Living Systems® Process Suite

Academy

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

Academy

This guide teaches you how to the develop a custom application for your business using LSPS:

- In the first part, we briefly introduce you to the ideas behind BPMN.
- The second part deals with basic LSPS concepts.
- In the third part, we introduce develop a simple custom application.

This guide assumes you have through the `Quickstart`.

Proceed to introduction to BPMN.

Chapter 2

Note on BPMN

Before we can get our hands dirty, let's introduce the BPMN standard and its goal: In the past, IT infrastructure relied on a model where it was *humans* who drove the business process and only the business data was stored, while any knowledge on how to work with the data, the *business process*, remained with humans.

For example, when a client wanted to open a bank account, they filled out an application form and a bank employee performed and overlooked the processing of the application. Hence errors could have sneaked and the processing could have been flawed due to human error easily.

With BPMN, you can capture such *business processes*. This BPMN representation of business processes provides common grounds for business people on the one side and programmers on the other: unlike code, the business people can understand the visualization, and check and analyze the process and programmers can work directly with the process, execute, test and fine-tune it.

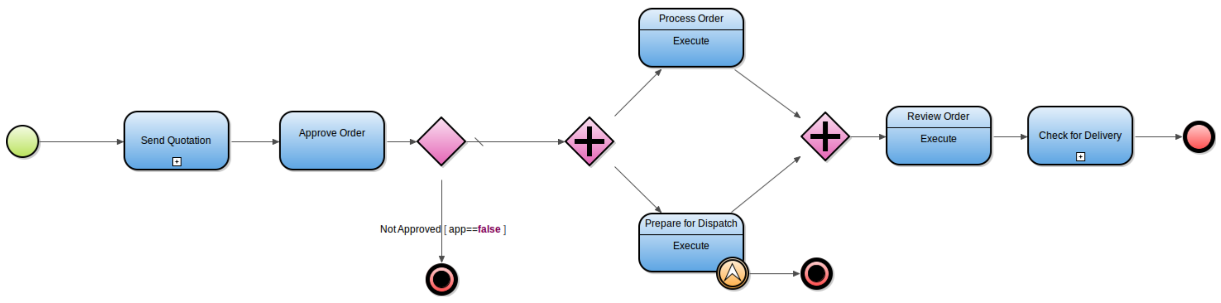


Figure 2.1 Process in BPMN notation

You can find information on BPMN and GO-BPMN, conservative extension of BPMN, as well as related mechanisms in the `GO-BPMN Modeling Language` guide.

Proceed to Introduction to LSPS.

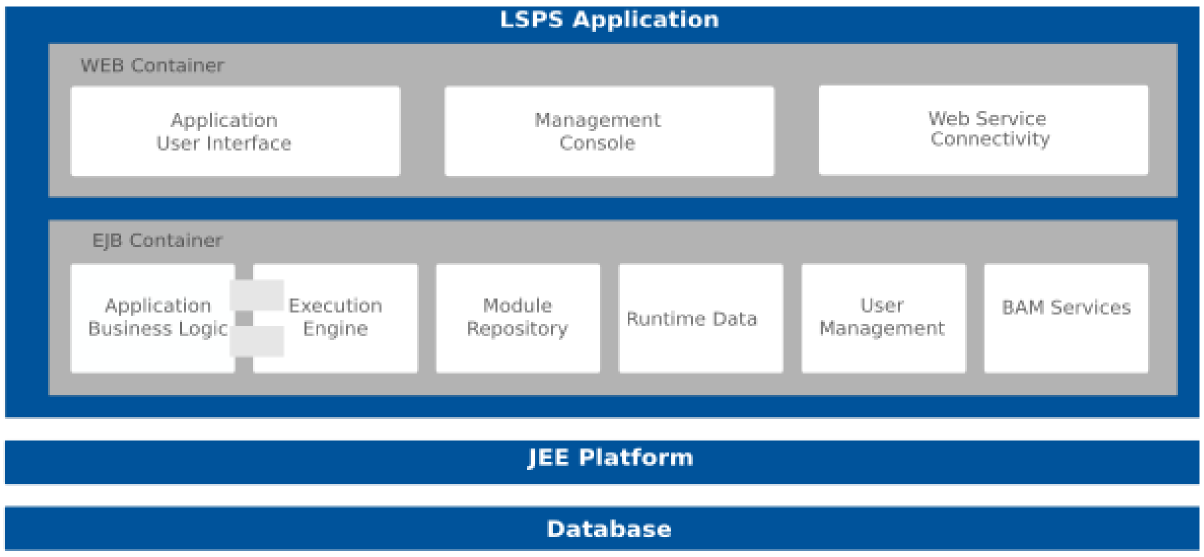
Chapter 3

Introduction to LSPS

LSPS is a set of tools for development of a customized BPMN solution, that is, create and execute business models, interact with then, and manage the execution and runtime data from a thick or a thin client.

With LSPS, you gain additional capabilities, such as, support for GO-BPMN, (extension of BPMN by goal-focused processes), role-based access control, graphical front-end page definitions, data type modeling support, along with access to API to build an entire solution.

The core of LSPS is the *LSPS Application*, a standard JEE enterprise application. It allows you to execute your models, manage users and their access to data, distribute work, communicates with external systems and users, etc.



You will design the models of your business processes using the **Process Design Suite**. Apart from being the development tool for models, it is also a thick client for management and administration of the *LSPS Application*.

When it comes to interactions with the *LSPS Application*, you have also other options:

- Users who are process participants access the data relevant to them through the web application called the **Application User Interface**. In the application, they can enter input for the processes.

- Administrators can use the command-line or web console called the **Management Console**. Both allow management of resources of the LSPS Application, such as, model upload, model run, user management, etc.

LSPS comes in two editions:

- **Cloud Edition** with the *Process Design Suite* that connects to the Cloud LSPS Server

You can design and run models, and access the web applications. However, you cannot customize the *LSPS Application*.

- **Enterprise Edition** comes with the *Process Design Suite* that can connect to any LSPS Server, tools for development of a customized *LSPS Application*, and resources for deployment and deployment management, including the LSPS Application EAR, CLI console, and database migration scripts.

Before you continue to the next chapter, `install PDS`.

You will now introduce the basic LSPS concepts and mechanisms:

- Expression Language, the dedicated language used in models,
- Models with the BPMN workflows,
- Business Data for business data,
- To-Dos and Documents, the means of communication with users,
- Goal Hierarchies of GO-BBPMN, the extension of BPMN that allows you to separates what to do from the ways it can be done.

Then we will create a simple model.

Proceed to Expression Language.

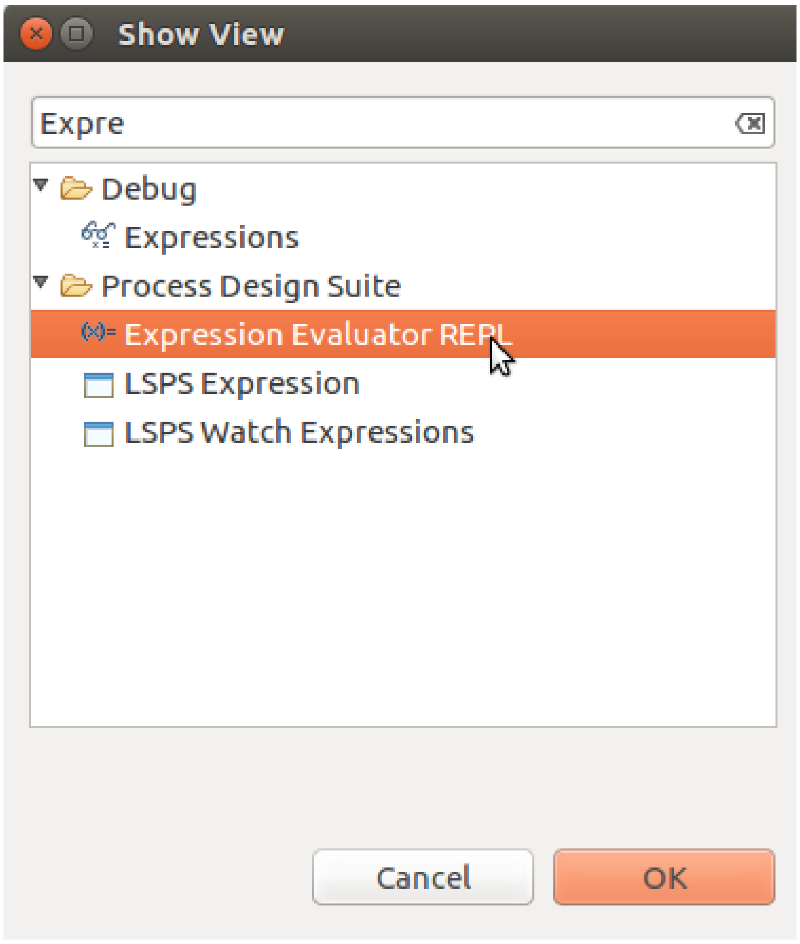
3.1 Expression Language

Business models created in LSPS use the LSPS Expression Language, a statically typed language with support for functional and object-oriented programming, to define value of properties, conditions, variable, assignments, etc.

The basic unit in the language is an expression, a piece of code with operators, literals, and keywords of the language. Each expression returns a single value, for example:

- `3+2` returns `5`;
- `def String var := "hello"` returns `"hello"`, the value of the right side of the expression;

To get the idea, evaluate expressions in the **Expression Evaluator REPL** view: To display it in your PDS, go to **Window** > **Show View** > **Other** and in the displayed dialog search for the view.



Enter an expression into the view and click **Evaluate** or press CTRL+Enter. To go through previously entered expressions, press the arrow-up key and to use auto-completion, press CTRL+Space.

This is all fine but an expression is a little bit too simple to accommodate all the logic that we might need to define say element properties: That is why definitions of properties, assignments, etc. can hold an expression block– a grouping of expressions. In REPL, you are working within such an expression block. To clear the block, click **Clear Output and Variables** button.

In a block, you can chain expressions with a semicolon `;` (the last expression does no require a semicolon). This allows you to use the expression variables: such variables must have a unique name within the given block. Mind also that they are visible within the block and its child blocks, while remaining invisible to any parent blocks. The return value of an expression block is the value returned by its last expression (comments are marked with `//` and are ignored):

Expression blocks can make use of expression variables: to declare such a variable, we use the `def` keyword with the syntax `def <TYPE> <VARIABLE_NAME>`).

```
//creates an expression variable logMessage
//and assigns it the value "hello":
def String logMessage := "hello ";
//appends "world!" to the logMessage vvariable:
logMessage + "world!"
//returns `"hello world!"`:
```

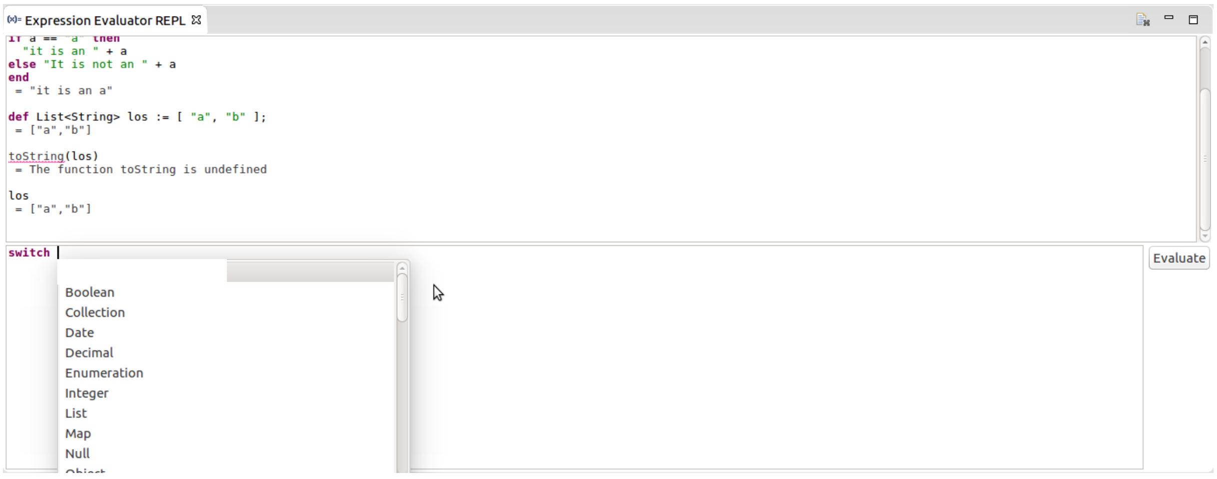
You can create an expression block explicitly with the `begin` and `end` keywords.

This expression block returns `3`:

```
begin
   def Integer var := 3;
   begin
    var := 5
   end;
  var;
end
```

while this expression block fails to return a value:

```
 begin
   def Integer var := 3;
   begin
    var := 5
   end;
  var;
end;
var;
```



```
if a == "a" then
  "it is an " + a
else "It is not an " + a
end
 = "it is an a"

def List<String> los := [ "a", "b" ];
 = ["a","b"]

toString(los)
 = The function toString is undefined

los
 = ["a","b"]
```

```
switch
```

Note that the REPL view supports calls to Standard Library resources, such as functions, data types, etc. For example, the call of the function `getYear()` returns the year from a date:

```
getYear(now()) //returns the current year
```

Note: When evaluating your expression in the Expression Evaluator REPL view, the expressions are evaluated outside of any model: under normal circumstances, on runtime, expressions are evaluated in a context of their model instance so they can access the data of the model instance, such as, when and by whom the instance was created. Therefore definitions from your resources and definitions that require execution such contexts, such as, `getCurrentPerson()` or `getCurrentTodo()`, are not allowed in REPL. If you want to know more about contexts, refer to `namespaces and contexts`.

Let's go through the operators and data types of the Expression Language:

- First, let's try some arithmetics:
  - `1 + 1`
  - `1 + 1.12`
  - `1/1.1`

  - `1%1.1`
  - `3*3`
  - `3/3`
  - `3**3`

- Let's try some concatenation:
  - `"Hello, " + "world!"`
  - `"Hello, number " + 1`
  - `"1" + 1`
  - `1 + "hello"`
    This results in an invalid expression since the + operator that has Integer and String as its input does not exist, while `"" + 1 + "hello"` will evaluate just fine.

- Let's try some comparing:
  - `"Anne" < "James"`
    Strings are compared lexicographically: A appears before J in alphabet.
  - `"James" like "J*"`
  - `"James" like "J???s"`
  - `"James" like "Jo*"`
  - `1 == 1,00`
  - `"Bob"<=>"Anne"`
  - `"Bob"<=>"John"`
  - `"Bob"<=>"Bob"`
  - `d'2015-12-24 20:00:00.000' < d'2017-12-24 20:00:00.000'`

- And now let's do some logic:
  - `true and false`
  - `true and true`
  - `true or false`
  - `false or true`
  - `true xor false`
  - `!true`

- Let's chain multiple expressions:
  - `"This will not return!";"Hello, " + #10 + "world!";`
  - `"This will not return!";"Hello, " + #10 + "world!"`
    Note that only the value of the last expression is returned.
  - `"This will not return!" "Hello, " + #10 + "world!"`: this is invalid.

- Create Collections with some items:
  - Sets: `{1, 2, 3}` and `{1, 1, 2, 3}`
  - Lists: `[1, 1, 2, 3]`
  - `[ 1 -> "Sunday", 2 -> "Monday"]` is a **Map**, more specifically *Map<Integer, String>*.
    Mind that Collections and Maps are immutable:

    ```
    def Set<Integer> mySet := { 1, 2, 3 };
    mySet.add(4);
    mySet
    ```

    The return value is `{ 1, 2, 3}`: The set with `4` was returned by `mySet.add(4)`, but the value has not been assigned to anything. To change the content of a Collection, create a new one and resign.

```
mySet := mySet.add(4)
 = {1,2,3,4}
```

- Evaluate the closure `{x:Integer -> "Integer" + x}`

  The closure has an Integer input parameter called `x` in the closure contexts: the closure a String: It is of type `{ Integer : String }`; However, it doesn't do much, since we cannot call it:

  - Create a variable that will hold the closure:

    ```
    def {Integer:String} closureVar := {x:Integer -> toString(x)};
    ```

  - Call the closure: `closureVar(5)`

    Important: It is recommended to always explicitly define the type of the closure inputs, in the example, we explicitly define that *x* is an *Integer* in `x:Integer`.

- Also type of value is a data type: `type(Integer)` returns the Integer Type, the Type being a data type.

- `&var` is a reference to a variable.

  ```
  //creates an expression variable:
  def String stringVar := "hello";
  &stringVar;
  *(&stringVar);
  ```

- Check types of objects:

  - with the `instanceof` operator of the Expression Language:

    ```
    "hello" instanceof String;
    ```

  - with the function `isInstance()` of the Standard Library:

    ```
    isInstance(3, Integer);
    ```

    The difference between the built-in and the function is that the built-in cannot take an expression as as its right side:

    ```
    def Type<Object> stringType := String;
    "hello" instanceof stringType;//does not evaluate
    isInstance("hello", stringType);
    ```

  Note: Along with a few other types, we have omitted the Record type, which is the parent of the Record types (Records): Records are complex data structure types that can contain multiple fields of non-complex data types (fields cannot be of a Record type) and be related to each other. They serve to represent your business data. While instances of Records are created in the Expression Language, it is not possible to create Record types themselves in the Expression Language since Records are modelled using the Modeling Language. You will model Records later.

Now, do the following:

1. Create a local variable called *closureVar* of type Closure, which takes a list of integers as its input parameter and returns a String.
2. Assign the variable a closure that concatenates the input list into a single String and returns the String.
3. Create a list with integers 1-100 and feed it to the closure variable.

The result should look something like this:

```
def {List<Integer>:String} closureVar;
closureVar := { myIntList:List<Integer> ->
     def String output;
     foreach Integer i in myIntList do
       output := output + i + ", "
     end;
     output
     };
def List<Integer> myIntList := 1..1000;
closureVar(myIntList)
```

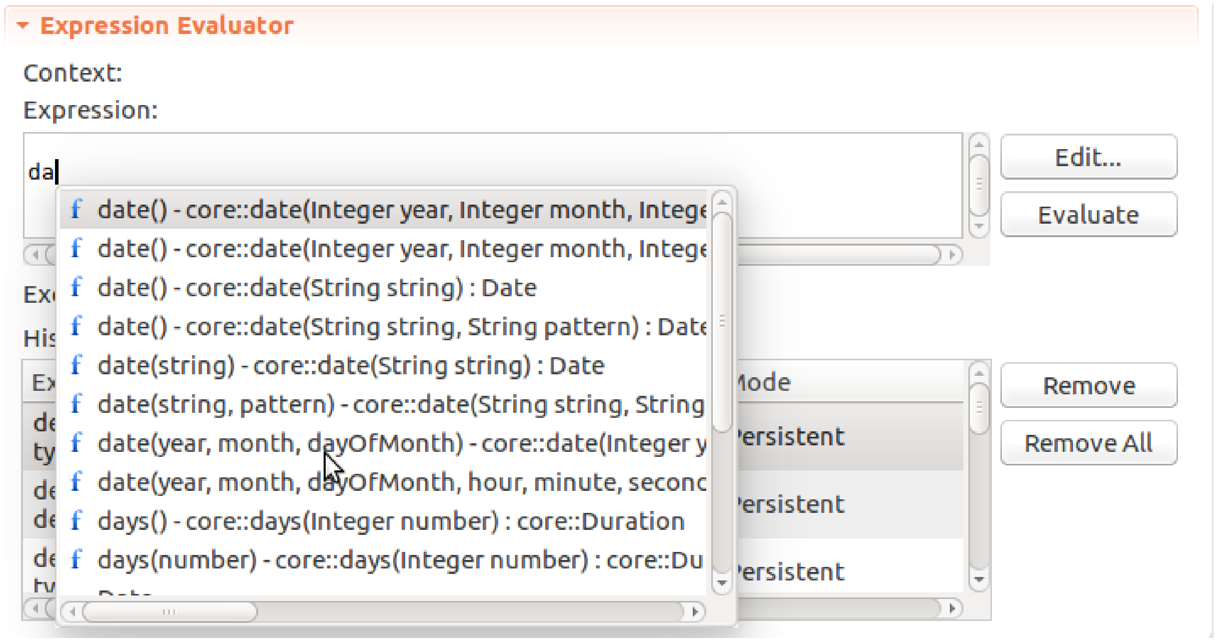
3.1.1 Function Call

A *function* is a set of instructions that can take arguments, perform the instructions and return a value: the return value will generally depend on the arguments.

Unlike closures, functions cannot be defined just in any expression block. You will define a custom function later. For now, we will use the functions provided by the Standard Library. To call functions, use the syntax `<FUNCTI↩ ON_CALL>(<ARG_1>, <ARG_>)`.

Let's try it out:

Creating a date value like `d'yyyy-MM-dd HH:mm:ss.SSS'` is quite cumbersome: luckily there is a function that will make your life much simpler: In the REPL view, start typing `date` and press Ctrl + space to display the auto-completion menu. Click on the `date` function you like and define its parameters so it returns the correct date value.


Expression Evaluator
Context:
Expression:
da
Edit...
Evaluate
Remove
Remove All
f date() - core::date(Integer year, Integer month, Intege
f date() - core::date(Integer year, Integer month, Intege
f date() - core::date(String string) : Date
f date() - core::date(String string, String pattern) : Date
f date(string) - core::date(String string) : Date
f date(string, pattern) - core::date(String string, String
f date(year, month, dayOfMonth) - core::date(Integer y
f date(year, month, dayOfMonth, hour, minute, seconc
f days() - core::days(Integer number) : core::Duration
f days(number) - core::days(Integer number) : core::Du
ersistent
ersistent
ersistent

3.1.1.1 Extension Method Call

If a function has one *required* parameter, it can be defined as an extension method: this means that you can call it with the syntax:

```
<ParameterObject>.<function_name>()
```

Hence, instead of writing `toString(myInteger)`, you can write `myInteger.toString()`, which is more convenient, since you can use autocompletion options of an Integer.

To find out, if a function of the Standard Library is an extension method, go to its declaration (click the function call while holding the CTRL key) and look for `@Extensionmethod` annotation or **Extension method** flag, depending on whether the function definition file uses the text or graphical editor.

Here, we check if the function definition has an Extension method flag and then rewrite the function call to an equivalent extension method call:

```
def List<Integer> myList := collect(1..10, {x:Integer -> x +2 });
(1..10).collect({x:Integer -> x +2 })
```

3.1.2 Type Hierarchy and Casting

Data types constitute a hierarchy: A type can be a subtype or a supertype of another type. If a type is a subtype of another type, it can be used instead of the supertype, but not vice versa. The strict typing prevents usage of incorrect types; for example, you cannot accidentally pass a Date value to a String variable unless you explicitly cast it to a String.

For the types of the language, the most generic data type is the Object type: all data types are subtypes of the Object type: when a variable is of the Object type, you can assign it a value of any type.

In a complex type, one complex type is a subtype of another complex type if their inner members are each other's subtypes: Map<KA, VA>; is subtype of Map<KB, VB>, if KA is a subtype of KB and VA is a subtype of VB.

Let's try it out:

In the REPL view, create an Object variable, for example, `def Object myObject`, and assign it a value of any type. To check the type of the value in the object, run `typeOf(myObject)`. The expression `def Object myObject := [1,2,3]; typeOf(myObject)`, returns type `List<Integer>`

Otherwise, the data type structure of the built-in data types is pretty flat:

- The Decimal type and Integer type with the Decimal type being the super type: Wherever you can use a Decimal value, you can use an Integer value.
- `Null` is a special data type with the sole value `null`: all other types are super types of the Null type so any data type can have the value `null`.
- The Collection type and, the Set and List types: Collection is abstract; it serves to allow you to decide whether something will be a Set or List later during execution.
- PropertyPath has Property as its subtype (A property path is a route to a Record field, such as, `Person.↩ name`).

Let's try it out; Evaluate the following:

- `def Collection<Object> myObject := [1,2,3]; typeOf(myObject)`
- `def Collection<Object> myObject := {1,2,3}; typeOf(myObject)`
- check the Null subtype:

  ```
  isSubtype(Null, String);
  isSubtype(type(Null), String);
  isSubtype(getType(null), Date);
  isInstance(null, String);
  ```

If you want to change the value type, typically from a supertype to a subtype, you can use the build-in cast mechanism. Let's try it out:

- `1 as Decimal`
- `1.00 as Integer`
- `1.12 as Integer`

Note that similarly to the case of `instanceof` vs `isInstance()`, there is operator `as` and the `cast()` function of the Standard Library. The `as` operator cannot take an expression as as its right side:

```
def Integer i := 1;
 = 1
~
typeOf(i)
 = Integer
~
123.00 as typeOf(i)
 = no viable alternative at input '('
~
cast(123.00, typeOf(i))
 = 123
```

3.1.3 Related Documentation

For details on the Expression Language, refer to the `Expression Language Guide`. For quick reference, use the `Expression Language Reference Card`.

Proceed to Models.

3.2 Models

To implement your *business process* in a model you will need to create a structure that will hold the model resources, allow their reuse, and keep them organized: All your resources must be created in or within a special directory called a GO-BPMN Project. Projects don't do much except holding the modules and some special resources (they are equivalent to Java Projects).

Projects hold modules and modules hold the logic and resources. Modules constitute your business models: They can contain processes, global variables, custom data types, queries, functions, etc. and *are reusable*: A module can import another module; once a module imports another modules, it can use its resources.

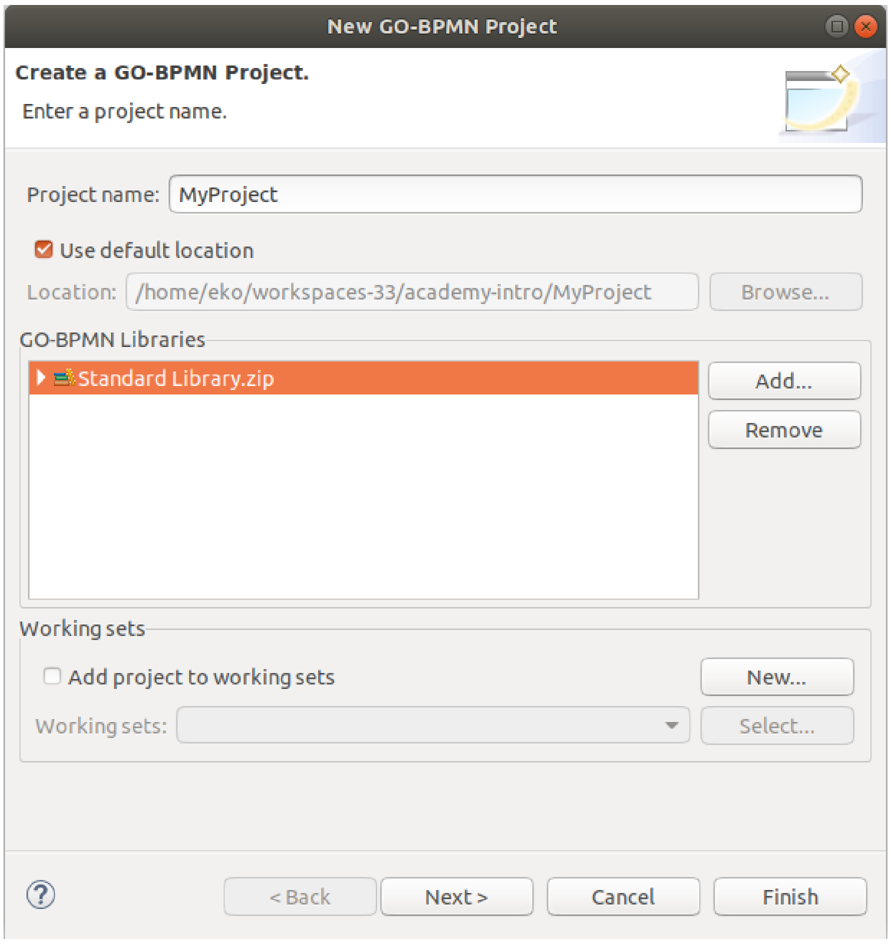
It is the modules that are uploaded to the server, while projects exist merely on design time in your workspace for your convenience.

Typically, at least one of the modules in your project is generally marked as executable: All resources of the module and its imported modules are referred to as a *model*. Once uploaded, you can create a model instance by running the module: A model instance then hold the context of the executable modules and all its module imports.

Let's create a GO-BPMN project with an executable module and create a model instance over it in the PDS Embedded Server:

1. In the GO-BPMN Explorer, right-click into empty space and go to **New** > **GO-BPMN Project**.

   (a) In the GO-BPMN Explorer, right-click into empty space and go to **New** > **GO-BPMN Project**. In the displayed dialog below the Location field is the list of libraries that will be imported into the project. Only the Standard Library is required.


New GO-BPMN Project

Create a GO-BPMN Project.

Enter a project name.

Project name: MyProject

☑ Use default location

Location: /home/eko/workspaces-33/academy-intro/MyProject

GO-BPMN Libraries

Standard Library.zip

Working sets

☐ Add project to working sets

Working sets:

(b) Name the project and click **Finish**.

> **Note:** If you created a new project to organize your resource, the project is not able to access resources from another project: to allow a project to use resources of another project, `reference it from the project`..

2. Create a module:

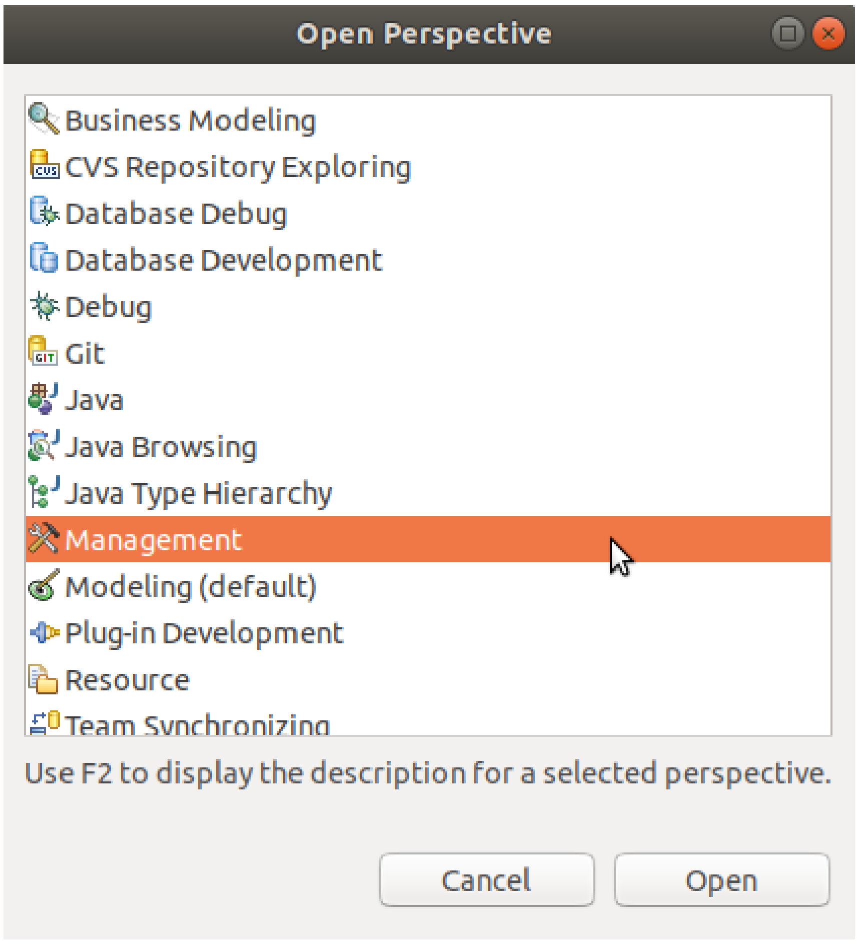
   (a) Right-click the project, go to New > GO-BPMN Module.

   (b) In the dialog box, enter the module name *main* and click **Finish**.

3. Install and connect to the PDS Embedded Server:

   (a) Click the **Start Embedded Server** button in the main toolbar: This will generate and start a local server with LSPS Application and connect your PDS to it.

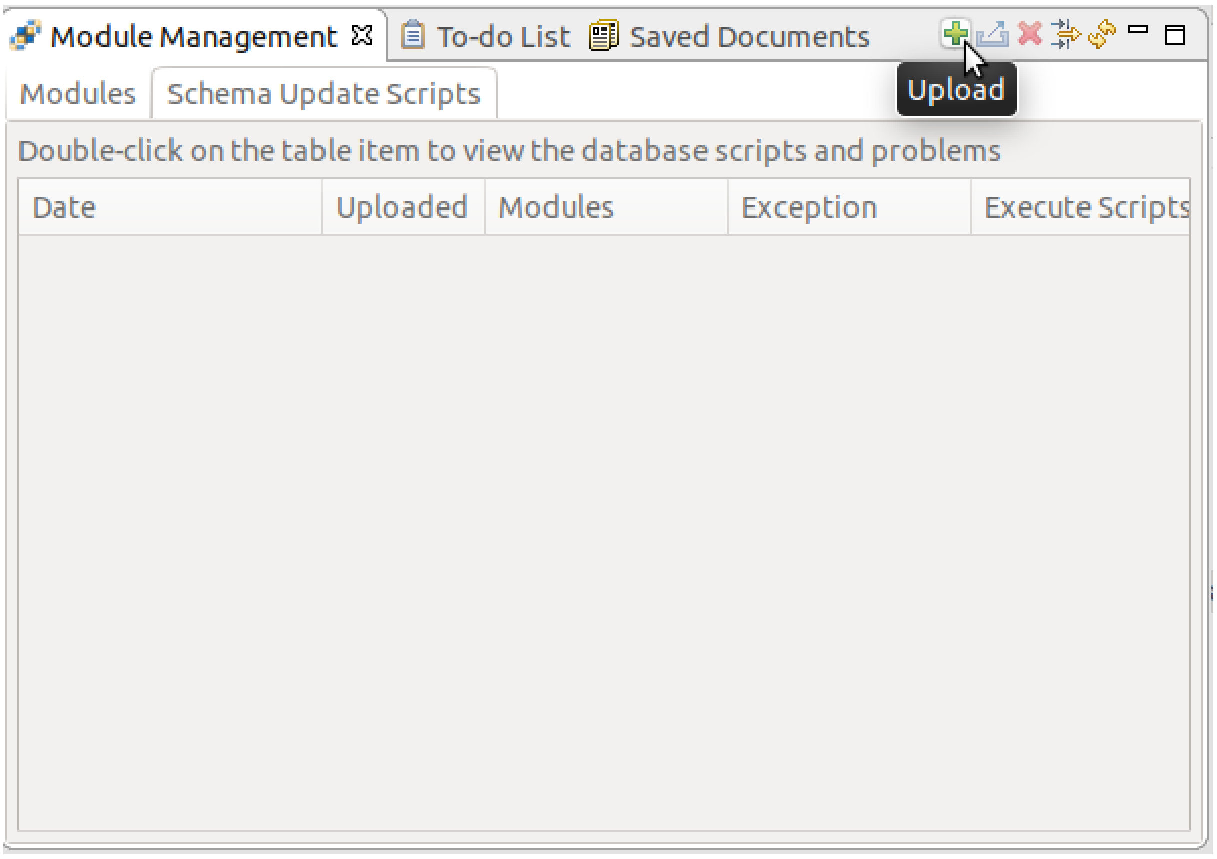
   (b) Switch to the *Management* perspective: click the **Open Perspective** button in the upper-right corner and select Management in the popup.



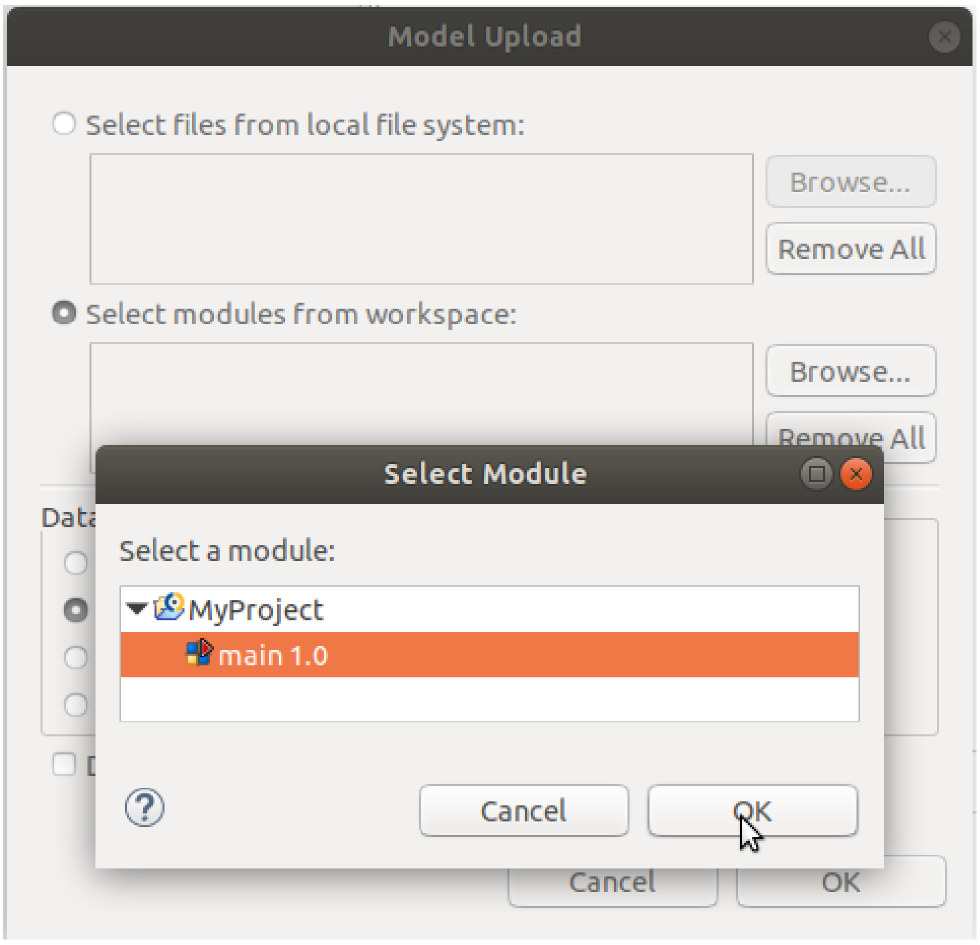
The Management perspective contains information about the resources on the LSPS server, to which your PDS is connected, in our case, the local PDS Embedded Server.

4. Upload *main* to the server:

   (a) In *Module Management*, click the **Upload** ✚ button

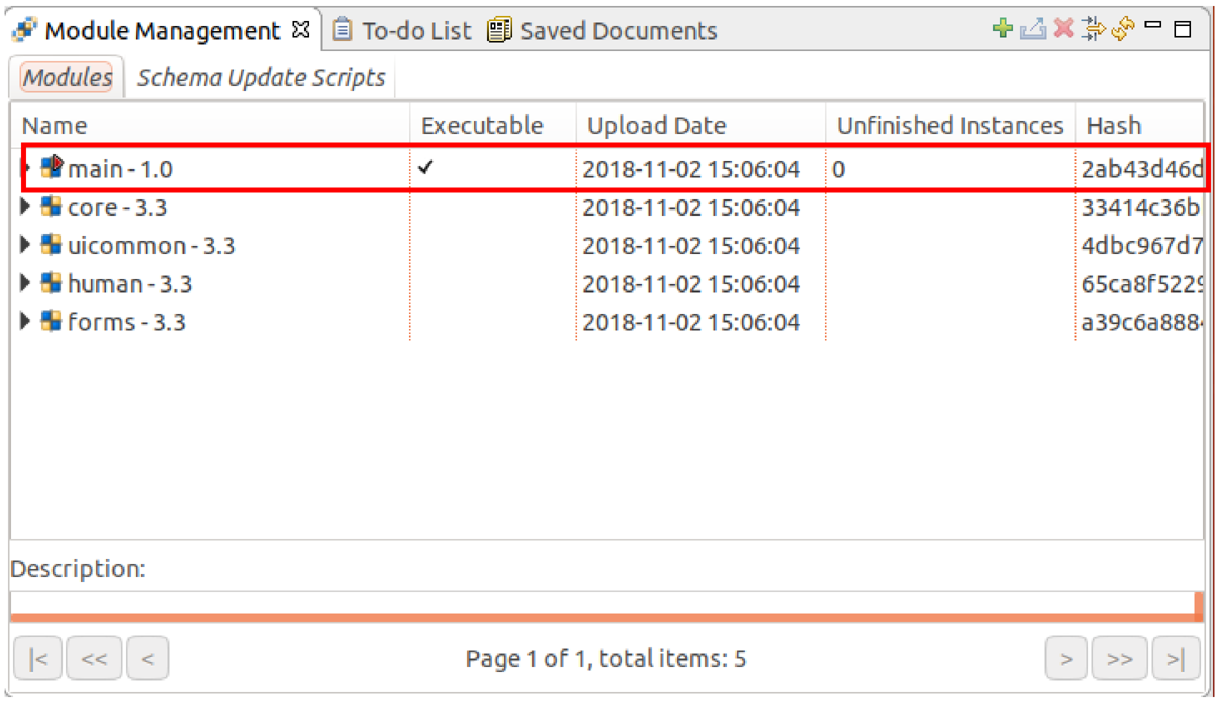


(b) In the *Module Upload* popup, select *Select modules from workspace* and select *mainModule*.



(c) Click **OK**.

(d) In the *Module Management* view, click the **Modules** tab.



Name	Executable	Upload Date	Unfinished Instances	Hash
main - 1.0	✓	2018-11-02 15:06:04	0	2ab43d46d
core - 3.3		2018-11-02 15:06:04		33414c36b
uicommon - 3.3		2018-11-02 15:06:04		4dbc967d7
human - 3.3		2018-11-02 15:06:04		65ca8f522
forms - 3.3		2018-11-02 15:06:04		a39c6a888

Page 1 of 1, total items: 5

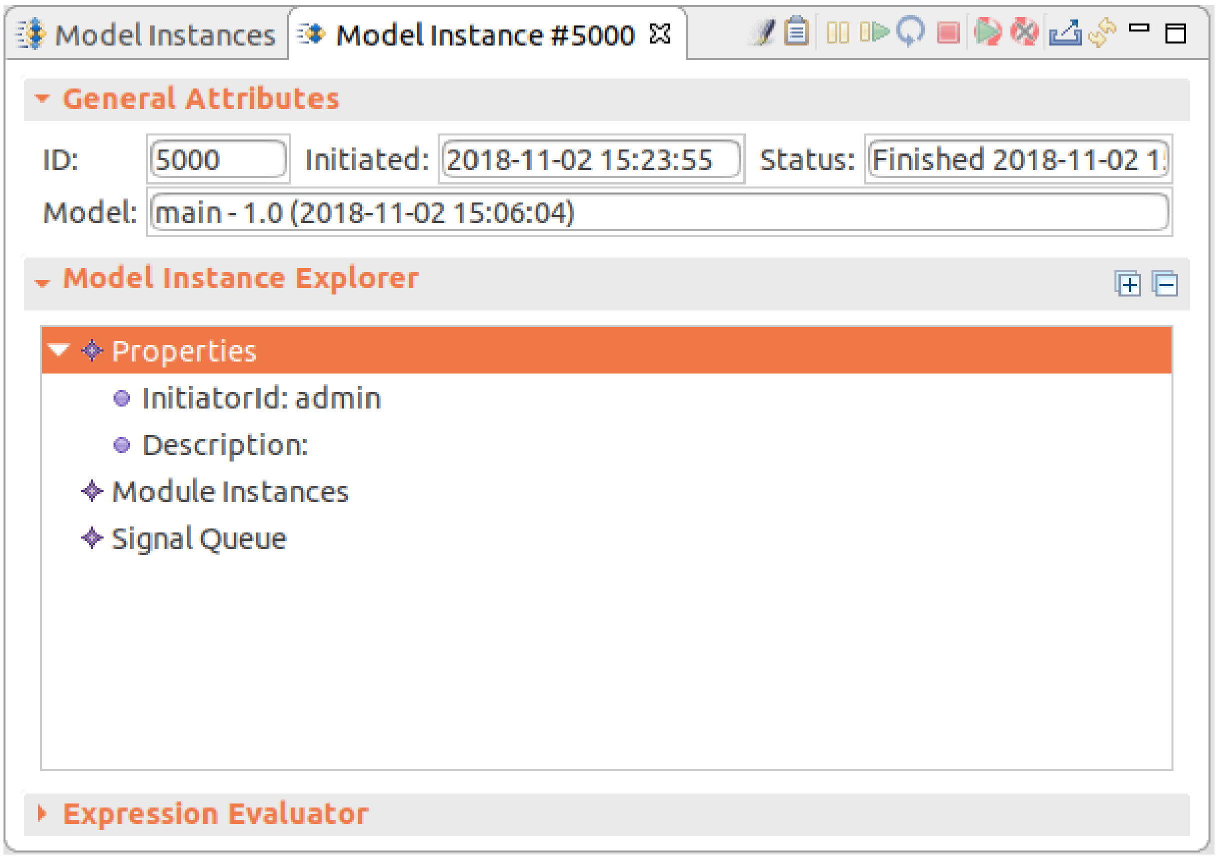
5. Create a model instance of *main*:

   (a) In the *Model Instances* view, click the **Create** ✚ button.

   (b) In the *Model Instance Creation* popup, select the module and click **OK**.

Note that the *Model Instances* view now contains an entry with the model instance we have just created: double-click it to display its details. There is not much to see, since the underlying module did not contain anything.

Though not much really happened, the server still created a model instance and *in* the model instance, it created an instance of the module. The underlying module is empty so there was nothing to do so the module instance finished and then the model instance finished.



General Attributes

ID: 5000 Initiated: 2018-11-02 15:23:55 Status: Finished 2018-11-02 1

Model: main - 1.0 (2018-11-02 15:06:04)

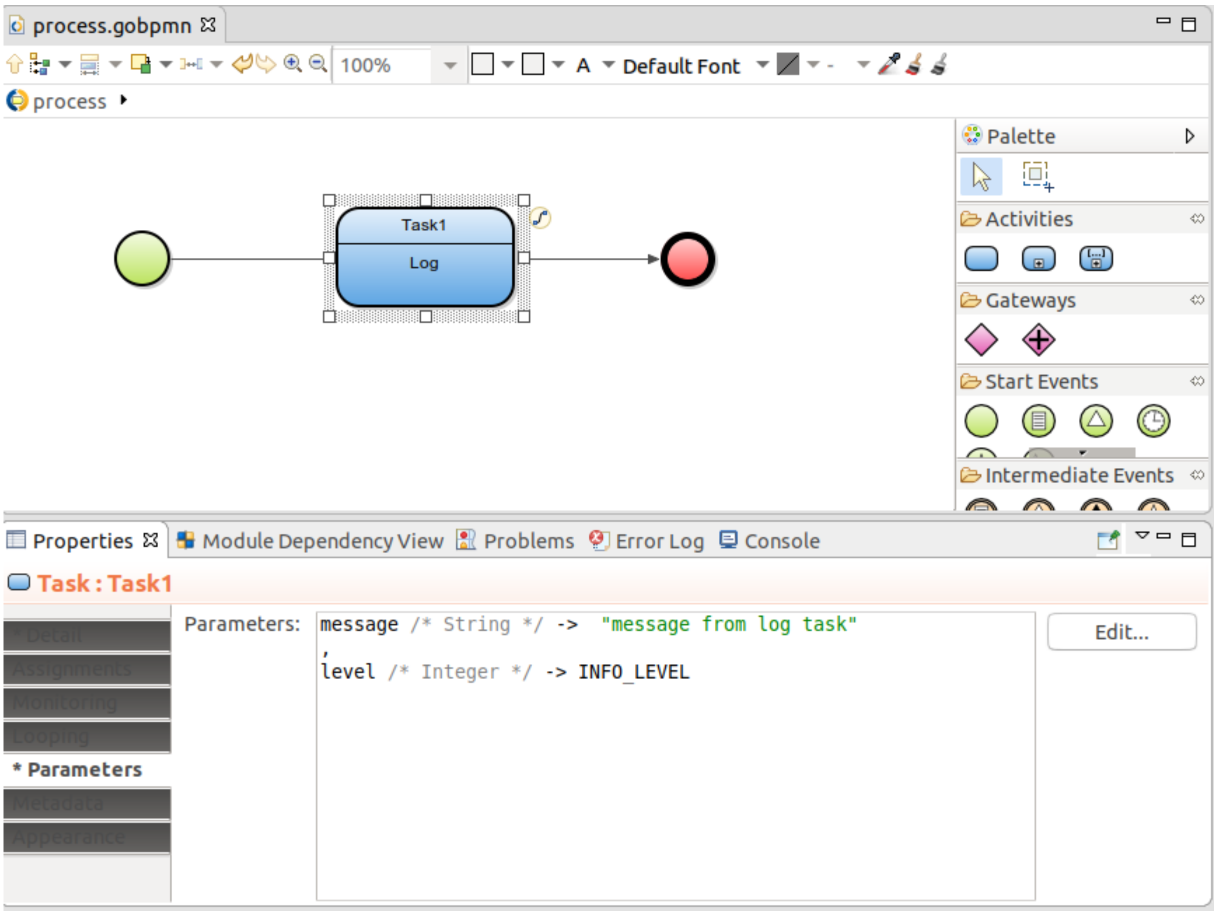
Model Instance Explorer

- Properties
  - InitiatorId: admin
  - Description:
- Module Instances
- Signal Queue

Expression Evaluator

Let's add some content to our module–a BPMN process–to execute some logic:

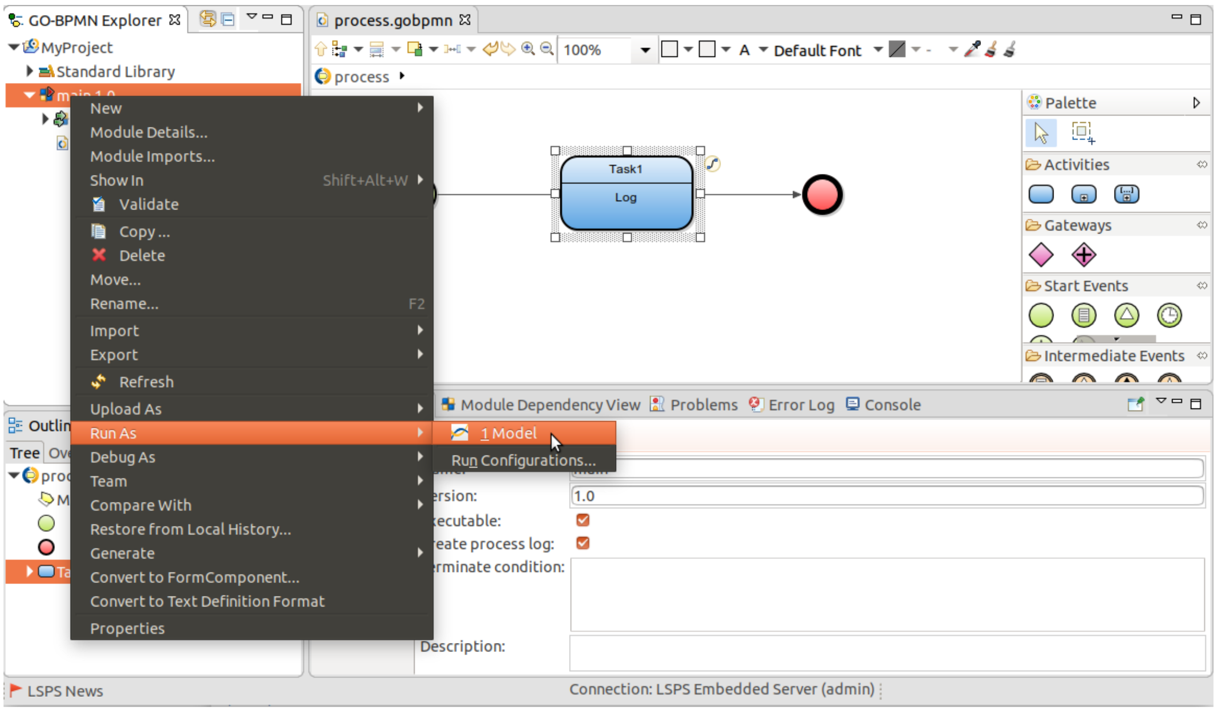
1. Open the *Modeling* perspective.
2. Right-click *main* and go to **New** > **Process Definition**
3. In the popup, enter the process name and select *BPMN-based process*.
4. Design a process with a None Start Event, a Log task, and a Simple End Event.
5. Set the Log task properties.



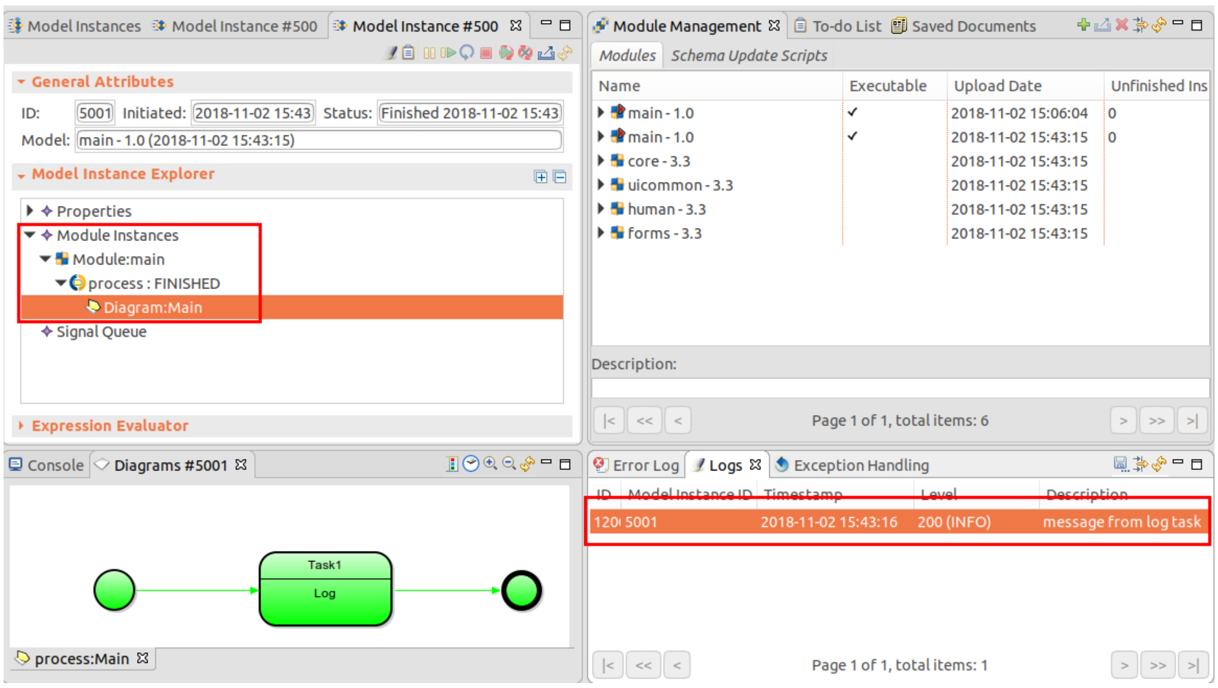
Parameters:

```
message /* String */ -> "message from log task"
,
level /* Integer */ -> INFO_LEVEL
```

6. Upload the main module and create its model instance: you can do so as we did before from the Management perspective or you can right-click the module and go to **Run As**> **Model**.



7. Switch to the Management perspective, refresh the *Model Instances* view (click the **Refresh** button in the view toolbar) and check the detail of the new model instance.



ID: 5001 Initiated: 2018-11-02 15:43 Status: Finished 2018-11-02 15:43
Model: main - 1.0 (2018-11-02 15:43:15)

- Properties
- Module Instances
  - Module:main
    - process : FINISHED
      - Diagram:Main
- Signal Queue

Name	Executable	Upload Date	Unfinished Ins
main - 1.0	✓	2018-11-02 15:06:04	0
main - 1.0	✓	2018-11-02 15:43:15	0
core - 3.3		2018-11-02 15:43:15	
uicommon - 3.3		2018-11-02 15:43:15	
human - 3.3		2018-11-02 15:43:15	
forms - 3.3		2018-11-02 15:43:15	

Description:

Page 1 of 1, total items: 6

ID	Model Instance ID	Timestamp	Level	Description
120	5001	2018-11-02 15:43:16	200 (INFO)	message from log task

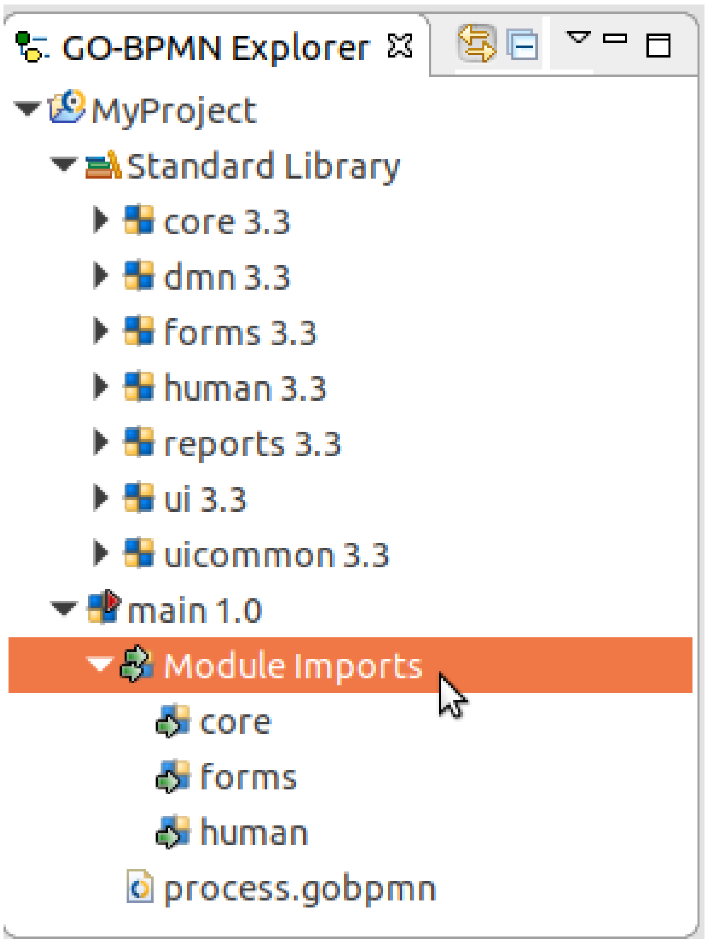
Page 1 of 1, total items: 1

Note the process instance in the details and the log entry in the *Logs* view.

3.2.1 Module Reuse

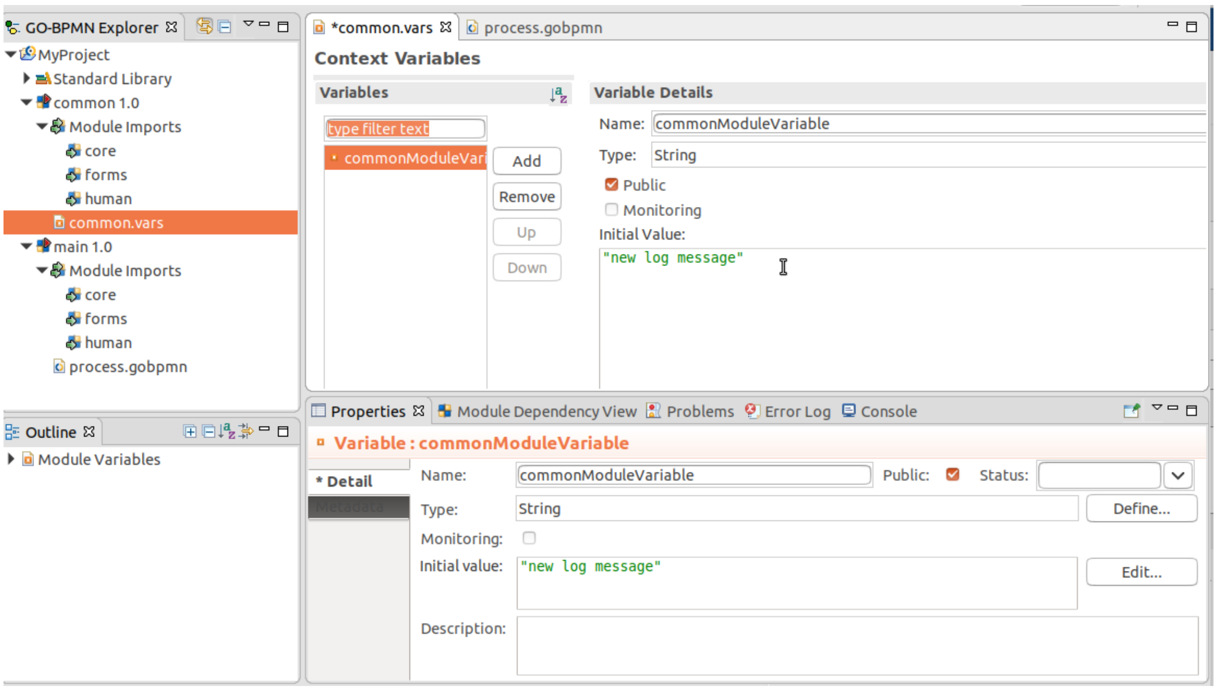
You have designed an executable module with a simple process and instantiated it on the PDS Embedded Server: the server created a model instance with the module instance and, in the module instance, the process instance, which got executed and logged a message.

When you uploaded the module for the first time, the modules of the Standard Library have been uploaded as well: these modules were imported into your module when you created it. Module imports appear as part of the module so you can use their content. This allows you to reuse what you or other users have already created, and potential reuse is what you should have in mind when structuring your model.



Let's create and import a module:

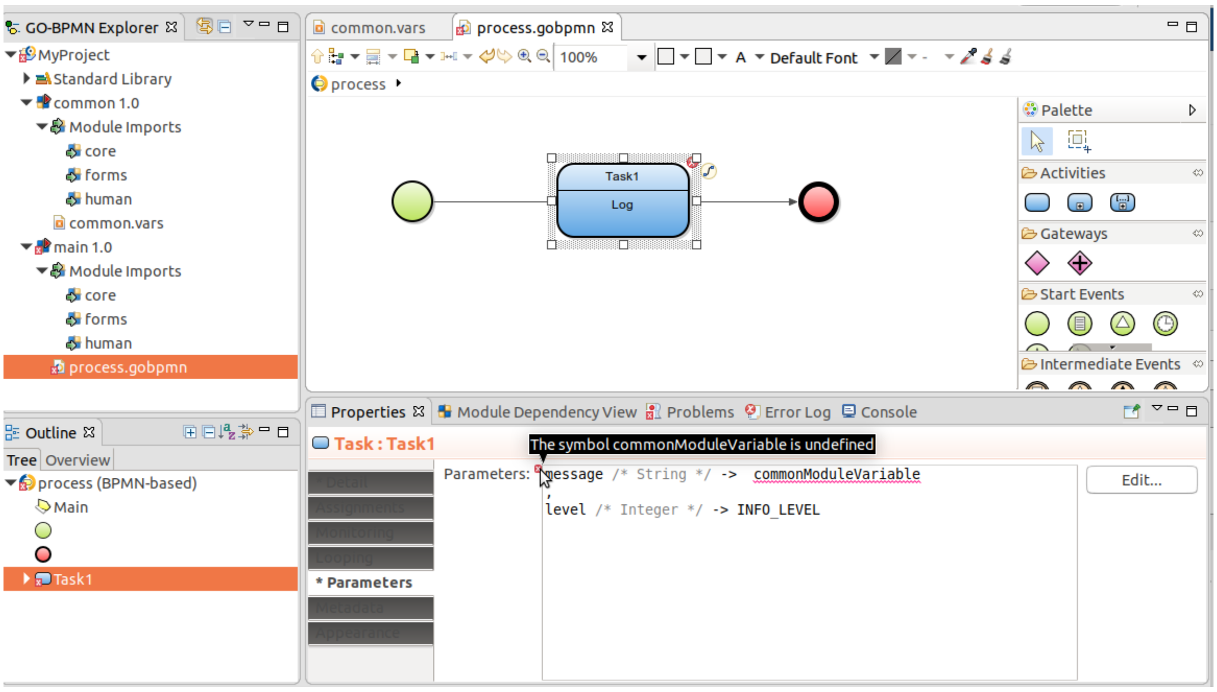
1. Create a *common* module in the project.
2. Create in a global variable in the module:
   (a) Right-click the module.
   (b) Go to **New** > **Variable Definition** and then click **OK**.
   (c) In the open editor, click **Add** and on the right, define a variable.



Name: commonModuleVariable
Type: String
☑ Public
☐ Monitoring
Initial Value:
```
"new log message"
```

Variable : commonModuleVariable
Name: commonModuleVariable Public: ☑ Status:
Type: String
Monitoring: ☐
Initial value: "new log message"
Description:

3. In the process in *main*, try to use the variable from *commonModule* (for example, in the message of the Log task): you will get a validation error saying that no such entity is available.

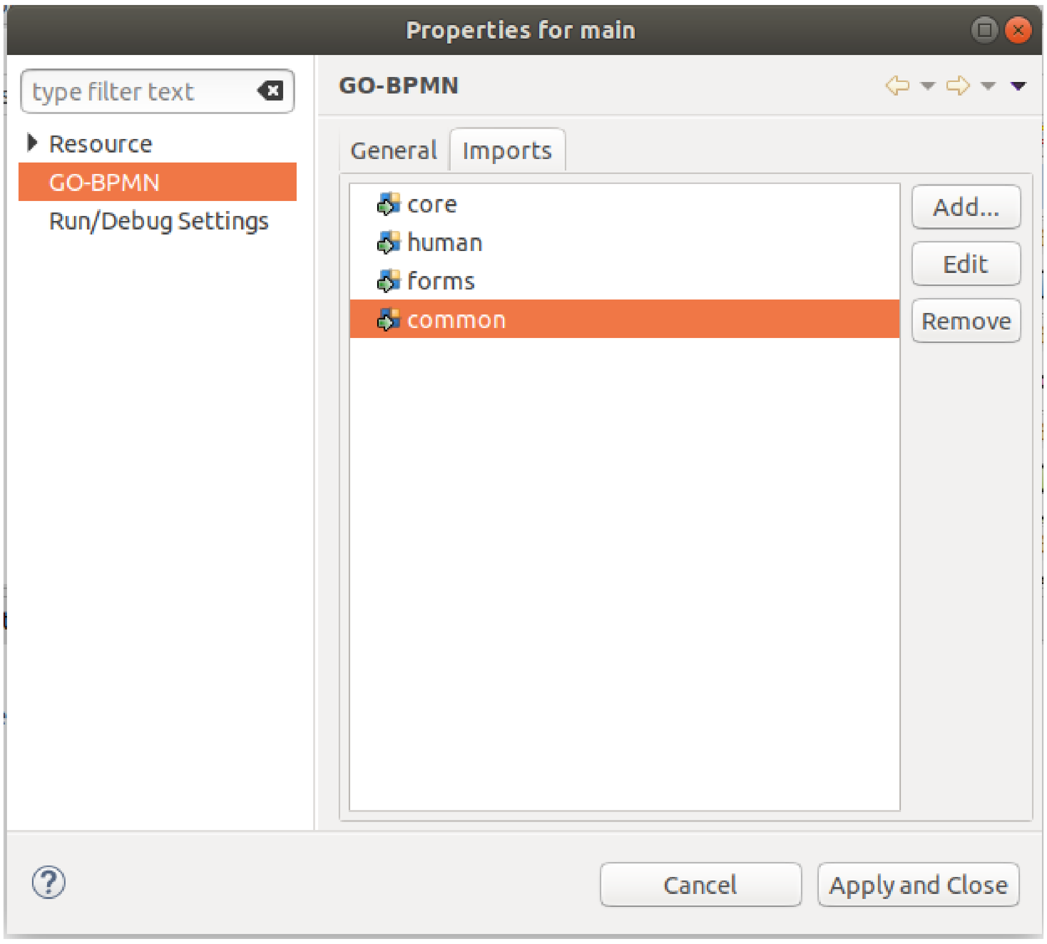


Task : Task1
The symbol commonModuleVariable is undefined
Parameters:
```
message /* String */ -> commonModuleVariable
,
level /* Integer */ -> INFO_LEVEL
```

4. Now import *common* into *main*:

   (a) In the GO-BPMN Explorer, double-click the Module Imports node in *main*.

   (b) In the dialog box, click Add and double-click *common*.

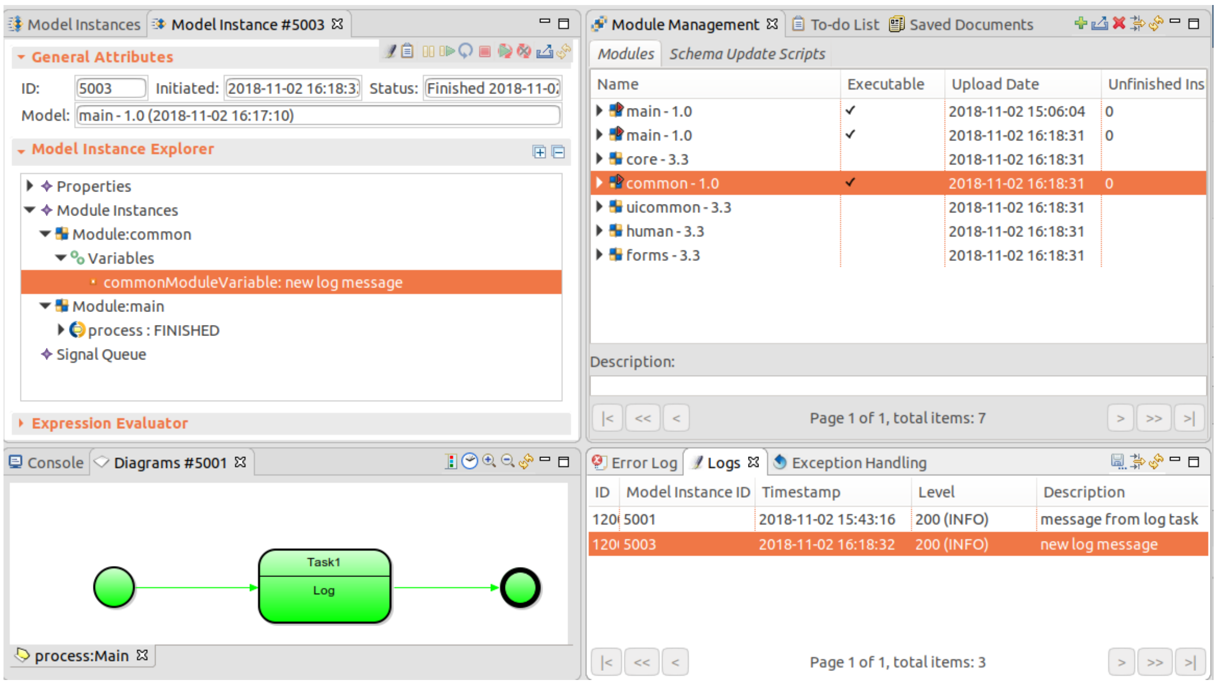


GO-BPMN

- core
- human
- forms
- common

(c) Watch your reference problem go away.

5. Switch to the *Management* Perspective.
6. Upload and run *main*
7. Refresh the views and check the details of the new model instance.



ID: 5003 Initiated: 2018-11-02 16:18:3 Status: Finished 2018-11-0
Model: main - 1.0 (2018-11-02 16:17:10)

- Properties
- Module Instances
  - Module:common
    - Variables
      - commonModuleVariable: new log message
  - Module:main
    - process : FINISHED
- Signal Queue

Name	Executable	Upload Date	Unfinished Ins
main - 1.0	✓	2018-11-02 15:06:04	0
main - 1.0	✓	2018-11-02 16:18:31	0
core - 3.3		2018-11-02 16:18:31	
common - 1.0	✓	2018-11-02 16:18:31	0
uicommon - 3.3		2018-11-02 16:18:31	
human - 3.3		2018-11-02 16:18:31	
forms - 3.3		2018-11-02 16:18:31	

Description:

Page 1 of 1, total items: 7

ID	Model Instance ID	Timestamp	Level	Description
120	5001	2018-11-02 15:43:16	200 (INFO)	message from log task
120	5003	2018-11-02 16:18:32	200 (INFO)	new log message

Page 1 of 1, total items: 3

Note the following:

- The module *common* has been uploaded as part of *main*.
- The model instance of *main* now contains two module instances (visible in the Model Instances Explorer on the right).
- The global variable value in the *common* module instance is `new log message` and so is the new log entry value in the Log view.

Note: Only if a module is marked as executable can the server create a model instance over the module: this is convenient if you are using a module as a resource for other modules and don't want to use it as a base for model instances by itself: Non-executable modules just sit in the module repository on the server and are used by model instances created based on executable modules. An executable module resembles to marking a method as `main`: It signalizes that this module is where your execution starts. Unlike with main, you can have multiple executable modules in one module. However, the other executable modules do not have any impact on the model instance.

Proceed to Business Data

3.3 Business Data

Every company works with data that have some structure. Let's take invoices for example: as a bare minimum they have an ID number and a date. The rest of the data is where it gets more complicated: the bill-to data and the list of purchased items consitute further data structures, which engage in mutual relationships. Such data structures are reflected as custom complex data types, called *Records*.

This chapter provides only a brief introduction into data types and data type modeling. Note that to create both sophisticated and efficient data models, you will need more than being able to work with LSPS: make sure to analyze your data model properly before you design it to prevent performance and maintenance issues in the future. More detailed information on data types is available in `the Data Type Model section of the GO-BPMN Modeling Guide`.

3.3.1 Creating a Record

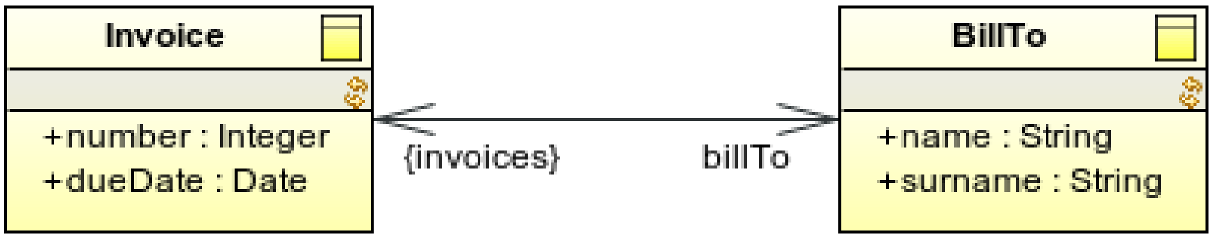
You will create a very simple data type model for invoicing:

1. First you need to create a *data type definition*. This is a file that will hold the data type model. Since the data type model will be used by the entire application, we will create it in the `common` module:

   Right-click the `common` nodule and go to **New** > **Data Type Definition**. Click **Finish** in the popup.

   Note that you can have your data model in multiple *data type definitions* and they will be still considered one data model. Therefore, your Records have to have different names even if they are in different files: What "separates" your data models is a module (the namespace is defined by the name of the module), not files.

   You can see the definition file in the GO-BPMN Explorer. It is opened in the data-model editor: the white empty space in the editor area is the file's diagram, which is a visualization unit: it displays the content or part of the content of the file. The complete content is in the Outline view. You will get back to diagrams later.
2. Create the Record for the invoice with the number and dueDate Fields.
3. Create the related BillTo record with fields name and surname.
4. Create a relationship between the Records and name the relationship ends.

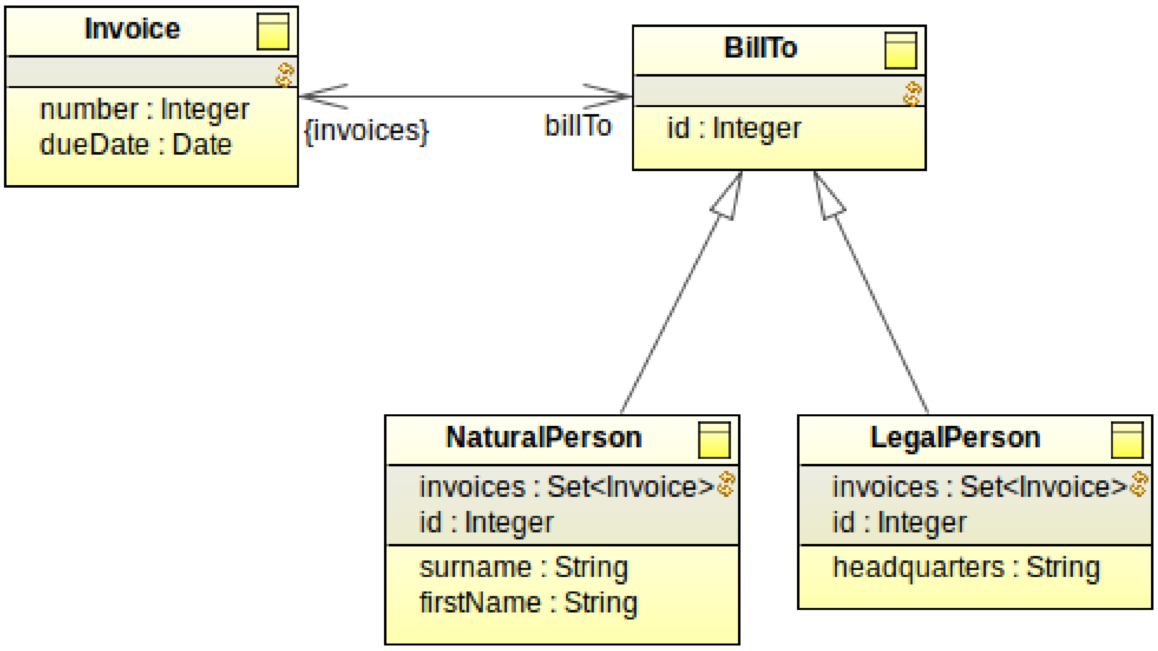


3.3.2 Using Inheritance

Let's rethink our BillTo Record: you could charge a company or a person; a company will not need a surname and a person will not need headquarters data. However, they are still considered the charged parties: to retain this structure, let's create child Records that will allow you to distinguish between them.

Add the id field to the BillTo record and create the child Records *NaturalPerson* and *LegalPerson* of the BillTo Record.

With a bit of luck you have now a model similar to this one:



The BillTo Record is the supertype of the other Records: this means that the child Records have all the fields and methods of the parent. You will get to methods later.

And there we have another problem: we do not want to create a BillTo record ever again: we want to charge always a natural or legal person. Hence, BillTo must be abstract: in the Properties of the Record select the **Abstract** flag.

Now, the assignment of an invoice to John is no longer valid: John must become a NaturalPerson.

```
john := new NaturalPerson (
  name > "John",
  surname > "Doe",
  invoices > {currentInvoice_1, currentInvoice_2}
)
```

For further details on inheritance, refer to the official documentation

3.3.3 Defining Methods

As we have mentioned above, Records are similar to classes in object-oriented programming. As such, they can define methods.

Let's define methods for NaturalPerson:

The assignment expression will now return john's name.

Now define the following methods for the abstract BillTo record:

```
BillTo {
   public Integer getId(){
     this.id
   }
   private void setId(Integer id){
     this.id := id;
   }
}
```

You can now perform the call `john.getId()` which will use the `getId()` method defined on the BillTo record. However, if you try to call the `setId()`, which is a private method on the abstract BillTo record, you will get a validation error.

3.3.4 Persisting Data

Right now all the data on invoicing gets trashed right after the model instance finishes. To persists the data so they remain available after the instance finishes, change the records to shared records: an instance of a shared record is stored immediately in the database. Any operation on the data of the record instance is performed on the persisted data. The same is true for record relationships.

Mark all Record as Shared in their properties:

You got a bunch of errors saying `Shared record <NAME> must specify at least one primary key`: That is because each shared record represents a database table and each record field represents a column in that table, for example, record Car with fields *make* and *license plate* is stored in table Car with columns *make* and *license plate*. Whenever you create an instance of a shared record, a row with the data is added to the table. To be able to find a particular record instance (row), you need to specify something that identifies it; in the example it is the license plate of the car: This unique property is the *primary key*.

The records already contain such properties, the *number* and *id* properties. All you need to do is mark them as primary keys. Also so you don't have to enter the primary-key manually everytime you are creating a record instance, set it to generated the primary keys automatically:

Note that the *BillTo* Record has an additional database property *O-R inheritance mapping*. Its default setting is *Each record to own table*. Let's take a look at what that means:

1. Start PDS Embedded Server and upload the module.
2. Open a database client, such as SQuirreL or DBeaver, and check the schema of the Record tables.

   For PDS Embedded Server, connect to `//localhost/./h2/h2;MVCC=TRUE;LOCK_TIMEO↩ UT=60000`; for jdbc the URL will be `jdbc:h2:tcp://localhost/./h2/h2;MVCC=TRUE;LOC↩ K_TIMEOUT=60000`. Both the name and password are `lsps`.

The tables for individual Records reflect the model: the BillTo table contains the id column only and the NaturalPerson table contains the id identical to the BillTo.id, surname, firstname, etc.

Let's test the other inheritance mapping:

1. Stop the server.
2. Reset the database or change the schema update strategy to *Drop and create modeled DB tabled*.
3. Set the Single table per hierarchy setting on the BillTo and Invoice records.
4. Start the server, upload the Module, and check the schema:

The **BILL_TO** and INVOICE table now contain the entire hierarchy along with the TYPE_ID column which holds the record type of the entity, such as invoice-types' NaturalPerson: there are no tables for the *NaturalPerson* or *LegalPerson* records.

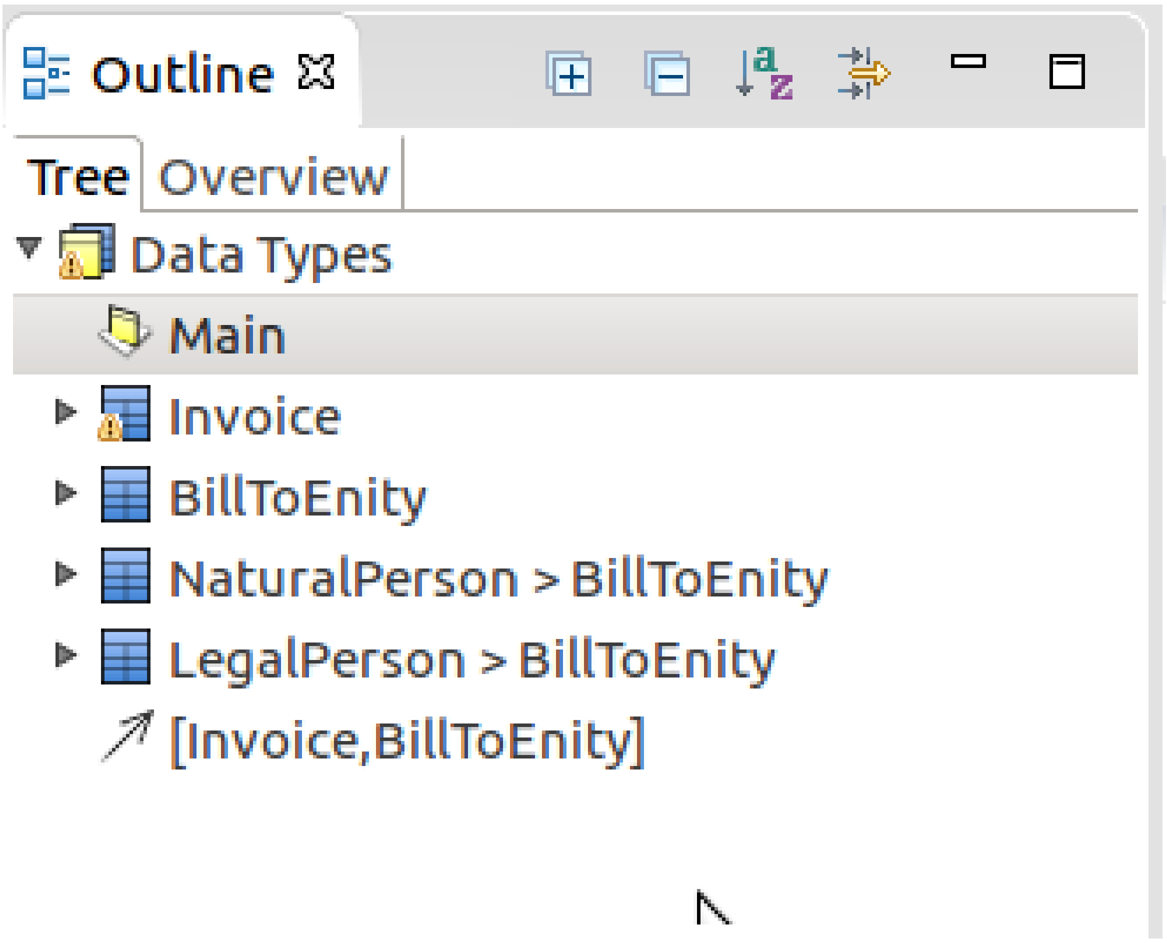
3.3.5 Changing a Data Model with Shared Records

At some point, you might need to change the data model for your business data. Let's say you need to separate the details of the invoice from the rest of the Invoice record. You remove the dueDate field, create a new *BasicData* record with the dueDate field and create a relationship between the two. For the underlying database this means a change of the Invoice table. If you can delete the invoicing database, drop the database and upload the model so the tables are created anew. However, if you need to preserve the existing data, you need to migrate the existing data so they fit the new database schema. This is an advanced topic and is covered in the `developer documentation` and `deployment instructions`.

To allow an easy migration of database, consider using a tool for tracking of database schema changes, such as, Flyway or Liquibase.

3.3.6 Using Diagrams to Organize Records

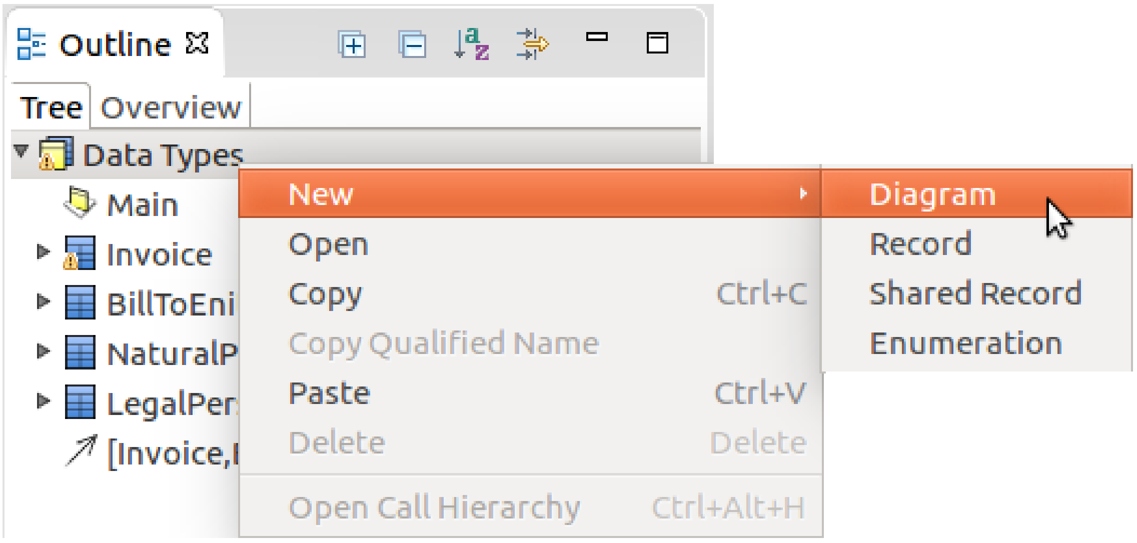
At this point, all elements of the data-type model are displayed in the default **Main** Diagram: When you insert a new element onto the canvas, two things happen: the element is created in the definition file and it is displayed in the *Main* diagram. One definition file can contain multiple diagrams and one element, such as a Record, can be displayed in the diagrams arbitrarily. It might not be displayed at all. To check the content of the definition file, look at the *Outline* view.



Note that diagrams do not represent a namespace. They are merely a presentation tool: If you insert two elements with the same name in two diagrams of the same definition file, you will get a error due to the name clash.

Create another diagram in the data-type definition file and display an existing Record in it:

1. In the *Outline* view, right-click the root *Data Types* node and then **New** > **Diagram**



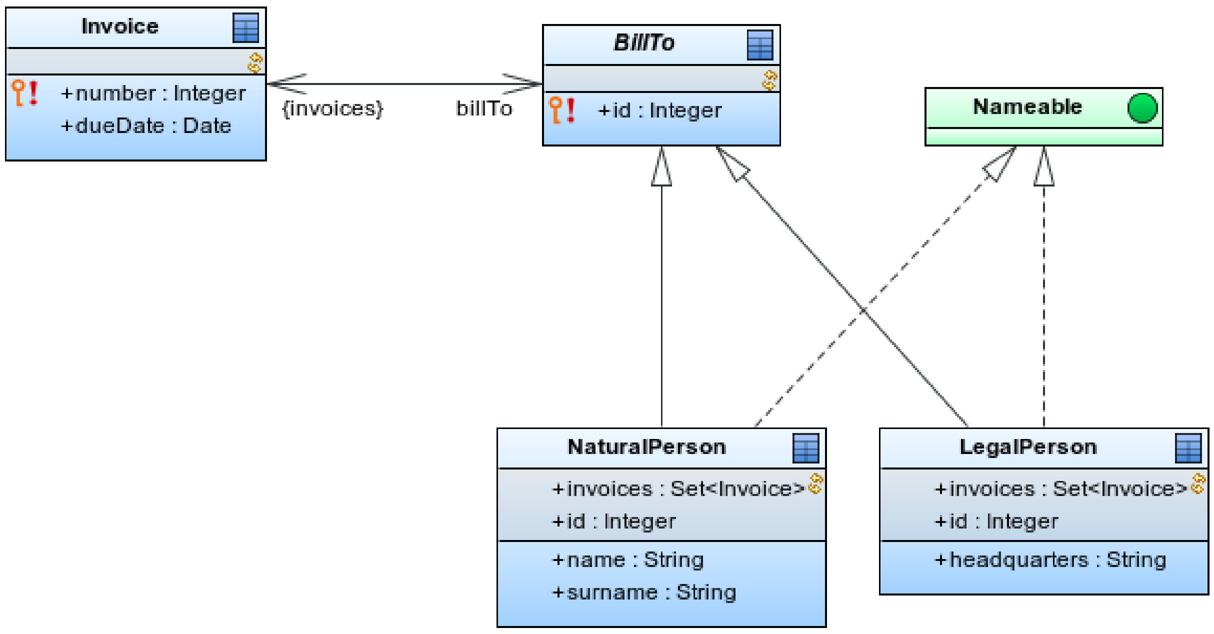
2. In the Properties view, enter the name of the diagram, for example Inventory. Rename the Main diagram to Invoicing.

3. To display elements in a diagrams, drag-and-drop them from the Outline view onto the canvas of the diagram.

For more information, refer to `the Diagram chapter of the GO-BPMN Modeling Guide`.

3.3.7 Using Interfaces

Create the `interface` *Nameable* with the `getName()` method; make the *NaturalPerson* and *LegalPerson* realize the interface.



Proceed to To-Dos and Documents

3.4 To-Dos and Documents

While the main ambition of BPMN and GO-BPMN processes is to automate your processes as much as possible, there is a good chance that your processes will now and then require input from a human user.

You can get such user input at a certain point of a process execution or at any point in time. Depending on this requirement, you will use either a to-do or a document:

- A `to-do` serves to get input at a particular moment during a process execution.
- A `document` serves to get input at any point in time; no process execution is required;

3.4.1 Getting Input for a Process

To get input at a particular moment during a process execution, we need to add a *user task* to the process flow: when the execution reaches the user task, it generates a so-called to-do, a work item with a form. A set of users, performers, can see the to-do from the *Application User Interface*; we say that the to-do has been assigned to the initial assignees. When one of the assignees opens the to-do, the to-do disappears from the to-do lists of other assignees: it is locked by the user. This user performs the actions and provides the information required by the to-do and once happy with the result they submit the form. On submit, the execution of the user task finishes and the execution of the process continues.

Hence, a user task must define the data for the to-do:

- the set of its *performers*,
- the content that is displayed when one of the perspective opens the to-do.

Note: In your custom Application User Interface, you will most likely substitute the default **To-Do List** with your custom implementation to display additional information about the to-dos. Instructions on how to do so are available in a `dedicated tutorial`.

Now, we will create a process with a user task, define a simple content for the to-do and demonstrate, how the end-user submits the to-do.

3.4.1.1 Creating a Process with a User Task

To create a process with a user task, do the following:

1. Create a process definition file.
2. Insert a user task.

You have now a process with a user task: there are multiple errors on the task since we have not defined its required parameters, yet; we will do so shortly.

Note that you do not need to insert a None Start Event or Simple End Event into the process; they are implied.

3.4.1.2 Defining To-Do Properties

When the *user task* becomes *alive*, the system generates a to-do with some content for the initial performers or assignees. Hence we need to define the content and the performers:

1. Let's first define the content:
   (a) Create a form definition file.

   Important: When creating the form definition, make sure the flag **Use FormComponent-based UI** *IS SELECTED*: PDS comes with two different implementations of forms; the flag defines which of the two is used in the new form. You are using the `newer implementation`.

   (b) In the form, insert a button and define its Click Listener expression so it submits the form (the expression is executed whenever the button is clicked).

2. Now, let's define the properties of the user task:
    (a) Select the user task on the canvas or in the Outline view.
    (b) Focus the Properties view.
    (c) In the Properties view, open the *Parameters* tab and define the following parameters:
        - Define the title of the todo, for example, `title -> "Submit Me"`
        - Define the admin user as the sole performer in the task properties: `performers -> {get↩ Person("admin")}`
          The `getPerson()` function call returns the **admin** user: no other user will see the to-do in their to-do list..
        - Define the form you created as the to-do content: this is done in the `uiDefinition` parameter of the user task, for example, `uiDefinition -> new SubmitForm().`
3. Save changes.

After you save changes, your project should be error-free.

3.4.1.3 Running the Model

Let us run the model: start PDS Embedded Server.

The server did the following:

1. It created a model instance.
2. It created an instance of the process in this model instance.
3. It triggered the execution of the process instance: this created an instance of the user task, which is now **alive** and generated a to-do.

The execution is now stuck on the User task waiting for the to-do to be submitted: open the Application User Interface as the admin user and submit the to-do.

When you clicked the Submit button in the to-do, the execution flow continued:

- in our case the to-do ceased to exist,
- the user task became finished; because there was no execution going on in the process instance,
- the process instance finished and subsequently also the model instance.

You can check the status in the Management perspective.

3.4.2 Getting Input at Any Point in Time

If you want to create a page that will take input from a user at any point in time, create a document: similarly to a to-do, its content is a form. Unlike a to-do, the document is accessible always as long as the module with the document is in the Module Repository.

> **Note:** Under the hood, whenever a user opens a new document, the system creates a new model instance. The instance contains only the instance of the document and finishes when the user submits the document.

3.4.2.1 Creating a Document Definition

You will now create a document with the same content as the to-do (we will use the same form).

To create a document, do the following:

1. Create a document definition file.
2. In the file, create a new document definition with the following properties:
   - UI Definition: the form with the submit button, for example, `new SubmitForm()`
   - Access right: access rights of the user to the document; set it so that any user can access the document; that means it must always evaluate to **true**
   - Navigation: page to go to after the document is submitted

     By default, the user is redirected to home page. This property overrides the home redirect. You could set it to the Documents page:

     ```
     { s:Set<Todo> -> new AppNavigation(code -> "documents")}
     ```

3.4.2.2 Uploading and Accessing a Document

Run or connect to your server and upload the module with the document.

Now you can open the document as any user.

Proceed to Goal Hierarchies

3.5 Goal Hierarchies

In LSPS, you can create BPMN processes just the way you know them; however, LSPS provides support for GO-↩ BPMN processes, processes with a conservative goal-based extension of BPMN.

Such a process has a goal-based unstructured layer, which allows you to separate what the process should achieve from how it should achieve it:

- what to achieve is defined in goal hierarchies
- how to achieve the required state is defined in plans in BPMN

Let you just dive in and demonstrate the goal-oriented approach on an example.

3.5.1 Defining Your Goals

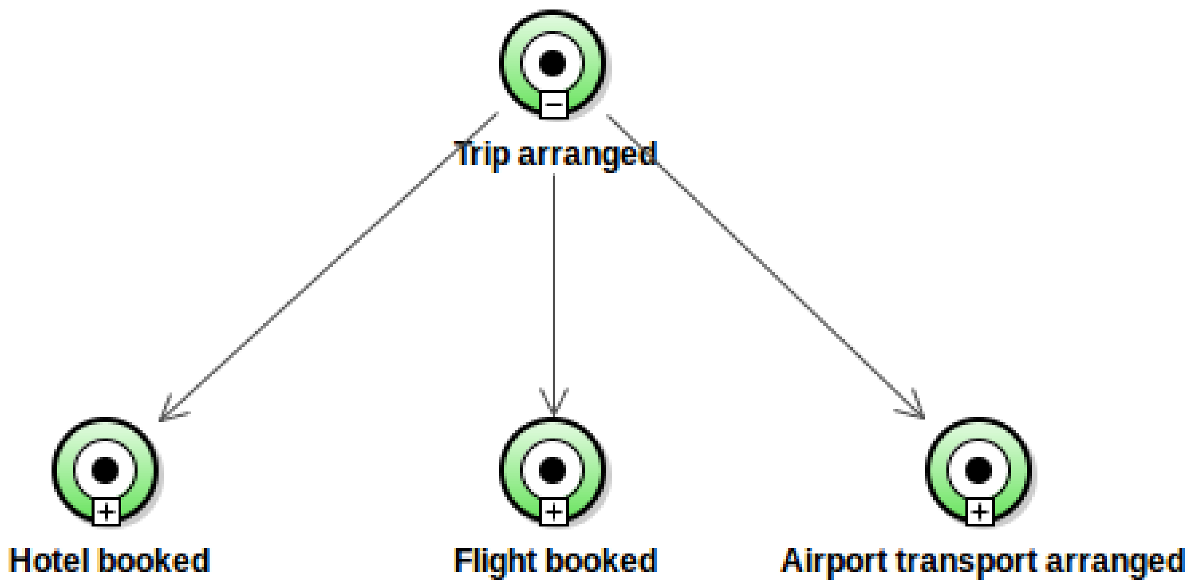
Let you consider a process for arranging a business trip:

- The main goal or purpose is to have the "Trip arranged".
- The trip is considered arranged after the accommodation, flights, and transport to and from the airport are arranged.

Once the trip is arranged, the purpose of the process has been met: this is the main goal: we identified the main goal, to arrange the trip, and the 3 sub-goals it comprises. In GO-BPMN, the goals are represented explicitly by Achieve Goals elements arranged in the required hierarchy.

In this case, the main goal will be the top Achieve Goal, that is, an Achieve Goal with no incoming "arrows" or decompositions: such goals are started, or triggered, by the process: when the process starts, the top goals start. The represent the ultimate required result.

As already defined above, to achieve the goal we must achieve multiple minor things subordinate to the main goal. The sub-goals are represented again by Achieve Goals. However, these are connected to their main goal and will be triggered all at once by the main goal: The sub-goals are executed in parallel manner.



Let's create the hierarchy.

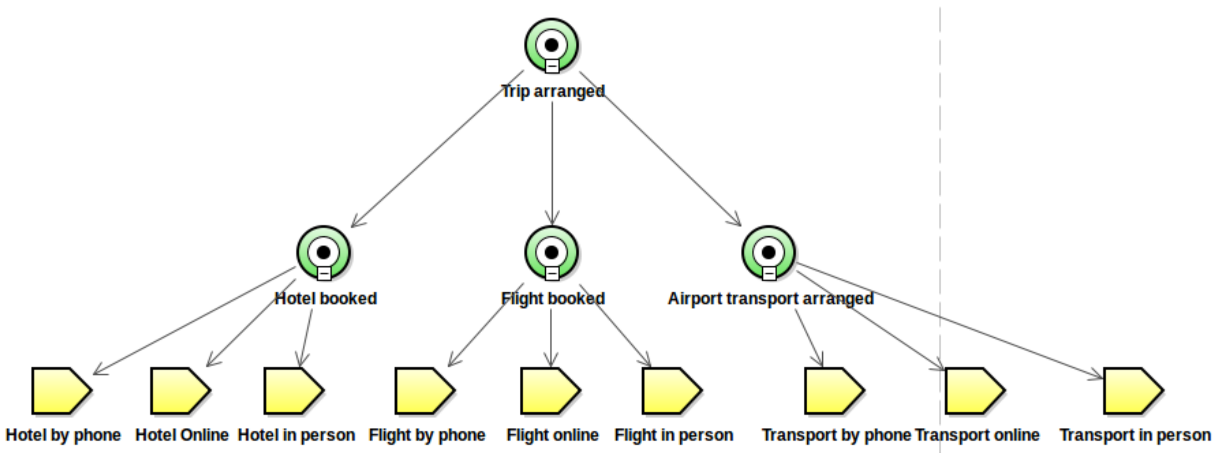
3.5.2 Defining How to Achieve Your Goals

Since we have defined what we want to achieve, let's get to the how-to-achieve part. This is defined by Plans.

Each goal can be achieved in one or multiple ways: You can book a hotel, tickets, and the airport transportation in different ways, for example, by phone or online.

Generally, it does not matter how we achieve it: Therefore, a goal can have multiple Plans. The inside of a Plan defines how to achieve the task ahead using the standard BPMN flows that perform the action.

Now, decompose the sub-goals into plans. You should end up with a hierarchy similar to the one depicted below.



How to achieve a Plan or a BPMN process is defined by a standard BPMN flow in the Plans or BPMN Processes.

Let's start by defining a dummy flow in one of the Plans.

This flow simply starts the plan execution and immediately finishes: the None Start Event produces a token which passes through the Flow to the Simple End Event, which consumes the token. In your processes, you will want to execute Tasks, split your flow with Gateways, etc.

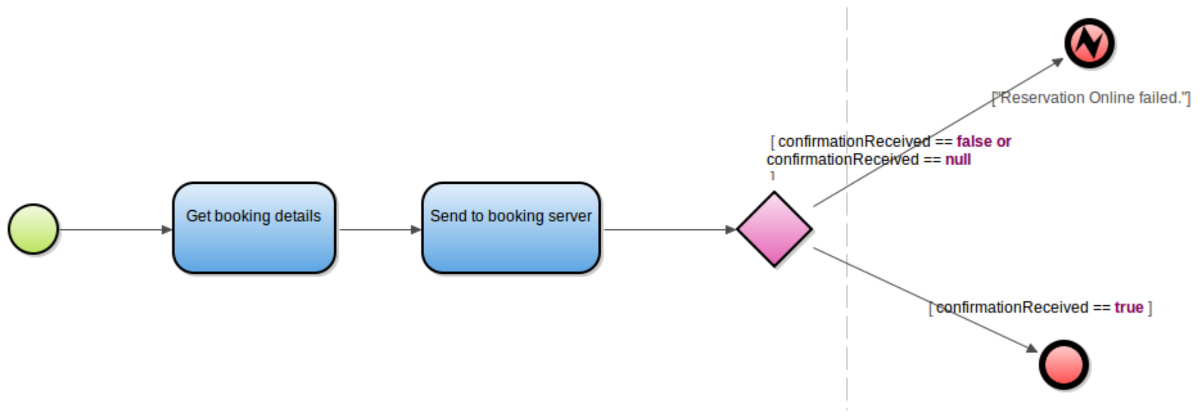

Get booking details
Send to booking server
[confirmationReceived == false or confirmationReceived == null]
["Reservation Online failed."]
[confirmationReceived == true]

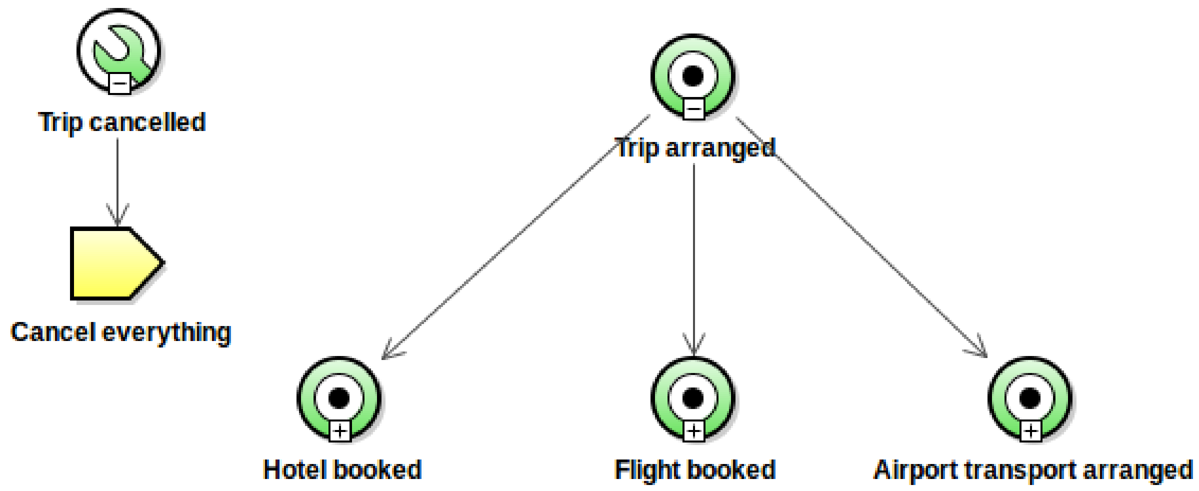
Figure 3.1 Example of a Plan body

3.5.3 Making Sure Things are on Track

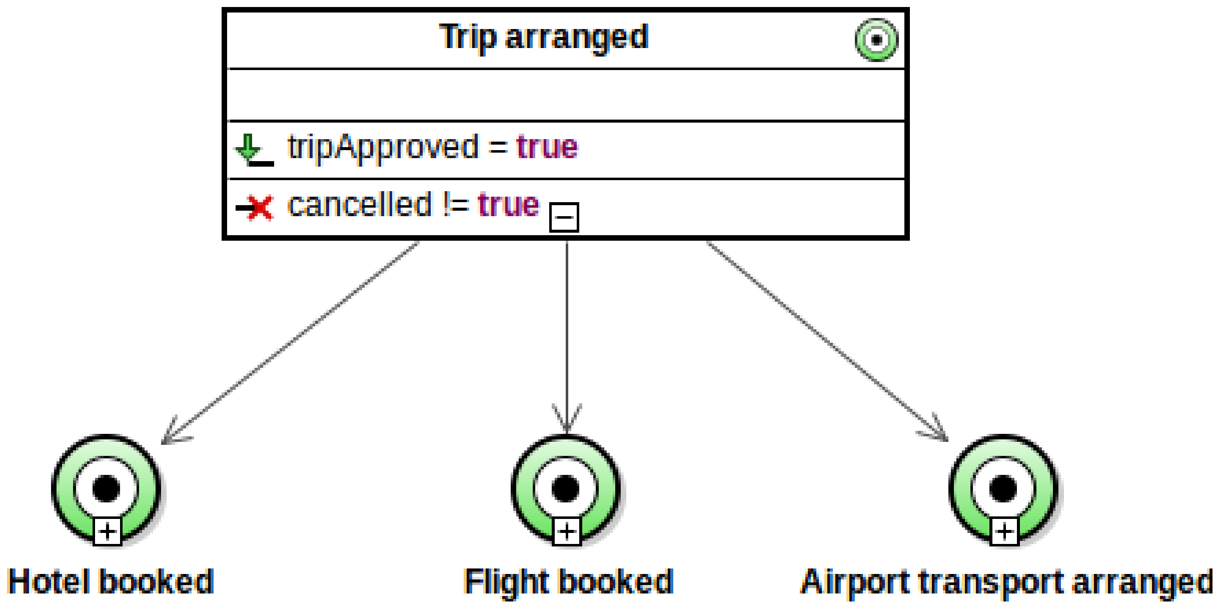
Some Goals or the entire Process might need to run only under certain conditions, for example, if the business trip gets cancelled, it does not make any sense to continue the execution. You might even need to rollback what was already arranged.

You can deal with conditions in goal hierarchies either using a Maintain Goal or with preconditions and deactivate conditions of Achieve Goals:

- If you want to rollback or compensate some action, use Maintain Goals: these goals define a condition: the moment the condition becomes false, the Maintain Goal triggers one of its Plans.



- If you need to check the condition before something happens, use pre-conditions of Achieve Goals.
- If you need to interrupt an Achieve Goal when something happens, define its deactivate condition.



3.5.4 Execution

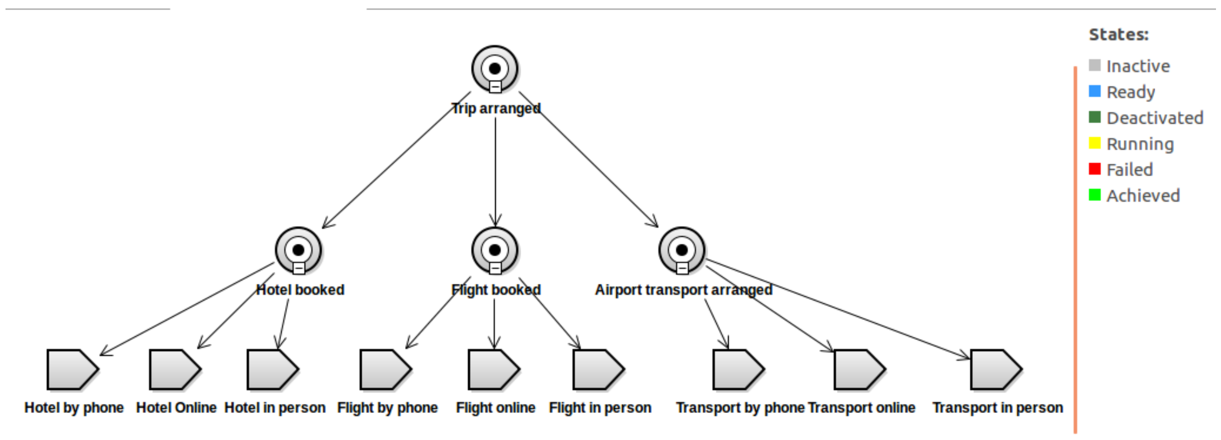
Let you take a look at the execution of a goal hierarchy and its process: note that the goal hierarchy is part of the process.

When you instantiate the model, the following happens:

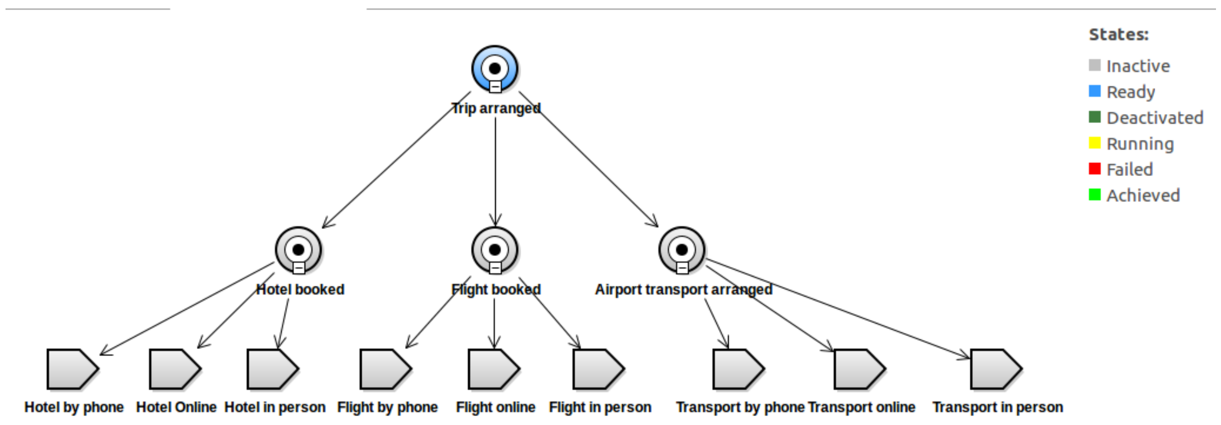
1. One process instance is created for each process: the context data is created (variables are created and have their initial values assigned).

   Note that only processes that have the flags Executable and Instantiate Automatically are instantiated immediately.

2. All Goals become inactive.



3. Top Goals become *Ready*.



4. Top Achieve Goals with their precondition evaluated to *true* become **Running**. If there is at least one such Goal, the process instance becomes *Running*.

   - If the running Goal is decomposed to sub-Goals, the sub-Goals become *Ready*. These then continue their life cycle (have their precondition checked, etc.). The goal is achieved, when all its sub-Goals become *achieved*.

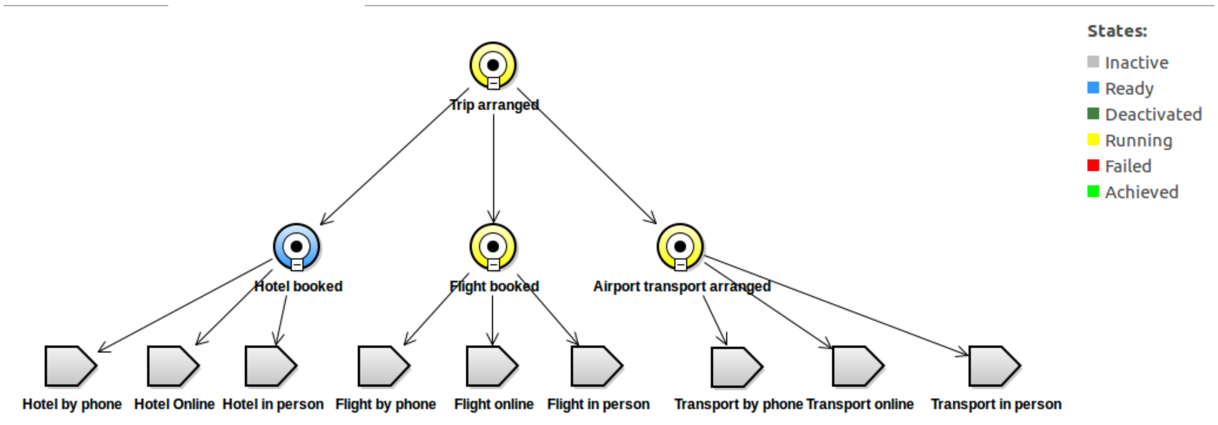


Figure 3.2 Triggered Subgoals

- If the running Goal is decomposed to Plans, the Plans have their conditions checked and **one Plan** with the condition *true* becomes *Running*: the None Start Event of its body is triggered.
  - If the body **ends with an Error End Event**, or it receives an error event, the Plan finishes as **Failed** (note the Plan must define the code of the error in its property Failure error codes) and the Goal triggers another Plan that was not executed yet.

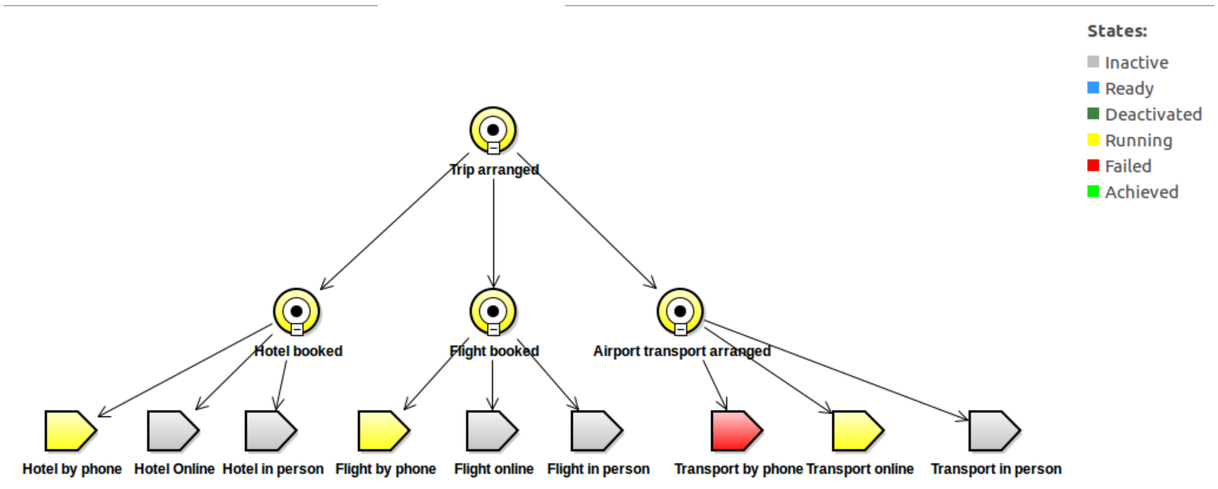


Figure 3.3 An alternative Plan running after a Plan failure

  - When the body ends with **another End Event**, the Plan finishes as **Achieved**.
  - When a Plan finishes as **Achieved**, its Goal becomes *Achieved*.

5. Once all Achieve Goals are *Failed* or *Achieved*, the process instance becomes *Finished*.

3.5.5 Summary

Goal-oriented approach to processes makes only sense if there are multiple parallel actions required to meet the goal.

If you want to use the goal-based approach, make sure the business process analysis is performed accordingly↩ : the "extraction" of goals requires a change in the mind set. However, this additional effort can result in more autonomoyou business processes.

When designing such a process, you should proceed as follows:

- Extract the purpose of your process into Goals.
- Identify the possible ways of how to achieve the leaf Goals of your hierarchy.
- Decompose the leaf Goals into Plans and design the Plan bodies.

3.5.5.1 Pros

- Dynamic flow execution: Some goals might never be triggered or triggered only after an event.
- Process instance restart based on conditions if Plans are well-structured, that is, they are minimalistic.
- Performing multiple process parts synchronously.

If your model does not require any of the above, consider using plain BPMN.

3.5.5.2 Cons

- Business Process Analysis of goal hierarchies might involve a steeper learning curve than that of BPMN processes.
- Only one Goal hierarchy is instantiated per process instance (no multi-instance Goals).
- Goal hierarchies do not support processing of BPMN events since there are other mechanisms that substitute these.
- Goal hierarchies are not legible if they contain conditions: the semantics are not obvioyou from the diagrams.
- If hierarchies have many Goals with conditions that are checked continuoyouly and contain execution, for example, assignment expressions, they can result in performance issues.

3.5.6 Documentation

- `Achieve Goals`
- `Maintain Goals`
- `Plans`

Proceed to Simple Model.

3.6 Simple Model

You are going to create an example model that will register clients: A client will log in to the application, enter their registration data, and submit the data. An agent will accept or reject the registration data.

You have to consider the following:

- *Data structure*: The specific business data involved is the registration data of the user. The data must remain available once provided: you will it as a shared Record.
- *People involved*: Two types of users, the client and the agent, are involved in the process. You will need two roles that will represent these user types.
- *Processes*:
  - The registration must be available to clients at all times and does not involve any decision-making logic or work distribution; no BPMN process is required. You will implement the registration as a Document available to all client users.
  - The approval request requires distribution of work to a group of users: the request should be available to agents after a client submits their data at first. You will implement the approval as a process triggered when a client submits the registration.

3.6.1 Prerequisites

First, create a GO-BPMN project with a module:

1. Make sure you are in the Modeling perspective: GO-BPMN Explorer should be displayed on the left.

   If this is not the case, go to **Window** > **Perspective** > **Open Perspective** > **Other** and select the *Modeling* perspective in the dialog.
2. Create a GO-BPMN project:
   (a) Go to **File** > **New** > **GO-BPMN Project**.
   (b) In the **New GO-BPMN Project** dialog, enter **ClientRegistration** as the project name and click **Finish**.
3. Create a GO-BPMN module in the project:
   (a) Right-click the *ClientRegistration* project and go to **New** > **GO-BPMN Module**.
   (b) In the dialog, enter the module name `clientregistration`.
   (c) Click **Finish**.

3.6.2 Data

The business data, the client's registration data, must be persisted so they remain available after the business-process finishes. Therefore, they must be store the data as a *shared* record:

1. Right-click the *clientregistration* module and go to **New** > **Data Type Definition**
2. In the dialog, enter the module name **common** and unselect the **executable module** option (it makes no sense to create a model instance over a module with a data model).
3. Click **Finish**.
4. In the displayed editor, create the **RegistrationData** shared Record with the following fields:

- **approved:** Boolean
- **firstName:** String
- **lastName:** String

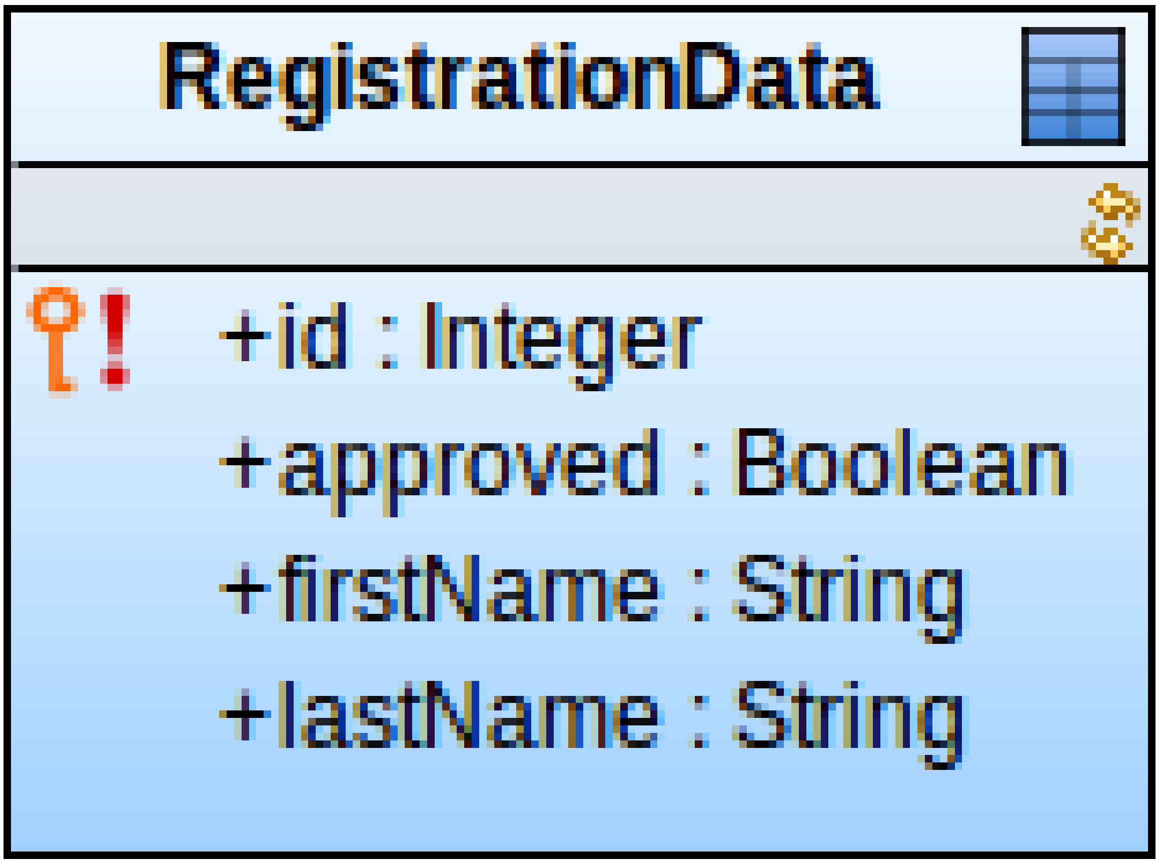


Figure 3.4 RegistrationData record

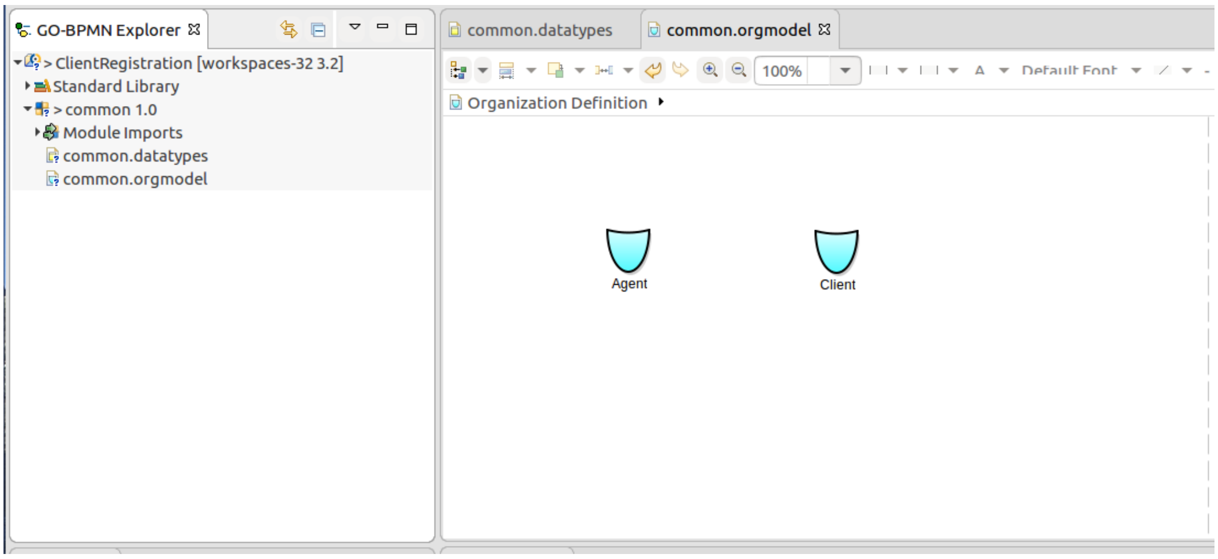
3.6.3 Users

The registration involves two types of users:

- The *client* who enters new registration data and
- the *agent* who approves the registration. To distinguish the users, you will use roles of an organization model. When you run the model, you will assign the roles to users.

Let's create the organization model:

1. In the *common* module, create the organization definition file: *right-click the common module*, go to **New** > **Organization Definition**.
2. Create the roles **Agent** and **Client**.
3. Save.



Now you can use calls `Agent()` and `Client()` to get all users with the given role.

Detailed information on how to design and use organization models is available `here`.

3.6.4 Processes

The registration comprises the entering of the registration data and the data approval approval.

3.6.4.1 Registration

You want a client wants to register, they open the registration form, fill it out, and submit it whenever they want. All occurs on a single page and no further immediate execution logic takes place: no timer or event triggering of actions, no involvement of other users, etc. Therefore, you can implement it as a document.

Create a document definition with a *registrationDocument* document with the following properties:

- Name: `registrationDocument`
- Title: `"Registration"`
- UI definition: `new ClientRegistration()`
- Access rights: `isPersonIn( getCurrentPerson(), Client() )`

Note that you haven't created the *ClientRegistration* UI definition, yet: There is an error reported on the UI Definition expression.

Let's create it:

1. Create the form definition file: right-click the *registration* module, go to **New** > **Form Definition**.
2. In the dialog, enter the name of the file `ClientRegistration`.

3. Make sure the **Use FormComponent-based UI** option *is* selected (The checkbox allows you to select the implementation to be used by the form definition. Two implementations are available: forms and ui. This option selects the `forms` implementation, in which you define behavior of components in the ui definition by primarily writing code as opposed to the `ui` implementation, which requires less coding but are more restrictive).

4. Click **Finish**.

5. In the form editor, design the form:

   (a) Insert a vertical layout.

   (b) Insert a form layout into the vertical layout.

   (c) Into the form layout, insert Text Fields for the first name and last name.

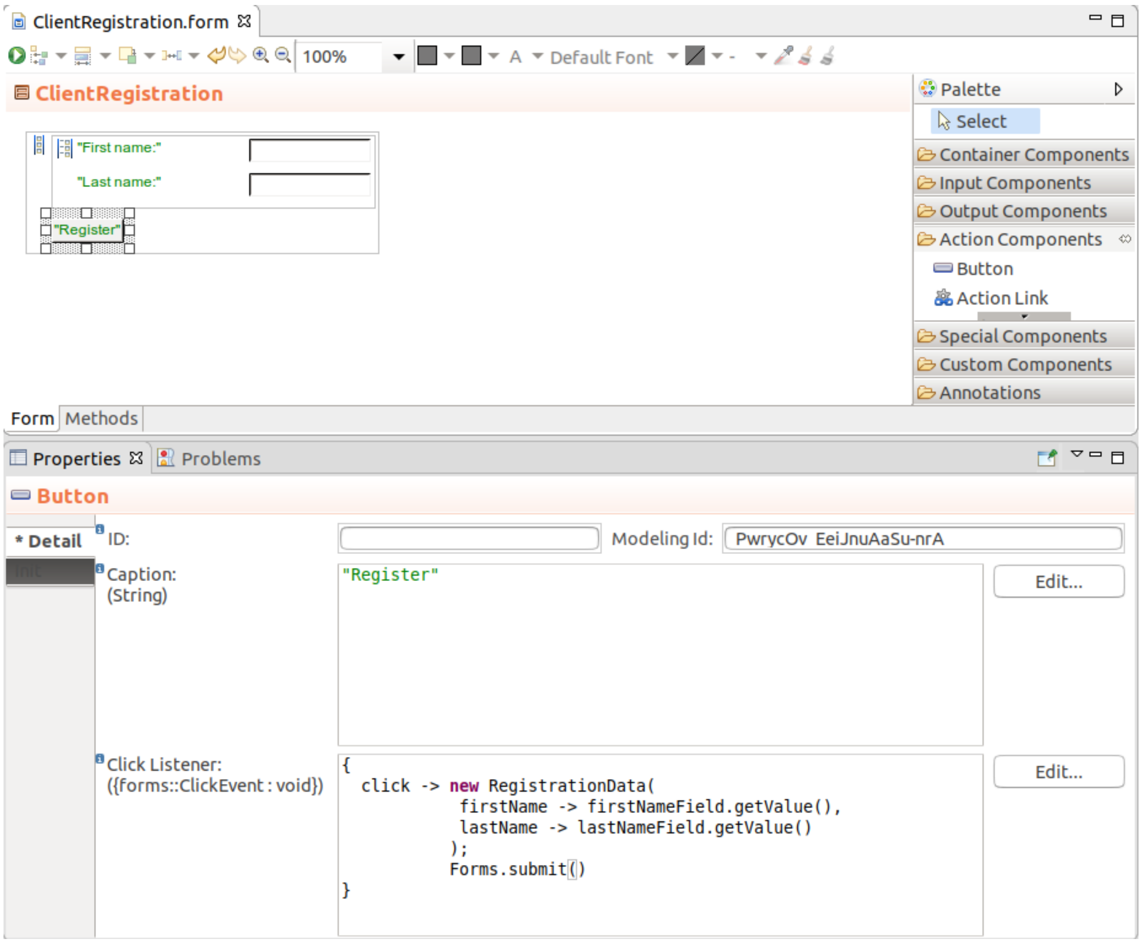
   (d) In the Properties views of the Text Fields:

   - Enter the IDs, `firstNameField` and `lastNameField`.
   - Enter the caption, for example `"First name:"`.
   - Set Binding to Value: this simply sets the value of the field to the literal value, which will not be recalculated on refresh.

6. Insert the *Register* button.

7. Define the expression that will be executed when the user clicks the **Register** button: Open the button properties, and in the *Click Listener* property define an expression which creates a new *RegistrationData* record with the values in the form fields and submits the form:

```
{
  click -> new RegistrationData(
             firstName -> firstNameField.getValue(),
             lastName -> lastNameField.getValue()
           );
           Forms.submit()
}
```



ID:
Modeling Id: PwrycOv_EeiJnuAaSu-nrA

Caption: (String)

```
"Register"
```

Click Listener: ({forms::ClickEvent : void})

```
{
  click -> new RegistrationData(
           firstName -> firstNameField.getValue(),
           lastName -> lastNameField.getValue()
          );
          Forms.submit()
}
```

3.6.4.2 Approval

Now you will create the approval process for the agent, which will behave as follows:

- It starts when the client clicks the *Register* button.
- It displays a form with the user data and the Approve and Reject buttons.
- It stores the data when the agent clicks one of the buttons.

1. Adjust the click listener of the *Register* button In the ClientRegistration form so it triggers this model when clicked (in the next step you will create an approval process that will be triggered by this call):

```
{ click ->
        def RegistrationData registration :=
            new RegistrationData(
               firstName -> firstNameField.getValue(),
               lastName -> lastNameField.getValue()
            );
        createModelInstance(
           synchronous -> true,
           model -> thisModel(),
           //passing the new registration to the model instance:
           processEntity -> registration,
           properties -> null);
        Forms.submit()
}
```

2. Create the process definition file for the approval process:

   (a) Right-click the module, go to **New** > **Process Definition**.

   (b) In the dialog, enter the name of the process `approval`, select the **BPMN-based process**.

   (c) Leave the *Executable* selected.

   If you unselect the Executable option, the process cannot be executed: this option is useful for processes which are meant for documentation purposes.

   (d) Leave the *Instantiate Automatically* option selected.

   You want to instantiate the process automatically when you create the model instance. If you unselect the *Instantiate automatically* option the process instance will not be created and triggered when the model instance is created; such processes are useful when they act as `reusable subprocesses`.

3. Now design the process:

   (a) Insert the None Start Event from the palette.

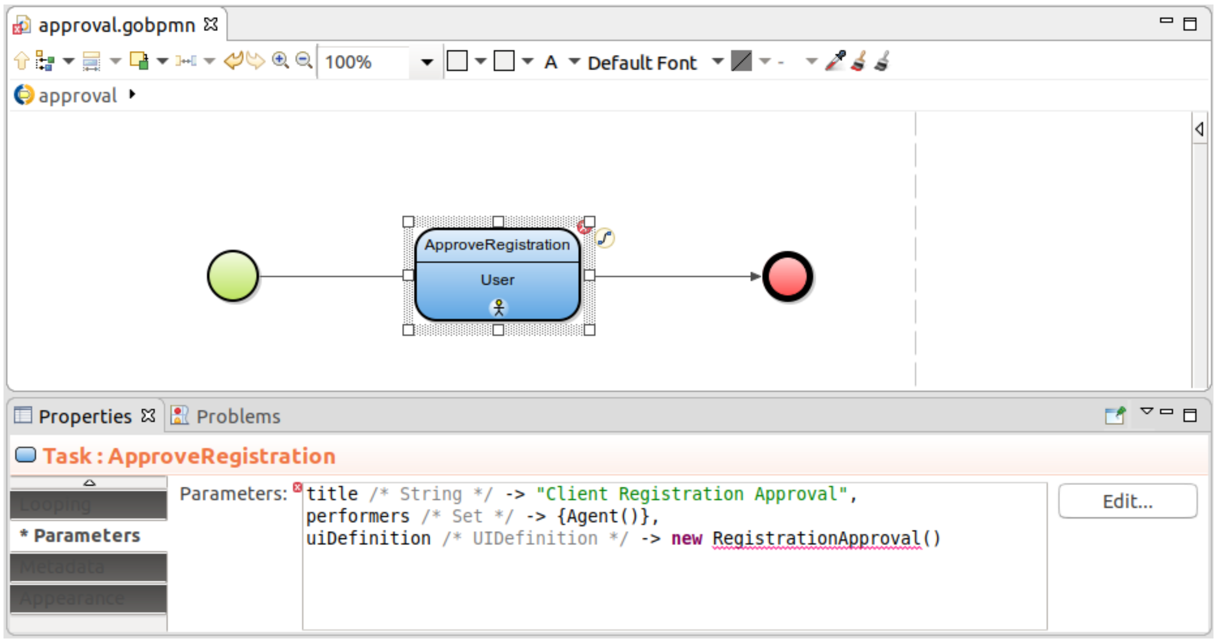
   (b) Drag-and-release the quicklinker of the None Start Event and in the context menu, select the **Task**.

   (c) Select the type of the task: our task requires a user action: select the **User** task.

   (d) Define the task parameters on the Parameters tab of its Properties view:

```
title /* String */ -> "Client Registration Approval",
performers /* Set */ -> {Agent()},
uiDefinition /* UIDefinition */ -> new RegistrationApproval()
```

   (e) Attach the Simple End Event to the task so the process finishes when the user submits the approval.



Task : ApproveRegistration

* Parameters

Parameters:

```
title /* String */ -> "Client Registration Approval",
performers /* Set */ -> {Agent()},
uiDefinition /* UIDefinition */ -> new RegistrationApproval()
```

4. Create the *RegistrationApproval* form:

   (a) Create a form variable *approvalRegistrationData* of type RegistrationData.

   (b) In the methods file of the form, create a form constructor that initializes the variable the value of the model entity (you are passing the entity in the `createModelInstance()` call of the **Register** button):

```
    RegistrationApproval {

  public RegistrationApproval() {
        approvalRegistrationData :=
           getProcessEntity(RegistrationData)
    }
}
```

5. Design the form content:

   (a) Insert a Vertical Layout component.

   (b) Insert Labels for firstName and lastName values into the Vertical Layout.

   (c) Set their Binding to Reference to the field of the *approvalRegistrationRequest* variable.

6. Insert the *Accept* button and implement the approve logic in its Click Listener expression:

```
{ click ->
    approvalRegistrationData.approved := true;
    Forms.submit();
}
```

7. Insert the *Reject* button and implement the reject logic in its Click Listener expression:

```
{ click ->
    approvalRegistrationData.approved := false;
    Forms.submit();
}
```

3.6.5 Running

Now upload your model, assign the Roles to users and follow the execution:

1. Connect PDS to an LSPS Server to which you want to upload your model: Consider using the `PDS Embedded Server`, which runs locally and is stored in your workspace.

   You can check if your PDS is connected to an LSPS Server in the status bar at the very bottom of PDS.

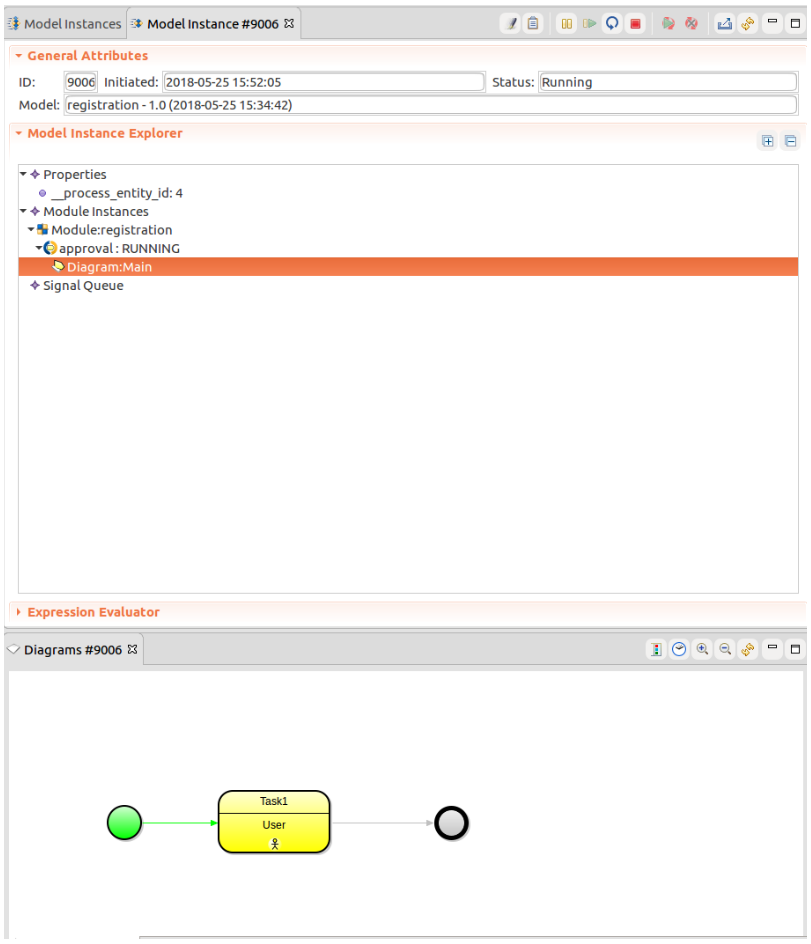
2. Upload the model to the module repository: In the GO-BPMN Explorer, right-click the *clientregistration* module and select **Upload As** > **Model**.

3. Assign the Client role to the admin user.

4. Log in to the Application User Interface as admin and submit the Registration document.

5. Back in the Management perspective, refresh the Model Instances view: a new model instance of registration appears. This was triggered by the `createModelInstance()` of the Register button.

6. Double-click the model instance: in the detail view, expand the model tree and double click the approval process diagram. Note that the id of the process entity is visible under the Properties node.


Model Instances | Model Instance #9006

General Attributes

ID: 9006 Initiated: 2018-05-25 15:52:05 Status: Running

Model: registration - 1.0 (2018-05-25 15:34:42)

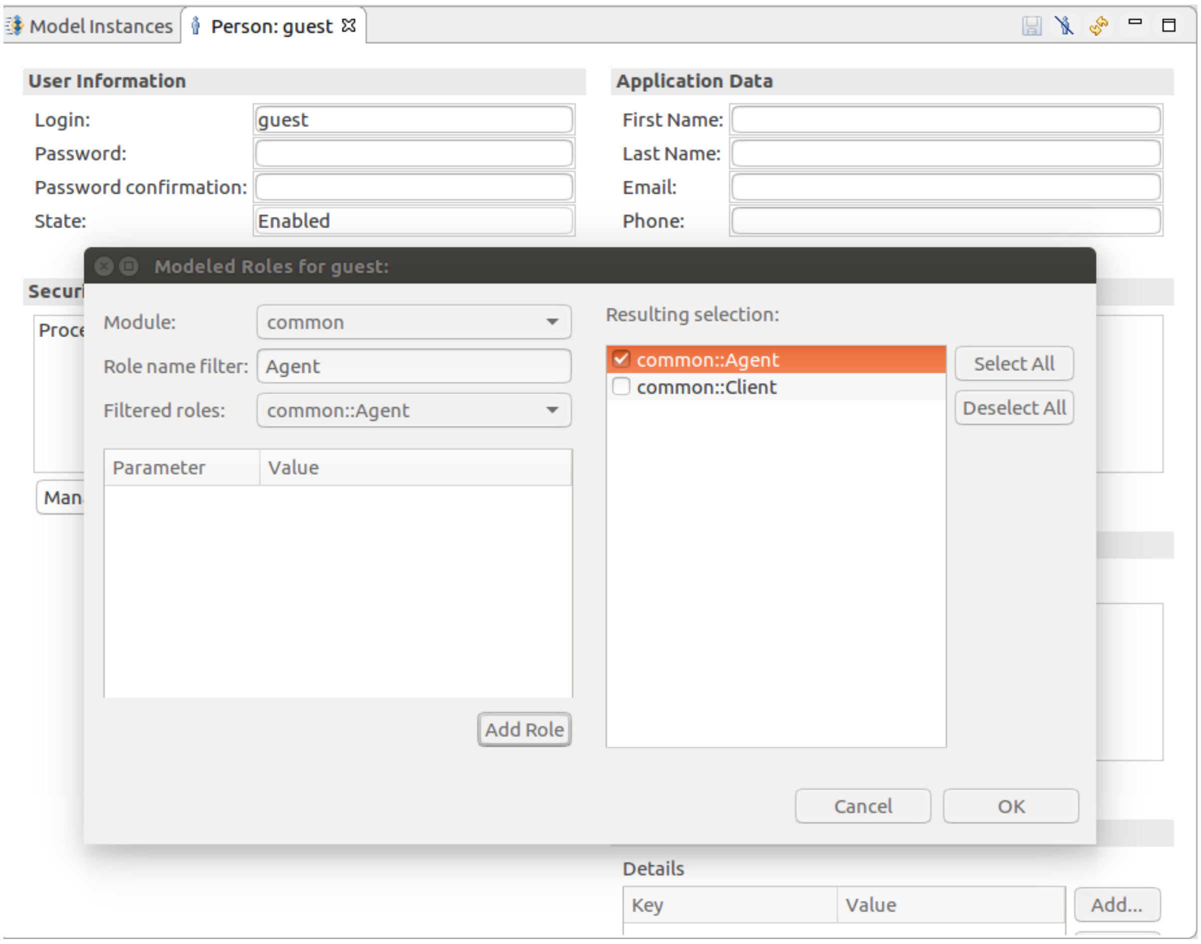
Model Instance Explorer

```
Properties
    __process_entity_id: 4
Module Instances
    Module:registration
        approval : RUNNING
            Diagram:Main
    Signal Queue
```

Expression Evaluator

Diagrams #9006

7. The process instance is stuck on the user task: the task generated a to-do for an end user with the role Agent; however there is no such user. Create a new `guest` user and assign the Agent role to the guest user.



Login:	guest
Password:	
Password confirmation:	
State:	Enabled

Module:	common
Role name filter:	Agent
Filtered roles:	common::Agent

Parameter	Value

Resulting selection:

- [✓ common::Agent]
- common::Client

Key	Value

8. Log in to the Application User Interface as *guest* and submit the approval todo under the Todo menu.

Now go to the Management perspective, refresh the diagram of your process instance: the instance finished and, since there is not other process instance running, also the entire model instance finished. Note that the registration data remained persisted in the system database and you could use it in another model instance.

Proceed to Example Application.

Chapter 4

Example Application

Now that you have a fairly good overview of LSPS, let's go through the entire scenario of creating your customized solution: You will gradually build up a simple customized ticketing system. Each chapter introduces a user story, which is followed by instructions on how to implement it.

In the first part, we implement our business process; in the Application second part, we create a customized application for the model.

You assume you have been through the previous academy chapters and have a good understanding of such fundamental concepts as *module*, *shared record*, *form*, *document*, *to-do*, etc. If you have not done so, please, return to the introductory part.

Proceed to Model.

4.1 Model

You will design a business model that will distribute work between by two types of users, the customer and the support specialist:

- Customers create a ticket at any time and answer to ticket comments.
- Support specialists will:
  - open the ticket,
  - add a comment after the customer does,
  - resolve the ticket,
  - view a list of their and new open tickets.

In addition, tickets will expire if the customer does not add a comment within a period of time.

4.1.1 Prerequisites

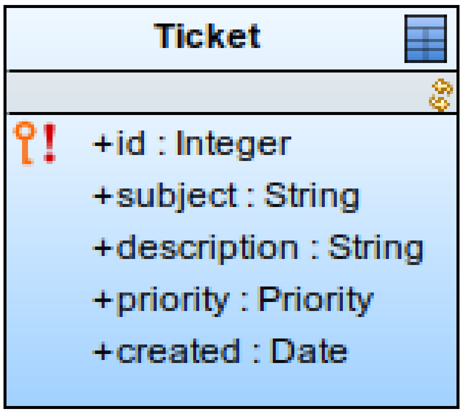
1. Open PDS with a new workspace.
2. Create the GO-BPMN project *ticket-application-model*.
3. Optionally, put the project under version control (you will deal with version control in the next part in detail).

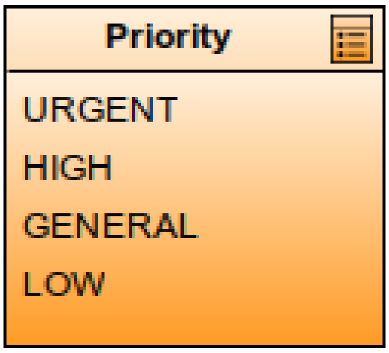
4.1.2 Raising a Trouble Ticket

User story: A customer raises a trouble ticket with an issue.

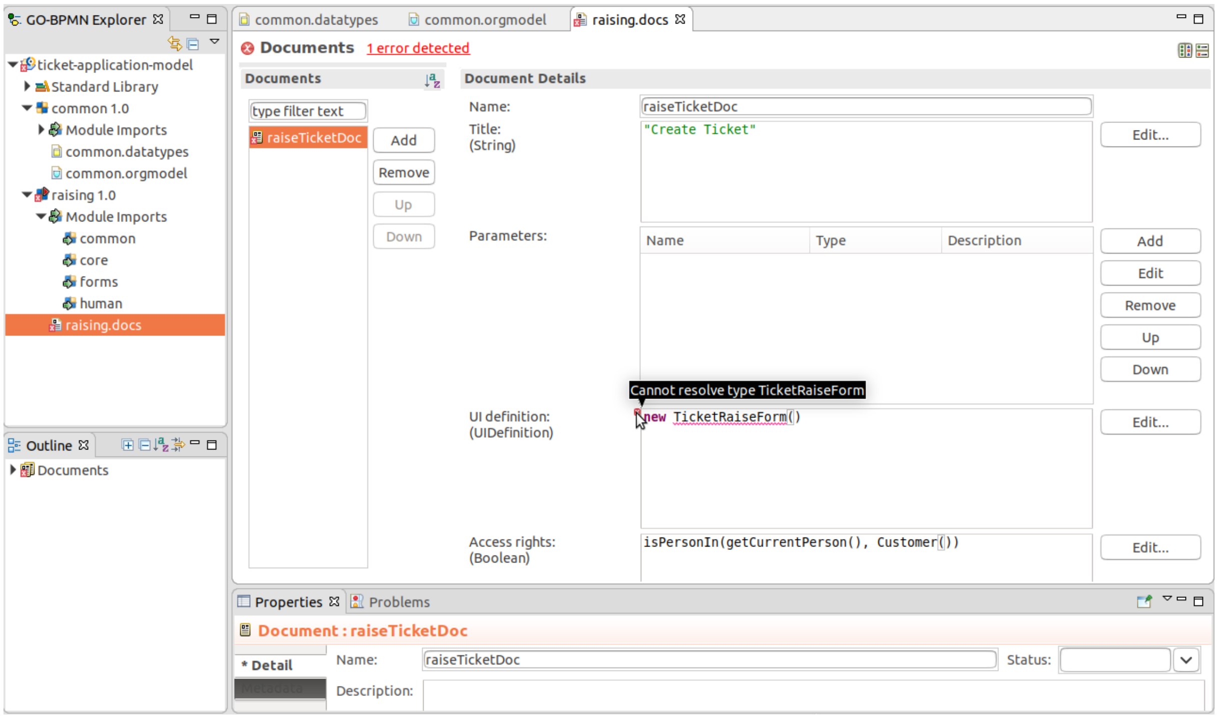
The customer creates a ticket with details, such as, subject, description, date, and priority.

1. Create a non-executable *common* module with the following:
   - a data type definition with the *Ticket* shared record Define the type of the *priority* field as the Priority type and create *Priority* enumeration.





   - an organization definition with a *Customer* role
2. Create a non-executable module, *raising*, for the resources for the ticket raising.
   (a) Import the *common* module.
   (b) Create a document definition with a Document available only to users with the Customer role: Set the *Access rights* expression to `isPersonIn(getCurrentPerson(), Customer())` and *UI Definition* set to `new TicketRaiseForm()`.


GO-BPMN Explorer
ticket-application-model
Standard Library
common 1.0
Module Imports
common.datatypes
common.orgmodel
raising 1.0
Module Imports
common
core
forms
human
raising.docs

Outline
Documents

common.datatypes | common.orgmodel | raising.docs

Documents 1 error detected

Documents
type filter text
raiseTicketDoc
Add | Remove | Up | Down

Document Details
Name: raiseTicketDoc
Title: (String) "Create Ticket"
Parameters: Name | Type | Description
Cannot resolve type TicketRaiseForm
UI definition: (UIDefinition) new TicketRaiseForm()
Access rights: (Boolean) isPersonIn(getCurrentPerson(), Customer())

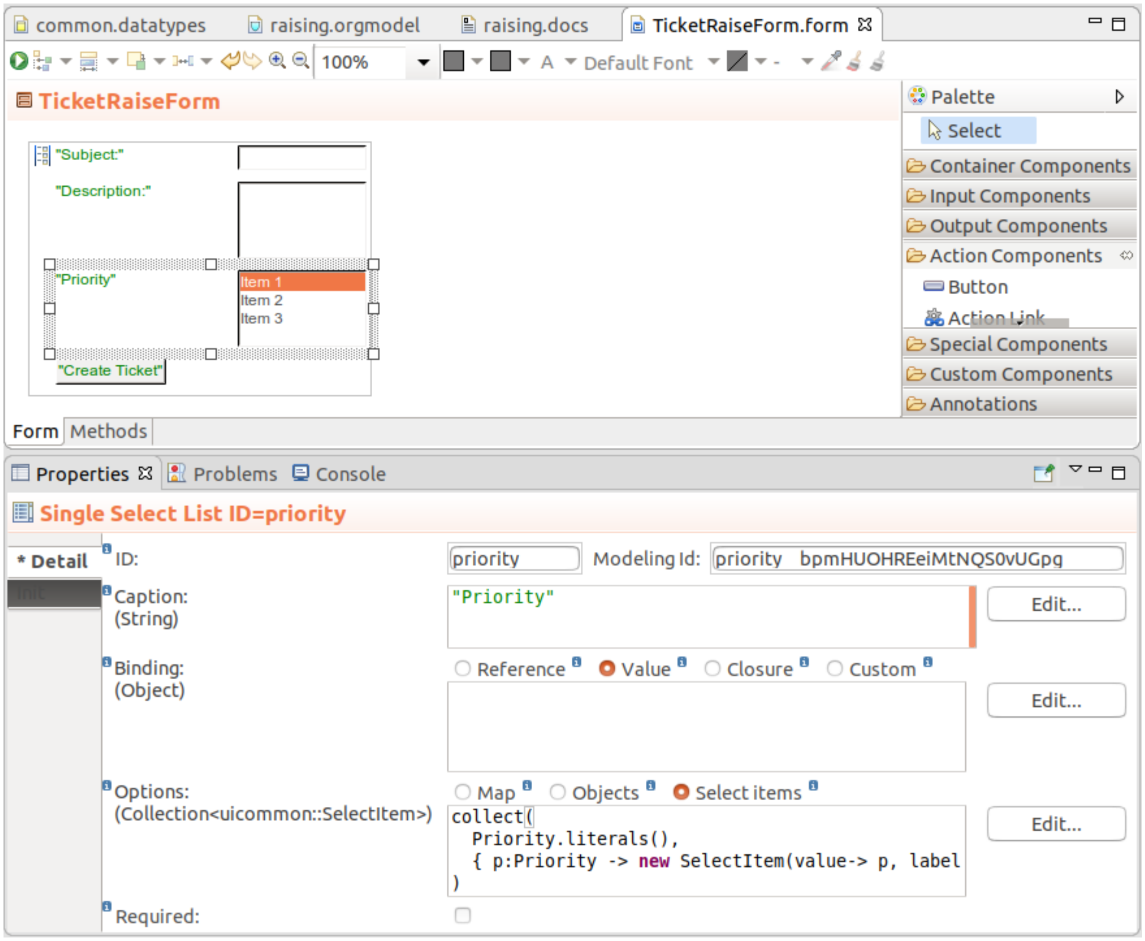
Properties | Problems
Document : raiseTicketDoc
Detail
Name: raiseTicketDoc
Status:
Description:

Figure 4.1 Resulting structure with document details

(c) Create the missing form *TicketRaiseForm* with the components:

- Form Layout parent component
- Text Field with ID `subject` and `Value` binding
- Text Area with ID `description` and `Value` binding
- Single Select List with ID `priority`, `Value` binding, and `Select items` options populated with the expression:

```
collect(
  //returns all literals of the Priority enumeration:
  Priority.literals(),
  // Over each priority literal, create a SelectItem:
  { p:Priority -> new SelectItem(value-> p, label -> p.toString())}
)
```



ID: priority Modeling Id: priority bpmHUOHREeiMtNQS0vUGpg

Caption: (String) "Priority"

Binding: (Object) Reference [✓ Value] Closure Custom

Options: (Collection<uicommon::SelectItem>) Map Objects [✓ Select items]

```
collect(
  Priority.literals(),
  { p:Priority -> new SelectItem(value-> p, label
)
```

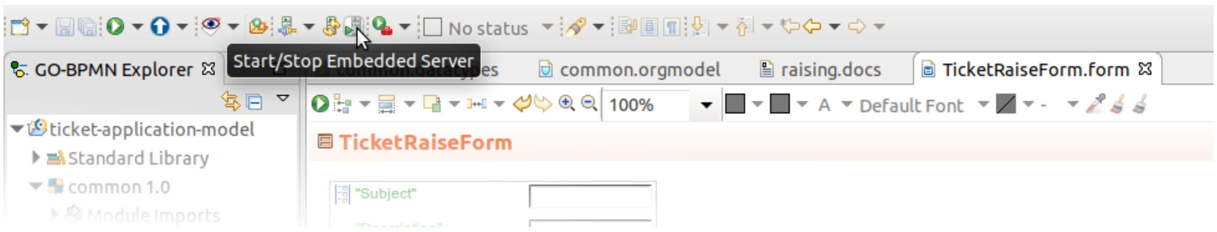
Required: ☐

Figure 4.2 Ticket raising form with the detail of the Priority field in the Properties

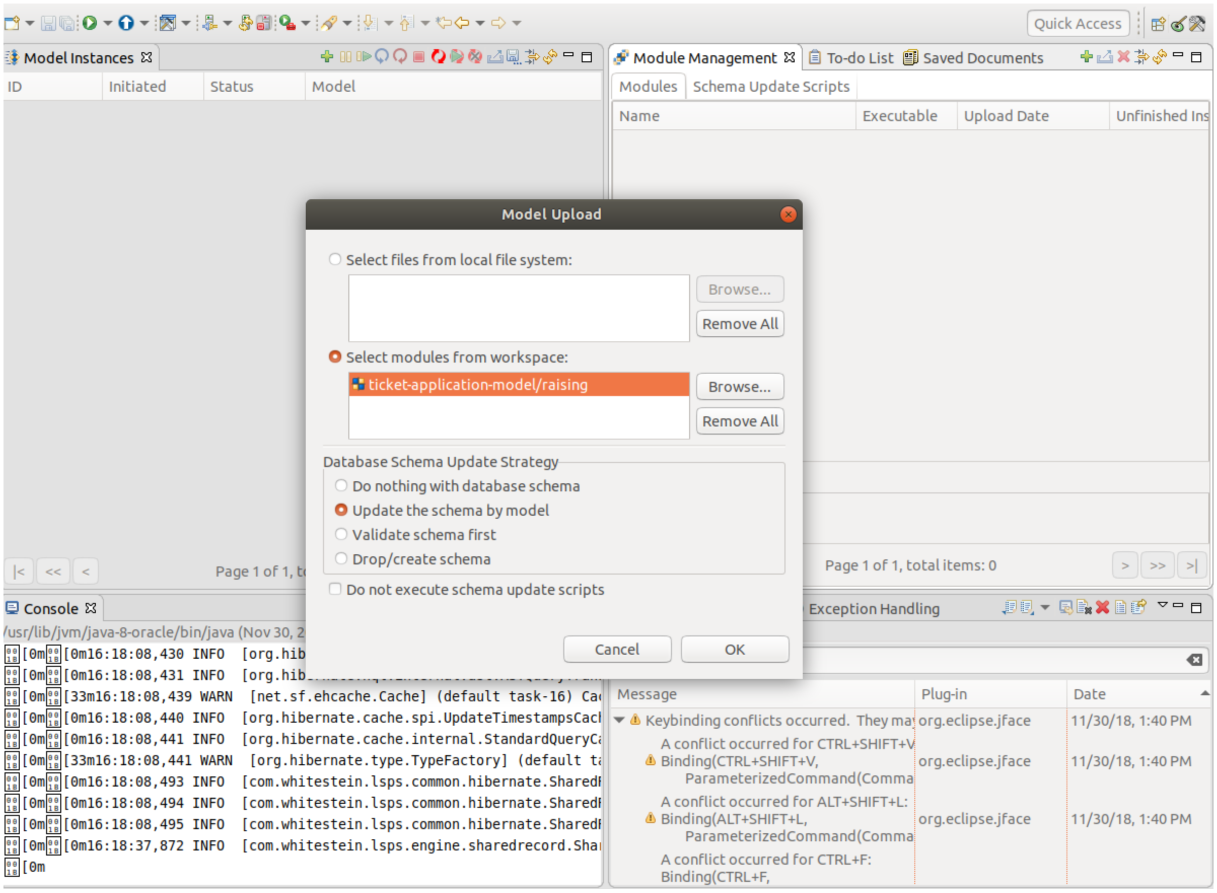
- Button component with the name **Create Ticket** with the following *Click Listener* expression:

```
//collects input from from, creates a new ticket,
//and submits the form:
{ e ->
   new Ticket(
    created -> now(),
    //subject of the new ticket is set to the value
    //in the subject Text Input component:
    subject -> subject.getValue(),
    description -> description.getValue(),
    priority -> priority.getValue()?? Priority.GENERAL as Priority
  );
  Forms.submit()
}
```

3. Run the PDS Embedded Server and switch to the *Management* perspective.



   (a) Upload the *raising* module.



Model Upload

- Select files from local file system:
- [✓ Select modules from workspace:] ticket-application-model/raising

Database Schema Update Strategy
- Do nothing with database schema
- [✓ Update the schema by model]
- Validate schema first
- Drop/create schema

Do not execute schema update scripts

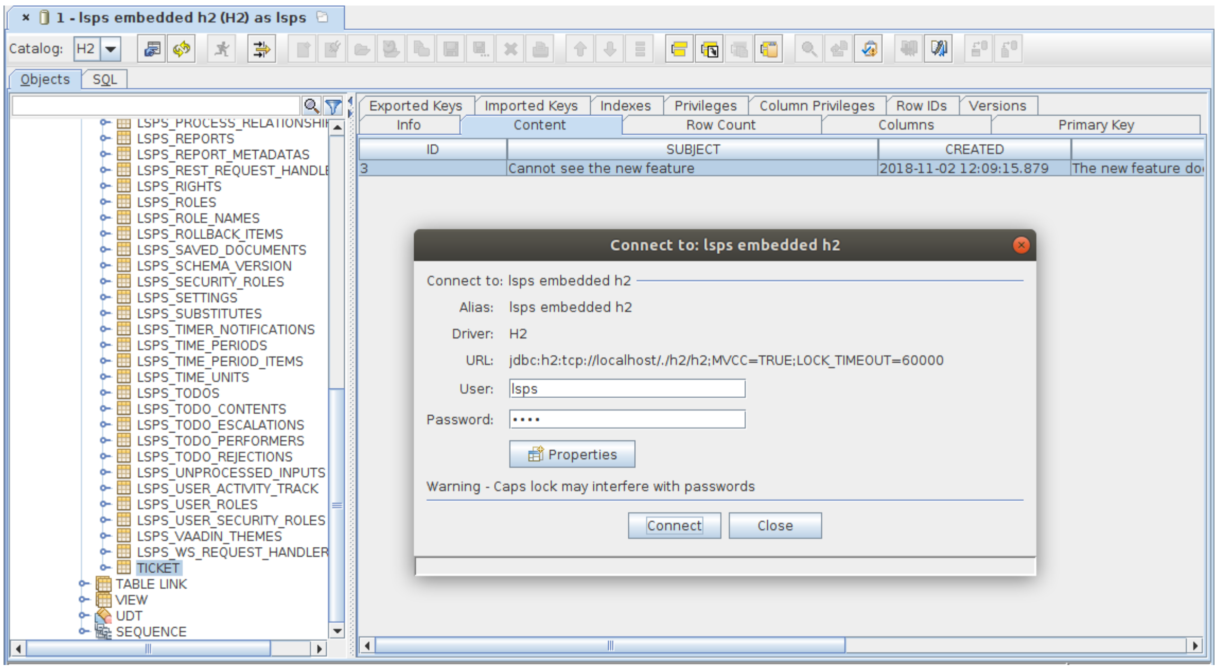
```
/usr/lib/jvm/java-8-oracle/bin/java (Nov 30, 2
[0m[0m16:18:08,430 INFO  [org.hib
[0m[0m16:18:08,431 INFO  [org.hib
[0m[33m16:18:08,439 WARN  [net.sf.ehcache.Cache] (default task-16) Ca
[0m[0m16:18:08,440 INFO  [org.hibernate.cache.spi.UpdateTimestampsCac
[0m[0m16:18:08,441 INFO  [org.hibernate.cache.internal.StandardQueryC
[0m[33m16:18:08,441 WARN  [org.hibernate.type.TypeFactory] (default t
[0m[0m16:18:08,493 INFO  [com.whitestein.lsps.common.hibernate.Shared
[0m[0m16:18:08,494 INFO  [com.whitestein.lsps.common.hibernate.Shared
[0m[0m16:18:08,495 INFO  [com.whitestein.lsps.common.hibernate.Shared
[0m[0m16:18:37,872 INFO  [com.whitestein.lsps.engine.sharedrecord.Sha
[0m
```

Message	Plug-in	Date
Keybinding conflicts occurred. They ma	org.eclipse.jface	11/30/18, 1:40 PM
A conflict occurred for CTRL+SHIFT+V Binding(CTRL+SHIFT+V, ParameterizedCommand(Comma	org.eclipse.jface	11/30/18, 1:40 PM
A conflict occurred for ALT+SHIFT+L: Binding(ALT+SHIFT+L, ParameterizedCommand(Comma	org.eclipse.jface	11/30/18, 1:40 PM
A conflict occurred for CTRL+F: Binding(CTRL+F,		

   (b) Assign the *Customer* role to a person: make sure to *save* the changes.

   (c) Log in to the Application User Interface (by default, `localhost:8080/lsps-application`) as the person and create a ticket.

4. Optionally, open your database client and check that the ticket table contains the new ticket. The embedded database is an H2 database with the user `lsps` and password `lsps`.


1 - lsps embedded h2 (H2) as lsps

Catalog: H2

Objects | SQL

LSPS_PROCESS_RELATIONSHIP
LSPS_REPORTS
LSPS_REPORT_METADATAS
LSPS_REST_REQUEST_HANDLE
LSPS_RIGHTS
LSPS_ROLES
LSPS_ROLE_NAMES
LSPS_ROLLBACK_ITEMS
LSPS_SAVED_DOCUMENTS
LSPS_SCHEMA_VERSION
LSPS_SECURITY_ROLES
LSPS_SETTINGS
LSPS_SUBSTITUTES
LSPS_TIMER_NOTIFICATIONS
LSPS_TIME_PERIODS
LSPS_TIME_PERIOD_ITEMS
LSPS_TIME_UNITS
LSPS_TODOS
LSPS_TODO_CONTENTS
LSPS_TODO_ESCALATIONS
LSPS_TODO_PERFORMERS
LSPS_TODO_REJECTIONS
LSPS_UNPROCESSED_INPUTS
LSPS_USER_ACTIVITY_TRACK
LSPS_USER_ROLES
LSPS_USER_SECURITY_ROLES
LSPS_VAADIN_THEMES
LSPS_WS_REQUEST_HANDLER
TICKET
TABLE LINK
VIEW
UDT
SEQUENCE

Exported Keys | Imported Keys | Indexes | Privileges | Column Privileges | Row IDs | Versions
Info | Content | Row Count | Columns | Primary Key

ID	SUBJECT	CREATED	
3	Cannot see the new feature	2018-11-02 12:09:15.879	The new feature do

Connect to: lsps embedded h2

Connect to: lsps embedded h2
Alias: lsps embedded h2
Driver: H2
URL: jdbc:h2:tcp://localhost/./h2/h2;MVCC=TRUE;LOCK_TIMEOUT=60000
User: lsps
Password: ••••

Properties

Warning - Caps lock may interfere with passwords

Connect | Close

Figure 4.3 Details of embedded database connection and the new ticket table

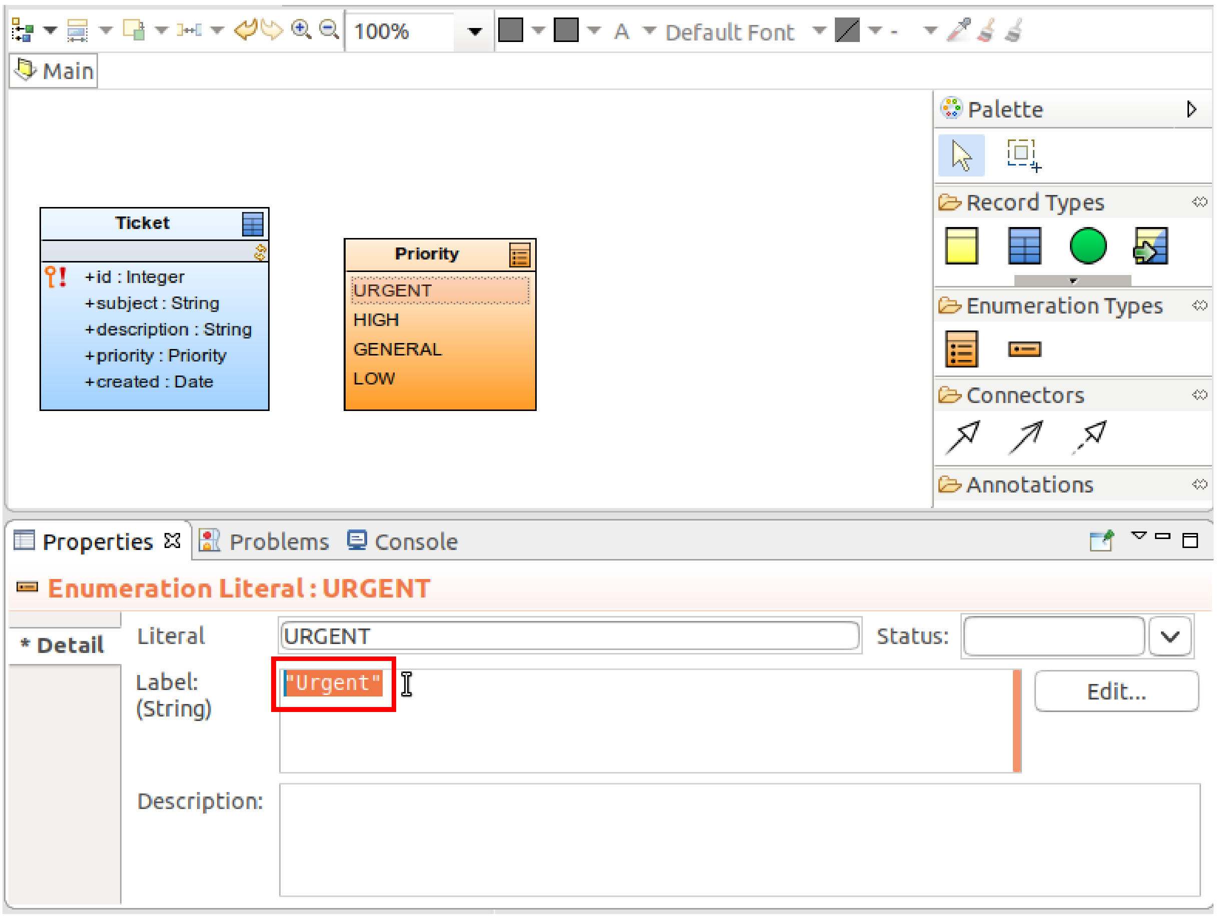
4.1.2.1 Summary

You have done the following:

- You have created the shared record *Ticket* so that the ticket data is persisted.
- You have created a customer role so you can work with a particular set of users.
- You have created document `roleTicketDoc` accessible only to a customer so that customer users can enter a new ticket as a new instance of the Ticket shared record.
- You have assigned the customer role to a user.
- You have uploaded the module with the document.
- You have create a new ticket as the customer user.

4.1.2.2 Tips and Takeaways

- Organize your data in modules so you can reuse them when necessary: In the example, the data type and organization definition are stored in the *common* module. This will allow you to reuse the definitions in new modules.
- Define labels for your records and their fields: you can then call `getLabel()` to get the label, for example, in the caption of a form component and keep labels in your forms consistent. This can be especially helpful when `localization of model resources` is required.



Ticket
+id : Integer
+subject : String
+description : String
+priority : Priority
+created : Date

Priority
URGENT
HIGH
GENERAL
LOW

Enumeration Literal : URGENT

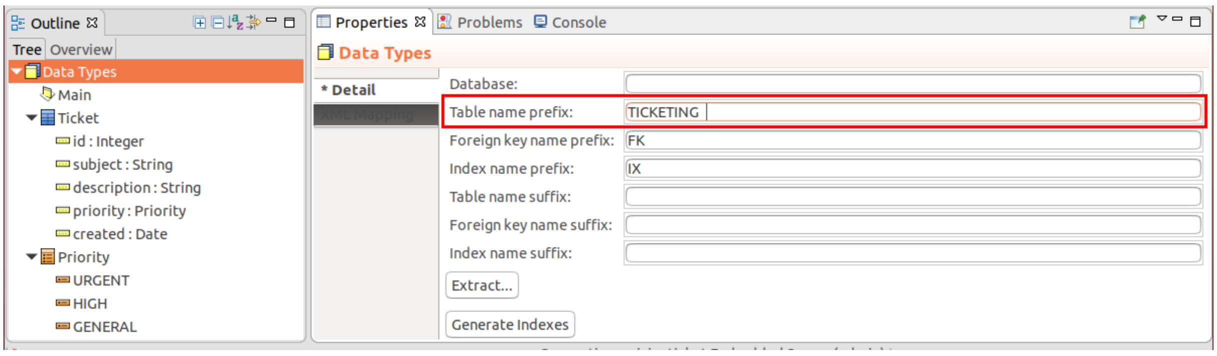
Literal	URGENT	Status:	
Label: (String)	"Urgent"		
Description:			

Figure 4.4 Label of an enumeration field

Options expression of the Single-Select List that uses enumeration labels

```
collect(
  Priority.literals(),
  { p:Priority -> new SelectItem(value-> p, label -> p.getLabel())}
)
```

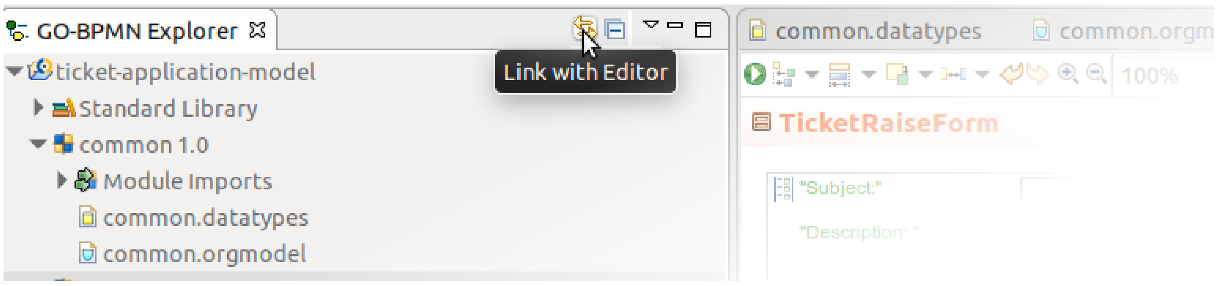
- Set a prefix for the tables of your data type definition so you can easily identity its database tables.



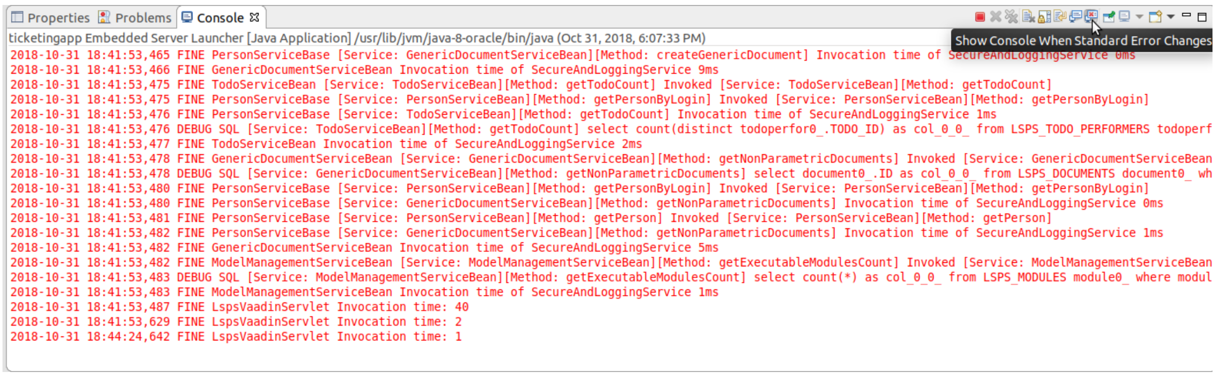
Data Types

Database:	
Table name prefix:	TICKETING
Foreign key name prefix:	FK
Index name prefix:	IX
Table name suffix:	
Foreign key name suffix:	
Index name suffix:	

- Click the *Link with Editor* arrow in *GO-BPMN Explorer* so that the file with the currently edited resource is always selected.

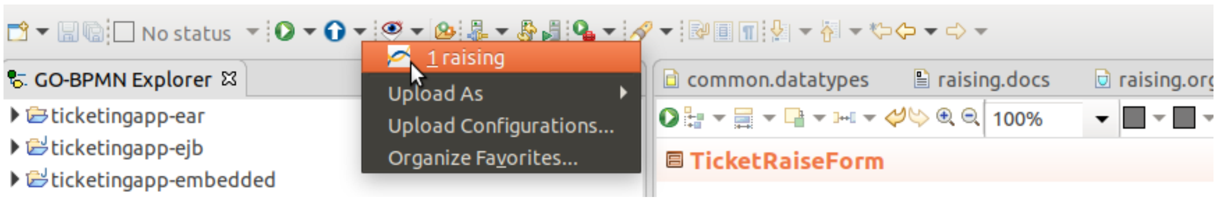


- Tweak the form presentation before you resort to `custom styling`:
  - Expand components in the layout by calling `c.setWidthFull()` on the *Init* tab of each input component.
  - Disallow empty selection for the Priority select box by calling `c.setNullSelection↩ Allowed(false).`
- Set the Console view so it gains focus only when an error is raised to prevent it from stealing focus unnecessarily .

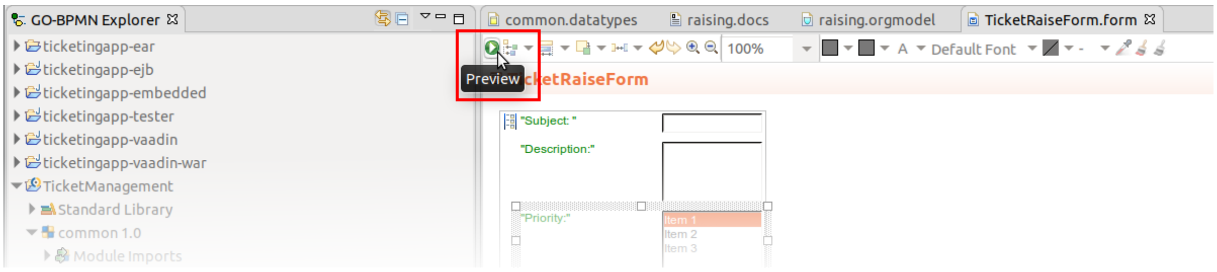


```
ticketingapp Embedded Server Launcher [Java Application] /usr/lib/jvm/java-8-oracle/bin/java (Oct 31, 2018, 6:07:33 PM)
2018-10-31 18:41:53,465 FINE PersonServiceBase [Service: GenericDocumentServiceBean][Method: createGenericDocument] Invocation time of SecureAndLoggingService 0ms
2018-10-31 18:41:53,466 FINE GenericDocumentServiceBean Invocation time of SecureAndLoggingService 9ms
2018-10-31 18:41:53,475 FINE TodoServiceBean [Service: TodoServiceBean][Method: getTodoCount] Invoked [Service: TodoServiceBean][Method: getTodoCount]
2018-10-31 18:41:53,475 FINE PersonServiceBase [Service: PersonServiceBean][Method: getPersonByLogin] Invoked [Service: PersonServiceBean][Method: getPersonByLogin]
2018-10-31 18:41:53,476 FINE PersonServiceBase [Service: TodoServiceBean][Method: getTodoCount] Invocation time of SecureAndLoggingService 1ms
2018-10-31 18:41:53,476 DEBUG SQL [Service: TodoServiceBean][Method: getTodoCount] select count(distinct todoperfor0_.TODO_ID) as col_0_0_ from LSPS_TODO_PERFORMERS todoperf
2018-10-31 18:41:53,477 FINE TodoServiceBean Invocation time of SecureAndLoggingService 2ms
2018-10-31 18:41:53,478 FINE GenericDocumentServiceBean [Service: GenericDocumentServiceBean][Method: getNonParametricDocuments] Invoked [Service: GenericDocumentServiceBean
2018-10-31 18:41:53,478 DEBUG SQL [Service: GenericDocumentServiceBean][Method: getNonParametricDocuments] select document0_.ID as col_0_0_ from LSPS_DOCUMENTS document0_ wh
2018-10-31 18:41:53,480 FINE PersonServiceBase [Service: PersonServiceBean][Method: getPersonByLogin] Invoked [Service: PersonServiceBean][Method: getPersonByLogin]
2018-10-31 18:41:53,480 FINE PersonServiceBase [Service: GenericDocumentServiceBean][Method: getNonParametricDocuments] Invocation time of SecureAndLoggingService 0ms
2018-10-31 18:41:53,481 FINE PersonServiceBase [Service: PersonServiceBean][Method: getPerson] Invoked [Service: PersonServiceBean][Method: getPerson]
2018-10-31 18:41:53,482 FINE PersonServiceBase [Service: GenericDocumentServiceBean][Method: getNonParametricDocuments] Invocation time of SecureAndLoggingService 1ms
2018-10-31 18:41:53,482 FINE GenericDocumentServiceBean Invocation time of SecureAndLoggingService 5ms
2018-10-31 18:41:53,482 FINE ModelManagementServiceBean [Service: ModelManagementServiceBean][Method: getExecutableModulesCount] Invoked [Service: ModelManagementServiceBean
2018-10-31 18:41:53,483 DEBUG SQL [Service: ModelManagementServiceBean][Method: getExecutableModulesCount] select count(*) as col_0_0_ from LSPS_MODULES module0_ where modul
2018-10-31 18:41:53,483 FINE ModelManagementServiceBean Invocation time of SecureAndLoggingService 1ms
2018-10-31 18:41:53,487 FINE LspsVaadinServlet Invocation time: 40
2018-10-31 18:41:53,629 FINE LspsVaadinServlet Invocation time: 2
2018-10-31 18:44:24,642 FINE LspsVaadinServlet Invocation time: 1
```

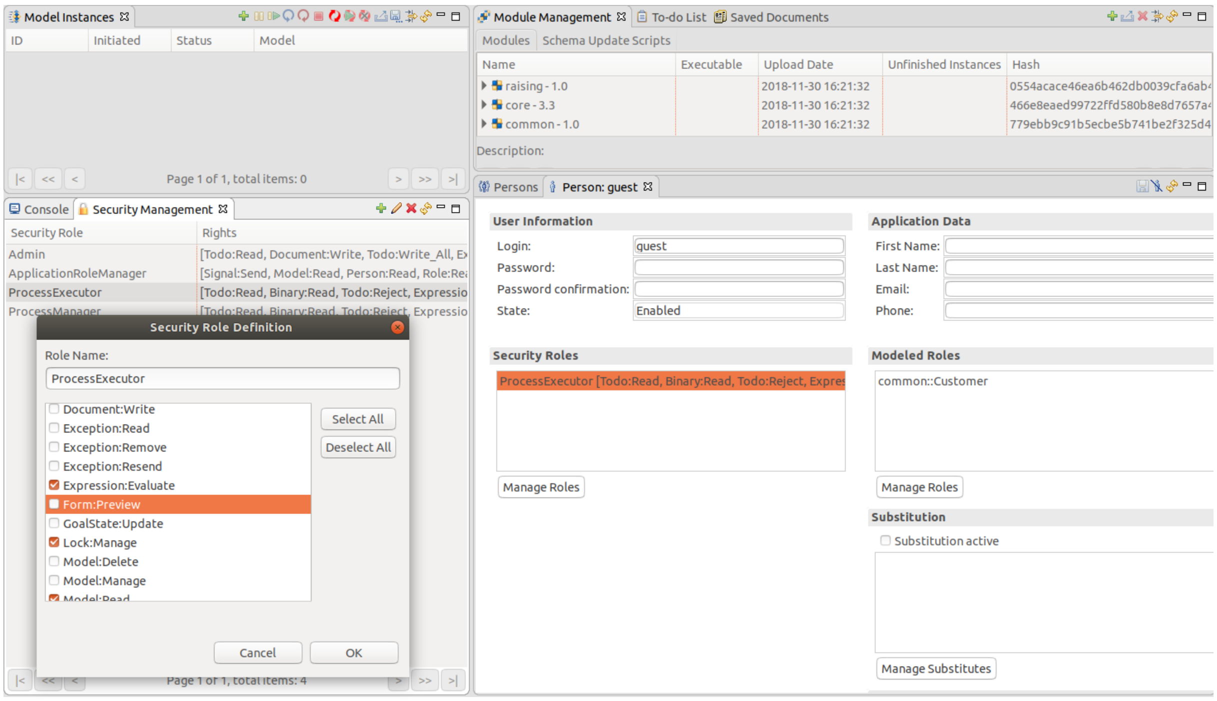
- Upload modules from workspace directly from GO-BPMN Explorer: right-click the module and go to **Upload As** > **Model**: To upload the most-recently uploaded module again, you can then use the Upload menu button.



- Display a preview of your form by clicking the **Preview** button in the form editor: This creates a dummy model instance with your form.



Note that your user must have a security role with the `Form:Preview` right to display such previews; You can modify the rights of a security role in the *Security Management* view in the Management perspective.



ID	Initiated	Status	Model

Page 1 of 1, total items: 0

Name	Executable	Upload Date	Unfinished Instances	Hash
raising - 1.0		2018-11-30 16:21:32		0554acace46ea6b462db0039cfa6ab
core - 3.3		2018-11-30 16:21:32		466e8eaed99722ffd580b8e8d7657a
common - 1.0		2018-11-30 16:21:32		779ebb9c91b5ecbe5b741be2f325d4

Description:

Security Role	Rights
Admin	[Todo:Read, Document:Write, Todo:Write_All, E
ApplicationRoleManager	[Signal:Send, Model:Read, Person:Read, Role:Re
ProcessExecutor	[Todo:Read, Binary:Read, Todo:Reject, Expressio
ProcessManager	[Todo:Read, Binary:Read, Todo:Reject, Expressio

Page 1 of 1, total items: 4

Role Name: ProcessExecutor

- ☐ Document:Write
- ☐ Exception:Read
- ☐ Exception:Remove
- ☐ Exception:Resend
- ☑ Expression:Evaluate
- ☐ Form:Preview
- ☐ GoalState:Update
- ☑ Lock:Manage
- ☐ Model:Delete
- ☐ Model:Manage
- ☑ Model:Read

User Information

Login:	guest
Password:	
Password confirmation:	
State:	Enabled

Application Data

First Name:	
Last Name:	
Email:	
Phone:	

Security Roles

ProcessExecutor [Todo:Read, Binary:Read, Todo:Reject, Expres

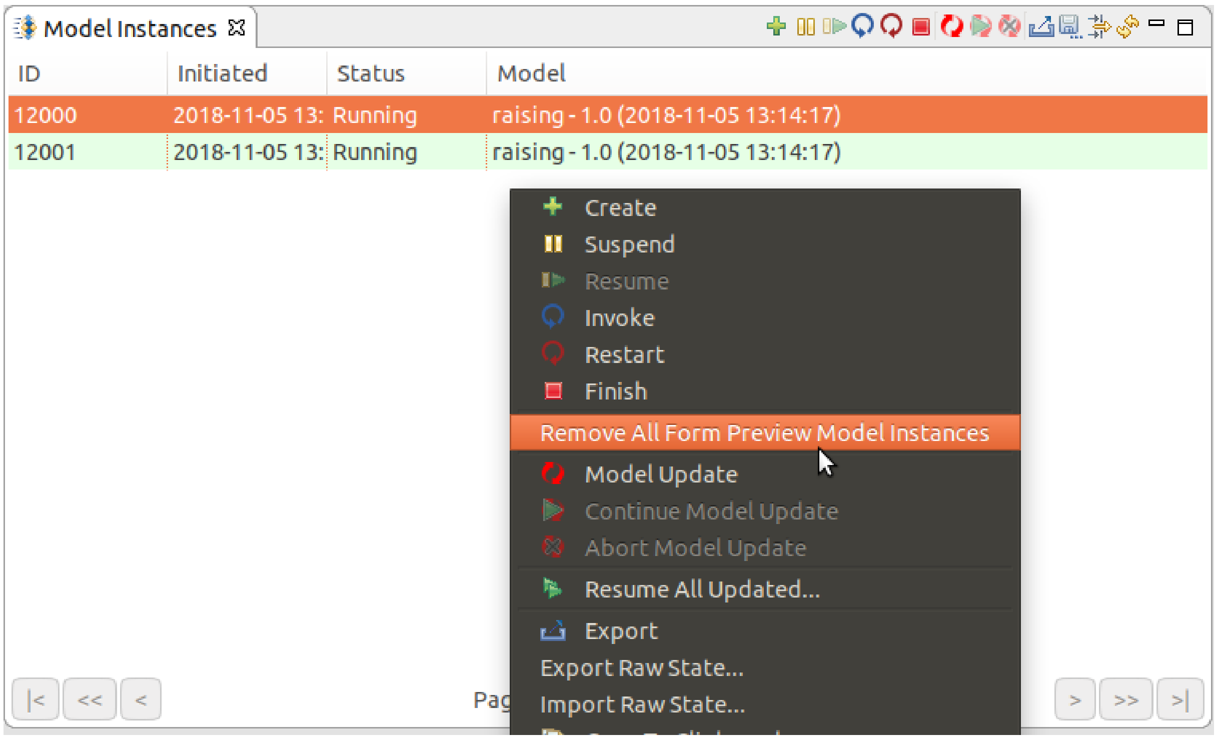
Modeled Roles

common::Customer

Substitution

☐ Substitution active

- To remove the model instances created by form previews, go to the Management perspective, right-click the *Model Instance* view and select **Remove All Form Preview Model Instances**.



ID	Initiated	Status	Model
12000	2018-11-05 13:	Running	raising - 1.0 (2018-11-05 13:14:17)
12001	2018-11-05 13:	Running	raising - 1.0 (2018-11-05 13:14:17)

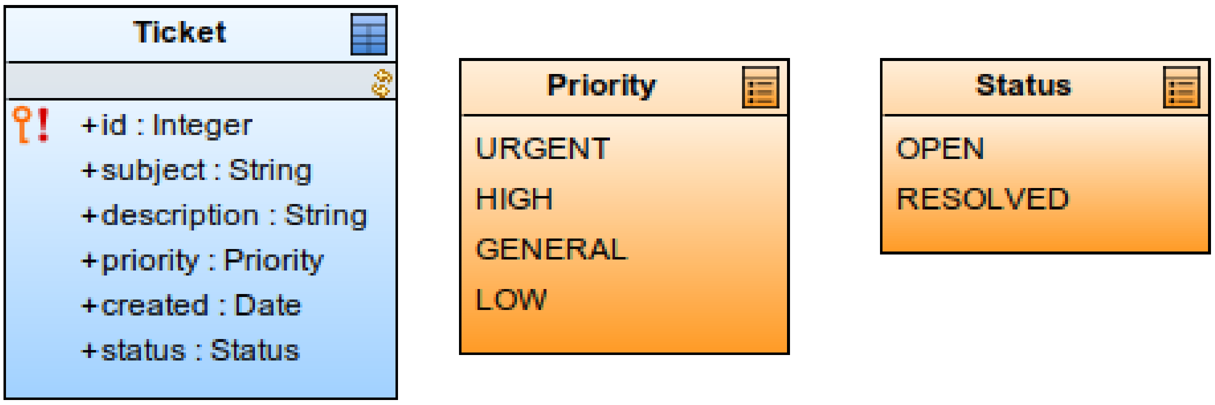
4.1.3 Resolving a Ticket

User story: A support specialist resolves the ticket.

When the customer submits the ticket, the status of the created ticket becomes *open* and the server creates a to-do for the support specialists.

1. In the *common* module:

   (a) Add field `status` of type *Status* to Ticket and create the *Status* enumeration. Set the labels for its literals.

   (b) In the organization definition, create the *SupportSpecialist* role.

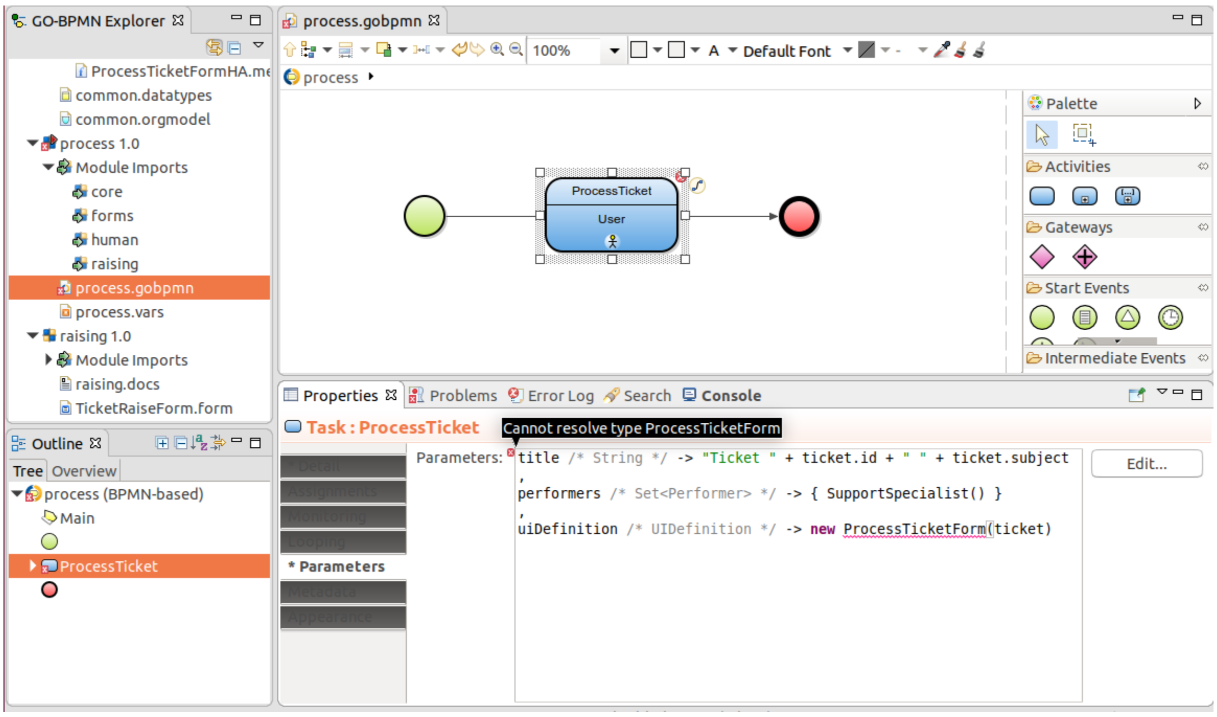
2. In the *TicketRaiseForm*, create the new ticket in the status *open*: adjust the *Click Listener* expression of the *Create Ticket* button:

```
{ e ->
   def Ticket newTicket := new Ticket(
    created -> now(),
    subject -> subject.getValue(),
    description -> description.getValue(),
    //sets priority to GENERAL if not set:
    priority -> priority.getValue()?? Priority.GENERAL as Priority,
    //added this line:
    status -> Status.OPEN
  );
  Forms.submit()
}
```

3. Edit the *Click Listener* expression further so it creates a model instance; pass the new ticket to the model instance:

```
{ e ->
   //saving new ticket in the newTicket expression variable:
   def Ticket newTicket := new Ticket(
    created -> now(),
    subject -> subject.getValue(),
    description -> description.getValue(),
    priority -> priority.getValue() as Priority,
    status -> Status.OPEN
  );
  //creating model instance with the newTicket as its process entity
  //(it is passed to the model instance as a special property):
  createModelInstance(
     synchronous -> false,
     //you will create the executable module *process* in the next step:
     model -> getModel("process"),
     processEntity -> newTicket,
     properties -> null);
  Forms.submit()
}
```

4. Create an executable module named *process* with the following:
   - *raising* module import (this will recursively import *common*)
   - global variable definition with the *ticket* variable initialized to the processEntity of type Ticket; set its initial value to `getProcessEntity(Ticket)`.
   - BPMN-based process with a User task for support specialists



Cannot resolve type ProcessTicketForm

Parameters:
```
title /* String */ -> "Ticket " + ticket.id + " " + ticket.subject
,
performers /* Set<Performer> */ -> { SupportSpecialist() }
,
uiDefinition /* UIDefinition */ -> new ProcessTicketForm(ticket)
```

5. In *common*, create a forms directory with the missing *ProcessTicketForm* that will display the *ticket* details:
   - Create the ticket form variable.
   - Store the *ticket* passed by the User task in the form variable:

```
//content of the ProcessTicketForm.methods file:
ProcessTicketForm {
//constructor that sets ticket form variable to the passed ticket argument:
  public ProcessTicketForm(Ticket ticket){
     this.ticket := ticket
  }
}
```

   - Create content:
     - Form Layout with the ticket details
       - Label with Value binding set to `ticket.subject`
       - Label with Value binding set to `ticket.description`
       - Label with Value binding set to `ticket.priority.getLabel()`
     - Button with the listener expression that sets the ticket as resolved and submits the to-do:

```
{ e ->
    ticket.status := Status.RESOLVED;
    Forms.submit();
}
```

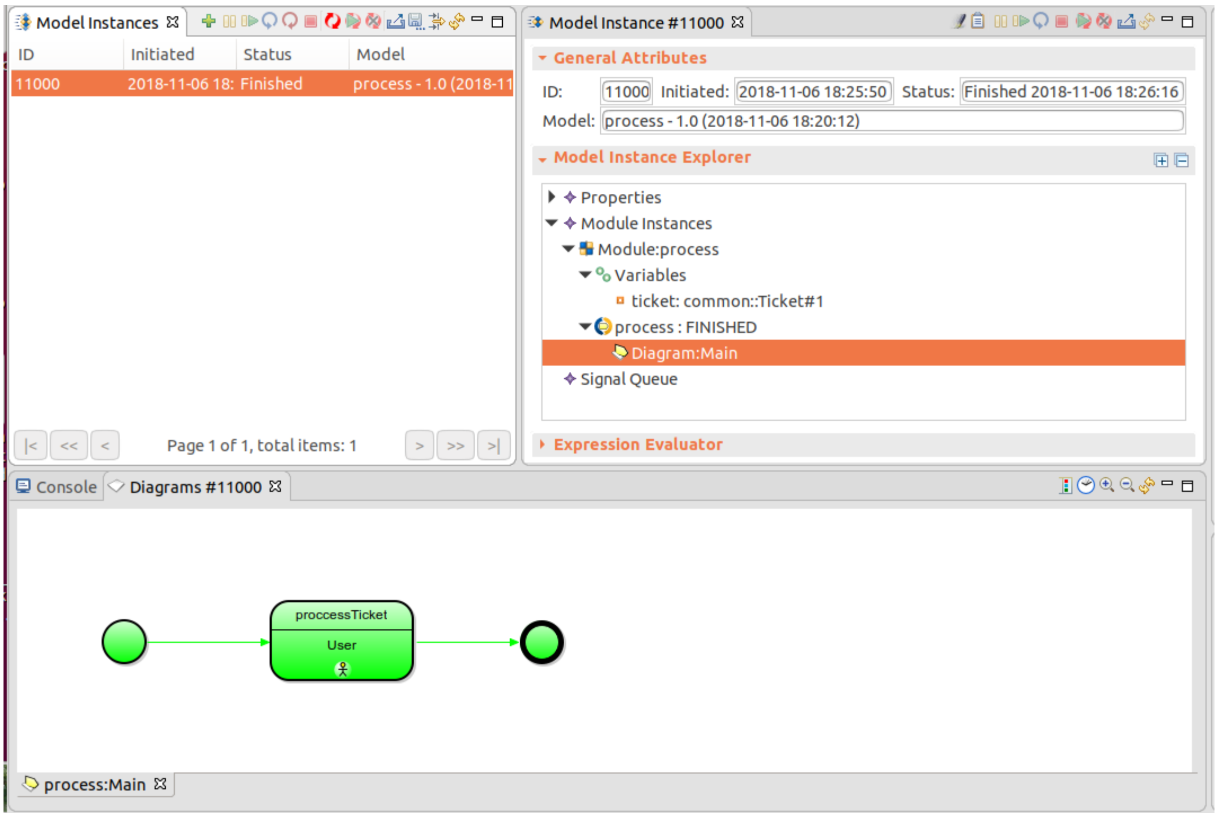
6. Upload the *process* module and test it (note that raising and common are uploaded as well):

   (a) Assign the *SupportSpecialist* role to a person. Make sure to *save* the changes.

   (b) Log in to the Application User Interface as the person with the *Customer* role; create a ticket from the document.

(c) Log in to the Application User Interface as the person with the *SupportSpecialist* role and resolve the ticket from the to-do.

(d) In the Management perspective, check the status on the server: a model instance with the ticket as its property was created and executed.



ID	Initiated	Status	Model
11000	2018-11-06 18:	Finished	process - 1.0 (2018-11

Page 1 of 1, total items: 1

General Attributes

ID: 11000 Initiated: 2018-11-06 18:25:50 Status: Finished 2018-11-06 18:26:16

Model: process - 1.0 (2018-11-06 18:20:12)

Model Instance Explorer

- Properties
- Module Instances
  - Module:process
    - Variables
      - ticket: common::Ticket#1
    - process : FINISHED
      - Diagram:Main
- Signal Queue

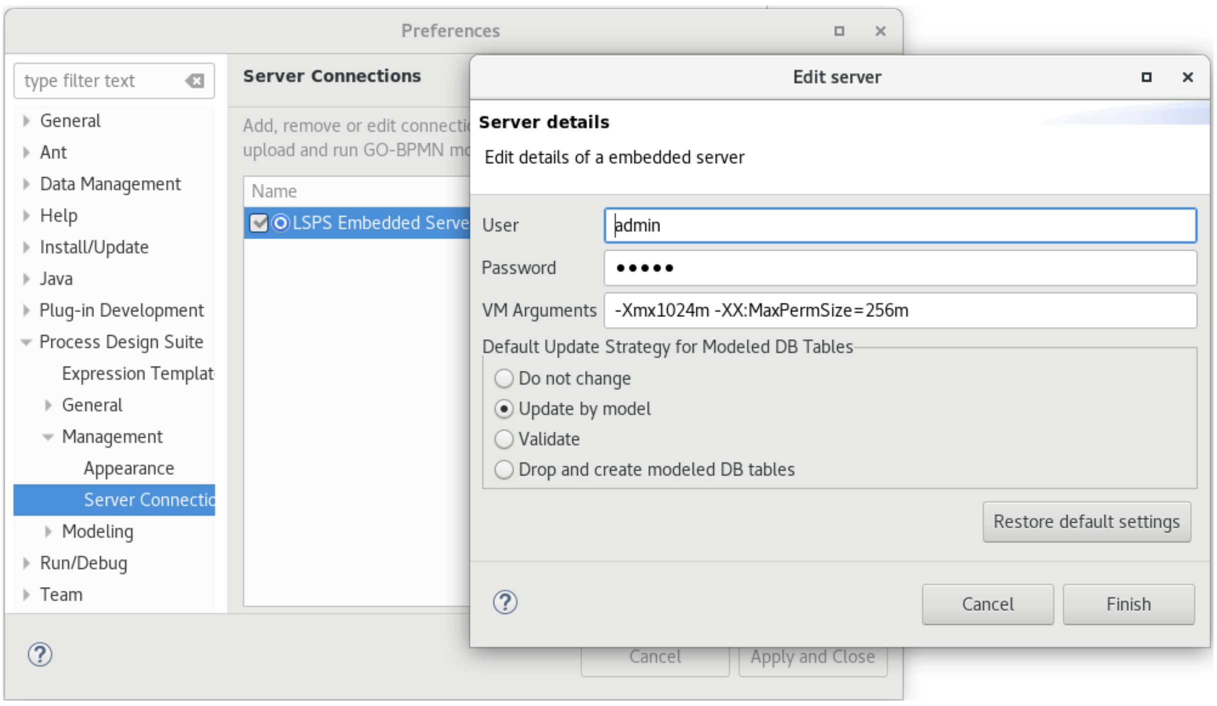
Expression Evaluator

Figure 4.5 Details of the model instance created by the document

7. Optionally, open your database client and check that the ticket table contains the new ticket.

4.1.3.1 Tips and Takeaways

- The details of a model instance might not be always available since it is governed by the `Create process log` flag of modules. Also the option can be `disabled globally on the server` to improve performance.
- Always try to minimize the amount of global variables.
- To delete the database of the PDS Embedded Server, go to **Server Connections** > **LSPS Embedded Server** > **Database Management**: in the displayed dialog box, click **Reset**. If the option is disabled, stop the server by clicking **Stop** first.
- To delete and recreate the data model whenever you upload a module, set the `database strategy` for your connection under the *Server Connections* menu. This sets the default strategy: Mind that it might be overriden in the upload configuration of models.



Server details

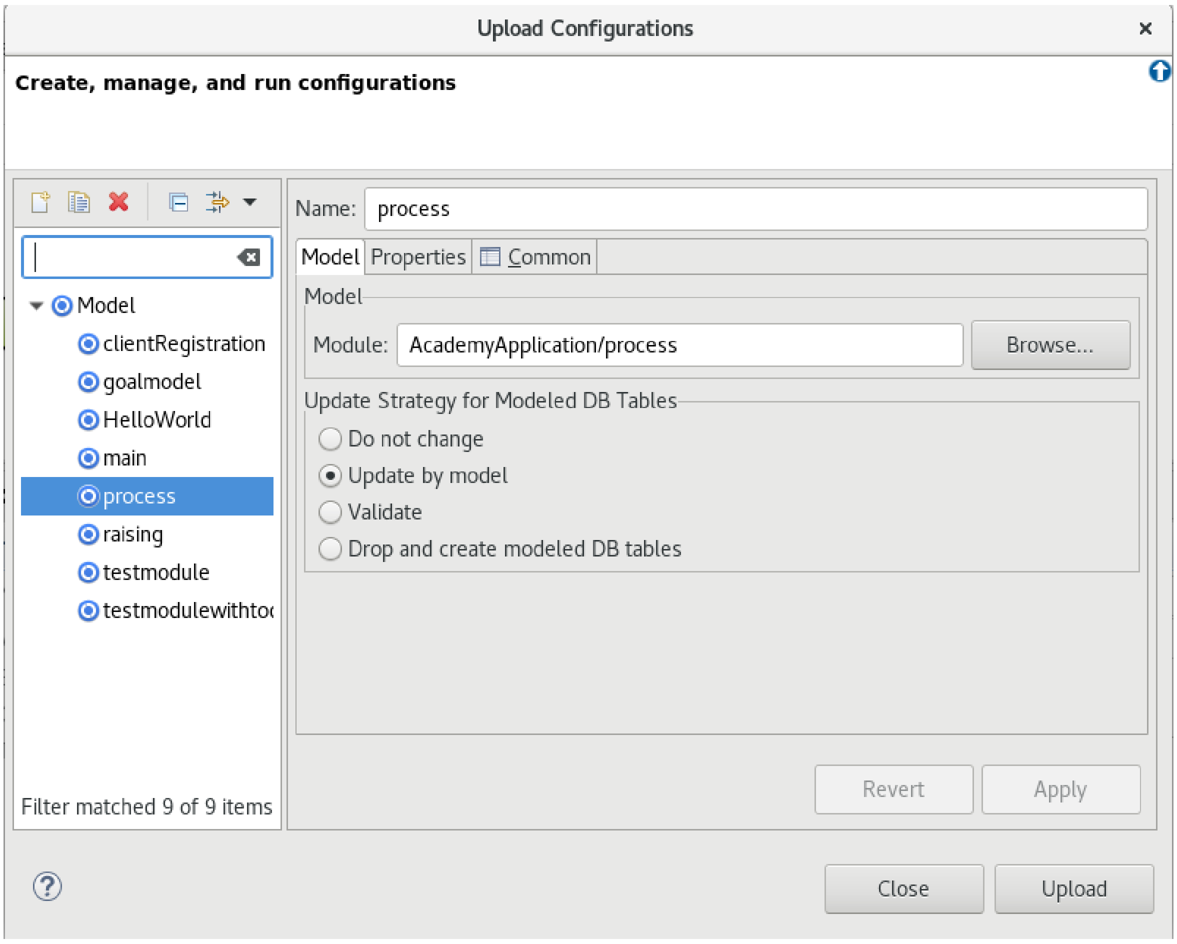
Edit details of a embedded server

User	admin
Password	●●●●●
VM Arguments	-Xmx1024m -XX:MaxPermSize=256m

Default Update Strategy for Modeled DB Tables

- ○ Do not change
- ● Update by model
- ○ Validate
- ○ Drop and create modeled DB tables

Figure 4.6 Default upload strategy setting for the embedded server


Upload Configurations

Create, manage, and run configurations

Name: process

Model | Properties | Common

Model

Module: AcademyApplication/process

Update Strategy for Modeled DB Tables

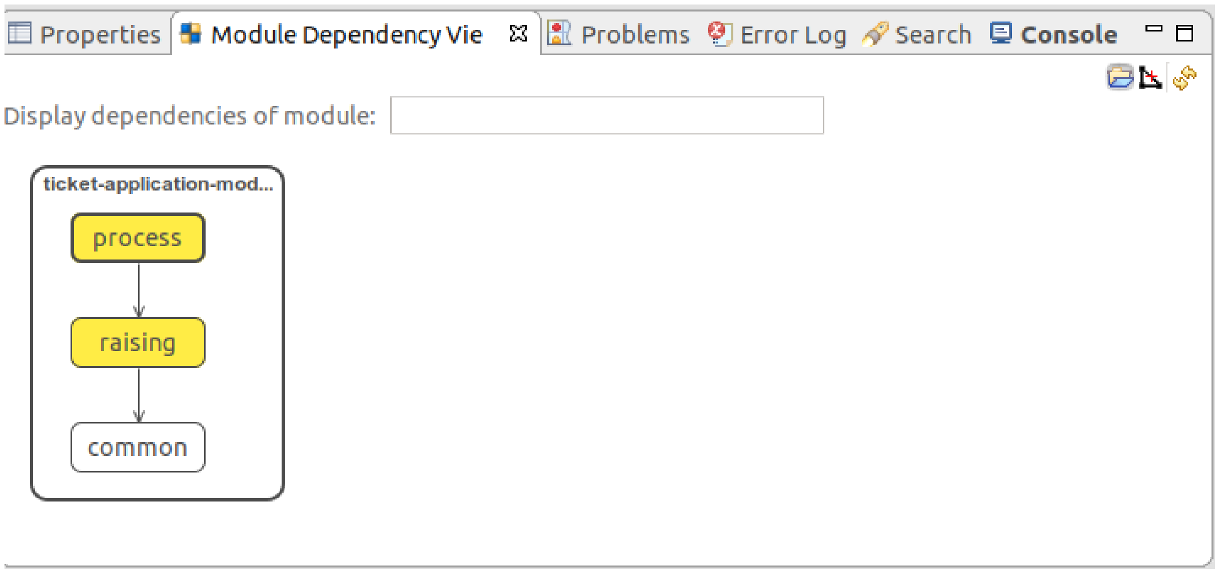
( ) Do not change
[✓ Update by model]
( ) Validate
( ) Drop and create modeled DB tables

Model: clientRegistration, goalmodel, HelloWorld, main, process, raising, testmodule, testmodulewithto

Filter matched 9 of 9 items

Figure 4.7 Upload stategy set on the upload configuration of the process model

- Check your module-imports schema in the **Module Dependency View**: To display the view, go to **Window** > **Show View** > **Module Dependency View**.



- To display the complete expression of your form, in the form editor, right-click anywhere in the form and select **Display Widget Expression**.

4.1.4 Commenting a Ticket

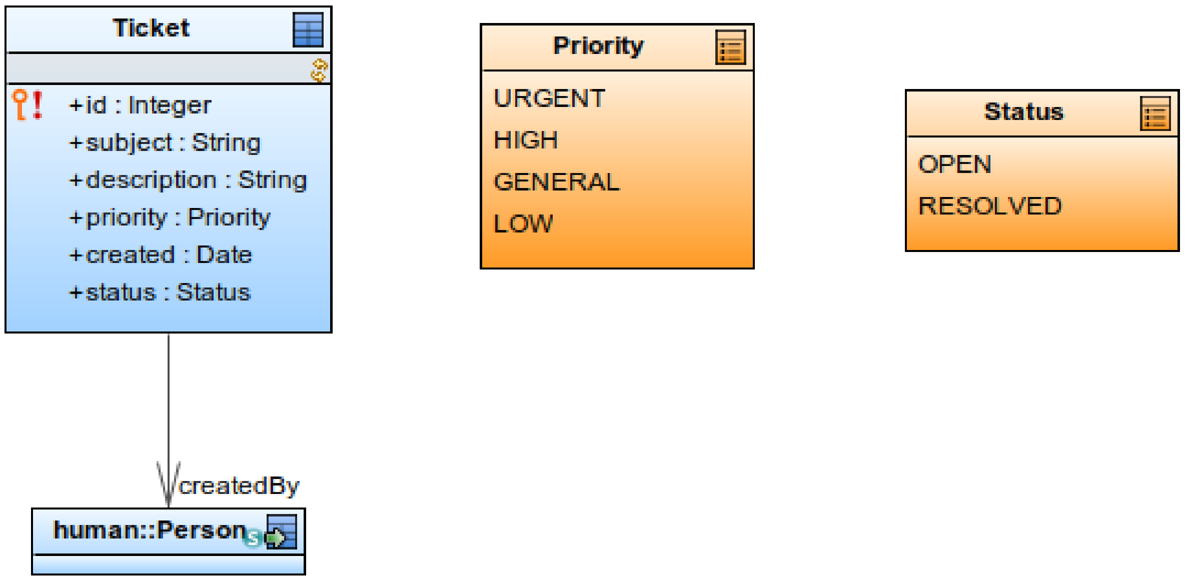
User story: The customer and support specialist add comments to the ticket.

This means you need to start tracking the customer who raised the ticket so you know whom to ask for further comments as well as the comments and their authors. For the time being, a ticket does not have to be handled by the same support specialist.

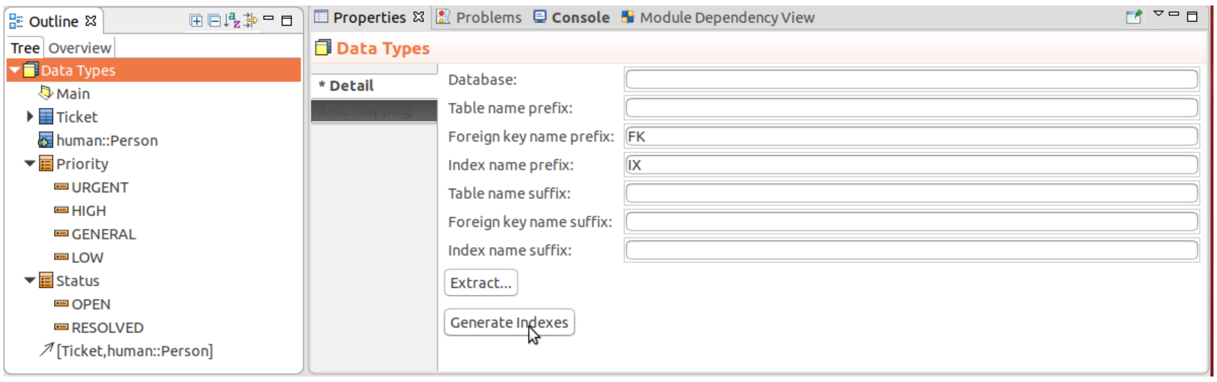
1. Extend the data model in the *common* module so that Ticket holds information on who created it:

   (a) Insert Record Import of *human:Person*.
   Person represents all users of the LSPS Application.

   (b) Create a data relationship between Ticket and Person.



   (c) Generate the index for Ticket to Person ID: right-click the data model root in the Outline view and, in the Properties view, click **Generate Indexes**.



Database:
Table name prefix:
Foreign key name prefix: FK
Index name prefix: IX
Table name suffix:
Foreign key name suffix:
Index name suffix:

2. In *raising::TicketRaiseForm*, edit *Click Listener* of the *Create Ticket* button: set the *createdBy* record to the user who is requesting the action from the document, that is, the current user:

```
{ e ->
   //saving new ticket in the newTicket expression variable:
   def Ticket newTicket := new Ticket(
    created -> now(),
    subject -> subject.getValue(),
    description -> description.getValue(),
    priority -> priority.getValue() as Priority,
    status -> Status.OPEN,
    //added this:
    createdBy -> getCurrentPerson()
  );
   ...
```

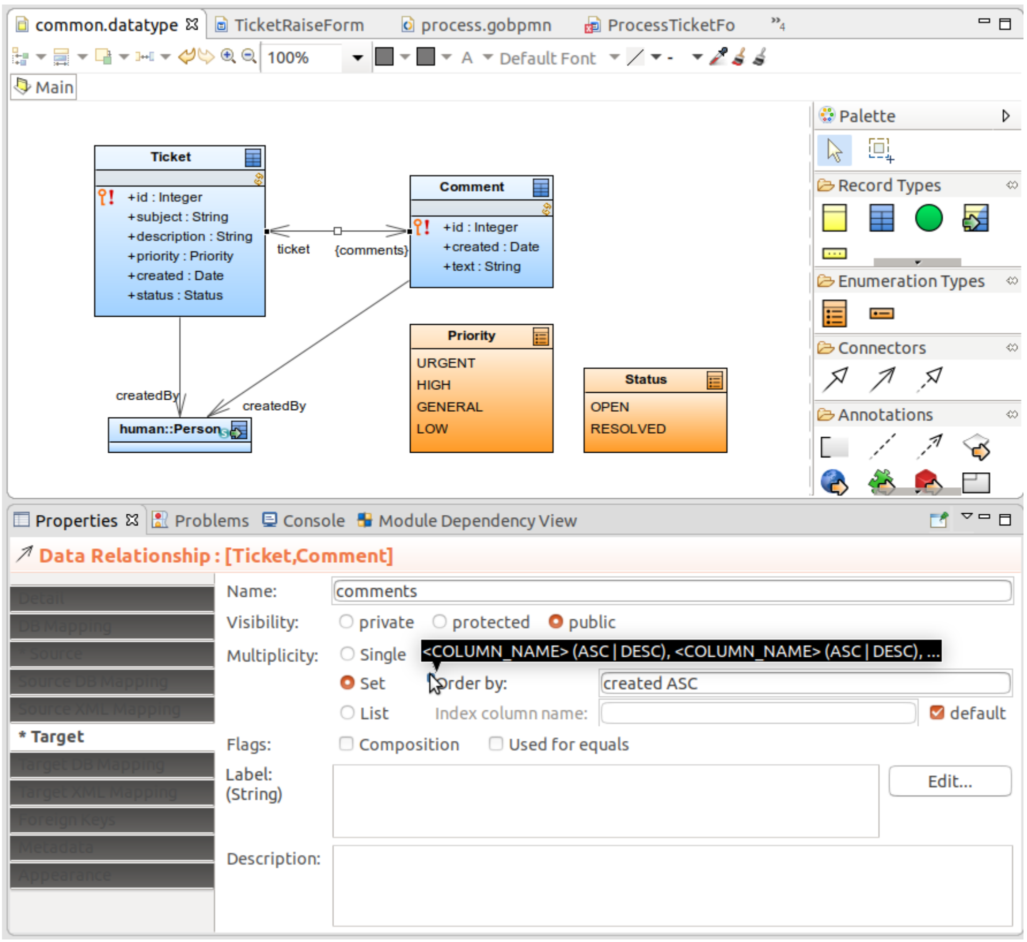
3. Extend the data model in the *common* module further so that Ticket holds its comments and comments hold data about who created them:

   (a) Create a *Comment* shared record.

   (b) Create a relationship between *Comment* and *Person*: it is sufficient to name the end directed toward Person, since you don't need to know what comments a person made.

   (c) Create a relationship between Ticket and Comment: name both ends so you can request comment from a ticket and request the ticket of a particular comment. The end directed toward *Comment* has the *Set* multiplicity, since a ticket can have multiple comments.

   (d) Generate indexes for the relationship).



Ticket
+id : Integer
+subject : String
+description : String
+priority : Priority
+created : Date
+status : Status

ticket {comments}

Comment
+id : Integer
+created : Date
+text : String

createdBy
createdBy
human::Person

Priority
URGENT
HIGH
GENERAL
LOW

Status
OPEN
RESOLVED

Data Relationship : [Ticket,Comment]

Name: comments
Visibility: private protected [✓ public]
Multiplicity: Single [✓ Set] List
<COLUMN_NAME> (ASC | DESC), <COLUMN_NAME> (ASC | DESC), ...
Order by: created ASC
Index column name: [✓ default]
Flags: Composition Used for equals
Label: (String)
Description:

Figure 4.8 Relationship end of Comment with ordering defined

4. Allow commenting in *ProcessTicketForm*:

   (a) Add a Text Area for a new comment with Value binding and ID `comment`.

   (b) Add a **Submit** button with the click action `{ e -> Forms.submit(); }` so the user can submit the to-do without resolving the ticket.

   (c) Add an *Add Comment* button with listener expression:

```
        {
           e -> new Comment(
                  created -> now(),
                  text ->  comment.getValue(),
                  ticket -> ticket,
                  createdBy -> getCurrentPerson()
                  );
               //clears the comment text area:
               comment.setValue("");
        }
```

5. Display all comments in *ProcessTicketForm*:

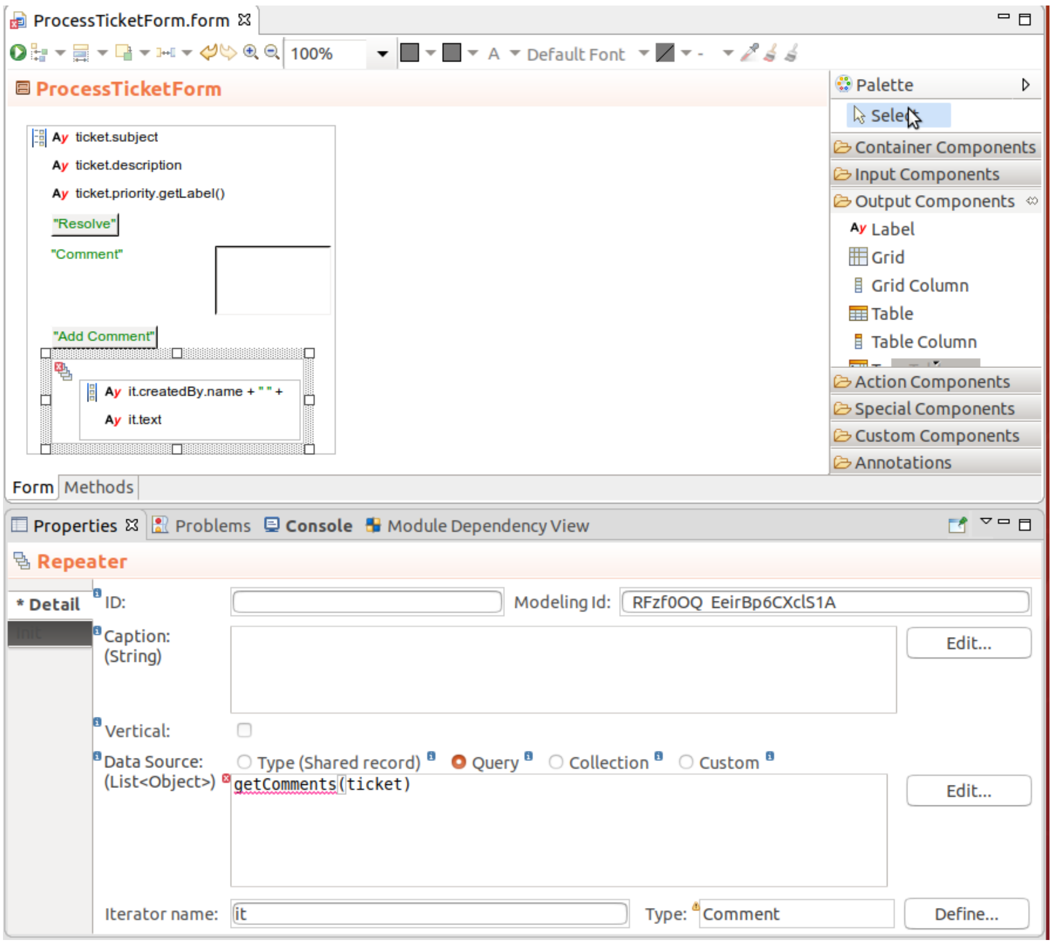
   (a) Add the Repeater component with ID `previousComments` which will display existing comments:

      i. Select the **Vertical** option so the instances of the child component are displayed above each other.

      ii. Define the data source as Query with the expression `getComments(ticket).`

      iii. Define the iterator type as `Comment.`

      iv. Insert the components that will display comment details into the repeater.



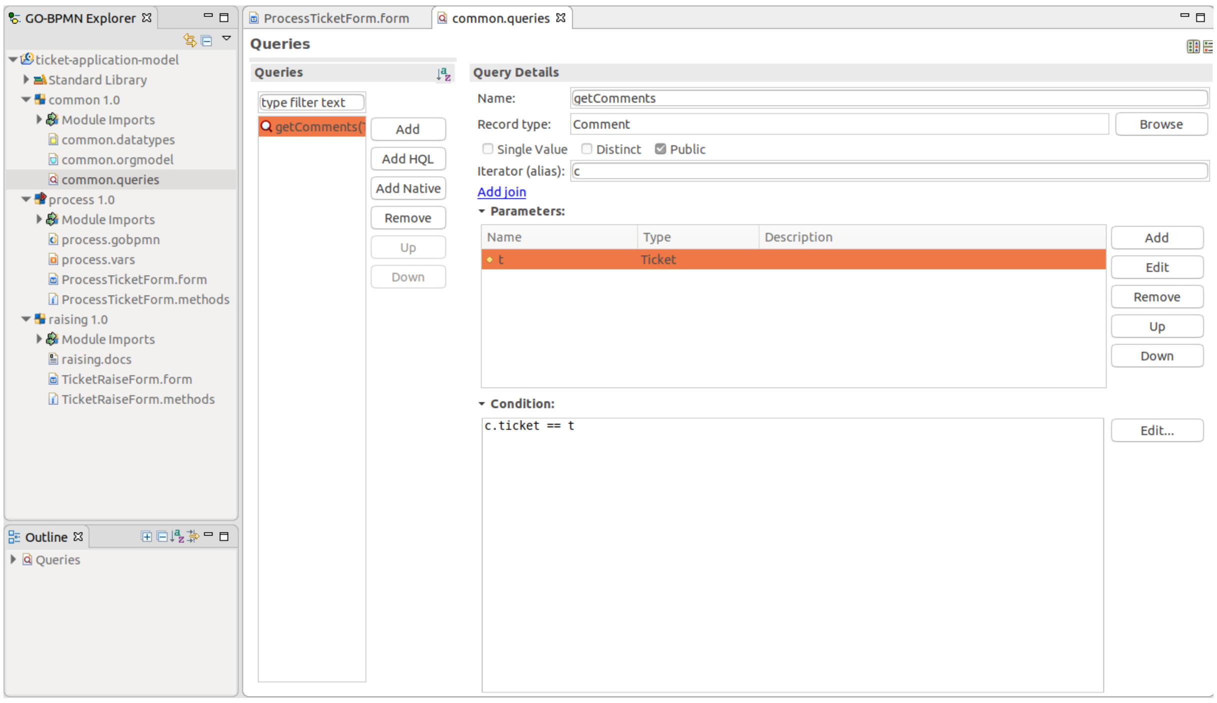
ID:
Modeling Id: RFzf0OQ_EeirBp6CXclS1A
Caption: (String)
Vertical: ☐
Data Source: (List<Object>) ○ Type (Shared record) ● Query ○ Collection ○ Custom

```
getComments(ticket)
```

Iterator name: it
Type: Comment

Figure 4.9 Repeater with the vertical layout as its immediate child

   (b) In the *common* module, create the missing *getComments(<Ticket>)* query, which returns the comments of a ticket.



Name: getComments
Record type: Comment
☐ Single Value ☐ Distinct ☑ Public
Iterator (alias): c

Parameters:

Name	Type	Description
t	Ticket	

Condition:

```
c.ticket == t
```

(c) Refresh the list of comments when a new comment is created from the click listener of the *Add Comment* button:

```
{
   e -> new Comment(
          created -> now(),
          text ->  comment.getValue(),
          ticket -> ticket,
          createdBy -> getCurrentPerson()
          );
       comment.setValue("");
       //ADDED: refreshes the list of previous comments
       //so they show the new comment:
       previousComments.refresh()
}
```

6. Extend the process that handles the ticket to allow the customer to react to the current comment from support:

   (a) Add an Exclusive Gateway between the ProcessTicket task and the Simple End Event.

   (b) On the Flow between the gateway and the end event, define the guard expression `ticket.↩ status==Status.RESOLVED` (the flow will be taken only when the ticket is resolved).

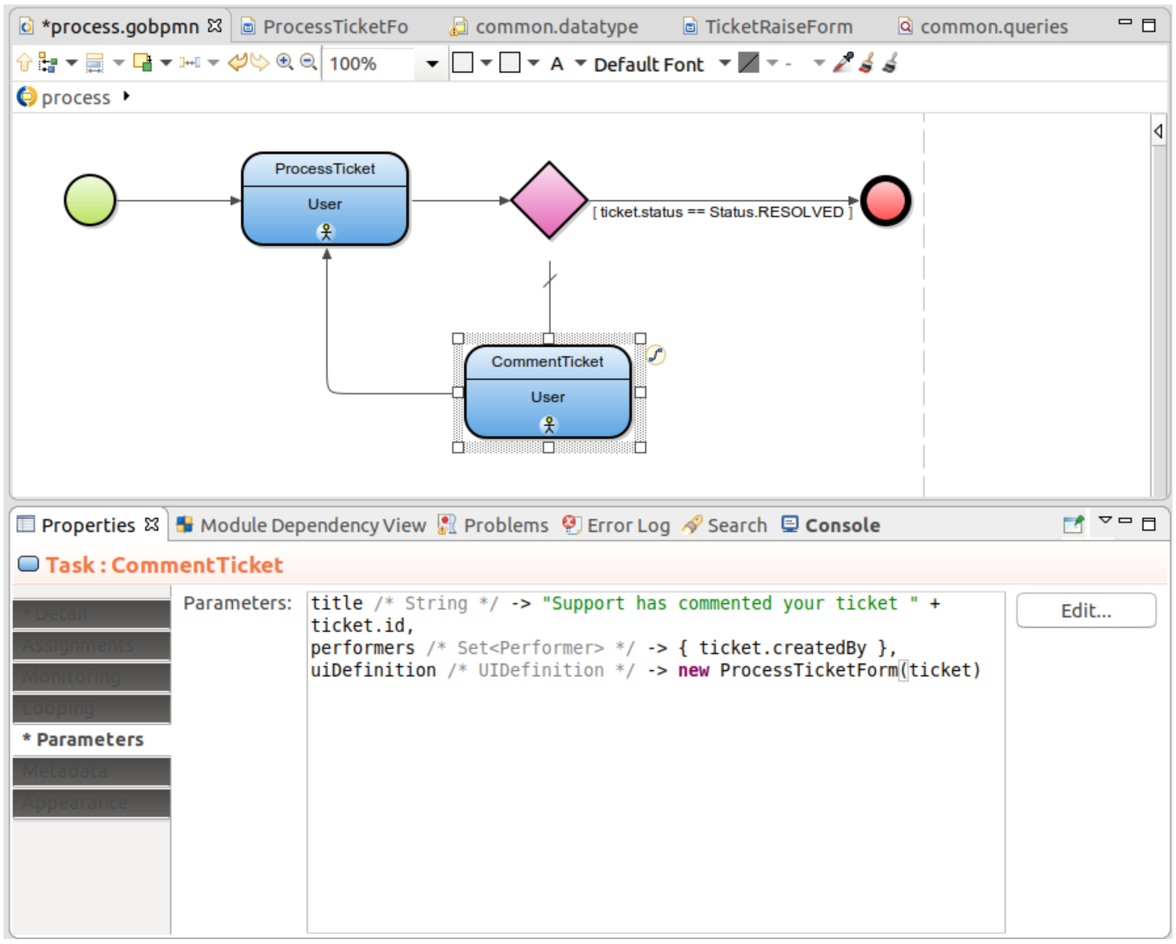
   (c) Create a User task and name it `CommentTicket`.

   (d) Set the parameters of the *CommentTicket* task to:

```
 title -> "Support has commented your ticket " + ticket.id,
performers -> { ticket.createdBy },
uiDefinition -> new ProcessTicketForm(ticket)
```

   (e) Connect the gateway to the *CommentTicket* task and set the Flow as default.

   (f) Connect the *CommentTicket* task to the *ProcessTicket* task.



Parameters:
```
title /* String */ -> "Support has commented your ticket " +
ticket.id,
performers /* Set<Performer> */ -> { ticket.createdBy },
uiDefinition /* UIDefinition */ -> new ProcessTicketForm(ticket)
```

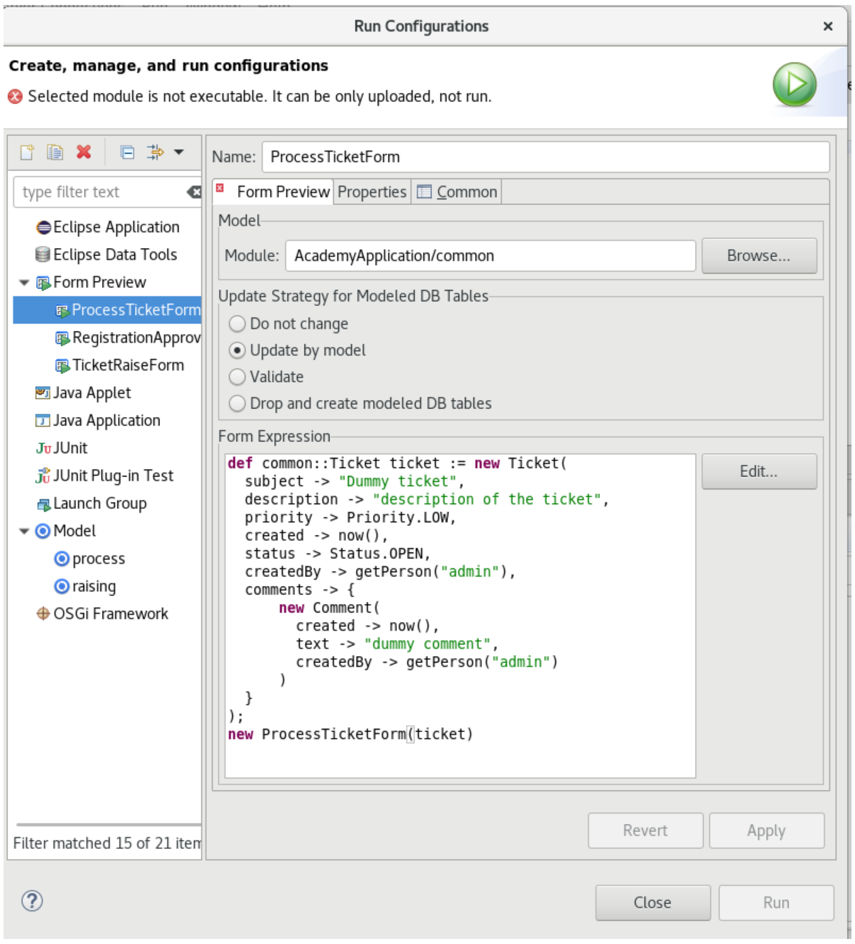
7. Adapt *ProcessTicketForm* for the *Comment Ticket* task: Simply hide or disable the **Resolve** button if the user does not have the SupportSpecialist role: on the Init tab of the button add the following:

```
if not isPersonIn(getCurrentPerson(), SupportSpecialist())
then
  c.setVisible(false)
end
```

8. Upload the *raising* and *process* modules and test the scenario.

4.1.4.1 Tips and Takeaways

- To preview parametric forms, such as, the *ProcessTicketForm*, set the parameter in its run configuration: First run the preview of the form. The preview will fail but the preview configuration is created: Now right-click the form and go to **Run As** > **Run Configurations** and edit the *Form Expression*.



Run Configurations

Create, manage, and run configurations

Selected module is not executable. It can be only uploaded, not run.

Name: ProcessTicketForm

Form Preview | Properties | Common

Model

Module: AcademyApplication/common

Update Strategy for Modeled DB Tables

( ) Do not change
(●) Update by model
( ) Validate
( ) Drop and create modeled DB tables

Form Expression

```
def common::Ticket ticket := new Ticket(
  subject -> "Dummy ticket",
  description -> "description of the ticket",
  priority -> Priority.LOW,
  created -> now(),
  status -> Status.OPEN,
  createdBy -> getPerson("admin"),
  comments -> {
      new Comment(
        created -> now(),
        text -> "dummy comment",
        createdBy -> getPerson("admin")
      )
  }
);
new ProcessTicketForm(ticket)
```

- Improve presentation of *ProcessTicketForm*:
  - Use Panels to divide comments and ticket details section;

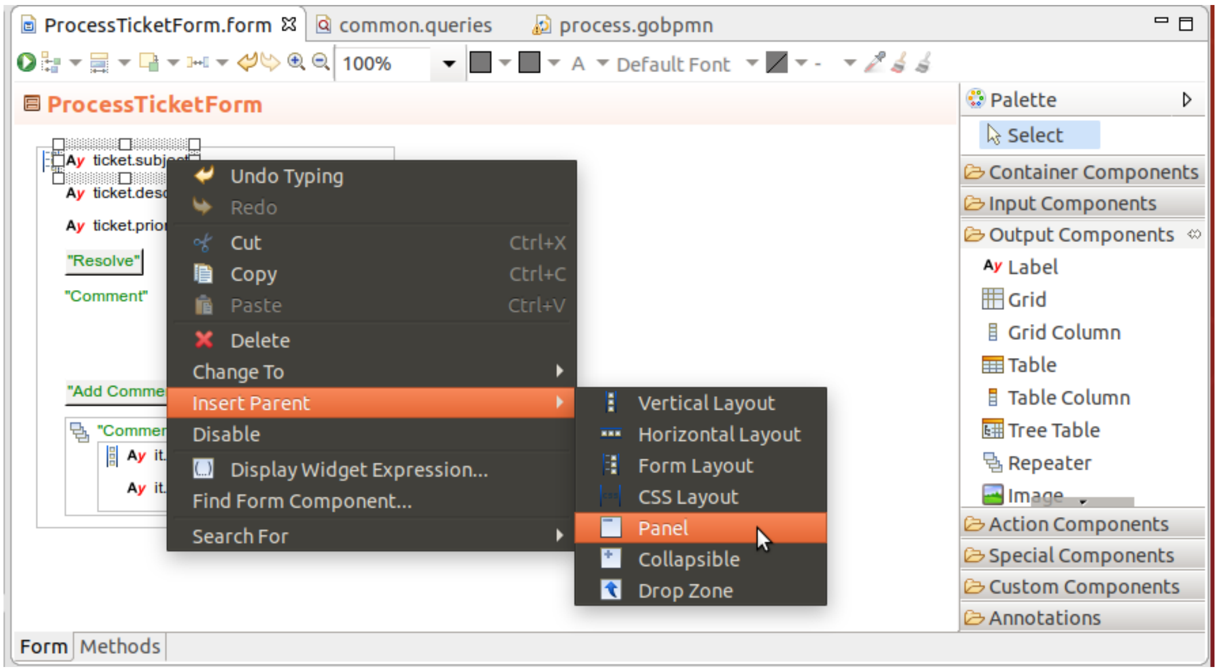
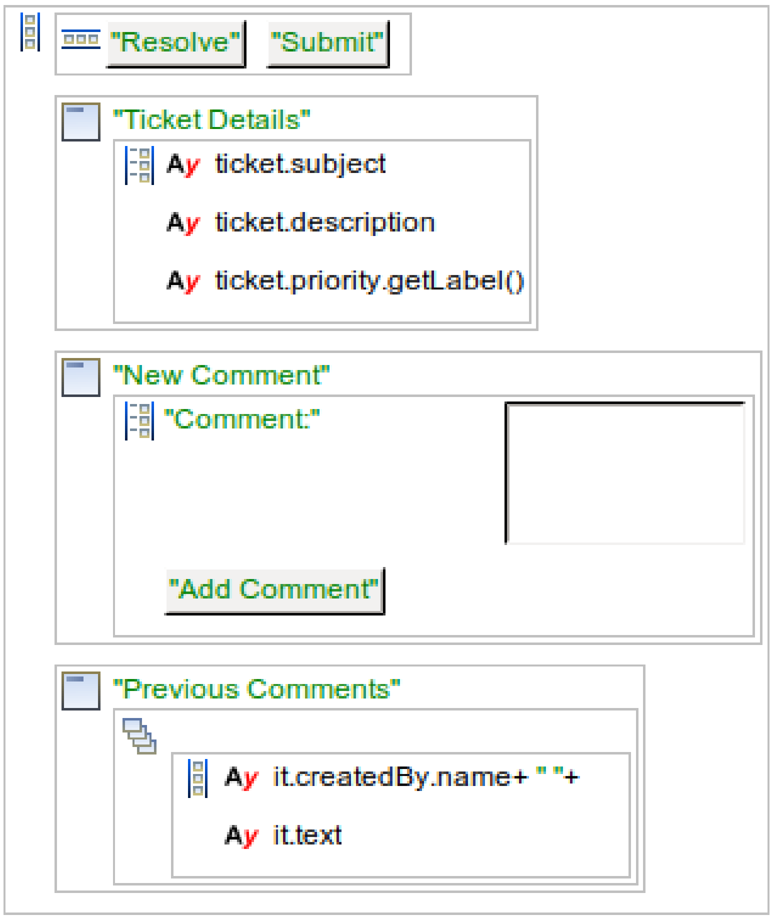


Figure 4.10 Wrapping a component in a container component

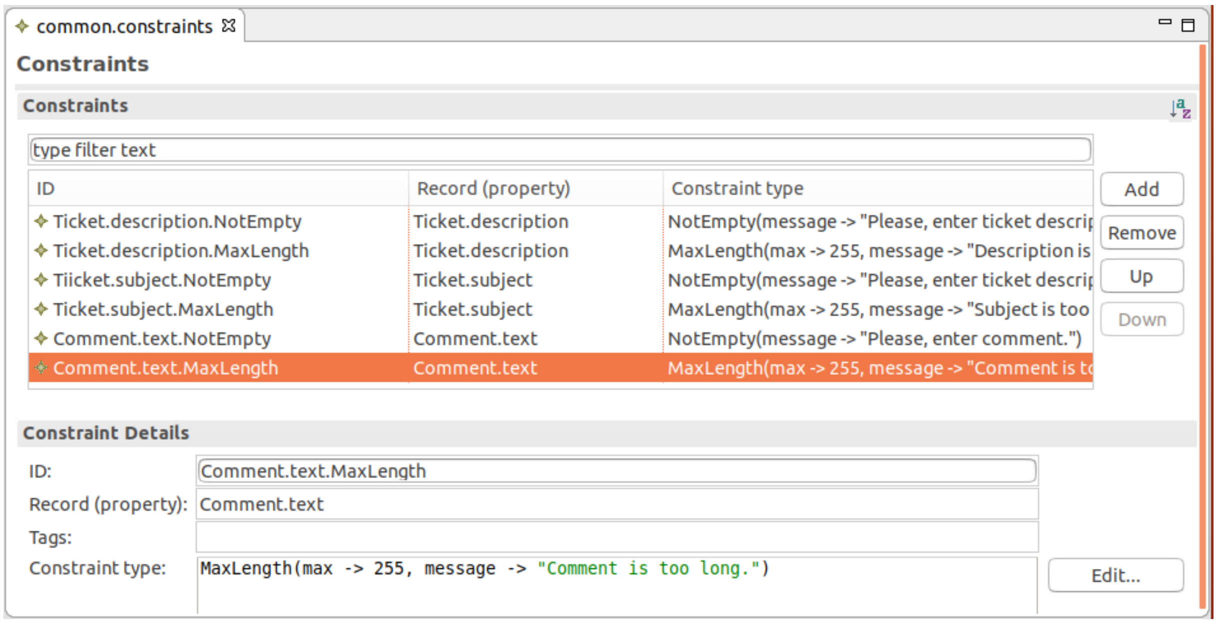
- Select the **Spacing** option on container components.
- Add captions.
- Move buttons to the components related to their actions.



4.1.5 Validating Ticket Details

User story: The customer must always define a subject and description of their ticket.

1. In *common*, define the restrictions for *Ticket* record fields in constraints.



Constraints

Constraints

type filter text

ID	Record (property)	Constraint type
Ticket.description.NotEmpty	Ticket.description	NotEmpty(message -> "Please, enter ticket descri
Ticket.description.MaxLength	Ticket.description	MaxLength(max -> 255, message -> "Description is
Tiicket.subject.NotEmpty	Ticket.subject	NotEmpty(message -> "Please, enter ticket descri
Ticket.subject.MaxLength	Ticket.subject	MaxLength(max -> 255, message -> "Subject is too
Comment.text.NotEmpty	Comment.text	NotEmpty(message -> "Please, enter comment.")
Comment.text.MaxLength	Comment.text	MaxLength(max -> 255, message -> "Comment is t

Constraint Details

ID:	Comment.text.MaxLength
Record (property):	Comment.text
Tags:	
Constraint type:	MaxLength(max -> 255, message -> "Comment is too long.")

The length restrictions prevent storing of values which would breach the length allowed by the data model.

2. Validate the data in the *TicketRaiseForm* form against the constraints:

   (a) Set the id of the *Create Ticket* button to `createButton`.

   (b) Since you need to validate the Ticket record before it is created, create the new ticket as a proxy record in the *Create Ticket* listener expression:

   ```
   { e ->
       def RecordProxySet recordProxySet := createProxySet(null);
       def Ticket newTicket := recordProxySet.proxy(Ticket);
   ~
       newTicket.created := now();
       newTicket.subject := subject.getValue();
       newTicket.description := description.getValue();
       newTicket.priority := priority.getValue()?? Priority.GENERAL as Priority;
       newTicket.status := Status.OPEN;
       newTicket.createdBy := getCurrentPerson();
   ~
       createModelInstance(
   ...
   ```

   (c) Still in the *Create Ticket* listener expression, after the *newTicket* proxy is populated with data, collect the messages from the constraints that are violated by the proxy.

   ```
   def List<ConstraintViolation> constraintViolations := validate(newTicket, null, null, null);
   ```

   (d) If the proxy does not violate any constraints, merge its proxy set with `recordProxySet.↩ merge(false);` and create the *process* model instance.

   (e) If the proxy violates a constraint, display the violations: call `showDataErrorMessages(constraint↩ Violations, createButton)`

Final click listener expression on the *Create Ticket* button

```
{ e ->
   def RecordProxySet recordProxySet := createProxySet(null);
   def Ticket newTicket := recordProxySet.proxy(Ticket);
~
   newTicket.created := now();
   newTicket.subject := subject.getValue();
   newTicket.description := description.getValue();
```

```
   newTicket.priority := priority.getValue()?? Priority.GENERAL as Priority;
   newTicket.status := Status.OPEN;
   newTicket.createdBy := getCurrentPerson();
~
  def List<ConstraintViolation> constraintViolations := validate(newTicket, null, null, null);
~
  if constraintViolations.isEmpty() then
    recordProxySet.merge(false);
    createModelInstance(
      synchronous -> false,
      model -> getModel("process"),
      processEntity -> newTicket,
      properties -> null);
    Forms.submit();
  else
    //display violation messages on create button:
    showDataErrorMessages(constraintViolations, createButton)
  end
}
```

4.1.5.1 Tips and Takeaways

- Extract the click listener expression from the *Create Ticket* button to the `TicketRaiseForm.submit↩ Ticket()` method for better maintenance and call the method from the click listener expression.

  ```
  //new click listener expression:
  { e ->
     submitTicket()
  }
  ```

- Make the ticket data in *TicketRaiseForm* easier to manage:

  1. Create a form variable for the ticket and for a record proxy set.
  2. Initialize them in the form constructor:

     ```
     public TicketRaiseForm(){
        recordProxySet := createProxySet(null);
        newTicket := recordProxySet.proxy(Ticket);
     }
     ```

  3. Remove their expression variables from the submitTicket() method.

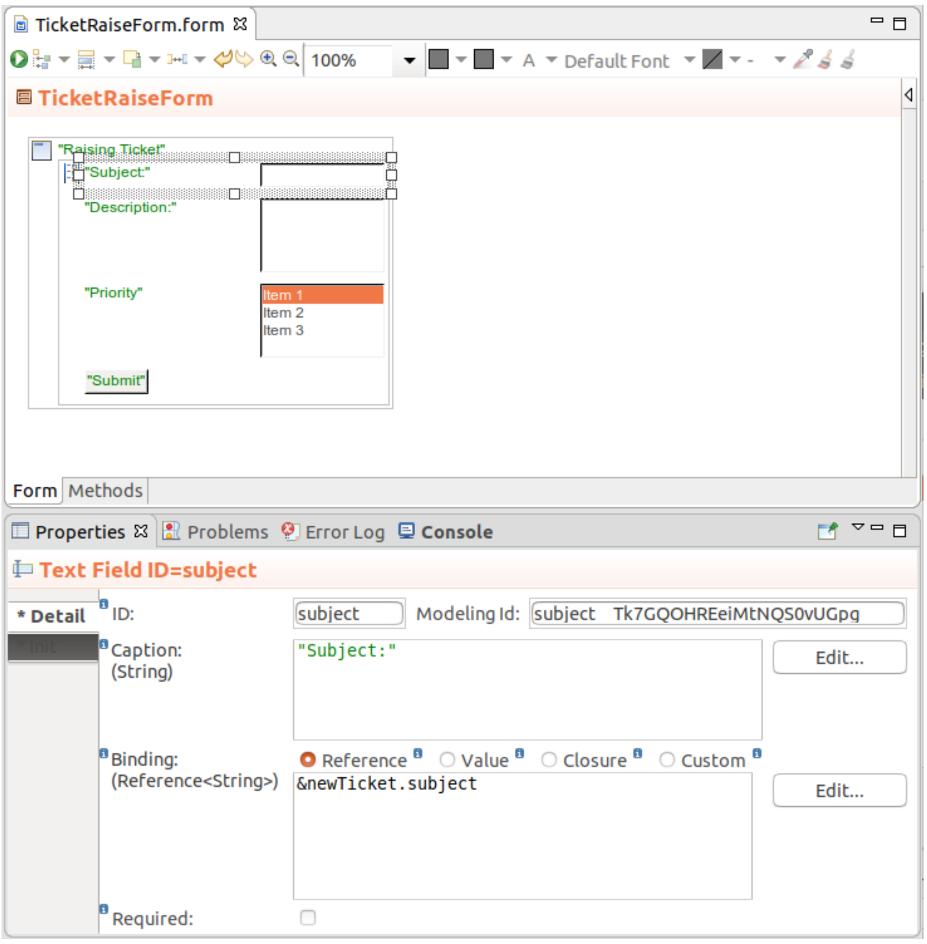
  Resulting *TicketRaiseForm* constructor and `submitTicket()` method

  ```
  TicketRaiseForm {
  ~
     public TicketRaiseForm(){
       recordProxySet := createProxySet(null);
       newTicket := recordProxySet.proxy(Ticket);
     }
     public void submitTicket(){
  ~
       newTicket.created := now();
       newTicket.subject := subject.getValue();
       newTicket.description := description.getValue();
       newTicket.priority := priority.getValue()?? Priority.GENERAL as Priority;
       newTicket.status := Status.OPEN;
       newTicket.createdBy := getCurrentPerson();
  ~
       def List<ConstraintViolation> constraintViolations := validate(newTicket, null, null, nu
  ~
      if constraintViolations.isEmpty() then
  ```

```
          recordProxySet.merge(false);
          createModelInstance(
            synchronous -> false,
            model -> getModel("process"),
            processEntity -> newTicket,
            properties -> null);
          Forms.submit();
        else
          //display violation messages on submitTicket:
          showDataErrorMessages(constraintViolations, submitButton)
        end
      }
  }
```

- Display the violations on the components that are bound to the record fields with the violation:

  1. Change the binding of the input components to *Reference* and define its Value as Reference to the respective field of the *newTicket* variable, for example, `&newTicket.subject` for the subject input field.



Text Field ID=subject

Detail

ID: subject   Modeling Id: subject_Tk7GQOHREeiMtNQS0vUGpg

Caption: (String) "Subject:"

Binding: (Reference<String>) [✓ Reference] Value Closure Custom
&newTicket.subject

Required: ☐

  2. In the *submitTicket()* method, change the component that displays the constraint messages to the form root: Define the id of the root layout as `formLayout` and adjust the showDataErrorMessages call. The messages will be displayed on the components bound to the respective record field.

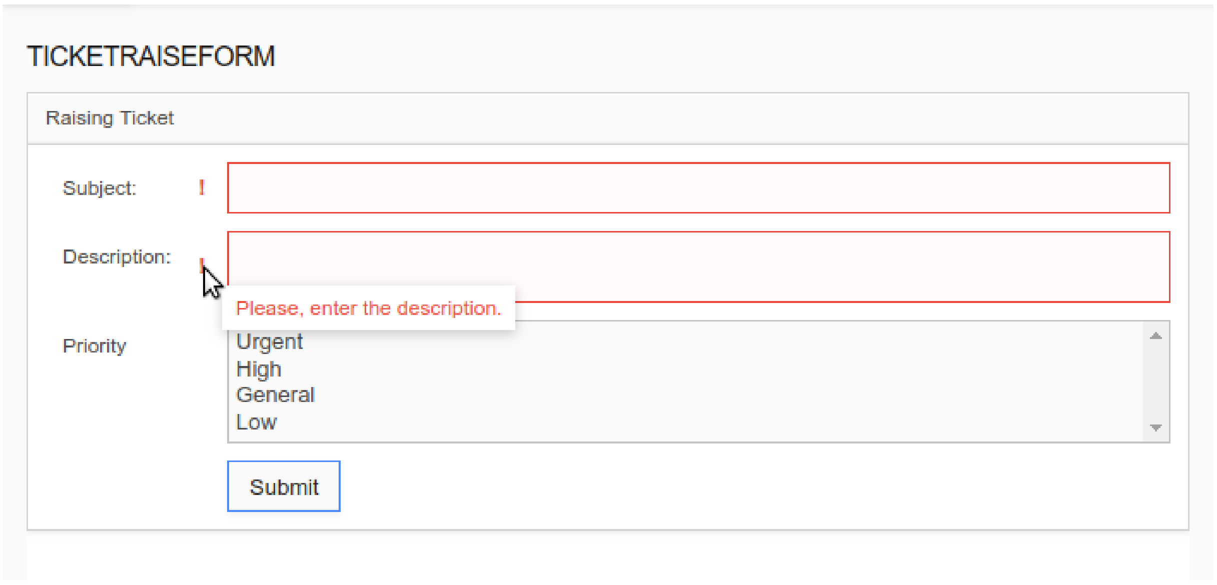
```
...
        createModelInstance(
          synchronous -> false,
          model -> getModel("process"),
          processEntity -> newTicket,
          properties -> null);
        Forms.submit();
      else
```

```
              //display violation messages on the form layout or
              //children of the form root for record bindings:
              showDataErrorMessages(constraintViolations, formLayout)
            end
          }
      }
```

- Analogously, adjust validation in *TicketDetailsForm*.

```
    //Click listener on the Add Comment button
    //in TicketDetailsForm:
    {
       e ->
          def RecordProxySet recordProxySet := createProxySet(null);
          def Comment newComment := recordProxySet.proxy(Comment);
    ~
          newComment.created := now();
          newComment.text := comment.getValue();
          newComment.ticket := ticket;
          newComment.createdBy := getCurrentPerson();
          def List<ConstraintViolation> constraintViolations := validate(newComment, null, null,
    ~
          if (constraintViolations.isEmpty()) then
            recordProxySet.merge(false);
            comment.setValue("");
            previousComments.refresh();
          else
            showDataErrorMessages(constraintViolations, comment)
          end
    }
```

Validation message on the component with constraint violation


TICKETRAISEFORM

Raising Ticket

Subject: !	
Description: !	
Priority	Urgent High General Low

Please, enter the description.

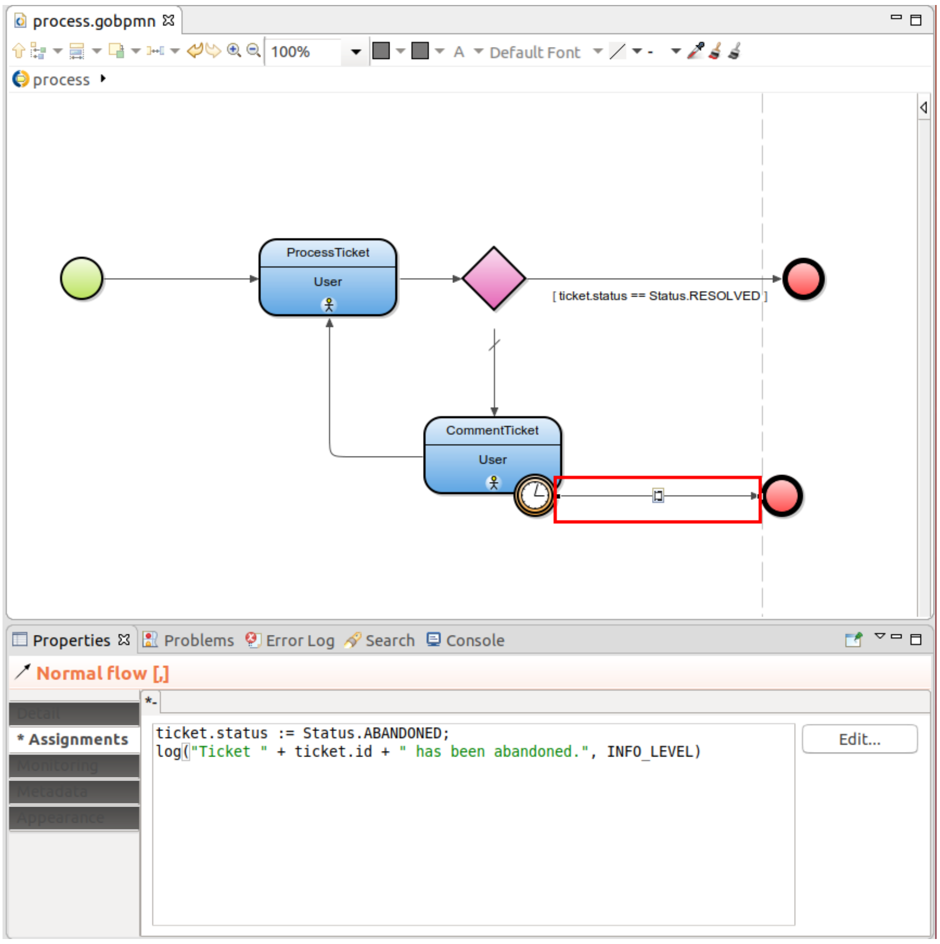
Submit

4.1.6 Killing an Idle Ticket

User story: If the customer does not respond to a comment of a support specialist within a given period of time, the ticket times out.

1. Add the ABANDONED status to the Status enumeration.
2. Adjust the process flow:
   (a) Add Timeout Intermediate Event on the boundary of CommentTicket and set its Date property to the point in time when the task should time out, for example, `now() + days(3)`.
   (b) Add an outgoing flow to a Simple End Event and change status of the ticket on the flow assignment:

```
ticket.status := Status.ABANDONED;
log("Ticket " + ticket.id + " has been abandoned.", INFO_LEVEL)
```



```
ticket.status := Status.ABANDONED;
log("Ticket " + ticket.id + " has been abandoned.", INFO_LEVEL)
```

3. Upload the *process* module and test the scenario when the ticket expires.

4.1.6.1 Tips and Takeaways

- Consider checking the semantics of `Timer Events`.

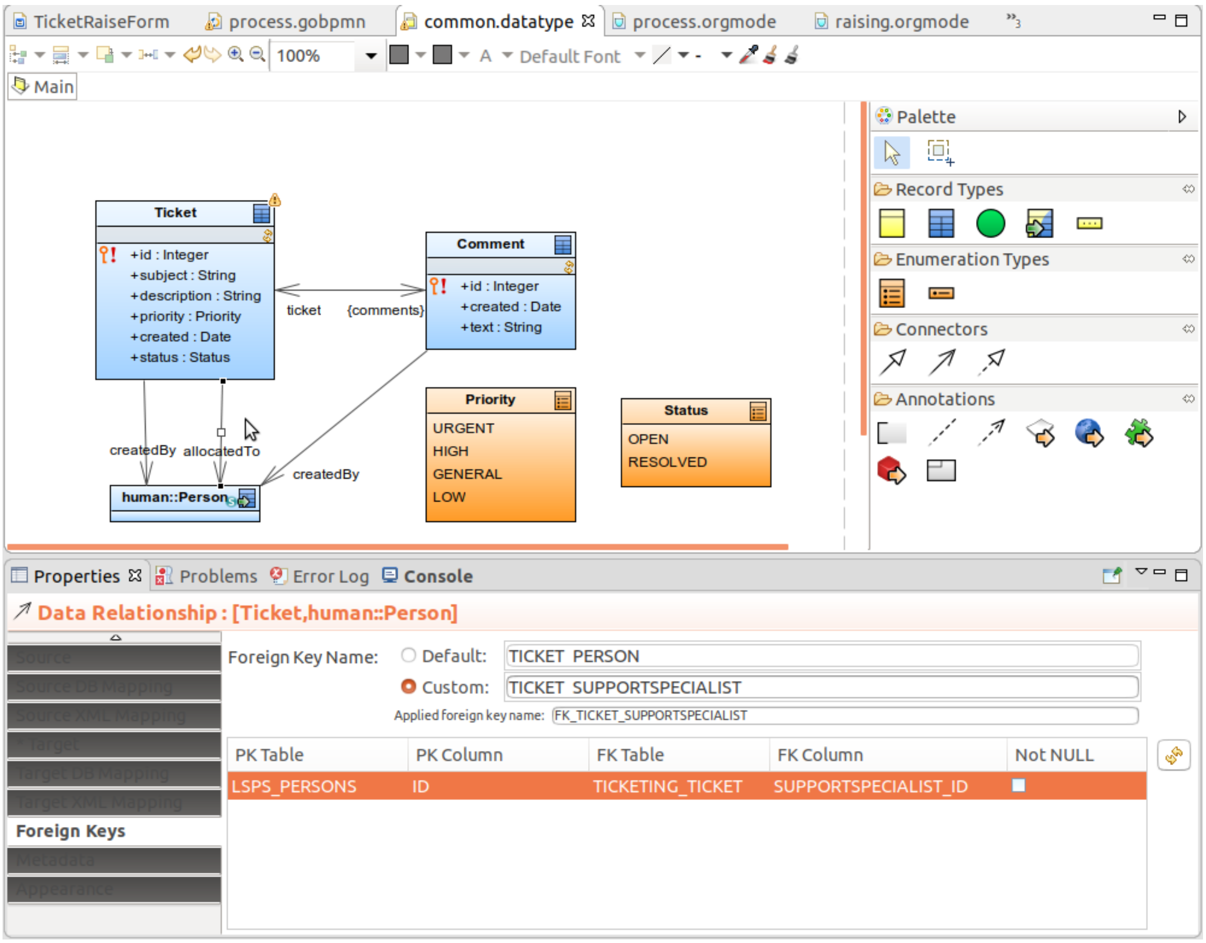
4.1.7 Assigning a Ticket to a Support Specialist

User story: Once a support specialist opens a ticket, it is handled exclusively by them for the rest of the ticket life (Currently, the *ProcessTicket* task can be handled by any user with the *SupportSpecialist* role).

1. In *common*, adapt the data type model:
   (a) Add another relationship of Ticket and Person.

(b) Change the foreign key names for the Ticket table.

(c) Generate indexes.



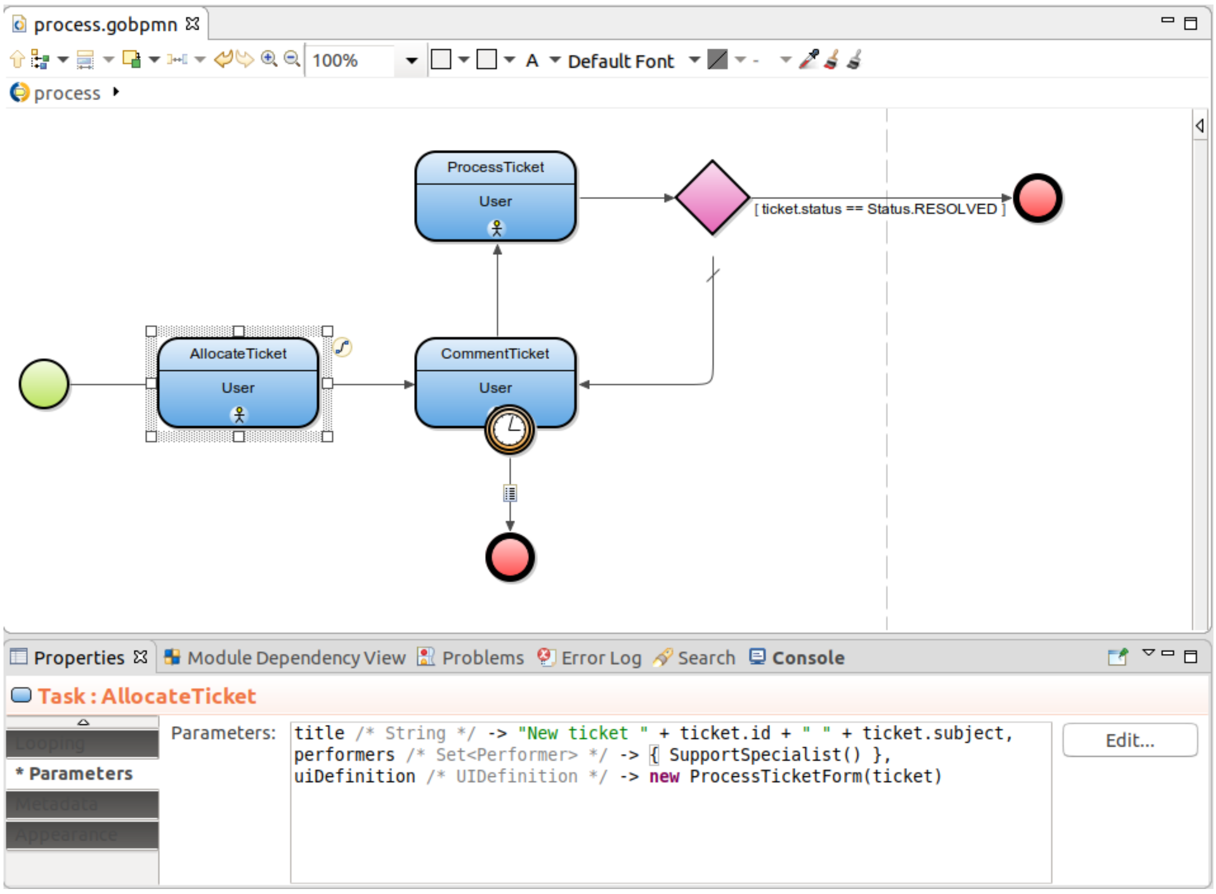
Data Relationship : [Ticket,human::Person]

Foreign Key Name: Default: TICKET_PERSON [✓ Custom: TICKET_SUPPORTSPECIALIST]

Applied foreign key name: FK_TICKET_SUPPORTSPECIALIST

PK Table	PK Column	FK Table	FK Column	Not NULL
LSPS_PERSONS	ID	TICKETING_TICKET	SUPPORTSPECIALIST_ID	☐

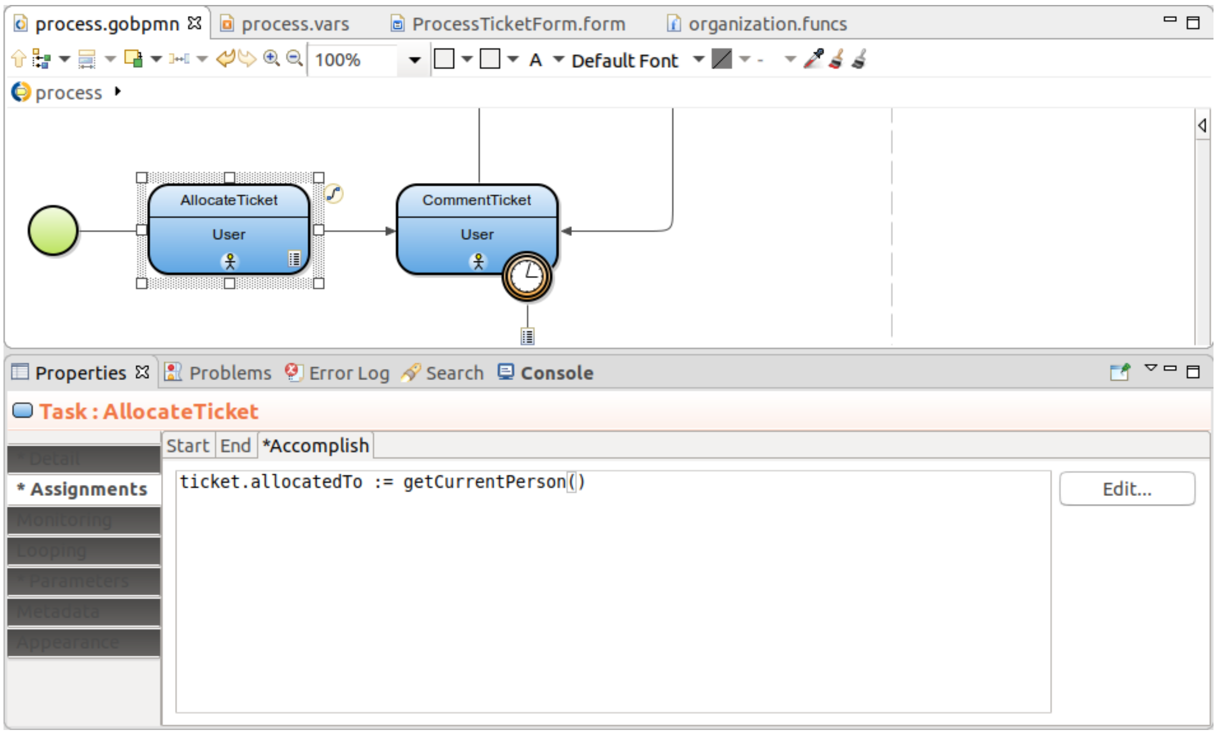
2. Adapt the ticket process: add a user task for support specialists that will allocate the ticket to the person who will deal with this to-do and adjust the rest of the flow.



Task : AllocateTicket

Parameters:
```
title /* String */ -> "New ticket " + ticket.id + " " + ticket.subject,
performers /* Set<Performer> */ -> { SupportSpecialist() },
uiDefinition /* UIDefinition */ -> new ProcessTicketForm(ticket)
```

3. Define the allocation on the AllocateTicket task on its Accomplish Assignments tab.



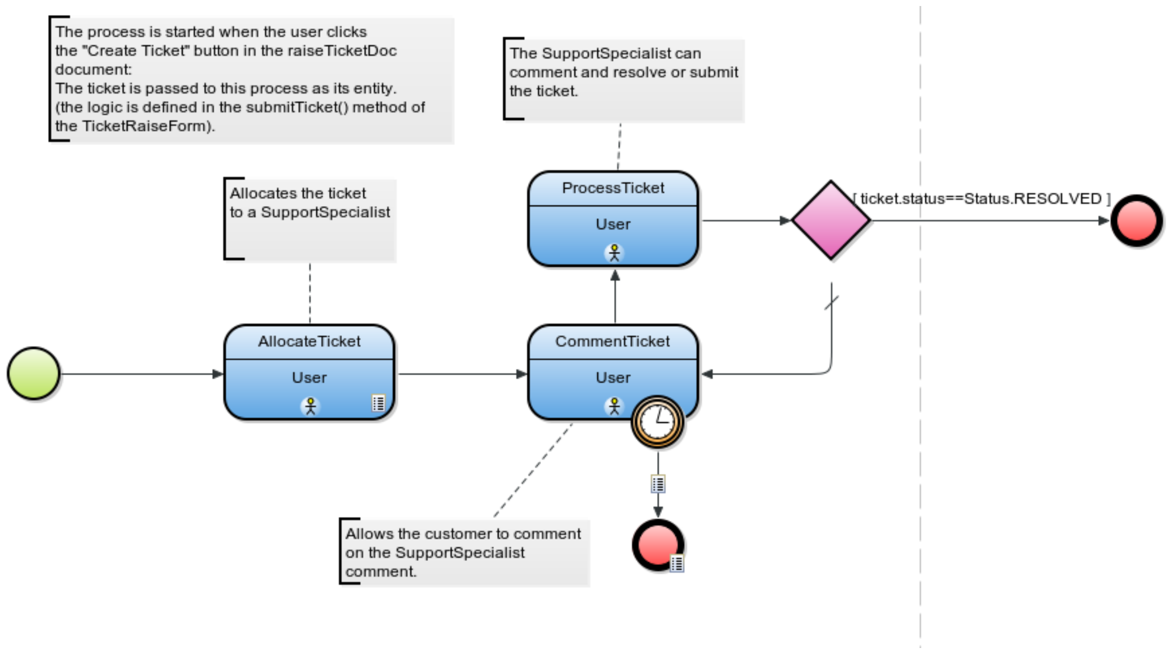
Task : AllocateTicket

```
ticket.allocatedTo := getCurrentPerson()
```

4. Adapt the *Performers* parameter of the *ProcessTicket* task to `performers /* Set<Performer> */ -> { ticket.allocatedTo }`

4.1.7.1 Tips and Takeaways

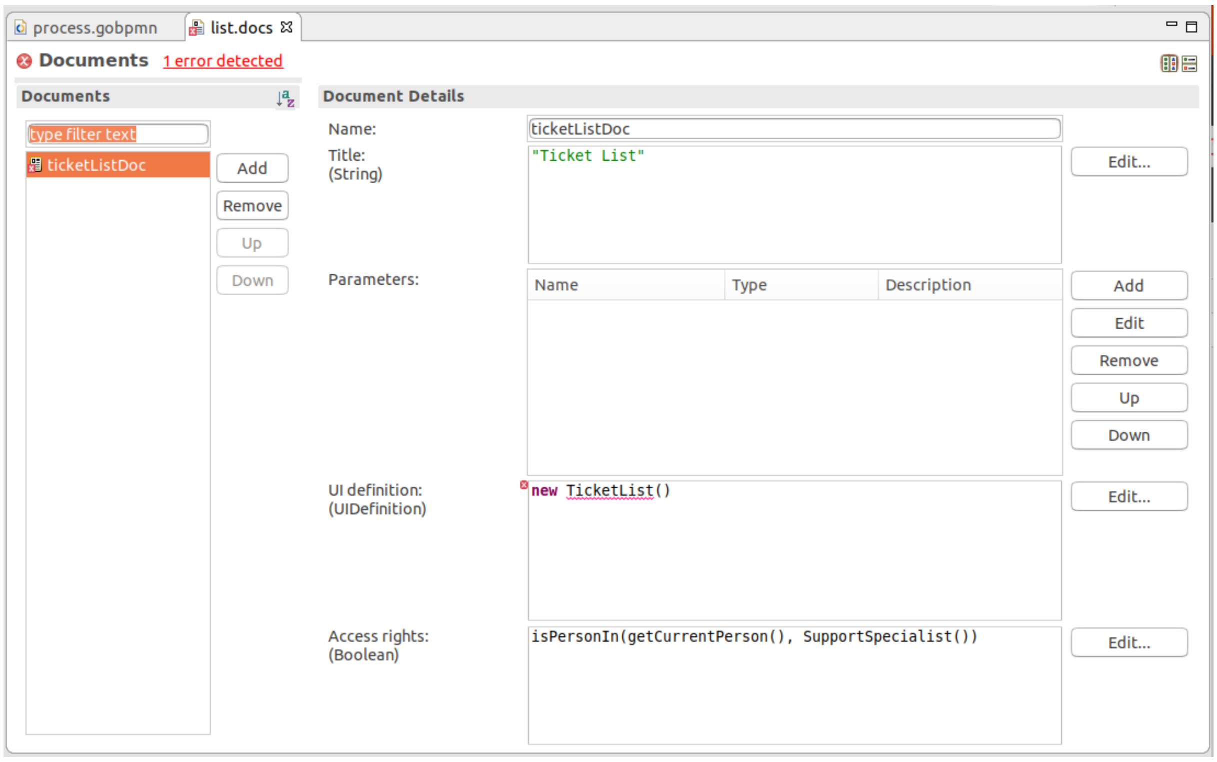
- Consider adding Annotations with comments on the process.


The process is started when the user clicks the "Create Ticket" button in the raiseTicketDoc document:
The ticket is passed to this process as its entity.
(the logic is defined in the submitTicket() method of the TicketRaiseForm).
The SupportSpecialist can comment and resolve or submit the ticket.
Allocates the ticket to a SupportSpecialist
ProcessTicket
User
[ticket.status==Status.RESOLVED]
AllocateTicket
User
CommentTicket
User
Allows the customer to comment on the SupportSpecialist comment.

4.1.8 Displaying a List of Tickets to a Support Specialist

User story: A support specialist can view all their open tickets on one page.

1. In *common*, create a query *getTickets* that returns all Tickets for a Person:
   - Name: getTickets
   - Record type: Ticket
   - Iterator: t
   - Parameters: supportSpecialist of type Person
   - Condition: `t.allocatedTo == supportSpecialist && t.status == Status.OP↩ EN`
   - Static ordering: `t.created`, DESC
2. Create a non-executable *list* module with *common* module import.
3. In the *list* module, create a document for the support specialist.


process.gobpmn | list.docs

Documents 1 error detected

Documents

type filter text

ticketListDoc

Add
Remove
Up
Down

Document Details

Name: ticketListDoc

Title: (String) "Ticket List" Edit...

Parameters:

Name	Type	Description

Add
Edit
Remove
Up
Down

UI definition: (UIDefinition)
```
new TicketList()
```
Edit...

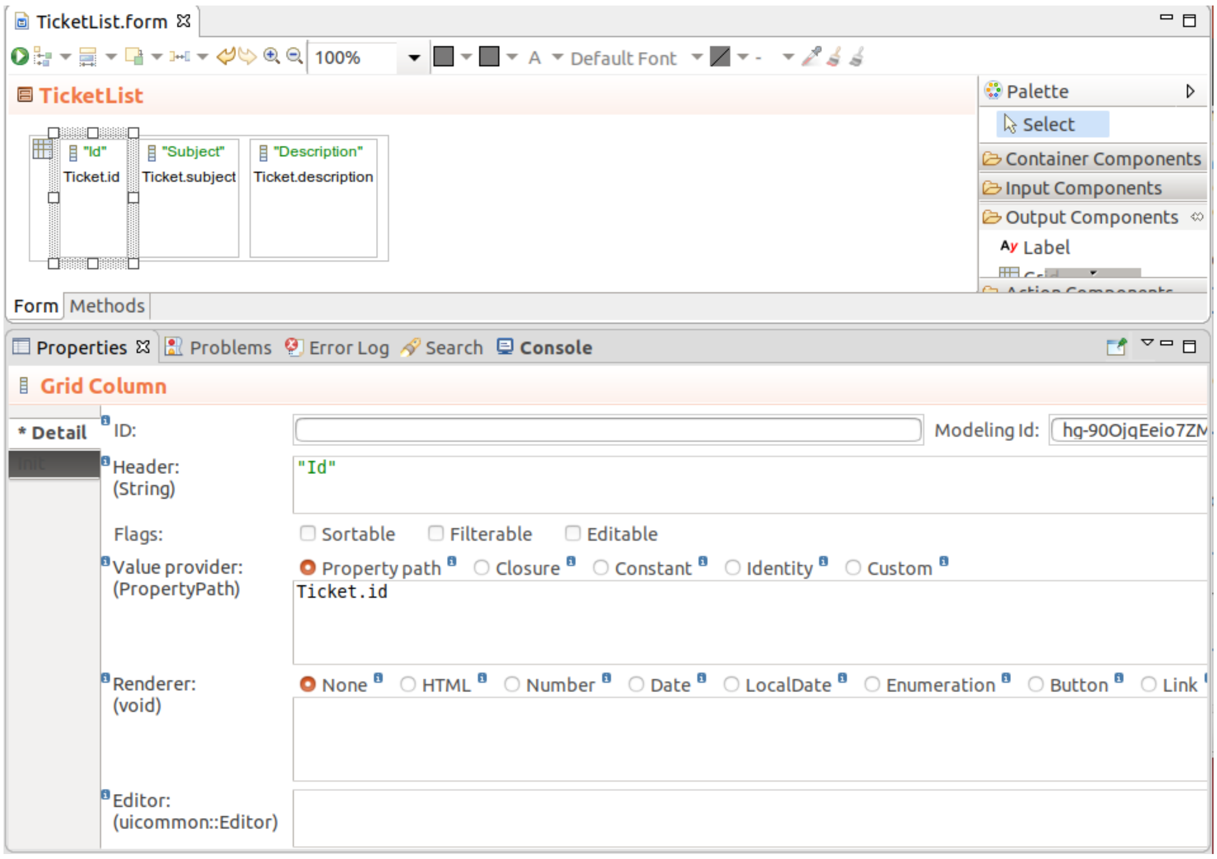
Access rights: (Boolean)
```
isPersonIn(getCurrentPerson(), SupportSpecialist())
```
Edit...

4. Create the form TicketList as depicted below.

   (a) Insert a Grid with ID `ticketGrid`.

   (b) Set the data source to Query with value *getTickets(getCurrentPerson())*.

   (c) Insert Columns with details.


TicketList.form

100% Default Font

TicketList

"Id"	"Subject"	"Description"
Ticket.id	Ticket.subject	Ticket.description

Palette
Select
Container Components
Input Components
Output Components
Label

Form | Methods

Properties | Problems | Error Log | Search | Console

Grid Column

* Detail

ID: Modeling Id: hg-90OjqEeio7ZM

Header: (String) "Id"

Flags: ☐ Sortable ☐ Filterable ☐ Editable

Value provider: (PropertyPath) [✓ Property path] Closure Constant Identity Custom
```
Ticket.id
```

Renderer: (void) [✓ None] HTML Number Date LocalDate Enumeration Button Link

Editor: (uicommon::Editor)

5. Upload the *list* module and check that the document is available for the users with the correct role.

4.1.9 Adding a Comment At Any Point

User story: The support specialist can view the details and add a comment to their open Tickets at any time.

Allow the support specialist to display details of a ticket from the ticket list and comment on the selected ticket.

1. Adapt *ProcessTicketForm*: You need to exclude the **Resolve** and **Submit** buttons from the ticket detail. To prevent copying the same code with ticket details, you need to create a new form with the ticket details and reuse this form in ProcessTicketForm:

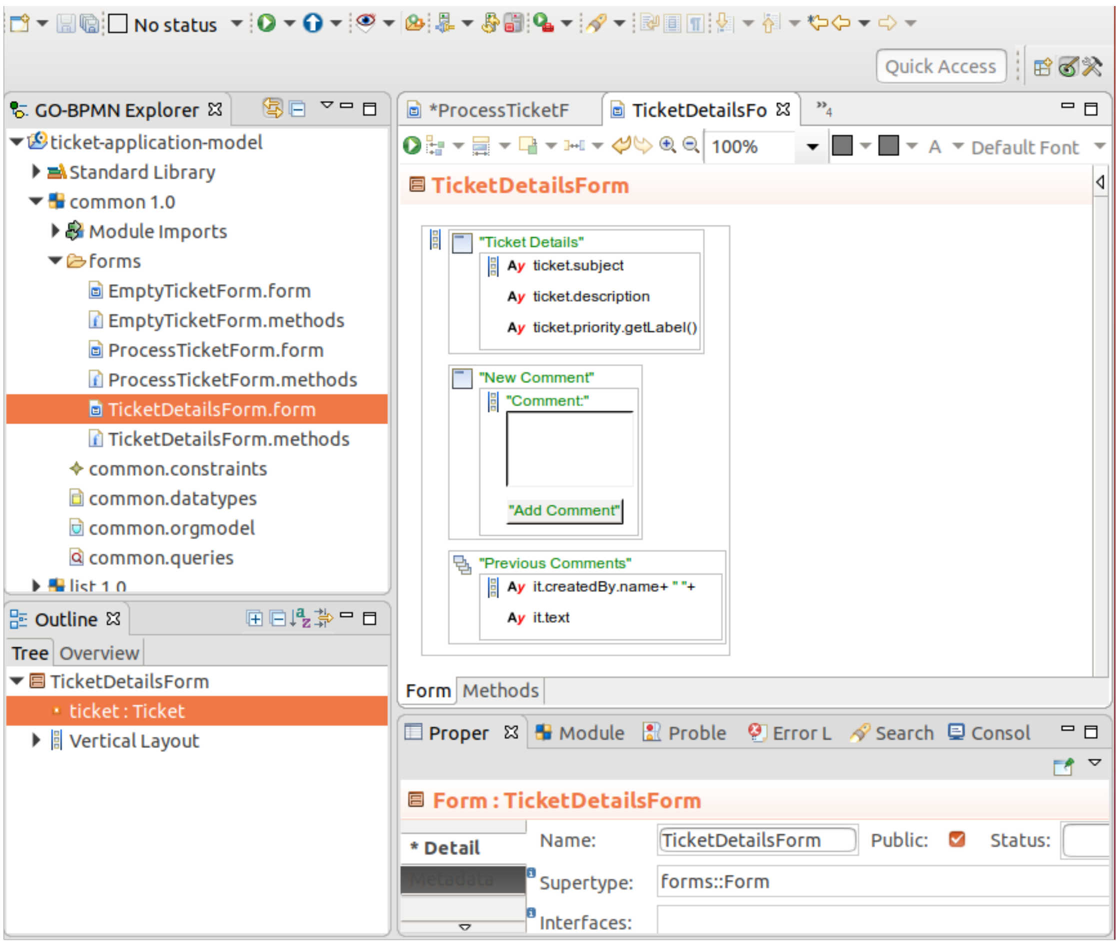
   (a) In the *common* module, create *TicketDetailsForm*:

      i. Cut all components from ProcessTicketForm apart from the buttons and insert everything into TicketDetailsForm: use copy-paste.

      ii. Create a local variable for the ticket.

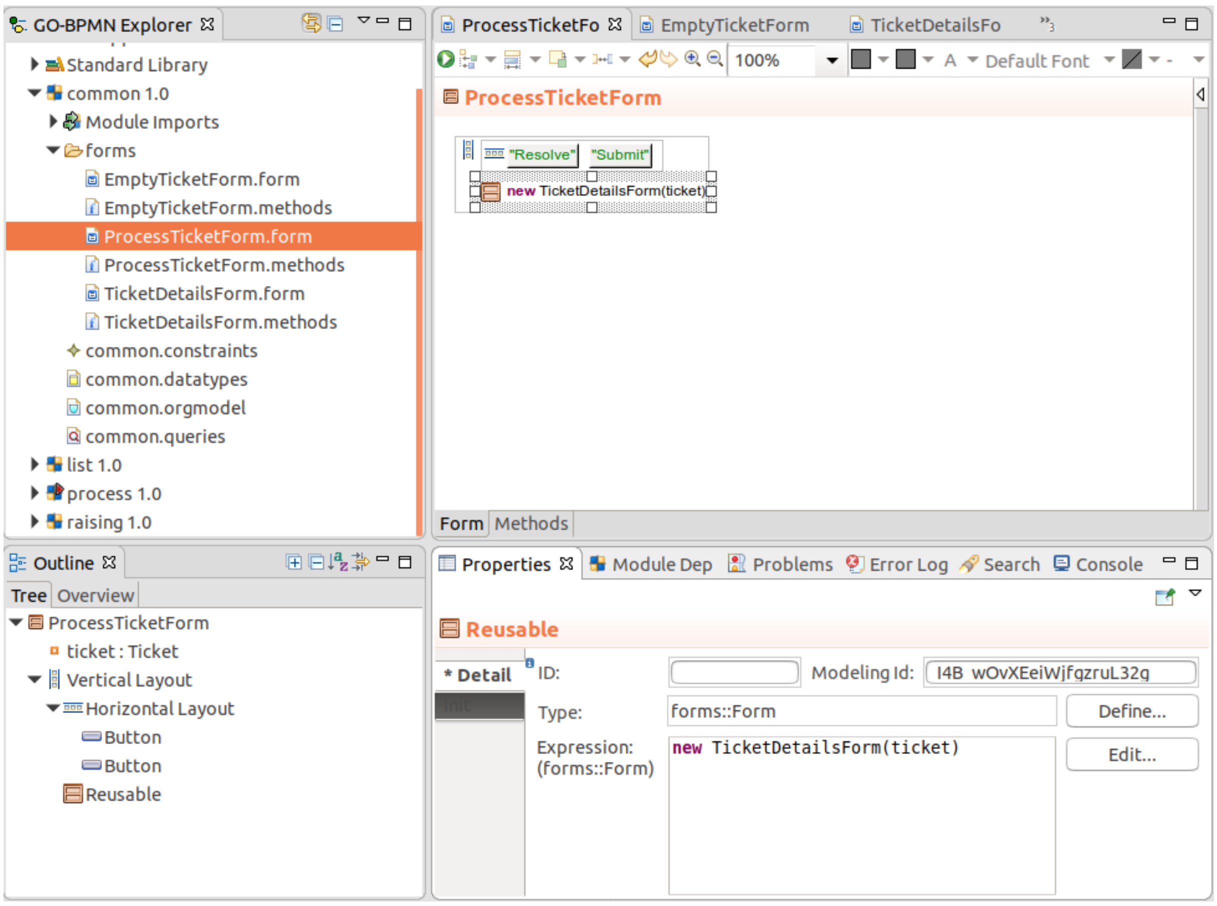
      iii. Create the parametric constructor with the ticket initialization.

```
 TicketDetailsForm {
  //constructor that initializes the form variable ticket
  //to the parameter argument:
  public TicketDetailsForm(Ticket ticket){
     this.ticket := ticket
  }
}
```



Name: TicketDetailsForm Public: ✓ Status:
Supertype: forms::Form
Interfaces:

2. Add a Reusable form component to ProcessTicketForm with TicketDetailsForm.



ID:
Modeling Id: I4B_wOvXEeiWjfqzruL32g
Type: forms::Form
Expression: (forms::Form)

```
new TicketDetailsForm(ticket)
```

3. Adapt *TicketList*:

   (a) Wrap the Grid of *TicketList* in a Vertical Layout and add a Vertical Split with ID `ticketSplit`.

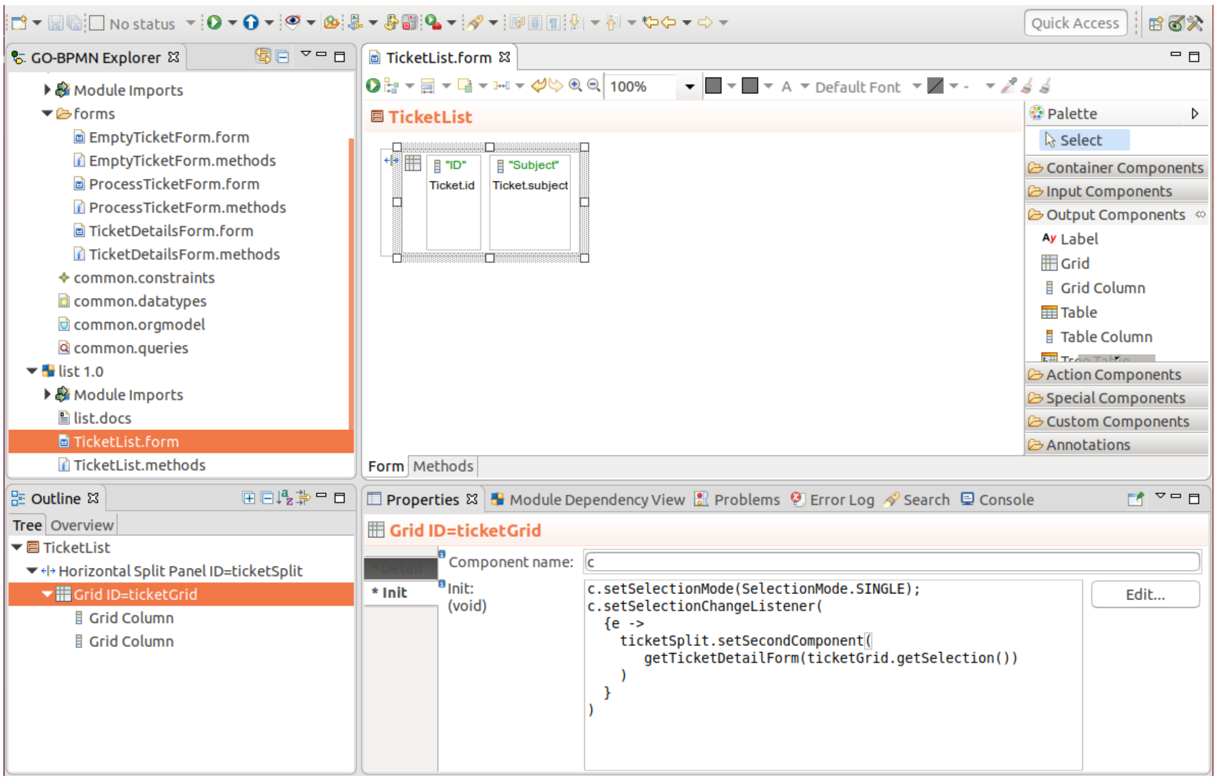
   (b) Enable selection in the grid: On the Init tab, call `c.setSelectionMode(SelectionMode.SI↩ NGLE)`

   (c) Still on the Init tab of the Grid, define to display the details of the selected ticket as the second component of the split, when the user selects a ticket in the grid.

```
 c.setSelectionChangeListener({e ->
~
    ticketSplit.setSecondComponent(
       getDetailForm(ticketGrid.getSelection())
    )
  }
)
```

4. Create the `getDetailForm()` method for the form. Note that selection can be null, either when the form is loaded or when the user unselects a ticket; You will display another form in such a case:

```
   private FormComponent getDetailForm(List<Object> selection){
~
       if ! selection.isEmpty() then
            (new TicketDetailsForm(selection[0] as Ticket)).getWidget()
         else
            (new EmptyTicketDetailsForm()).getWidget()
         end
  }
```



Component name: c

* Init

Init: (void)

```
c.setSelectionMode(SelectionMode.SINGLE);
c.setSelectionChangeListener(
  {e ->
    ticketSplit.setSecondComponent(
       getTicketDetailForm(ticketGrid.getSelection())
    )
  }
)
```

5. In *common*, create *EmptyTicketDetailsForm*.

4.1.10 What to do Next

Optionally, go through the `agile-pattern tutorial`.

Then proceed to Application

4.2 Application

Note: To follow through the steps below, you need the Enterprise Edition of LSPS.

In the previous chapters, you uploaded your modules to the PDS Embedded Server. This is an application server with the LSPS Application already deployed.

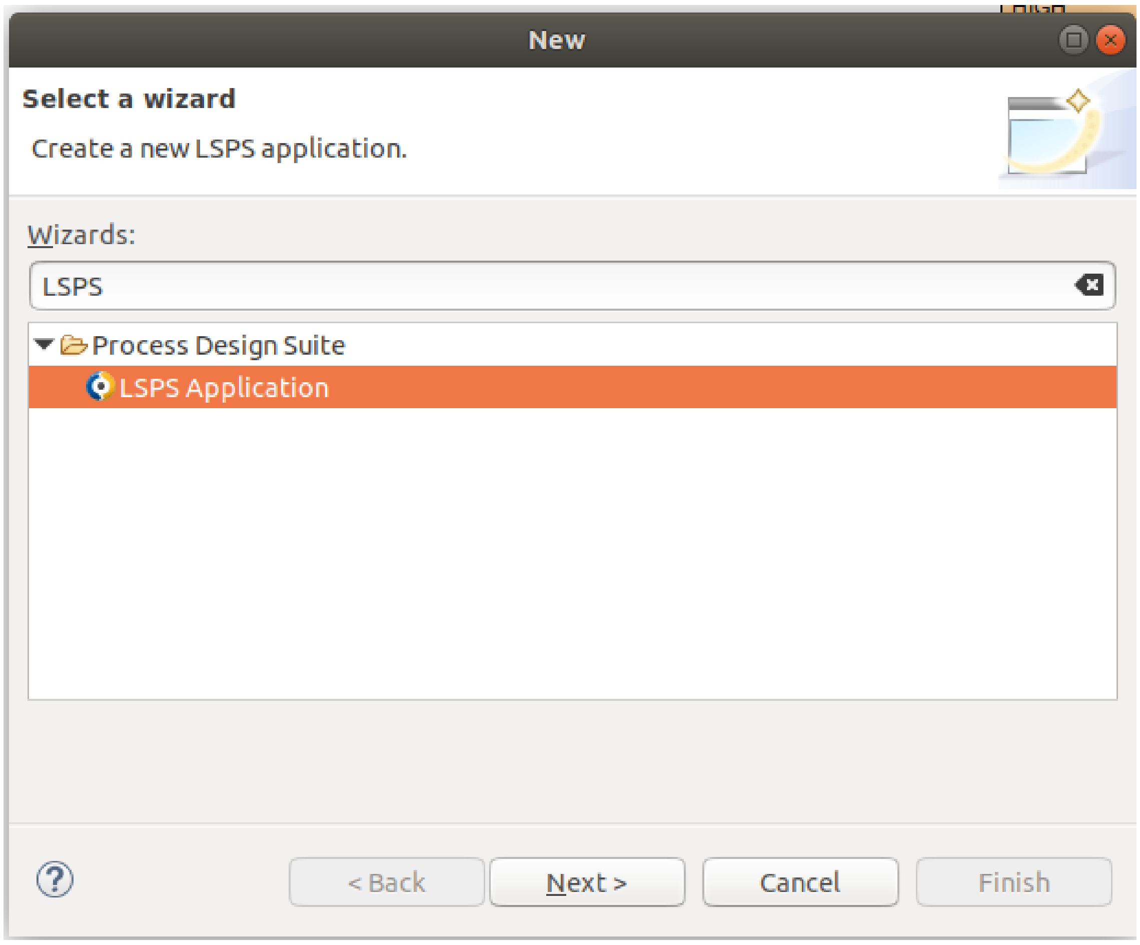
In real-world scenarios, such a solution is obviously not sufficient since it is not possible to customize the LSPS Application User Interface or add new business logic to the application.

To perform such customizations, you need the LSPS Repository delivered as a separate file. With the repository set up, you can generate the *Application User Interface* source and LSPS Application sources with their API, so you can modify the application:

- customize the Application User Interface
- add Java code, EJBs
- add custom task types, functions, form components.

4.2.1 Prerequisites

1. Make sure you have `LSPS Enterprise Edition and the LSPS Repository`.
2. Generate an LSPS Application:
   (a) Go to **File** > **New** > **Other**
   (b) Select **LSPS Application**.



   (c) Provide application details:
   - Application name: ticketapp
   - Application namespace: org.acme
   - Application namespace: org.acme
   - Base java package name: org.acme.ticketapp

   (d) Wait until the notification popup appears.
   Note that PDS generates not only the application but also an embedded server and related resources.
3. Put the resources including your model under version control. Consider excluding the following from version control:

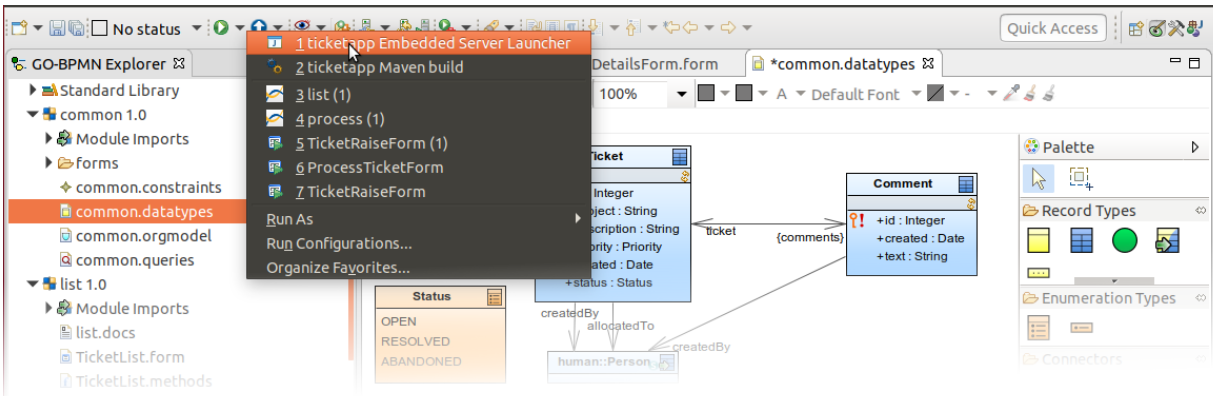
```
# builds:
target/
# eclipse project information
#(generated by maven build):
.project
# eclipse workspace data:
.metadata
eclipse-target
```

4. For directories with GO-BPMN projects, make sure to add the `.project` directories to version control since these are not generated by maven: If using git, add `!.project` to .gitignore in the directories.

Local git repo setup

```
~/ticketing-app$ ls
ticketingapp ticketingapp-model
~/ticketing-app$ ls ticketingapp/
archetype-resources  pom.xml  ticketingapp-ear ticketingapp-ejb
ticketingapp-embedded  ticketingapp-tester  ticketingapp-vaadin
ticketingapp-vaadin-war
~/ticketing-app$ echo "target/
> .project
> .metadata
> eclipse-target" > .gitignore
~/ticketing-app$ echo '!.project' > ticketingapp-model/.gitignore
~/ticketing-app$ git init
Initialized empty Git repository in /home/eko/ticketing-app/.git/
~/ticketing-app$ git add *
~/ticketing-app$ git add .gitignore
~/ticketing-app$ git commit -am"init"
```

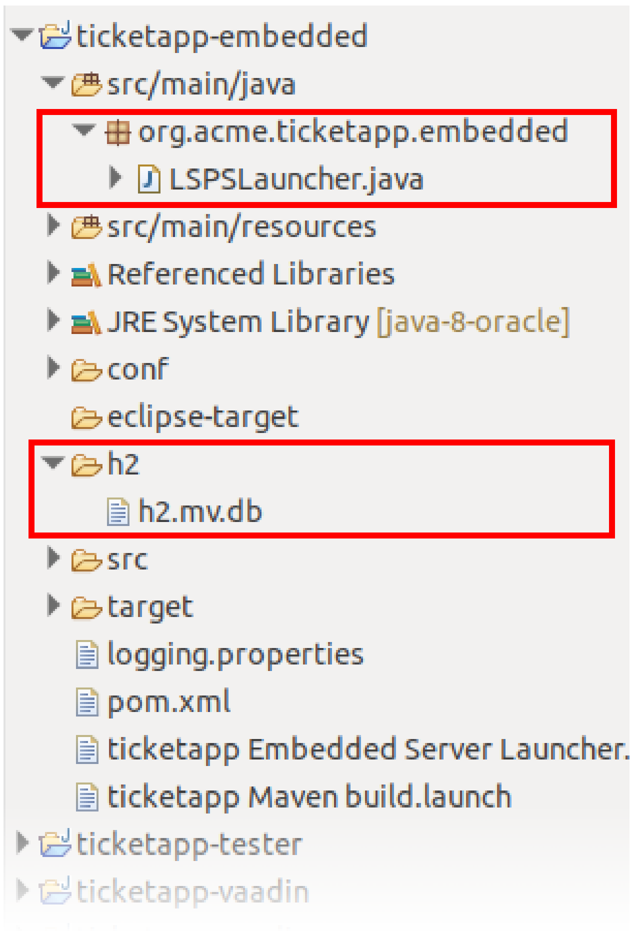
5. Back in PDS, run the SDK embedded server with the application.



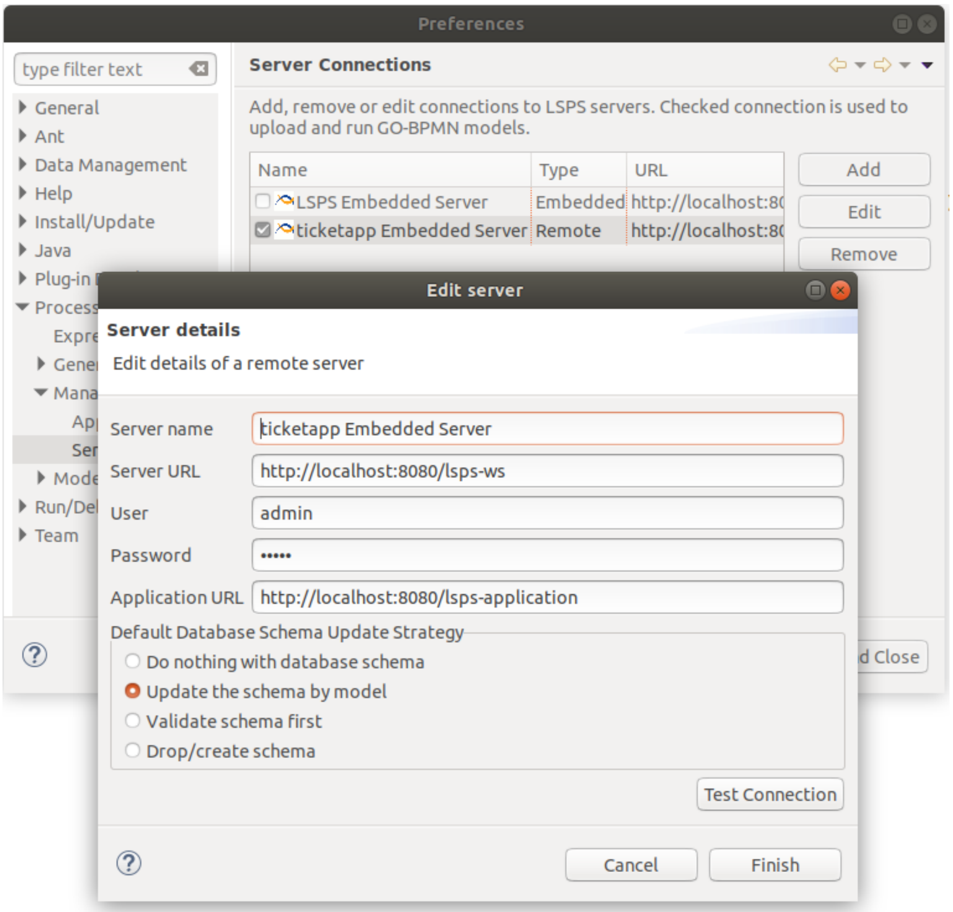
The server is located in the `ticketapp-embedded` project:

- The Launcher you just used is in its *embedded* package.
- The database in the *h2* directory.

6. Since we are going to work with the LSPS Application API, switch to the Java perspective.



Note also that the connection has been added to the Server Connections list and you can define its properties including the database-schema update strategy.



Server Connections

Add, remove or edit connections to LSPS servers. Checked connection is used to upload and run GO-BPMN models.

	Name	Type	URL
☐	LSPS Embedded Server	Embedded	http://localhost:8(
☑	ticketapp Embedded Server	Remote	http://localhost:8(

Server details

Edit details of a remote server

Server name	ticketapp Embedded Server
Server URL	http://localhost:8080/lsps-ws
User	admin
Password	•••••
Application URL	http://localhost:8080/lsps-application

Default Database Schema Update Strategy

- Do nothing with database schema
- [✓ Update the schema by model]
- Validate schema first
- Drop/create schema

If you go to `localhost:8080/lsps-application/ui`, you will access the LSPS Application of the SDK Embedded Server you have just started up.

4.2.2 Displaying Only Relevant Items in Navigation Menu

User story: In the *Application User Interface*, users can access from their navigation menu only the documents they have access to and the to-do list.

Create your implementation of the menu:

1. In the `core` package of the `<YOUR_APP>-vaadin` project, create the `AppMainMenu` class that extends *DefaultMainMenu*.
2. Create the `getNavigator()` method to allow integration with View switching.
3. Copy the implementation of the DefaultMainMenu.createMenu() method to the AppMainMenu implementation.
4. Remove the `addRunModelMenuItem(navigationMenu)` and `addDocumentsMenuItem(navigation↩ Menu)` calls.
5. Add navigation to the documents for raising a ticket and the list of tickets with with `adddDocumentMenu↩ Item().`
6. Make the `AppAppLayout.createMainMenu()` method return *AppMainMenu*.

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

7. Rebuild your application (you can use the maven launcher in the drop-down of the Run button on the main toolbar).
8. Restart the server.
9. Upload the `raising` and `list` modules
10. Assign the SupportSpecialist and Customer roles to users.
11. Check the main menu of the users.

AppMainMenu class

```
public class AppMainMenu extends DefaultMainMenu {
~
  public AppMainMenu(String closeOnTapAreaId) {
    super(closeOnTapAreaId);
  }
~
  private AppNavigator getNavigator() {
    LspsUI ui = (LspsUI) UI.getCurrent();
    return ui.getNavigator();
  }
~
  @Override
  protected NavigationMenu createMenu() {
    NavigationMenu navigationMenu = new NavigationMenu(createUserMenu());
~
    LspsUI ui = (LspsUI) UI.getCurrent();
    UserInfo user = ui.getUser();
~
    if (user.hasRight(HumanRights.READ_ALL_TODO) || user.hasRight(HumanRights.READ_OWN_TODO)) {
      addTodoListMenuItem(navigationMenu);
    }
    if (hasRightToOpenDocument("raising::raiseTicketDoc")) {
      navigationMenu.addDocumentItem("New Ticket", "raising::raiseTicketDoc", null, VaadinIcons.AN
    }
    if (hasRightToOpenDocument("list::ticketListDoc")) {
      navigationMenu.addDocumentItem("My Tickets", "list::ticketListDoc", null, VaadinIcons.CLIPBO
    }
~
    return navigationMenu;
  }
}
```

4.2.2.1 Tips and Tweaks

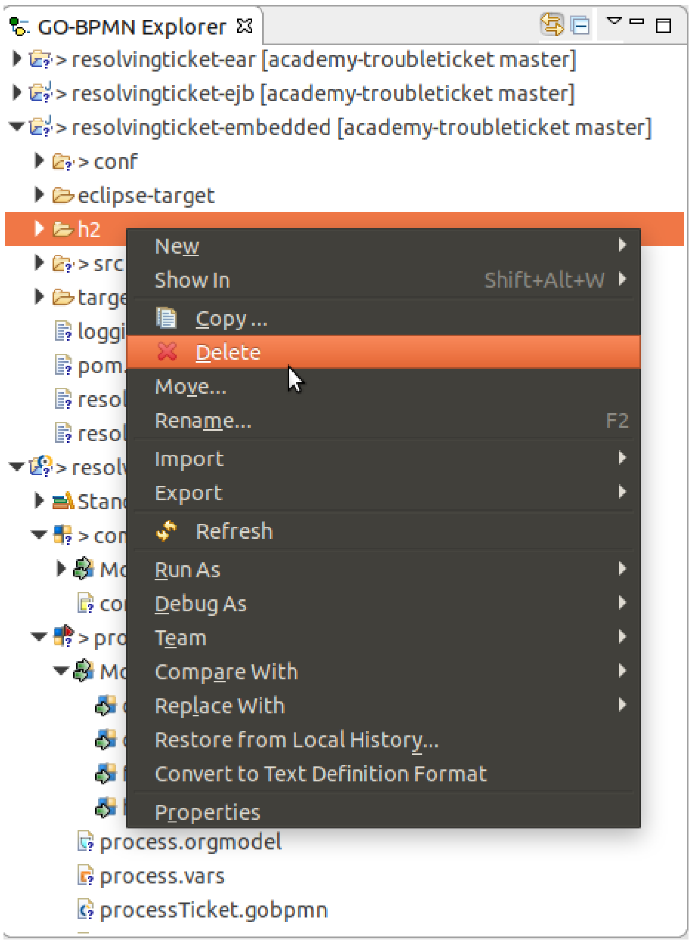
- You can now access the document directly from the navigation menu, but the menu items contain the `???` characters: these signalize that the system failed to find the localization string. Let's create the string:
  1. Instead of String literals for the navigation item labels, use a localization key name:

```
...
    if (hasRightToOpenDocument("raising::raiseTicketDoc")) {
      navigationMenu.addDocumentItem("nav.newticket", "raising::raiseTicketDoc", null, Va
    }
    if (hasRightToOpenDocument("list::raiseTicketDoc")) {
      navigationMenu.addDocumentItem("nav.usertickets", "list::ticketListDoc", null, Vaad
    }...
```

2. Open properties file in the `com.whitestein.lsps.vaadin.webapp` package (`<YOUR_APP>-vaadin` project), for example, the **localization.properties** file with the default localizations.
3. Add the localization keys with their values:

```
# navigation menu items
nav.todoitems = To-do List
nav.documents = Documents
nav.runProcess = Run Model
# These are new keys:
nav.newticket = New Ticket
nav.usertickets = My Tickets
```

- Run the embedded server preferably in debug mode so your changes can be applied on the fly when possible.
- To delete the server database, stop the server remove the `h2` directory in the embedded project.

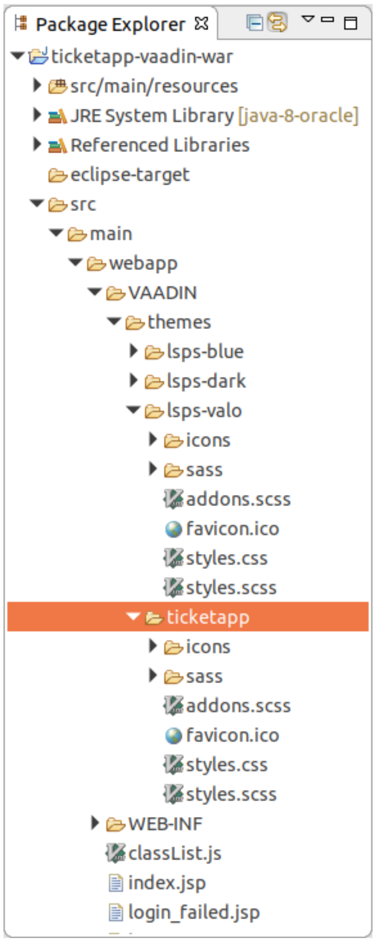

GO-BPMN Explorer
> resolvingticket-ear [academy-troubleticket master]
> resolvingticket-ejb [academy-troubleticket master]
> resolvingticket-embedded [academy-troubleticket master]
> conf
eclipse-target
h2
New
Show In Shift+Alt+W
Copy...
Delete
Move...
Rename... F2
Import
Export
Refresh
Run As
Debug As
Team
Compare With
Replace With
Restore from Local History...
Convert to Text Definition Format
Properties
process.orgmodel
process.vars
processTicket.gobpmn

4.2.3 Changing Application Colors

User story: The *Application User Interface* follows the black-and-white color schema.

You can switch between the available style themes on the Settings page: click Settings in your profile menu to navigate to the page. By default the *lsps-valo* is used. You will use this theme as base for a customized theme; you can pick the dark or blue theme if it is closer to the look you want to achieve.

1. Back in PDS, first set up your theme:
   (a) Create a copy of the *lsps-valo* theme in the `vaadin-war` project.



(b) In *style.scss* in the theme directory, change the theme class name to the name of the theme directory.

```
.ticketapp {
@include addons;
@include lsps-valo-base;
@include theme-app;
 }
```

(c) In ticketapp-vaadin, go to *org.acme.ticketapp.core.AppUIProvider* and add the `getTheme↩ Manager()` method:

```
@Override
  protected ThemeManager getThemeManager() {
    ImmutableSet<String> themes = ImmutableSet.of("lsps-dark", "lsps-valo", "lsps-blue",
    return new ThemeManager("ticketapp", themes);
  }
```

(d) Rebuild the application and restart the sdk embedded server.

(e) Go to `localhost:8080/lsps-application` and log in to make sure that the theme has been loaded successfully.

2. Set up a sass compiler configuration, so you can easily preview the changes of the theme:

(a) Go to Run > Run Configurations.

(b) In the left pane, select Java Application and click the New button in the caption area of the pane.

(c) On the right, define the following:

- Name: a name of the configuration
- Project: <YOUR_APP>-vaadin-war
- Main class: com.vaadin.sass.SassCompiler

(d) Switch to the Arguments tab and define the input scss file and output css file as input Program arguments: the output css file must be located in the theme directory of the target directory and have the name styles.css:

```
 src/main/webapp/VAADIN/themes/ticketapp/styles.scss
src/main/webapp/VAADIN/themes/ticketapp/styles.css
```

3. Set the *productionMode* parameter in `ticketapp-vaadin-war/src/main/webapp/WEB-IN↩ F/web.xml` to `false`.
4. In `ticketapp-vaadin-war/src/main/webapp/VAADIN/themes/ticketapp/sass/_↩ variables.scss`, edit or add the theme properties:

```
$valo-menu-background-color:  hsl(210, 0%, 98%);
$v-background-color: hsl(210, 0%, 98%);
$v-focus-color: #adadad;
```

5. Run the Sass compiler configuration.
6. Run the embedded server in *debug*.

4.2.4 Customizing the Login Page

User story: Login and logout pages follow the style.

1. Open `ticketapp-vaadin-war/src/main/webapp/login.jsp`.
2. Change the title to `<title>Trouble Ticket Application</title>`.
3. In `ticketapp-vaadin-war/src/main/webapp/VAADIN/themes/ticketapp/sass`, create ∗_login.scss∗ with the rules for the login page:

```
@mixin _login {
    &.login-page #loginHeader {
       background-color: white !important;
       background-image: url("../img/logo_splash.png") !important;
       background-position: center bottom;
       background-repeat: no-repeat;
       background-size: 300px auto;
    }
    &.login-page form table {
       background-color: white;
       background-image: none;
    }
    &.login-page input{
       border: 1px solid #c9c9c9;;
    }
    &.login-page .v-button .v-button-caption {
       color: #c9c9c9 !important;
    }
}
```

4. Import the mixin in styles.scss and include it in your theme:

```
@import "sass/_variables";
//added:
@import "sass/_login";
...
.ticketapp {
  @include addons;
  @include lsps-valo-base;
  @include theme-app;
  //added:
  @include _login;
}
```

5. Run Sass compiler and check the login page in your browser.

4.2.5 Removing the Logo from the Top Bar

To remove logo located in the upper-left corner, you need to remove TopBanner from NavigationMenu:

1. In the *core* package in *ticketapp-vaadin*, create `AppNavigationMenu` that extends *NavigationMenu*.
2. Override `createTopBanner()` method.

```
 public class AppNavigationMenu extends NavigationMenu {
~
  public AppNavigationMenu(Component userSubmenu) {
    super(userSubmenu);
  }
~
  //this removes the logo slot:
  @Override
  protected Component createTopBanner() {
    return null;
  }
}
```

3. Make *AppMainMenu* use `AppNavigationMenu` instead of the default NavigationMenu:

```
...
  @Override
  protected NavigationMenu createMenu() {
    //changed NavigationMenu to AppNavigationMenu:
    NavigationMenu navigationMenu = new AppNavigationMenu(createUserMenu());
```

4. Rebuild the application and restart the server.

4.2.6 Exporting a Ticket to PDF

User story: Users can export the details of a ticket to a PDF from ticket details.

There is no support for PDF export out-of-the-box: You will use the Apache pdfbox to implement a custom function for PDF-export.

1. Add the pdfbox dependency to your ejb project:
   (a) In a text editor, add the dependency to the root pom.xml in `<WORKSPACE>/ticketappfile`

```
...
      <dependency>
        <groupId>com.whitestein.lsps.monitoring</groupId>
        <artifactId>lsps-monitoring-client</artifactId>
        <version>${lsps.version}</version>
      </dependency>
      <!-- Added: -->
      <dependency>
        <groupId>org.apache.pdfbox</groupId>
        <artifactId>pdfbox</artifactId>
        <version>2.0.12</version>
      </dependency>
```

   (b) In the pom.xml in *ticketapp-ejb*, add the pdfbox dependency:

```
...
        <artifactId>lsps-os-exec</artifactId>
        <scope>provided</scope>
      </dependency>
      <!-- Added: -->
      <dependency>
        <groupId>org.apache.pdfbox</groupId>
        <artifactId>pdfbox</artifactId>
      </dependency>
...
```

   (c) Run maven build.

2. In the *common* module, create a function definition file with the definition of the exportTicketToPdf() function:

```
public void exportTicketToPdf(Ticket t)
  native org.acme.ticketapp.pdf.PdfExporter.exportTicketToPdf;
```

3. In the *ticketapp-ejb* project, create the *org.acme.ticketapp.pdf* package

   (a) Create the `PdfExporter` class.

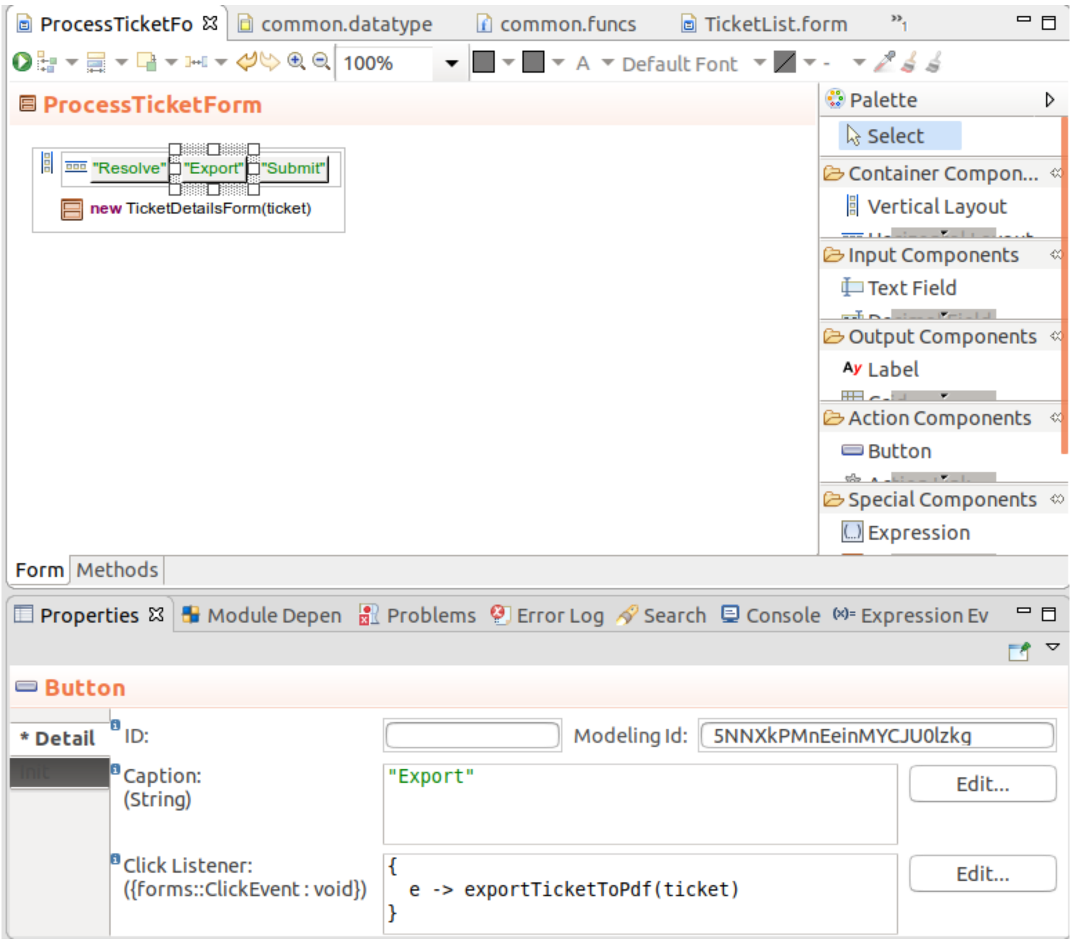
   (b) Implement the `exportTicketToPdf(ExecutionContext ctx, RecordHolder ticket)` method.
   Example demo implementation of the PdfExporter class:

```
public class PdfExporter {
~
  private PDPage myPage = new PDPage();
  private PDDocument doc = new PDDocument();
  private PDPageContentStream content;
~
  public void exportTicketToPdf(ExecutionContext ctx, RecordHolder ticket) {
~
    SetHolder coms = (SetHolder) ticket.getProperty("comments");
~
    String title = ticket.getProperty("id") +
        (String) ticket.getProperty("subject") +
        ticket.getProperty("created");
~
    try {
      content = new PDPageContentStream(doc, myPage);
      doc.addPage(myPage);
~
      content.setFont(PDType1Font.TIMES_ROMAN, 12);
      content.setLeading(14.5f);
~
      printTitle(title);
      printComments(coms);
~
      content.close();
~
      doc.save("ticket.pdf");
      doc.close();
~
      Desktop.getDesktop().open(new File("ticket.pdf"));
    } catch (IOException e) {
      throw new RuntimeException("PDF export failed", e);
    }
~
  }
~
  private void printTitle(String title) throws IOException {
    content.beginText();
    content.newLineAtOffset(25, 700);
    content.showText(title);
    content.endText();
  }
~
  private void printComments(SetHolder comments) throws IOException {
~
    int y = 680;
    for (Object commentRecordHolder : comments) {
      String commentText = (String) ((RecordHolder) commentRecordHolder).getProperty("tex
      Date commentDate = (Date) ((RecordHolder) commentRecordHolder).getProperty("created
      RecordHolder author = (RecordHolder) ((RecordHolder) commentRecordHolder).getProper
~
      content.beginText();
      content.newLine();
      content.newLineAtOffset(25, y);
      content.showText(commentText + " on " + commentDate + " by " + author.getProperty("
      content.endText();
~
      y -= 20;
    }
  }
}
```

Note the following:

- The com.whitestein.lsps.engine.lang.ExecutionContext parameter is passed automatically and gives you access to such data as the current namespace, model instance, module variables, etc.
- To acquire related records, we are using the `getProperty()` call.
- Records are reflected as *RecordHolder* objects, sets as *SetHolder* object, etc.

4. In ProcessTicketForm, add the **Export** button that calls the function.



ID:
Modeling Id: 5NNXkPMnEeinMYCJU0lzkg
Caption: (String) "Export" Edit...
Click Listener: ({forms::ClickEvent : void})
```
{
  e -> exportTicketToPdf(ticket)
}
```
Edit...

5. Rebuild the application.
6. Restart the SDK Embedded Server.
7. Upload the *process* model.
8. Test the PDF export.

4.2.7 Relevant Documentation

Further information and instructions on how to customize LSPS Application are available in the `Development Guide` and in `Javadoc`.

4.2.8 What to do next?

Consider taking a look at `tutorials`.