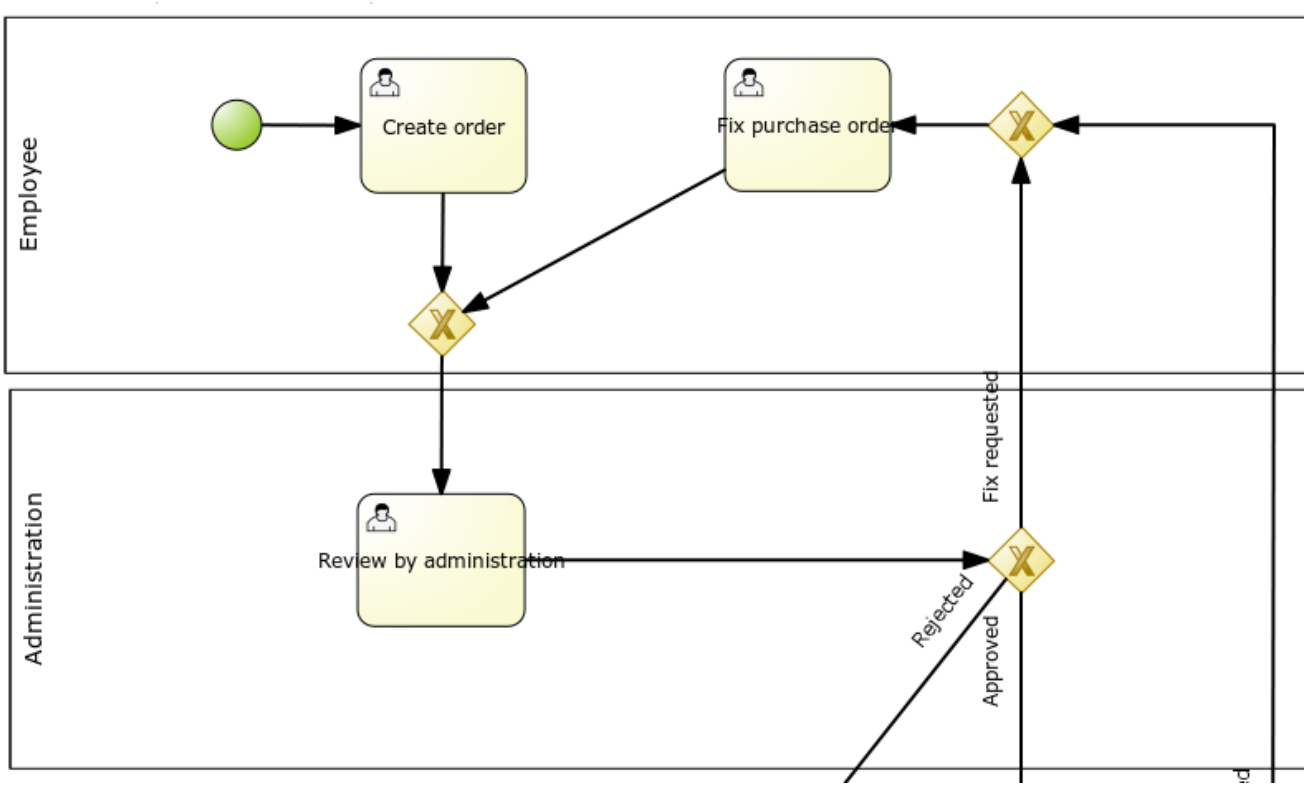


Figure 2.1. Process example

This form must look like the following in execution:

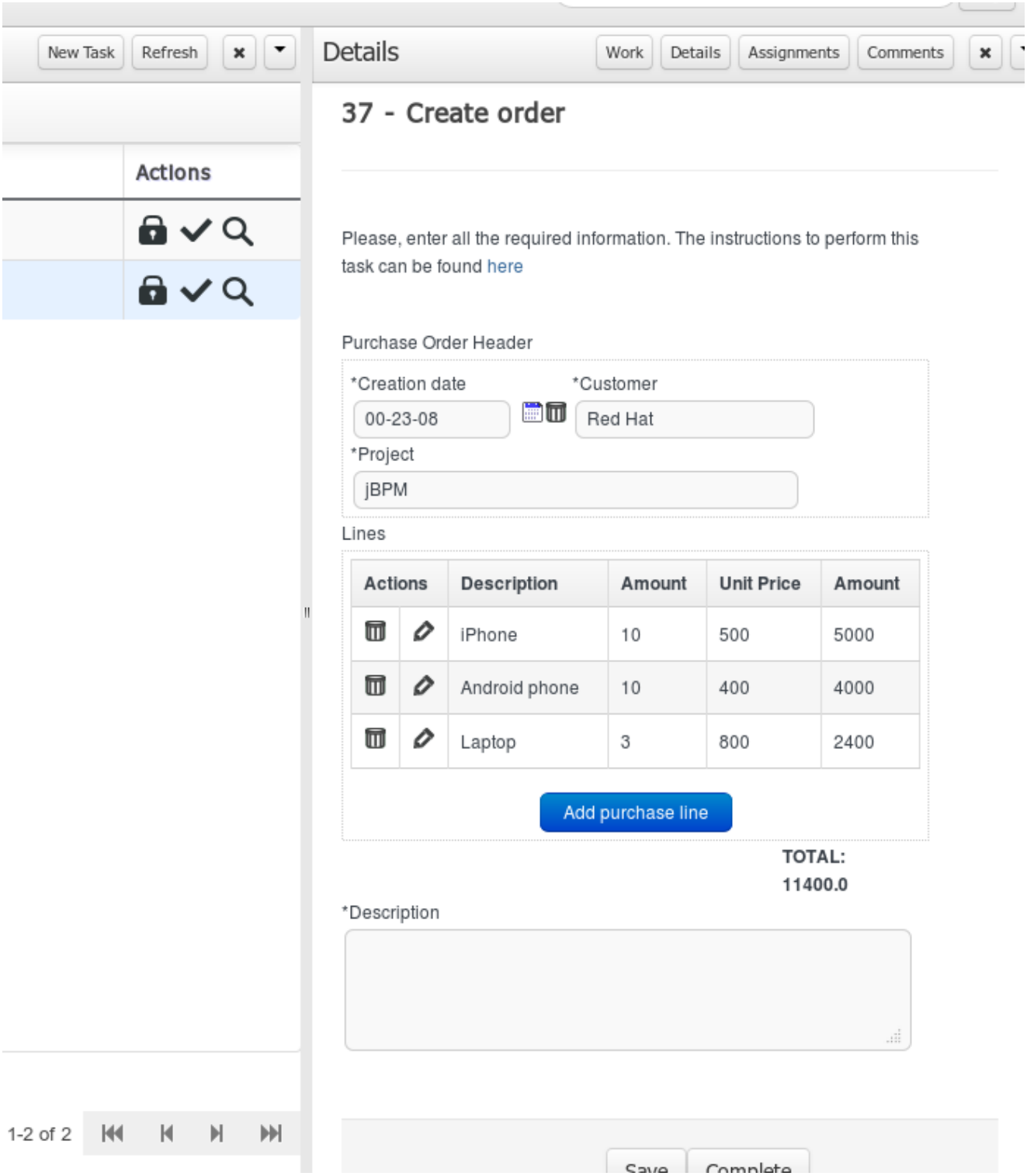


Figure 2.2. Process example

2.1. Configure process and human tasks

To hold values capture by forms, process variables can be created. These variables can be of a simple type like 'String' or a complex type. These complex types can be defined by using the Data Modeler tool, or be just regular POJOs (Plain Java Objects) created with any Java IDE.

In this example, we define a variable 'po' of type 'org.jbpm.examples.purchases.PurchaseOrder', defined with the Data Modeler tool.

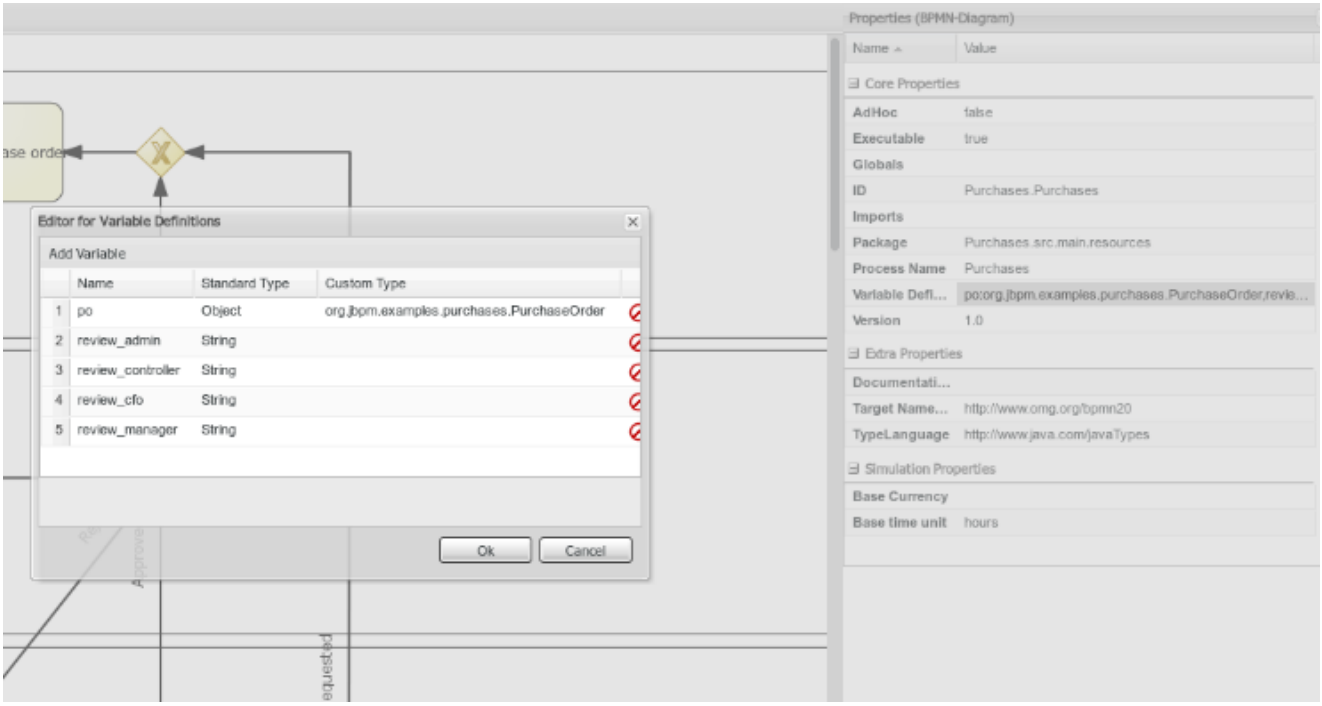


Figure 2.3. Process variable definition

This variable is declared in the 'variables definition' property for the process.

After that, we must configure which variables are set as input parameters to the task, which ones will receive the response back from the form and establish the mappings. This is done by setting the 'DataInputSet', 'DataOutputSet' and 'Assignments' properties for any human task. See screenshots below for details.

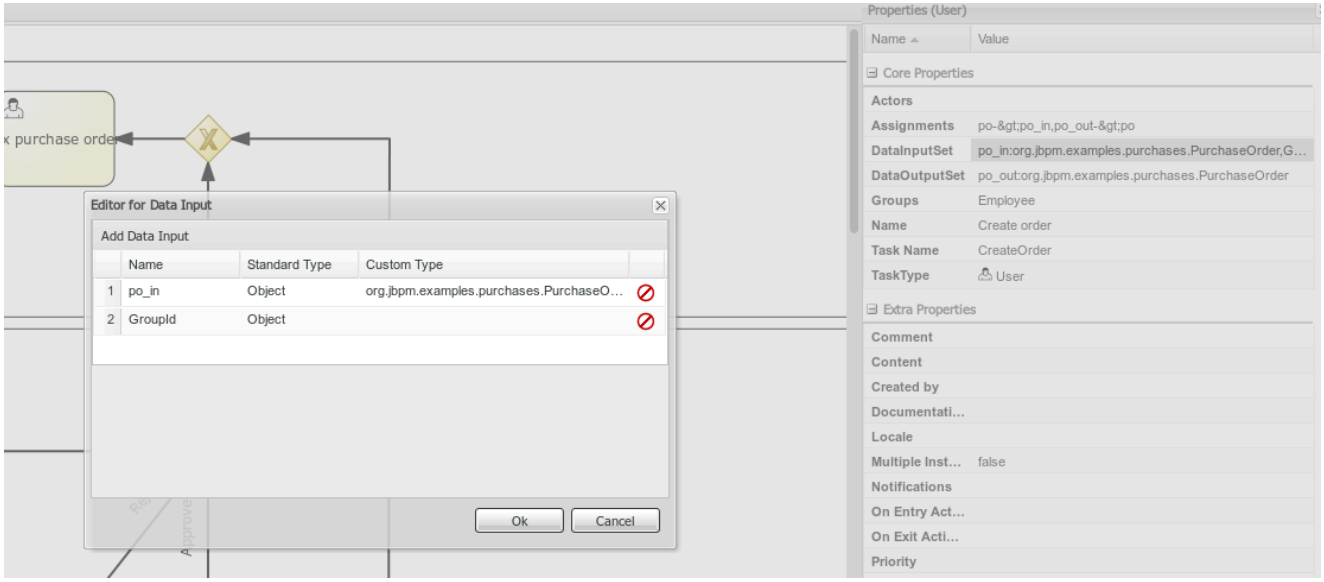


Figure 2.4. Data input variable definition

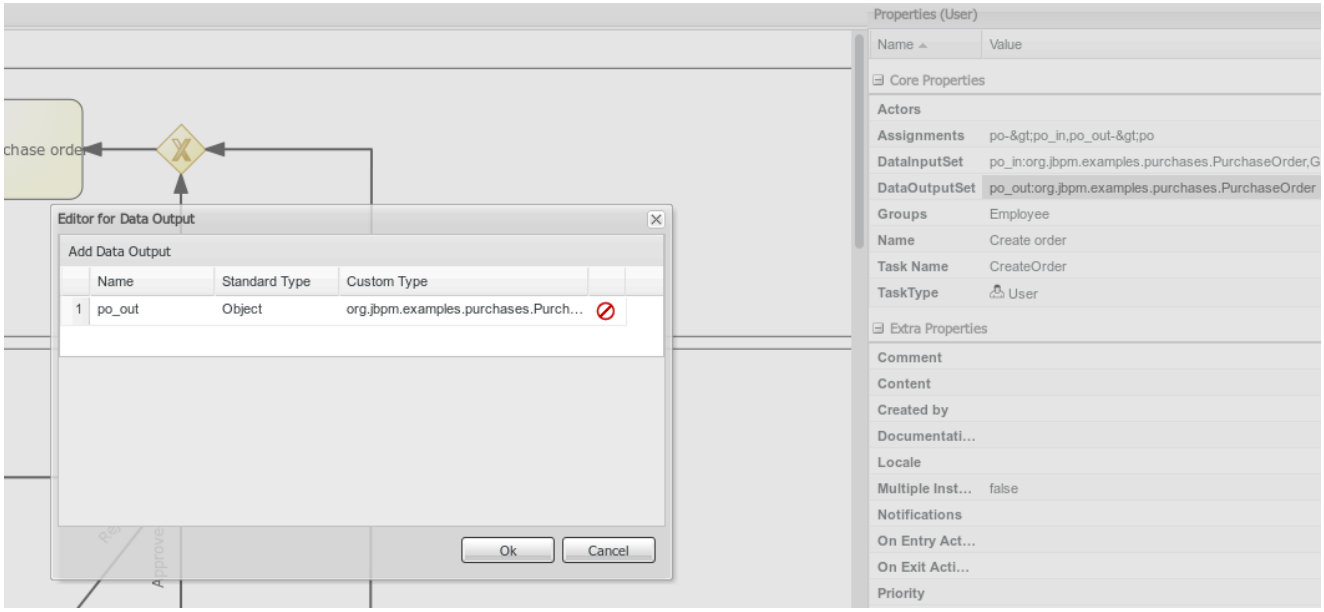


Figure 2.5. Data output variable definition

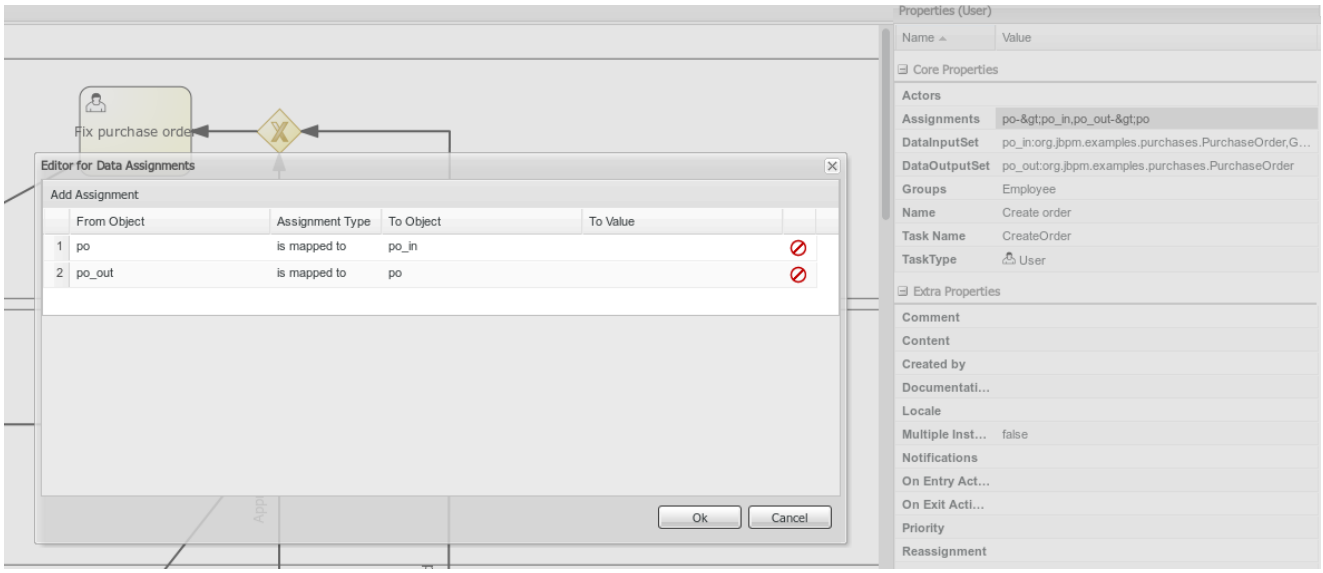


Figure 2.6. Variable mapping definition

2.2. Generate forms from task definitions

The Process Designer module provides some functionality to generate the forms automatically from task and variable definitions, as well as easily open the right form from the modeler.

This is done with the following menu option.

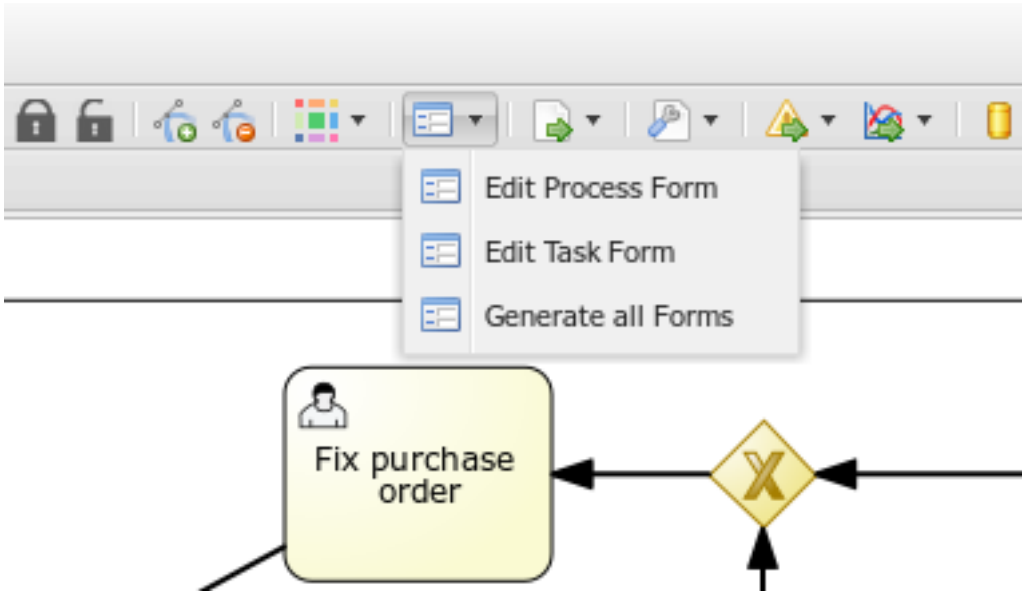


Figure 2.7. Form automatic generation

You can also click on the icon on top of task to open the form directly.

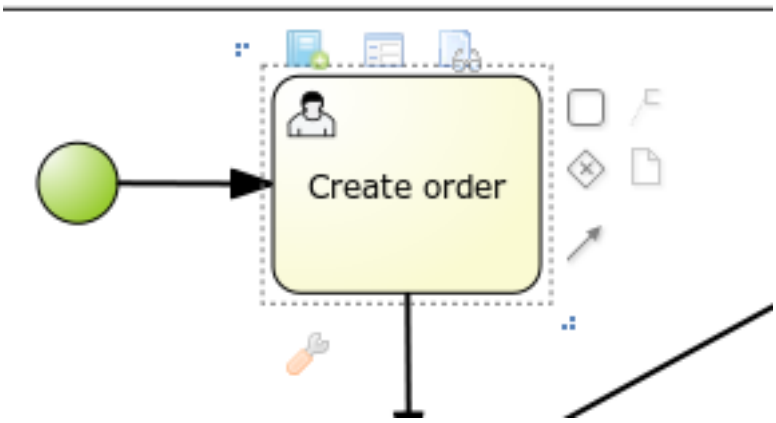


Figure 2.8. Access to form edition

Forms are related to tasks by following a naming convention. If a form with a name formName-taskform is defined in the same package as the process, then this form is used by the human task engine to display and capture information from user.

Also, if a form named ProcessId-task form is created, it will be used as the initial form when starting this process.

For example, for our process the following forms would be generated.

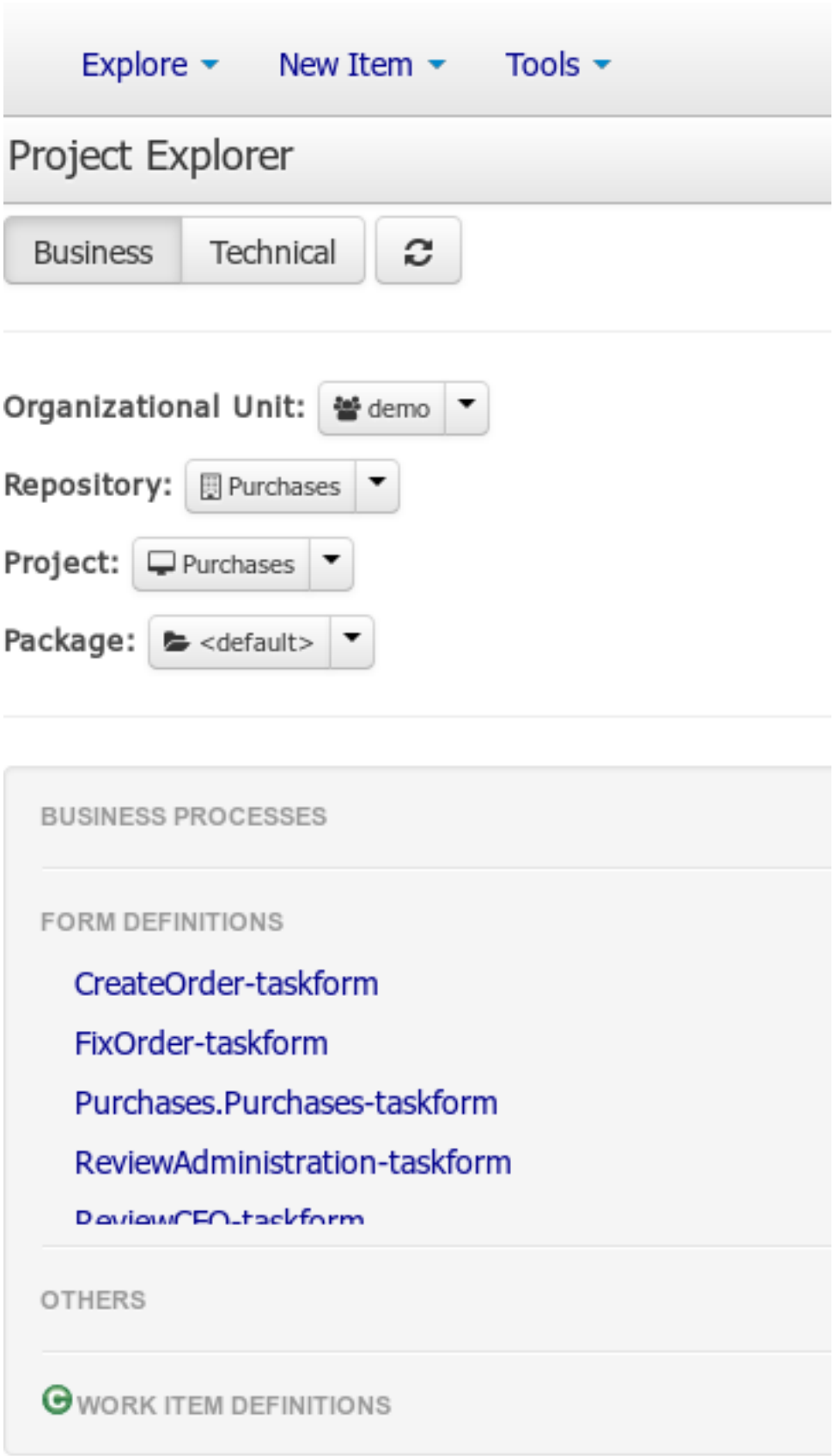


Figure 2.9. Access to form edition

2.3. Edit forms

Once the forms have been generated, you can start editing them. There are several artifacts that are generated in the previous process, but also can be created manually.

2.3.1. Form generated description

When the form has been generated automatically, this tab contain the process variables as data origins. This allow bind form fields with them, this relation it's linked creating data bindings.

A data binding define how task inputs will be mapped to form variables, and when the form is validated and submitted, how the values will update the task outputs.

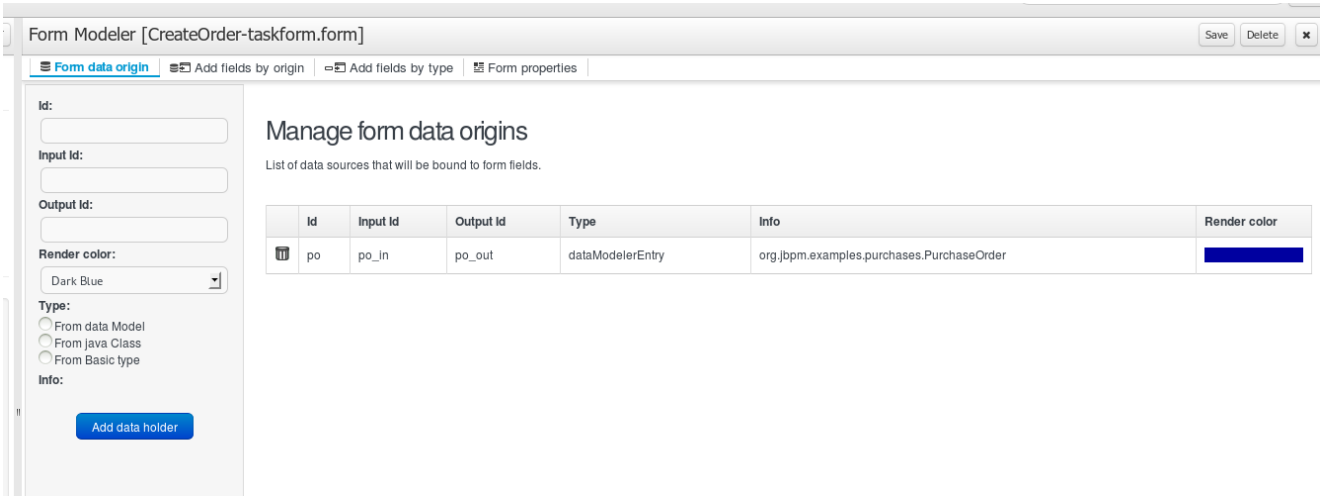


Figure 2.10. Generated form

For example, for this process, the following bindings are generated. Notice that the identifiers are automatically generated. You can have as many data origins as required, and can use a different colour to identify it.

In automatic form generation, a data origin is created for each process variable. The generated form have a field for each data origin bindable item (view FieldTypes) and this automatic fields have the binding defined too.

When these fields are displayed in editor the color of the data origin is shown over the field to make easy view if the field is correctly bound and the data origin implied.

2.3.2. Customizing form

We can change the way the form is displayed to the user in the task list. Next, we will show different levels of customization that will allow change it

2.3.2.1. Moving fields

The fields may be placed in different regions of the form. To move a field the user can access the contextual menu of the field and select 'Move field'.

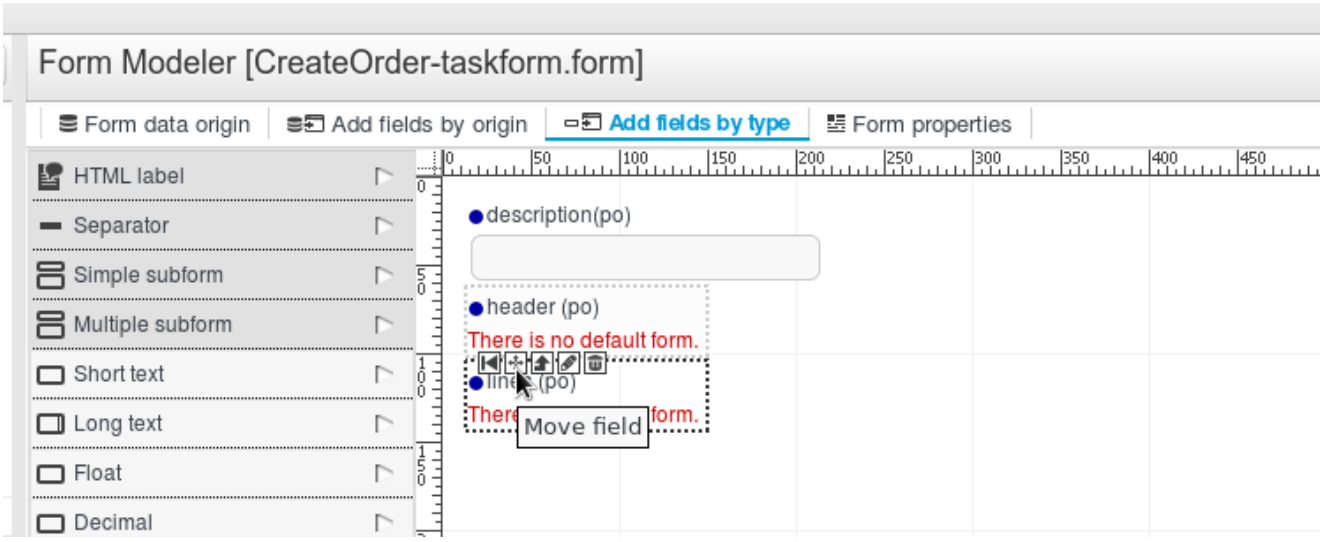


Figure 2.11. Move field option

This will display the different regions of the form where you can place it.

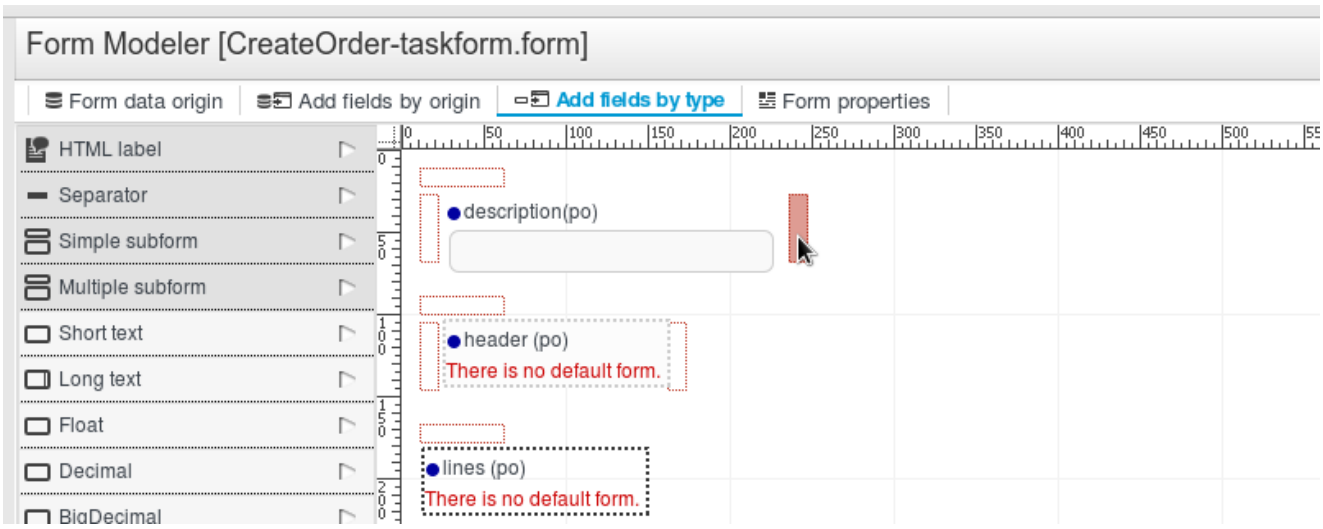


Figure 2.12. Destination areas to move the field

A field can be moved to the first or the last region with the contextual icons for that purpose.

2.3.2.2. Adding new fields

You can add fields to forms either by its origin or by selecting one type of form field.

Let's see what has been created automatically for this purchase order form.

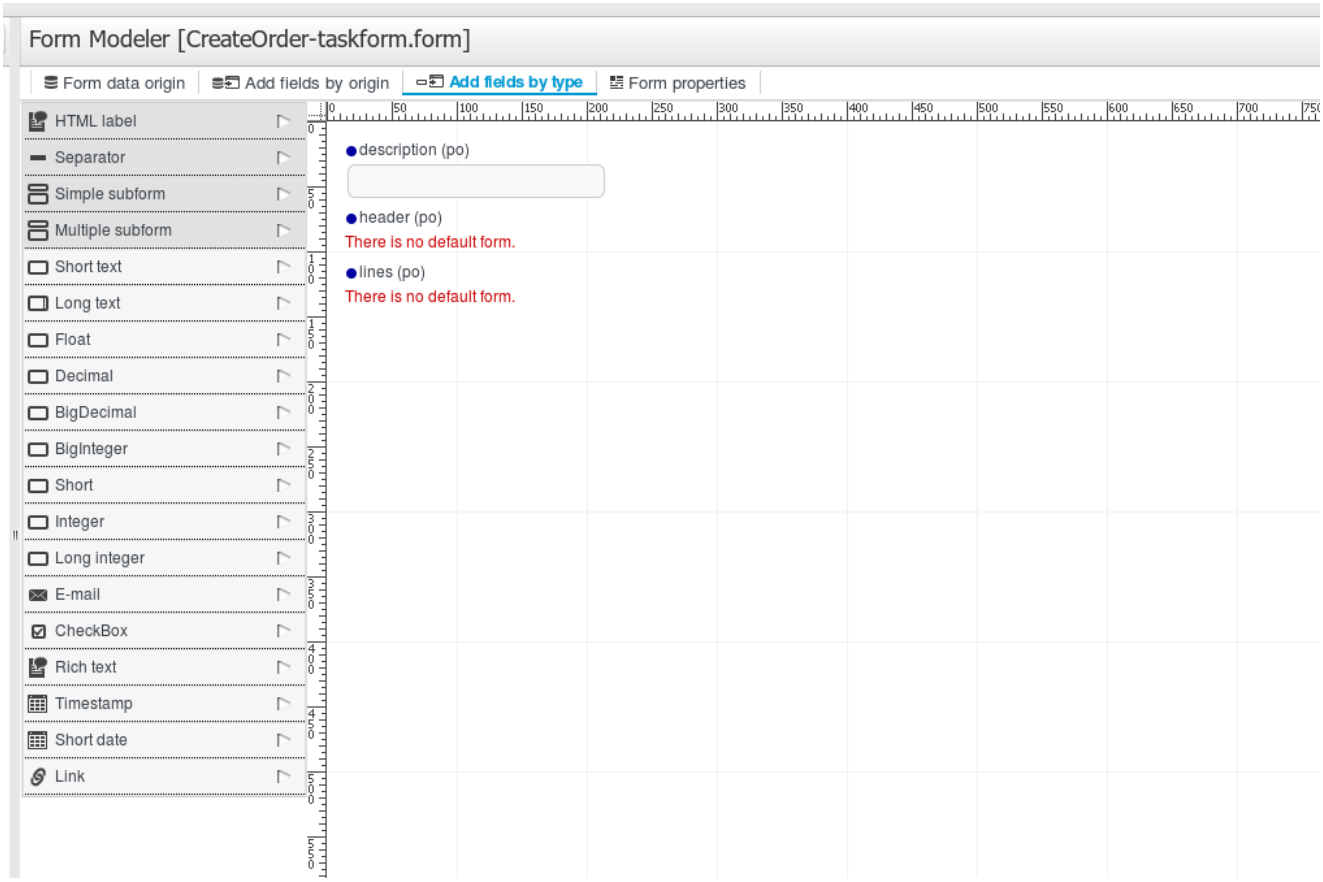


Figure 2.13.

Note

All the properties have been added by default, but are not still configured.

Note

- Add fields by origin: this tab allows you to add fields to the form based on the data origins defined. These fields will have the correct configuration on the "Input binding expression" and "Output binding expression" properties so whe the form is submitted the fields values will be stored in the corresponding Data Origin.

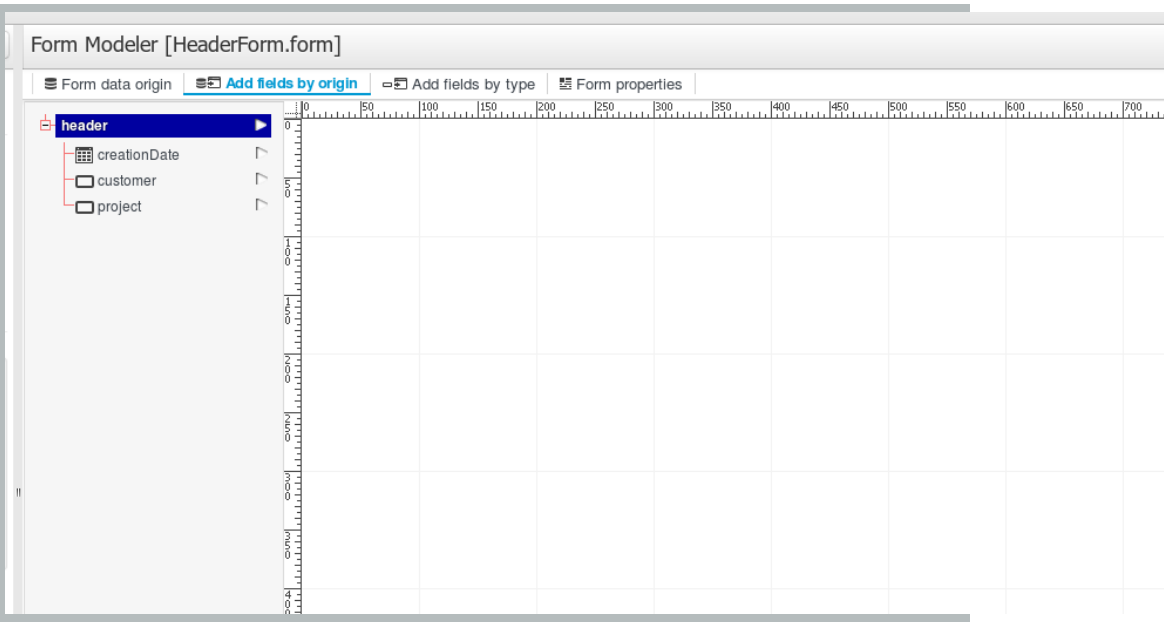


Figure 2.14. Add field by origin

- Add fields by type: this tab allows you to freely add fields to the form from the Field Types palette on the Form Modeler. This fields won't be storing it's value on any Data Origin until they have a correct configuration on the "Input binding expression" and "Output binding expression" properties.

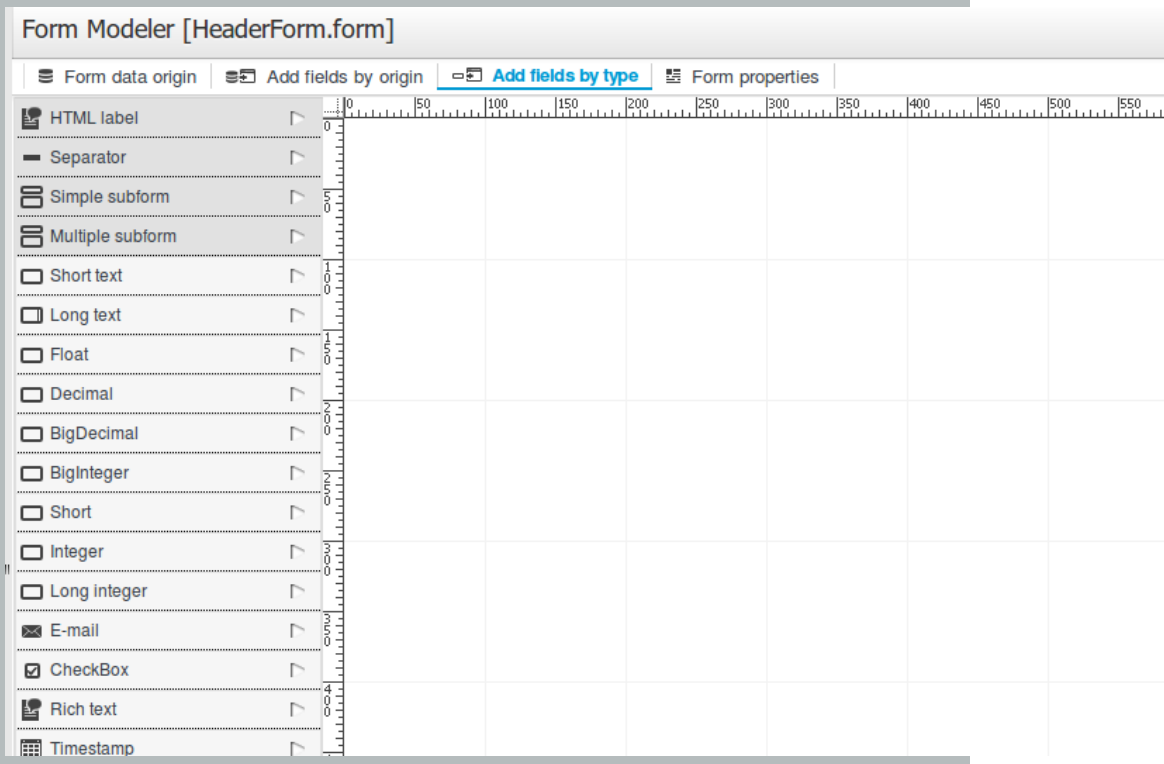


Figure 2.15. Add field by type

To see a complete list of the available field types go to *Field types section*.

Notice the data model 'po' of type 'org.jbpm.examples.purchases.PurchaseOrder' is composed of three properties.

- Simple : property of type text (description). We will adjust the view settings.
- Complex: property of type object (header).
- Complex: property of type array of objects (lines)

Now all these properties had to be configured.

2.3.2.3. Field configuration

Each field can be configured to enhance performance in the form. There are a group of common properties, that we call ‘Generic field properties’ and a group of specific properties that depends on the field type.

2.3.2.3.1. Generic field properties

There are a group of properties that are common to all field types. We will detail them below:

Table 2.1.

Field type	Can change the field type to other compatible field types
Field Name	Will be used as identifier in formulas calculation
Label	The text that will be shown as field label
Error message	When something goes wrong with the field, like validations,.. this message will be displayed
Label ccs class	Allows enter a class css to apply in label visualization
Label css style	to enter directly the style to apply to the label.
Help text	The text introduced is displayed as alt attribute to help to the user in data introduction
Style class	Allows enter a class css to apply in field visualization
Css style	to enter directly the style to apply to the label.

Read Only	When this check is on, the field will be used only for read
Input binding expression	This expression defines the link between field and process task input variable. It will be used in runtime to set the field value with that task input variable data.
Output binding expression	This expression defines the link between field and process task output variable. It will be used in runtime to set that task output variable.

2.3.2.3.2. Specific field properties

Let's explain the specific properties of each field type:

- Short Text (java.lang.String)
  - Compatible field type: Long text, E-mail, Rich text
  - Specific properties
    - Size: input text length.
    - MaxLength: Maximum number of characters allowed.
    - Required: Indicates if it’s mandatory to fill this field.
    - Show html: indicates whether the contents of the field is interpreted as html in show mode.
    - Formula. to enter expressions that will be evaluated to set the field value. These expressions are descrived in *Formula & expression section* .
    - Range value. A range formula allows you to let you specify the values that the user can select from an specific field. These expressions are described in *Formula & expression section*
    - Pattern. Allow introduce an expression to specify the validation of the field. In case that the field value introduced hasn’t match the expression, and error is thrown and the error message has to be shown.
    - Default Value formula. Expression to set the field default value.
- Long Text (java.lang.String)
  - Compatible field type: Long text, E-mail, Rich text
  - Specific properties
    - Size: input text length.

    - MaxLength: Maximum number of characters allowed.
    - Required: Indicates if it's mandatory to fill this field.
    - Height: The number or rows to show at text area.
    - Formula. to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section* .
    - Range value. A range formula allows you to let you specify the values that the user can select from an specific field. These expressions are described in *Formula & expression section*
    - Pattern. Allow introduce an expression to specify the validation of the field. In case that the field value introduced hasn't match the expression, and error is thrown and the error message has to be shown.
    - Default Value formula. Expression to set the field default value.
- Float (java.lang.Float)
    - Specific properties
        - Size: input text length.
        - MaxLength: Maximum number of characters allowed.
        - Required: Indicates if it's mandatory to fill this field.
        - Formula. to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section* .
        - Range value. A range formula allows you to let you specify the values that the user can select from an specific field. These expressions are described in *Formula & expression section*
        - Pattern. Allow introduce an expression to specify how the Float value has to be displayed. The pattern allowed is show in section pattern in *http://docs.oracle.com/javase/6/docs/api/java/text/DecimalFormat.html* [http://docs.oracle.com/javase/6/docs/api/java/text/DecimalFormat.html]
        - Default Value formula. Expression to set the field default value.
- Decimal (java.lang.Double)
    - Specific properties
        - Size: input text length.
        - MaxLength: Maximum number of characters allowed.

    - Required: Indicates if it’s mandatory to fill this field.
    - Formula. Used to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section* .
    - Range value. A range formula allows you to let you specify the values that the user can select from an specific field. These expressions are described in *Formula & expression section* .
    - Pattern. Allow introduce an expression to specify how the Double value has to be displayed. The pattern allowed is show in section pattern in *http://docs.oracle.com/javase/6/docs/api/java/text/DecimalFormat.html* [http://docs.oracle.com/javase/6/docs/api/java/text/DecimalFormat.html]
    - Default Value formula. Expression to set the field default value.
- BigDecimal (java.math.BigDecimal)
    - Specific properties
        - Size: input text length.
        - MaxLength: Maximum number of characters allowed.
        - Required: Indicates if it’s mandatory to fill this field.
        - Formula. Used to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section* .
        - Range value. A range formula allows you to let you specify the values that the user can select from an specific field. These expressions are described in *Formula & expression section* .
        - Pattern. Allow introduce an expression to specify how the BigDecimal value has to be displayed. The pattern allowed is show in section pattern in *http://docs.oracle.com/javase/6/docs/api/java/text/DecimalFormat.html* [http://docs.oracle.com/javase/6/docs/api/java/text/DecimalFormat.html]
        - Default Value formula. Expression to set the field default value.
- Big integer (java.math.BigInteger)
    - Specific properties
        - Size: input text length.
        - MaxLength: Maximum number of characters allowed.
        - Required: Indicates if it’s mandatory to fill this field.

    - Formula. Used to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section* .
    - Range value. A range formula allows you to let you specify the values that the user can select from an specific field. These expressions are described in *Formula & expression section* .
    - Default Value formula. Expression to set the field default value.
- Short (java.lang.Short)
    - Specific properties
        - Size: input text length.
        - MaxLength: Maximum number of characters allowed.
        - Required: Indicates if it’s mandatory to fill this field.
        - Formula. Used to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section* .
        - Range value. A range formula allows you to let you specify the values that the user can select from an specific field. These expressions are described in *Formula & expression section* .
        - Default Value formula. Expression to set the field default value.
- Integer (java.lang.Integer)
    - Specific properties
        - Size: input text length.
        - MaxLength: Maximum number of characters allowed.
        - Required: Indicates if it’s mandatory to fill this field.
        - Formula. Used to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section* .
        - Range value. A range formula allows you to let you specify the values that the user can select from an specific field. These expressions are described in *Formula & expression section* .
        - Default Value formula. Expression to set the field default value.
- Long Integer (java.lang.Long)
    - Specific properties

    - Size: input text length.
    - MaxLength: Maximum number of characters allowed.
    - Required: Indicates if it’s mandatory to fill this field.
    - Formula. Used to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section* .
    - Range value. A range formula allows you to let you specify the values that the user can select from an specific field. These expressions are described in *Formula & expression section* .
    - Default Value formula. Expression to set the field default value.
- E-mail (java.lang.String)
    - Compatible field type: Short text, Long text, Rich text
    - Specific properties
        - Size: input text length.
        - MaxLength: Maximum number of characters allowed.
        - Required: Indicates if it’s mandatory to fill this field.
        - Default Value formula. Expression to set the field default value.
- Checkbox (java.lang.Boolean)
    - Specific properties
        - Required: Indicates if it’s mandatory to fill this field.
        - Default Value formula. Expression to set the field default value.
- Rich text: (java.lang.String)
    - Compatible field type: Short text, Long text, E-mail
    - Specific properties
        - Size: input text length.
        - MaxLength: Maximum number of characters allowed.
        - Required: Indicates if it’s mandatory to fill this field.
        - Height: The number or rows to show at text area.
        - Default Value formula. Expression to set the field default value.

- Timestamp (java.util.Date)
  - Compatible field type: Short date
  - Specific properties
    - Size: input text length.
    - Required: Indicates if it’s mandatory to fill this field.
    - Formula. to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section* .
    - Default Value formula. Expression to set the field default value.
- Short date (java.util.Date)
  - Compatible field type: Timestamp
  - Specific properties
    - Size: input text length.
    - Required: Indicates if it’s mandatory to fill this field.
    - Formula. to enter expressions that will be evaluated to set the field value. These expressions are described in *Formula & expression section* .
    - Default Value formula. Expression to set the field default value.
- Simple subform (Object)
  - For more details see section *Simple Object (Subform field Type)*.

    Specific properties
    - Default form. Show the list of available forms to select what one will be displayed to show the object.
- Multiple subform (Multiple Object)
  - For more details see section *Arrays of objects.(Multiple subform field Type)*.

    Specific properties
    - Default form. Show the list of available forms to select what one will be displayed to show the object when no other form is configured with an specific purpose.
    - Preview form. If a form is specified, it will be used to show the item details
    - Table form. If a form is specified, it will be used to show the table columns when the item list is showed

- New item text. Text to show at New Item button
- Add item text. Text to show at Add Item button
- Cancel text. Text to show at Cancel button
- Allow remove Items. If this check is selected, the form allow remove items in table view.
- Allow edit items. If this check is selected, the form allow edit items in table view.
- Allow preview items. If this check is selected, the form allow preview items in table view.
- Hide creation button. Check to not show the creation button
- Expanded. If is checked, when a new item is being added, the field display the table with the existing items and the creation form at same time
- Allow data enter in table mode. Allow modify data in table view directly.

2.3.2.3.3. Complex Fields Configuration

There are two types of complex fields: fields representing an object, and fields representing an object array.

Once the field is added to the form, either automatically or manually, it must be configured so that the form had to know how to display the objects that will contain in execution time.

Next we describe how can be the configuration process:

- The first thing to do is define how the contained object will be displayed. This is done creating a form that represents the object.
- In case of the object array, you can define a form to show in preview(edition), or to show when table is shown

Once the form to represent the object, the parent form has to be configured to use them in the parent Subform or Multiple subform.

Below we will describe how the setup would be:

2.3.2.3.3.1. Simple Object (Subform field Type)

One possible way of setting the value for an object property is by using an existing form, and embedding this form into the parent. This is called subform.

In this example, the Purchase Order header data is held in an object. Therefore, we must create a form to enter all the purchase order header data and link it from the parent task form.

We will follow the steps:

1. Create new form.

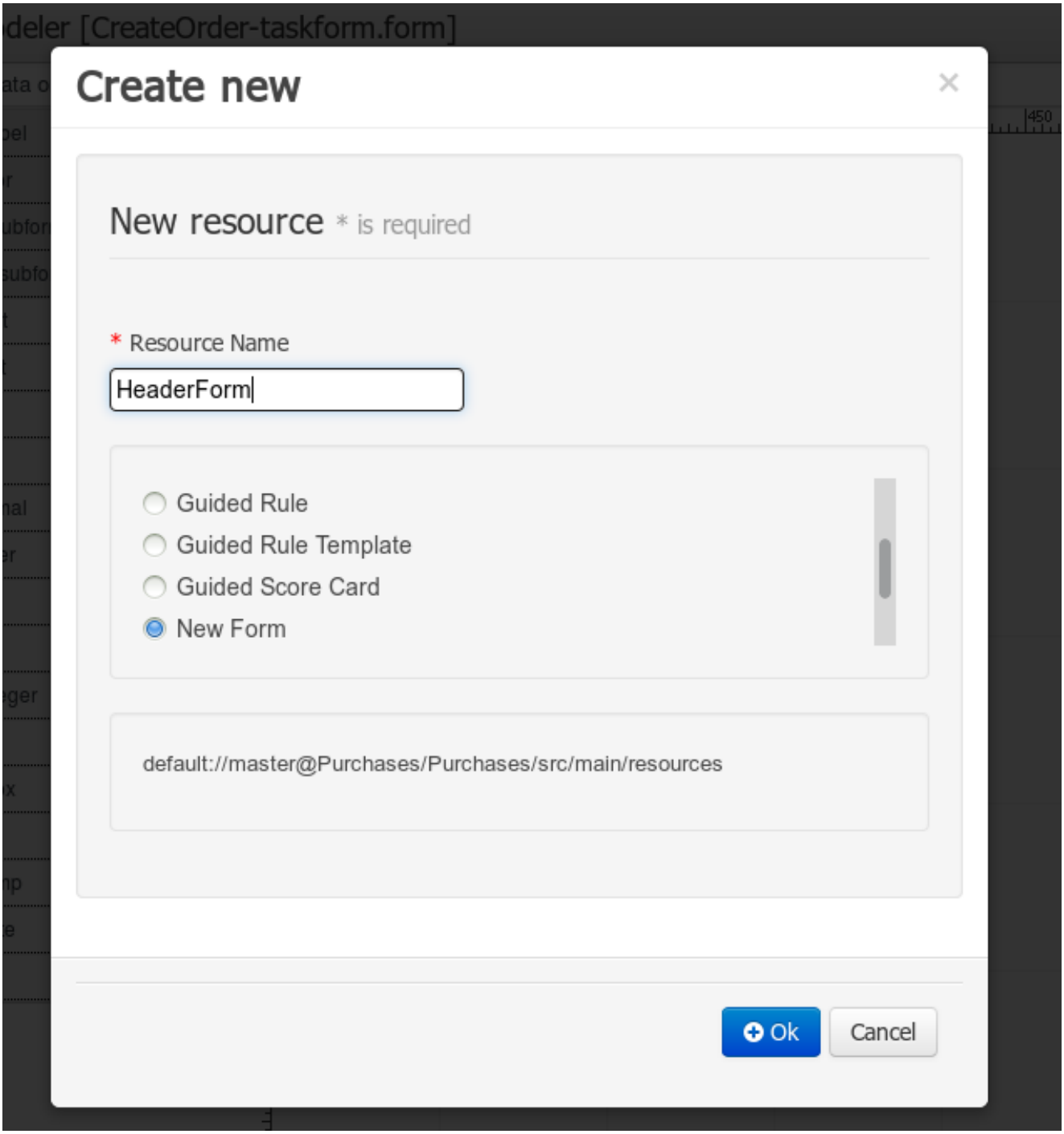


Figure 2.16. Create new form

2. Create new data origin, selecting the type of the purchase order header.

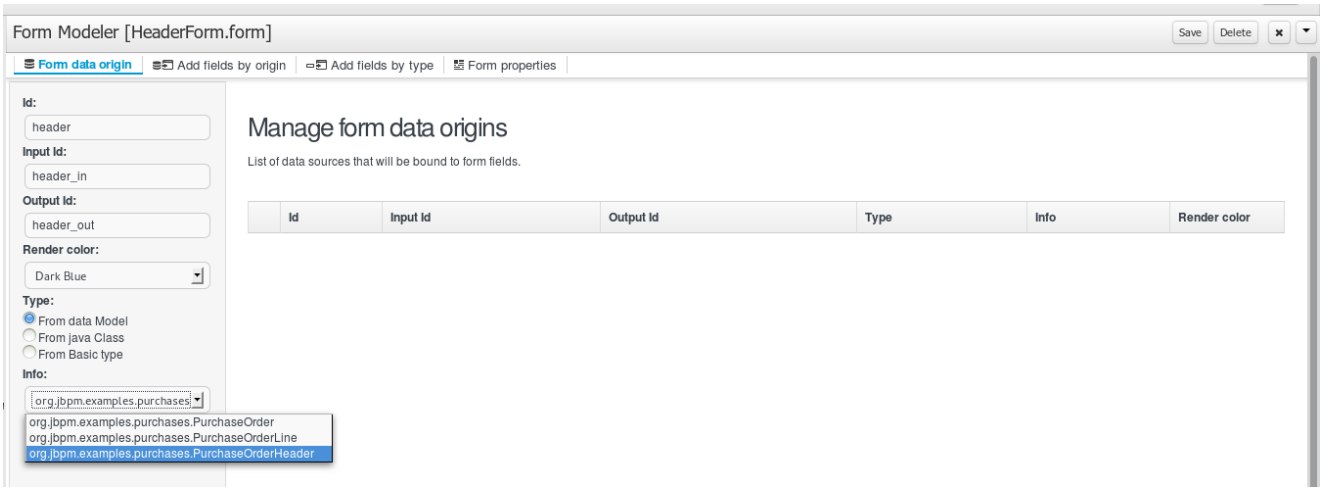


Figure 2.17. Create new data origin

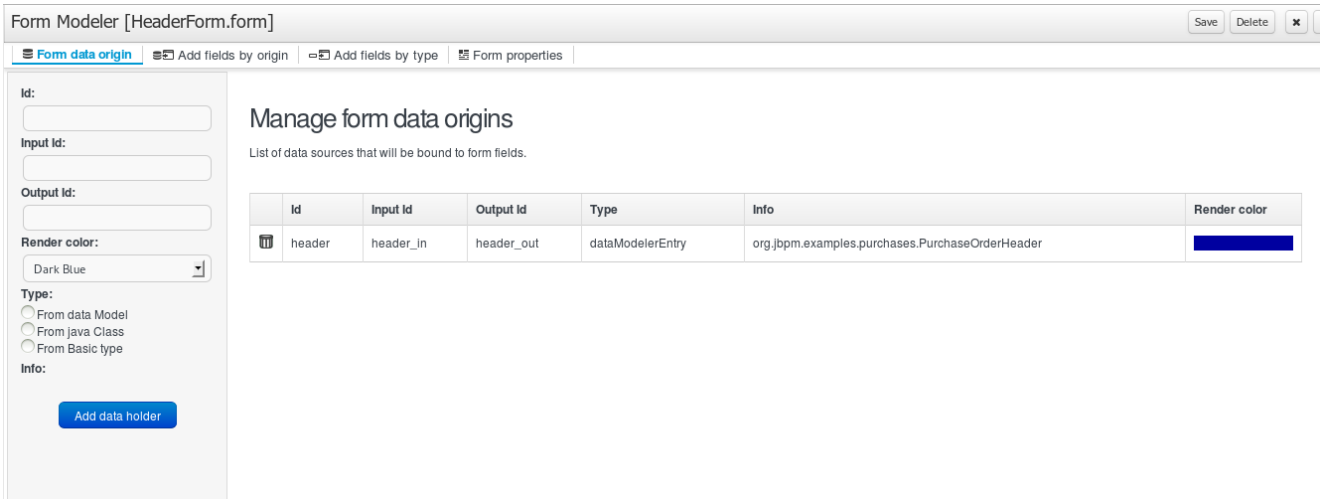


Figure 2.18. Data origin

3. Add fields by origin. All the properties are shown, and can be added to the form, either one by one or all of them at once.

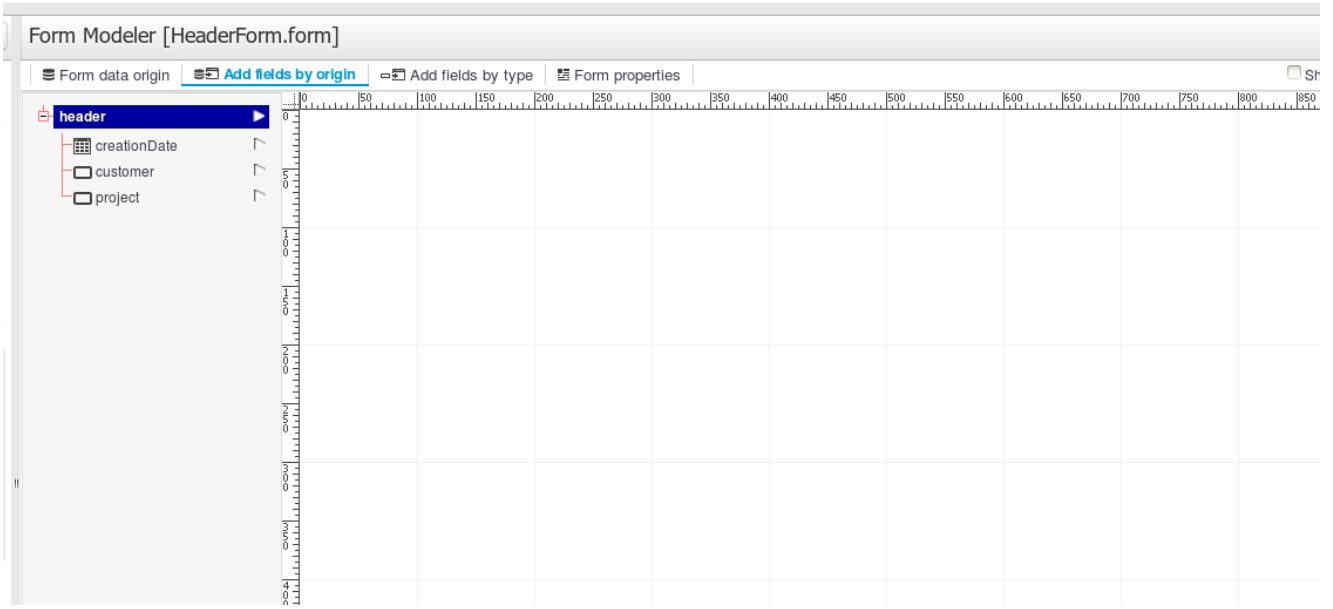


Figure 2.19. Add fields by origin

All the properties have been added to the form, and now we can edit each of them and move them around.

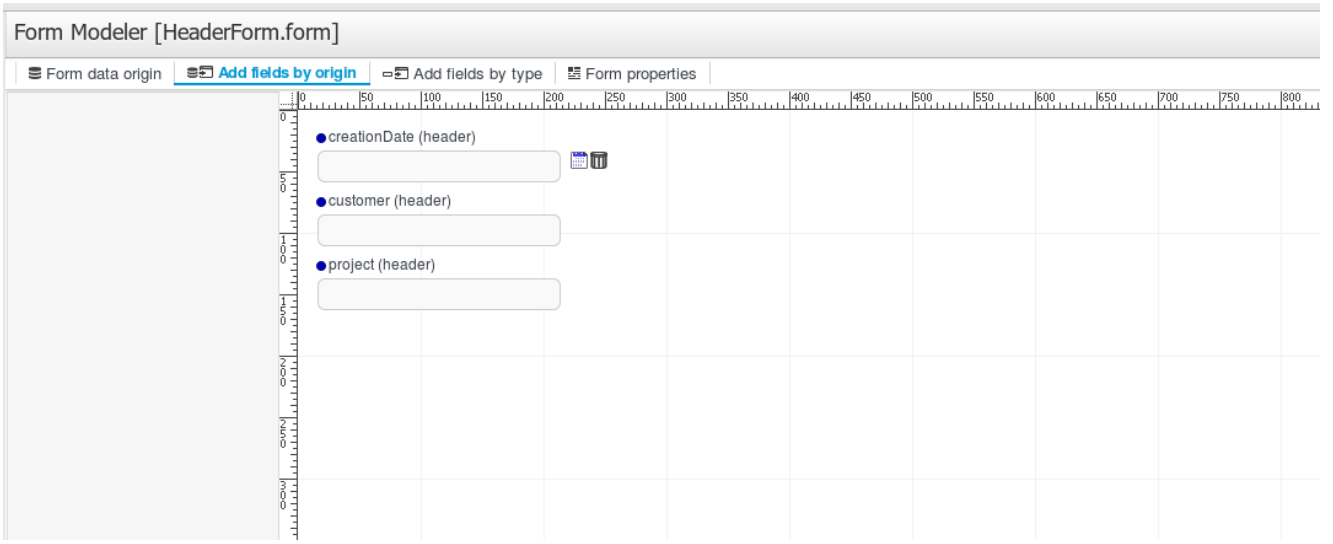


Figure 2.20. All data origin fields added

4. Configure the fields and customize form.

5. Once the form has been saved, open the initial parent form and set the field property 'Default form'.

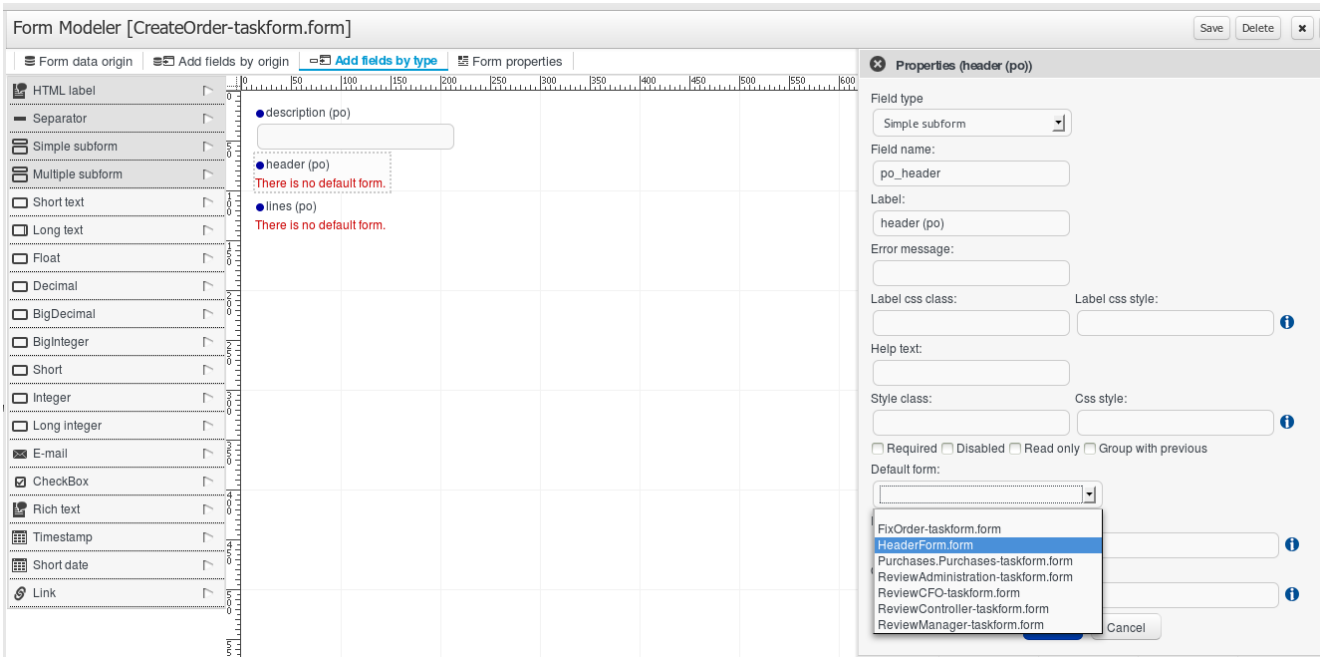


Figure 2.21. Configure the parent form

This will insert the subform inside the parent form, and will be shown as below:

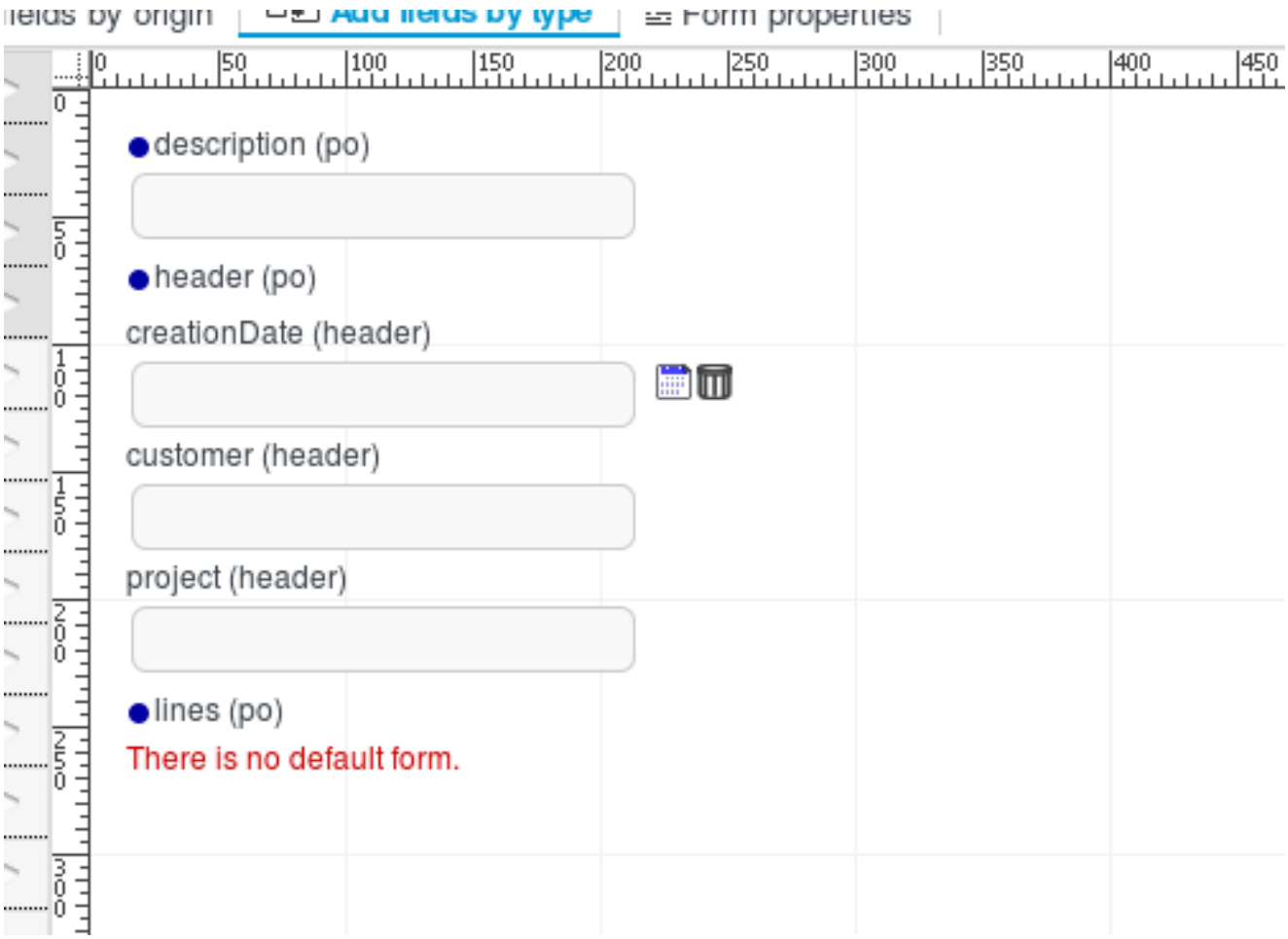


Figure 2.22. Parent form visualization after subform configuration

2.3.2.3.3.2. Arrays of objects.(Multiple subform field Type)

Now, we want to be able to create, edit and remove purchase order lines, by displaying a table with all the values and being able to capture information through a form. This will be done as follows:

Create a form that will hold and capture the information for each line's value (description, amount, unitPrice and total), following the same steps as above. This will be done as follows:

1. Create new form.

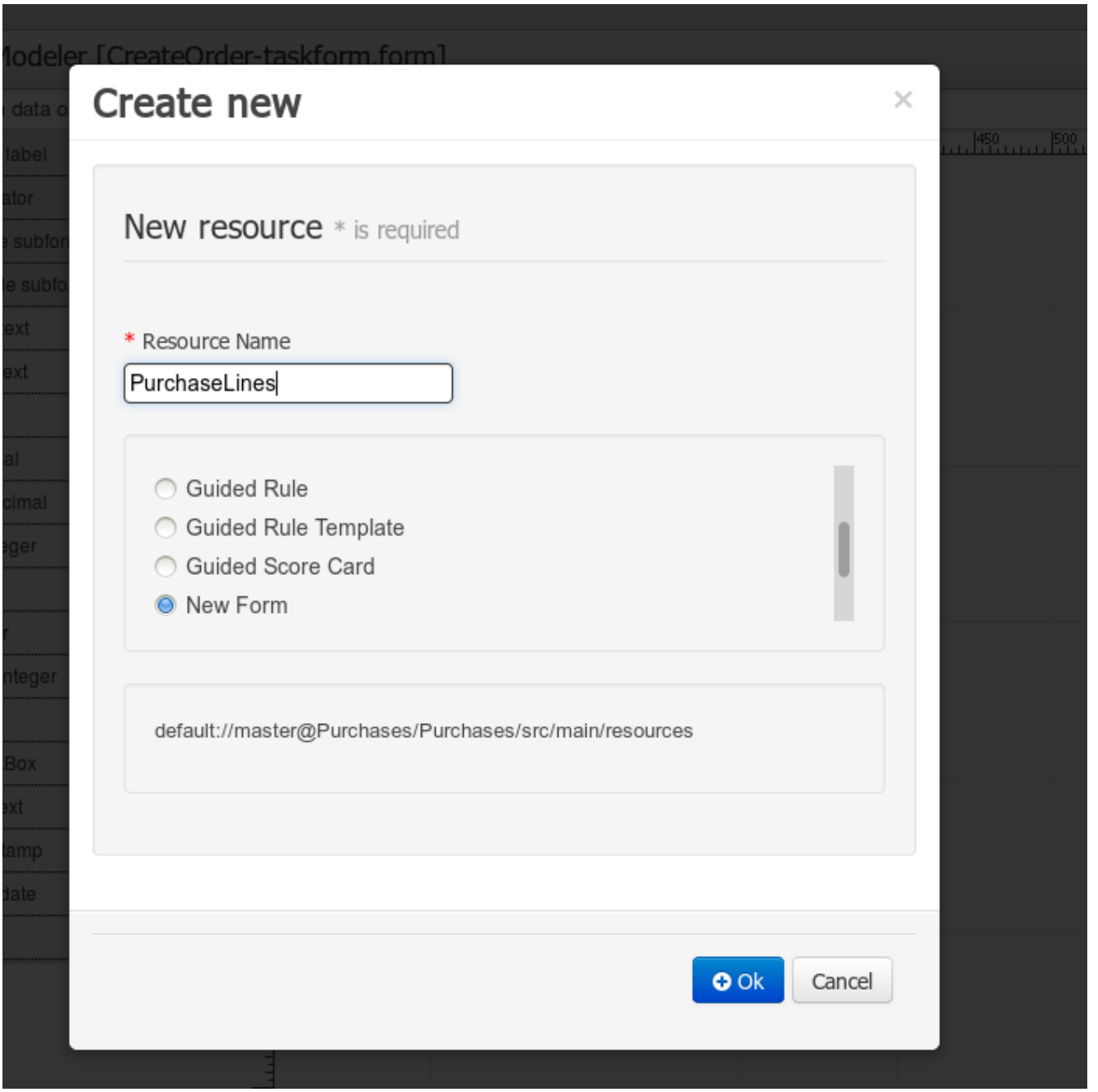


Figure 2.23. Create new form

2. Create new data origin.

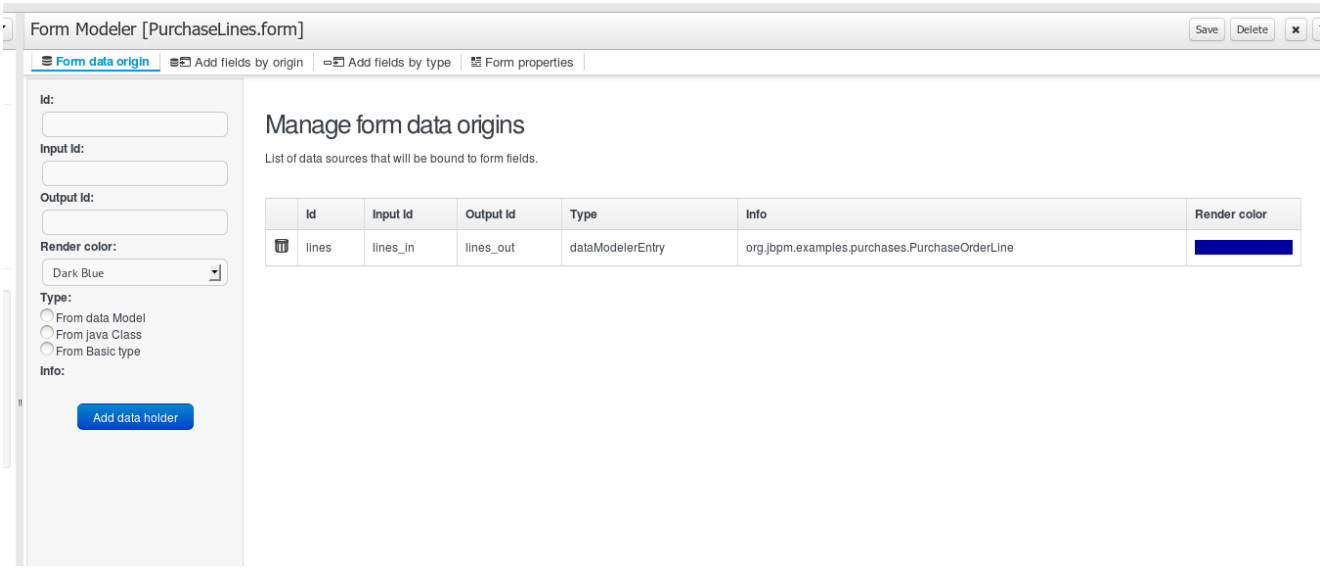


Figure 2.24. Create new data origin

3. Add fields by origin. All the properties are shown, and can be added to the form, either one by one or all of them at once.

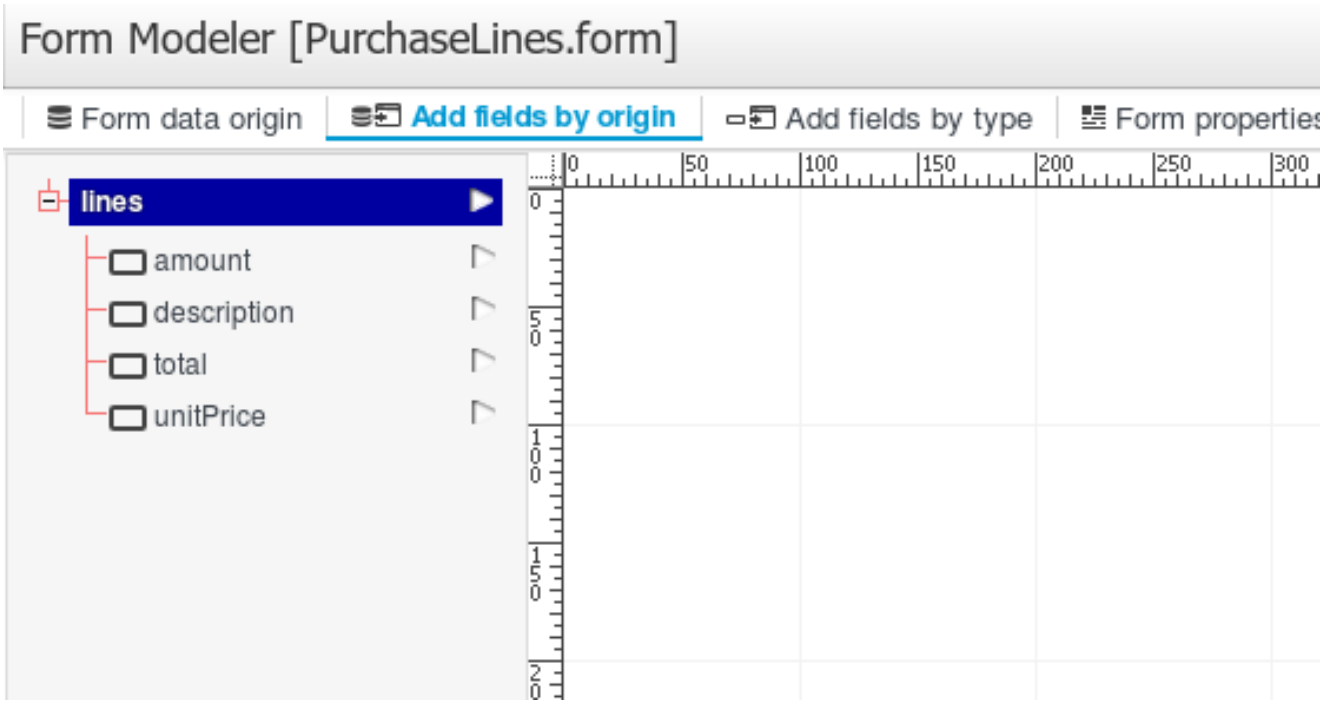


Figure 2.25. Configure the parent form

4. Customize form. Change display options to improve the form visualization

5. Configure the fields. After creating the basic form structure, we can use a formula to calculate automatically the total field. This formulas and expressions are described in *Formula & expression section* .

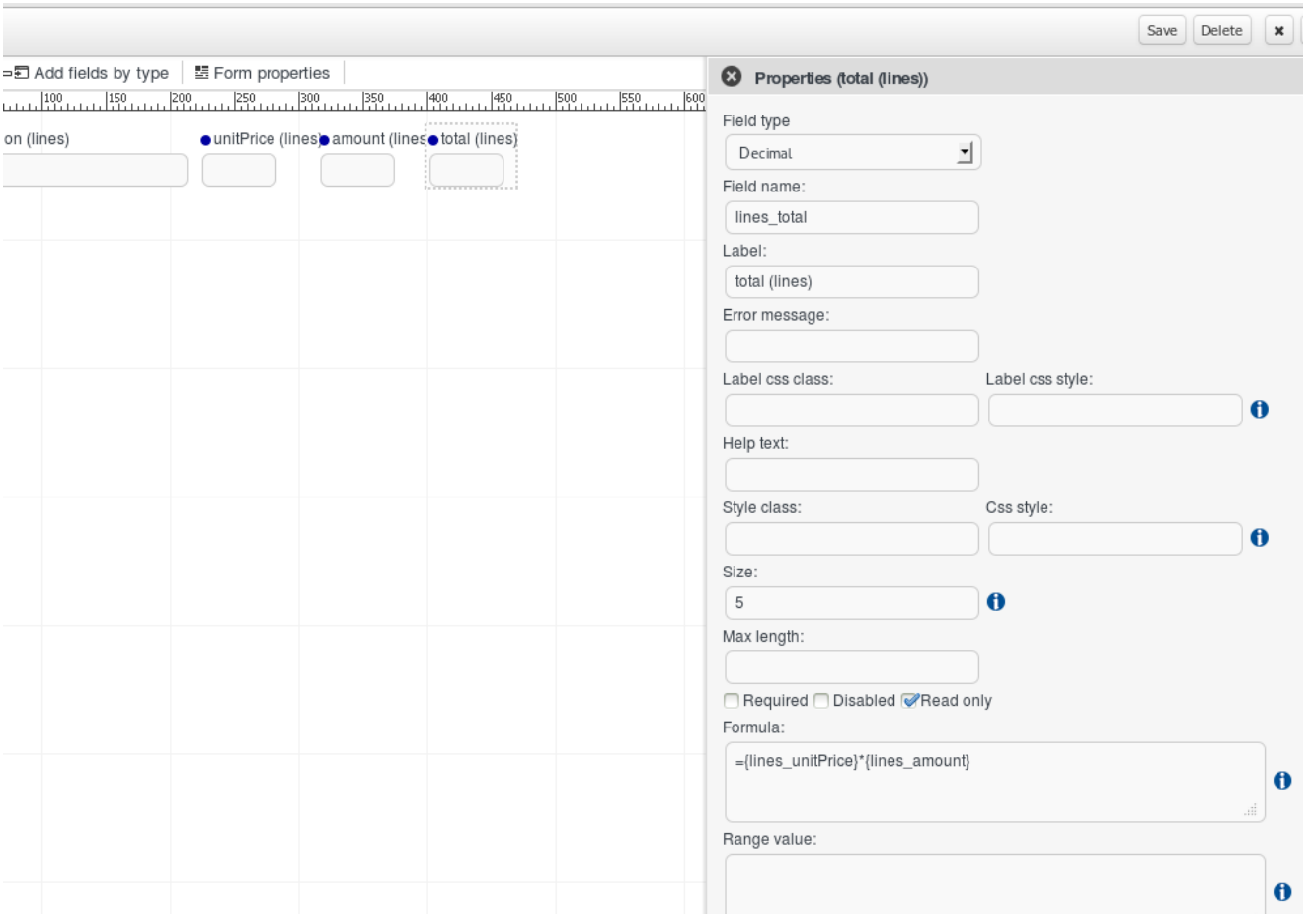


Figure 2.26. Configuring formulas

6. Finally, we save the lines form and go back to the parent form and configure all the lines properties.

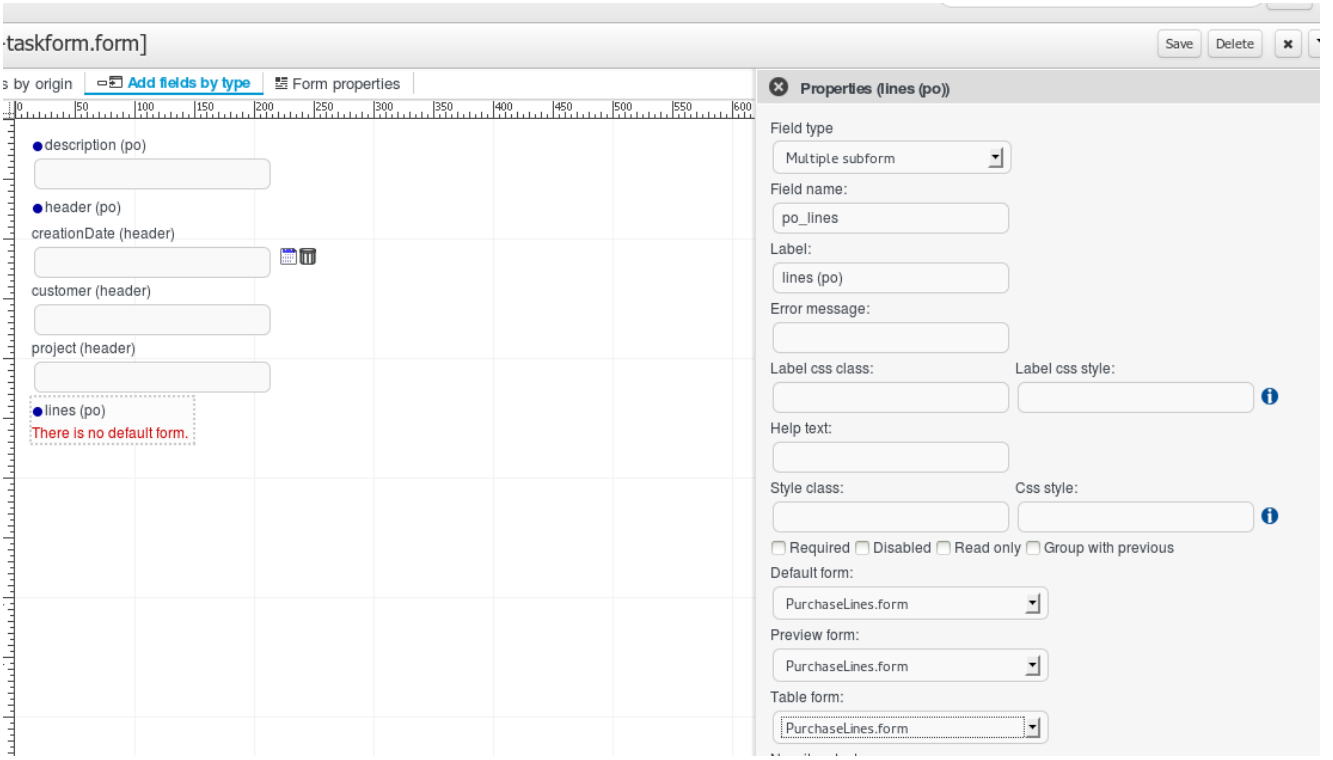


Figure 2.27. Configure the parent form

2.3.2.3.4. Formulas

Form Modeler provides a Formula Engine that you can use to automatically calculate field values. That Formula engine supports Java and XPATH expressions to access the form fields values. Let's see some examples.

- Setting a Default value formula

  Imagine that you have a form that contains a date field "Creation date" that has to be set by default with the current date. To do that you should edit the field properties and set a Default value formula like:

```
=new java.util.Date();
```

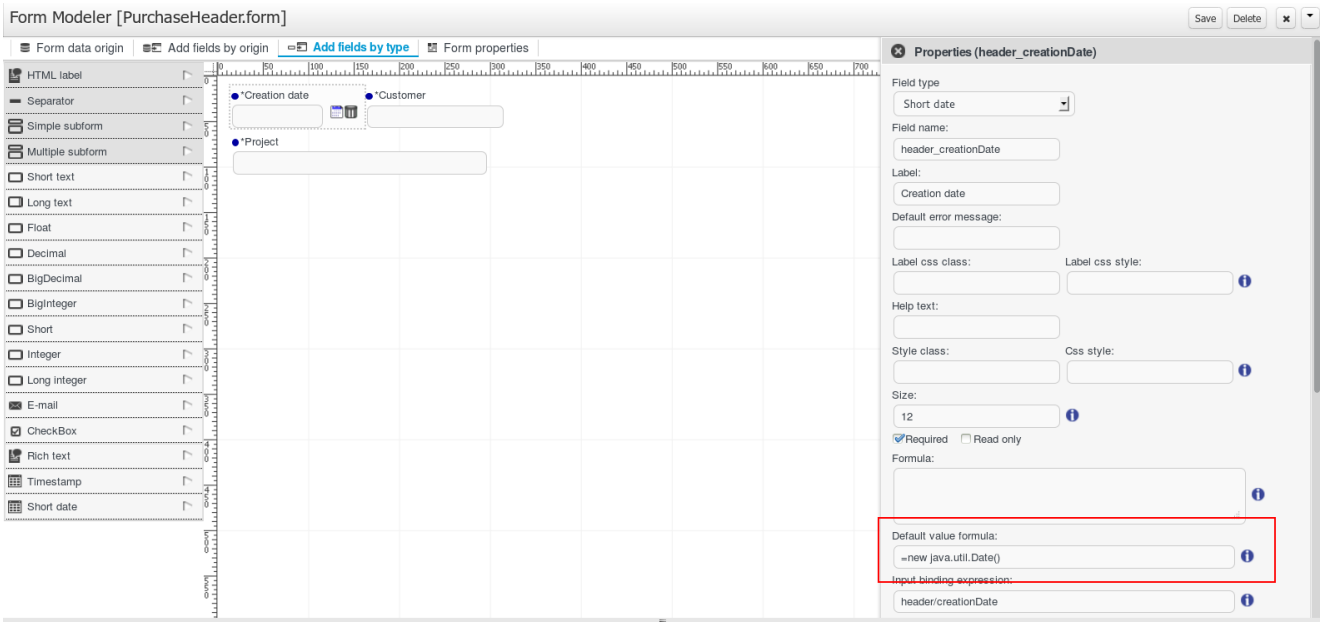


Figure 2.28. Setting default value formula

After setting a Default formula value on a field properties, when the form is rendered by the first time the field will have the specified value.

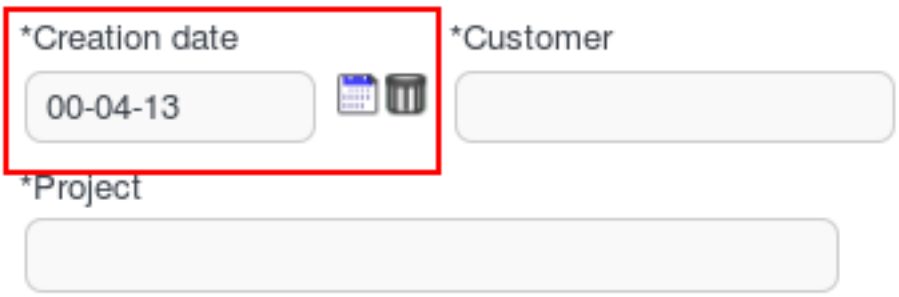


Figure 2.29. Rendering field with default formula

As you can see, you can use a default formula any expression that return a value supported for the field.

- Setting a Formula

The formula engine allows you to calculate formulas that depend on other Field values using XPATH expressions to refer to fields values like {a_field_nane}, standard operators (+, -, *, /, %...) to operate with them or calls to Java Functions for more complex operations.

To start let's see how you can create a formula to calculate the line_total of a Purchase Order Line. Look at the image below and look at the formula on the line_total properties.

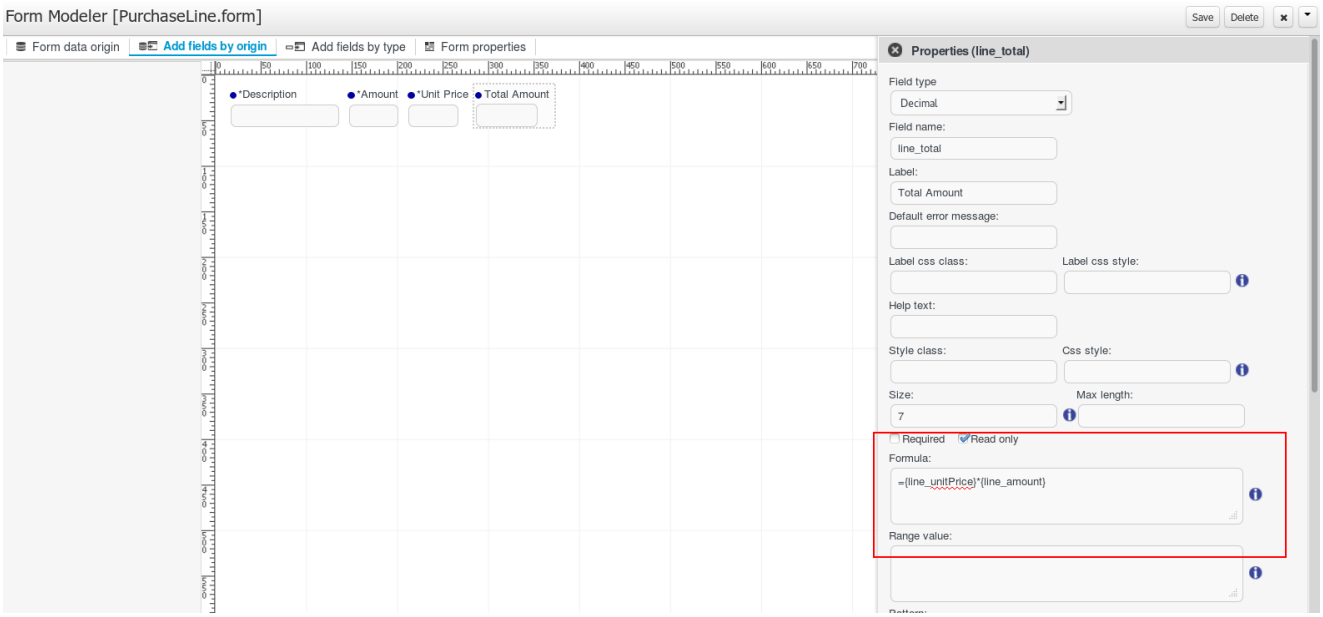


Figure 2.30. Rendering field with default formula

With this expression:

```
={line_unitPrice}*{line_amount}
```

we're forcing the Total of the line value to be the result of the the Unit price multiplied by the Amount, so when the user fills the Amount and Unit Price fields automatically the Total Amount field value is going to be calculated and filled with the operation result:

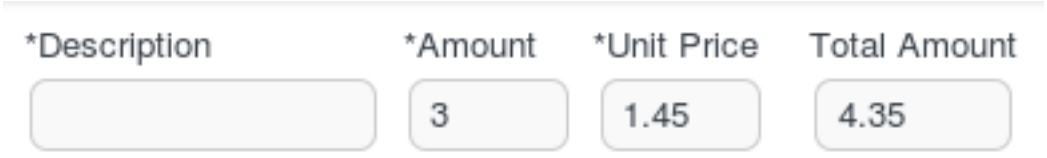


Figure 2.31. Rendering field with default formula result

Is possible to create formulas to operate with values stored in subforms using expressions like

```
={a_field/a_subform_field}
```

Look at the next image to see how it works:

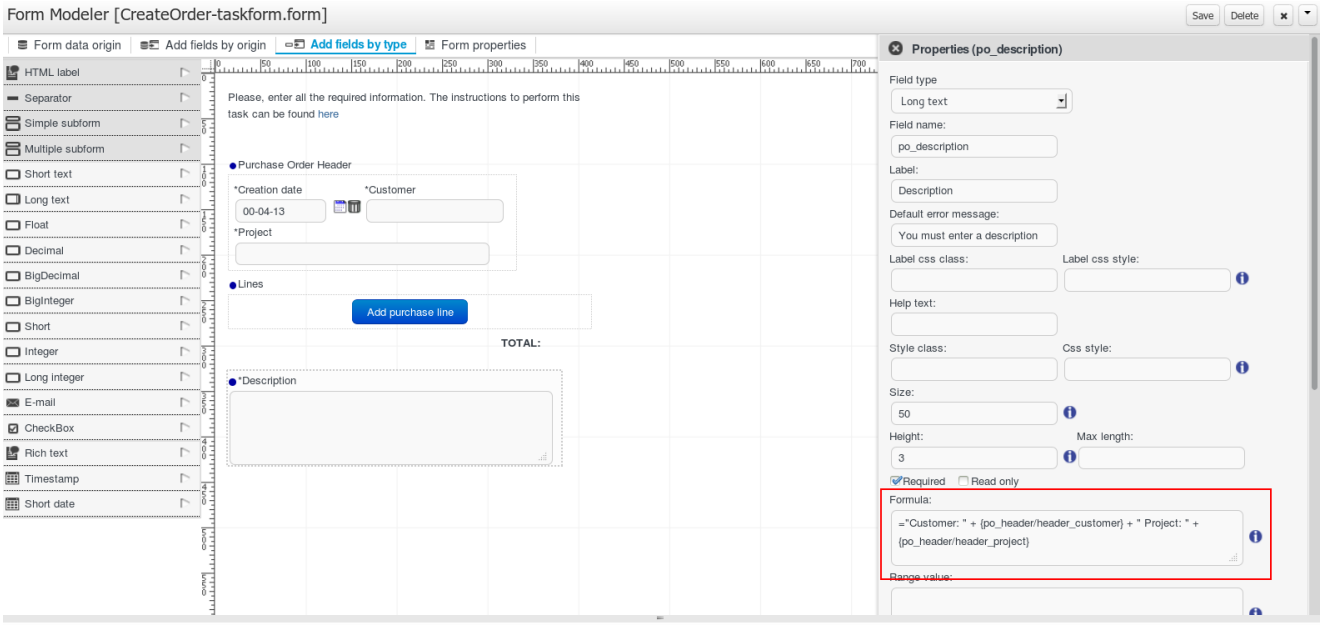


Figure 2.32.

This form has a subform field called po_header that is showing a form with the fields header_creationDate, header_customer and header_project. We want the Description field on our parent form to show some information from the header. Look at the Description field properties formula.

```
="Customer: " + {po_header/header_customer} + " Project: " + {po_header/header_project}
```

This formula returns a text when the fields header_customer and header_projects are filled on the child form, so from now the parent form will be filled like this:

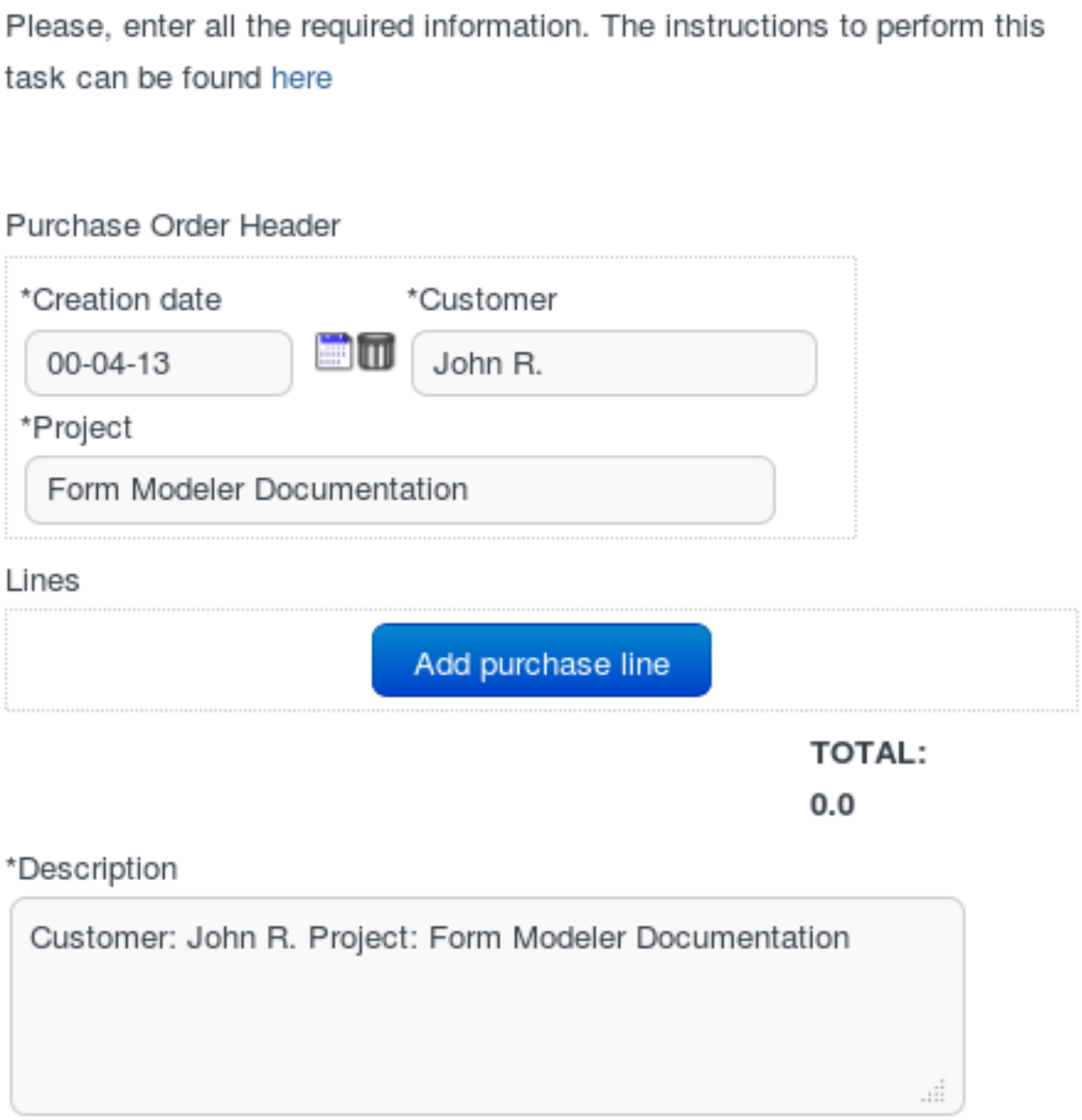


Figure 2.33.

Ok, you've seen how to create formulas that access to a subform fields values, now we are going to see how to work with values stored in Multiple Subforms. Imagine that we have a Purchase Order Line form that contains a multiple subform of Purchase Order Lines, and we want to calculate the total amount of the lines created. Look at the image below and how the TOTAL field is configured.

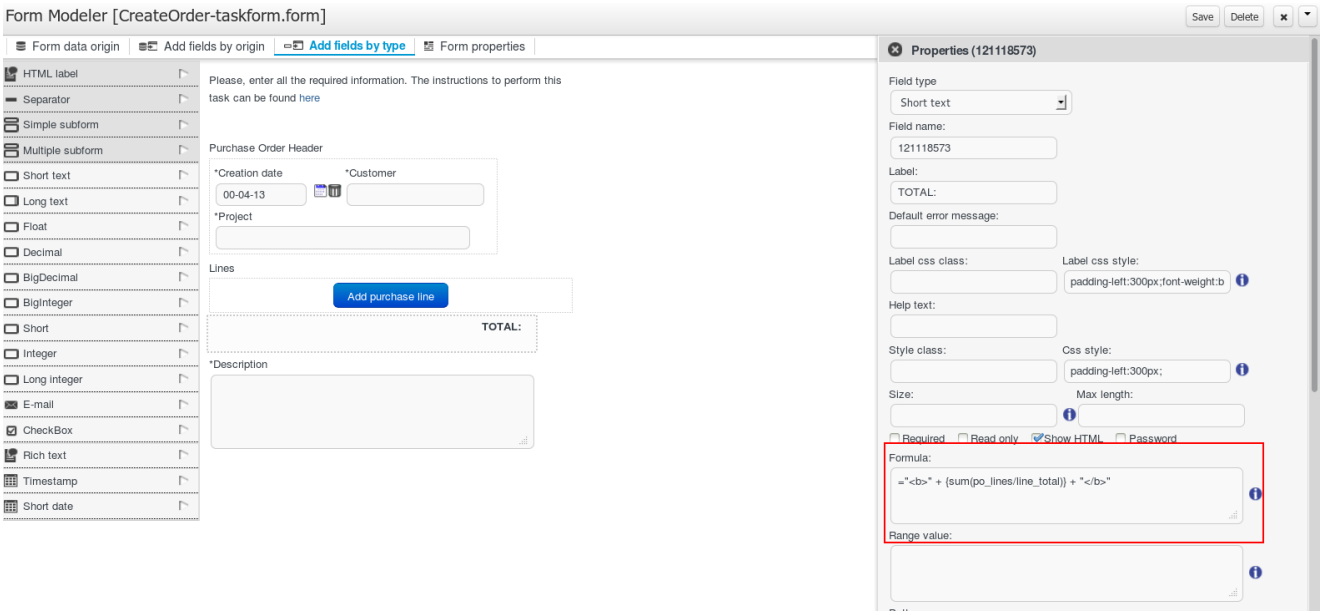


Figure 2.34.

On the formula expression:

```
="
<b>" + {sum(po_lines/line_total)} + "</b>
"
```

we are using the XPATH function sum() that is going to sumarize the totals of all the lines. So after creating some Lines the form will look like this

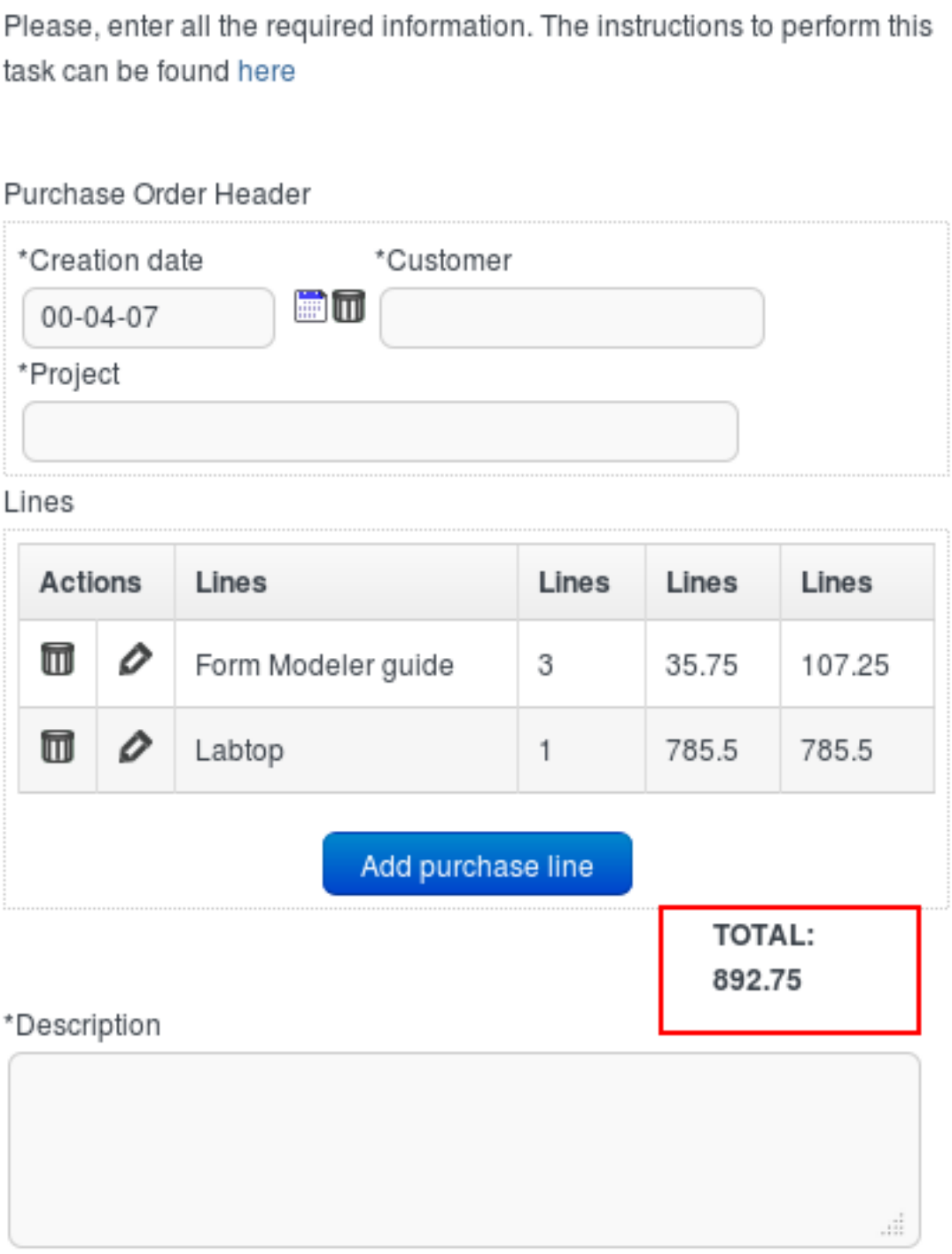


Figure 2.35.

Note that the line_total child field corresponds with the field line_total field on then form selected as a Default Form selected on the Lines field configuratio

On this sample we are using the sum() XPATH function to calculate the total of the Purchase Order, but XPATH provides a lot of possibilities to select values from a set of children and also a lot functions to summarize values (sum, count, avg...). For more information about XPATH you can take a look at *http://www.w3schools.com/xpath/*

- Setting a Range Formula

A range formula allows you to let you specify the values that the user can select from an specific field, showing it like a select box. It can be used on all simple types except Dates and Checkboxes.

To see how it works look the next image and look at the Review Status field configuration.

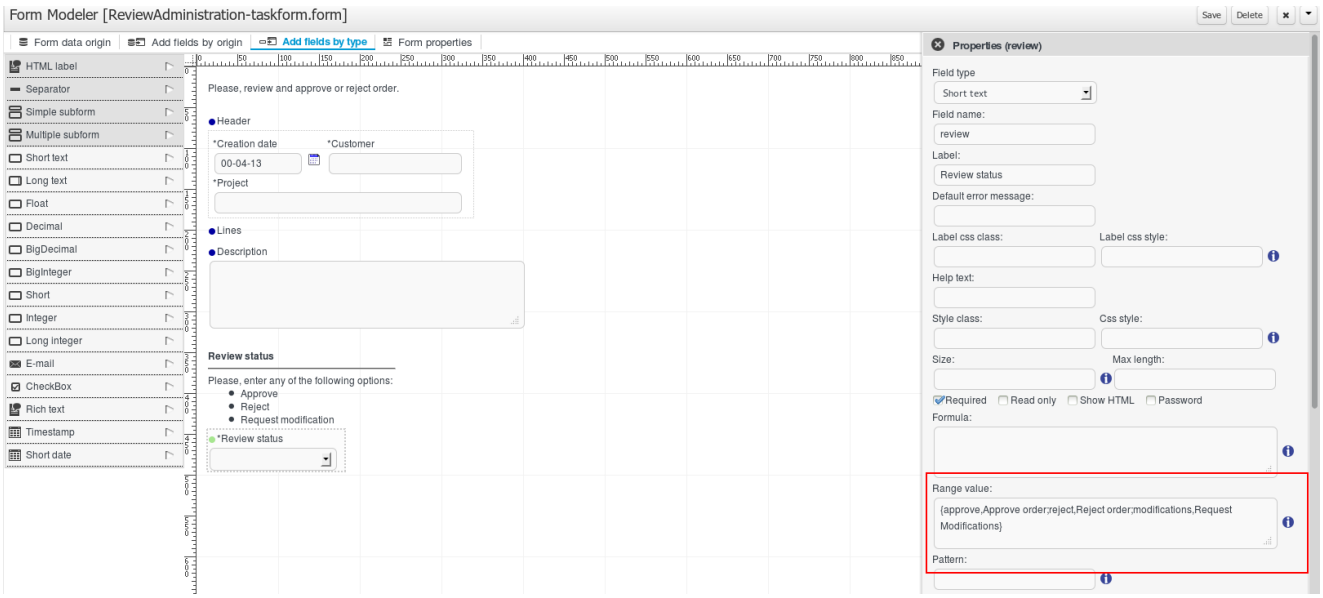


Figure 2.36. Setting default value formula

As you can see that field is being shown as a select box and it has a range formula that specifies the values like this:

```
{approve,Approve order;reject,Reject order;modifications,Request
 Modifications}
```

This expression is defining 3 duos of value/"text to show" separated with the character ',' and each of this duos is separated from each other other with the ';' character. So due this formula the resulting select box will show:

Table 2.2.

Value stored in input	Text shown on Select Box
approve	Approve order
reject	Reject order
modifications	Request Modifications

2.3.2.4. Customizing form layout

When you need an extra customization level and have more control over the html that is displayed. The form modeler provides the ability to edit the html directly.

To use this functionality, the user have to specify that in the ‘Form properties’ tab, 'Custom form layout' option and save.

Now the form is displayed with the custom html. To access this html editing we click on the icon 'Edit'

The html editor is displayed, the html code will define how the form has to be shown. In this editor the user can directly create the html i locate the fields and labels with the syntax described below:

$field{fieldName} for field identified fieldName

$label{fieldName} for field identified fieldName label

These expressions will be replaced by the field or label rendering when the form will be shown.

Form modeler also provides two ways to help in the form html creation.

- 'Insert form elements'

  Two select: one for the fields and another for the labels. Clicking on that, the field or label text is added to html. These selects only show the form fields haven’t been added yet.

- 'Generate template based on'

  This functionality generates the html using all fields (default, alignment fields or Not aligned) depending on the selected value and overwrite the html.

2.3.3. Field types

There are three types of field types that you can use to model your form:

- Simple types

  These field types are used to represent simple properties like texts, numeric, dates, etc. The supported Field types are:

Table 2.3.

Name	Description	Java Type	Default on generated forms
Short Text	Simple input to enter short texts.	java.lang.String	yes
Long Text	Text area to enter long text.	java.lang.String	no
Rich Text	HTMLEditor to enter formatted texts .	java.lang.String	no
Email	Simple input to enter short text with email pattern.	java.lang.String	no
Float	Input to enter short decimals.	java.lang.Float	yes
Decimal	Input to enter number with decimals.	java.lang.Double	yes
BigDecimal	Input to enter big decimal numbers.	java.math.BigDecimal	yes
BigInteger	Input to enter big integers.	java.math.BigInteger	yes
Short	Input to enter short integers	java.lang.Short	yes
Integer	Input to enter integers.	java.lang.Integer	yes
Long Integer	Input to enter long integers	java.lang.Long	yes
Checkbox	Checkbox to enter true/false values	java.lang.Boolean	yes
Timestamp	Input to enter date & time values	java.util.Date	yes
Short Date	Input to enter date values.	java.util.Date	no

- Complex types

  These field types are made to deal with properties that are Java Objects instead of basic types. These field types need extra forms to be created in order to show and write values onto the specified Java Object/s

Table 2.4.

Name	Description	Java Type	Default on generated forms
Simple subform	Renders the a form, it is used to deal with 1:1 relationships.	java.lang.Object	yes
Multiple subform	This field type is used to deal with 1:N relationships. It allows to create, edit and delete a set child Objects.Text area to enter long text.	java.util.List	yes

- Decorators

  Decorators are a type of field types that don't store data in the Object shown on the form. They can be used with aesthetic purpose

Table 2.5.

Name	Description
HTML label	Allows the user to create HTML code that will be rendered in the form
Separator	Renders an HTML separator

2.3.3.1. Custom Field Types

Is possible to extend the platform to add Custom Field Types that make a specific field (of any type) on the form to look and behave totally different than the standard platform fields. On this section we will take a look on how to create them and how to configure them.

2.3.3.1.1. How to create Custom Field Types

Basically a Custom Field Type is a Java class that implements the *org.jbpm.formModeler.core.fieldTypes.CustomFieldType* interface and is packaged inside inside a jar file that is placed on the Application Server classpath or inside the application War.

Lets take a look at *org.jbpm.formModeler.core.fieldTypes.CustomFieldType*:

```
package org.jbpm.formModeler.core.fieldTypes;

import java.util.Locale;
```

```
import java.util.Map;

/**
* Definition interface for custom fields
*/
public interface CustomFieldType {
    /**
    * This method returns a text definition for the custom type. This text will be
    * @param locale                The current user locale
    * @return                      A String that describes the field type on the sp
    */
    public String getDescription(Locale locale);

    /**
    * This method returns a string that contains the HTML code that will be used to
    * shown on screen
    * @param value                 The current field value
    * @param fieldName             The field name
    * @param namespace             The unique id for the rendered form, it should b
* @param required          Determines if the field is required or not
    * @param readonly              Determines if the field must be shown on read or
    * @param params                A list of configuration params that can be set c
    * @return                      The HTML that will be used to show the field val
    */
    public String getShowHTML(Object value, String fieldName, String namespace, boo

    /**
    * This method returns a String that contains the HTML code that will show the i
    * @param value                 The current field value
    * @param fieldName             The field name
    * @param namespace             The unique id for the rendered form, it should b
* @param required          Determines if the field is required or not
    * @param readonly              Determines if the field must be shown on read or
    * @param params                A list of configuration params that can be set c
    * @return                      The HTML code that will be used to show the inpu
    */
    public String getInputHTML(Object value, String fieldName, String namespace, bo

    /**
    * This method is used to obtain the field value from the submitted values.
    * @param requestParameters     A Map containing the request parameters for the
    * @param requestFiles          A Map containing the java.io.Files uploaded on t
    * @param fieldName             The field name
    * @param namespace             The unique id for the rendered form, it should b
 * @param previousValue       The previous value of the current field
* @param required          Determines if the field is required or not
    * @param readonly              Determines if the field must be shown on read or
    * @param params                A list of configuration params that can be set c
    * @return                      The value of the field based on the submitted fo
```

```
            */
            public Object getValue(Map requestParameters, Map requestFiles, String fieldNam
        }
```

As you can see this Interface defines the methods that determines how the field has to be shown on the screen for when the form is shown on insert(getInputHTML(...)) or readonly (getShowHTML(...)) mode. It also provides the method (getValue(...)) that reads the needed parameters from the request and to obtain the correct field value. Te returned value type must match with the type of the field added on the form. So (for example) you can create a File input that uploads a file to a server folder and saves a String with the storage path as the field value, so on your forms you can turn all the text compatible fields (Short Text, Long Text, Rich Text and Email) on Input File.

To see ho can it be done look at the example on *https://github.com/droolsjbpm/jbpm-form-modeler/tree/master/jbpm-form-modeler-sample-custom-types/jbpm-form-modeler-custom-file-type*.

Please note that this is just a sample and it only should be used with learning purposes.

2.3.3.1.2. Configuring and using Custom Field Types

Now let's see how to use and configure and use a Custom Field type. Following the example on the previous chapter, we have created a File Input type and we have it already installed on our application. So now we are going to create a new form and add a Short Text property and turn it into a File Input and edit the field properties changing the Field Type from *Short text* to *Custom field*.

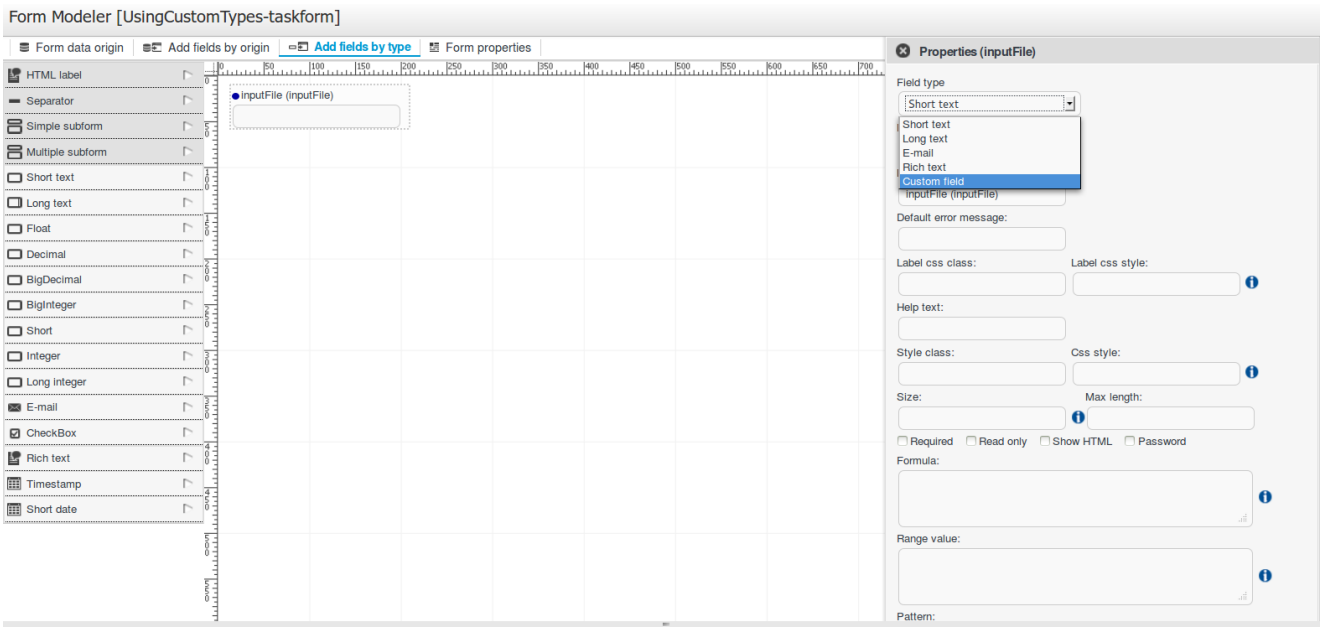


Figure 2.37.

Note

Changing a field type to *Custom field*.

After changing the field type a new set of properties will appear:

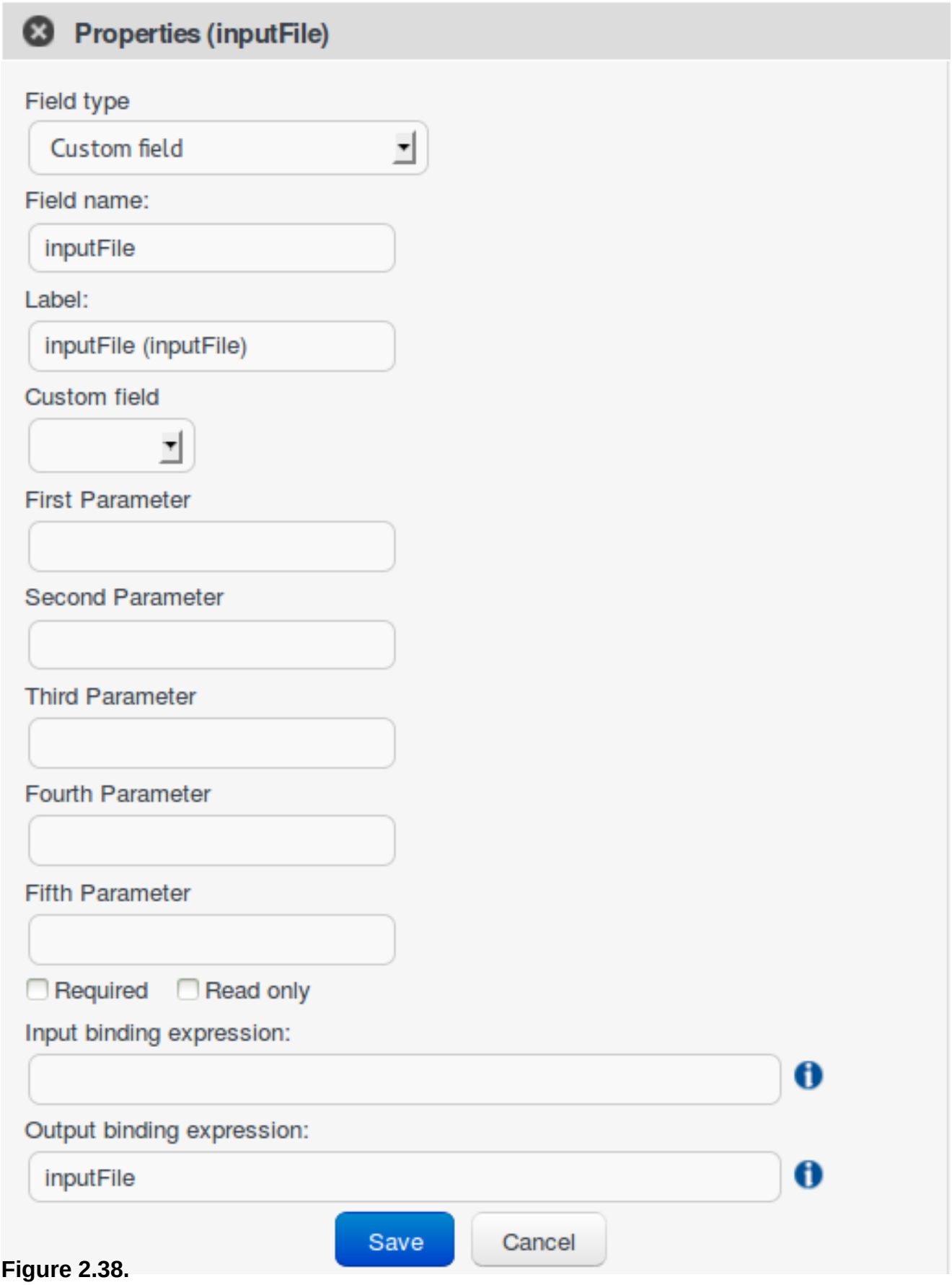

Properties (inputFile)
Field type
Custom field
Field name:
inputFile
Label:
inputFile (inputFile)
Custom field
First Parameter
Second Parameter
Third Parameter
Fourth Parameter
Fifth Parameter
Required
Read only
Input binding expression:
Output binding expression:
inputFile
Save
Cancel

Figure 2.38.

Note

The properties are:

Table 2.6.

Field type	Can change the field type to other compatible field types
Field Name	Will be used as identifier in formulas calculation
Label	The text that will be shown as field label
Custom field	A list containing all the Custom Field Types available on the platform
First parameter	A String parameter that can be user to pass custom configuration neede by the Custom Field Type implementation
Second parameter	A String parameter that can be user to pass custom configuration neede by the Custom Field Type implementation
Third parameter	A String parameter that can be user to pass custom configuration neede by the Custom Field Type implementation
Fourth parameter	A String parameter that can be user to pass custom configuration neede by the Custom Field Type implementation
Fifth parameter	A String parameter that can be user to pass custom configuration neede by the Custom Field Type implementation
Required	Indicates if it’s mandatory to fill this field.
Read Only	When this check is on, the field will be used only for read
Input binding expression	This expression defines the link between field and process task input variable. It will be used in runtime to set the field value with that task input variable data.
Output binding expression	This expression defines the link between field and process task output

		variable. It will be used in runtime to set that task output variable.

So opening the Custom field select box we'll be able to select the *File Input* from the available custom types:

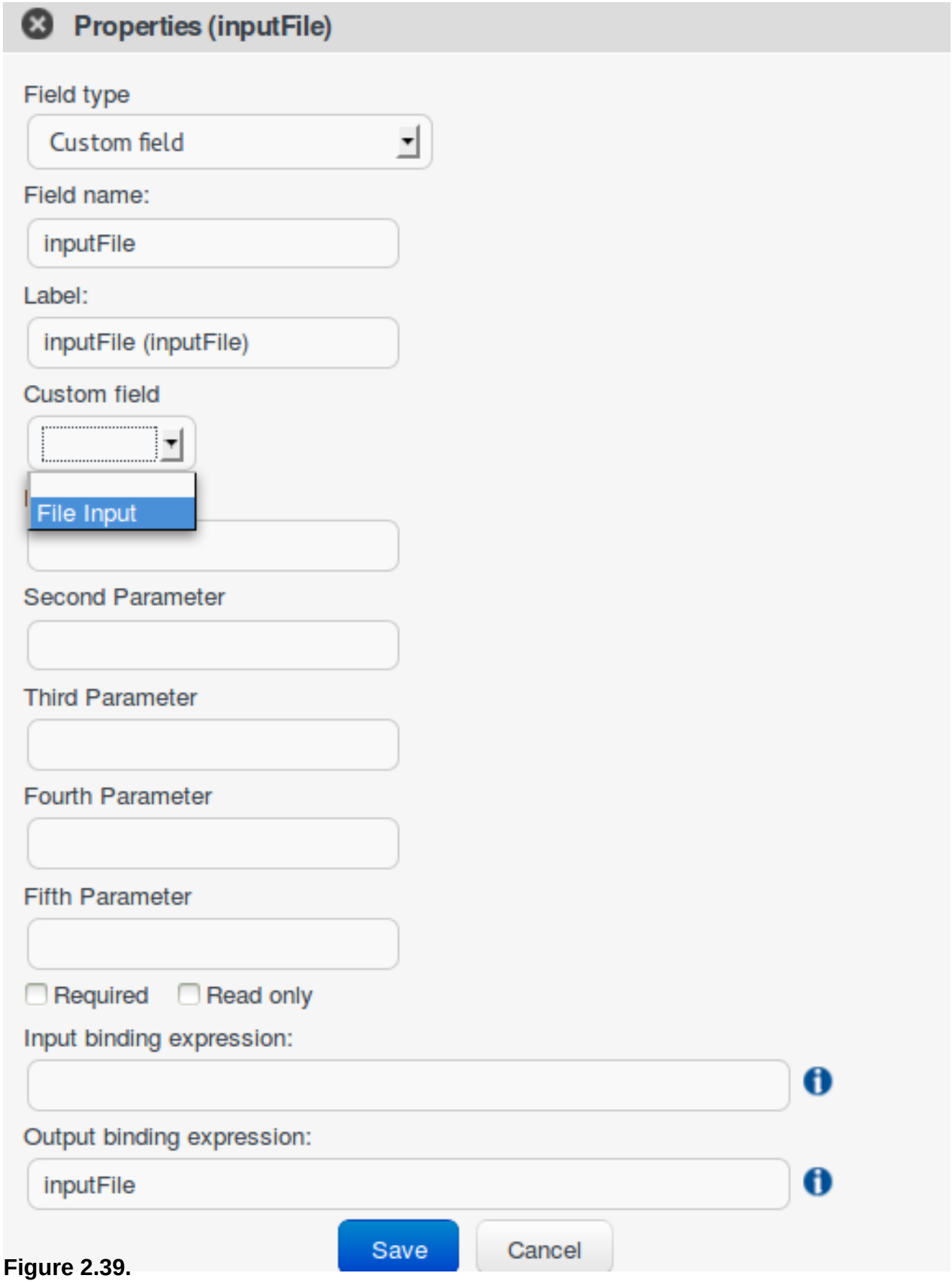

Properties (inputFile)
Field type
Custom field
Field name:
inputFile
Label:
inputFile (inputFile)
Custom field
File Input
Second Parameter
Third Parameter
Fourth Parameter
Fifth Parameter
Required
Read only
Input binding expression:
Output binding expression:
inputFile
Save
Cancel

Figure 2.39.

Note

Available custom types

After selecting the *File Input* type on the list and saving the field properties the form will look like:

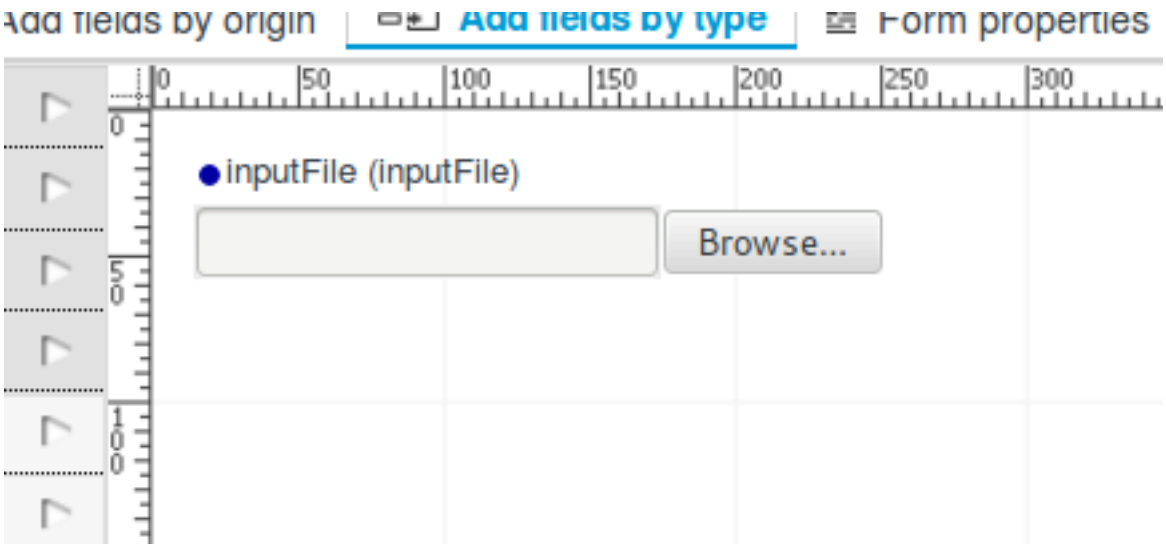


Figure 2.40.

If we build a simple process and configure a *Short text* to be shown as the sample *File Input*, if we build the project on runtime the field will behave uploading the choosen files to the server and allowing the user to download it like this:

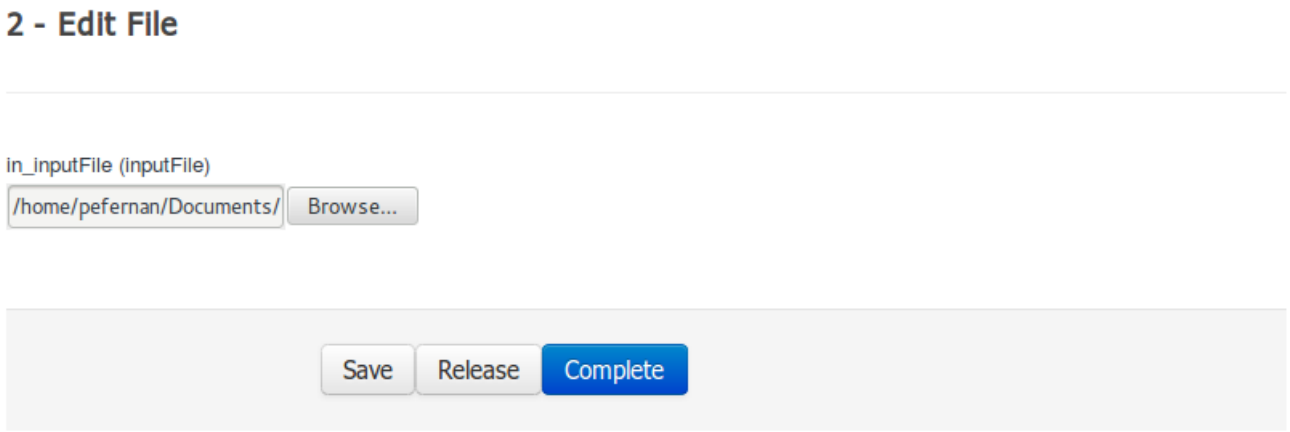


Figure 2.41.

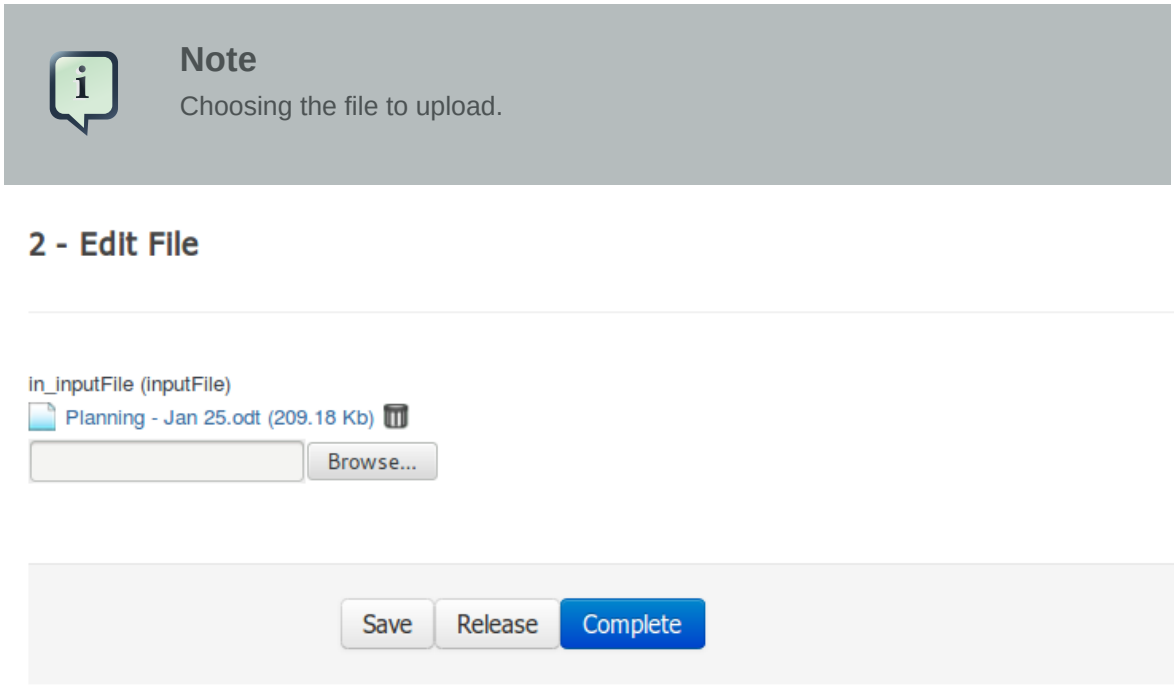
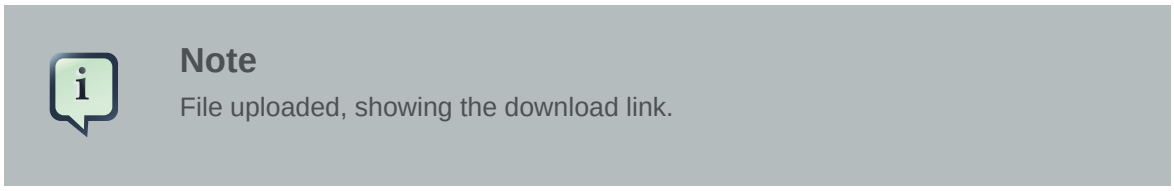


Figure 2.42.



If we take a look at what's the process variable value, we'll see that is storing a String with the file path stored in server.

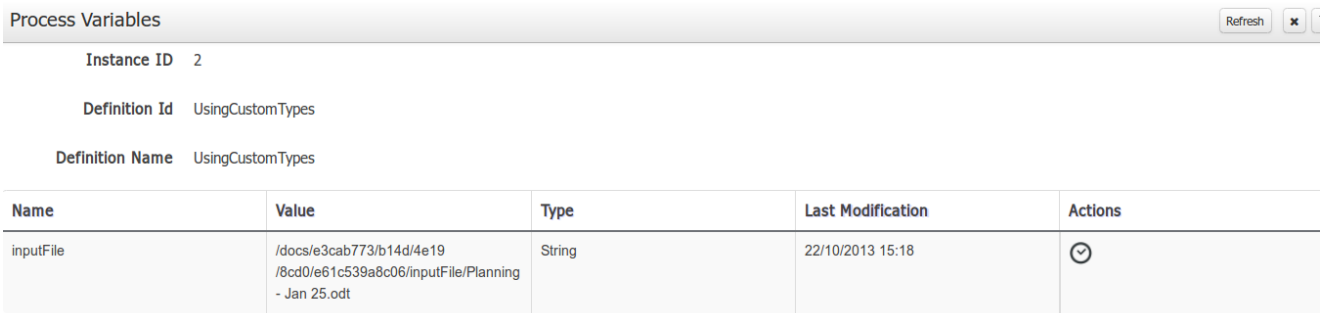


Figure 2.43.

Chapter 3. Data Modeller

3.1. What is Data Modeller

Neither the Drools platform (the rules engine) nor the jBPM platform (the business process engine) make sense if they do not have some kind of data to work with.

Typically, a business process analyst or data analyst will capture the requirements for a process or application and turn these into a formal set of interrelated data structures. The new Data Modeller tool provides an easy, straightforward and visual aid for building both logical and physical data models, without the need for advanced development skills or explicit coding, and transparently integrate and avail them for use by both platforms. Its main goals are to make data models into first class citizens in the process improvement cycle and allow for full process automation through the integrated use of data structures (and the forms that will be used to interact with them - see the chapter on the form modeller).

Note

The data modeller tool effectively replaces the former Drools Fact Modeller. The latter is therefore no longer available.

3.2. First steps to create a data model

By default, a data model is always constrained to the context of a project. For the purpose of this tutorial, we will assume that a correctly configured project already exists.

To start the creation of a data model inside a project, take the following steps:

1. From the home panel, select the authoring perspective

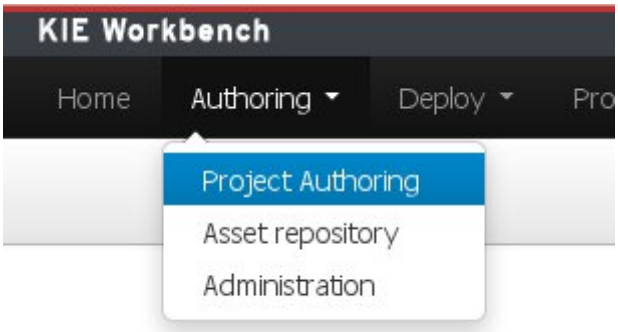


Figure 3.1. Go to authoring perspective

2. If not open already, start the Project Explorer panel

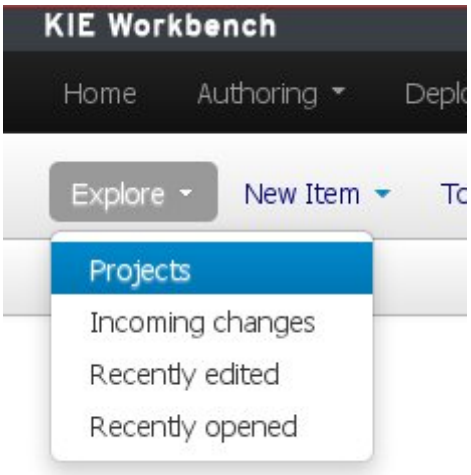


Figure 3.2. Open project explorer panel

3. From Project Explorer panel (the "Business" tab), select the organizational unit, repository, and the project the data model has to be created for. For this tutorial's example, the values "Tutorial", "Examples", and "Purchases" were respectively chosen

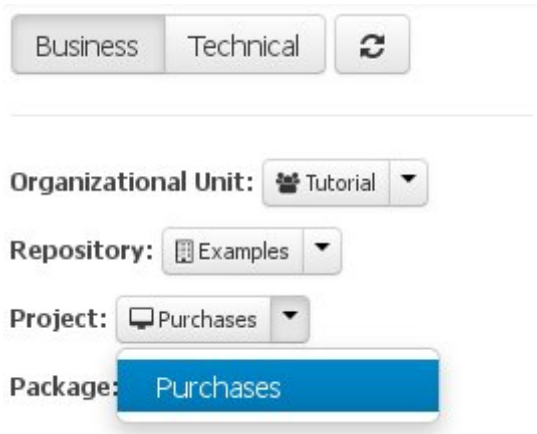


Figure 3.3. Choose project

4. Open the Data Modeller tool by clicking on the "Tools" authoring-menu entry, and selecting the "Data Modeller" option from the drop-down menu

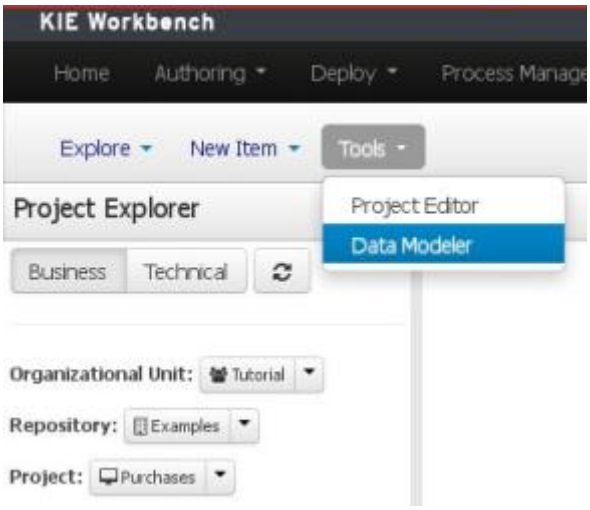


Figure 3.4. Open data modeller

This will start up the Data Modeller tool, which has the following general aspect:

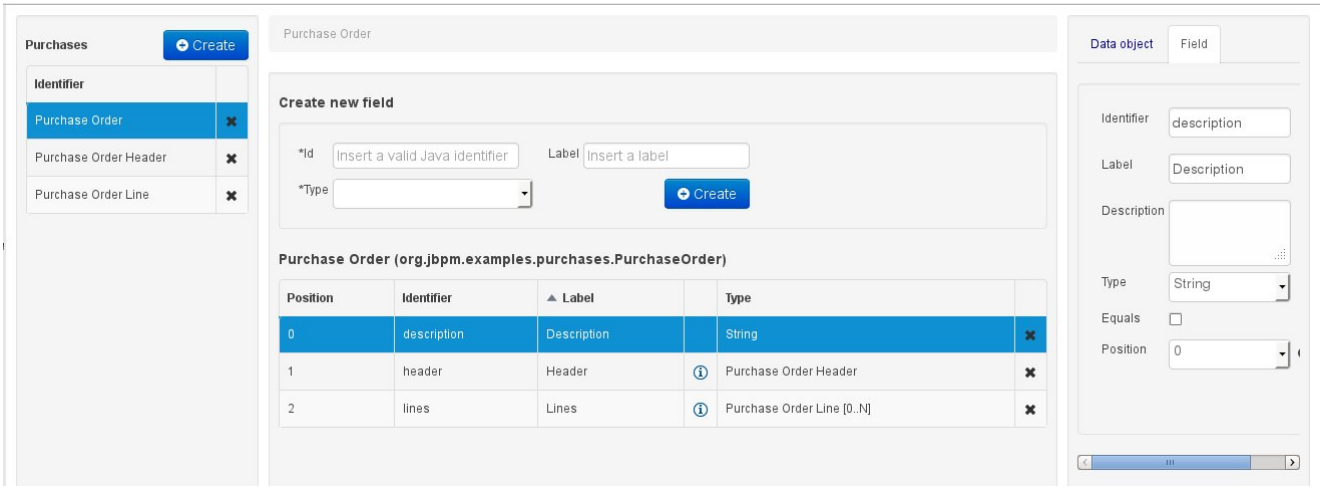


Figure 3.5. Data modeller overview

The Data Modeller panel is divided into the following sections:

- The leftmost "model browser" section, which shows a list of already existing data entities (if any are present, as in this example's case). Above the list the project's name and a button for new object creation are shown. Note that as soon as any changes are applied to the project, an '*' will be appended to the project's name to notify the user of the existence of non-persisted changes.

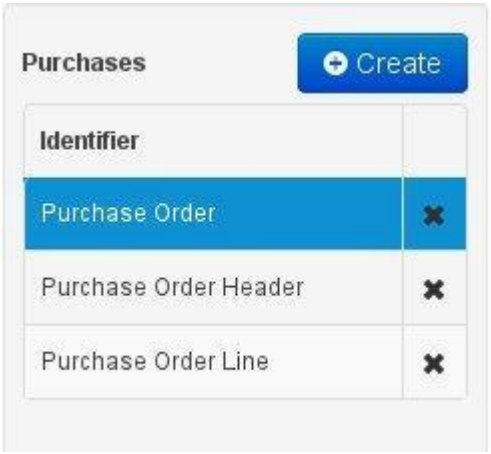


Figure 3.6. The data model browser

- The central section consists of three distinct parts:

  At the top, the "bread crumb widget": this is a navigational aid, which allows navigating back and forth through the data model, when accessing properties that themselves are model entities. The bread crumb trail shown in the image indicates that the object browser is currently visualizing the properties of an entity called "Purchase Order Line", which we accessed through another entity ("Purchase Order"), where it is defined as a field.

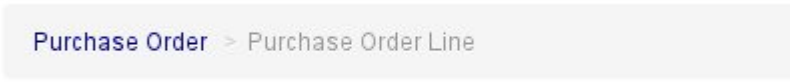


Figure 3.7. The bread crumb

the section beneath the bread crumb widget, is dedicated to the creation of new fields.

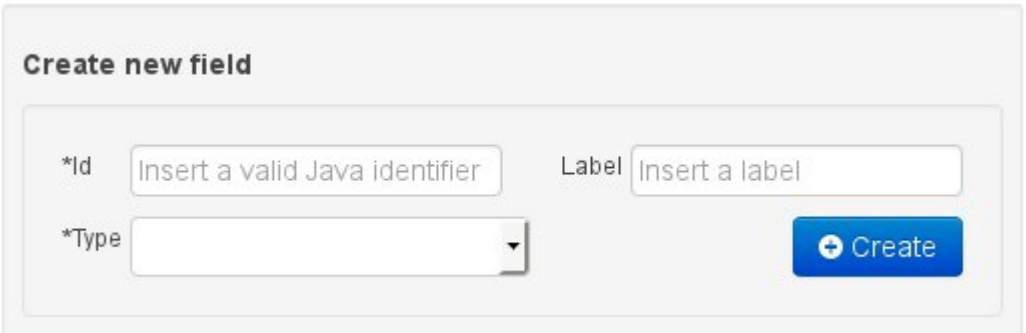


Figure 3.8. New field creation

the bottom section comprises the Entity's "field browser", which displays a list of the currently selected data object's (in the model browser) fields.

Figure 3.9. The entity field browser

- The "entity / field property editor". This is the rightmost section of the Data Modeller screen which visualizes a tabbed pane. The Data object tab allows the user to edit the properties of the currently selected entity in the model browser, whilst the Field tab enables edition of the properties of any of the currently selected object's fields.

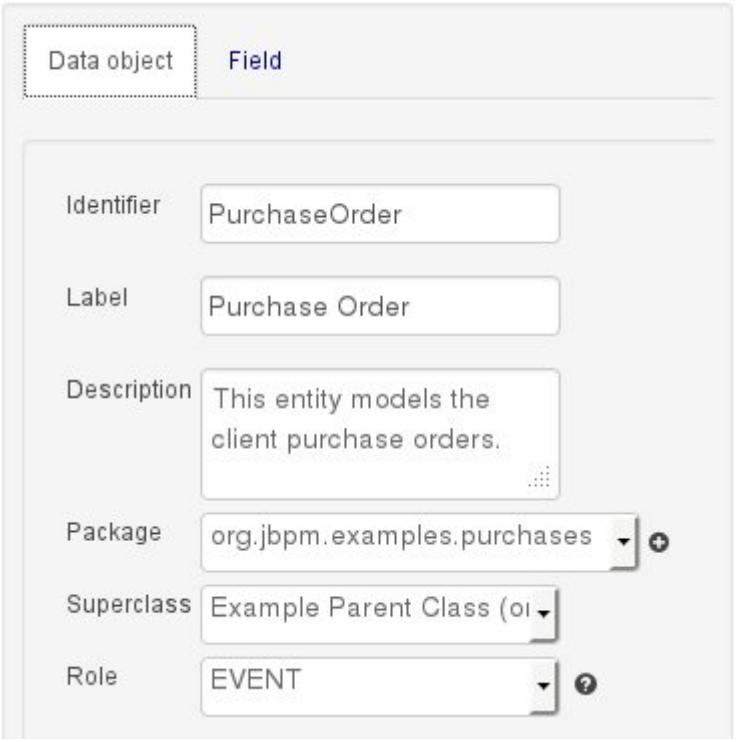


Figure 3.10. The entity/field property editor

3.3. Entities

A data model consists of data entities which are a logical representation of some real-world data. Such data entities have a fixed set of modeller (or application-owned) properties, such as its

internal identifier, a label, description, package etc. Besides those, an entity also has a variable set of user-defined fields, which are an abstraction of a real-world property of the type of data that this logical entity represents.

Creating a data entity can be achieved either by clicking the "Create" button in the model browser section (see fig. "The data model browser" above), or by clicking the one in the top data modeller menu:

Figure 3.11. Starting creation of an entity from the top menu

This will pop up the new object screen:

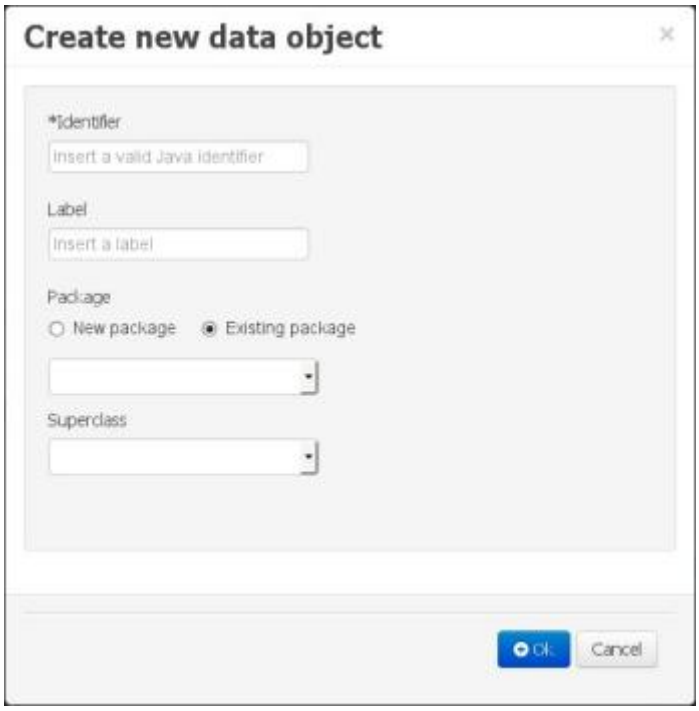


Figure 3.12. The new entity pop up screen

Some initial information needs to be provided before creating the new object:

- The object's internal identifier (mandatory). The value of this field must be unique per package, i.e. if the object's proposed identifier already exists in the selected package, an error message will be displayed.

- A label (optional): this field allows the user to define a user-friendly label for the data entity about to be created. This is purely conceptual info that has no further influence on how objects of this entity will be treated. If a label is defined, then this is how the entity will be displayed throughout the data modeller tool.
- A package (mandatory): a data entity must always be created within a package (or name space, in which this entity will be unique at a platform level). By default, the option for selecting an already existing package will be activated, in which case the corresponding drop-down shows all the packages that are currently defined. If a new package needs to be defined for this entity, then the "New package" option should be selected. In this case the new to be created package should be input into the corresponding text-field. The format for defining new packages is the same as the one for standard Java packages.
- A superclass (optional): this will indicate that this entity extends from another already existing one. Since the data modeller entities are translated into standard Java classes, indicating a superclass implies normal Java object extension at the generated-code level.

Once the user has provided at least the mandatory information, by pushing the "Ok" button at the bottom of the screen the new data entity will be created. It will be added to the model browser's entity listing.

It will also appear automatically selected, to make it easy for the user to complete the definition of the newly created entity, by completing the entity's properties in the Data Object Properties browser, or by adding new fields.

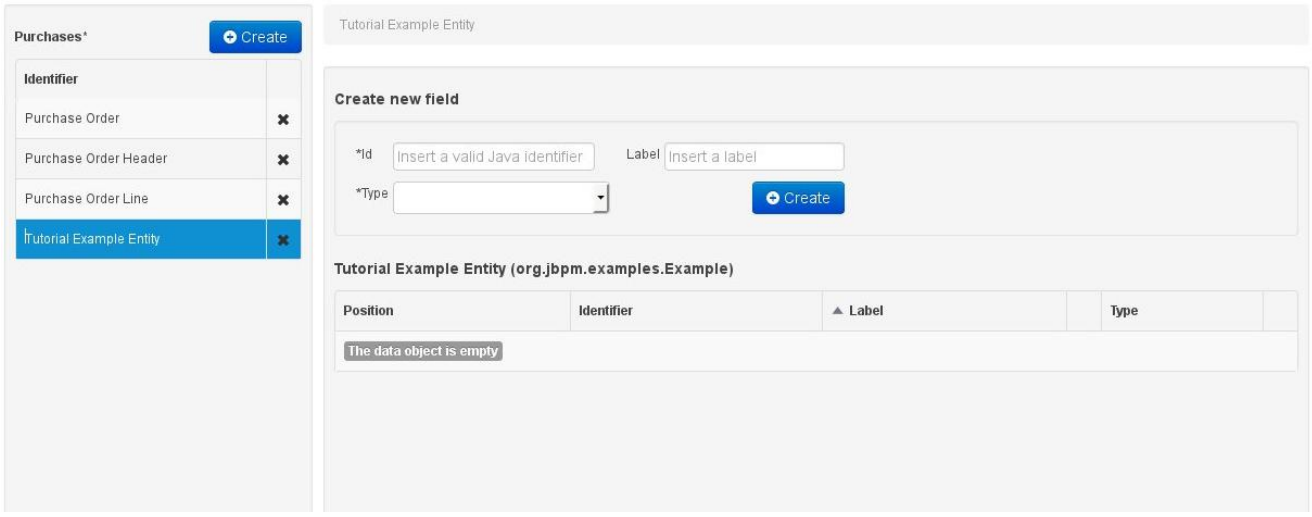


Figure 3.13. New entity has been created

Note

As can be seen in the above figure, after performing changes to the data model, the model name will appear with an '*' to alert the user of the existence of un-persisted changes to the model.

In the Data Modeller's object browsing section, an entity can be deleted by clicking upon the 'x' icon to the right of each entity. If an entity is being referenced from within another entity (as a field type), then the modeller tool will not allow it to be deleted, and an error message will appear on the screen.

3.4. Properties & relationships

Once the data entity has been created, it now has to be completed by adding user-defined properties to its definition. This can be achieved by providing the required information in the "Create new field" section (see fig. "New field creation"), and clicking on the "Create" button when finished. The following fields can (or must) be filled out:

- The field's internal identifier (mandatory). The value of this field must be unique per data entity, i.e. if the proposed identifier already exists within current entity, an error message will be displayed.
- A label (optional): as with the entity definition, the user can define a user-friendly label for the data entity field which is about to be created. This has no further implications on how fields from objects of this entity will be treated. If a label is defined, then this is how the field will be displayed throughout the data modeller tool.
- A field type (mandatory): each entity field needs to be assigned with a type.

  This type can be either of the following:

  1. A 'primitive' type: these include most of the object equivalents of the standard Java primitive types, such as Boolean, Short, Float, etc, as well as String, Date, BigDecimal and BigInteger.

Figure 3.14. Primitive field types

  2. An 'entity' type: any user defined entity automatically becomes a candidate to be defined as a field type of another entity, thus enabling the creation of relationships between entities. As

can be observed in the above figure, our recently defined 'Tutorial Example Entity' already appears in the types list and can be used as a field type, even for a field of itself. An entity type field can be created either in 'single' or in 'multiple' form, the latter implying that the field will be defined as a collection of this type, which will be indicated by the extension '[0..N]' in the type drop-down or in the entity fields table (as can be seen for the 'Lines' field of the 'Purchase Order' entity, for example).

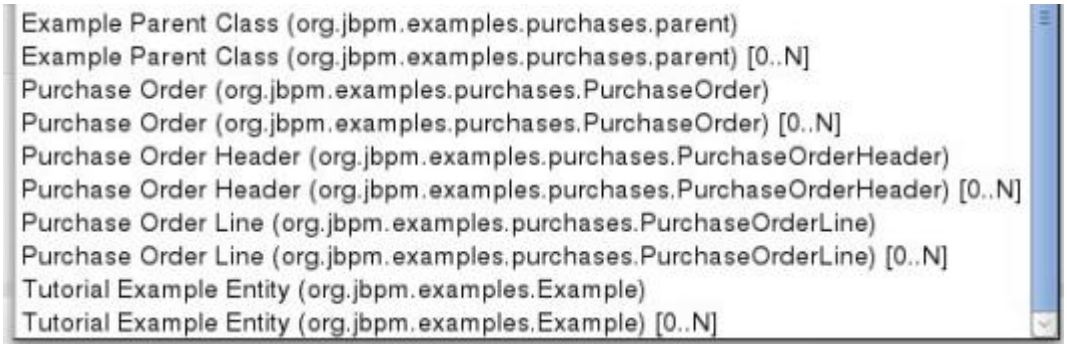


Figure 3.15. Entity field types

When finished introducing the initial information for a new field, clicking the 'Create' button will add the newly created field to the end of the entity's fields table below:

Figure 3.16. New field has been created

The new field will also automatically be selected in the entity's field list, and its properties will be shown in the Field tab of the Property editor. The latter facilitates completion of some additional properties of the new field by the user (see below).

At any time, any field (without restrictions) can be deleted from an entity definition by clicking on the corresponding 'x' icon in the entity's fields table.

3.5. Additional options

As stated before, both entities as well as entity fields require some of their initial properties to be set upon creation. These are by no means the only properties entities and fields have. Below we will give a detailed description of the additional entity and field properties.

3.5.1. Additional entity properties ("Data object tab")

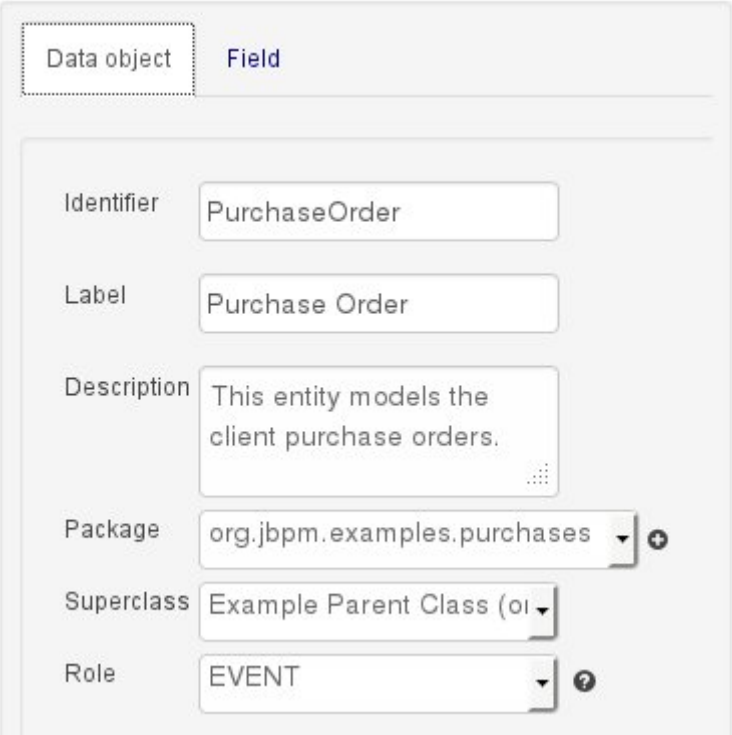


Figure 3.17. The entity's properties

- Description: this field allows the user to introduce some kind of description for the current entity, for documentation purposes only. As with the label property, this is conceptual information that will not influence the use or treatment of this entity or its instances in any way.
- Role: this property allows the assignment of a Role to the entity. The Role is a concept inherited from Drools Fusion, which for the time being only allows one possible value ("Event"). An entity that is designated with this value will be treated by the rules engine as an event type Fact (See Drools Fusion for more information on this matter).

3.5.2. Additional field properties ("Field tab")

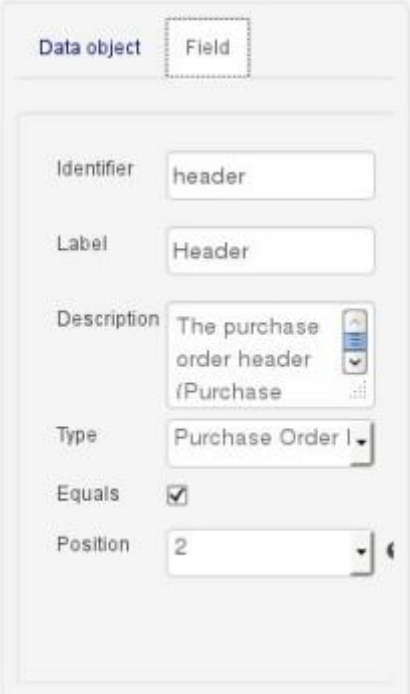


Figure 3.18. The entity's field properties

- Description: this field allows the user to introduce some kind of description for the current field, for documentation purposes only. As with the label property, this is conceptual information that will not influence the use or treatment of this entity or its instances in any way.
- Equals: checking this property for an entity field implies that it will be taken into account, at the code generation level, for the creation of both the equals() and hashCode() methods in the generated Java class. We will explain this in more detail in the following section.
- Position: this field requires a zero or positive integer. When set, this field will be interpreted by the Drools engine as a positional argument (see the section below and also the Drools documentation for more information on this subject).

3.6. Generate data model code.

The data model in itself is merely a visual tool that allows the user to define high-level data structures, for them to interact with the Drools Engine on the one hand, and the jBPM platform on the other. In order for this to become possible, these high-level visual structures have to be transformed into low-level artifacts that can effectively be consumed by these platforms. These artifacts are Java POJOs (Plain Old Java Objects), and they are generated every time the data model is saved, by pressing the "Save" button in the top Data Modeller Menu.

Figure 3.19. Save the data model from the top menu

At this time each entity that has been defined in the model will be translated into a Java class, according to the following transformation rules:

- The entity's identifier property will become the Java class's name. It therefore needs to be a valid Java identifier.
- The entity's package property becomes the Java class's package declaration.
- The entity's superclass property (if present) becomes the Java class's extension declaration.
- The entity's label and description properties will translate into the Java annotations "@org.kie.workbench.common.services.datamodeller.annotations.Label" and "@org.kie.workbench.common.services.datamodeller.annotations.Description", respectively. These annotations are merely a way of preserving the associated information, and as yet are not processed any further.
- The entity's role property (if present) will be translated into the "@org.kie.api.definition.type.Role" Java annotation, that *IS* interpreted by the application platform, in the sense that it marks this Java class as a Drools Event Fact-Type.

A standard Java default (or no parameter) constructor is generated, as well as a full parameter constructor, i.e. a constructor that accepts as parameters a value for each of the entity's user-defined fields.

The entity's user-defined fields are translated into Java class fields, each one of them with its own getter and setter method, according to the following transformation rules:

- The entity field's identifier will become the Java field identifier. It therefore needs to be a valid Java identifier.
- The entity field's type is directly translated into the Java class's field type. In case the entity field was declared to be multiple (i.e. '[0..N]'), then the generated field is of the "java.util.List" type.
- The equals property: when it is set for a specific field, then this class property will be annotated with the "@org.kie.api.definition.type.Key" annotation, which is interpreted by the Drools Engine, and it will 'participate' in the generated equals() method, which overwrites the equals() method of the Object class. The latter implies that if the field is a 'primitive' type, the equals method will simply compares its value with the value of the corresponding field in another

instance of the class. If the field is a sub-entity or a collection type, then the equals method will make a method-call to the equals method of the corresponding entity's Java class, or of the java.util.List standard Java class, respectively.

If the equals property is checked for *ANY* of the entity's user defined fields, then this also implies that in addition to the default generated constructors another constructor is generated, accepting as parameters all of the fields that were marked with Equals. Furthermore, generation of the equals() method also implies that also the Object class's hashCode() method is overwritten, in such a manner that it will call the hashCode() methods of the corresponding Java class types (be it 'primitive' or user-defined types) for all the fields that were marked with Equals in the Data Model.

- The position property: this field property is automatically set for all user-defined fields, starting from 0, and incrementing by 1 for each subsequent new field. However the user can freely changes the position among the fields. At code generation time this property is translated into the "@org.kie.api.definition.type.Position" annotation, which can be interpreted by the Drools Engine. Also, the established property order determines the order of the constructor parameters in the generated Java class.
- The entity's role property (if present) will be translated into the "@org.kie.api.definition.type.Role" Java annotation, that *IS* interpreted by the application platform, in the sense that it marks this Java class as a Drools Event Fact-Type.

As an example, the generated Java class code for the Purchase Order entity, corresponding to its definition as shown in the following figure purchase_example.jpg is visualized in the figure at the bottom of this chapter. Note that the two of the entity's fields, namely 'header' and 'lines' were marked with Equals, and have been assigned with the positions 2 and 1, respectively).

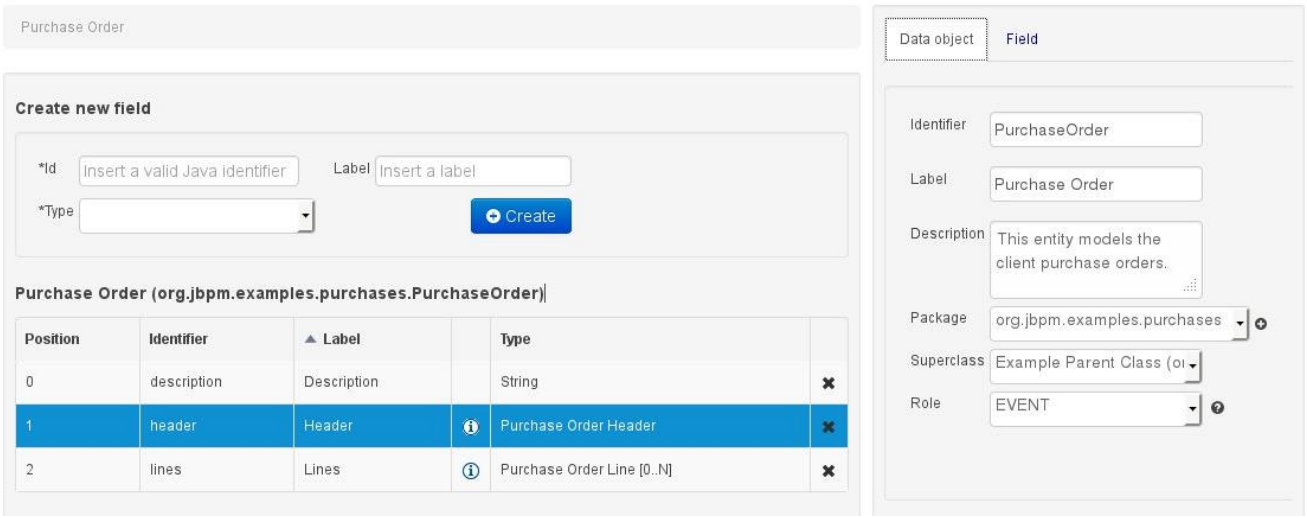


Figure 3.20. Purchase Order configuration

```
package org.jbpm.examples.purchases;

/**
 * This class was automatically generated by the data modeler tool.
 */
@org.kie.api.definition.type.Role(value =
  org.kie.api.definition.type.Role.Type.EVENT)
@org.kie.workbench.common.services.datamodeller.annotations.Label(value =
  "Purchase Order")
@org.kie.workbench.common.services.datamodeller.annotations.Description(value =
  "This entity models the client purchase orders.")
public class PurchaseOrder extends org.jbpm.examples.purchases.parent
                           implements java.io.Serializable {

static final long serialVersionUID = 1L;

  @org.kie.workbench.common.services.datamodeller.annotations.Label(value =
    "Description")
  @org.kie.api.definition.type.Position(value = 0)
  @org.kie.workbench.common.services.datamodeller.annotations.Description(value =
    "A description for this purchase order.")
  private java.lang.String description;

  @org.kie.workbench.common.services.datamodeller.annotations.Label(value =
    "Lines")
  @org.kie.api.definition.type.Position(value = 1)
  @org.kie.workbench.common.services.datamodeller.annotations.Description(value =
   "The purchase order items (collection of Purchase Order Line sub-entities).")
  @org.kie.api.definition.type.Key
  private java.util.List<org.jbpm.examples.purchases.PurchaseOrderLine> lines;

  @org.kie.workbench.common.services.datamodeller.annotations.Label(value =
    "Header")
  @org.kie.api.definition.type.Position(value = 2)
  @org.kie.workbench.common.services.datamodeller.annotations.Description(value =
    "The purchase order header (Purchase Order Header sub-entity).")
  @org.kie.api.definition.type.Key
  private org.jbpm.examples.purchases.PurchaseOrderHeader header;

  public PurchaseOrder() {}

  public PurchaseOrder(
    java.lang.String description,
    java.util.List<org.jbpm.examples.purchases.PurchaseOrderLine> lines,
    org.jbpm.examples.purchases.PurchaseOrderHeader header )
  {
    this.description = description;
    this.lines = lines;
    this.header = header;
```

```
}

public PurchaseOrder(
  java.util.List<org.jbpm.examples.purchases.PurchaseOrderLine> lines,
  org.jbpm.examples.purchases.PurchaseOrderHeader header )
{
  this.lines = lines;
  this.header = header;
}

public java.lang.String getDescription() {
  return this.description;
}

public void setDescription(  java.lang.String description ) {
  this.description = description;
}

public java.util.List<org.jbpm.examples.purchases.PurchaseOrderLine>
  getLines()
{
  return this.lines;
}

public void setLines(
  java.util.List<org.jbpm.examples.purchases.PurchaseOrderLine> lines )
{
  this.lines = lines;
}

public org.jbpm.examples.purchases.PurchaseOrderHeader getHeader() {
  return this.header;
}

public void setHeader(  org.jbpm.examples.purchases.PurchaseOrderHeader
  header )
{
  this.header = header;
}

@Override
public boolean equals(Object o) {
  if (this == o) return true;
  if (o == null || getClass() != o.getClass()) return false;
  org.jbpm.examples.purchases.PurchaseOrder that =
    (org.jbpm.examples.purchases.PurchaseOrder)o;
  if (lines != null ? !lines.equals(that.lines) : that.lines != null)
    return false;
  if (header != null ? !header.equals(that.header) : that.header != null)
```

```
      return false;
    return true;
  }

  @Override
  public int hashCode() {
    int result = 17;
    result = 13 * result + (lines != null ? lines.hashCode() : 0);
    result = 13 * result + (header != null ? header.hashCode() : 0);
    return result;
  }
}
```

3.7. Using external models

Using an external model means the ability to use a set for already defined POJOs in current project context. In order to make those POJOs available a dependency to the given JAR should be added. Once the dependency has been added the external POJOs can be referenced from current project data model.

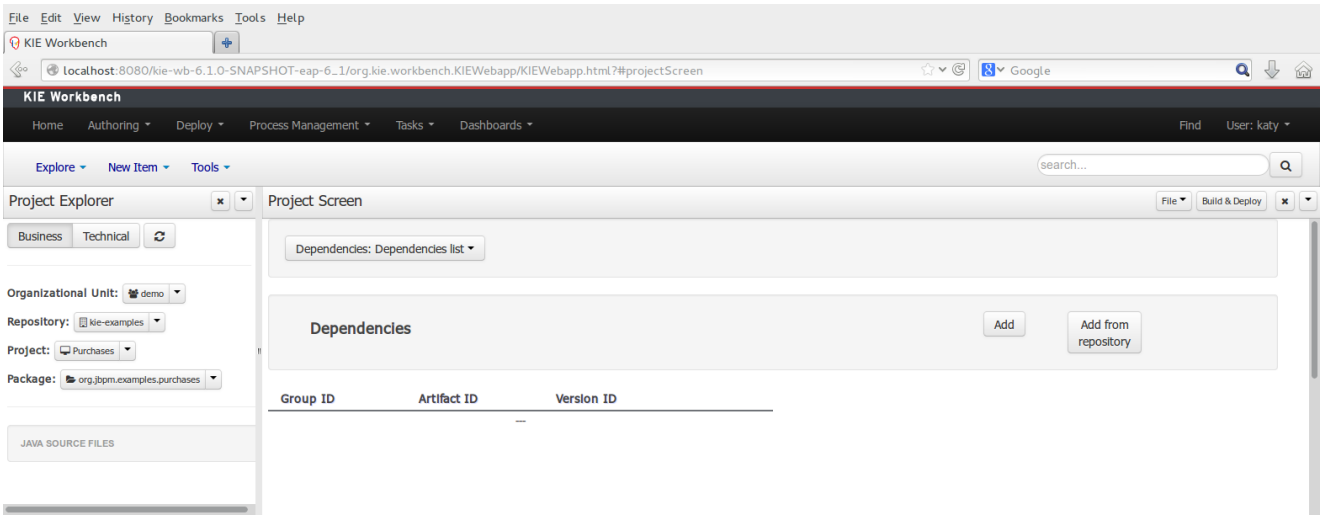
There are two ways to add a dependency to an external JAR file:

- Dependency to a JAR file already installed in current local M2 repository (typically associated the the user home).
- Dependency to a JAR file installed in current Kie Workbench/Drools Workbench "Guvnor M2 repository". (internal to the application)

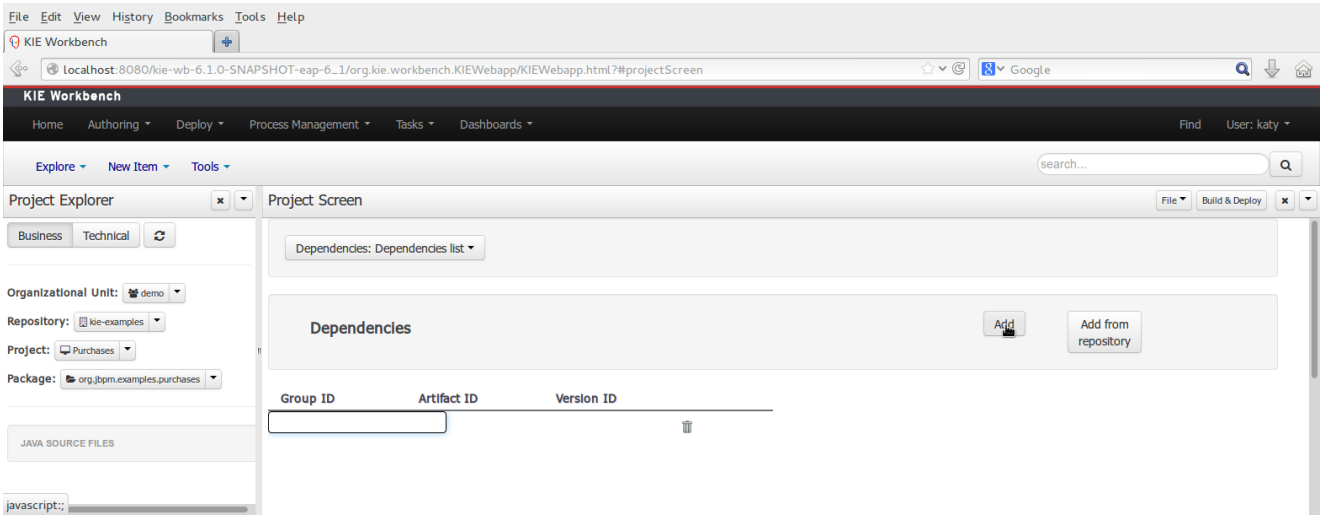
3.7.1. Dependency to a JAR file in local M2 repository

To add a dependency to a JAR file in local M2 repository follow this steps.

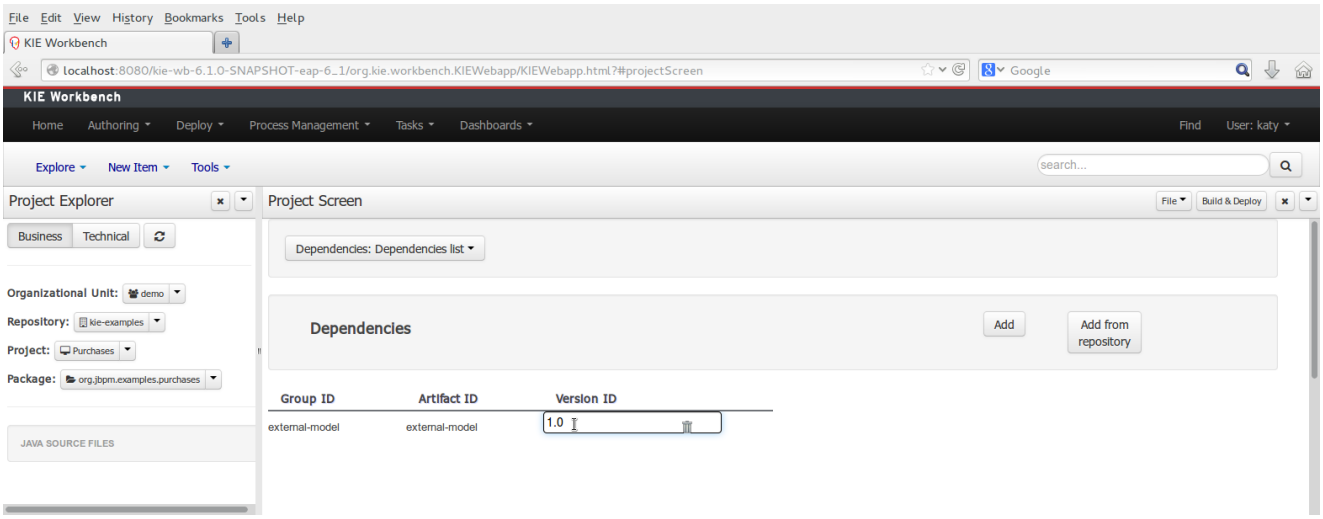
3.7.1.1. Open the Project Editor for current project and select the Dependencies view.



3.7.1.2. Click on the "Add" button to add a new dependency line.

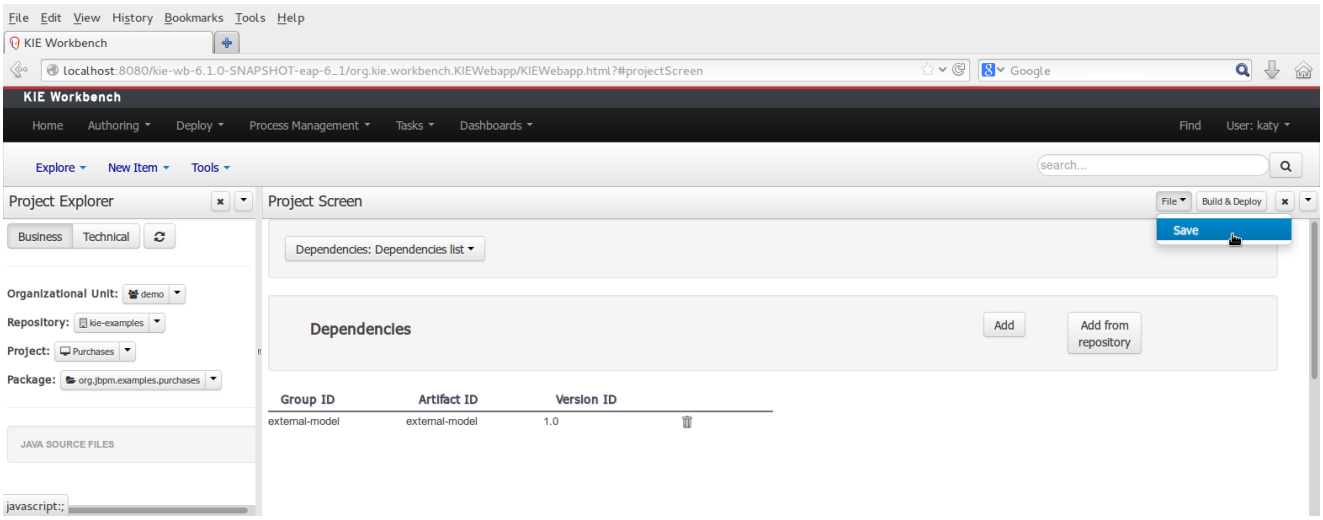


3.7.1.3. Complete the GAV for the JAR file already installed in local M2 repository.



3.7.1.4. Save the project to update its dependencies.

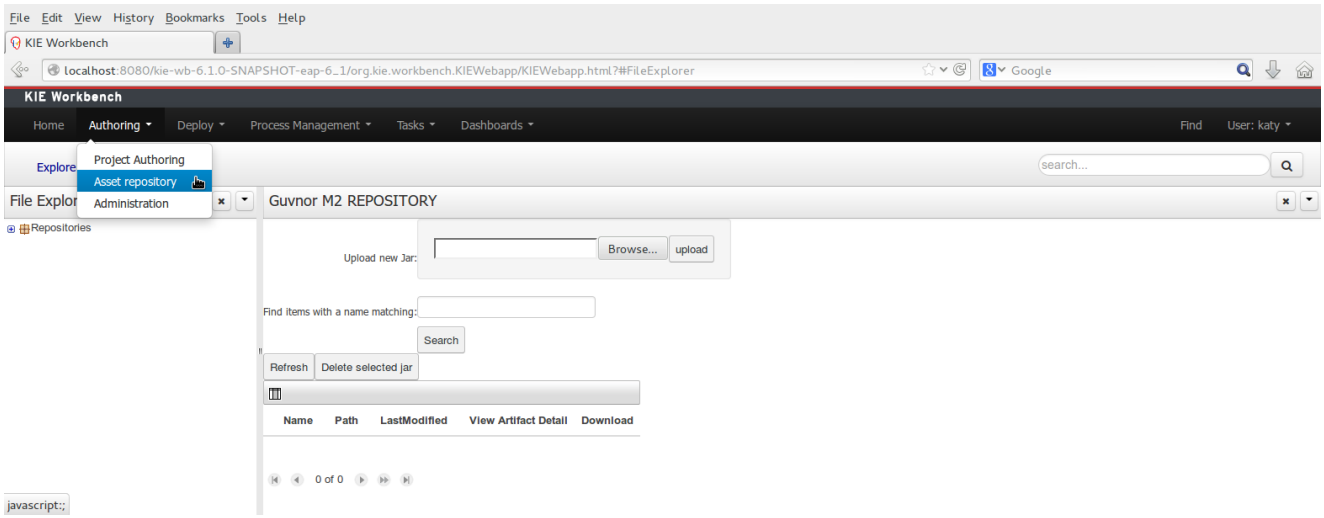
When project is saved the POJOs defined in the external file will be available.



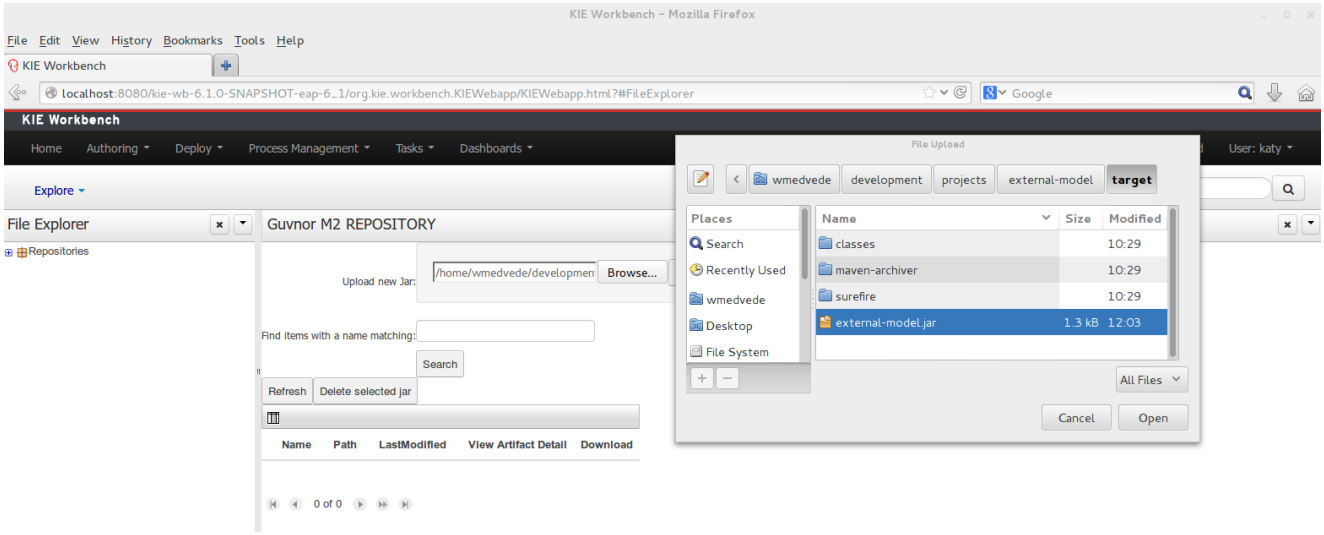
3.7.2. Dependency to a JAR file in current "Guvnor M2 repository".

To add a dependency to a JAR file in current "Guvnor M2 repository" follow this steps.

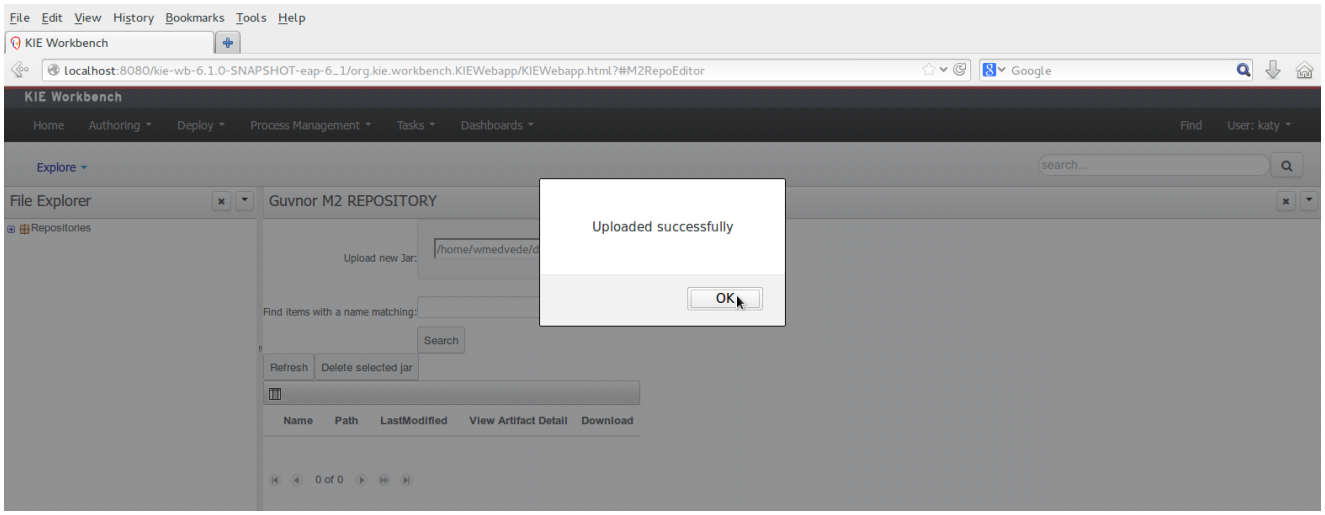
3.7.2.1. Open the Guvnor M2 Repository editor.



3.7.2.2. Browse your local file system and select the JAR file to be uploaded using the Browse button.

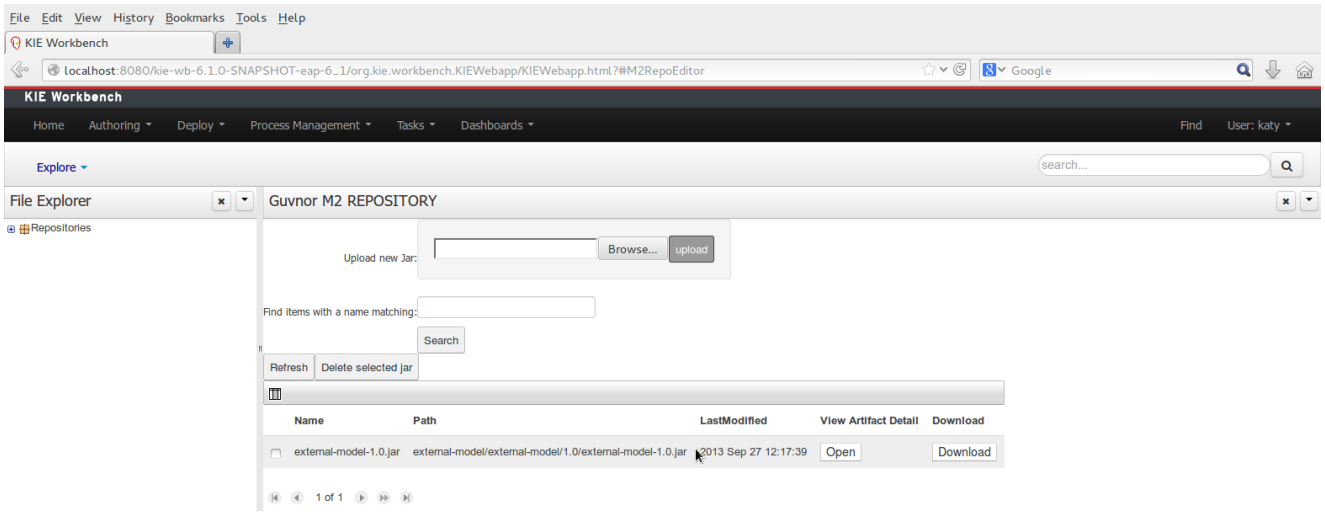


3.7.2.3. Upload the file using the Upload button.



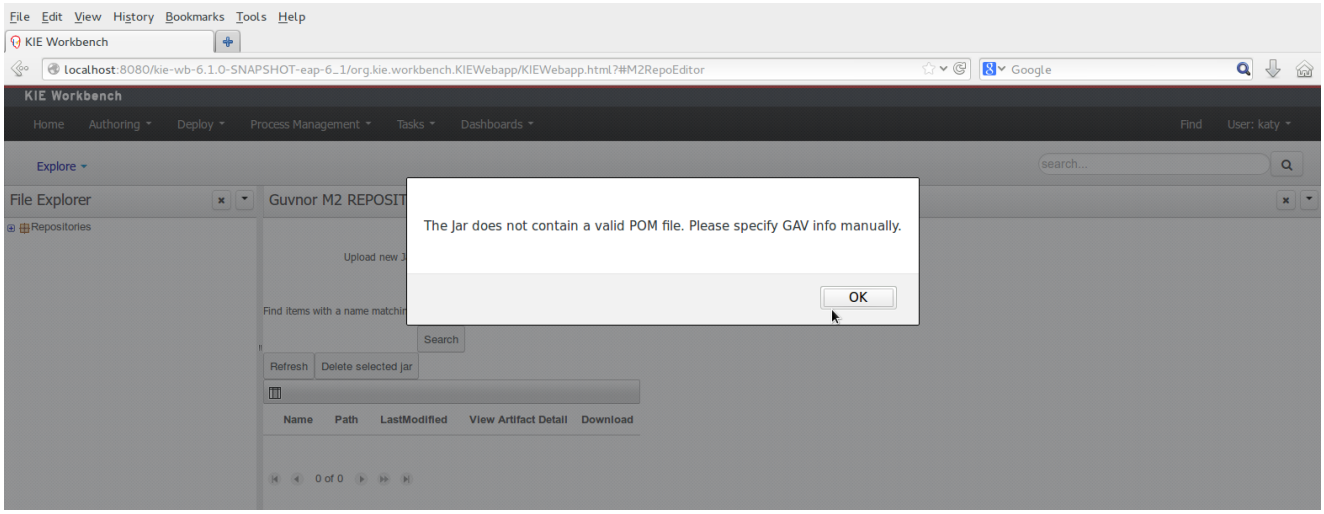
3.7.2.4. Guvnor M2 repository files.

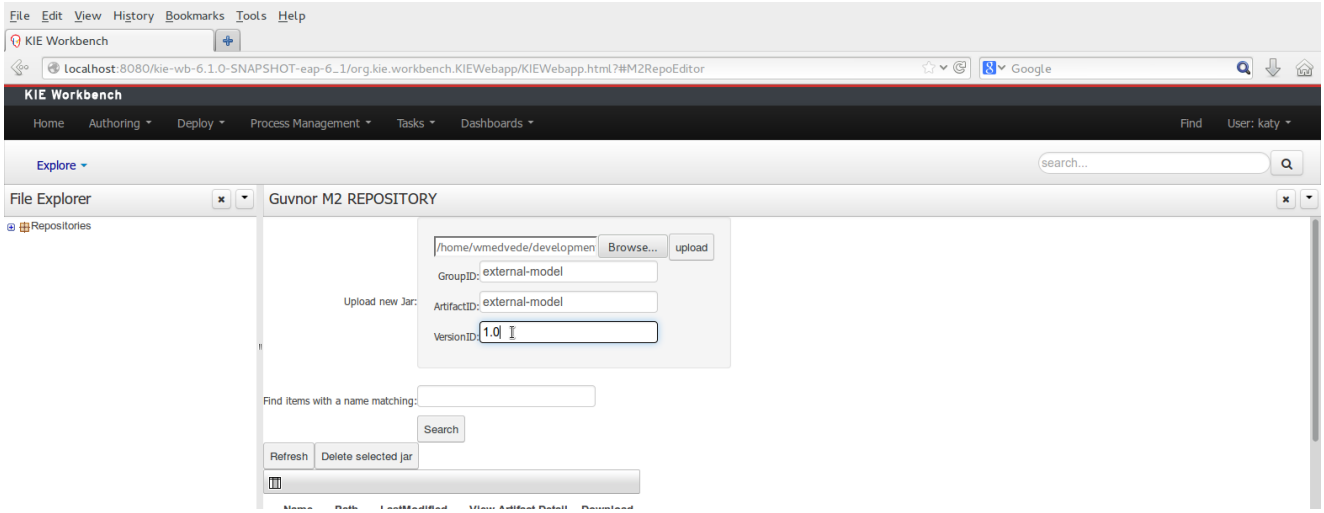
Once the file has been loaded it will be displayed in the repository files list.



3.7.2.5. Provide a GAV for the uploaded file (optional).

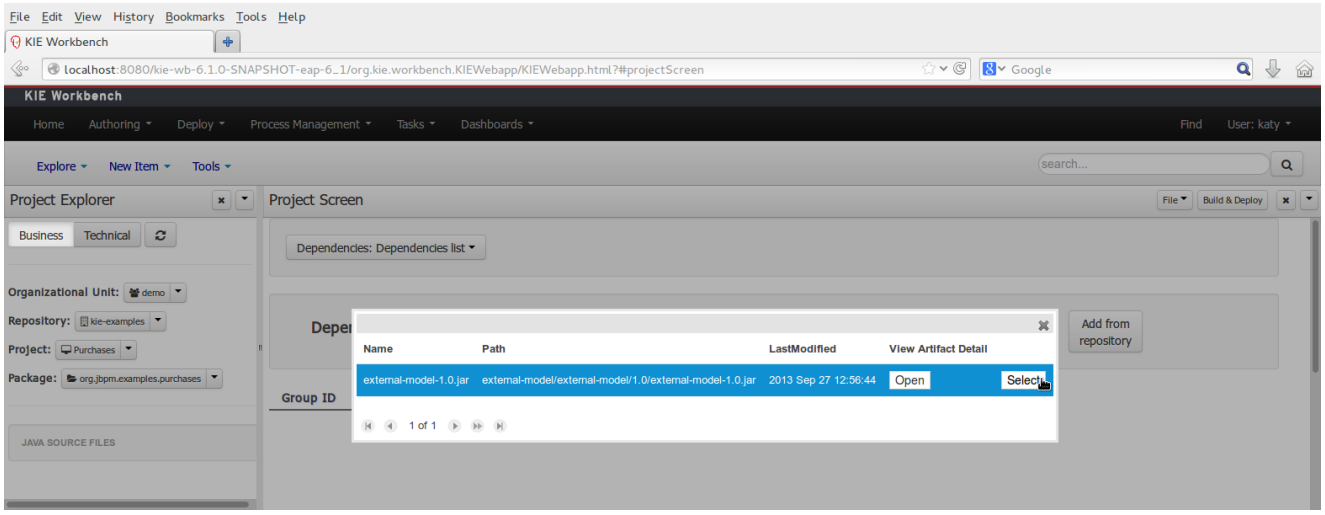
If the uploaded file is not a valid maven JAR (don't have a pom.xml file) the system will prompt the user in order to provide a GAV for the file to be installed.





3.7.2.6. Add dependency from repository.

Open the project editor (see bellow) and click on the "Add from repository" button to open the JAR selector to see all the installed JAR files in current "Guvnor M2 repository". When the desired file is selected the project should be saved in order to make the new dependency available.



3.7.3. Using the external objects

When a dependency to an external JAR has been set, the external POJOs can be used in the context of current project data model in the following ways:

- External POJOs can be extended by current model data objects.
- External POJOs can be used as field types for current model data objects.

The following screenshot shows how external objects are prefixed with the string " -ext- " in order to be quickly identified.