WHITESTEIN
Technologies

Living Systems® Process Suite

Academy

Living Systems Process Suite Documentation

3.1
Tue Jan 12 2021

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

Copyright © 2007-2021 Whitestein Technologies AG
All rights reserved.

Copyright © 2007-2021 Whitestein Technologies AG.

This document is part of the Living Systems® Process Suite product, and its use is governed by the corresponding license agreement. All rights reserved.

Whitestein Technologies, Living Systems, and the corresponding logos are registered trademarks of Whitestein Technologies AG. Java and all Java-based trademarks are trademarks of Oracle and/or its affiliates. Other company, product, or service names may be trademarks or service marks of their respective holders.

Contents

Chapter 1

Academy

This guide leads you through the main features of LSPS: Each chapter focuses on a particular part of the product and where applicable provides links to further information.

After you have been gone through the Academy guide, you should have a good overview of LSPS features and should be able to work independently with the help of reference documentation.

Chapter 2

BPMN and LSPS

2.1 What is BPMN?

Originally IT infrastructure relied on a model where only the business data was stored leaving any knowledge on how to work with the data with humans: for example, if a client wanted to open a bank account, the client filled out an application form and a bank employee performed and overlooked the processing of the application, which could lead to errors.

BPMN is a standard that defines how to capture and execute such processing in a set of steps that achieve your goal called a *business process*. These steps can be performed by people or automatically. Business processes are represented visually and executed in a standardized way: the visual representation 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.

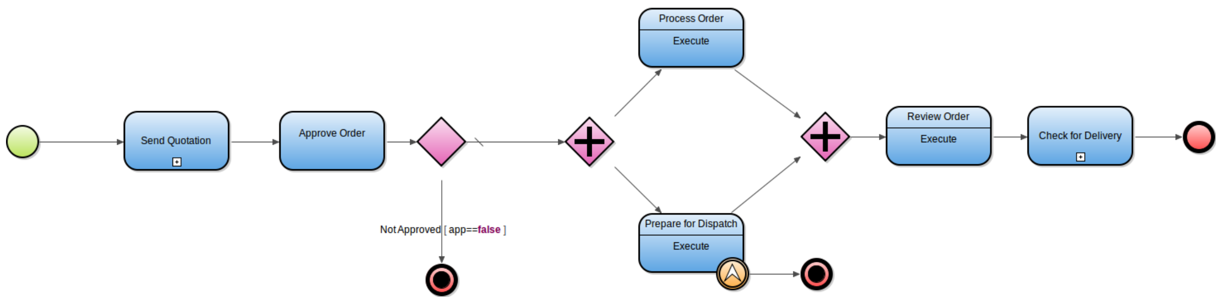
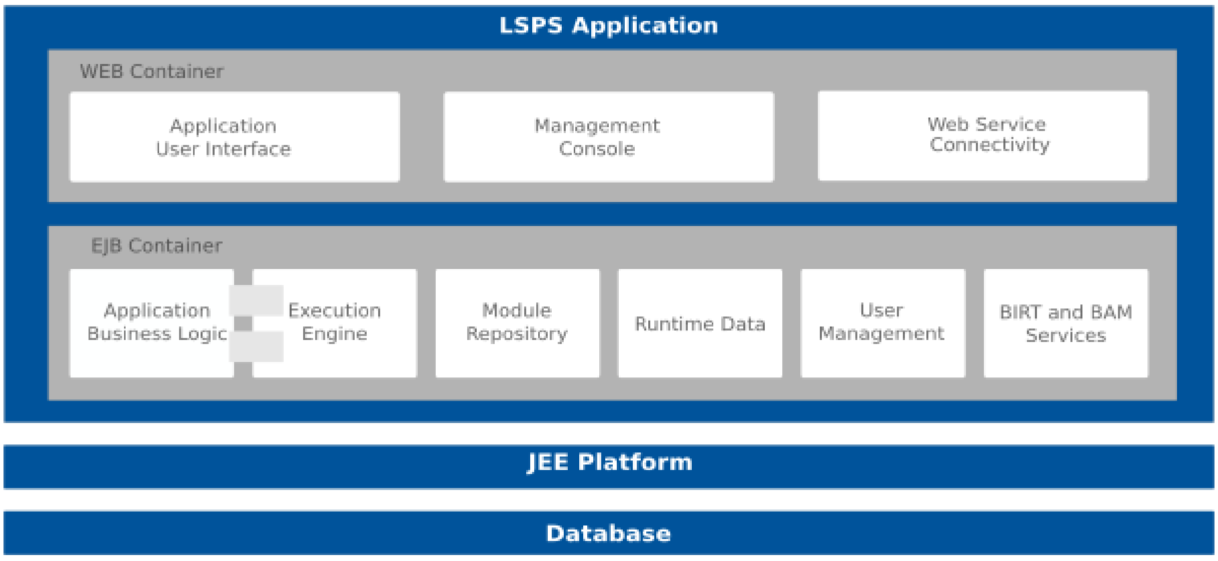


Figure 2.1 Process in BPMN notation

2.2 What is LSPS?

LSPS is a set of tools that allows you to design and execute BPMN processes. Built up based on BPMN, LSPS expands its capabilities: it adds support for GO-BPMN, conservative goal-focused processes; role-based access control; front-end page definitions, data type models, and much more.

The core of LSPS is the LSPS Application, a standard JEE enterprise application. The application stores your models, executes them, manages users, distributes work, provides access to the data from web applications, communicates with external systems and users, etc.



To create the models of your business processes for the LSPS Application, you will use the **Process Design Suite**. Apart from being a development tool for models, it is also a thick client for management and administration of the LSPS Application resources.

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

- If you are a process participant, use the **Application User Interface**, a web application that serves to get input for processes from human users.
- If you are an administrator, use the cli console or **Management Console**, a web application that serves to manage the resources of the LSPS Application, such as, models, model instances, users, etc.
- If you need to integrate with other systems or tools, use the web services API.

Depending on your needs, consider to install the following:

- **Enterprise Edition** comes with the Process Design Suite and LSPS SDK, which allows you to generate the LSPS Application EAR with the source code of the *Application User Interface*.
- **Runtime Suite** contains the LSPS Server EAR, CLI console, and database migration scripts. It is intended as an administration tool bundle.

To install the enterprise edition, follow the instructions in the `Installation Guide`.

Chapter 3

Hands-on Example

3.1 Goals

We are going to build an example with a functioning client registration: a client will log in to the application and enter their registration data. After they submit the data, an employee will review and confirm the data.

We will need the following:

1. Custom data type for the registration data
2. Roles that represent the client and the employee
3. Forms for registration and for confirmation of registration
4. Variables that hold the registration data
5. Process that will steer through the registration and the approval

3.2 Model Structure

When you want to create a Model, you create a structure for your resources:

- GO-BPMN Project: directory in your workspace (workspace is the directory PDS uses to store your projects during a session)
- GO-BPMN Module: special type of directory with resources that can be imported to other modules
- Definition files: files with certain type of data needed by the module

First we need to create the structure that will hold our resources: all GO-BPMN resources in your workspace must be stored in a special directory called GO-BPMN project. Though not uploaded to the server, they are intended for better organization of your sources.

To create a GO-BPMN project, do the following:

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 on the main menu and select the Modeling perspective in the dialog.

2. On the main menu, go to File > New > GO-BPMN Project.

3. In the **New GO-BPMN Project** dialog, enter the **HandsOn** project name and confirm.

In the dialog, notice the GO-BPMN Libraries section below: these are libraries with useful resources; however, often you will need only a subset of the available libraries: Generally the only library you will need is the Standard Library. Feel free to remove the other libraries to keep your workspace clean.

The GO-BPMN Explorer contains your project directory.

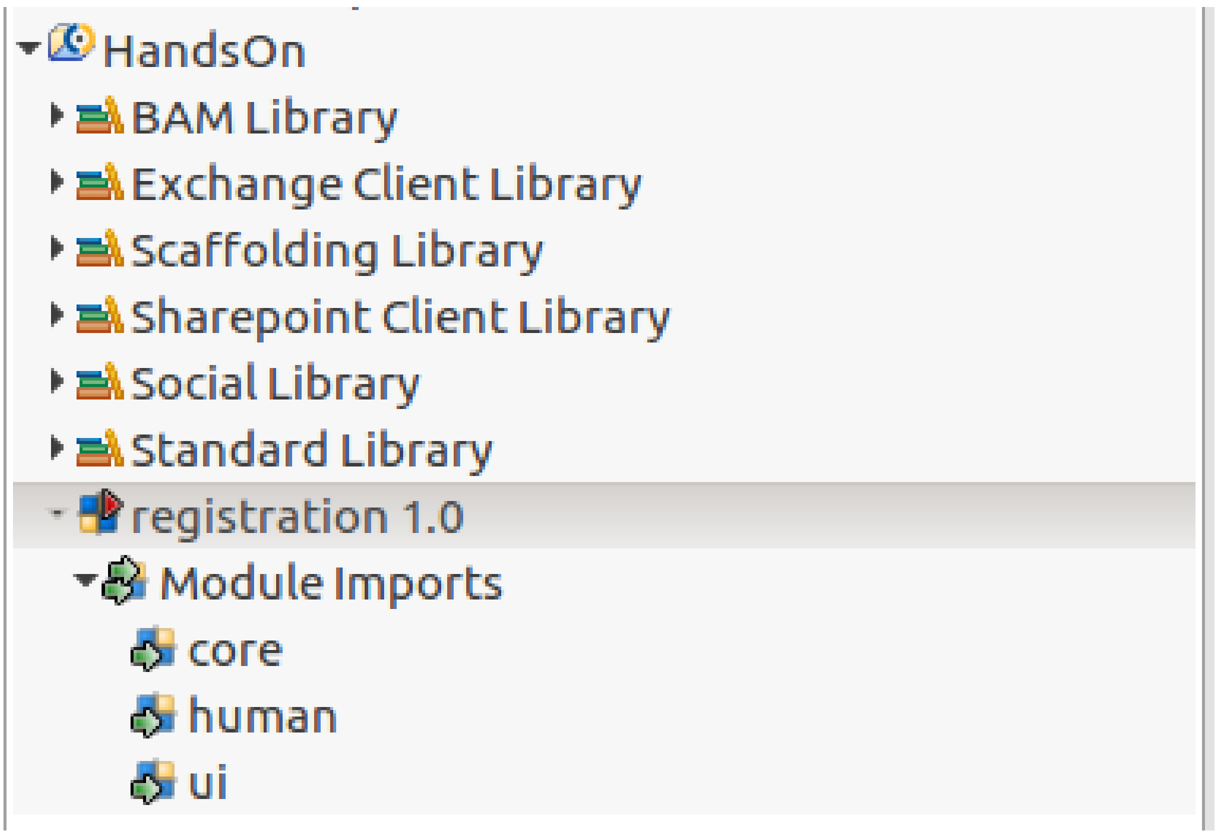
Now you will create GO-BPMN modules: modules serve to organize your model resources, and if they are marked as executable they can become model instances:

1. Right-click the project and click New > GO-BPMN Module.

2. In the dialog, enter the module name **registration**: notice the **executable module** option: if you unselect the option, you will not be able to create a model instance of the module. Non-executable modules serve to hold such resources as data models, role models, etc. If you create a process in this module, you will not be able to execute it by itself. For now, leave the option selected.

3. Click **Next**: the dialog displays the list of Library modules that you can import to your module: if you left only the Standard Library in your project, only the modules of this library are listed: this is the minimum you need to create models.

4. Click **Finish**. Now the project contains a module: notice the red arrow marker on the module icon: this means it is an executable module. You are ready to create module resources with your business data.



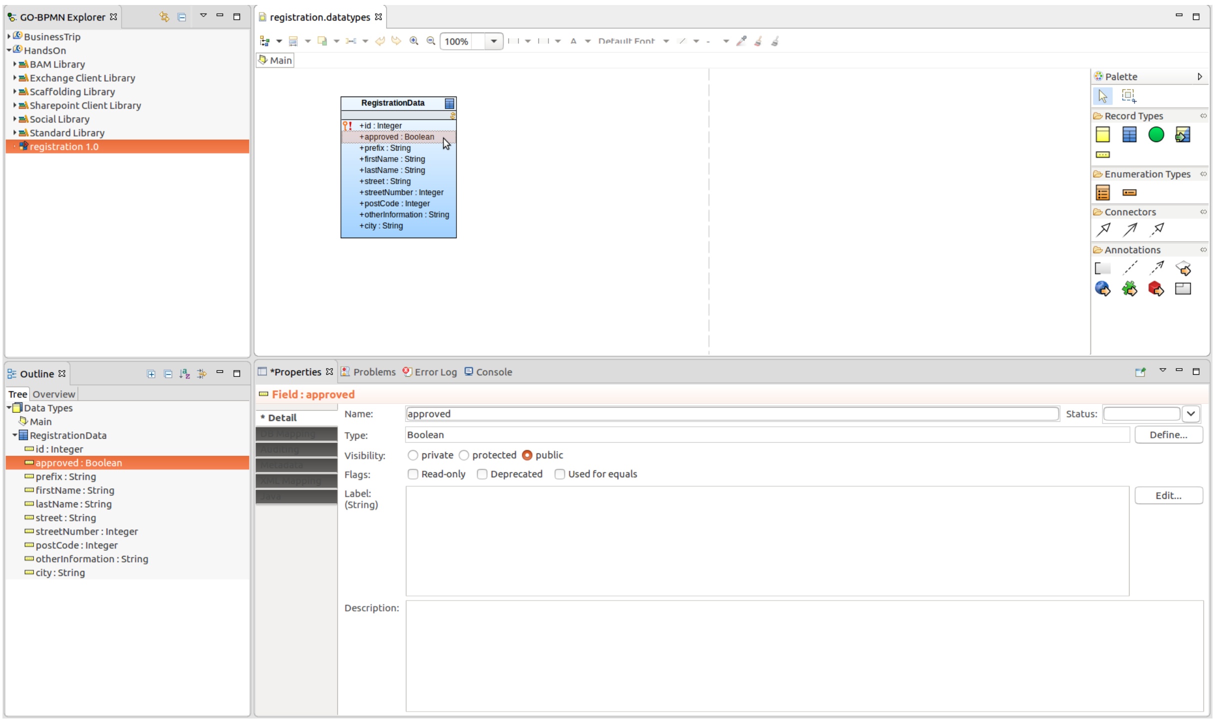
3.3 Data Model

Data models allow you to create data structures with internal structure for your business data, such as, your orders, clients, products, etc. For example, you could create an Order with its ID, date when it was placed, the client who placed it, its status. This all is then considered a single data structure called a **Record**. Records can then be designed to have relationships to other Records: The Order record could have a relationship to a Client record and the Client record could have relationship to an Account, etc.

You can find information on data types and on how to create them here.

Let's define the custom data type we will need to save individual registration applications:

1. Create the data type definition file that will hold the custom data types: right-click the module, go to New > Data Type Definition.
2. In the dialog, enter the name of the file, **registration.datatypes**
3. Click **Finish**: in the editor on the right, you will create the data structures. The elements you can use are in the palette on the right: hover the mouse over the icon in the palette to display information about the elements.
4. Create a shared Record: click the blue Shared Record icon in the palette and then click into the canvas. Instances of shared Record are automatically persisted in the mapped database entry and their changes are reflected in the mapped unlike common Records (yellow icons in the palette). Note that the Record was automatically created with the **id** field as its primary key that is automatically generated.
5. Name the shared Record to **RegistrationData**.
6. Add a Record Fields into the Record: select the Record and press Insert.
7. Set the types of individual Fields in their Properties view or directly in the record
   - **approved:** Boolean
   - **prefix:** String
   - **firstName:** String
   - **lastName:** String
   - **street:** String
   - **streetNumber**: String
   - **postCode**: Integer
   - **otherInformation**: String
   - **city**: String



RegistrationData
+id : Integer
+approved : Boolean
+prefix : String
+firstName : String
+lastName : String
+street : String
+streetNumber : Integer
+postCode : Integer
+otherInformation : String
+city : String

Data Types
Main
RegistrationData
id : Integer
approved : Boolean
prefix : String
firstName : String
lastName : String
street : String
streetNumber : Integer
postCode : Integer
otherInformation : String
city : String

Field : approved

Detail

Name: approved Status:
Type: Boolean Define...
Visibility: private protected [✓ public]
Flags: Read-only Deprecated Used for equals
Label: (String) Edit...
Description:

3.4 Organization Model

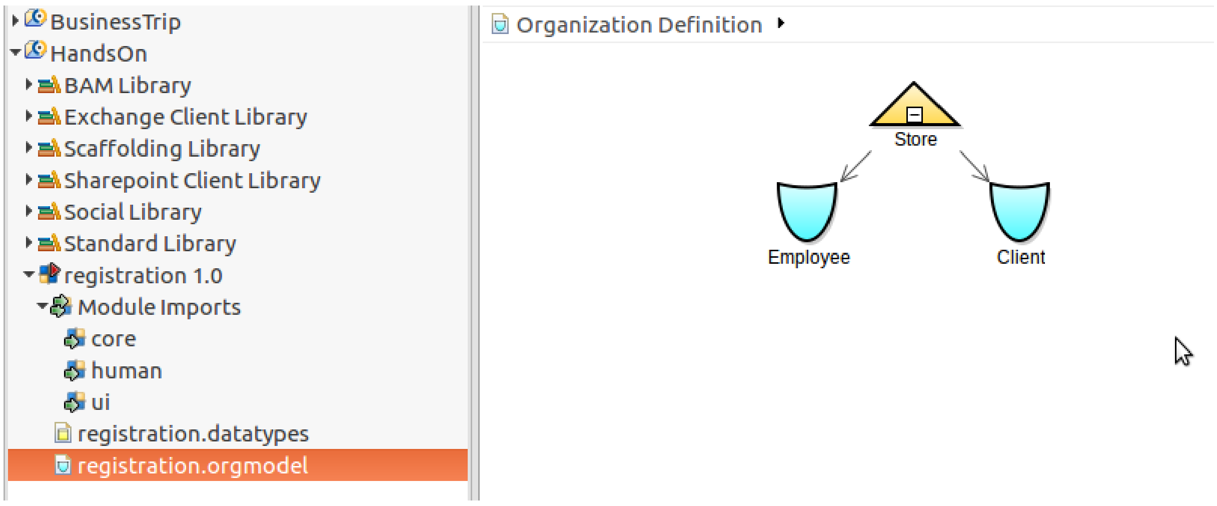
Organization models serve to stream work to certain groups of people or to group people for other purposes: for example, you want to send your health check results only to HR people. Such groups are primarily represented by Roles, for example, Admin, Engineer, Secretary, Doctor. You create relationships of such Role to Organization Units or other Roles.

Detailed information on how to create Organization Models and work with them is available `here`.

Let's define the organization model we will need. The elements in the model will represent the groups of people possibly involved in the registration process:

1. Create the organization definition file: **right-click the module**, go to **New** > **Organization Definition**. In the editor displayed on the right, you will create the organization structures: again, the elements you can use are in the palette on the right.
2. Create an organization unit and name it **Store.**
3. Decompose the unit into a role: click the quicklinker circle icon next to the unit and pull and release: select Role from the context menu.
4. Name the role **Employee**.
5. Decompose the unit into another role and name it **Client**.
6. Save the model.

The organization model you just created allows you to address both the client and the employee as the **Store** unit. Note that the way you create the organization model should reflect the way you need the users grouped and sorted for your business process: usually it will not reflect your real organization structure.



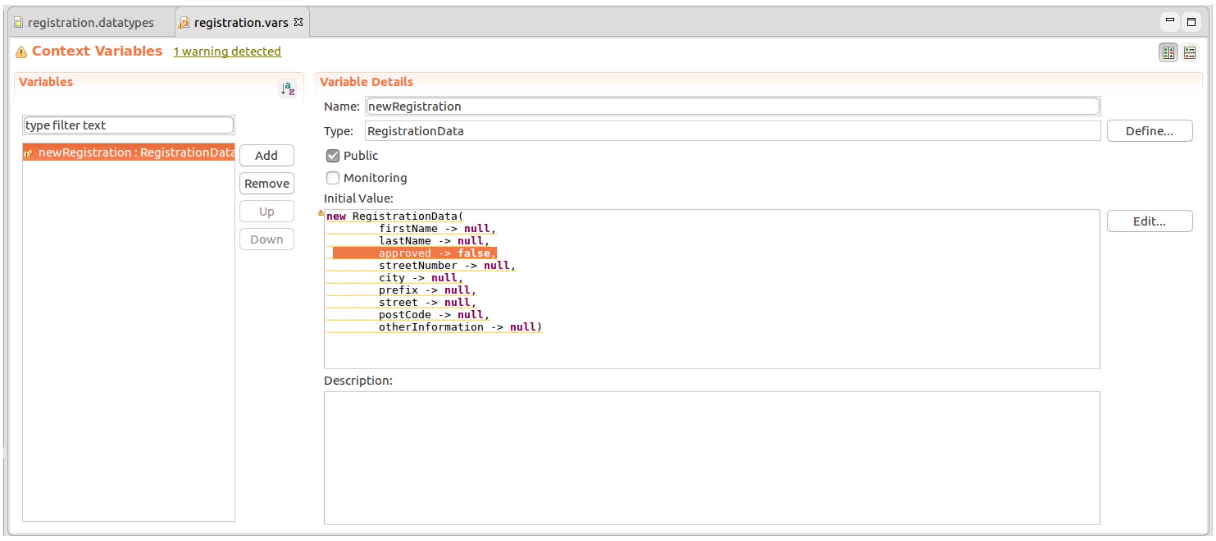
3.5 Variables

Variables serve to store values of a certain type, be it of a built-in type or your Record type. They exist in a particular context, such as, a process context, and might not be accessible from everywhere: to prevent issues with access, we will create a global variable to store registration data since global variables are accessible from the entire module.

Let's define a global variable for the registration data:

1. Create the variable definition file: **right-click the module**, go to **New** > **Variable Definition**.
2. In the editor displayed on the right, click **Add**.
3. In the Variable Details section, define the variable name as **newRegistration** and set its type to Registration Data.
4. Initialize the variable value with **approved** set to **false**.

The variable will be created at the moment the model will start: since it is an instance of a shared Record (the RegistrationData Record) it will create a record in the database. You will then modify its value via forms.



Name: newRegistration
Type: RegistrationData
Public [✓]
Monitoring [ ]
Initial Value:

```
new RegistrationData(
        firstName -> null,
        lastName -> null,
        approved -> false,
        streetNumber -> null,
        city -> null,
        prefix -> null,
        street -> null,
        postCode -> null,
        otherInformation -> null)
```

Description:

3.6 Process

Now we need to create the process that will get registration data from the user and meets the following requirements: It starts when the model is run.

- It displays to a client a form where they enter their data.
- It stores the data when the user submits the form.

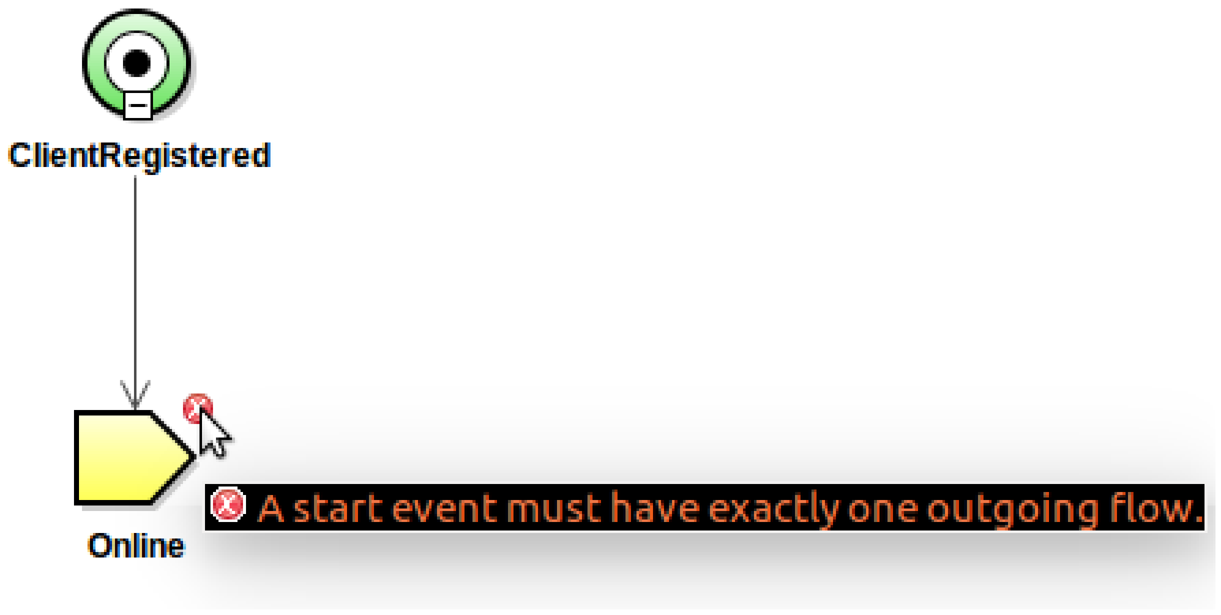
Now we will create the business process for registration: we will use the GO-BPMN process though you could use a BPMN process just as well. The process will navigate the respective users through the form we have created previously and save the data into the database on registration approval. Since we are using a GO-BPMN process, we need to identify our goal: the goal is to register a user. The way to achieve it, is to have the user enter the registration details and another user to approve the registration data.

Let's model the Goal process:

1. Create the process definition file:
   (a) Right-click the module, go to **New** > **Process Definition**.
   (b) In the dialog, enter the name of the process **registration**, select the **Goal-based process** as its type and select the Executable option.

   If you do not select the Executable option, the process will not be executed automatically: this option is useful if you are using a process in a Sub-Process.

2. Now design the Goal hierarchy:
   (a) Right-click into empty space of your canvas and select Achieve Goal in the context menu (this is equivalent to picking it from the palette).
   (b) Name it **ClientRegistered** so it reflects the target state.
   (c) Decompose the Achieve Goal to a Plan: grab the quicklinker and pull it to an empty spot; select Plan from the context menu.
   (d) Name the Plan **Online** since the user will register online; the Goal could be decomposed into another Plan called **ByEmail** for example.
   (e) Save your model: the Plan will be marked with a red error marker: this means that the Plan contains an error: hover over the marker to see what the problem is: in this case, the None Start Event, that is automatically inserted into a Plan and is a standard BPMN element, is missing an outgoing flow: Let's fix this.



A start event must have exactly one outgoing flow.

3. Create the body of the Plan with the actual process workflow that defines how to meet the Plan:

    (a) Click the Plan to open its plan body.

    (b) Drag-and-release the quicklinker of the StartEvent and in the context menu, select the **Task**, a standard BPMN element. You will be prompted to select the type of the task: in this case, the task should display a registration form: this is performed by the User task type. Therefore, select the **User** task.

    (c) Define the task properties in the Properties view:

    (d) Define the name of the task on the Details tab as **GetClientDataTask.**

    (e) You want to display the task only to users, who are clients so **performers** will be set to the Client role of your organization model. The content will be define by the **GetClientData** form, which you will create in the next step: Open the Parameters tab in the Properties view of the task and define the parameters:

    ```
    title /* String */ -> "GetClientData",
    performers /* Set */ -> {Client()},
    uiDefinition /* UIDefinition */ -> clientRegistration()
    ```

    (f) In the next step, we want to display another form to another user: drag-and-release the quicklinker of the **GetClientDataTask**. Name it **GetApproval** and set its parameters to

    ```
    title /* String */ -> "GetApproval",
    performers /* Set */ -> {Employee()},
    uiDefinition /* UIDefinition */ -> approval()
    ```

    (g) Since after the approval, the process should finish, attach the Simple End Event to the GetApproval task.

3.7 Forms

Now let's create what the front-end user should see and work with: the **clientRegistration** and **approval** forms. First the clientRegistration form:

1. Create the form definition file: right-click the module, go to New > Form Definition.

2. In the dialog, enter the name of the file *clientRegistration*.

3. Make sure not to select the **Use FormComponent-based UI**. This flag activates the experimental forms version, in which you can create forms in a more code-like way without event-driven approach.

4. Click **Finish**.

5. Design the form as depicted below: insert the respective component in the palette and then click into the respective position in the form.

6. For each Text Box and Text Area, do the following in their Properties view:

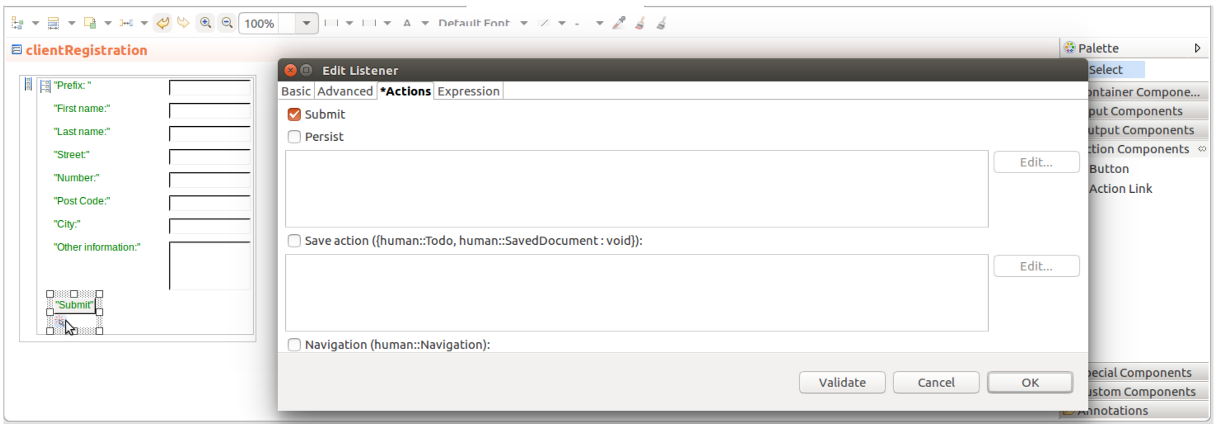
    (a) Insert its name in the Label field.

    (b) Set its Binding to the field of the newRegistration variable, for example, in the prefix field, set the binding to `&newRegistration.prefix`.

    The `&` operator indicates a reference to the given memory slot and the `.` operator allows you to navigate through the record structure. With the binding, the values will be stored in the parameter object.

    (c) On the Register button, define what should happen when the user clicks it: in the Properties view, go to the Event Handling tab and click the Add button in the Private Listeners section: this will create a listener object attached to the button: to make the listener listen to a click, it must be an ActionListener: this is default option so you do not need to change it.

    (d) Go the Actions tab and select the Submit option: this will make sure, that the data in the form will be saved and the form discarded when the user clicks the button.



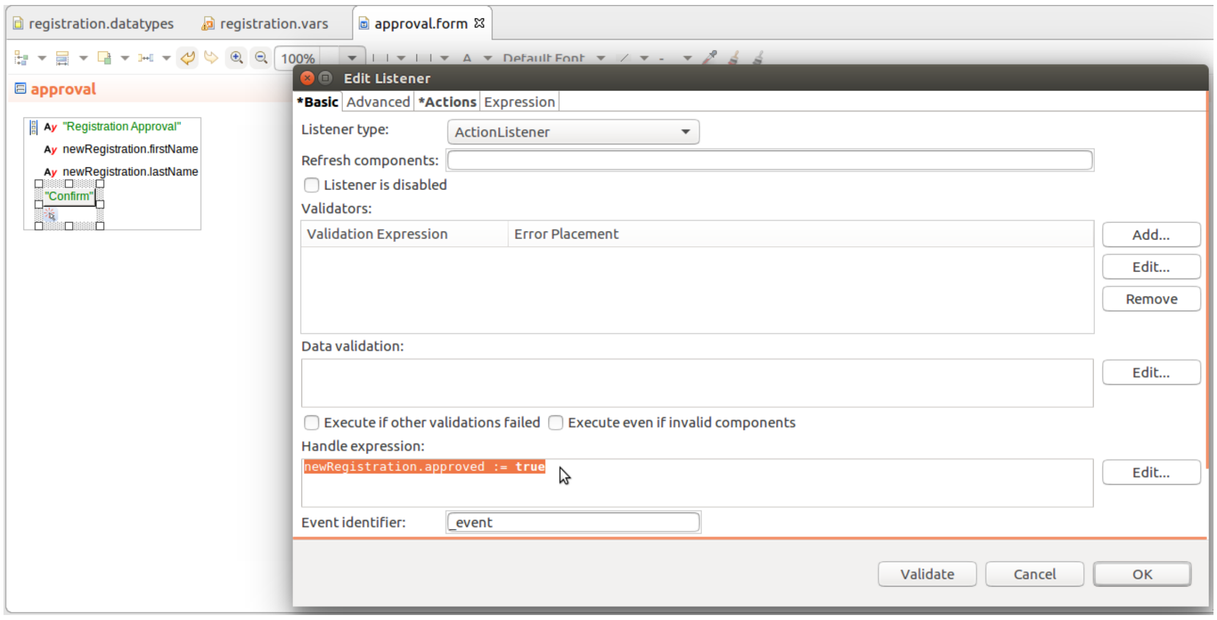
[✓] Submit

[ ] Persist

[ ] Save action ({human::Todo, human::SavedDocument : void}):

[ ] Navigation (human::Navigation):

Now the approval form: Proceed analogously. In addition, set the listener on the Approval button to set the approved field to true in its Handle Expression as depicted below.



Listener type: ActionListener

Refresh components:

[ ] Listener is disabled

Validators:

Data validation:

[ ] Execute if other validations failed [ ] Execute even if invalid components

Handle expression:

```
newRegistration.approved := true
```

Event identifier: _event

3.8 Running the Model

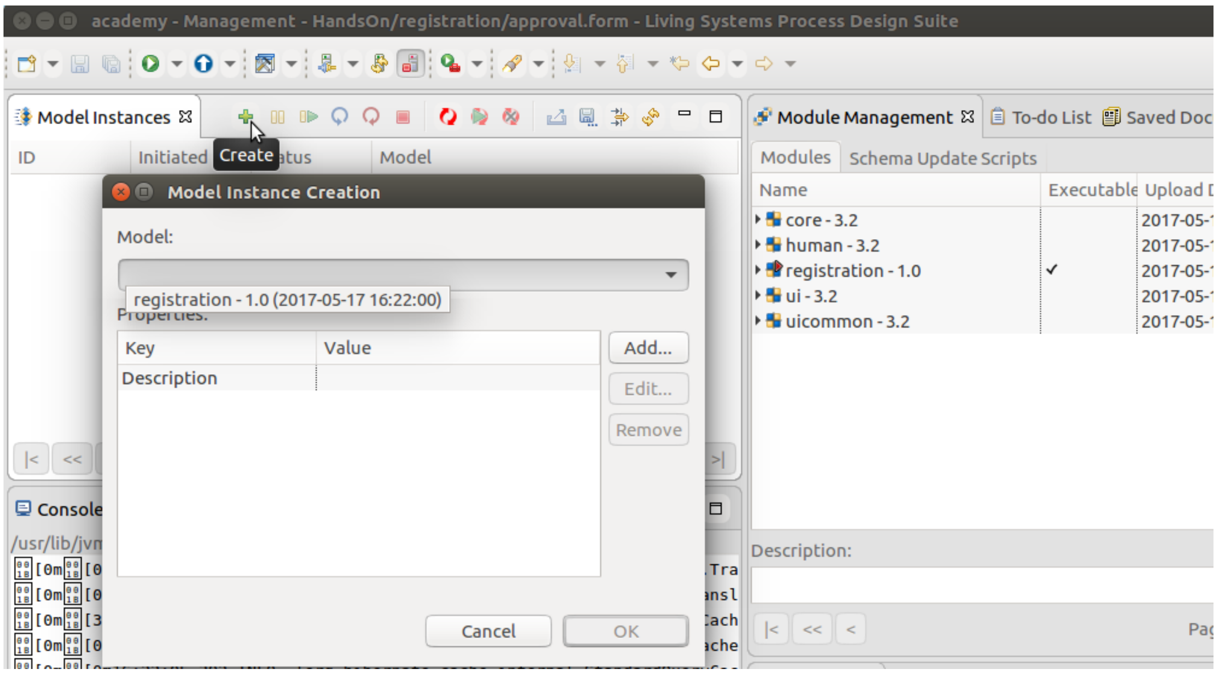
Now you will upload your model, create its instance, assign the Roles to users and follow the execution.

Before we can run the model, we need to set up a connection: 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.

To run and manage the model, do the following:

1. In the Modeling perspective, right-click the module and select *Upload As* > *Model*: now, your module is in the LSPS Server Module repository.
2. Switch to Management perspective: click the Management button in the upper-right corner of PDS.
3. In the Model Instances view, click the **Add** button and select the module.



Model Instance Creation

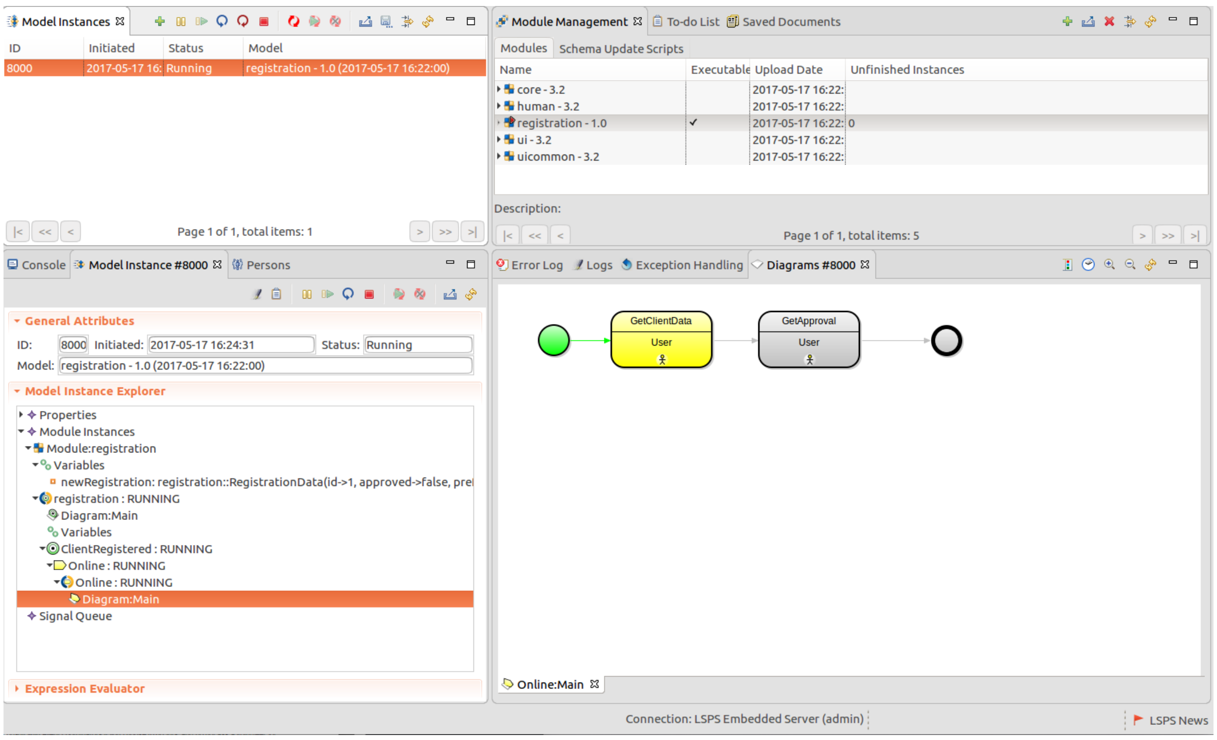
Model: registration - 1.0 (2017-05-17 16:22:00)

Properties:

Key	Value
Description	

Name	Executable	Upload D
core - 3.2		2017-05-
human - 3.2		2017-05-
registration - 1.0	✓	2017-05-
ui - 3.2		2017-05-
uicommon - 3.2		2017-05-

4. Click the **Refresh** button in the view: a new entry appears; this is your model instance.
5. Double-click the model instance: a detail view appears. In the model Instance Explorer, click the diagram of the process to display its status.



ID	Initiated	Status	Model
8000	2017-05-17 16:	Running	registration - 1.0 (2017-05-17 16:22:00)

Page 1 of 1, total items: 1

Name	Executable	Upload Date	Unfinished Instances
core - 3.2		2017-05-17 16:22:	
human - 3.2		2017-05-17 16:22:	
registration - 1.0	✓	2017-05-17 16:22:	0
ui - 3.2		2017-05-17 16:22:	
uicommon - 3.2		2017-05-17 16:22:	

Description:

Page 1 of 1, total items: 5

General Attributes

ID: 8000 Initiated: 2017-05-17 16:24:31 Status: Running

Model: registration - 1.0 (2017-05-17 16:22:00)

Model Instance Explorer

```
Properties
Module Instances
  Module:registration
    Variables
      newRegistration: registration::RegistrationData(id->1, approved->false, pre
    registration : RUNNING
      Diagram:Main
      Variables
      ClientRegistered : RUNNING
        Online : RUNNING
          Online : RUNNING
            Diagram:Main
Signal Queue
```

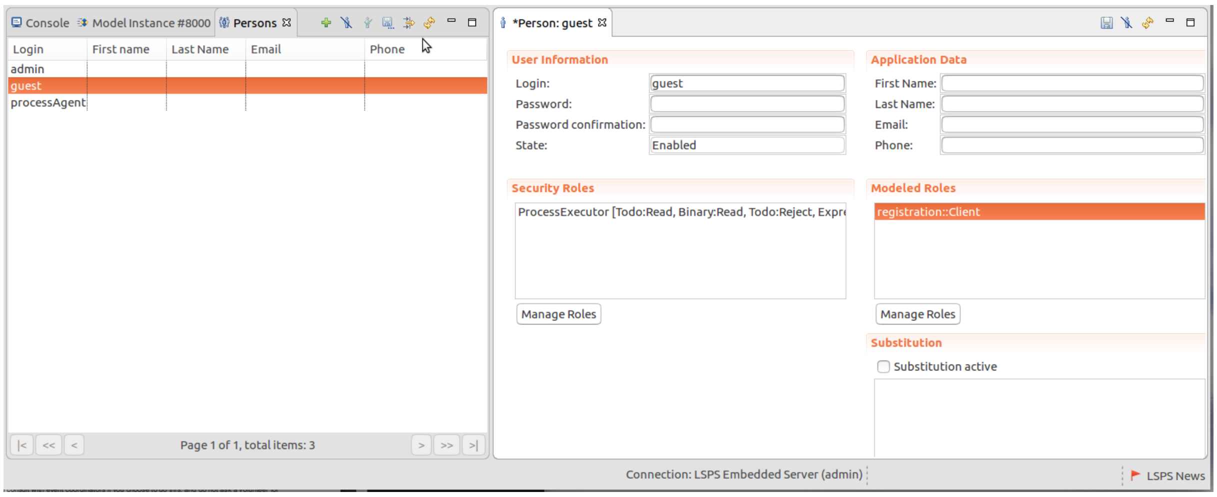
Connection: LSPS Embedded Server (admin)

The process instance is stuck on the first User task: the task generated a to-do for an end user with the role Client; however no such user exists:

Let's assign the role to the guest user:

1. Open the Persons view: Click the Management Views button in the toolbar and select Persons.

2. In the displayed view, double-click the guest entry.
3. In the displayed guest details, click the **Manage Roles** button *below the Modeled Roles* section: in the dialog box, select the Client role.
4. Click Save.



Login	First name	Last Name	Email	Phone
admin				
guest				
processAgent				

Page 1 of 1, total items: 3

Login: guest
Password:
Password confirmation:
State: Enabled

First Name:
Last Name:
Email:
Phone:

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

registration::Client

Connection: LSPS Embedded Server (admin)

5. Now add the Employee role to the admin user.

3.8.1 Working with the Application User Interface

Let's submit and approve the registration from the Application User Interface:

1. **Open the application** in your browser and log in as the *guest* user with the password *guest*.
2. **Click the To-Do list** on the left: a to-do that was produced by the GetRegistration User task should be available in your to-do list: provide the information and click Submit.

   You can now **go to PDS and in the Management perspective**, refresh the diagram of your process instance: it will be now "stuck" on the second User task.
3. Log out from the application and **log in as the admin user** with the password admin.
4. **Click the To-Do list** on the left: a to-do that was produced by the GetApproval User task should be available in your to-do list: provide the information and click Submit.

   Now you can **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.

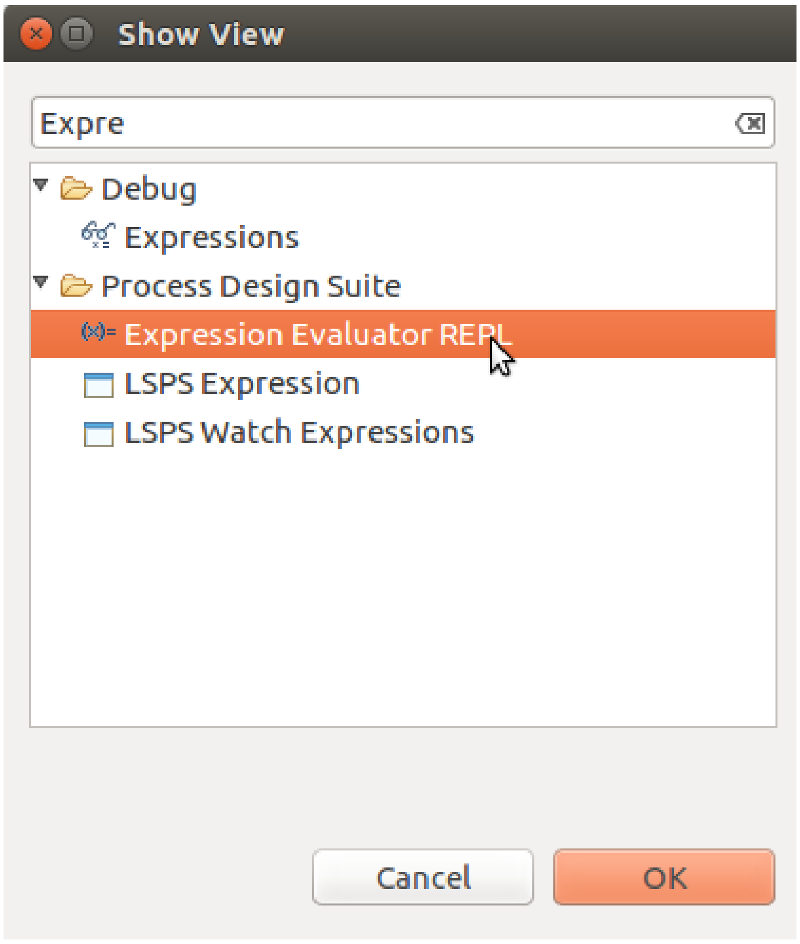
Chapter 4

Expressions

Properties, conditions, assignments, etc. are all defined as expressions of the LSPS Expression Language. Expressions use of operators, literals, keywords, and other constructs of the Expression Language, that end up returning a value of a type. Also literals are of a type, which does not change (unless cast but we will get to that later).

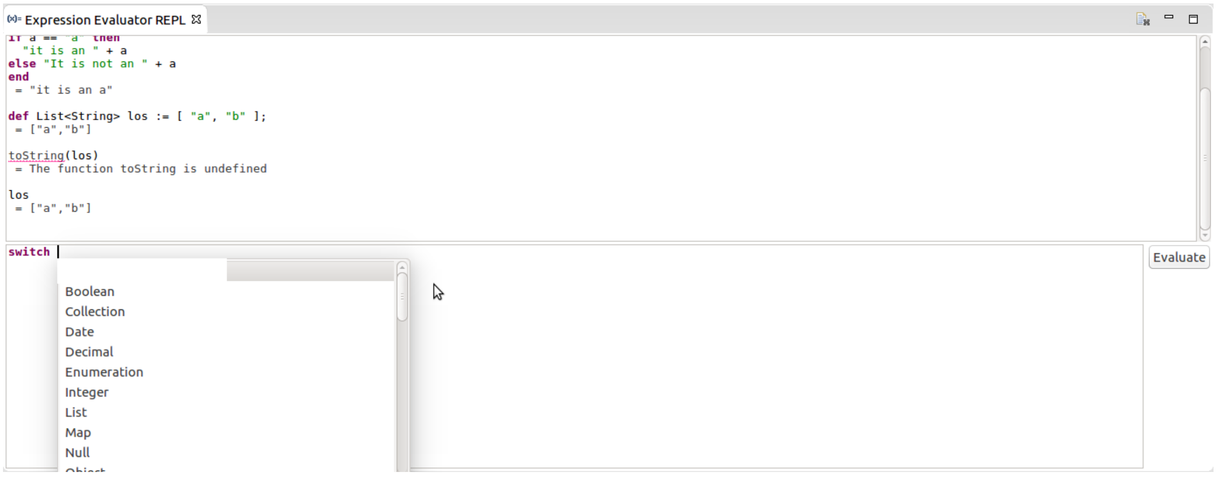
4.1 Evaluating Expressions

You can test expressions in the **Expression Evaluator REPL** view. By default, the view is not displayed: To display it, go to **Window** > **Show View** > **Other** and in the displayed dialog search for the view.



To test an expression and display its return value, enter the expression into the view and click **Evaluate** or press CTRL+Enter (Enter will make a new line in your expression). Note that you can list the history of expression by pressing the arrow-up key and display auto-completion options by pressing CTRL+Space.

> **Important:** The REPL view does not support calls to Standard Library resources, such as functions, data types, etc.



```
  "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:** When evaluating your expression in the Expression Evaluator REPL view, the expressions are evaluated in a "dummy context" outside of any model: under normal circumstance, on runtime, expressions are evaluated in a context of their model instance: so they can access data in their parent contexts. Contexts are represented by Instances of name-space elements: These elements include Modules, Processes, Plans, and sub-Processes. To fully understand when a context is created and what data it holds, refer to `namespaces and contexts`.

Let's try it out:

Enter the following into the Expression Evaluator REPL and try to predict the return value:

- First, let's try some arithmetics:
  - `1 + 1`
  - `1 + 1.12`
  - `1/1.1`
  - `1%1.1`
  - `3**3`
- Let's try some concatenation:
  - `"Hello, " + "world!"`
  - `"Hello, number " + 1`
  - `1 + "hello"` This results in an invalid expression since the + operator is interpreted as addition, not concatenation.
- 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`
  - `false and false`
  - `true or false`
  - `false or true`
  - `true or true`
  - `true xor false`
  - `false xor true`
  - `true xor true`
  - `true xor true`
  - `!true`
  - `!false`
- Now, chain multiple expressions:
  - `"This will not return!";"Hello, " + #10 + "world!"`
    Note that only the value of the last expression is returned.
- Create Collections with items of some data type:
  - Sets: `{1, 2, 3}` and `{1, 1, 2, 3}`
  - Lists: `[1, 1, 2, 3]`
    Sets cannot contain items of the same value while Lists can.
    Mind that Collections are immutable: when you decide to change a List you need to create it anew.
- `[ 1 -> "Sunday", 2 -> "Monday"]` is a **Map**. In this case its of the type Map<Integer, String>.
- Evaluate the closure `{x:Integer -> "Integer" + x}`

  The closure has an Integer input parameter called `x` in the closure contexts: the closure a String, hence it is of the type `{ Integer : String }`

We have been through some basic data types but there are other data types to explore and we will get gradually to all of them. Let's take a sneak peek and evaluate the following:

- `type(Integer)` returns the Integer type.
- `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 all have the value `null`.
- `&var` is a reference to a variable.

Note: We do not mention Records, which are complex data structures, and the related data types one of the reasons being that it is not possible to create a new Record type with the Expression Language since Records are modelled in diagrams similarly to processes.

4.2 Creating Local Variables

In expressions, you can create local variables using the syntax `def <TYPE> <VARIABLE_NAME>`.

Try this in REPL view:

```
def Boolean boolVar;
```

Though the Boolean variable exists now, it has no value: its value is `null` of the type `Null` as we mentioned above. To assign a value to a variable, use the assignment operator `:=`.

```
boolVar := true
```

Local variables cannot be accessed from outside of their scope, expression block. An expression block is the expression, a loop, an if block, and an explicit block between the keywords `begin` and `end`.

Evaluate the following:

- Create a block with a local variable:

  ```
  begin
    def Boolean boolVar := true
  end
  ```

- Call the variable:

  ```
  boolVar
  ```

If you need a variable that you can access from throughout the model or a particular namespace, use either global or local variables: however, these are part of modeling and we will deal with them in later courses on model resources.

4.2.1 Check Point

1. Create a local variable called *closureVar* of the 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 entire expression 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)
```

4.3 Functions

You can consider functions special types of closures: they take arguments of certain types and return a value of a certain type, plus they can be used whenever you would use a closure: They are sort of closure constants.

To call functions from an expression, use the syntax `<FUNCTION_CALL>(<PARAMETER_1>, <PARAMETER_2>)`.

However, it is not possible to create a new function: you can only call existing function.

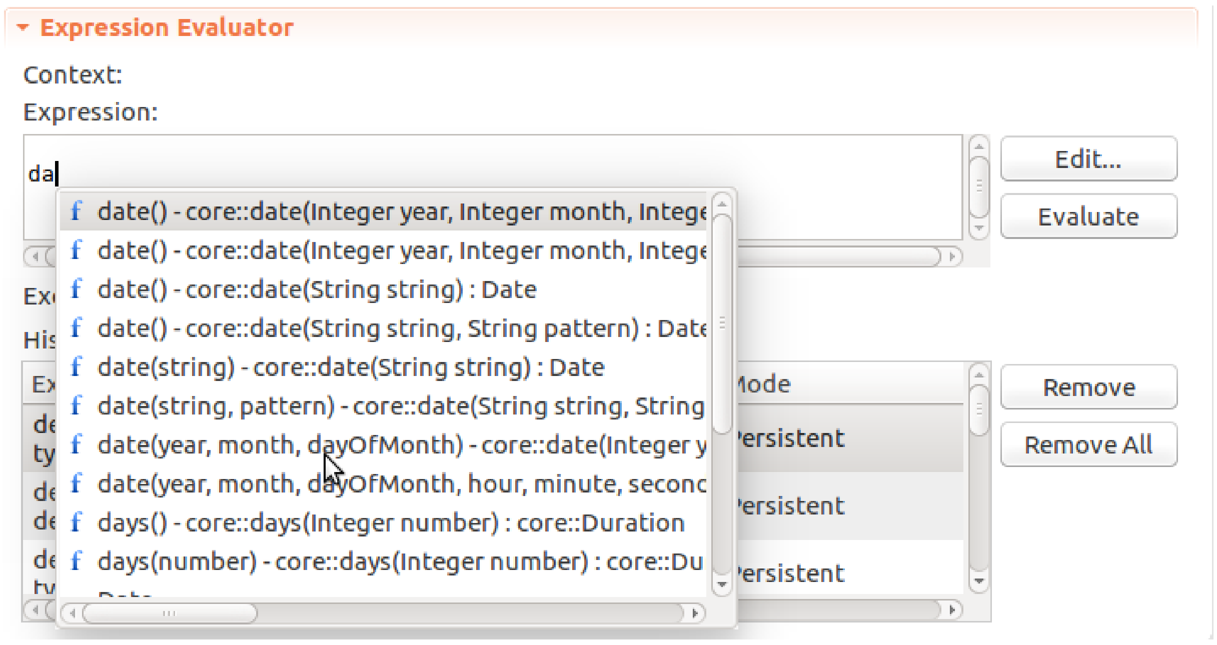
Note: We will create functions as part of model resources in another course.

Luckily, there are plenty of functions available in the Standard Library which is automatically available in your Module. And unluckily, you cannot call them from the Expression Evaluator REPL: you need to use the Expression Evaluator of a Model instance. Open the Expression Evaluator over a model instance:

1. Create a model and run it on the PDS Embedded Server. If you need detailed instructions, proceed as instructed in the `Quickstart guide`.
2. Switch to the Management perspective.
3. Refresh the Model Instances view and double-click the model instance to open its detail.
4. In the lower part of the detail view, expand the Expression Evaluator: here you can evaluate expressions in the contexts of the model instance. You can select a running context in the Model Instance Explorer above the Expression Evaluator. For this course, you do not need to select any context: the expressions are automatically evaluated in the context of the Model instance.
5. To make your life a bit easier when evaluating expression in the Expression Evaluator, make use of the following features:
   (a) Press Ctrl and arrow up and down to go throw the previously entered expression.
   (b) Press Ctrl and space to display auto-completion.
   (c) Press Ctrl and enter to evaluate the expression.
6. Evaluate the expression `"Hello!"`. This is simply a String literal, which surprisingly enough returns itself: `"Hello!"`
7. Let us now evaluate the expression `Hello!`. This is not a valid literal nor a variable name nor anything else that could be recognized as an expression: so what you have here is an invalid expression and the Evaluator tells you so.

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 easier: the `date()` function. In the Expression Evaluator, start typing `date` and press Ctrl + space to display the auto-completion dialog with the date functions. Click the one you like and define its parameters so it returns the correct date value.



Context:
Expression:

```
da
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
```

4.4 Type Hierarchy and Casting

Data types are arranged in a hierarchy which restricts relationships between data types. A type can be a subtype or a supertype of another type: If a type is a subtype of another type, it can be always used instead of the supertype, but not vice versa. Since the typing is so strict it prevents you from using incorrect types; for example, you cannot accidentally pass an Integer 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 is a subtype of the other 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 Expression Evaluator of your model instance, create an Object variable, for example, `def Object my↩ Object`, and assign it a value of any type. To check the type of value the object holds, run `typeOf(myObject)`. If you evaluate `def Object myObject := [1,2,3]; typeOf(myObject)`, the expression will return the type `List<Integer>`

Otherwise, the data type structure of the built-in data types is pretty flat: there are only two other relationships:

- 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 just as well.
- The Collection type and, the Set and List types: Collection is however abstract; it serves to allow you to decide whether something will be a Set or List later during execution. 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)`

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: In the Expression Evaluator, evaluate the following expressions:

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

4.5 Related Documentation

For details on features of the Expression Language, refer to the `Expression Language Guide`. As a reference, consider using the `Expression Language Reference Card`.

Chapter 5

Structure of GO-BPMN Process Models

A Process Model is a sum of an executable module and all its resources including any imported modules: in PDS, it is not represented explicitly since any executable module can become a model but also be imported into other modules, potentially also executable one.

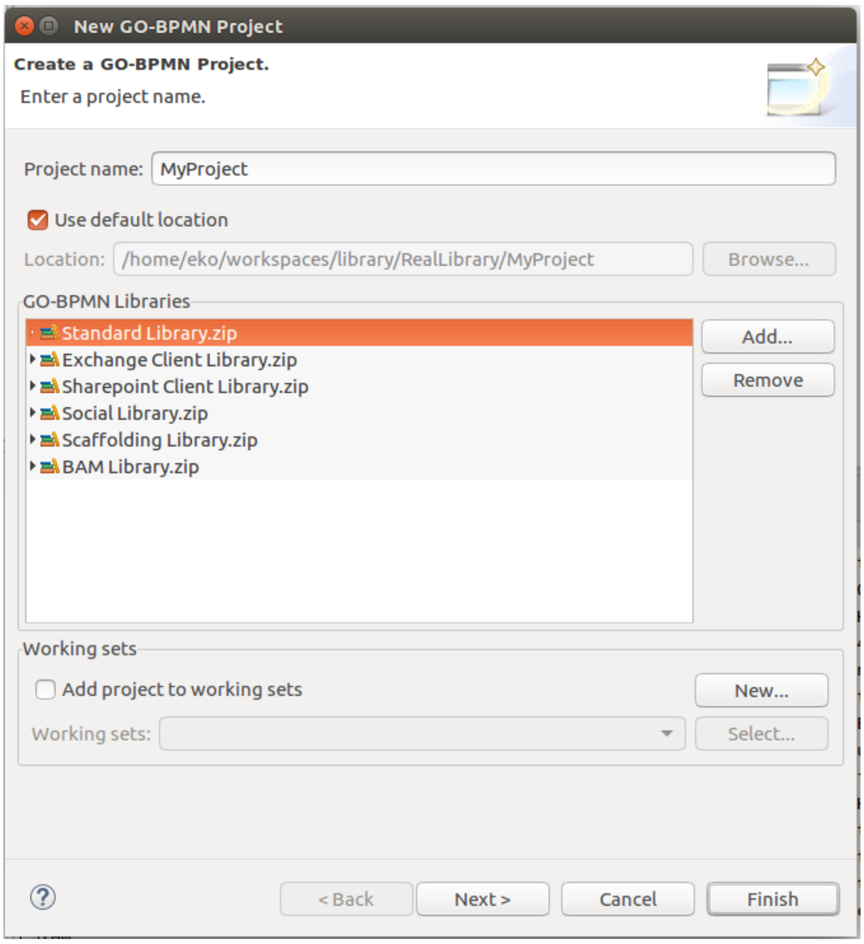
A GO-BPMN Project is by default created with the Standard Library and the modules import the library by default as well. On design time, modules can be created only in Projects. To further organize the content of projects and modules, you can create folders in Projects and Module: these are not uploaded to the server.

5.1 Project

A Project is a directory and serves only for organization of your sources: they are not influence semantics of your model and are not uploaded to the LSPS Server. If you decide to create other projects to organize your resource, by default, you will not be able to use resources from one project in another: to be able to access stuff from another Project, you need to explicitly enable the access by referencing it. We will get to referencing later. Let us create and examine a GO-BPMN project now.

5.2 Create a GO-BPMN Project

1. 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, there is the list of libraries that will be imported into the project. The only one that is required is the Standard Library: the other libraries provide additional resources you might not required.


New GO-BPMN Project

Create a GO-BPMN Project.

Enter a project name.

Project name: MyProject

☑ Use default location

Location: /home/eko/workspaces/library/RealLibrary/MyProject

GO-BPMN Libraries

- Standard Library.zip
- Exchange Client Library.zip
- Sharepoint Client Library.zip
- Social Library.zip
- Scaffolding Library.zip
- BAM Library.zip

Working sets

☐ Add project to working sets

Working sets:

2. Name the project and click **Finish**.

5.3 Module

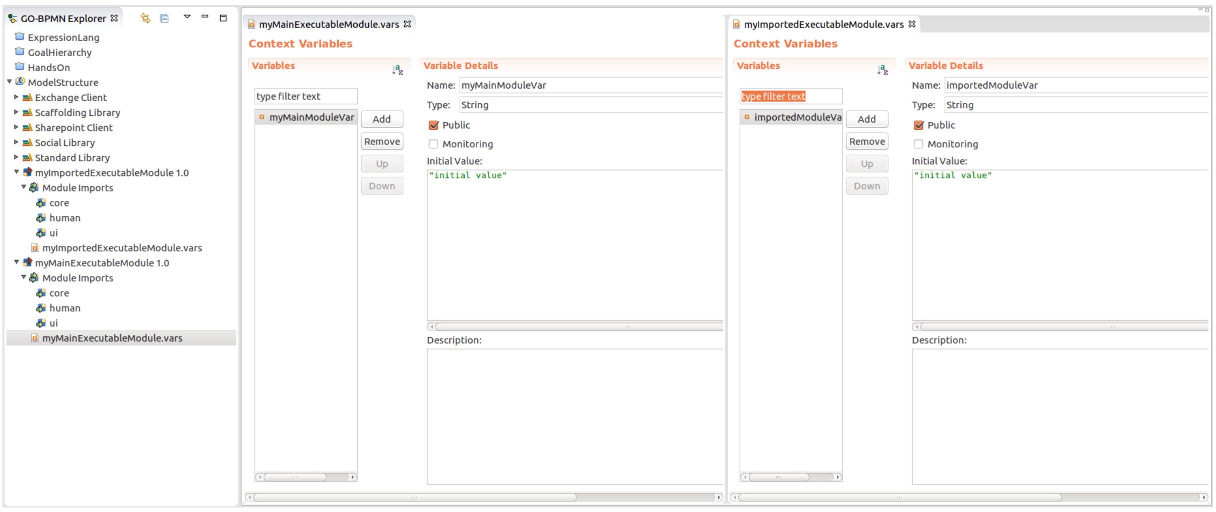
Modules are similar to java jar files. They follow a strict structure and have dependencies, called imports. A Module must be in a Project and all resources must be encapsulated in a Module. Note that what gets uploaded to the server is a Module: a Project exist merely on design time.

A Module holds most resources: processes, variables, custom data types, queries, functions, etc. Ideally it should be a reusable self-sustainable collection of resources. When you upload a module nothing much happens: the resources are simply uploaded to a repository. However, if the Module is executable, you can then instantiate it. Marking a module as executable is basically like 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. How so? You can import them into other Modules: when you run your executable module any other imported executable modules are run as well. Note that you can create folders in Projects and Module to organize your resources. Their behavior is the same as the Projects behavior: on upload, they are ignored.

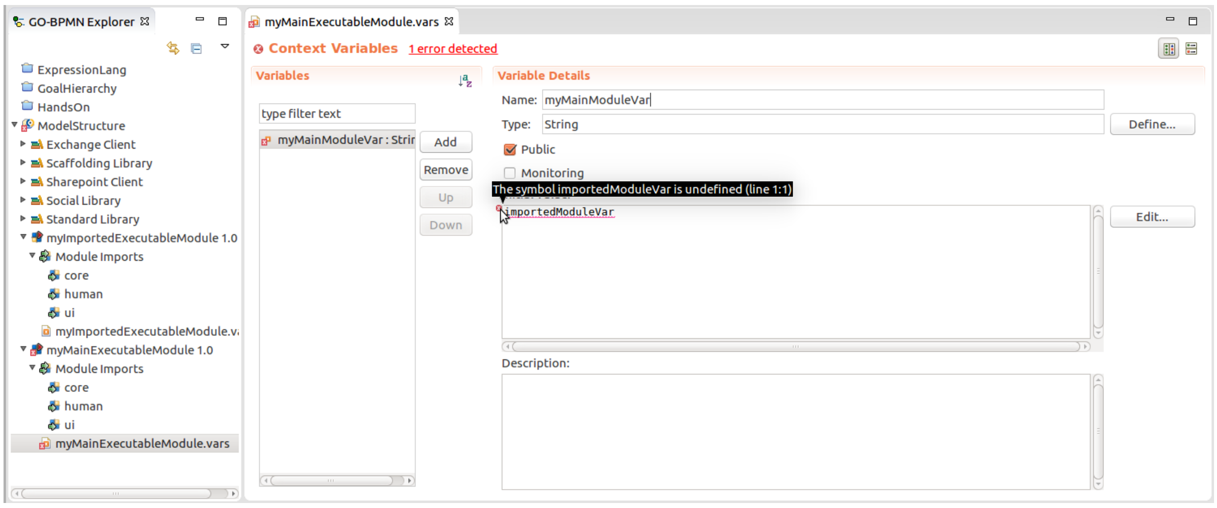
Let's create an executable module:

1. Now create a module in the project: right-click the project, go to New > GO-BPMN Module. Name it *my↩ MainExecutableModule*. Make sure to leave the executable flag selected.
2. Create another module in the project: Name it *myImportedExecutableModule*. Make sure to leave the executable flag selected.

3. Let's create in both modules some global variables: right-click the module name, go to **New** > **Variable Definition** and then click **OK**. In the open editor, click **Add** and on the right, define a variable. These will serve us to check what we can access in the modules.



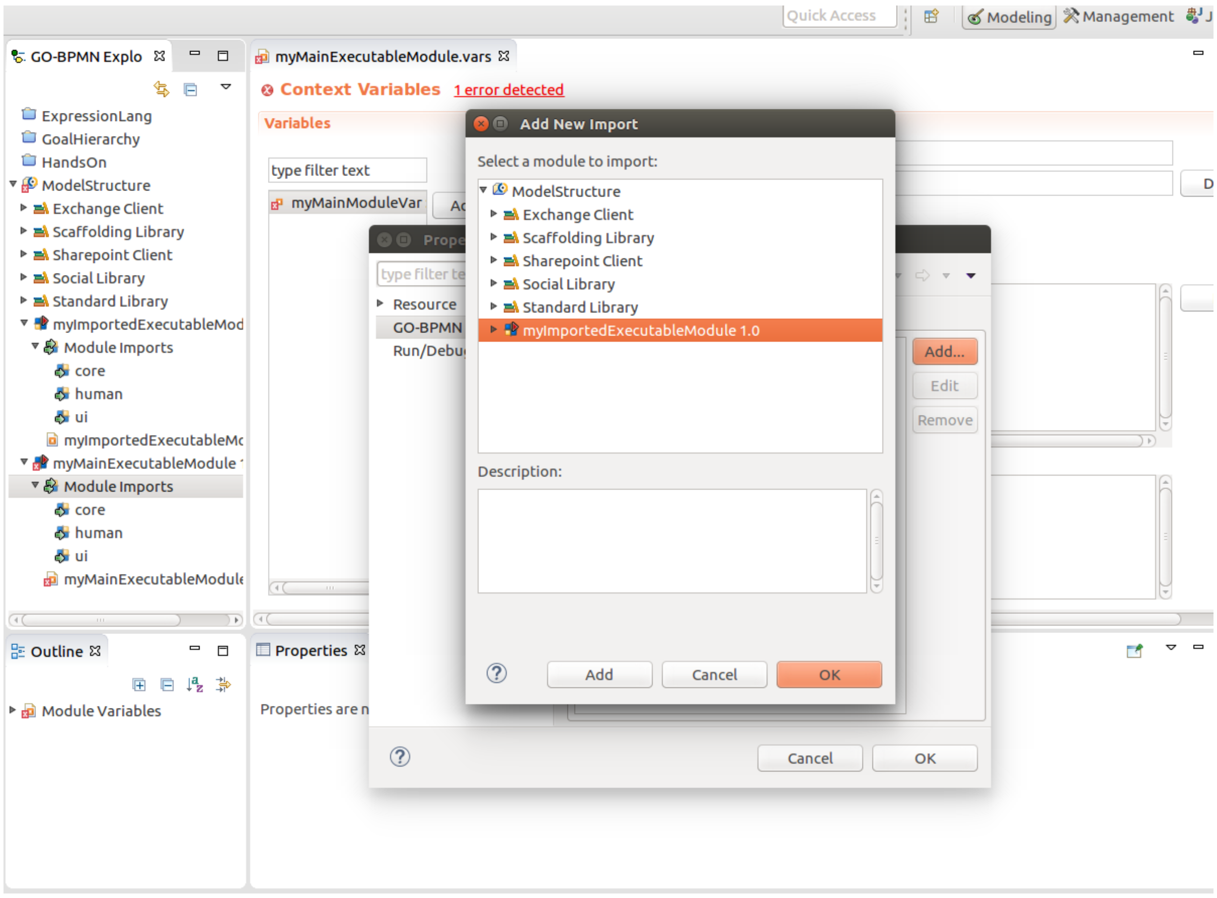
4. In a global variable in *myMainModule*, try to use the variable from the *importedModule*: you will end up with a validation error, that no such entity is available.



To solve this problem, import the importedModule into myMainExecutableModule.

5. In the GO-BPMN Explorer, double-click the Module Imports node in myMainExecutableModule.

6. In the dialog box, click Add and double-click myImportedModule.



7. Watch your reference problem go away: the resources of the importedModule are now available in the my↩ MainModule.
8. You can visualize module dependencies in the Module Dependency View: on the main menu, click **Window** > **Show View** > **Module Dependency View**.

5.4 Task: Create a Project

Create another GO-BPMN Project with a Module and import the myMainExecutable module into your new module: you will need to reference the project.

Chapter 6

Defining Data Structures for 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 are further data structures and they depend on each other. Such data structures are reflected as custom complex data types, called Records.

You create them in dedicated data type definitions in a visual editor which makes the process easy and intuitive: individual data structures, like invoices, items, people, etc. are represented by Records, which are similar to classes in OOP.

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`.

6.1 Creating Records

We will create a Record called *Invoice* and another Record called *BillTo*. To include data about the Invoice in the BillTo record, we will add an invoice field to the BillTo.

Let's do that:

1. First you need to create a *data type definition* in your Module. This is a file that will contain the data type model with the Records:

   Right-click the Module and go to **New** > **Data Type Definition**. Provide the name and click **Finish**.

   The name of the file is not important since the server does not care for file name: you can have your data model in multiple *data type definitions* and they will be still considered one data model: what "separates" your data models is a module (the namespace is defined by the name of the module), not files.

2. You can see the definition file in your Module in the GO-BPMN Explorer and it should open in the data-model editor: the white empty space of the canvas in the editor is actually a diagram, which is a visualization unit: it displays the content or part of the content of the data type definition. The actual content of the definition file is in the Outline view. We will return to diagrams later.

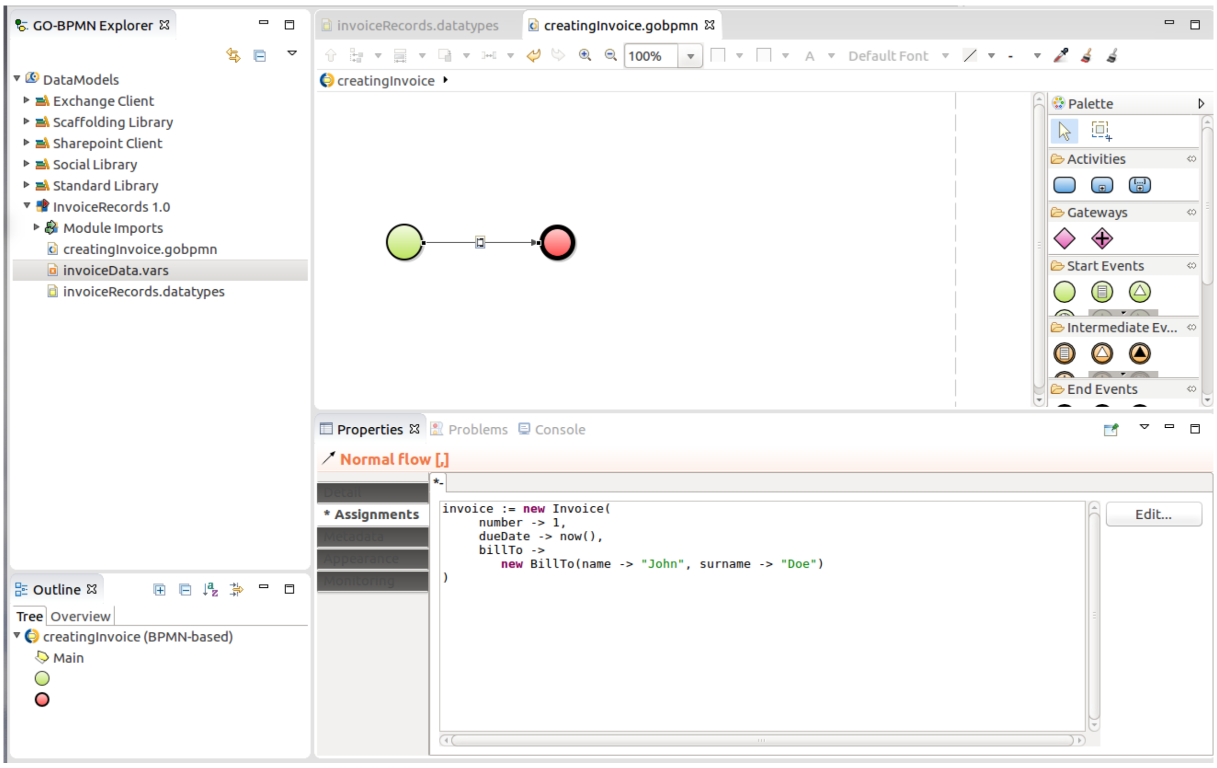
   In the diagram, create the Invoice Records with the fields *number* of the type Integer and *dueDate* of the type Date.

3. Create another record, the **BillTo** record, with the fields `name` and `surname` of the type String.

   To quicken the modeling process, insert a new Field with the **Insert** key and move to the field type with the tabulator key. You can copy and paste the Records as well as their Fields.

4. To add the bill-to data to the Invoice, create the *billTo* Field of the type **BillTo** in the Invoice.

   Now you can create instance of the Invoice record with the data about the invoice stored in its fields, for example, in a process on a Flow assignment: the expression will be executed when a token is passed from the Start Event to the Flow. We store the Record instance in the global variable `invoice`. The data of the global variable can be accessed from anywhere within the module with the expression `invoice.bill↩ To.name`.



```
invoice := new Invoice(
     number -> 1,
     dueDate -> now(),
     billTo ->
        new BillTo(name -> "John", surname -> "Doe")
)
```

6.2 Creating Relationships

You have probably spotted a problem in the data model: What if the same person places multiple orders? You will end up creating multiple invoices with the same Person data again and again. If the data of the person change, you will need to update them in every single Invoice instance. We can solve this by associating the two Records with a relationship: the Invoice record will be only *related* to a BillTo record and so multiple Invoices can be related to the same BillTo person:

1. Delete the billTo field.

2. Grab the quicklinker on the Invoice record and pull it to the BillTo record. Then define the properties of the relationship.

Note that we have named both relationship ends so we can navigate from the Invoice to BillTo and vice versa (that means, we can now write expressions like `myInvoiceInstance.billTo` and `myBillTo.invoices` to access the related record instance).

Also, we have set the multiplicity on the *invoices* end to Set: this allows us to associate multiple invoices with a single BillTo. Here is an example:

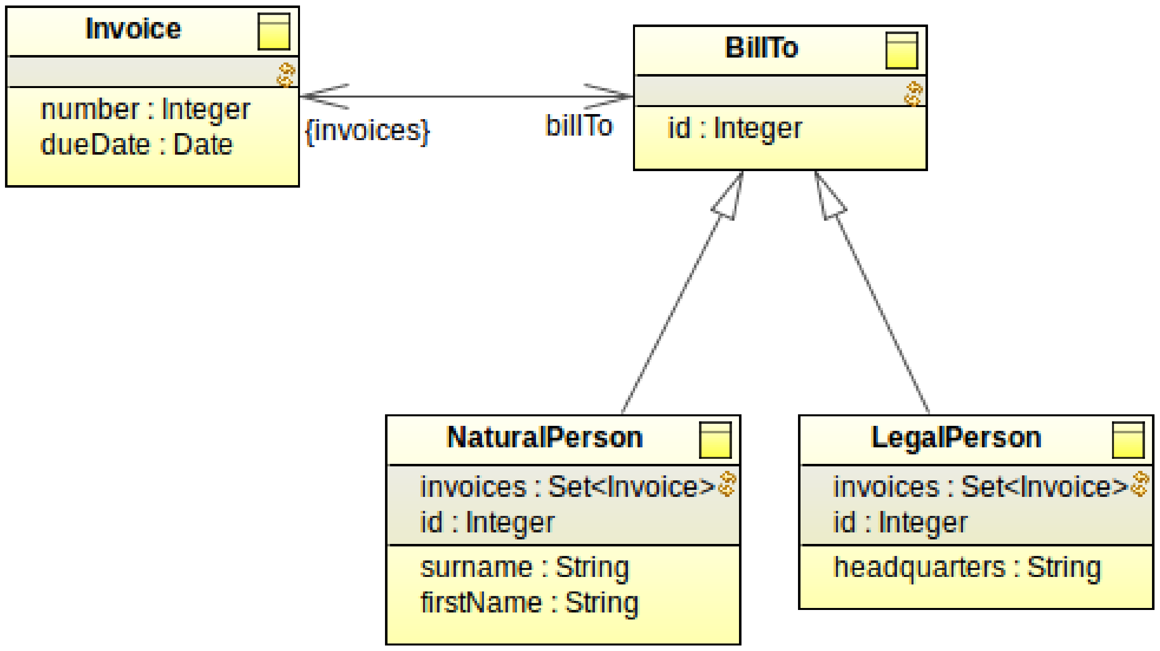
```
currentInvoice_1 := new Invoice(
    number > 1,
    dueDate > now()
);
currentInvoice_2 := new Invoice(
    number > 1,
    dueDate > now()
);
//john is a global variable of the type BillTo: john := new BillTo(
  name > "John",
  surname > "Doe",
  invoices > {currentInvoice_1, currentInvoice_2 }
)
```

6.3 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 us to distinguish the charged parties.

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 of the parent automatically.

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. 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}
)
```

Records can create hierarchies: For further details on inheritance, refer to the `official documentation`

6.4 Persisting Data

You now have a data type model with a little bit of complexity. But all the data you create gets trashed right after you model instance finishes. How can you persist the data so it remains available forever (or at least as long as your database is available)? Just flag the Records as shared: shared records are reflected in the underlying database instantaneously.

Let's persist the Records and take a look at what is going on in the database: **Mark all Record as Shared** in their properties.

We got a bunch of errors saying `Shared record <NAME> must specify at least one primary key` : since we are creating database tables for individual records, primary keys are required. Now we do not want to create the key by hand so set it to be generated automatically:

Note that the parent *BillTo* Record has 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 the PDS Embedded Server and upload the module.
2. Open a database client, such as SQuirreL, and check the schema of the Record tables.

   For the 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;LOCK↩ _TIMEOUT=60000`. Both the name and password are `lsps`.

You will see that the tables for individual Records reflect the model: BillTo table contains only the id column; NaturalPerson contains 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/create.
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:

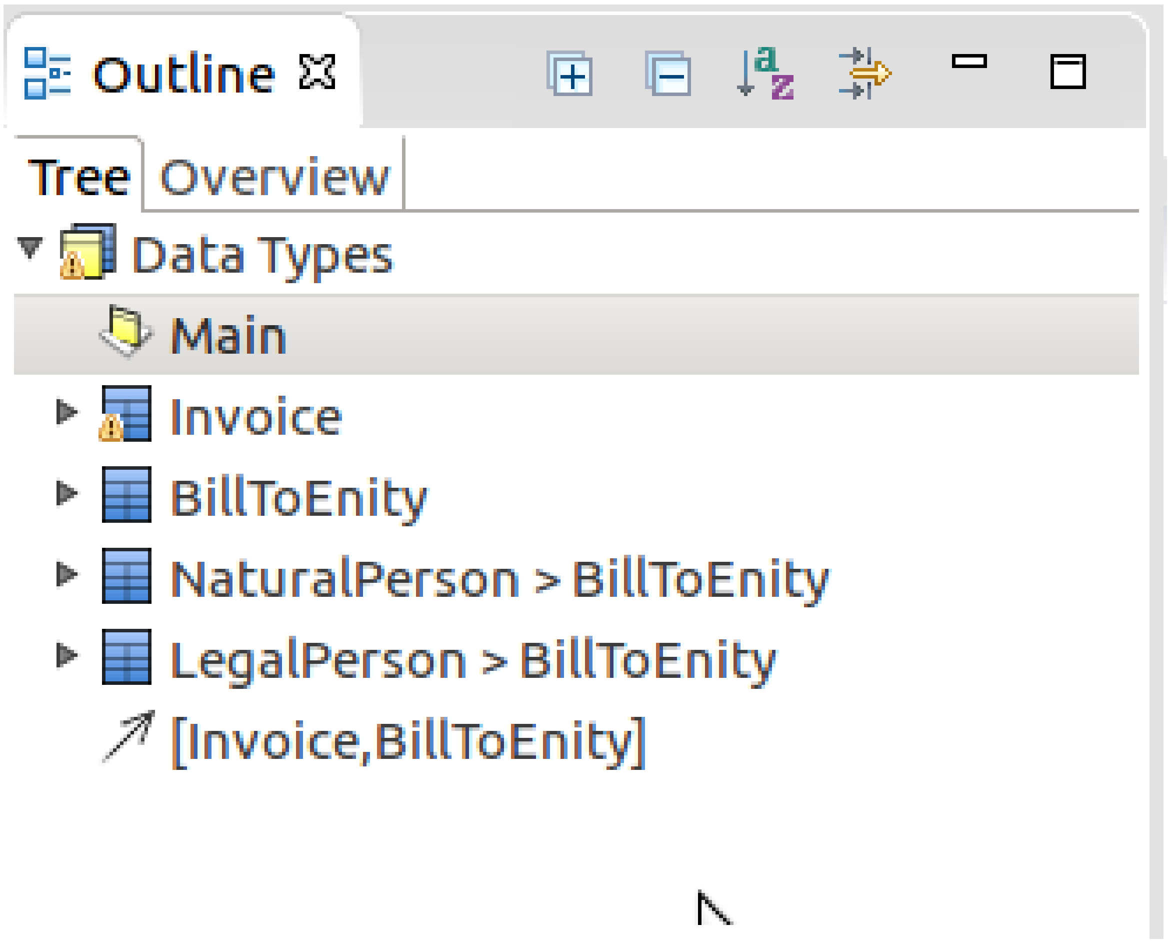
The **BILL_TO** and INVOICE table now contain the entire hierarchy schema along with the TYPE_ID column which hold the record type of the entity, such as invoice-types' NaturalPerson: there are no tables for the *NaturalPerson* or *LegalPerson* records.

6.4.1 Organizing Records with Multiple Diagrams

Until now we have created all elements of our data type model in the default **Main** Diagram. When we inserted an element into the canvas of the diagram, the element was displayed in the diagram and created in the definition file.

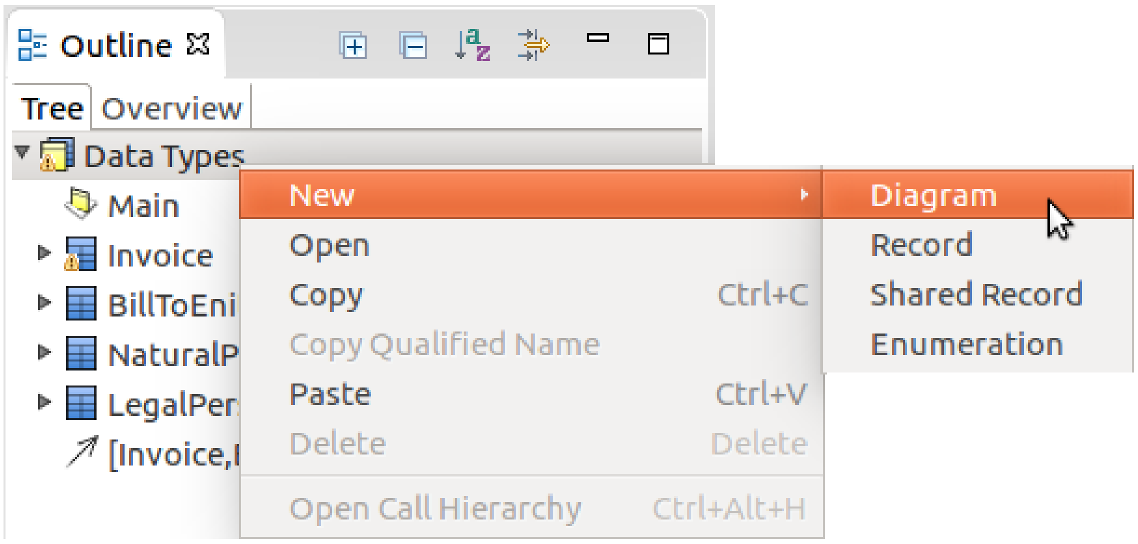
Note that diagrams do not represent a namespace: they are merely a presentation tool. If you create two Records with the same name in two diagrams, this will result in a name clash.

The content of the definition file is displayed in the *Outline* view.



Create another diagram in your 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. You could rename the Main diagram to Invoicing so it is easier to work with your diagrams.
3. There is a good chance that you might want to create a relationship between items in Inventory and invoices: since we want to have the inventory-related Records in the new diagram, you want to display that Record also in this diagram: drag-and-drop the Record from the Outline view onto the canvas.

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

6.5 Using Records from Other Definitions

To display element from another data type definition file, possibly from a definition file in an imported Module, use the Record Import element, which is available in the palette of the editor.

Chapter 7

Customizing the LSPS Application

In the previous chapters, you uploaded your modules to the PDS Embedded Server. This is an application server with the LSPS Application already deployed. The server runs on your computer locally and its LSPS Application cannot be modified.

In real-world scenarios, such a solution is obviously not sufficient since it is not possible to cstomize the LSPS Application User Interface: You cannot change its look or implement custom business logic.

To perform such customization, you need to modify the EAR of the LSPS Application and deploy it to a supported application server with the required setting (to set up such an application server, refer to the `deployment instructions`).

> **Important:** Before you start customizing the Application User Interface, analyze your business processes thoroughly: extract the knowledge into a systematic description with happy flows and *all border cases*.

7.1 Generating the LSPS Application

To generate the maven archetype of the LSPS Application and related resources, you need `LSPS Enterprise Edition with SDK installed`: from a PDS with SDK, you can `generate your application`. The maven archetype of the application along with the SDK Embedded Server, and launch and build configurations will be created and added to your workspace. Note that LSPS Application is a standard enterprise application: To deploy it on a supported application server, all you need to do is to build the application's sole EAR and deploy it to a supported application server as described `here`.

Using the generated resources, you can do the following:

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

7.2 Changing the Theme of the Application User Interface

Create a theme derived from the lsps-valo theme for your Application User Interface. You will need to:

1. `Set up the Sass Compiler`.
2. Copy the lsps-valo directory in the `<YOUR_APP>-vaadin-war/src/main/webapp/VAADI↩ N/themes/`.
3. Register your scheme with the `com.eko.ekoapp.vaadin.util.Constants` class and set it as D↩ EFAULT.
4. Rebuild the application and restart the server.
5. Change the `styles.scss` theme, recompile it with the Sass Compiler run and check the changes in your browser.

7.3 Remove Items from the Navigation Menu

Typically, you do not want the end user to have access to all their documents and to-dos: remove and add navigation items from the menu as instructed `here`. You can do much more with your Application User Interface: basic procedures are described in the SDK guide.

Chapter 8

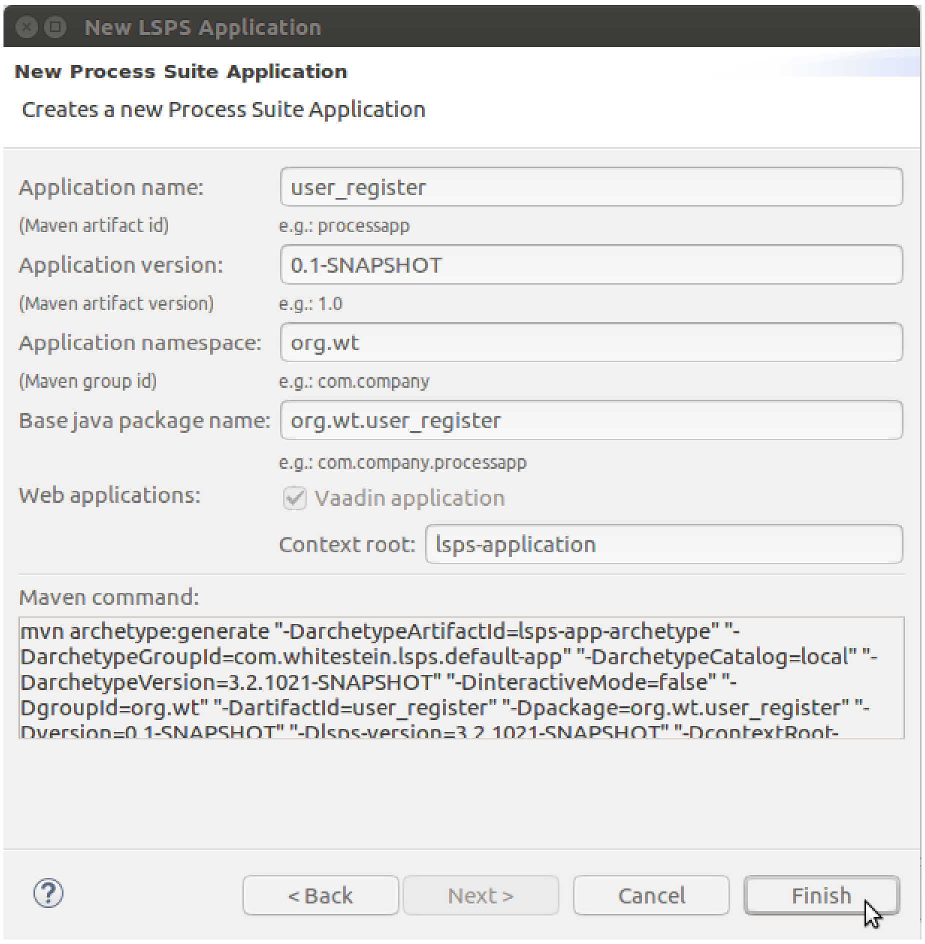
Accessing Data from other Data Sources

To access data from external resources, we will set up connection to the datasource, create an Entity and manage it via an EJB. This will be typically helpful when you have an existing database or your database is populated by an external system, and you want to obtain and manipulate the data from the code of your LSPS Application.

8.1 Prerequisites

Make sure you have the following ready:

- You have set up a database or you have access to a database. It is a good idea to connect to the database with an SQL client.
- You have generated a LSPS Application with the related resources in PDS.



New Process Suite Application

Creates a new Process Suite Application

Application name: (Maven artifact id)	user_register (e.g.: processapp)	
Application version: (Maven artifact version)	0.1-SNAPSHOT (e.g.: 1.0)	
Application namespace: (Maven group id)	org.wt (e.g.: com.company)	
Base java package name:	org.wt.user_register (e.g.: com.company.processapp)	
Web applications:	✓ Vaadin application	
	Context root:	lsps-application

Maven command:

```
mvn archetype:generate "-DarchetypeArtifactId=lsps-app-archetype" "-
DarchetypeGroupId=com.whitestein.lsps.default-app" "-DarchetypeCatalog=local" "-
DarchetypeVersion=3.2.1021-SNAPSHOT" "-DinteractiveMode=false" "-
DgroupId=org.wt" "-DartifactId=user_register" "-Dpackage=org.wt.user_register" "-
Dversion=0.1-SNAPSHOT" "-Dlsps-version=3.2.1021-SNAPSHOT" "-DcontextRoot-
```

8.2 Configuring Data Source

To configure a data source, so it is accessible from the SDK Embedded Server, and you can use its entities do the following:

1. In `<YOUR_APP>-embedded/conf/conf/openejb.xml`, define the data source.

   For example:

   ```
   ...
   JdbcUrl jdbc:h2:tcp://localhost/h2/h2;MVCC=TRUE;LOCK_TIMEOUT=60000
   Username lsps
   Password lsps
   </Resource>\<!-- adding this Resource tag:-->
   <Resource id="jdbc/USERS_DS" type="javax.sql.DataSource">
   JdbcDriver com.mysql.cj.jdbc.Driver
   JdbcUrl jdbc:mysql://localhost:3306/training_users;
   Username root
   Password root </Resource>
   ```

2. Restart the SDK Embedded Server.

8.3 Creating the Entity

1. Create a new package in the ejb project with the entity class.

   ```
   @Entity
   @Table(name = "ORDERS_USER")
   public class User {
     @Id
     private Integer id;
     @Column(name = "FIRST_NAME")
     private String firstName;
     public Integer getId() { return id; }
     public String getFirstName() {
     return firstName;
     }
   }
   ```

2. Create `<YOUR_AP>-ejb/src/main/resources/resources/META-INF/persistence.↩ xml` and define the persistence unit with the external data source.

   ```
   <?xml version="1.0" encoding="UTF-8"?>
     <persistence xmlns="http://java.sun.com/xml/ns/persistence"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com
                  version="2.0">
       <persistence-unit name="<UNIT_NAME>" transaction-type="JTA">
       <provider>org.hibernate.ejb.HibernatePersistence</provider>
       <jta-data-source><DATASOURCE_ID></jta-data-source>
       <mapping-file>META-INF/<PROJECT_NAME>-entities.xml</mapping-file>
       <validation-mode>NONE</validation-mode>
       <properties>
         <property name="hibernate.cache.region.factory_class" value="org.hibernate.cache.ehcach
         <property name="net.sf.ehcache.configurationResourceName" value="META-INF/lsps-ehcache.
         <!-- JBoss specific parameters -->
         <property name="jboss.as.jpa.providerModule" value="application" />
         <property name="jboss.as.jpa.adapterClass" value="com.whitestein.lsps.common.hibernate.
       </properties>
     </persistence-unit>
   </persistence>
   ```

3. Create the mapping file for the persistence unit.

```
<?xml version="1.0" encoding="UTF-8" ?>
  <entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
                   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm http://java
                  version="2.0">
  <entity class="org.eko.orderusersapp.entity.User" />
</entity-mappings>
```

4. Create the ehcache configuration file for the persistence unit.

```
<?xml version="1.0" encoding="UTF-8"?>
  <ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
           name="<UNIT_NAME>" updateCheck="false" monitoring="off" dynamicConfig="false">
  <cacheManagerPeerProviderFactory class="com.whitestein.lsps.common.ehcache.JmsCacheManagerP
  <defaultCache eternal="true" maxElementsInMemory="0" overflowToDisk="false" >
    <cacheEventListenerFactory class="com.whitestein.lsps.common.ehcache.JmsCacheReplicatorFa
  </defaultCache>
  <cache name="org.hibernate.cache.internal.StandardQueryCache" maxBytesLocalHeap="10000000"
    <cacheEventListenerFactory class="com.whitestein.lsps.common.ehcache.JmsCacheReplicatorFa
  </cache>
  <cache name="org.hibernate.cache.spi.UpdateTimestampsCache" maxElementsInMemory="1000" eter
    <cacheEventListenerFactory class="com.whitestein.lsps.common.ehcache.JmsCacheReplicatorFa
  </cache>
</ehcache>
```

8.4 Registering EJB

If you want to use the entity via an EJB in LSPS modules, do the following:

1. Create the bean class with an entity manager.

```
@Stateless
@PermitAll
@Interceptors({ LspsFunctionInterceptor.class })
public class UserBean {
    @PersistenceContext(unitName = "user-unit")
    private EntityManager em;
    public String getUsers(ExecutionContext context) {
        User user = em.find(User.class, 1);
        System.out.println(user.getFirstName());
        return user.getFirstName();
    }
}
```

2. Register the EJB in the ComponentServiceBean class:

```
@EJB
private UserBean userBean;
@Override
protected void registerCustomComponents() {
    register(userBean, UserBean.class);
}
```

8.5 Calling EJB Method from LSPS

Create a function definition file with a function that will use the keyword **native** to call the EJB method.