WHITESTEIN
Technologies

Living Systems® Process Suite

Software Development Kit

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

Contents

1 LSPS Application **1**

2 Setting Up **3**

3 Model Instance **5**

3.1 Transactions in Model Instances . . . 6

3.2 Execution Levels . . . 9

4 Customizing Application User Interface **11**

4.1 Customizing Themes . . . 11

4.1.1 Creating Themes . . . 12

4.1.1.1 Setting up the Sass Compiler for a Theme . . . 12

4.1.2 Setting the Default Theme . . . 13

4.2 Customizing Content and Layout . . . 14

4.2.1 Implementing LspsUI . . . 14

4.2.2 Adding Item to the Navigation Menu . . . 15

4.2.3 Adding Header and Footer . . . 15

4.2.4 Adding a Locale . . . 16

5 Custom Objects **17**

5.1 Registering EJBs of Custom Objects . . . 17

5.2 Custom Functions . . . 18

5.2.1 Creating a Custom Function . . . 18

5.3 Custom Task Type . . . 20

5.3.1 Implementing a Custom Task Type . . . 20

5.3.2 Generating Code for Handling Task Types . . . 21

5.4 Custom Form Components . . . 22

5.4.1 Creating Custom Component with Java Implementation . . . 22

5.4.2 Creating Custom Component with Expression Language Implementation . . . 26

6 Creating a Record Instance **29**

6.1 Working with Data Constraints 32

7 Persisting Data using Records **33**

7.1 Customizing Entity Auditing 33

7.1.1 Adding a Field to the Revision Entity 34

7.1.2 Adding a Related Record to the Revision Entity 34

7.1.3 Example Implementation of a Custom Revision Listener 36

8 Model JUnit Tests **39**

8.1 Prerequisites 39

8.2 Creating JUnit Tests 40

8.3 Running JUnit Tests 41

9 Integration **43**

9.1 Mail Server Configuration of the SDK Embedded Server 43

9.2 LDAP 43

10 Building the LSPS Application **45**

10.1 Building and Running the LSPS Application for Development Purposes 45

10.2 Building the LSPS Application for Deployment 46

11 Performance Tuning **47**

11.1 Pre-Loading Modules 47

11.2 Setting Dumping of Model Instances with Exceptions 47

11.3 Decreasing Frequency of Goal-Condition Checks 48

11.4 Improving Shared Record Search 48

11.5 Disabling System Cache Regions 48

11.6 Disabling Hibernate Statistics 48

12 Appendix: Creating Custom Form Component with Custom Events **49**

Chapter 1

LSPS Application

The LSPS Application is a JEE application that comprises the front-end and back-end server application of the LSPS Suite.

You can generate the LSPS Application and adapt it to build your customized business solution as follows:

- Implement custom logic including custom functions, form components, and task types as part of the *Application Business Logic* using the LSPS Server API.
- Adapt the layout, appearance, and content of the *Application User Interface* as required by your enterprise.

The architecture of the LSPS Application is as follows:

- The LSPS Server in the EJB container comprises the following:
  - **Execution Engine**: Engine that interprets models
  - **Module Repository**: Repository with compiled modules
  - **Runtime Data**: Runtime data of model instances
  - **User Management**: Service that manages users, roles, and rights stored in database
  - **BIRT and BAM services**: Services related to BIRT and BAM
  - **Web Services**: Services and data related to web services
- The web container part comprises the following:
  - **Application User Interface**: Thin front-end client that allows the user to interact with the execution via to-dos and documents
  - `Management Console`: Thin client that allows the user to manage the content of the LSPS Server
  - `Business Activity Monitor`: Thin client that allows the user to work with BIRT reports based on models' KPIs

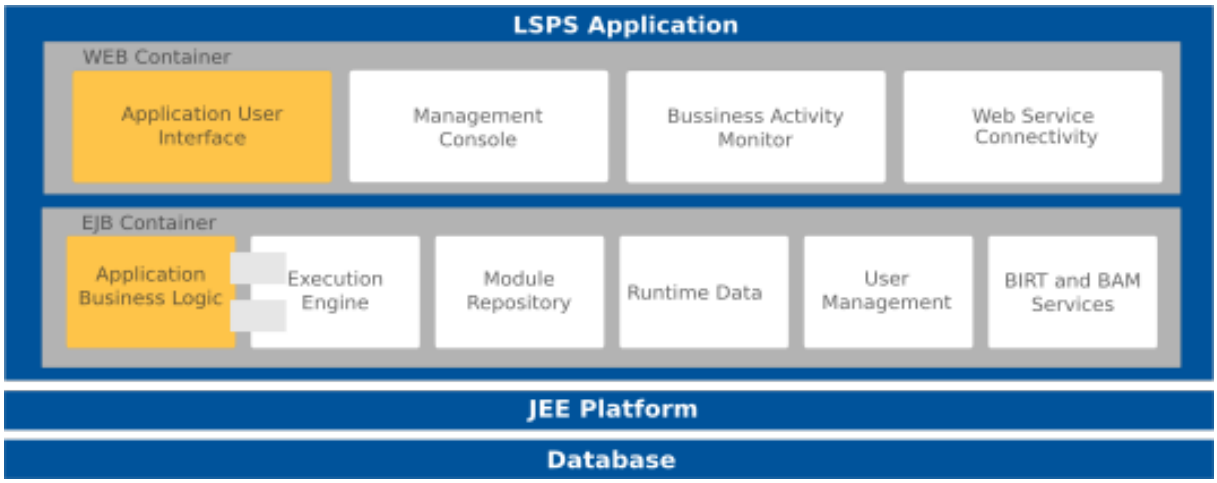


Figure 1.1 LSPS Application architecture

The LSPS Application is generated by LSPS SDK, which is installed as part of PDS, the IDE for model development and a thick client for LSPS Servers and structured as follows:

- `ear`: project for building the EAR archive from the ejb and war projects
- `ejb`: *Application Business Logic* with Java classes implementing custom items visible to model execution
- `embedded`: files that are needed for the SDK Embedded Server

  Important: Note that this is not the LSPS Embedded Server: check the referenced libraries of the `<APP>-embedded` project.
- `tester`: JUnit testing resources
- `vaadin`: *Application User Interface* resources with JavaScript and style sheets
- `vaadin-war`: project for building the WAR archive of the Application User Interface including Vaadin themes and resources.

Chapter 2

Setting Up

Before you can generate the LSPS Application, make sure the following requirements are met:

- You have installed the Living Systems® Process Design Suite with SDK (run the LSPS installer and, in Step 5, make sure that both SDK and PDS are selected).
- You have installed Maven and set up the Maven repository.
- You have set up the Maven repository when you installed PDS.
- You have set up the M2_REPO variable in PDS.

  To create and set the `M2_REPO` variable, go to **Window** > **Preferences**; then **Java** > **Build Path** > **Classpath Variables**; click **Add** and define the M2_REPO variable.

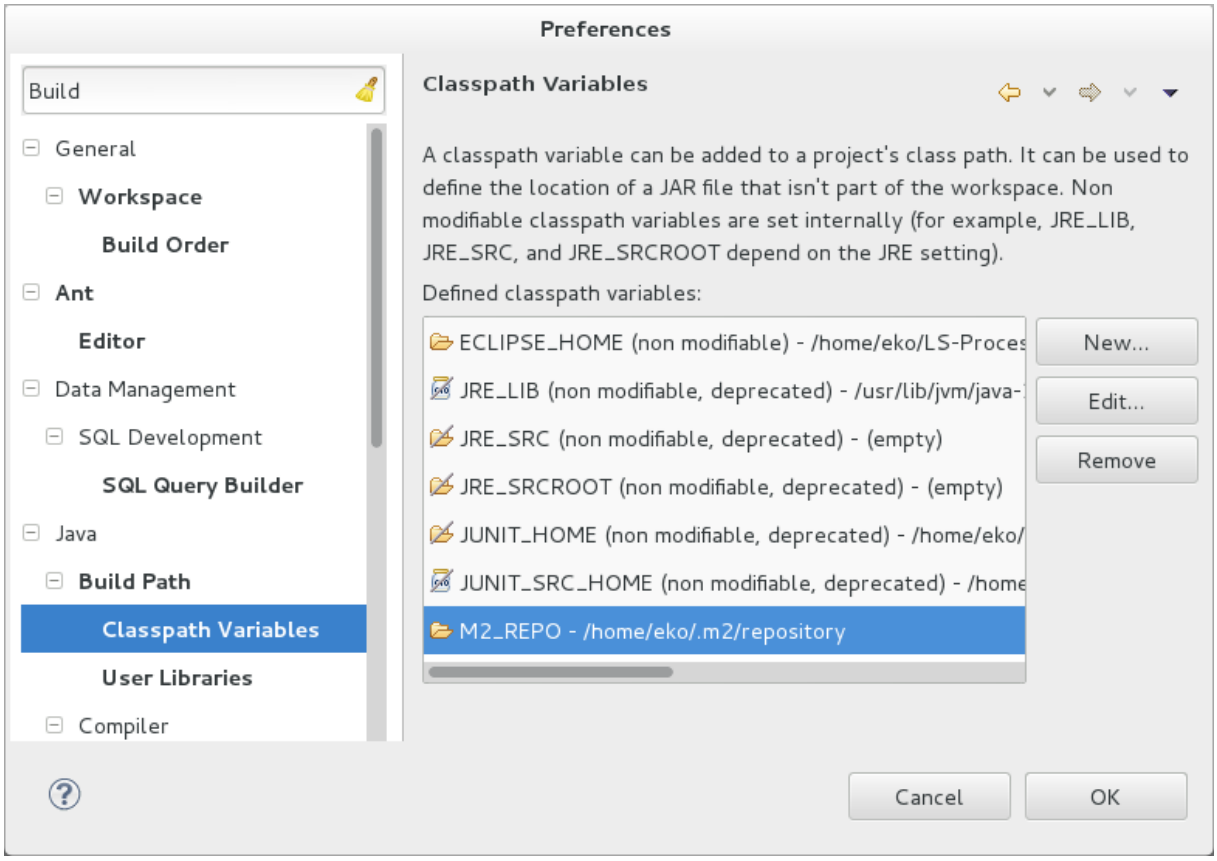


Figure 2.1 M2_REPO variable set up

To generate the LSPS Application from the provided maven artifact, do the following:

1. Go to File > New > Other.
2. In the New dialog, find and select LSPS Application.
3. Click Next.
4. In the New LSPS Application dialog, define the maven properties of the application.
5. Click **Finish** and wait for the building process to finish.

Note that the system also generated a launcher for the SDK Embedded Server and the compilation configuration for the LSPS Application, both accessible from the Run menu:

- The Maven build configuration for compilation of the application
- The Embedded Server configuration runs the SDK Embedded Server, and builds and deploys

> **Important:** When you generate the application artifact, the system generates an SDK Embedded Server and its launcher called `<YOURAPP> Embedded Server Launcher`; note that this is a different server form the PDS Embedded Server and cannot be launched with the **Start Embedded Server** button  the application EAR (it runs the `main()` method of the `<APP_PACKAGE>.embedded.LSPSLauncher` class).

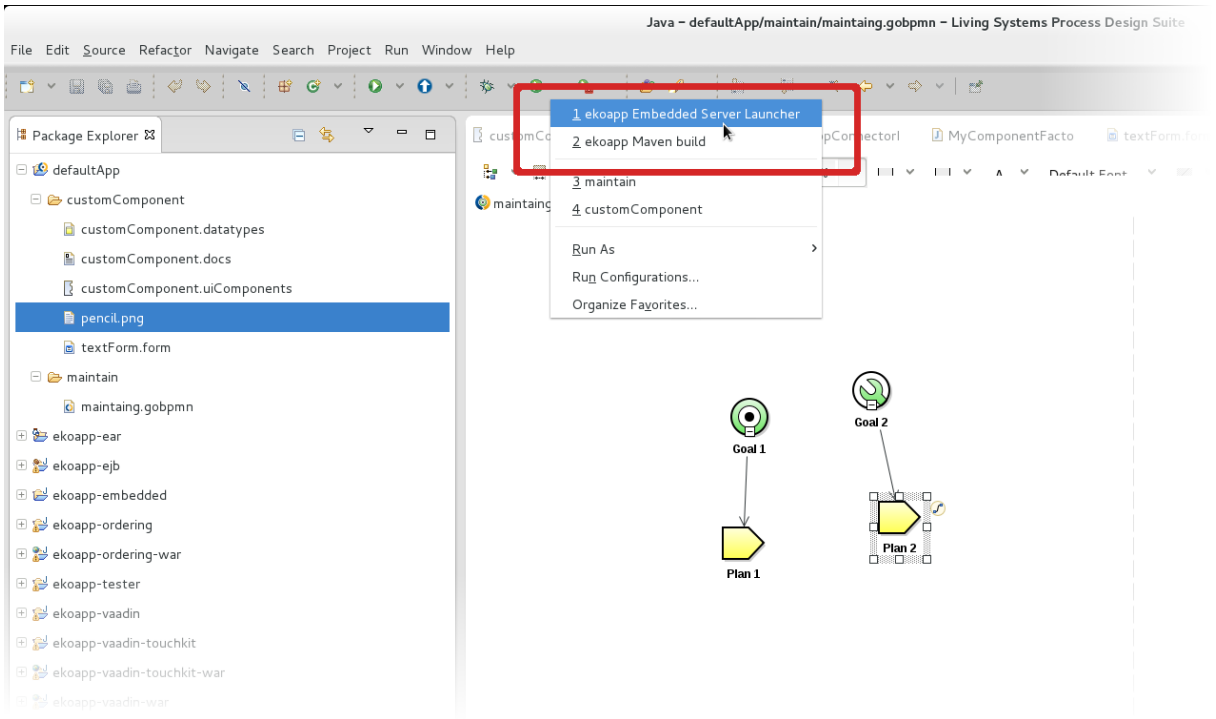


Figure 2.2 Application Launcher

API Javadoc is available in the `documentation/apidocs` folder.

Chapter 3

Model Instance

When you create an instance of a Model, the execution of the Model instance is managed by the execution engine based on the underlying Model stored in the Module Repository of the engine. The Model instance itself holds only its properties, information about its parent Model, and contexts with runtime data. The contexts are created based on elements that represents a namespace in the Model: every Module instance, every process instance, every sub-process instance, etc. has its own context;

Note: Before you can create an instance of a Model, you must upload it to the LSPS Server:

Note that if an executable Module imports other executable Modules, all Modules are instantiated as Module instances of one Model instance and start their execution.

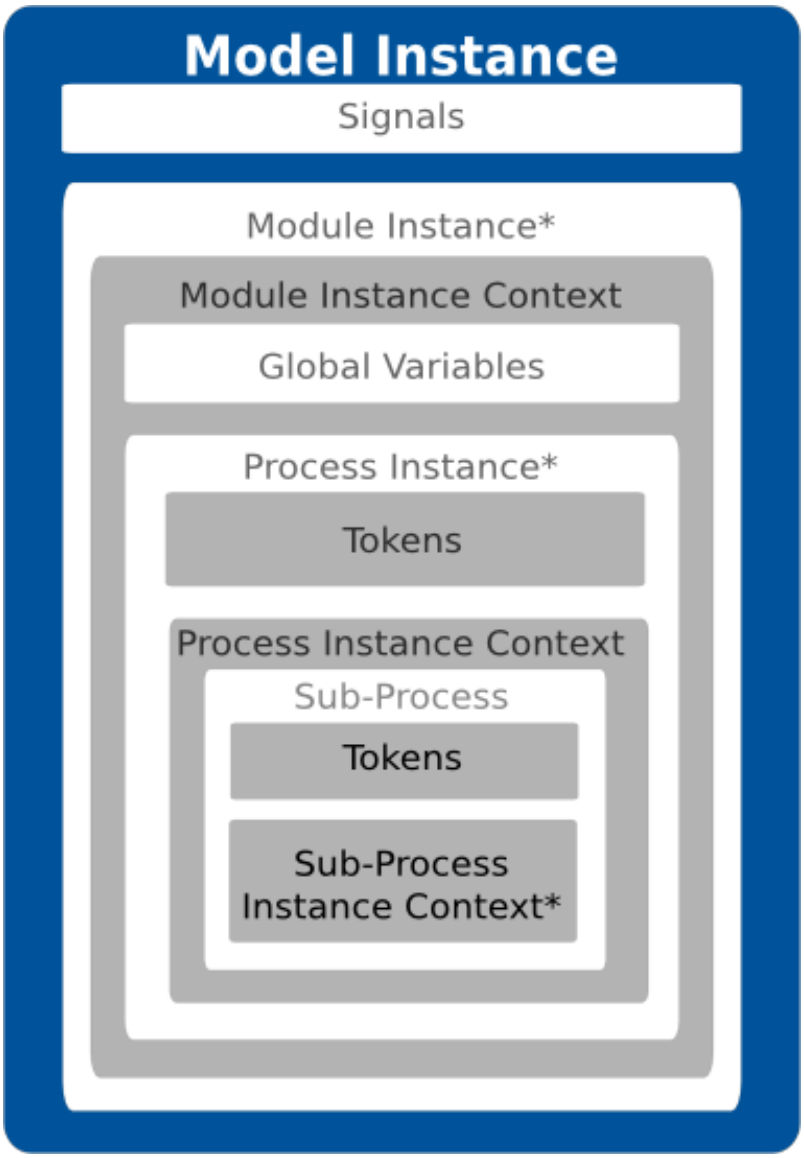


Figure 3.1 Structure tree of a Model instance

If you want to inspect the exact structure of a Model instance, you can export a Model instance to XML, for example, from the Management perspective of PDS.

3.1 Transactions in Model Instances

When a Model instance is executed, its execution is split in transactions: the model instance status is persisted on every transaction commit. Note that database transaction boundaries are bound to session boundaries.

When a model is instantiated, the following happens:

1. All its executable modules are instantiated.
2. The first database transaction for the model instance is created.
3. All BPMN and Goal processes in the executable modules are instantiated including any imported processes.

4. On Goal processes, all top goals are triggered and become either *ready* or *running* if their precondition evaluates to true. The running Goals trigger their sub-goals or plans.

5. Tokens are generated on all start events of the relevant process instances and on activities with no incoming flows. Every token is "moved" as far ahead as possible, that is, until it hits a wait point. A wait point is considered the following:

    - asynchronous task (user task, web service task, etc.)
    - intermediate event that requires a time slip (timer; signal, if the signal has not yet been received, etc.)
    - parallel join gateways that wait for a token

All moves of all tokens comprise one EJB transaction. The transaction is committed or rolled back and closed along with the session and the status of the model instance is persisted.

> **Important:** Instances of shared Records are persisted in the Model instance as the entities' primary key and record type: however the value is stored in the respective database table and can be used by other Model instances or systems. In every new transaction, the system fetches the values of the shared Records from the database. Therefore consider the possible performance impact when working with large amounts of record instances.

If some tokens remained in the model instance, the model instance remains in the RUNNING state, but no actions are performed, the model instance is invoked (woken up) when some of the following happens:

- A human task is submitted.
- A Timer is triggered.
- A condition with a shared record is met.
- A signal is received.
- A web service request/response is received.
- An asynchronous task is triggered.
- An Admin action occurs (an expression is evaluated, a context value changed, etc.).

Transactions in a Model instance with two Process instances

The first transaction includes generating of tokens on None Start Events and moving them to the closest wait points; in this case, 3 human tasks. The system then waits for the tasks to be submitted. Note that the order of the transactions is just an example order and it depends on the order in which the human tasks are submitted and the condition evaluation time.

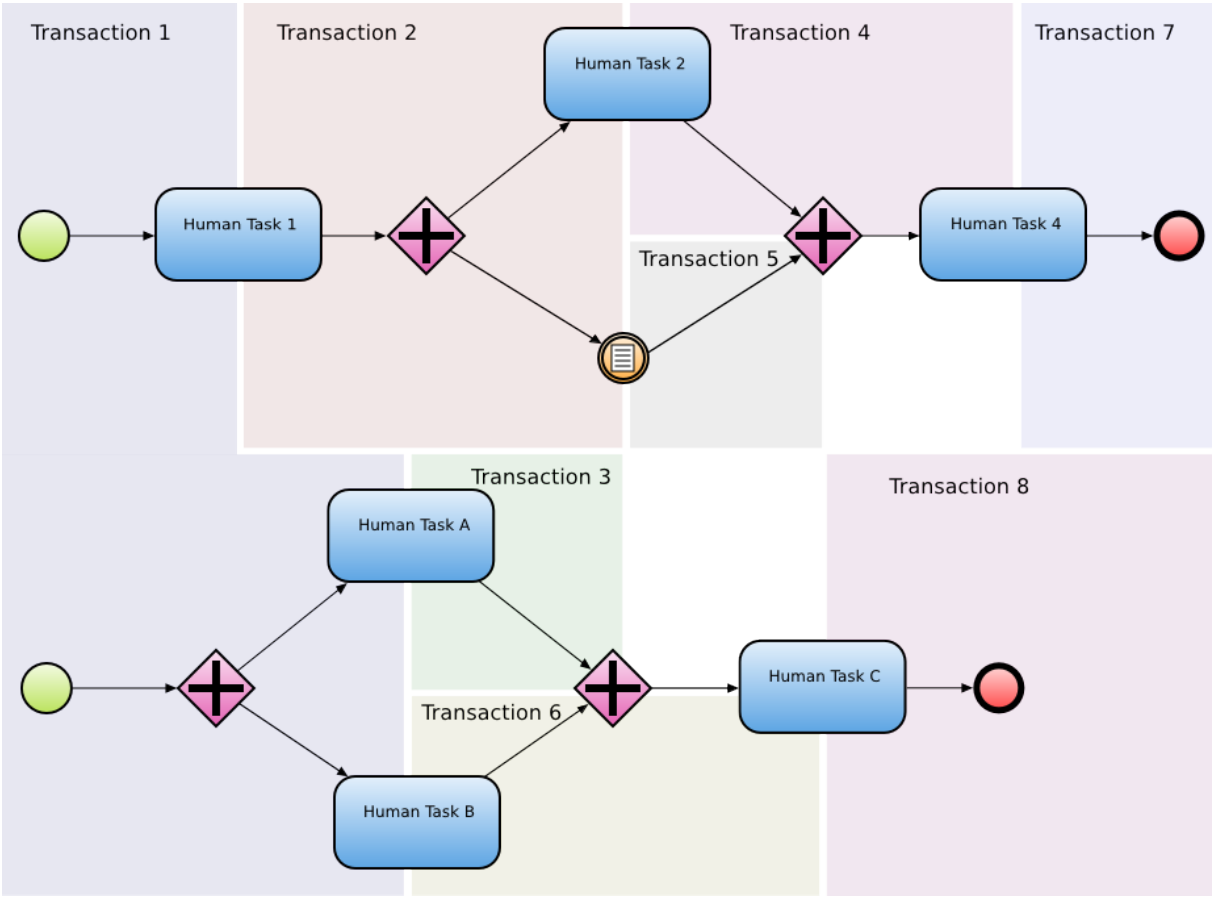


Figure 3.2 Transactions in a Model Instance

On model instance invocation, the following happens:

1. A new transaction and session are created based on the persisted model instance data. The persisted model instance data includes:
    - tokens with their positions
    - execution context (variables, referenced values, shared records)
2. Every token is "moved" as far ahead as possible until it hits a wait point.
3. Signals are handled, goals activated and deactivated, values changed, etc.
4. Every token is "moved" as far ahead as possible.

That means that an invocation is in general identical with one transaction; however, if a transaction can include a custom object, such as, a function, which can create its own transaction. Such a transaction is still part of the same invocation.

The transaction is committed or rolled back and closed along with the session (the status of the model instance is persisted).

3.2 Execution Levels

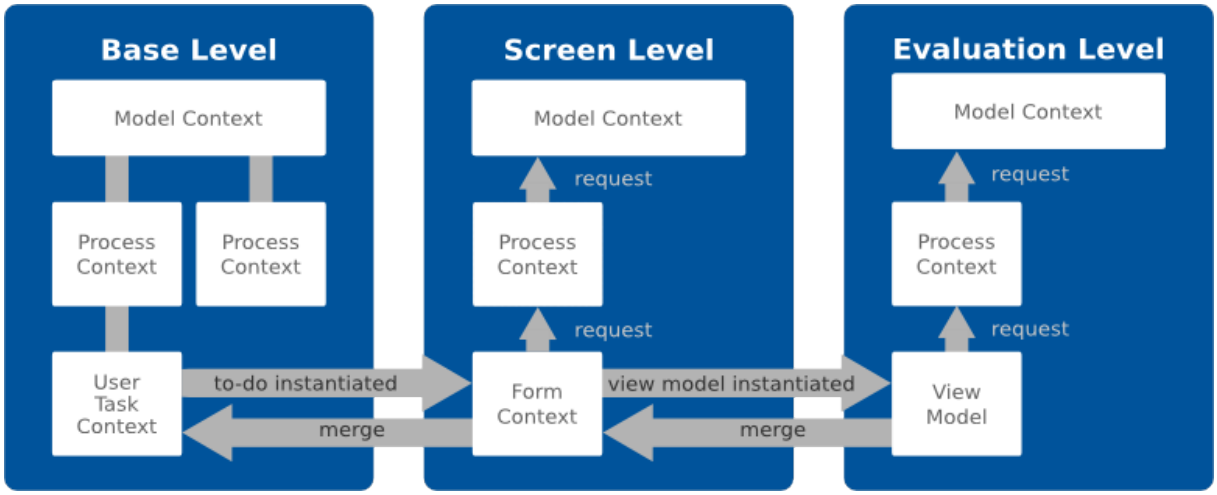
A contexts of a Model instance can exist on different execution levels. Execution levels allow an element to have different context data on different levels and so to have different "versions" on runtime.

The execution level of the model instance, called the **base level** or **0** level, holds the "real" model instance contexts; consequently, for example, changes on Shared Records on this level are reflected instantly in the database.

The structure of execution levels is hierarchical: levels created on top of the base level are denoted as `0:1`, `0:2` etc. Levels created on top of the level `0:1` are denoted as `0:1:1`, `0:1:2` etc. Changes performed in `0:1` are visible only in `0:1` and in its child levels. However, the changes are not visible in the base level or in the other levels that are on the same level in the hierarchy (`0:2`, etc.).

Contexts of non-base levels store only values that are changed: Other values are not stored.

Data of contexts of non-base levels are loaded on request: when you requests an entity that is not yet present in the context on the given level, the system requests the data from the context of the super level. The requesting continues up to the base level.



This mechanism is used to separate data used by forms from the "real" data and merge them only when required.

- To create an execution level, call the method `com.whitestein.lsps.engine.state.xml.EvaluationLevelUtils.nextSublevel(String, ModelInstance)`.
- To merge changes from a level into its super level, use the `com.whitestein.lsps.engine.lang.EvaluationLevelMerger.mergeLevel(String)`.

  This method can merge for example changes from level `0:1:1` to level `0:1`. When changes are merged to the base level, they become visible to the processes of the model instance in case of variables and non-shared records and they become visible to other model instances in the case of shared records. On merge to the parent level, the system checks for data conflicts: For example, if a variable is changed in the levels `0:1` and `0:2` and both levels are merged to their parent level 0, a conflict is detected. In the case of records, the conflict check is performed on each property: If the property P1 of a record R is changed in the level "0:1" and property P2 of the same record R is changed in the level "0:2" no conflict is detected during merge.
- To clean changes in a level, call the method `com.whitestein.lsps.engine.state.xml.EvaluationLevelUtils.cleanDataOfEvaluationLevel(ModelInstance, String)` or `com.whitestein.lsps.engine.state.xml.EvaluationLevelUtils.cleanDataOfLevelAndSublevels(ModelInstance, String)`.

  Note that the `com.whitestein.lsps.engine.state.xml.EvaluationLevelUtils.cleanDataOfLevelAndSublevels(ModelInstance, String)` method cleans changes in the level and in the child levels.

- To remove a level, use the method `com.whitestein.lsps.engine.state.xml.EvaluationLevelUtils.removeDataOfEvaluationLevel(ModelInstance, String)` or `com.whitestein.lsps.engine.state.xml.EvaluationLevelUtils.removeDataFromLevelAndSublevels(ModelInstance, String)`.
- To check if there are changes in the non-base levels of a model instance, call `com.whitestein.lsps.engine.state.xml.ModelInstance.isDirty(boolean)`.

Restrictions

- A shared record instance created in a non-base level and not merged into the base level is not registered in the entity manager and hence not returned by queries.
- Functions with side effects can cause changes in application state even if evaluated in a non-base evaluation level. Such functions are, for example, `createModelInstance()` and `sendSignal()`.

  Note: The GUI mechanism provided by the *ui* module makes use of execution levels:

  - Each form is created in the so-called *screen level*
  - Contexts of View Models are created on underlying levels referenced to as *evaluation levels*.

Chapter 4

Customizing Application User Interface

Basic customization of the Application User Interface, such as, color schema, logo, margin, and font can be changed from the Management Console using the Dynamic Theme feature (refer to the `Management Console` guide).

To customize the appearance of your application more dramatically, you need to adjust the code of your application:

- Vaadin theme: SCSS and properties files in the `vaadin-war` project
- Component definition in the `vaadin` project : The component definition defines the layout of the application, such as its navigation menu, content of the application components, etc.

Generally it should suffice to modify the application appearance in the application Vaadin theme, that is, in the theme's SCSS resources and properties files. However, if you want to modify the components present in the application front-end, you need to create the code for the component and integrate it in the application code.

> **Note:** You can also implement your own custom objects and use them to perform customizations to the application behavior.

Related topics:

- Customizing Themes
- Customizing Content and Layout

4.1 Customizing Themes

To create and modify a themes in PDS with your application, we recommend to download the theme from the Sampler through the Process Management Console and import it into the vaadin-war project of your LSPS application.

You will need to set up the Sass compiler to compile your themes and register your scheme with the `<YOUR_APP>.vaadin.util.Constants` class.

4.1.1 Creating Themes

To create a custom theme for your application, proceed as follows:

1. Get the base code for your theme:
   - Create a copy of one of the theme directories in `<YOUR_APP>-vaadin-war/src/main/webapp/VAADIN/themes`.
   - Alternatively, download it from the Web Management Console:
     (a) Run the application and download a theme from the Process Management Application:
     (b) Go to <SERVER>/lsps-management and log in.
     (c) Click the Dynamic Themes button in the sidebar.
     (d) Click the Sampler () button and then the Download () button.
     (e) Enter the target zip file name.
     (f) Extract the content of the zip file to `<PATH_TO_YOUR_APP>-vaadin-war/src/main/webapp/VAADIN/themes/<YOUR_THEME_NAME>` folder.
     (g) In PDS, go to `<YOUR_APP>-vaadin-war/src/main/webapp/VAADIN/themes`, refresh the directory.
2. Change to the theme directory and edit the main class name in the `style.scss` file.

   Every theme has a `style.scss` file that includes all relevant scss file in the theme. It is the included scss files you need to change: if possible, introduce your changes to the `sass/theme-<THEME_NAME>_variables.scss` file before modifying any other scss files.
3. Register your scheme with the `<YOUR_APP>.vaadin.util.Constants` class.
4. Set up the Sass Compiler configuration and compile your theme.
5. Compile and deploy your application.

   If you are running a server, the new css files are hot-deployed on sass compilation so it is not necessary to compile and re-deploy the application.

4.1.1.1 Setting up the Sass Compiler for a Theme

To compile the modified themes, you need to set up the Saas compiler.

To set up the run configuration of the Sass compiler in PDS, do the following:

1. Go to **Run** -> **Run Configuration**.
2. In the left pane, select Java Application and click the New button in the caption area of the pane.
3. On the right, define the following:
   (a) Name: a name of the configuration
   (b) Project: `<YOUR_APP>-vaadin-war`

(c) Main class: `com.vaadin.sass.SassCompiler`

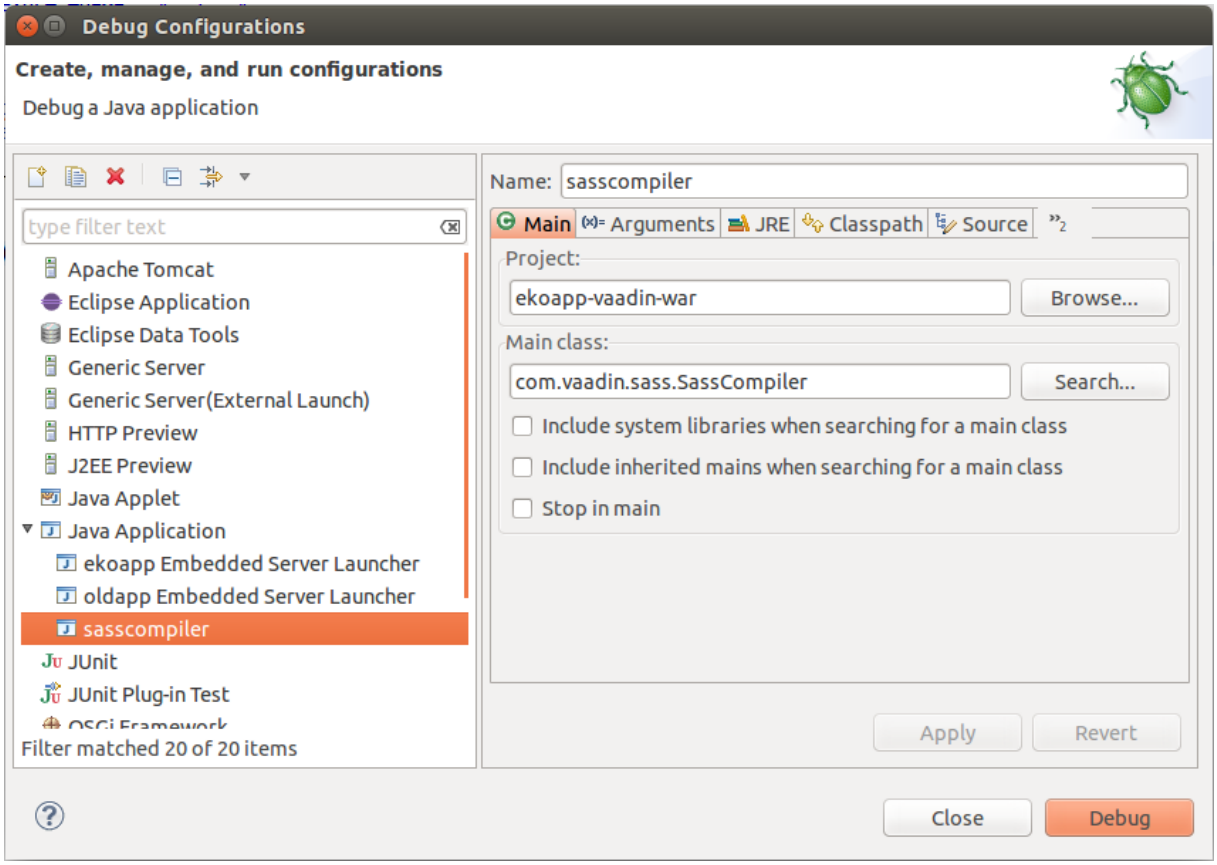


Figure 4.1 Sass Compiler Run Configuration For the blue Theme

4. Switch to the Arguments tab and define the input scss file and output css file as **Program arguments** (make sure to define the scss file as the first argument): the output css file must be located in the theme directory and have the name `styles.css`.

   For the provided themes, the main scss is the style.sccs file, which includes the relevant scss files so for the themes delivered with LSPS, the arguments are defined as follows:

```
src/main/webapp/VAADIN/themes/<THEME_NAME>/styles.scss
src/main/webapp/VAADIN/themes/<THEME_NAME>/styles.css
```

4.1.2 Setting the Default Theme

To set the default theme for your application, open the Constants.java class located in `<YOUR_APP>-vaadin/src/main/java/<YOUR-PCKG>/vaadin/util/` and the DEFAULT_THEME value.

```
public static final String DEFAULT_THEME = "my_theme";
```

Make sure the theme is among the THEMES value.

4.2 Customizing Content and Layout

You can change the layout of the components of your Application User Interface as well as the components that appear in the application by editing `LspsUI` which extends Vaadin's UI component.

The underlying default component tree of the Application User Interface is as follows:

- LspsUI that extends Vaadin's UI

  It represents the root component and holds the user info and connector to the LSPS server. It can contain only one AppLayout object as its child.

- applayout: An AppLayout instance that extends CustomComponent and implements ViewDisplay

  Since it is a CustomComponent, it can have only one child component.

- layout: A CustomLayout instance which loads the layout template `<APP>-vaadin-war/src/main/webapp/VAADIN/themes/<THEME>/layouts/page.html`.

  It holds the following layout components of the default Application User Interface:

  - navigation: NavigationMenu that holds the standard LSPS menu items
  - userMenu: NavigationMenu that holds custom menu items

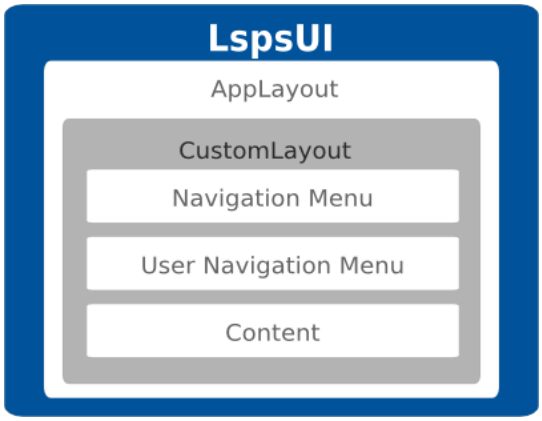


Figure 4.2 Structure of the Default LspsUi

4.2.1 Implementing LspsUI

The components of the default application screen are defined in the vaadin.core package. The root screen component is defined in the LspsUI class that extends the Vaadin UI class.

Its *init()* method creates the content root layout and a connector between LSPS forms and your app, called the AppConnector, and calls the initLayout() method which assembles the screen content.

Therefore, if you want to change the components of the application, you will be mostly editing the `initLayout()` methods.

If you decide to implement the LspsUI class anew to create your own custom application, make sure it meets the following requirements:

- It must declare the LSPS widget set for the Vaadin servlet.

```
@Widgetset("com.whitestein.lsps.vaadin.widgets.WidgetSet")
public class LspsUI extends UI implements ErrorHandler { ...
```

- It must provide an implementation of LspsAppConnector. LspsAppConnector is an interface that defines the binding contract between the LSPS vaadin renderer and the rest of the application.
- It must provide an implementation of LspsFormConnector. LspsFormConnector is an interface for a connector between a form and a view.
- It must use JAAS for user authentication. The setting is available in the webapp project of the Default LSPS Application and that in the `login.jsp` and in the `WEB-INF/web.xml`.

4.2.2 Adding Item to the Navigation Menu

To add a custom item to the navigation menu, do the following:

1. Open your AppLayout.java file.
2. Locate the `//Examples:` comment in the buildMenu() method: call the addDocumentItem() or addRun↩ ModelItem() method calls as appropriate to add your items to the menu.

   The examples demonstrate how to add custom items to the Navigation menu: to preview them, set the **if** condition to true, compile and run the application.

When adding custom items, consider whether the user permissions need to be taken into account: if this is required, use the respective methods, such as, User's `hasRightToOpenDocument()`.

4.2.3 Adding Header and Footer

To add a footer or header to the content area of the application by modifying the Vaadin component tree, do the following:

1. Create a Vaadin component with the header or footer.

   ```
   //example header defined as member variable of the AppLayout class:
   Component header = new Label("header");
   //example footer defined as member variable of the AppLayout class:
   Component footer = new Label("footer");
   ```

2. Create a layout component that will hold the contentArea and the header or footer. The layout must have height set to 100% and expand ratio 1.

   ```
   VerticalLayout contentAreaWithHeaderAndFooter = new VerticalLayout(header, contentArea, footer);
   contentAreaWithHeaderAndFooter.setHeight("100%");
   contentAreaWithHeaderAndFooter.setExpandRatio(contentArea, 1);
   ```

3. Modify the mainLayout to hold the contentAreaWithHeaderAndFooter instead of the contentArea.

   ```
   mainLayout.addComponents(menuArea, contentAreaWithHeaderAndFooter);
   mainLayout.setExpandRatio(contentAreaWithHeaderAndFooter, 1);
   ```

4.2.4 Adding a Locale

To provide a new localization setting and sources, do the following:

1. Create a properties file with the name localization_<LANGUAGE_CODE>.properties with the translations.
   Use one of the <YOUR_APP>-vaadin/src/main/resources/com/whitestein/lsps/vaadin/webapp/lo
   properties file as a template for your properties file.
2. Add the translation properties file to the <YOUR_APP>-vaadin/src/main/resources/com/whitestein/lsps/
   directory.
3. Add the option to the language picker on the Settings screen: open the <YOUR_APP>-vaadin/src/main/java/org/<
   OUR_APP>/<>/vaadin/page/AppSettingsView.java and add the definition to the create↩
   SettingsSection() method. The language code is based on the java.util.Locale class.

```
private VerticalLayout createSettingsSection(LspsUI ui) {
  VerticalLayout settings = new VerticalLayout();
  settings.setSpacing(true);
~
  Label settingsHeader = new Label("<h2>" + ui.getMessage("settings.applicationSection") + "<
  settings.addComponent(settingsHeader);
~
  this.languages = new OptionGroup(ui.getMessage("settings.language"));
  languages.addStyleName("ui-spacing");
  languages.addItem("en_US");
  languages.setItemCaption("en_US", English);
  languages.addItem("de_DE");
  languages.setItemCaption("de_DE", Deutsch);
  languages.addItem("sk_SK");
  languages.setItemCaption("sk_SK", Slovensky);
  //Italian localization setting:
  languages.addItem("it_IT");
  languages.setItemCaption("it_IT", Italiano);
~
  languages.setValue(ui.getLocale().toString());
  settings.addComponent(languages);
  return settings;
}
```

4. Compile and redeploy your application if necessary.

Chapter 5

Custom Objects

In your application, you can create implementations of the following custom objects:

- functions: implemented in the Expression Language or as a Java class methods
- task types: implemented as Java classes implementing the ExecutableTask interface
- form components: implemented in the Expression Language or as Java classes implementing the UI↩ Component interface

Every custom object has its implementation either in the Expression Language or in Java and its definition in a definition file: for example, a custom form component must be implemented as a Java class that extends the `UI↩ Component` interface or as an expression that returns a custom component in the Expression Language; and its custom component definition must be in a definition file so you can use it for modeling.

When implementing your custom objects in Java, you can implement them as POJOs or as EJBs: it is recommended to implement them as POJOs unless the object needs to use a server service. In such a case, the EJB must be then registered with the Execution Engine using the `ComponentServiceBean`.

- Custom Functions
- Custom Task Type
- Custom Form Components

5.1 Registering EJBs of Custom Objects

Once you have created your EJB implementation of a custom object, you need to register it with the Execution Engine:

1. In your ejb project, create a ComponentServiceBean class that extends com.whitestein.lsps.common.↩ ComponentServiceBase.
2. Inject the EJB:
    - For custom tasks, use the ExecutableTask interface with the bean name set to the implementing class name.

```
@EJB(beanName="SendGoodsTask")
private ExecutableTask sendGoodsTask;
```

- For custom functions, use your local interface that declares all functions used in the model.

```
@EJB(beanName="ShippingFeeFunctions")
private ShippingFeeFunctions shippingFeeFunctions;
```

3. Register the custom component implementing the `registerCustomComponents()` method:
   - If you have your interface for your implementation:

```
@Override
protected void registerCustomComponents() {
    //register(<task_instance>, <task_interface>.class);
    register(sendGoodsTask, SendGoodsTask.class);
    //register(<function_instance>, <function_interface_class>.class);
    register(shippingFeeFunctions, ShippingFeeFunctions.class);
}
```

   - If you do not have an interface for your implementation:

```
@Override
protected void registerCustomComponents() {
    register(<task_instance>, <task_implementation>.class);
    register(<function_instance>, <function_implementation>.class);
}
```

5.2 Custom Functions

In LSPS, a function is declared in a function definition file. The file can be then distributed in a library module to allow other users to make use of the function.

Once declared a function can be implemented in Java or in the Expression Language. For Java implementation, you will need to create and deploy a class with the method that the function will call, while if using the Expression Language the expression is defined directly in the function definition.

5.2.1 Creating a Custom Function

To create a custom function with implementation in Java or the Expression Language, do the following:

1. In the Modeling perspective, create or open a function definition file.
2. Add a new function and define the function main details:
   - **Name**: name used to call the function
   - **Return type**: data type of the return value
   - **Generic types**: comma-separated list of abstractions of data types used in parameters

     Generic types allow functions to operate over a parameter that can be of various data types (not only a single data type) in different calls. The concept is based on generics as used in Java. You can also make a generic data type extend another data type with the `extends` keyword. The syntax is then `<generic_type_1> extends <type1>, <generic_type_2> extends <type2>` (for details, refer to the Expression Language User Guide).
   - **Public**: function visibility
   - **Variadic**: function arity

     A function is variadic if its last parameter can be declared variadic, that is, zero or more occurrences of that last parameter are allowed.

- **Has side effects**: if true, on validation, the info notification about the function having a side effect is suppressed

  A function is considered to have side effects if one of the following is true:

  – The function modifies a variable outside of the function scope.
  – The function creates a shared record.
  – The function modifies a record field.
  – The function calls a function that causes a side effect.

- **Deprecated**: if true, on validation, a notification about that the called function is deprecated is displayed.

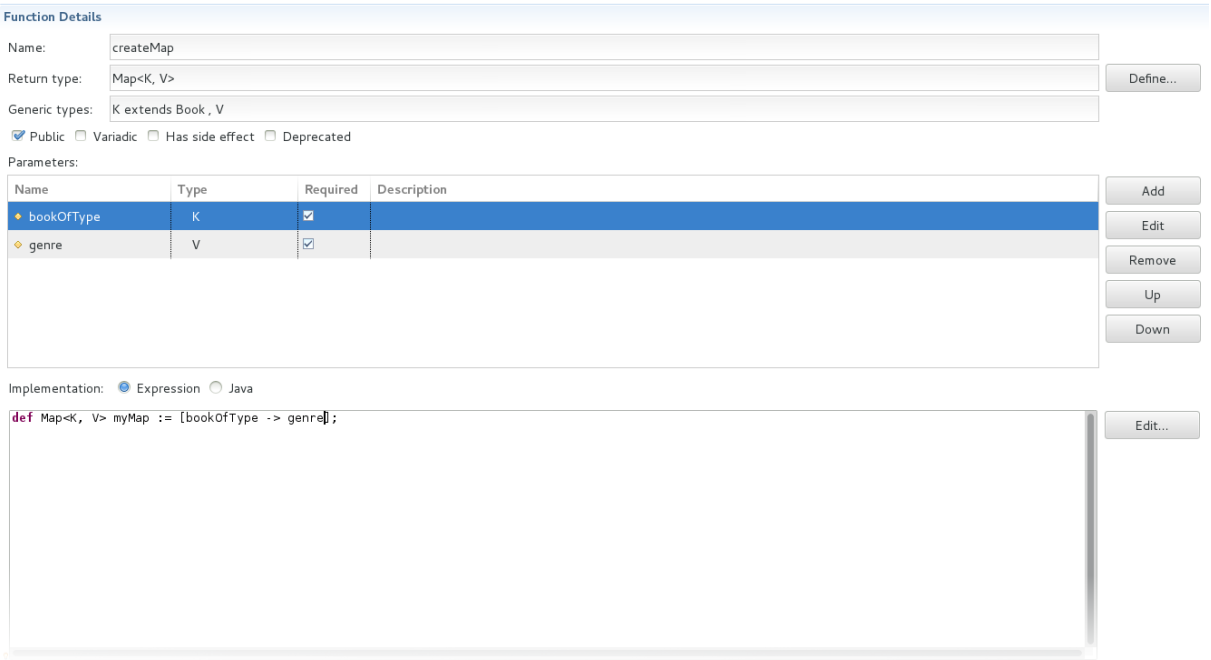


Figure 5.1 Example Function with Generic Parameters

3. Define the input parameters. For every parameter you need to define the following:

   - Name: parameter name unique within the function declaration
   - Type: data type of the parameter
   - Required: if checked, every function calls must define the parameter. The required property does not provide any additional runtime check of the parameter value.
   - Description: optional description of the parameter

     Note: Functions can be overloaded: functions with the same name but different parameters are considered different functions.

4. Define the implementation:

   - **Expression**: function implementation in the Expression Language

     ```
     new ITEmployee(
         Name -> &newEmployee.name,
         Birthdate -> &newEmployee.birthdate,
         Salary -> "You need to set the salary",
         CurrentPosition -> getPosition(newEmployee);
     return true)
     ```

   - **Java method implementation**: define the path to the method that implements the function (package, class, and method name)

     ```
     org.example.eko.Systemutils.getWeekday
     ```

5. In the ejb project of your application, create the package and class with the method.
6. Deploy the implementation as part of your Application User Interface.

You can now use the function call in your Module. Consider distributing the Module with the function declaration as a library.

5.3 Custom Task Type

To implement your own Task Types, you will need to define the work that a task of your type should perform as a Java class that extends the ExecutableTask interface and define the task type in a task type definition file along with its underlying Record.

Methods in Task Life Cycle

- `start`: Method called when the task execution starts.
- `TaskContext.getParameter(String)`: method that acquires the parameters of the task defined as a String

5.3.1 Implementing a Custom Task Type

When implementing a custom Task Type, the implementing Java class must implement the interface Executable↩ Task. Therefore you will need to create the class in your Application User Interface, and compile and deploy it to your server.

To implement a custom task, do the following:

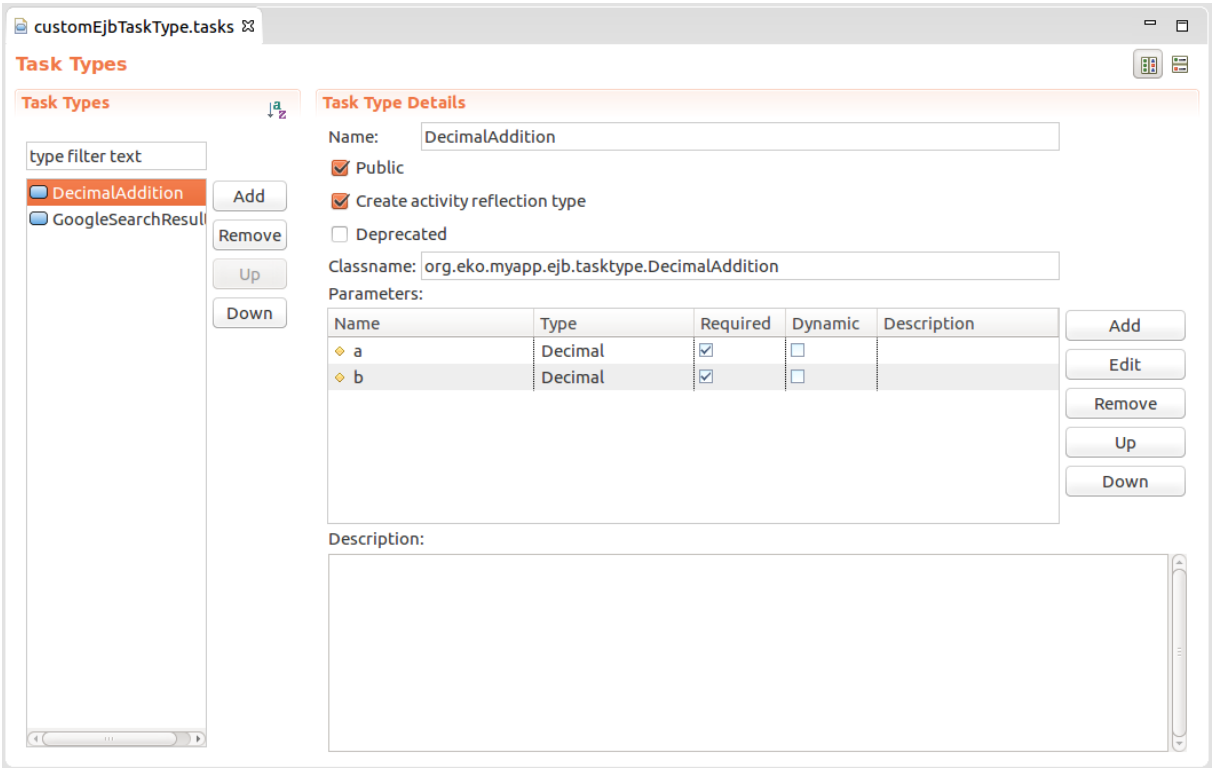
1. In a workspace with your Application User Interface, switch PDS to the Java perspective.
2. In the `<APP_NAME>-ejb` project, create your task class (right-click the package and go to New Class).
   Make sure, the class implements the `com.whitestein.lsps.engine.ExecutableTask` interface.
3. If you implementation depends on libraries that are not imported by default, do the following:
   (a) Add the library as a dependency to the project's pom file.
   Alternatively, define the dependency with its version in the pom.xml in the application directory: you will not need to define the version in the project's pom.xml.
   (b) On the command line, go to the application folder and run `mvn eclipse:eclipse`.
   (c) Refresh the resources in Eclipse.
4. Compile and deploy the Application User Interface.
5. Create the Task Type definition:
   (a) Switch to the Modeling perspective.
   (b) In the GO-BPMN Explorer, double-click the `task type definition`.
   (c) In the Task Type Editor, click Add under Task Types.
   A new task type is added to the list and Task Type Details area with its details appear.
   (d) Under Task Type Details in the Name text box, type the task name.
   (e) Select relevant flags:
   - Public: select to allow access from importing modules

- Create activity reflection type: select to create a Record that represents the task type
  You can then send the Record as a parameter to the Execute task.
- Deprecated: select to display a validation marker for deprecated elements.

(f) In the Classname text box, type the fully qualified name of the implementation class.

(g) In the Parameters list box, define the task parameters.
Check Dynamic to wrap the parameter value in a non-parametric closure. The parameter is then processed as `{ -> <parameter_value> }`. Note that the task implementation has to be able to process the closure parameter.

6. In the Description text box, provide the task description.



5.3.2 Generating Code for Handling Task Types

To allow you to handle tasks from within your Application User Interface code, you can generate the Java source code for task type declarations.

The generated code does the following:

- places the task implementation in the correct directory structure (for example, 'com->whitestein->lsps->tasks->demo->MyTask.java');
- defines a class declaration as an implementation of the com.whitestein.lsps.engine.ExecutableTask interface;
- creates a variable for each parameter in the task and the related setter methods used by the Living Systems Process Navigation Engine to access such values;
- defines an initial structure for the task documentation.

To generate the code for handling of the task types from a Module, do the following:

1. In the GO-BPMN Explorer view, right-click the GO-BPMN module.
2. Click **Generate** -> **Task Java Sources**.
3. In the Task Source Code Generation dialog box:
   (a) Select the check boxes of the relevant tasks.
   (b) In the Destination folder text box, specify the destination path.
   (c) Click Finish.

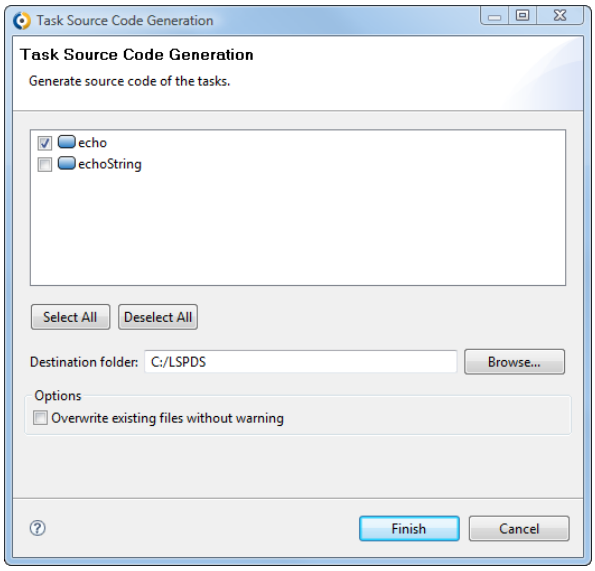


Figure 5.2 Task source code generation

5.4 Custom Form Components

You can implement a custom component in Java or in the Expression Language.

When creating a custom form component, you need the following:

- Implementation of your form component for the LSPS server

  You will need to create it in the Expression Language or as a Java class. When implemented in Java, the custom component class must implement the `UIComponent` interface to make sure it behaves like standard components. All classes implementing custom components must be registered in the LspsAppComponent↩ Factory class.

- Definition of your form component for PDS

  You will need to create a custom component definition and in case of Java implementation also the respective Record.

5.4.1 Creating Custom Component with Java Implementation

These instructions describe how to implement a custom form component, in this case, a custom slider with binding.

However, they do not contain information on how to implement a custom event for a custom component. For instructions on how to create a custom component with a custom event, refer to Creating Custom Forms with Custom Events.

To implement a custom component, do the following:

1. Create the custom component class in your application (in the default application, it is recommended to implement custom form components in the `<YOUR_APP>.vaadin.util` package in the `<YOUR_APP>-vaadin` project).

   The implementation must meet the following:

   - **Implement the `com.whitestein.lsps.vaadin.ui.components.UIComponent` interface.**

     Typically, it will extend a class that implements `com.whitestein.lsps.vaadin.ui.components.UIComponent`. Note that it is not recommended to extend any of the form components of the Standard Library since their methods might change.
   - **Define a constructor with UIComponentData as its input parameter.**

     The constructor should set the UIComponentData values. Component data are a wrapper of the component Record that also holds values of the record fields.
   - **Implement the *getComponentData()* method, which returns the component data.**
   - **Implement the *getWidget()* method, which returns the Vaadin component to be rendered on the client.**

     This method is useful if you want to use multiple Vaadin components to render a single LSPS custom component. Generally, you want the method to `return this`.
   - **It implements the *refresh()* method.**
     An example Slider Component implementation

```
public class SliderComponent extends Slider implements UIComponent {
~
    private final UIComponentData componentData;
~
    public SliderComponent(UIComponentData componentData) {
        super();
        this.componentData = componentData;
    }
~
    @Override
    public UIComponentData getComponentData() {
        return componentData;
    }
~
    @Override
    public AbstractComponent getWidget() {
        return this;
    }
~
    @Override
    public void refresh() {
        final LspsProperty property = new LspsProperty(this);
        property.setLocalizeBindingValue(true);
        setPropertyDataSource(property);
        getComponentData().getComponentFactory().applyCommonProperties(this);
    }
}
```

     An example Label Component implementation

```
import com.vaadin.ui.Label;
import com.whitestein.lsps.vaadin.ui.UIComponentData;
import com.whitestein.lsps.vaadin.ui.components.UIComponent;
import com.whitestein.lsps.vaadin.ui.events.UIEvent;
import com.whitestein.lsps.vaadin.util.UIComponents;
import com.whitestein.lsps.vaadin.util.Variant;
~
public class UIText extends Label implements UIComponent {
~
    private final UIComponentData uic;
~
```

```
        public UIText(UIComponentData uic) {
            this.uic = uic;
        }
    ~
        @Override
        public UIComponentData getComponentData() {
            return uic;
        }
    ~
        @Override
        public void refresh() {
    ~
            String text = Variant.definitionOf(this).getPropertyValue("text").closure()
        .inScope(this).call().string().valueOrNull();
            text = uic.getScreen().getContextHolder().getAppConnector()
        .getLocalizer().getLocalizedString(text, this);
            setCaption(text);
    ~
        }
    }
```

2. If you need additional Vaadin components, add the respective Vaadin add-on to the generated application:
   (a) Create a GWT XML in <YOUR_APP>-vaadin-war/src/main/resources/com/whitestein/lsps/vaadin/webapp directory.
   (b) Edit the file and append appropriate <inherits> XML element.
   (c) Enable automatic compilation of the your widget sets: open the <YOUR_APP>-vaadin-war/pom.↩ xml file and configure the maven Vaadin plugin.
   (d) In the pom file, uncomment the vaadin-client-compiler dependency.
   (e) Open the LspsUI Java file and modify the @Widgetset annotation, to reference your widget set, for example `@Widgetset("com.whitestein.lsps.vaadin.webapp.MyWidgetSet")`
   (f) Open the <YOUR_APP>-vaadin-war/pom.xml and add maven dependency to the Vaadin component jar file.
3. Modify the `LspsAppComponentFactory` class: uncomment the `createComponent` method and modify it to return your component when the respective Record is requested.

```
import com.whitestein.lsps.vaadin.LspsAppConnector;
import com.whitestein.lsps.vaadin.ui.UIComponentData;
import com.whitestein.lsps.vaadin.ui.UIComponentFactoryImpl;
import com.whitestein.lsps.vaadin.ui.components.UIComponent;
~
public class LspsAppComponentFactory extends UIComponentFactoryImpl {
~
    public LspsAppComponentFactory(LspsAppConnector connector) throws NullPointerException {
        super(connector);
    }
~
    @Override
    protected UIComponent createComponent(UIComponentData componentData) {
        final String type = componentData.getDefinition().getTypeFullName();
        switch (type) {
          case "CustomComponentModule::SliderRecord":
            return new SliderComponent(componentData);
          case "CustomComponentModule::TextComponentRecord":
            return new UIText(componentData);
          }
        return super.createComponent(componentData);
    }
}
```

4. Deploy your application.

5. In the referenced module, `CustomComponentModule` in the example, create the component Records which extend the respective **ui::UIComponent** record. Add any additional fields which the user needs to populate when they will use the component (make sure these are handled in your implementation properly).

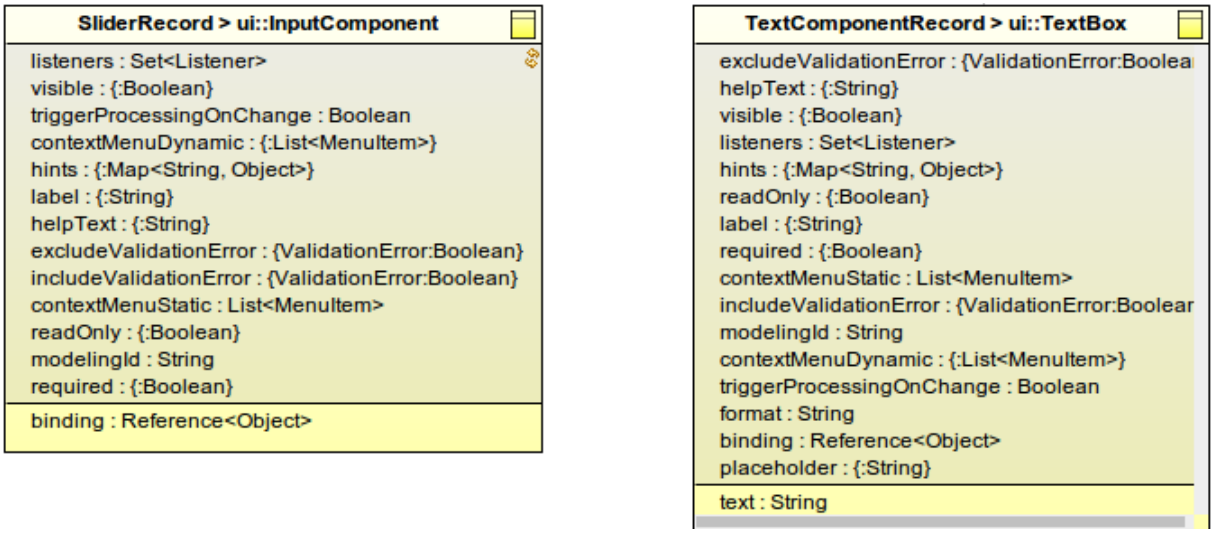


Figure 5.3 Custom component records

6. In your model, create a custom component definition in a custom component definition file.

   Set the Implementation property of the custom component definition to Data Type and enter the component record (in the example, `CustomComponentModule::TextComponentRecord` and `CustomComponentModule::SliderRecord`).

7. In the Properties area, define the component properties that will be available for editing in the Properties view:

   - Property Name: name of the underlying record field
   - Display Name: name displayed in the Properties view
   - Type: data type of the property (needed if the Implementation is defined as an expression)
   - Edit Style: child component edit style
     - **EXPRESSION**: Property is edited as an expression in the component.
     - **DYNAMIC_EXPRESSION**: Property is edited as an expression in the component and automatically wrapped as a non-parametric closure (the parameter is processed as `{ -> <parameter_value> }`).
     - **COMPONENT**: Property is handled as a child component.
     - **COMPONENT_LIST**: Property can be inserted multiple times as a child component.
   - Mandatory: whether the property value must be specified
   - Displayed in Editor: if set to true, the defined value of the property is displayed in the graphical depiction of the component in the Form editor

     Note that only one property can be displayed in the component graphical depiction.

     If the custom component extends a non-abstract UIComponent, it is rendered as its UIComponent parent and the Displayed in Editor setting is ignored.

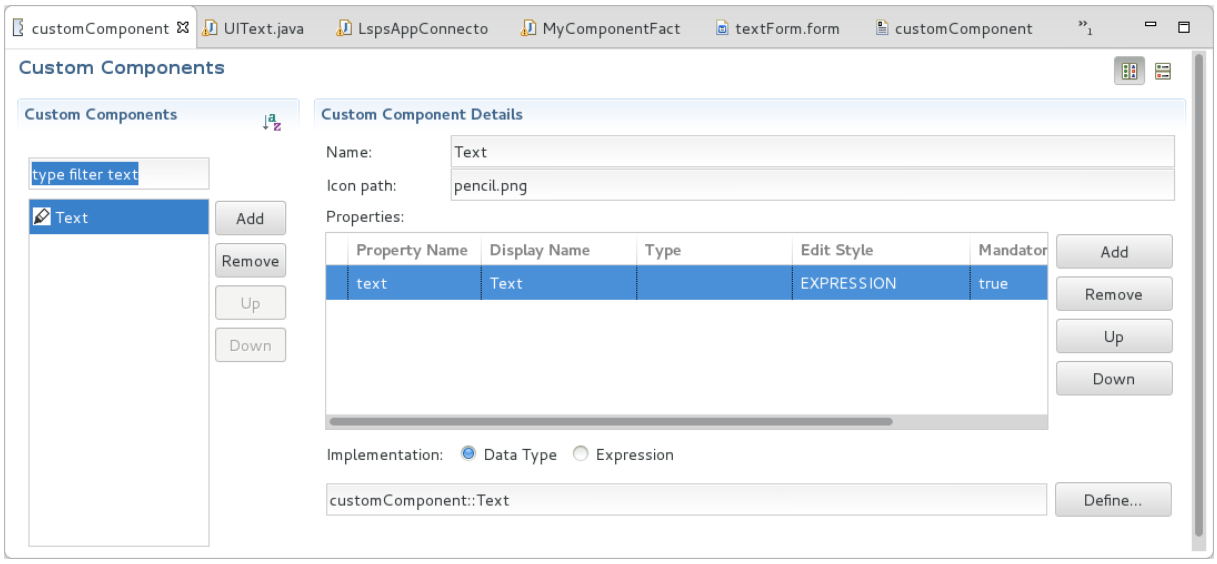


Figure 5.4 A custom component definition with implementation defined as data type

You can now use the custom components in your forms. Consider distributing the components as part of a Library.

5.4.2 Creating Custom Component with Expression Language Implementation

To create a custom component implemented in the Expression Language, do the following:

1. In your module, create a Record which extends a `ui::UIComponent` record. Add any additional fields which the user needs to populate when they will use the component.
2. Create a custom component definition in a custom component definition file.
3. In the Properties area, define the component properties that will be available for editing in the Properties view:
   - Property Name: name of the underlying record field
   - Display Name: name displayed in the Properties view
   - Type: data type of the property (needed if the Implementation is defined as an expression)
   - Edit Style: child component edit style
     - **EXPRESSION**: Property is edited as an expression in the component.
     - **DYNAMIC_EXPRESSION:** Property is edited as an expression in the component and automatically wrapped as a non-parametric closure (the parameter is processed as `{ -> <parameter_value> }`).
     - **COMPONENT**: Property is handled as a child component.
     - **COMPONENT_LIST**: Property can be inserted multiple times as a child component.
   - Mandatory: whether the property value must be specified
   - Displayed in Editor: if set to true, the defined value of the property is displayed in the graphical depiction of the component in the Form editor

     Note that only one property can be displayed in the component graphical depiction.

     If the custom component extends a non-abstract UIComponent, it is rendered as its UIComponent parent and the Displayed in Editor setting is ignored.

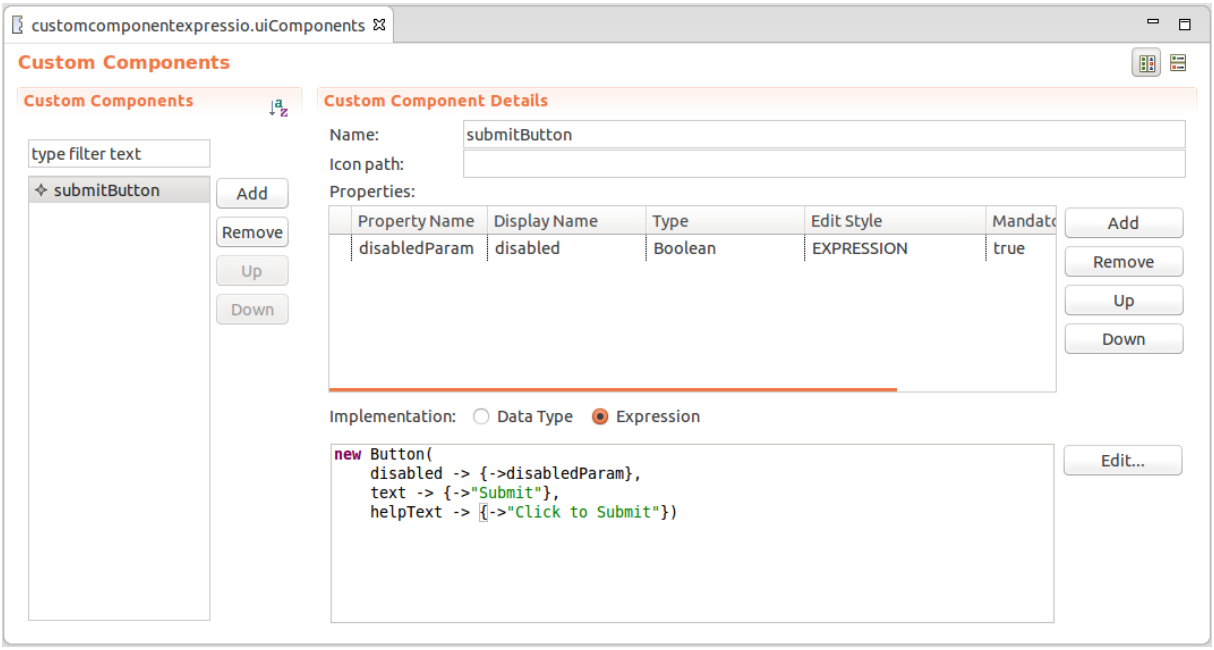


Figure 5.5 A custom component with a parameter

4. Set the Implementation property of the custom component definition to Expression and enter an expression that returns a custom component.

```
//create checkbox:
def ui::CheckBox cb := new ui::CheckBox();
def Boolean checked;
//bind checkbox to variable checked:
cb.binding := &checked;
//activate immediate mode for checkbox:
cb.triggerProcessingOnChange := true;
//handle value change of checkbox: refresh the checkbox list to have all boxes unchecked:
cb.listeners := {
  new ValueChangeListener(
    refresh -> { a->
      {checkBoxList }
    },
    handle -> { a ->
      if not checked then
        *(checkBoxList.binding) := {};
      end;
    }
  )
};
//return checkbox:
cb;
```

Chapter 6

Creating a Record Instance

You can create instances of Records from your code with the

If you need to work with Record instances in your code, be it custom functions, tasks, or your custom application, you can generate Java classes through which you can handle Record instances.

This allows you to decouple the Record instance from its definition better.

Example of difference in coding

```
// original LSPS
RecordHolder record = ctx.getNamespace().createRecord("core::ConstraintViolation");
record.setProperty("message", msg);

// better way with generated Java records
ConstraintViolationRecord better = new ConstraintViolationRecord(record);
better.setMessage(msg);
```

You can then regenerate the classes whenever the Record definition changes.

When you generate Java wrappers for Records, the system creates the following:

- subpage of the target package for every Module
- class for every Record in the respective subpackage
- interface for every Record classes
- RecordWrapperFactoryImpl
- RecordWrapperFactoryImpl for individual modules

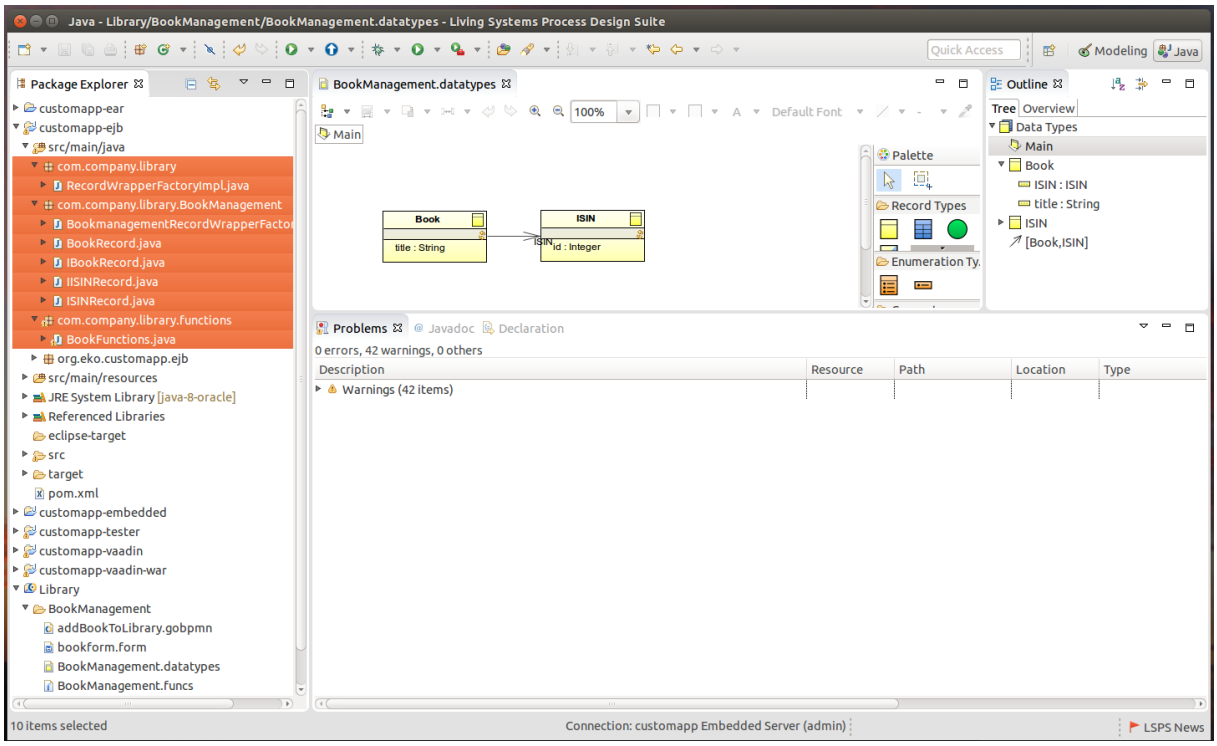


Figure 6.1 Generated Java Record wrappers

To generate Java classes, do the following:

1. Right-click a Module or project and go to **Generate** > **Record Java Sources**.

2. In the dialog, define the export properties:

   - Source folder: path in a project where to place the generated sources
   - Package: target package name
   - Additional record types: other Records that should be included in the operation (The parameter is primarily intended for including Library Records)
   - Class name prefix: prefix of exported class names
   - Class name suffix: suffix of exported class names
   - Class extends superclass: superclass of the generated Record classes
   - Class implements interface: interface of the generated Record classes
   - Selected modules: the system generates Java classes for the selected Modules and any dependencies

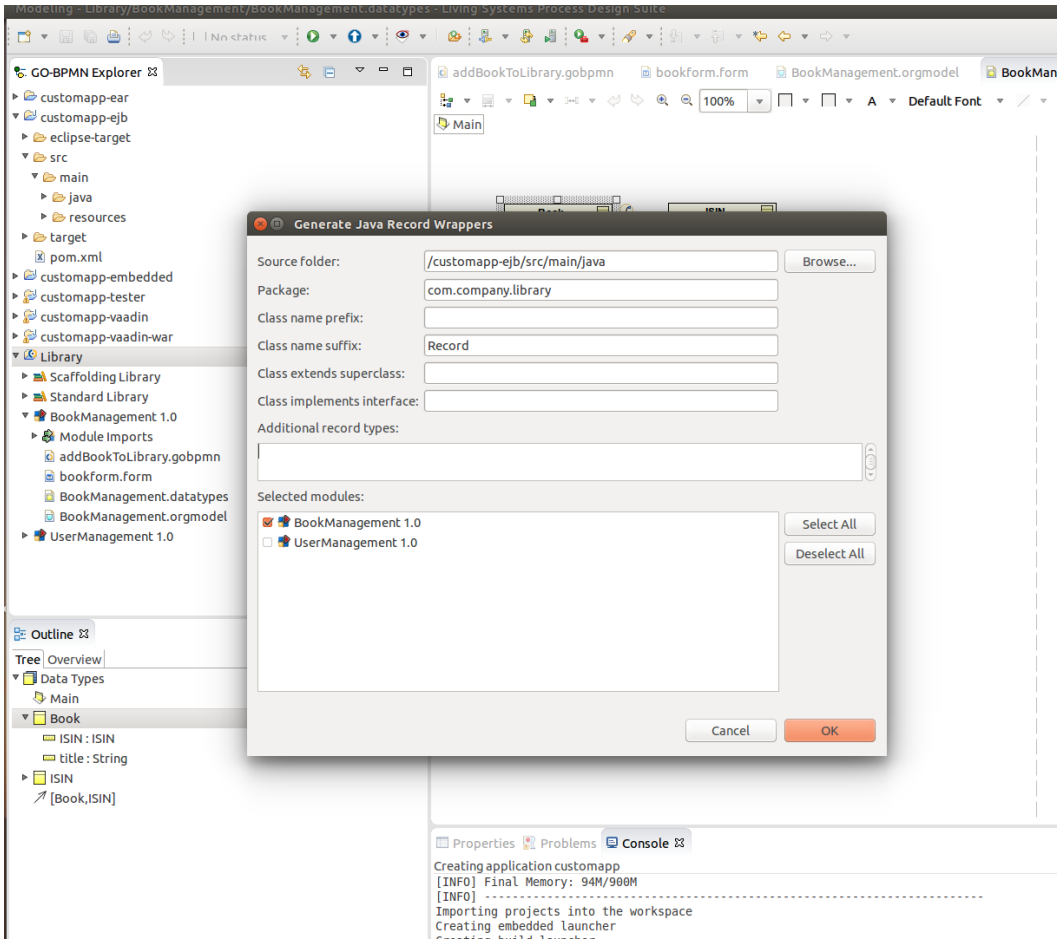


Figure 6.2 Generating Java Record wrappers

You can use the generated classes in implementation of custom function or task or anywhere in your application code as necessary.

Implementation of a custom function and its definition

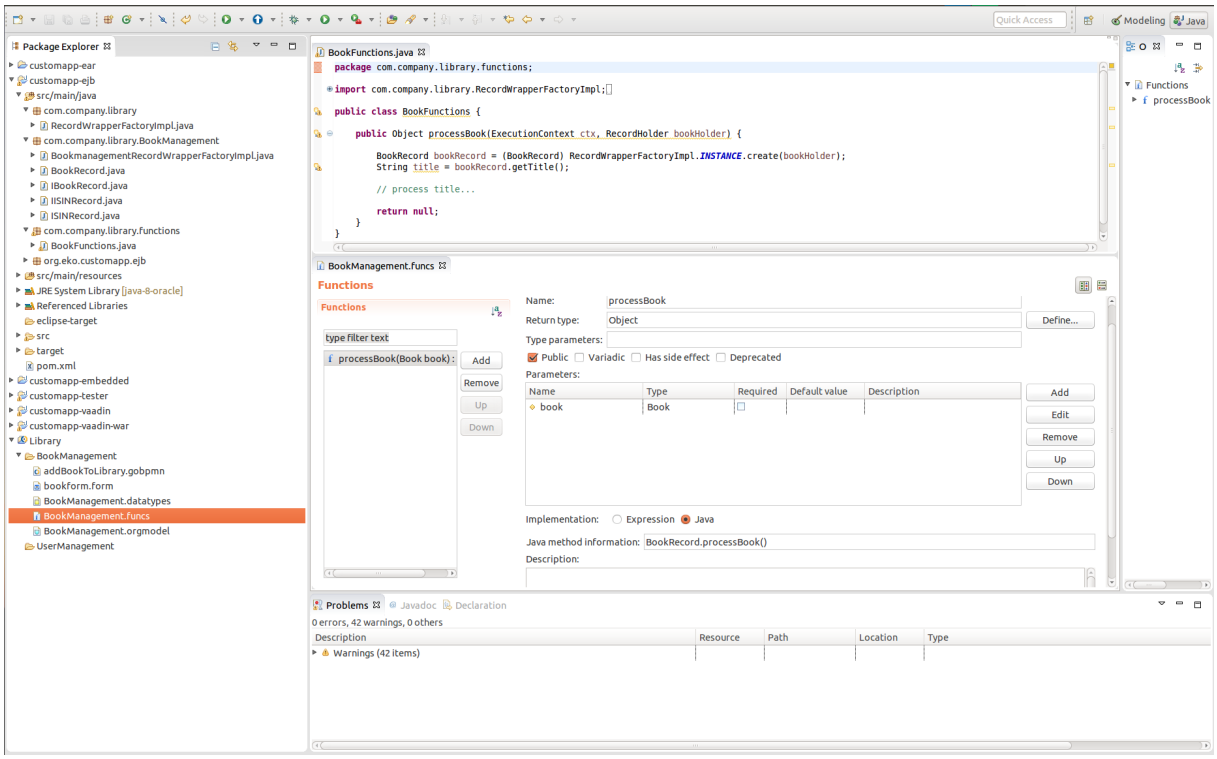


Figure 6.3 Custom Function implementation and definition

6.1 Working with Data Constraints

getConstraints

The getConstaint() function returns a collection of all constraints that are applied to a given record type and properties. These can be potentially null.

```
executionContext.getProcessModel().getConstraints(recordA, propertyA)
```

findTag

The findTag() function returns a validation tag with the given qualified name (may be simple or ∗ full).

```
executionContext.getProcessModel().findTag(qid)
```

Chapter 7

Persisting Data using Records

To persist business data from Model instances, define the respective records as shared. These will have their values persisted in the database.

Each record and the relevant relationships are reflected in a database table. When you create a new instance of the record or relationship, a new entry is added to the respective table. The mapping of shared records to tables makes use of Hibernate as its ORM library.

Mapping of shared records to database occurs in accordance with the following rules:

- Every shared record is stored in its own table.
- If there are relationship between two records the mapping is realized as follows:
  - If the relationship is 1:1, then the foreign key of the source record is placed in the table of the target record.
  - On the side without the foreign key, fetching is always eager.
  - If the relationship is 1:N, then the foreign key of the source record is placed in the table of the target record.
  - If the relationship is N:N, then the join table with the target-source-record primary keys is created.

Fetching of record instances is governed by Hibernate's principles with the transactions of the LSPS Server.

7.1 Customizing Entity Auditing

The revision history of shared Records, or `auditing`, relies on the *Revision Entity* that holds the revision ID and its timestamps: When an instance of an audited shared record changes, the Revision Entity calls the Revision Listener implemented by the *LspsRevisionListener class*. The listener creates a new revision instance with the revision data.

> **Important:** Information on how to set up and use auditing and the related database schema is available in the `Modeling guide`.

Hence if you want to record custom revision information, you need to do the following:

1. Expand the Revision Entity shared record by adding a Field or creating a related shared Record.
2. Implement a custom RevisionListener that extends LspsRevisionListener that will get the data for the field or related shared record.

> **Important:** On WebSphere, it is not possible to implement a custom RevisionListener.

7.1.1 Adding a Field to the Revision Entity

To create a custom Revision Entity with additional field so as to store additional revision information, do the following:

1. Add the field to the Entity Revision shared Record.

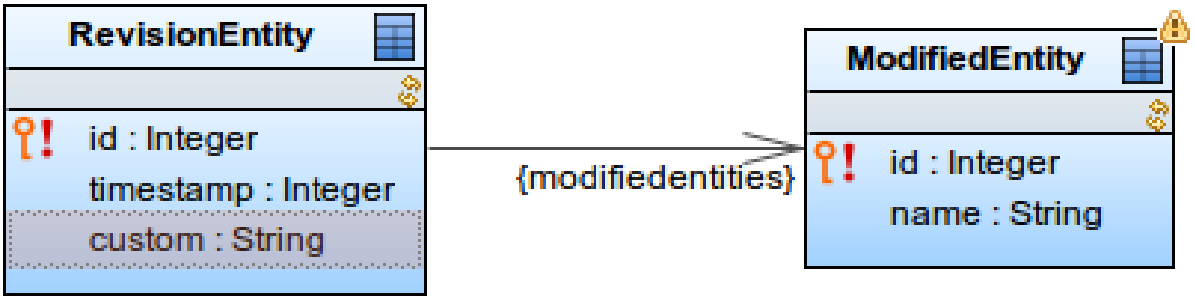


Figure 7.1 Revision entity record with the custom field

2. Implement the custom RevisionListener:

   (a) In the ejb package of your application, create a class that extends the *LspsRevisionListener* class and implements *EJBRevisionListener*.

   (b) Override the newRevision(Object revisionEntity) method of the your RevisionListener class:

   - Call the newRevision() method of LspsRevisionListener on the input Revision Entity: `super.newRevision(revisionEntity);`
   - Set the value of the field on the Revision Entity object. `((MapSharedRecordEntity) revisionEntity).set("USER", securityService.getPrincipalName());`

   An example implementation is here.

   Important: Including complex logic in your Revision Listener class, such as, contacting an external system to acquire data, might result in performance issues.

3. Register the ejb.

4. Open the Properties view of the Revision Entity record.

5. Open the Auditing tab and insert the class name of your RevisionListener into the *Revision Listener class* property.

6. Deploy the application and the Module with the RevisionEntity data model.

7.1.2 Adding a Related Record to the Revision Entity

To create a custom Revision Entity with a related share Record so as to add complex custom information to the revisions, do the following:

1. Create the related shared Record.

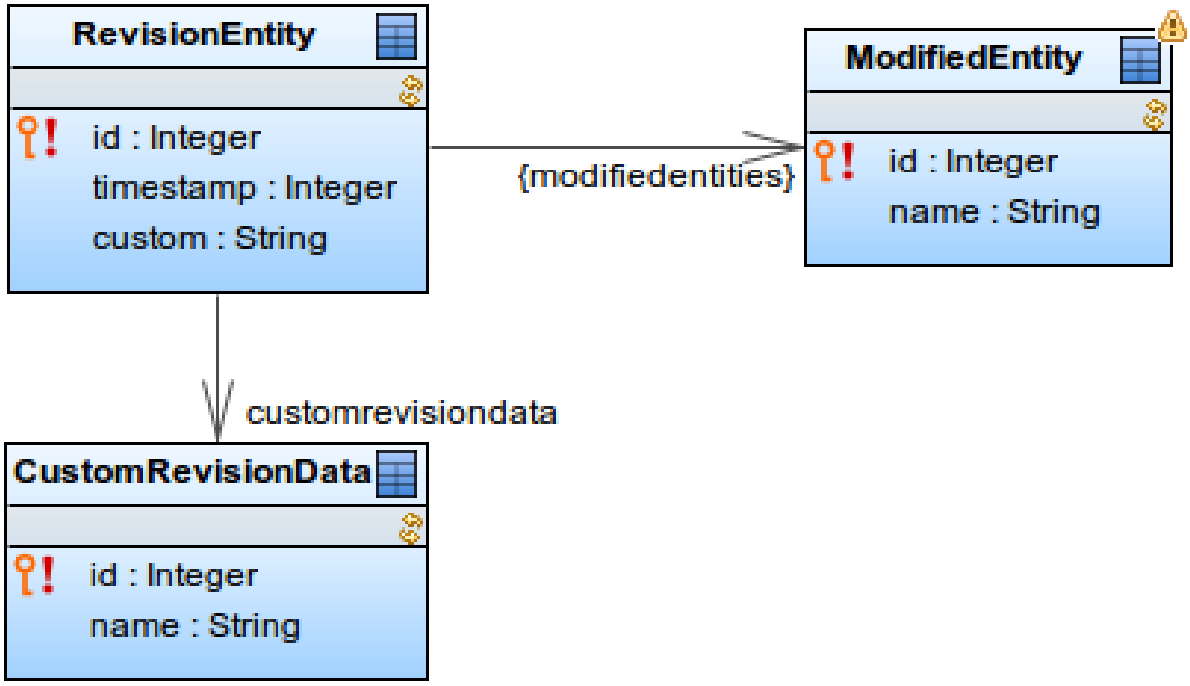


Figure 7.2 Revision entity record with the custom field and a related record

2. Deploy the Module with your Revision records to the server to create their Hibernate entities.

3. Implement the custom RevisionListener:

   (a) In the ejb package of your application, create a class that extends LspsRevisionListener class and implements EJBRevisionListener.

   (b) Override the *newRevision(Object revisionEntity)* method in your RevisionListener class:

   - Call the newRevision() method of LspsRevisionListener on the input Revision Entity: super.new↩ Revision(revisionEntity);
     i. Obtain the Hibernate entity of your Revision Entity Record, for example:

     ```
     SharedRecordContext sharedRecordContext =
       SharedRecordContextProvider.INSTANCE.getSharedRecordContextByJndi("");
     SharedRecordNamingInfo recordNamingInfo =
       sharedRecordContext.getNamingInfoForEntityName(((ExternalRecordEntity) revisionE
     ```

     ii. Obtain the entity name of the property of the Revision record from Hibernate.java; for example: `recordNamingInfo = sharedRecordContext.getNamingInfo↩ ForTableName("<YOUR_RECORD_NAME>");`

     iii. Once you have the hibernate name of the property you need to set, open a hibernate session and set the value of the property:

     ```
     Session session = SharedRecordUtils.getSessionFactory(null).getCurrentSession();
     MapSharedRecordEntity entity = new MapSharedRecordEntity("CustomRevisionData");
     entity.set("NAME", "my custom value");
     session.persist(entity);
     ((MapSharedRecordEntity) revisionEntity).set("S_CUSTOM_REVISION_ENTITY_CUSTOMREVISIONDATA", entity);
     ```

     Important: Including complex logic in your Revision Listener class, such as, contacting an external system to acquire data, might result in performance issues.

4. Register the ejb.

5. In your data model, adjust the Entity Revision shared record to use your implementation and upload the resources.

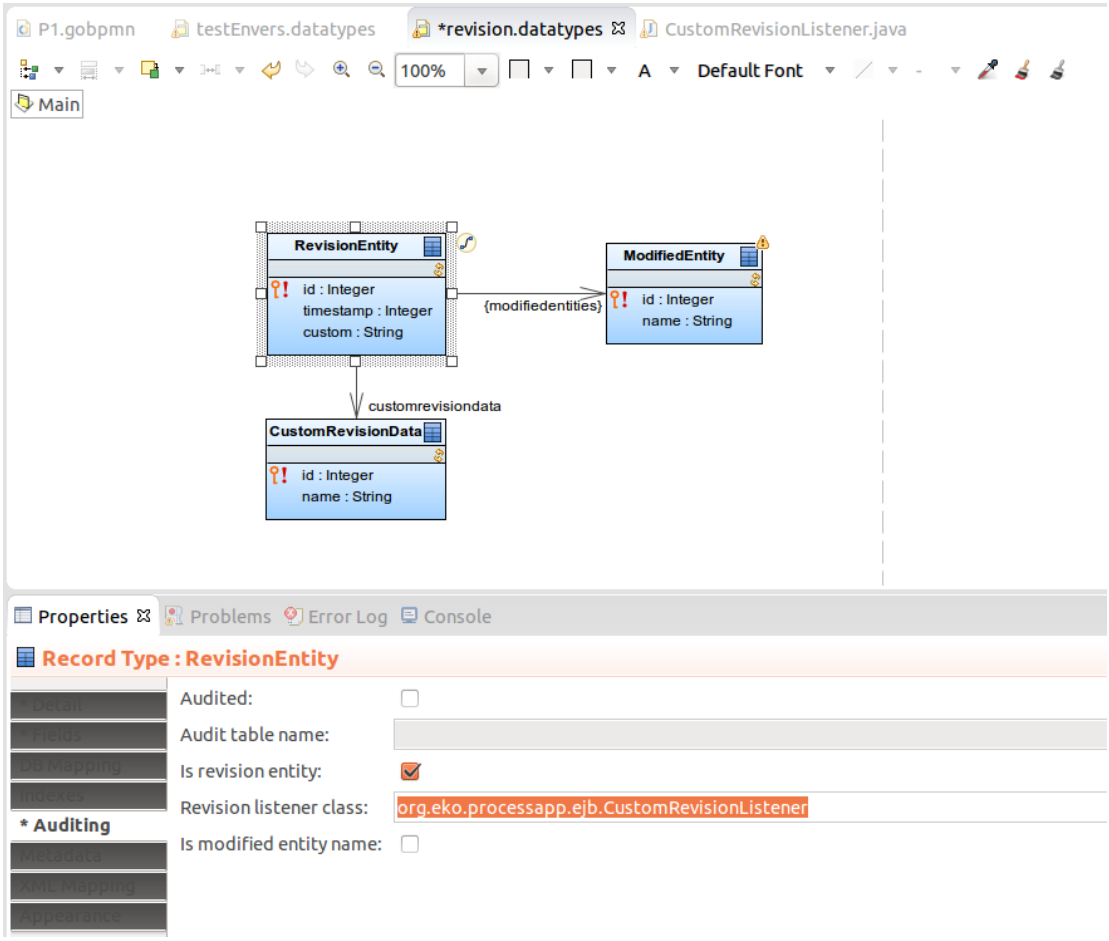


Figure 7.3 Revision Entity shared Record implemented by a custom RevisionListener class

7.1.3 Example Implementation of a Custom Revision Listener

Custom implementation of the RevisionListener must extend the LspsRevisionListener and implement EJBRevisionListener. Once you created your RevisionListener implementation make sure to register it in the *ComponentServiceBean*:

Example of a custom RevisionListener for a RevisionEntity record with a custom field user

```
import java.io.Serializable;
~
import javax.annotation.security.PermitAll;
import javax.ejb.EJB;
import javax.ejb.Stateless;
~
import org.hibernate.Session;
import org.hibernate.envers.RevisionType;
~
import com.whitestein.lsps.common.ejb.SecurityManagerServiceLocal;
import com.whitestein.lsps.common.hibernate.SharedRecordContext;
import com.whitestein.lsps.common.hibernate.SharedRecordContextProvider;
import com.whitestein.lsps.common.hibernate.SharedRecordNamingInfo;
import com.whitestein.lsps.common.hibernate.SharedRecordUtils;
import com.whitestein.lsps.hibernate.envers.EJBRevisionListener;
import com.whitestein.lsps.hibernate.envers.LspsRevisionListener;
```

```
import com.whitestein.lsps.model.sharedrecord.ExternalRecordEntity;
import com.whitestein.lsps.model.sharedrecord.MapSharedRecordEntity;
~
@Stateless
@PermitAll
public class CustomRevisionListener extends LspsRevisionListener implements EJBRevisionListener {


    //injects the security service (for the user field):
    @EJB
    private SecurityManagerServiceLocal securityService;
~
    @Override
    public void newRevision(Object revisionEntity) {
        super.newRevision(revisionEntity);
~
        //persisting into a custom field in the Revision Entity record:
        ((MapSharedRecordEntity) revisionEntity).set("USER", securityService.getPrincipalName());
    }
}
```

Example of a custom RevisionListener for a RevisionEntity record with the related record *CustomRevisionData*

```
import java.io.Serializable;
~
import javax.annotation.security.PermitAll;
import javax.ejb.EJB;
import javax.ejb.Stateless;
~
import org.hibernate.Session;
import org.hibernate.envers.RevisionType;
~
import com.whitestein.lsps.common.ejb.SecurityManagerServiceLocal;
import com.whitestein.lsps.common.hibernate.SharedRecordContext;
import com.whitestein.lsps.common.hibernate.SharedRecordContextProvider;
import com.whitestein.lsps.common.hibernate.SharedRecordNamingInfo;
import com.whitestein.lsps.common.hibernate.SharedRecordUtils;
import com.whitestein.lsps.hibernate.envers.EJBRevisionListener;
import com.whitestein.lsps.hibernate.envers.LspsRevisionListener;
import com.whitestein.lsps.model.sharedrecord.ExternalRecordEntity;
import com.whitestein.lsps.model.sharedrecord.MapSharedRecordEntity;
~
@Stateless
@PermitAll
public class CustomRevisionListener extends LspsRevisionListener implements EJBRevisionListener {


    //injects the security service (for the user field):
    @EJB
    private SecurityManagerServiceLocal securityService;
~
    @Override
    public void newRevision(Object revisionEntity) {
        super.newRevision(revisionEntity);
~
        //persisting into a record related to the Revision Entity record:
~
        //obtain the names of the for properties in the hibernate entity:
        //SharedRecordContext sharedRecordContext = SharedRecordContextProvider.INSTANCE.getShared
        //SharedRecordNamingInfo recordNamingInfo = sharedRecordContext.getNamingInfoForEntityName
        //obtain the name of the hibernate entity for your table:
```

```
        //recordNamingInfo = sharedRecordContext.getNamingInfoForTableName("CustomRevisionData");
~
        MapSharedRecordEntity entity = new MapSharedRecordEntity("CustomRevisionData");
        entity.set("NAME", "xxx");
~
        Session session = SharedRecordUtils.getSessionFactory(null).getCurrentSession();
        session.persist(entity);
~
        ((MapSharedRecordEntity) revisionEntity).set("S_CUSTOM_REVISION_ENTITY_CUSTOMREVISIONDATA'
    }
}
```

Example EJB registration

```
@EJB(beanName = "CustomRevisionListener")
    private EJBRevisionListener customRevisionListener;
~
    @Override
    protected void registerCustomComponents() {
        register(customRevisionListener, CustomRevisionListener.class);
    }
```

Chapter 8

Model JUnit Tests

This part provides information on how to perform JUnit testing on your models.

> **Note:** This document instructs you to generate the Default `LSPS Application` and modify the generated sample testing project. The sample testing project is provided for your convenience as part of the Default LSPS Application but you can create your own classes and projects as well. Mind that if you decide to create your own project, you will need to import the jars required by the testing classes, either manually or using maven, create the `pom.xml` file with the *test* target, and manually create the `test.properties` file.

You can create JUnit tests using the API of the following packages:

- **com.whitestein.lsps.test**: API for model and model instance management, to-do management, and log management
- **com.whitestein.lsps.test.web**: Basic class for testing front-end applications
- **com.whitestein.lsps.test.web.components**: API for testing over form components in front-end applications

> **Important:** To be able to use the API of **com.whitestein.lsps.test.web** and **com.whitestein.lsps.test.web.components**, purchase the Vaadin TestBench license.

For detailed documentation, refer to the Javadoc documentation in the documentation/apidocs directory in PDS.

8.1 Prerequisites

Before you work with the testing resources of the LSPS Application, do the following:

- Enable the modeling IDs so you can identify form components on runtime: make sure your server is running with the `-Dcom.whitestein.lsps.vaadin.ui.debug=true` property (in PDS, go to Runtime Connection > Runtime Connection Settings; select the connection and click Edit).
- If you want to create tests of the UI, purchase the Vaadin TestBench 4 license.

8.2 Creating JUnit Tests

Note: If you do not want to use the resources generated as part of the Default LSPS Application, you can create them manually: either you import all the dependencies or create a custom pom.xml that will pull them for you. After `mvn clean install` and run `mvn eclipse:eclipse` make sure to refresh the GO-BPMN Explorer.

To create a JUnit test in the Application User Interface, do the following:

1. Open the workspace with the Default LSPS Application.
2. Open your `<APPLICATION>-tester` project with the sample test classes:
   - `SampleNonUIIT.java`: a dummy sample test class that uploads a model, creates its instance and evaluates an expression in the context of the Model instance
   - `SampleUIIT.java`: a dummy sample test class that tests execution of To-Dos
   - `test.properties`: relative path to the tested model and to the Standard Library Modules
   - `pom.xml`: Maven POM file with dependencies Since the tests need a running LSPS server, Maven compilation does not run the tests by default. The `pom.xml` file therefore defines the `lsps.tester` parameter, which allows you to run the JUnit tests on compilation:

     ```
     mvn clean install -Dlsps.tester
     ```

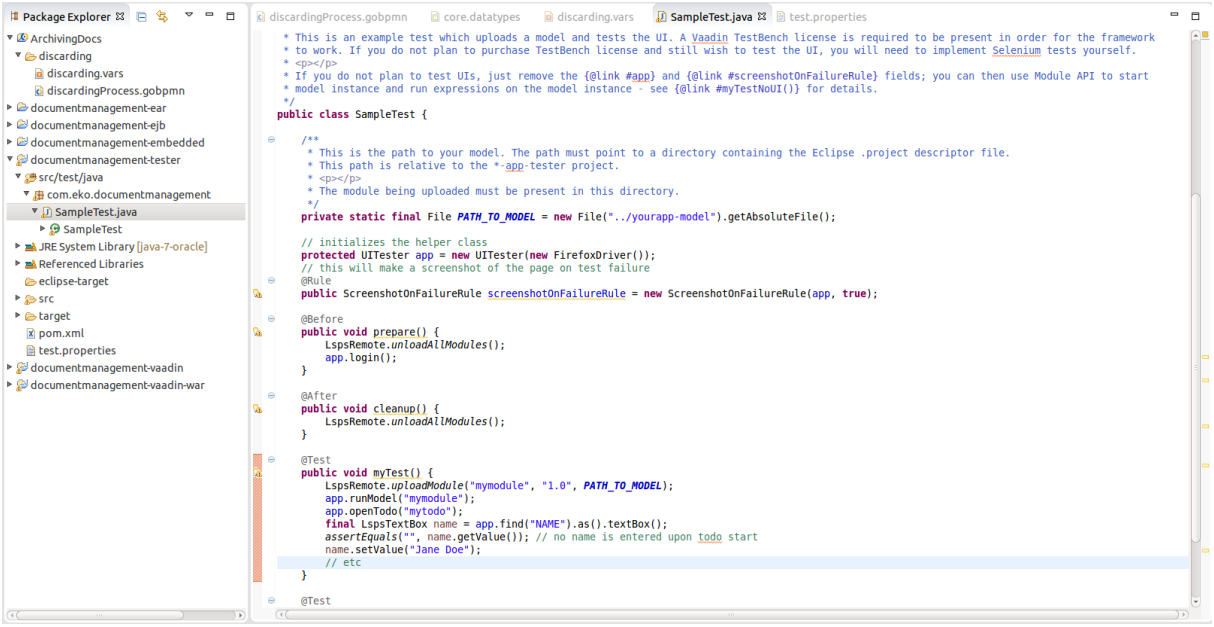


Figure 8.1 SampleTest class in the default LSPS Application

3. Modify or create a testing class:
   (a) Open the application `tester` project.
   (b) Expand the `src` package and open the `SampleNonUIIT.java` or `SampleUIIT.java` file. Note that the testing classes are JUnit 4 tests and hence require annotations, such as `@Before`. The classes provide multiple LSPS-specific testing methods from com.whitestein.lsps.test:

   When testing the GUI, you need to create a UITester instance. For tests that do not require GUI testing, you can perform LspsRemote calls.

(c) Edit the sample test file.

From tests, you can access only the model context: Data from child contexts, such as Sub-Process variables, cannot be used for testing.

4. Edit the `test.properties` file to point to the location with the project with your model if applicable. Optionally, provide paths to libraries.
5. If you have modified the `pom.xml` file or provided paths to custom libraries in `test.properties`, open a terminal/command line and go to the location of the test project:
   (a) synchronize maven and eclipse: run `mvn eclipse:eclipse`
   (b) re-build the maven artifact to acquire the dependencies: run `mvn clean install`.
6. Refresh the GO-BPMN Explorer.

8.3 Running JUnit Tests

To run your JUnit tests of model, do the following:

1. Make sure you have synchronized maven and eclipse (run `mvn eclipse:eclipse`) and re-build the maven artifact to acquire the dependencies (run `mvn clean install`) if necessary.
2. Refresh the GO-BPMN Explorer.
3. Run your LSPS server (typically the SDK Embedded Server with your application using the launcher).
4. Run the test:
   - To run the test on other than the localhost server with port 8080, edit the JVM parameters:
     (a) Go to Run > Run Configurations
     (b) Double-click JUnit and create your configuration for the target JUnit class with the JVM arguments `-Dselenium.host=<SERVER_IP>` and `-Dselenium.port=<SERVER_PORT>`. Make sure you enabled the modeling IDs, that is, your server is running with the `-Dcom.whitestein.lsps.vaadin.ui.debug=true` property.
   - In PDS, right-click the Java test class and click Run As JUnit Test.

     Alternatively, on the command line, go to the location of the application tester project and run `mvn clean install -Dlsps.tester`. Make sure you enabled the modeling IDs, that is, your server is running with the `-Dcom.whitestein.lsps.vaadin.ui.debug=true` property.

Chapter 9

Integration

9.1 Mail Server Configuration of the SDK Embedded Server

To configure the SMTP settings of the SDK Embedded Server of the Default Application User Interface, set the respective properties in the<APP>-embedded/conf/conf/conf.d/openejb.xml file:

```
mail.transport.protocol=smtp
mail.smtp.host=mailsmtp.whitestein.com
mail.smtp.port=25
mail.from=lsps@whitestein.com
mail.smtp.user=lsps_user
mail.smtp.auth=true
mail.smtp.starttls.enable=true
mail.smtp.password=<PASSWORD>
password=<PASSWORD>
```

> **Important:** The SDK Embedded Server fails to communicate with an SMTP server that requires MD5Digest authentication. This causes the sendEmail() calls to fail. To force another authentication method, specify the following property in the <APP>-embedded/conf/conf/conf.d/openejb.xml file: 'mail.smtp.sasl.mechanisms=PLAIN

> **Note:** For configuration of mail sessions for the PDS Embedded Server, which is a WildFly server, use the server web console at localhost:9990/management. For configuration of other servers, refer to the section for installation of your server in the Deployment and Configuration guide.

9.2 LDAP

The default Application User Interface uses its custom person management. The related services are implemented in the pm-exec.jar in the application bundle. If you want the application to use an LDAP server for authentication and authorization, you need to provide your implementation of the person management services in a custom 'pm-ldap-exec.jar' file.

To set your LSPS application to use LDAP on your application server, do the following:

1. In your application, create the pm-ldap-exec ejb project.
2. Implement the following beans in the project:

- `PersonManagementServiceBean` stateless bean that implements the following interfaces:
  - `com.whitestein.lsps.os.ejb.PersonManagementServiceLocal`
  - `com.whitestein.lsps.os.ejb.PersonManagementServiceRemote` (optional)
- ProcessServiceBean stateless bean that implements the following interfaces:
  - `com.whitestein.lsps.os.ejb.PersonServiceLocal`
  - `com.whitestein.lsps.os.ejb.PersonServiceRemote` (optional)
- `PersonSecurityRoleChangePlugin` stateless bean that implements the following interfaces:
  - `com.whitestein.lsps.orgstructure.entity.SecurityRoleChangePlugin`

3. In your `pom.xml` of your EAR project, change the dependency.

```
<dependency>
    <groupId>com.whitestein.lsps.person-management</groupId>
    <artifactId>lsps-pm-ldap-exec</artifactId>
</dependency>
```

4. Rebuild and deploy your application.

Chapter 10

Building the LSPS Application

Once you have customized the LSPS Application as required, you can test the customization on the SDK Embedded Server or build the EAR and deploy it a supported application server.

10.1 Building and Running the LSPS Application for Development Purposes

To simplify the development and customization of the LSPS Application, the SDK comes with a launcher for the SDK Embedded Server: the launcher is created when you generate the LSPS Application along with Maven build configuration for the application. The launcher configuration runs the SDK Embedded Server with an exploded deployment of your application on its classpath (The configuration runs the `main()` method in the `<APP_PACKAGE>.embedded.LSPSLauncher` class).

Hence to build your application and run it on the SDK Embedded Server, do the following:

1. Build your application:
   (a) Expand the context menu on the External Tools button ( ).
   (b) Click `<APP_NAME> Maven build`.
2. Run the SDK Embedded Server launcher:
   (a) Expand the context menu on the External Tools button ( ).
   (b) Click `<APP_NAME> Embedded Server Launcher`.

   Important: When you generated the LSPS Application, the system generated the SDK Embedded Server with LSPS and its launch configuration. Note that this is a different server from the PDS Embedded Server. Check the referenced libraries of the `<APP>-embedded` project for the server libraries.

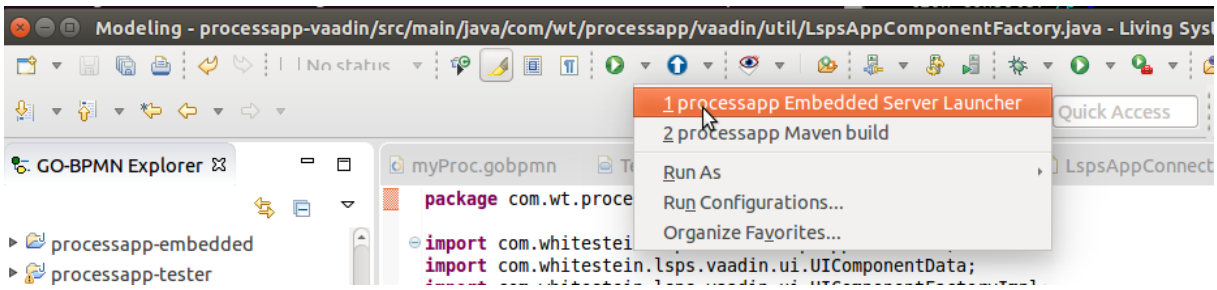


Figure 10.1 Running the LSPS Application with the generated launcher

10.2 Building the LSPS Application for Deployment

To build the application EAR so you can deploy it on a supported application server, run `mvn clean install` in the directory with the root pom.xml. Consider running the build with the provided tests with `mvn clean install -Dlsps.tester`. Amongst other tests, the build will also check whether the *jboss-deployment-structure.xml* contains the required modules.

The output EAR file is located in the target directory. To deploy follow the `deployment instructions for your server`.

Consider preparing database update scripts. You might want to make use of the `database schema update scripts generated on module upload`.

Chapter 11

Performance Tuning

The following chapter provides basic information on how to improve performance of your Application User Interface.

11.1 Pre-Loading Modules

To compile modules on server restart and prevent their compiling later, you can pre-load the modules: On server start, the server will compile the modules defined by the INITIAL_MODELS_SQL entry in the LSPS_SETTINGS database table.

To enable and define pre-loading of modules when the server is restarted, do the following:

1. Set SQL logging level of Hibernate to `ALL`.

   For the embedded Wildfly server, add `org.hibernate.SQL.level=ALL` to `<LSPS_WORKSPACE>\.LSPSEmbedded\wildfly<VERSION>\standalone\configuration\logging.properties`.

2. Log in the LSPS database as administrator.

   On the default H2 database, the DB URL is `//localhost/h2/h2;MVCC=TRUE;LOCK_TIMEOUT=60000`, the user name and password are `lsps`.

3. In the LSPS_SETTINGS table, add the INITIAL_MODELS_SQL key and the value with a select. The select must return MODEL_ID entries you want to pre-load.

```
INSERT INITIAL_MODELS_SQL into LSPS_SETTINGS (ID,VALUE) VALUES ('INITIAL_MODELS_SQL','select distinct ID
       from LSPS_MODULES where CREATE_PROCESS_LOG = true;');
```

11.2 Setting Dumping of Model Instances with Exceptions

If a model instantiation fails with an exception (it interpretation fails), the model instance data is lost. This behavior prevent possible performance issues due to too large model instances with exceptions. However, you enable storing of the errors with the marshalled model instances in the database.

To enable the setting, perform the following insert on the LSPS database:

```
INSERT into LSPS_SETTINGS(ID,VALUE) values ('DUMP_MODEL_INSTANCE_ON_EXCEPTION','true')
```

11.3 Decreasing Frequency of Goal-Condition Checks

If a model contains many goals with conditions, checking of the conditions might cause performance issues since the conditions are checked whenever any of the goals changes its status or a token is moved. This happens when the interpretation strategy is set to `FULL_PARALLEL`. However, the default setting is `BPMN_FIRST`.

You can set the checking to happen only when all transactions finish (all tokens are moved as far as possible).

To do so, issue the following insert statement on the LSPS database:

```
INSERT into lsps_settings (id, value) values('INTERPRETATION_STRATEGY','BPMN_FIRST')
```

11.4 Improving Shared Record Search

To improve performance of Shared Record search, define indexes on the Records and their Relationships.

11.5 Disabling System Cache Regions

Check the setting of cache regions in the `<YOUR_APP>-ejb/src/main/resources/cache-regions.properties` file of your LSPS application.

11.6 Disabling Hibernate Statistics

Make sure that the Hibernate statistics feature is deactivated: you can do so on the MBean tab of JConsole, the node *com.whitestein.lsps* > *Statistics* -> *Attributes* > *StatisticsEnabled*

Chapter 12

Appendix: Creating Custom Form Component with Custom Events

To create a custom form component, you need to do the following:

1. Create the implementation:
   (a) Implement your component as a UIComponent subclass and implements UIComponent
   (b) Implement a custom event as a UIEvent subclass.
2. Create the component support for PDS:
   (a) Create the data type model with records of your component, your event and any related data types.
   (b) Create the custom component definition with the data type of the component and relevant properties.
3. Create a form with the listener to test your component.

Implementing a Custom Event

In the vaadin project, create a class for your event:

- It must implement the UIEvent.
- The constructor must have as its second parameter the relevant data type.
  The parameter can be based on the record related to the component record.
- Override the `getEventProperties()` method so it returns a hashmap of the custom event properties.

```
package com.whitestein.colorpicker.vaadin.util;
~
import java.util.HashMap;
import java.util.Map;
~
import com.vaadin.shared.ui.colorpicker.Color;
import com.whitestein.lsps.lang.Decimal;
import com.whitestein.lsps.lang.exec.RecordHolder;
import com.whitestein.lsps.vaadin.ui.components.UIComponent;
import com.whitestein.lsps.vaadin.ui.events.UIEvent;
~
```

```
public class UIColorPickEvent extends UIEvent {
~
  private final Color newColor;
~
  public UIColorPickEvent(UIComponent component, Color newColor) {
    super(component, colorpicker::ColorPickEvent);
    this.newColor = newColor;
  }
~
  private static RecordHolder toColor(UIComponent context, Color color) {
    final RecordHolder c = context.getComponentData().getScreen().getScreenContext().getNamespace
    c.setProperty("r", new Decimal(color.getRed()));
    c.setProperty("g", new Decimal(color.getGreen()));
    c.setProperty("b", new Decimal(color.getBlue()));
    c.setProperty("a", new Decimal(color.getAlpha()));
    return c;
  }
~
  @Override
  //creates java hashmap -> fieldname to value; then creates the uicolorpickevent recordholder;
  protected Map<String, ?> getEventProperties(UIComponent component) {
    final Map<String, Object> result = new HashMap<String, Object>(super.getEventProperties(compon
    result.put("color", toColor(component, newColor));
    return result;
  }
}
```

Implementing Custom Component

To create an example custom component, create a class implementing your component:

- It must implement UIComponent.
- It must define a constructor with UIComponentData as its input argument.

  Make sure the UIComponentData is defined as a class variable, so you can use it in the getComponentData method.

  - Create the custom listener

    The listener should override the method that creates the event on the component, in the example color↩ Changed(ColorChangeEvent e), so the event is transformed into our custom event and then fired and enters the event queue when UIComponents.fireAndProcess() method is called.
  - Register the component with Vaadin components.
  - Create listeners and context for the component (UIComponents.afterCreate(this)).

```
public UIColorPicker(UIComponentData data) {
  this.data = data;
  ColorChangeListener listener = new ColorChangeListener() {
 
    @Override
    public void colorChanged(ColorChangeEvent event) {
      final Color newColor = event.getColor();
      UIComponents.fireAndProcess(new UIColorPickEvent(UIColorPicker.this, newColor));
    }
 
  };
  //registered to vaadin's color picker
  addColorChangeListener(listener);
  UIComponents.afterCreate(this);
```

- Defines the refresh() method for the component.

  The method is called when the component is refreshed.

```
@Override
public void refresh() {
  Variant.RecordVariant color = Variant.definitionOf(this).getPropertyValue("color").closure()
      .inScope(this).call().record();
  color.checkType("colorpicker::Color").checkPresent();
  setColor(toColor(color));
}
 
private static Color toColor(Variant.RecordVariant color) {
  return new Color(color.getPropertyValue("r").decimal().get().intValue(),
      color.getPropertyValue("g").decimal().get().intValue(),
      color.getPropertyValue("b").decimal().get().intValue(),
      color.getPropertyValue("a").decimal().or(new Decimal(255)).intValue());
  }
```

- Implement the getComponentData() method.

```
package com.whitestein.colorpicker.vaadin.util;
 
import com.vaadin.shared.ui.colorpicker.Color;
import com.vaadin.ui.ColorPicker;
import com.vaadin.ui.components.colorpicker.ColorChangeEvent;
import com.vaadin.ui.components.colorpicker.ColorChangeListener;
import com.whitestein.lsps.lang.Decimal;
import com.whitestein.lsps.vaadin.ui.UIComponentData;
import com.whitestein.lsps.vaadin.ui.components.UIComponent;
import com.whitestein.lsps.vaadin.ui.events.UIEvent;
import com.whitestein.lsps.vaadin.util.UIComponents;
import com.whitestein.lsps.vaadin.util.Variant;
 
public class UIColorPicker extends ColorPicker implements UIComponent {
 
  private final UIComponentData data;
 
  public UIColorPicker(UIComponentData data) {
    this.data = data;
    ColorChangeListener listener = new ColorChangeListener() {
 
      @Override
      public void colorChanged(ColorChangeEvent event) {
        final Color newColor = event.getColor();
        UIComponents.fireAndProcess(new UIColorPickEvent(UIColorPicker.this, newColor));
      }
 
    };
    //registered to vaadin's color picker
    addColorChangeListener(listener);
    UIComponents.afterCreate(this);
  }
 
  @Override
  public void refresh() {
    Variant.RecordVariant color = Variant.definitionOf(this).getPropertyValue("color").closure()
        .inScope(this).call().record();
    color.checkType("colorpicker::Color").checkPresent();
    setColor(toColor(color));
  }
```

```
 
  private static Color toColor(Variant.RecordVariant color) {
    return new Color(color.getPropertyValue("r").decimal().get().intValue(),
        color.getPropertyValue("g").decimal().get().intValue(),
        color.getPropertyValue("b").decimal().get().intValue(),
        color.getPropertyValue("a").decimal().or(new Decimal(255)).intValue());
  }
 
  @Override
  public UIComponentData getComponentData() {
    return data;
  }
}
```

Registering Component in Component Factory

Modify the `LspsAppComponentFactory` class: uncomment the `createComponent` method and add the constructor call for your component that is called when the respective Record is requested.

```
package org.eko.ekoapp.vaadin.util;
 
import org.eko.ekoapp.vaadin.components.UIText;
 
import com.whitestein.lsps.vaadin.LspsAppConnector;
import com.whitestein.lsps.vaadin.ui.UIComponentData;
import com.whitestein.lsps.vaadin.ui.UIComponentFactoryImpl;
import com.whitestein.lsps.vaadin.ui.components.UIComponent;
 
public class MyComponentFactory extends UIComponentFactoryImpl {


    public MyComponentFactory(LspsAppConnector connector)
            throws NullPointerException {
        super(connector);
    }
 
    @Override
    protected UIComponent createComponent(UIComponentData componentData) {
        String type = componentData.getComponentDefinition().getType()
                .getFullName();
        if (type.equals("customComponentModule::TextComponentRecord")) {
            return new UIText(componentData);
        }
        return super.createComponent(componentData);
    }
}
```

Creating Custom Component Support for PDS

To create a support for your component in PDS, do the following:

1. Create a data type model that reflects the components, events, and any related data types defined in your application: make sure their are located in the module and have the name defined in their implementation (in the example, the color picker, color, and color-change listener).

   Note that a data type must have the correct super types:

- A Listener type must have ui::Listener or its subtype as its super type.
- A Component type must have ui::UIComponent or its subtype as its super type.
- An Event type must have ui::Event or its subtype as its super type. It contains the field `source` that holds the component that produced the event and a field with the data the implementation requires.

2. Create the custom component definition: Use the component record as the implementation (refer to Application User Interface Forms Guide).
3. When using the component, define the listener as an expression.

```
new colorpicker::ColorPickListener(
  refresh -> {a->{PICKER}},
  handle -> {e:ColorPickEvent-> color:=e.color; debugLog({->"Color was picked. " + e})}
)
```