WHITESTEIN
Technologies

Living Systems® Process Suite

Customizing the LSPS Application

Living Systems Process Suite Documentation

3.2
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 Main Page **1**

1.1 Architecture Overview . . . 1

2 Setup **3**

2.1 Recommended Structure . . . 3

2.2 Generating the LSPS Application . . . 4

2.2.1 Generating the LSPS Application from PDS . . . 5

2.2.2 Generating the LSPS Application from the Command Line . . . 8

2.3 Importing an LSPS Application to PDS Workspace . . . 9

2.4 Configuring SDK Embedded Server . . . 10

2.4.1 Configuring Mail Server of SDK Embedded Server . . . 10

2.4.2 Configuring Data Source of SDK Embedded Server . . . 11

2.4.2.1 Deleting the System Database of SDK Embedded Server . . . 11

3 Customization **13**

3.1 Customizing Application User Interface . . . 13

3.1.1 Customizing a Theme . . . 13

3.1.1.1 Creating a Custom Theme . . . 14

3.1.1.2 Adding a Sass Rule . . . 15

3.1.1.3 Compiling a Modified Theme . . . 15

3.1.1.4 Modifying General Theme Settings . . . 16

3.1.1.5 Setting the Default Theme . . . 17

3.1.2 Customizing Behavior . . . 17

3.1.2.1 Browser Session Timeout . . . 17

3.1.2.2 Customizing Authentication . . . 17
3.1.2.3 Setting the Home Page . . . 17
3.1.2.4 Importing JavaScript . . . 18
3.1.3 Customizing Content . . . 18
3.1.3.1 Adding an Item to the Navigation Menu . . . 19
3.1.3.2 Adding a Header and Footer . . . 19
3.1.3.3 Customizing the Login and Logout Page . . . 19
3.1.3.4 Customizing Content of the About Dialog . . . 20
3.1.3.5 Adding a Locale . . . 20
3.1.3.6 Creating a Custom Page . . . 21
3.2 Creating a Custom Object . . . 23
3.2.1 Custom Functions and Task Types . . . 24
3.2.1.1 Creating a Function . . . 24
3.2.1.2 Creating a Task Type . . . 33
3.2.1.3 Custom Objects as EJBs . . . 40
3.2.1.4 Using Entities . . . 41
3.2.2 Custom Form and UI Components . . . 41
3.2.2.1 Creating a UI Component . . . 42
3.2.2.2 Creating a Forms Component . . . 51
3.3 Working with a Model . . . 62
3.3.1 Execution Levels . . . 62
3.3.1.1 Creating an Execution Level . . . 63
3.3.1.2 Merging an Execution Level . . . 63
3.3.1.3 Cleaning an Execution Level . . . 63
3.3.1.4 Checking for Changes on an Execution Level . . . 64
3.3.2 Creating a Record . . . 64
3.3.2.1 Generating Classes and Interfaces for Records . . . 64
3.3.2.2 Checking a Record Constraint . . . 67
3.3.3 Throwing a Signal . . . 67
3.3.4 Throwing an Error . . . 68
3.3.5 Creating Hooks on Model Execution . . . 69
3.3.6 Invoking the Command-Line Console . . . 69
3.4 Customizing Entity Auditing . . . 70
3.4.1 Adding a Field to the Revision Entity . . . 70
3.4.2 Adding a Related Record to the Revision Entity . . . 71
3.4.3 Example Implementation of a Custom Revision Listener . . . 73

4 Build **77**

4.1 Building and Deploying LSPS Application during Development . . . 77

4.2 Building the LSPS Application EAR . . . 79

4.3 Dependency Management . . . 79

4.3.1 Adding a Module in the Build . . . 79

4.3.2 Adding Dependencies . . . 80

4.3.3 Removing Dependencies . . . 81

5 Tests **83**

5.1 Prerequisites . . . 83

5.2 Running JUnit Tests . . . 84

5.3 Creating JUnit Tests . . . 84

5.4 Testing Record Values . . . 85

6 Integration **87**

6.1 Implementing a Custom Person Management . . . 87

6.2 Adding an MXBean . . . 88

6.3 Accessing Data from other Data Sources . . . 89

7 Preparing Updates and Upgrades **93**

7.1 Preparing Module Update . . . 93

7.2 Update of the LSPS Application . . . 94

7.2.1 Preparing LSPS Upgrade to a New Patch Version . . . 94

7.2.2 Preparing LSPS Upgrade to a New Minor or Major Version . . . 95

Chapter 1

Main Page

To meet the requirements of your business, generally you need to not only create models of your processes and related resources in your BPMN solution, but also customize the look and execution logic of the LSPS Application or its parts.

PDS comes with the SDK extension, which allows you to generate the LSPS Application and setup the development environment easily.

The application exposes the API so you can implement custom business logic, customize the layout, appearance, and content of the Application User Interface, which is the front-end process application, as well as write automated tests and develop integration layers.

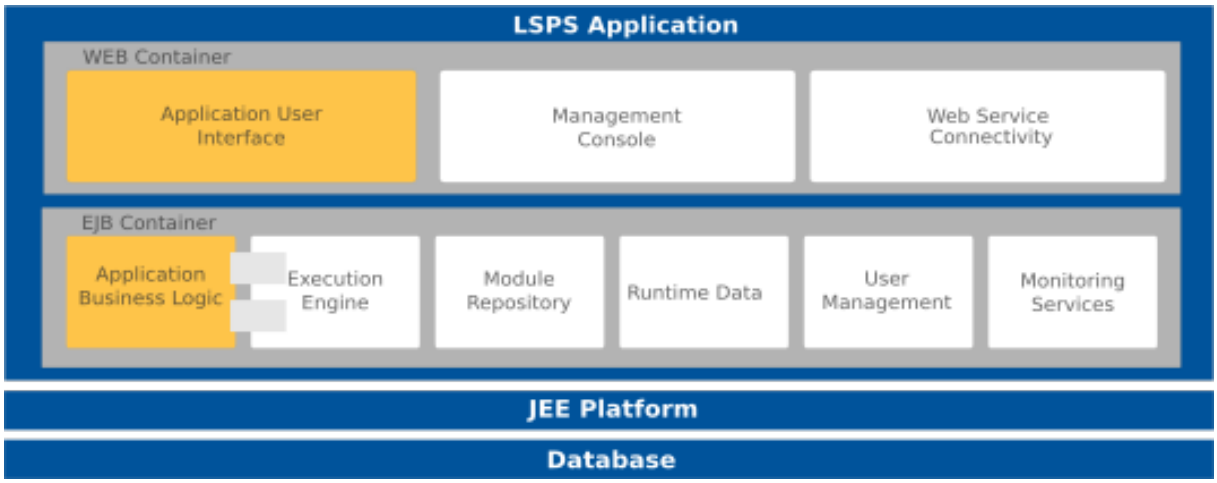


Figure 1.1 LSPS Application architecture with customizable parts highlighted in orange

1.1 Architecture Overview

The LSPS Application is a standard JEE application and comprises the following:

- The EJB container parts:
  - **Module Repository** stores compiled modules.

  - **Execution Engine** manages model instances based on their model.
  - **Runtime Data** holds the runtime data of model instances.
  - **User Management** manages users, roles, and rights.
  - **Web Services** provides server-management API.

- The web container parts:
  - `Application User Interface` provides front-end for users to interact with the execution.
  - `Management Console` allows administrators to manage the LSPS Application resources from their browser.

Chapter 2

Setup

To set up your environment, you will first of all need either to generate sources of a new LSPS Application or import an existing LSPS Application.

The application sources come in a set of projects with the exposed API and maven dependencies:

- `ear`: project for building the LSPS Application EAR archive
- `ejb`: Java classes implementing custom items
- `embedded`: SDK Embedded Server files with the embedded-server launcher
- `tester`: JUnit testing resources
- `vaadin`: web application resources with JavaScript and Java classes
- `vaadin-war`: project for building the WAR archive with presentation resources including Vaadin themes.

2.1 Recommended Structure

Consider keeping the LSPS Application and related resources in a dedicated directory, for example, `java`, and your modules in another directory, to allow for better manageability.

Example project structure

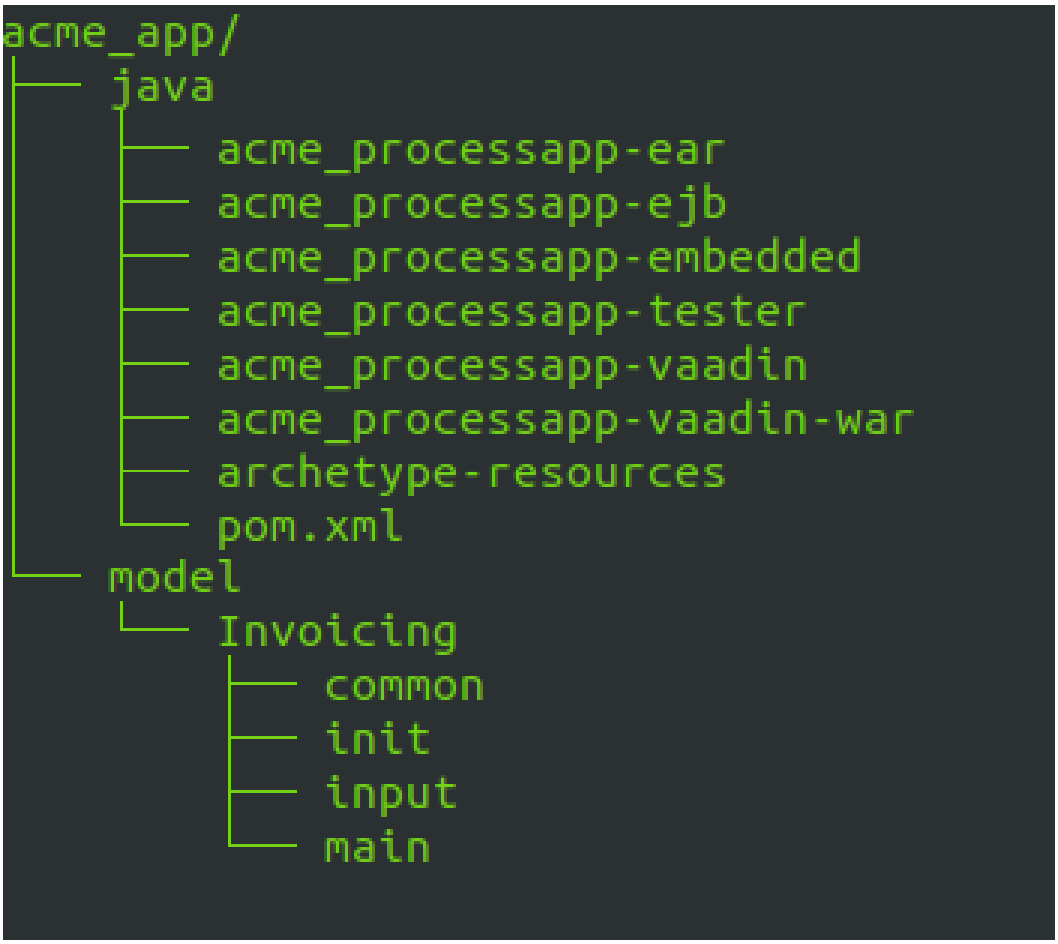


Figure 2.1 Example project structure

2.2 Generating the LSPS Application

With `LSPS SDK installed`, you can generate the LSPS Application

- directly from PDS: this will add launchers for maven build and SDK embedded server to your PDS; However, the application is generated directly in the workspace.
- from the command line: the application sources are separated from workspace resources; Once generated, you can be import the application to a new workspace.

Before you generate or import an *LSPS Application*, make sure you have met the following requirements:

- Living Systems® Process Suite Enterprise Edition *with SDK* is installed.

  If you have not installed SDK, reinstall Process Suite with the SDK option selected.
- **Maven and the Maven repository** are available.

  You probably have set up the Maven repository as part of the installation.

- PDS has the *M2_REPO classpath variable* defined.

  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.

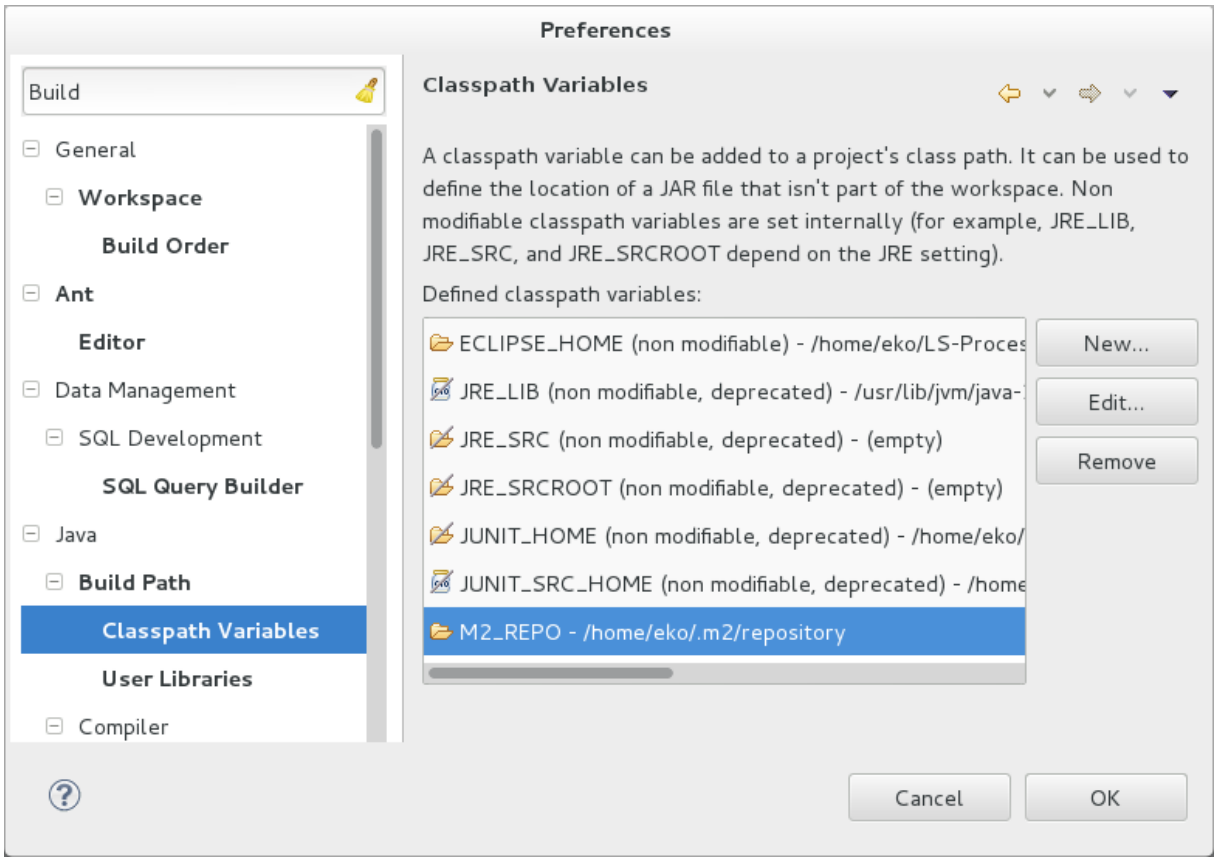


Figure 2.2 M2_REPO variable set up

- The Application User Interface must be built with Oracle JDK 1.8.

The Javadoc API documentation of the LSPS Application is available in the `<PDS_HOME>/documentation/apidocs` directory and in the newest minor version `online`.

2.2.1 Generating the LSPS Application from PDS

SDK of the PDS allows you to generate the LSPS Application, which are sources of the Application User Interface that expose its API, an SDK Embedded Server, and build and launch configurations. This setup allows you to modify the sources of the application, run the server, and build and deploy the application on-the-fly.

> **Important:** The *SDK Embedded Server* with the generated run configuration called `<YOURAPP> Embedded Server Launcher`; is a different server form the *PDS Embedded Server*.

To generate the application and resources in PDS, do the following:

1. Consider using the `recommended structure` and open your workspace which will hold the LSPS Application in a dedicated folder, for example, `java` folder to isolate it from your models.

2. Go to File > New > Other

3. In the New dialog box, select **LSPS Application** and click **Next**.

4. In the updated dialog, enter the details of your application.

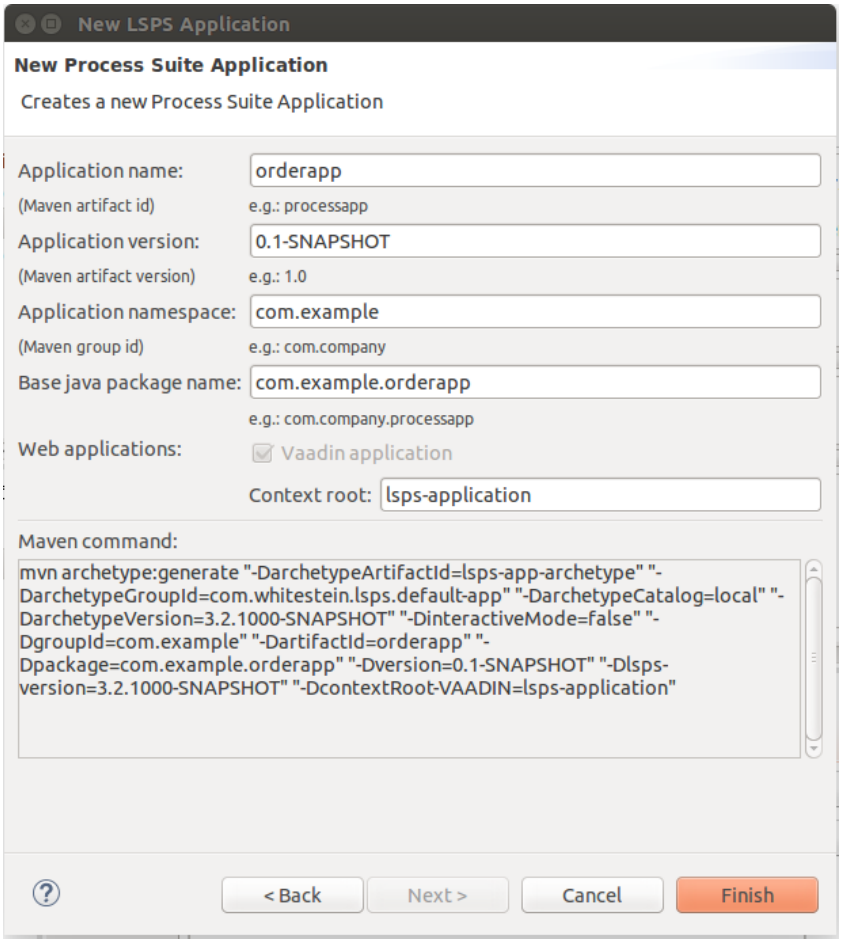


Figure 2.3 Application details

Alternatively, switch to Java perspective and go to New > LSPS Application.

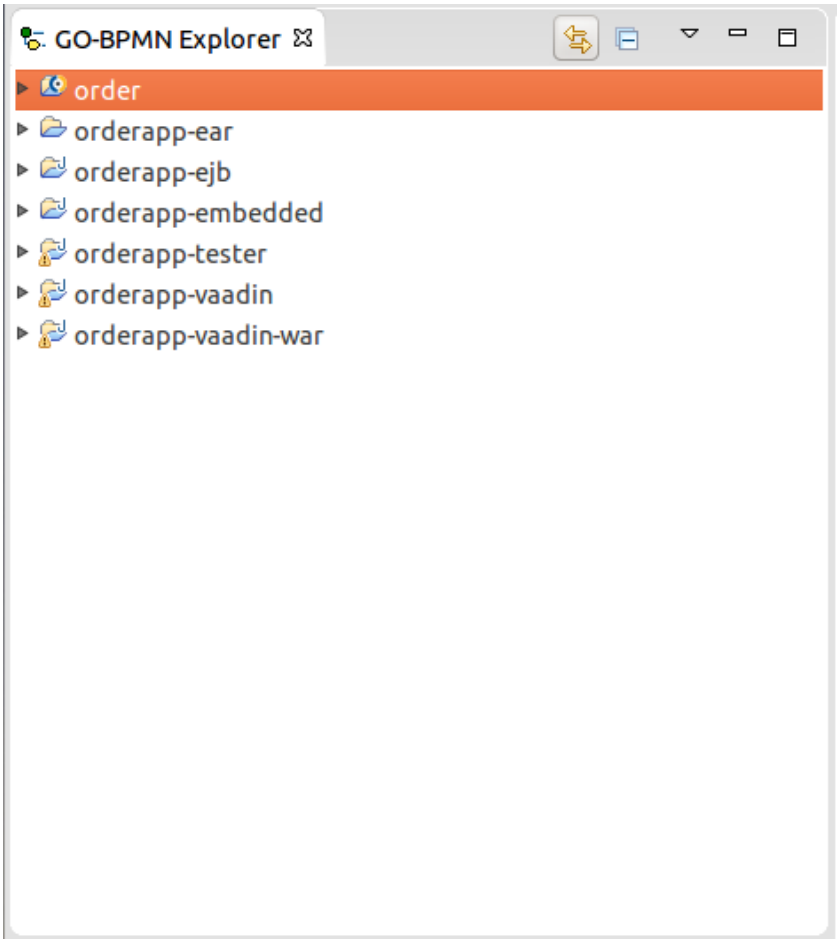


Figure 2.4 Generated application sources

5. Set up version control over the application folder:

   - Create an initial commit: it will separate your customization from the initial default application. When upgrading to new LSPS, you will apply all commits apart from this initial commit on the new application (refer to Preparing LSPS Upgrade to a New Minor or Major Version).

   - Consider ignoring the `.project` directories of the java project of the application: These are generated by maven and differ depending on your environment. Note that is not the case for `.project` directories in modules: these are not generated by maven and must be tracked by your version control system.

The generated application is part of your PDS workspace: It is recommended to remove the workspace directories, create a new workspace in another location, and import the application into this new workspace. Also consider the recommended source structure.

You can customize the application and easily re-build and it on SDK Embedded Server to check the results.

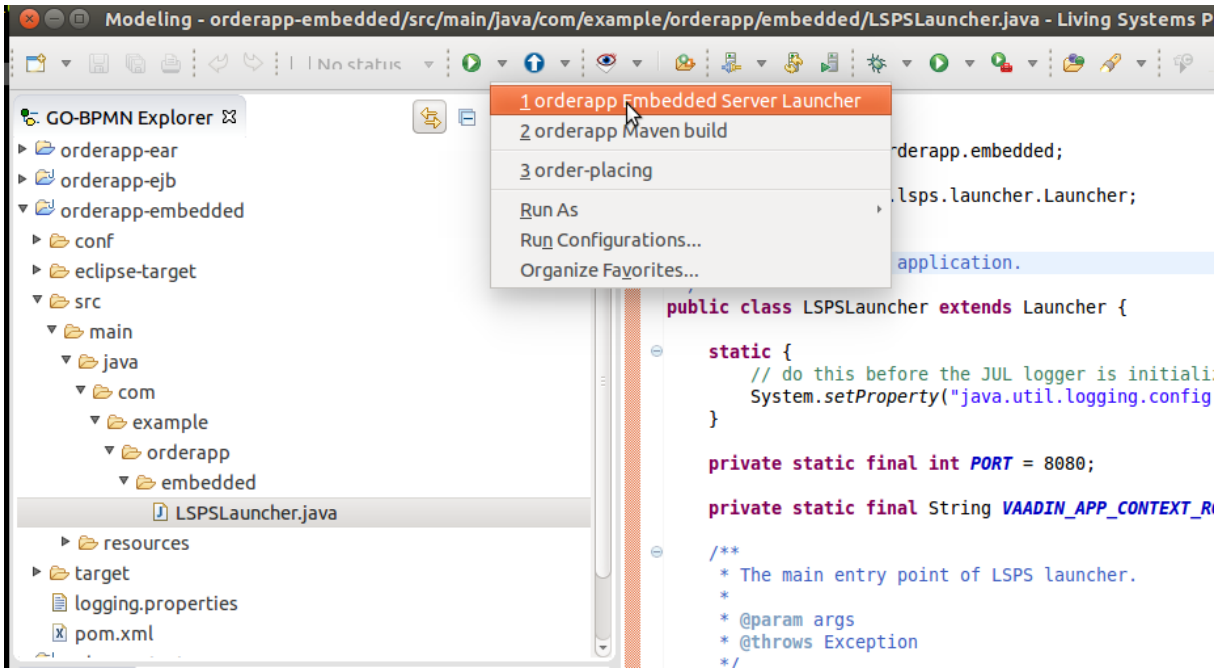


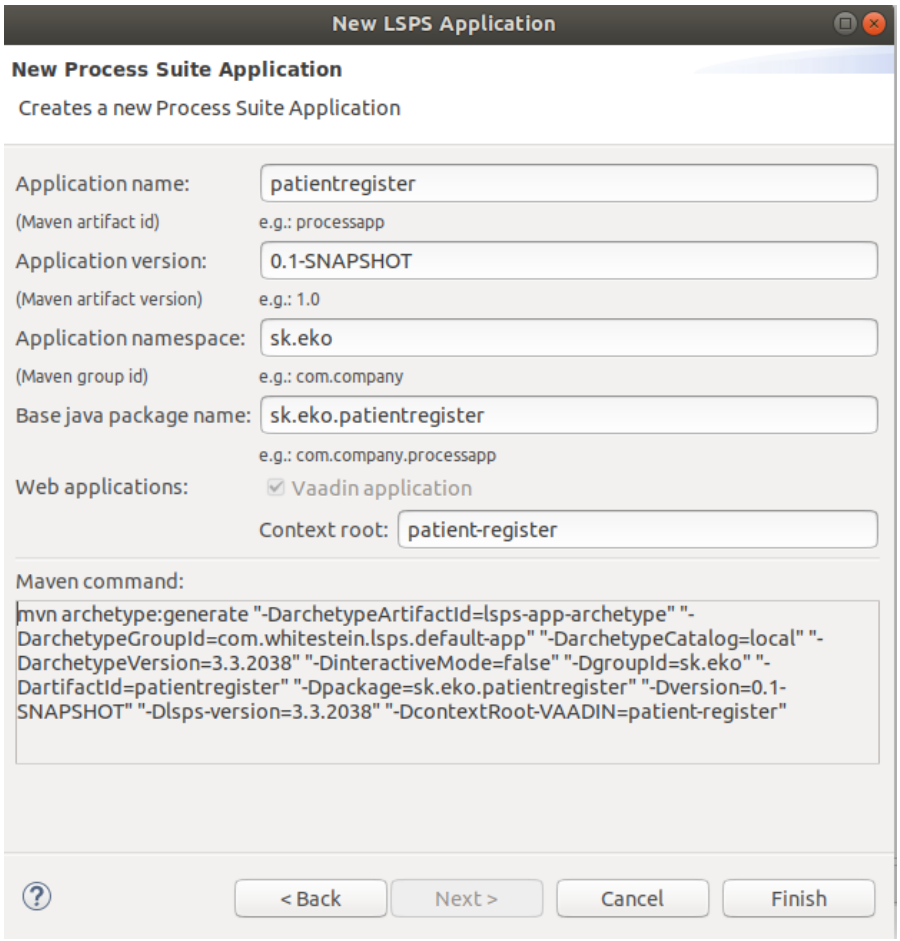
Figure 2.5 Launching SDK Embedded Server with the LSPS Application

2.2.2 Generating the LSPS Application from the Command Line

1. Create a directory for the application and models on your filesystem (refer to Recommended Structure).

2. In a directory, create a directory for the application, for example, a `java` folder to isolate the application from your models.

3. In the *java* folder, generate the `lsps-app-archetype` artifact: To get an example for the maven command, open PDS, go to **File** > **New** > **Other** and locate LSPS application in the popup. Note that the custom archetype is not available in the central maven repo: it is installed into either your local repo or the LSPS maven repo when installing PDS with SDK.



4. Put the main directory (parent of java) under version control:
   - Create an initial commit: In the future, when you will be migrating to a newer version of the application, you will apply all commits starting from the next commit on the new application version (refer to Preparing LSPS Upgrade to a New Minor or Major Version).
   - Consider ignoring the `.project` directories of the java project of the application: These are generated by maven and differ depending on your environment. Note that is not the case for `.project` directories in modules: these are not generated by maven and must be tracked.
5. In the `java/<LSPS_APP>/` directory, generate eclipse resources with `mvn eclipse:eclipse`.
6. Import the application to PDS workspace.

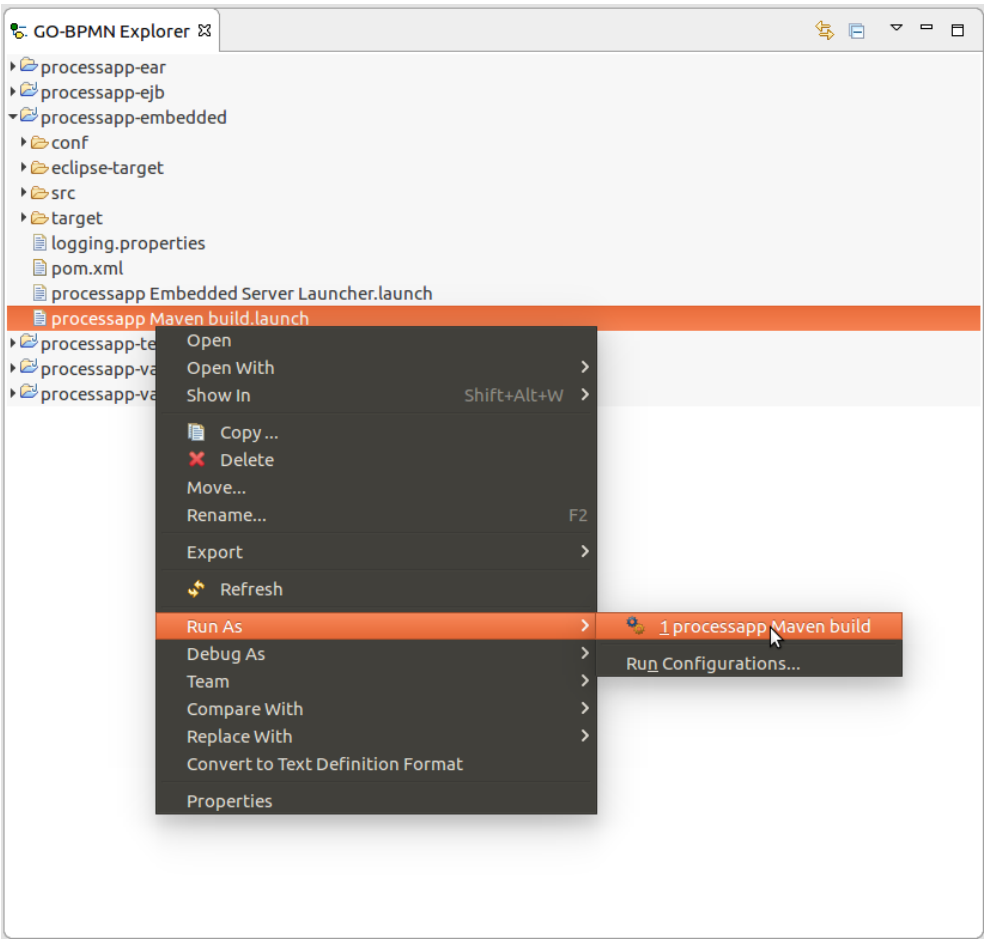
2.3 Importing an LSPS Application to PDS Workspace

To import an existing LSPS Application, do the following:

1. On the command line, go to your application root.
2. Optionally, clean up the file system: run `mvn clean eclipse:clean`.
3. If required, rebuild the eclipse resources: run `mvn eclipse:eclipse`.
4. Open PDS.
5. Define the M2_REPO classpath variable: 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, typically `~/.m2/repository/`.

6. Go to **Import** > **Existing Projects into Workspace** and select the root directory with the application projects.
7. If your projects contain errors, refresh the projects: select all project, right-click the selection and click **Refresh**. Also consider cleaning the projects: go to **Project** > **Clean**.

   You can clean up and build your application from the command line with `mvn clean eclipse:clean install eclipse:eclipse`. and refresh the resources in your workspace, or you can use the Maven build launcher:



Note If the imported application was generated from PDS, not from the command line, you can build the application with the the maven build launcher configuration in the `<YOUR_APP>-embedded` project: right-click it and select **Run As** > <**YOUR_APP**> **Maven Build**. Next time you can run the build the application from the **Run** menu or from the drop-down of the **Run** icon  on the toolbar.

2.4 Configuring SDK Embedded Server

2.4.1 Configuring Mail Server of SDK Embedded Server

To configure the SMTP settings of SDK Embedded Server, set the respective properties in the `mail/LSPS_MAIL` element of the`<APP>-embedded/conf/conf/openejb.xml` file:

```
 mail.transport.protocol=smtp
mail.smtp.host=mailsmtp.whitestein.com
mail.smtp.port=25
```

```
mail.from=<YOUR@EMAIL.com>
mail.smtp.user=lsps_user
mail.smtp.auth=true
mail.smtp.starttls.enable=true
mail.smtp.password=<PASSWORD>
password=<PASSWORD>
```

Important: SDK Embedded Server fails to communicate with an SMTP server that requires MD5↩ Digest authentication. This causes the `sendEmail()` calls to fail. To force another authentication method, specify the following property in the `<APP>-embedded/conf/conf/openejb.xml` file: `mail.smtp.sasl.mechanisms=PLAIN`

2.4.2 Configuring Data Source of SDK Embedded Server

To add a data source to the SDK Embedded server, do the following:

1. In `<YOUR_APP>-embedded/conf/conf/openejb.xml`, define the resource and its properties.
   Example mysql data source

```
...
   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 server.

Now you can map a data type model to the data source:

1. In the Outline view, click the root node of the hierarchy.
2. In the Properties view of the data type hierarchy, set database to resource id.
3. Set the table name prefix so you can identify your tables easily.
4. Upload the modules to create the database tables.

You can also create entities for the data base entries and use them via EJBs with EntityManager.

2.4.2.1 Deleting the System Database of SDK Embedded Server

To delete the system database of SDK Embedded Server, remove the `h2` directory in the `<YOUR_A↩ PP>-embedded` project.

Chapter 3

Customization

You can customize the LSPS Application in the following ways:

- customize the Application User Interface web application,
- implement custom functions, task types and form components,
- modify the data in the model instances.

In addition, you can also customize auditing of shared records

3.1 Customizing Application User Interface

Application User Interface is the web application that allows users to interact with the system via To-Dos and Documents. It is part of the LSPS Application and is ready to be customized to meet your presentation and execution requirements.

To adapt the *Application User Interface*, you will generally want to do the following:

- customize its theme,
- adapt its behavior,
- add or modify existing components and create custom pages.

3.1.1 Customizing a Theme

When customizing the theme of the Application User Interface, use the exposed variables of the themes if possible.

If further customization is required, create your own theme. Consider setting up the sass compiler to be able to preview your changes instantly in the browser.

3.1.1.1 Creating a Custom Theme

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

1. In workspace with the sources of your Application User Interface, create the theme resource:
   (a) Open `<YOUR_APP>-vaadin-war/src/main/webapp/VAADIN/themes` with default themes.
   (b) Create a copy of the `lsps-valo`, `lsps-dark` or `lsps-blue`.
   (c) Rename the directory to the theme name.
   (d) In the theme directory in *styles.scss*, change the theme class name:

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

2. Register the theme so it is available in the *Settings* of your application: open the Constants.java class located in `<YOUR_APP>-vaadin/src/main/java/<YOUR-PCKG>/vaadin/util/` and add the theme name to THEMES. Set it as default in the `DEFAULT_THEME` constant.

```
public static final List<String> THEMES = Arrays.asList("lsps-valo", "lsps-dark", "lsps-blue
```

Figure 3.1 New default theme setting

3. Change to the theme directory and edit the respective scss files.

   The default themes have a *style.scss* file that import the relevant scss files. 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 file.

   Make sure none of your custom classes clash with system classes (Vaadin and GWT classes).

4. Consider setting up the run configuration for the Sass compiler so you don't have to rebuild the entire application to see the theme changes:

   Once set up, you simply introduce your change to the scss files and run the saas compiler configuration to preview your changes. If you are running the SDK Embedded server with the application, the new css files are hot-deployed so it is not necessary to compile and re-deploy the application.

5. By default, the Application User Interface is in production mode, which does not support this live-preview feature. To apply the css changes instantly, run the application in debug mode: set the *productionMode* parameter in `web.xml` in the `<YOUR_APP>-vaadin-war` project to `false`.

3.1.1.2 Adding a Sass Rule

To create a new rule in your theme, do the following:

1. Create an scss file in the sass directory of your theme, for example, ∗_textarea.scss∗.
2. Create the rule in the file as a mixin.

```
@mixin _textarea {
  .v-textarea {
    width: 100% !important;
    min-width: 100px;
  }
}
```

3. Import the file and rule in the theme's styles.scss:

```
...
@import "../../../VAADIN/themes/wtpdfviewer/wtpdfviewer.scss";
@import "sass/_textarea.scss";
~
.my-theme {
  @include addons;
  @include lsps-valo-base;
  @include theme-app;
  @include _textarea;
}
...
```

4. Run the Sass compiler or maven build configuration.

3.1.1.3 Compiling a Modified Theme

If you want to preview the changes in LSPS Application often without recompiling other themes, do the following:

1. Set the Application User Interface to run in debug mode: set the `productionMode` parameter in `<YOUR_APP>-vaadin-war/src/main/webapp/WEB-INF/web.xml` to `false`.
2. Set up a Sass compiler configuration that will run `mvn exec:java -Dexec.mainClass="com.vaadin.sass.SassCompiler" -Dexec.args="<INPUT_SCSS> <OUTPUT_CSS>"`:
   (a) Go to Run > Run Configuration.
   (b) In the left pane, select Java Application and click the New button in the caption area of the pane.
   (c) On the right, define the following:
      i. Name: a name of the configuration
      ii. Project: `<YOUR_APP>-vaadin-war`
      iii. Main class: `com.vaadin.sass.SassCompiler`

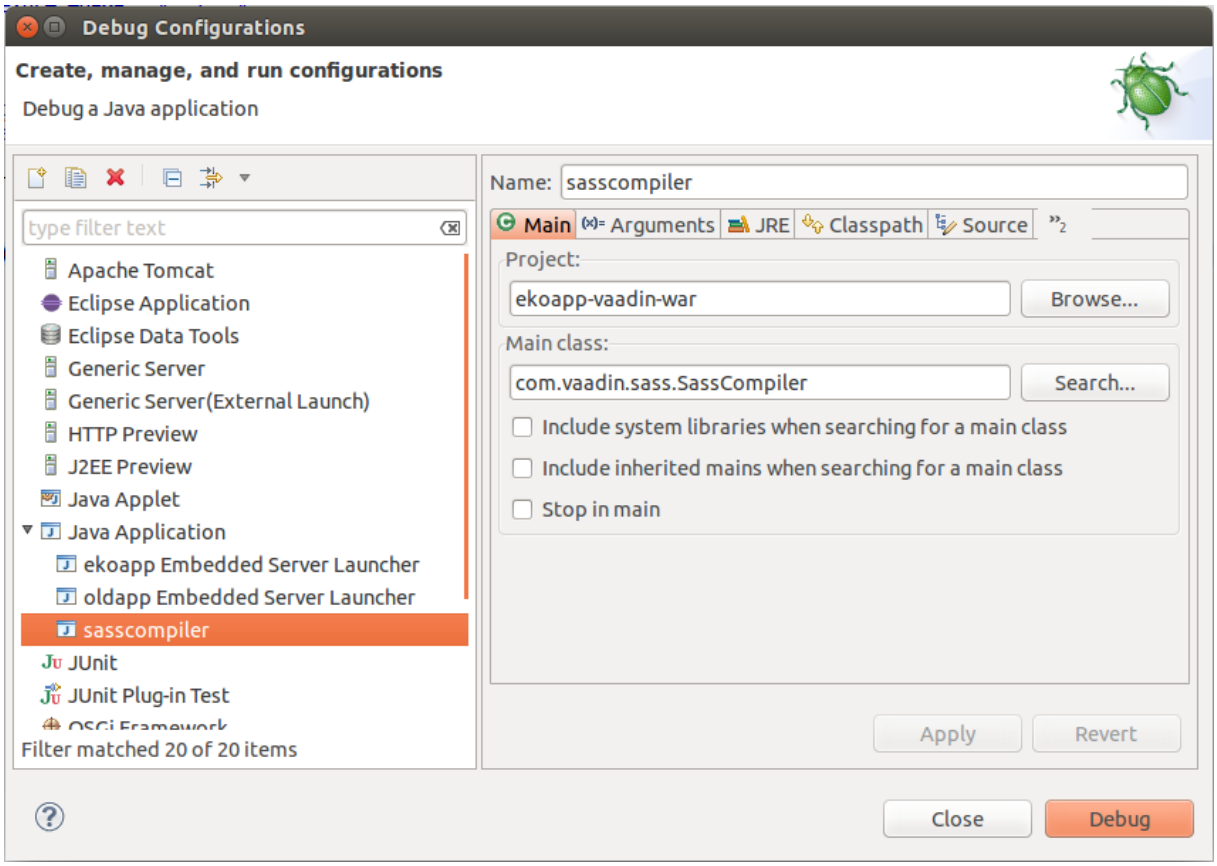


Figure 3.2 Run configuration for Sass compiler

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

The main scss is the style.sccs file with the relevant scss-files imports so the arguments are defined as follows:

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

3. Introduce your scss rules.
4. Run the Saas compiler configuration.
5. Open and refresh the browser with the application. Make sure the correct theme is used: check the selected theme on the Settings page of the application.

Important: In Google Chrome, make sure to have the Chrome DevTools displayed (press the **F12** key) and caching is disabled (go to the *Network* tab and select **Disable cache**)

3.1.1.4 Modifying General Theme Settings

The application default style is based on Vaadin's Valo and extends this theme: The theme's Settings are exposed in the `src/main/webapp/VAADIN/themes/\<THEME_NAME>/sass/_variables.scss` file of the ∗<YOUR_APP>-vaadin-war∗ project. Therefore if you want to change only the general theme properties, do so in the _variables.scss file:

The application themes are located in the `src/main/webapp/VAADIN/themes/` directory.

3.1.1.5 Setting the Default Theme

To set the default theme for your application, open the Constants.java class locate 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.

3.1.2 Customizing Behavior

3.1.2.1 Browser Session Timeout

When the user becomes inactive, the countdown of the session timeout period starts. The countdown keeps running until the user clicks into the UI including clicking empty space or focusing a field. Once the countdown has lapsed, the session is considered inactive and is destroyed by the respective service, which runs every 5-6 minutes or so.

Session timeout is set in minutes in the `<session-timeout>` element in the `<YOUR_APP>-vaadin-war/web.xml` project. Set it to `-1` to disable it.

```
<session-config>
    <session-timeout>15</session-timeout>
</session-config>
```

> **Note on Websphere's LTPA tokens:** Authentication on Websphere is done via LTPA tokens, which define their expiration time. The expiration time is not influenced by user activity.
>
> If a session expires while the LTPA token is valid, the user remains logged in: Next time the user accesses the application, they are assigned a new session, but as their LTPA-token cookie is still valid, they are logged in to the application automatically.

3.1.2.2 Customizing Authentication

You can customize the authentication to the Application User Interface in the `AppUIProvider` class: by default, `getUIClass()`, that serves the UI, checks whether the user is logged in and if this is not the case, it checks whether the user exists and is active. Adapt the method as required.

3.1.2.3 Setting the Home Page

To set the page which is loaded after the user has logged in and when they click the logo, modify `openHomePage()` of the *LspsUI* class:

- sets to a view:

```
public void openHomePage() {
 navigateTo(<VIEW>.ID);
 // for example:
 // navigateTo(getNavigator().documentsViewId());
}
```

- sets to a document:

```
  public void openHomePage() {
  openDocument("<FULLY_QUALIFIED_DOCUMENT_NAME>", null);
  //for example: openDocument("module::docMainScreen", null);
 }
```

3.1.2.4 Importing JavaScript

Important: We strongly recommend to use JavaScript sparingly and exclusively ∗with the aim to change the presentation layer of your application.

To add a JavaScript to your custom application, do the following:

1. Add your JavaScript file to the `<YOUR_APP>-vaadin-war/src/main/webapp/VAADIN/js/` directory.
2. Annotate the LspsUI class of your custom application as follows:

   ```
   @com.vaadin.annotations.JavaScript({ "vaadin://js/<YOUR_JS>.js" })
   ```

   For example:

   ```
   @com.vaadin.annotations.JavaScript({ "vaadin://js/hello.js" })
   ```

3. Build and deploy the application.
4. Check if the script is available under http://localhost:8080/myproject/VAADIN/js/test.↩ js

3.1.3 Customizing Content

The components, their behavior, and layout of the Application User Interface is defined by the `LspsUI` class↩ : Hence to change these, you need to modify the implementation or implement your own *LspsUI* class. In the LSPS Application, the resources are located in the *vaadin.core* package of the <YOUR_APP>-vaadin project.

Note: The content of the Documents and To-Dos is defined in Modules using forms: to modify these, you need to change the design of their forms or related Module resources.

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

- LspsUI represents the root component and holds the user info and connector to the LSPS server. Its *init()* method creates the content root layout and a connector between LSPS forms and your application, called the AppConnector, and calls the initLayout() method which assembles the screen content.
- applayout extends CustomComponent and implements ViewDisplay

  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

To change the components of the application, you will be mostly editing the `initLayout()` method of the App↩ Connector class.

If you decide to implement the LspsUI class, 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, interface that defines the binding contract between the LSPS vaadin renderer and the rest of the application.
- It must provide an implementation of LspsFormConnector, interface for a connector of forms and views.
- It must use JAAS for user authentication: The setting is available in the `login.jsp` and in the `WEB-IN↩ F/web.xml` of the webapp project.

3.1.3.1 Adding an Item to the Navigation Menu

To add a custom item to the navigation menu or remove existing items, do the following:

1. Open your **AppLayout.java** file.
2. Locate the `//Examples:` comment in the `buildMenu()` method.

   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.

   - To remove items from the menu, either comment out the `if` statement with the respective navigation menu item above the example or add an `if` statement which will be check if the user has the required permissions.
   - To add your items to the menu, call the `addDocumentItem()` or `addRunModelItem()` method calls as appropriate.

     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 `hasRight()`
     – Document's `hasRightToOpenDocument()` (false if user has no right to access given document as defined by the document access expression)

3.1.3.2 Adding a 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, contentAr
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);
```

3.1.3.3 Customizing the Login and Logout Page

To modify the login page, logout page, and the page displayed when login action failed, modify the `login.jsp`, `login_failed.jsp`, and `logout.jsp` in the <app>-vaadin-war project of your application.

To change the logo of the area where you enter your credentials, proceed as follows:

1. If you have not done so yet, create a custom theme.

2. In the *login.jsp* file, set your theme as the default theme.

```
...
   var storage = window.localStorage;
   var theme = "my-theme";
```

3. Create the rules for the login header:

   (a) Create a file for the rule in the sass directory of your theme, for example ∗_login.scss∗.

   (b) In the file, define a mixin with the rules.

   - The upper part of the login has the *loginHeader* id.
   - The lower part with credential input has the selector `&.login-page form table`.

4. Run Sass compiler and check the login page.

Example login customization

```
@mixin _login {
 &.login-page #loginHeader{
    background: white url(../img/splashscreen.png) !important;
    background-position: center !important;
    background-size: 400px auto !important;
    background-repeat: no-repeat !important;
 }
 &.login-page form table {
   background-color: white !important;
   background-image: none !important;
 }
 &.login-page input {
   border: 1px solid #c9c9c9 !important;
 }
 &.login-page .v-button .v-button-caption {
   color: #c9c9c9 !important;
 }
}
```

3.1.3.4 Customizing Content of the About Dialog

The information for the dialog is pulled from the *app-version.properties* file which is bound to the maven build properties.

> **Note:** On SDK Embedded Server the properties are not resolved since the application is not deployed as an EAR.

3.1.3.5 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 createSettingsSection() 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;
}
```

3.1.3.6 Creating a Custom Page

Important: We instruct you to use the forms Module to implement the GUI of your page in the examples. However, this feature is experimental and the resulting form is not compatible with the ui Modules, which is the previous and *supported* forms implementation.

To create a page that will be always available to front-end users, do the following:

1. Switch to the *Modeling* perspective.
2. Create a GO-BPMN Project with an executable Module.
3. Create a document definition in the Module (right-click the Module and go to New -> Document Definition).
4. In the editor with the docs file, click Add and define the document details on the right.

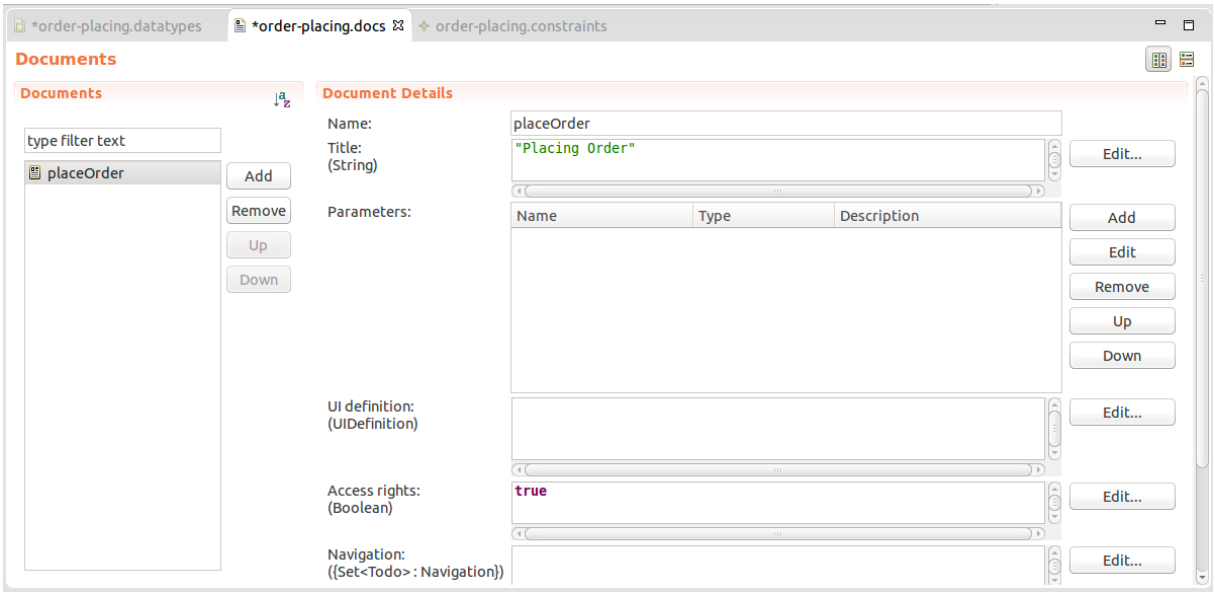


Figure 3.3 Definition of the document with order page

Information on documents is available in the Process Design Suite Guide.

5. In the UI definition field define the page content. Information on forms is available in the Process Design Suite Guide.

6. Upload the Module to your server:

   (a) Make sure the server is running.

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

   (c) Go to your application (for the Embedded Server http://localhost:8080/lsps-application).

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

   (e) Click the document name.

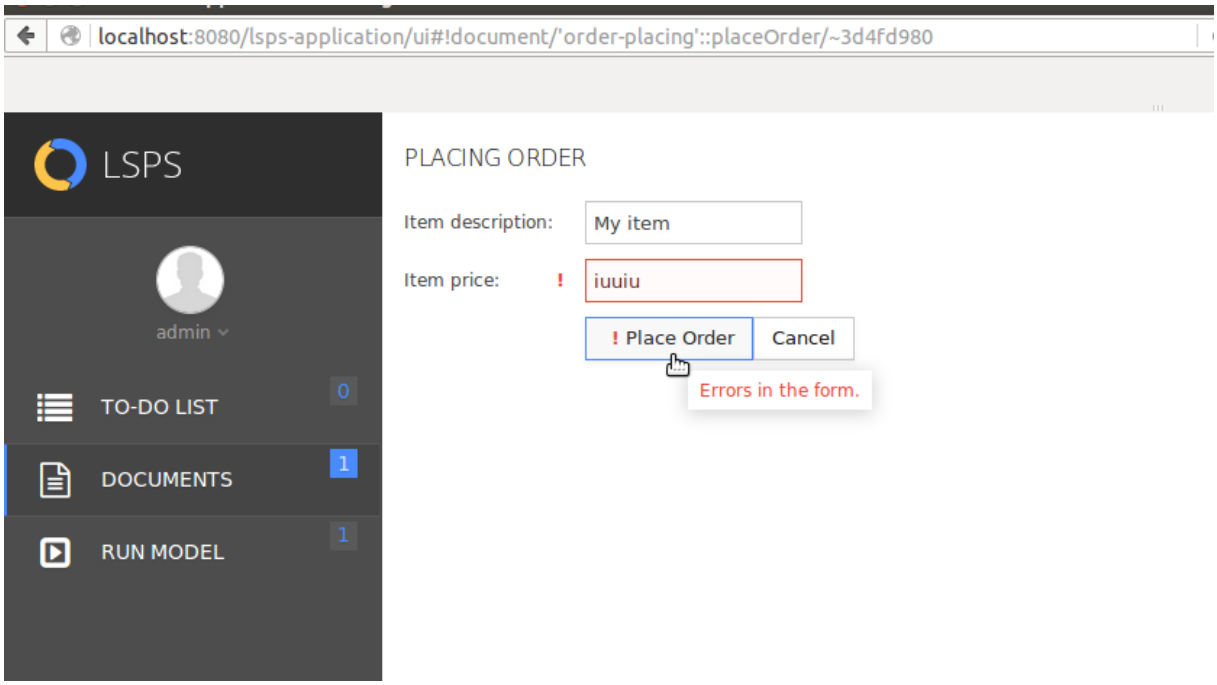


Figure 3.4 Order page

3.2 Creating a Custom Object

You can create custom object:

- To implement custom business logic in your own models, create custom function or task type.
- To add components for your UI or Forms.

When implementing custom objects in Java, you can pass model objects as parameter arguments to custom objects or access model objects via the context parameter or another object; For example, you can pass a record as an argument to your java method and then access its related records.

To work with object of Expression-Language data types in Java, use their implementing Java class.

Expression Language Type	Class
String	java.lang.String
Boolean	java.lang.Boolean
Binary	com.whitestein.lsps.lang.exec.BinaryHolder
Decimal	com.whitestein.lsps.lang.Decimal
Integer	com.whitestein.lsps.lang.Decimal
Date	java.util.Date
Reference	com.whitestein.lsps.lang.exec.ReferenceHolder
Collection	com.whitestein.lsps.lang.exec.CollectionHolder
List	com.whitestein.lsps.lang.exec.ListHolder
Set	com.whitestein.lsps.lang.exec.SetHolder
Type	com.whitestein.lsps.lang.type.Type
Map	com.whitestein.lsps.lang.exec.MapHolder

Expression Language Type	Class
Closure	com.whitestein.lsps.lang.exec.ClosureHolder
Record	com.whitestein.lsps.lang.exec.RecordHolder
Enumeration	com.whitestein.lsps.lang.exec.EnumerationImpl
Property	com.whitestein.lsps.lang.exec.Property
Object	java.lang.Object

3.2.1 Custom Functions and Task Types

To implement custom business logic in your own models, create custom functions or task types. If you want to use the server services and data, register functions as EJBs. If you are defining JPA entities, make sure to add the classes to the respective mapping file.

3.2.1.1 Creating a Function

To create a custom function, you will need to

- declare the function in a function definition file
- and then implemented it either in the Expression Language or as a Java method.

Note: Before you create your own function definition, check the Standard Library for a similar function.

3.2.1.1.1 Function Declaration

Functions are declared in function definition files. The files are created as any other definition file, however note that there are two types of these files which differ in the way they are edited:

- Function definitions edited in the visual function editor that provides graphical support

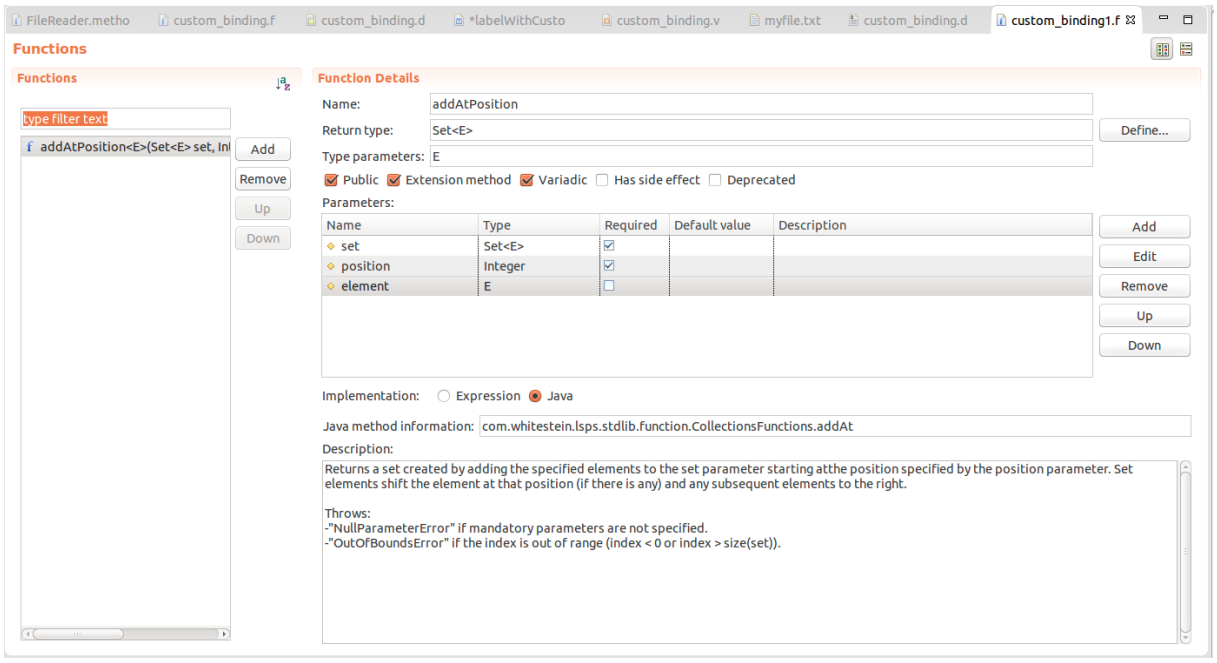


Figure 3.5 Function Editor with a function definition

- Function definitions edited in the text function editor for more coding-like experience

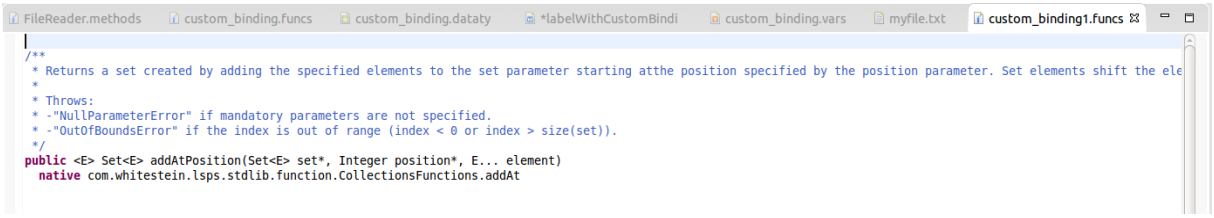


Figure 3.6 Text function editor with a function definition

Whether the visual or text function Editor is used depends on the type of function definition file: a definition file created for one editor cannot be used by the other. However, you can convert the visual function definition file to the text function definition file from the function definition context menu.

3.2.1.1.1.1 Declaring a Function in the Visual Editor

To declare a function in the visual editor, do the following:

1. In the Modeling perspective, create or open a function definition file:
   (a) Right-click your Module.
   (b) Go to **New** > **Function Definition**
   (c) In the *New Function Definition* popup:
   - Enter the name of the definition file
   - Make sure the *Use text definition format* option is **unselected**.
2. Add a new function and define the details:
   - **Name**: name used to call the function
   - **Return type**: data type of the return value

     Note: Previously, to define a function that returned no value, the user could set the Return type to `Null`. Since this is the type that is a sub-type of all types, this is discouraged. Use `void` as return type on functions that do not return any value.
   - **Type parameters**: comma-separated list of abstractions of data types used in parameters Type parameters allow functions to operate over a parameter that can be of different data types in different calls. The concept is based on generics as used in Java. You can also make the 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
   - **Extension method**: whether the function `can be used as an extension method`
   - **Variadic**: function arity
     A function is variadic if zero or more occurrences of its 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.

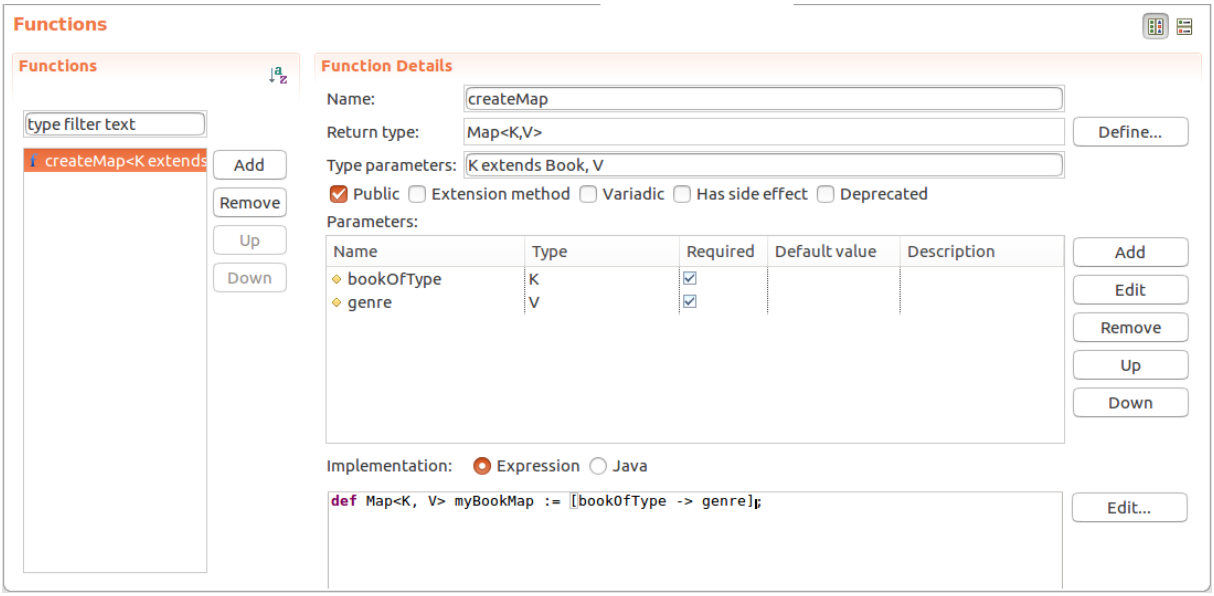


Figure 3.7 Example function with type parameters

3. Define the input parameters. Note that functions can be overloaded.

   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.
   - **Default value**: if no value for the parameter is passed in the function call, the defined default value is used.
   - **Description**: optional description of the parameter

4. Select the implementation type:

   - **Java**: enter the name to the method with its package

     If you have not implemented the method yet, you can get the Java signature for the function method, for example, `public MapHolder createMap(ExecutionContext ctx, Object book, Object genre) throws ErrorException`: right-click the function declaration in Outline and select *Copy Java Signature* to copy it to clipboard. You can then paste it with Ctrl + V.
   - **Expression**: enter the implementation

3.2.1.1.1.2 Declaring a Function in the Text Editor

To create or edit a text function definition, do the following:

1. Open the function definition file that you created with the **Use text definition format** flag or open such a file.

   To convert a visual function definition file to a text function file, right-click the function definition in the GO-↩ BPMN Explorer and click **Convert to Text Definition Format**.

   Important: The conversion is not reversible.

2. In the editor, declare the function following the function syntax (for further information on the syntax, refer to the Expression Language Guide).

   Form of function syntax

```
 <visibility> <return_type> <function_name> (<parameters>){
  <implementation>
}
```

   Example function declaration and implementation in the Expression Language

```
 public Integer getArithmeticMean(List<Integer> integersToProcess){
    sum(integersToProcess)/integersToProcess.size()
}
~
public Integer getArithmeticMean(Integer... integersToProcess){
    sum(integersToProcess)/integersToProcess.size()
}
```

3.2.1.1.2 Function Implementation

You can implement your function either in the Expression Language or in Java:

- Function implementation in the Expression Language does not require adding of code to the server application and can be distributed as part of a GO-BPMN Library or as a Module import.
- If the capabilities of the Expression Language are not sufficient, implement your function in Java. You will need deploy the implementation to the LSPS Server as part of your LSPS Application.

3.2.1.1.2.1 Implementing a Function in the Expression Language

When implementing a function in the Expression Language, you enter the implementation into the function definition file along with the function declaration: the syntax depends on whether you are using the text or visual function editor.

- When in text editor, use the following syntax:

```
<visibility> <return_type> <function_name> (<parameters>) <implementation>
```

- When in visual editor, enter the expression that returns the required output below the declaration.

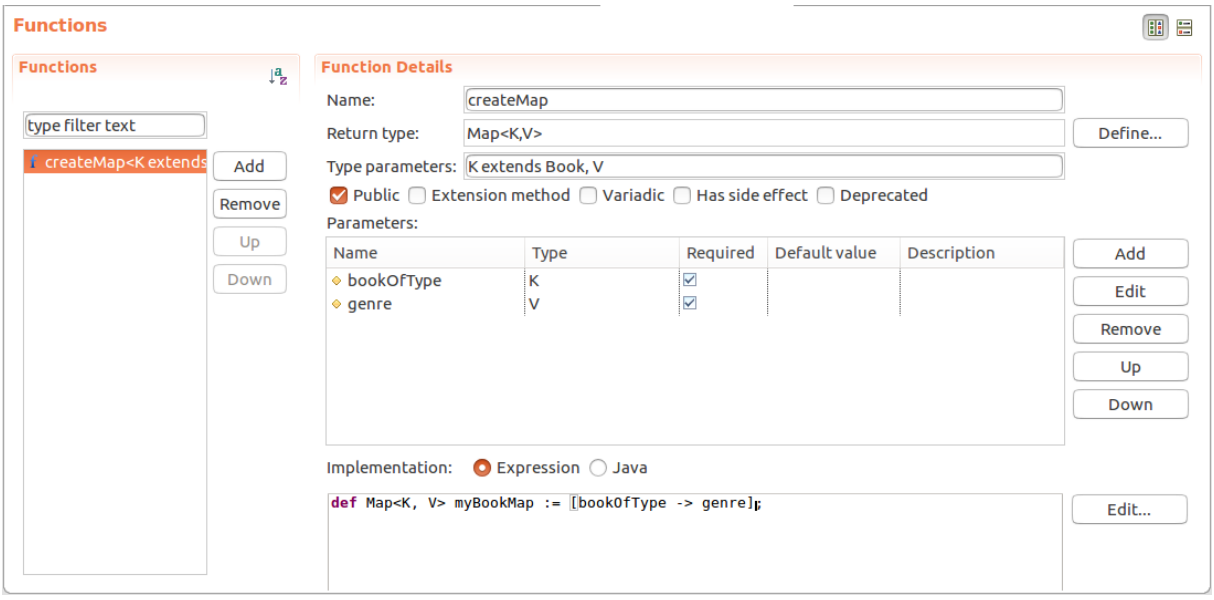


Figure 3.8 Function declaration and definition in the Expression Language

3.2.1.1.2.2 Implementing a Function in Java

When implementing a function as a Java method, you will need to create and deploy a class with the method to the LSPS Server as part of your custom LSPS Application. The implementation in Java can be a POJO or an EJB. The method must be *public*.

Note that the call to the method from LSPS has the context of the function as its first argument.

3.2.1.1.2.3 Implementing a Function as POJOs

If you do not need to inject and use application EJBs into your function, implement your function method as a method of a POJO:

1. In the *ejb* project of your application, create a package with a class that will implement the function:
   (a) Define the method signature: you can copy it from the function definition file if you have already declared it: in the function definition file, right-click the function declaration in Outline and select *Copy Java Signature*.
   (b) Implement the logic in a public method of the class.
   (c) To access resources of the model instance, such as, variables, signal queue, etc., add the following:
   - ExecutionContext input argument to the method call, for example, `public Decimal average(ExecutionContext ctx)`; the returned context is the parent module context.
   - Work with the data from the execution context (parent Module context) with its respective method, for example, `ctx.getNamespace().setVariableValue("myStringVar", "new value");`
2. In the function definition file, define the native statement to call the method:

```
public Boolean isIntPrime(Integer i*)
      native org.whitestein.myapp.customfunctions.PrimeUtils.isPrime;
```

3. Rebuild and re-deploy the LSPS Application.

3.2.1.1.2.4 Implementing a Function as an EJB

If you need to inject and use application EJBs into your function, implement your function method as a method of an EJB. It is recommended to create the resources in a dedicated package of the <YOUR_APP>-ejb project.

1. Create the EJB:
   (a) Create the interface with the methods which the EJB will implement.
   Example interface for an EJB function

```
 @Local
public interface Calculator {
~
  Decimal add(ExecutionContext ctx, Decimal a, Decimal b);
}
```

   (b) Create the EJB and the implementing methods.
   Example stateless bean

```
 @Stateless
@PermitAll
public class CalculatorBean implements Calculator {
~
  @Override
  public Decimal add(ExecutionContext ctx, Decimal a, Decimal b) {
    return a.add(b);
  }
}
```

2. Register the EJB:
   (a) Create the EJB in the constructor of `ComponentServiceBean` in the *ejb* package.

```
@EJB
private <INTERFACE> <BEAN_NAME>;
```

   (b) Call the static `register()` on the EJB in the `registerCustomComponents()` method.

Example EJB registration

```
    public ComponentServiceBean() {
    super(new ConcurrentHashMap<String, Object>(), new ConcurrentHashMap<Class<?>, List<Objec
  }
~
  @EJB
  private Calculator calculatorBean;
~
  @Override
  protected void registerCustomComponents() {
    register(calculatorBean, Calculator.class);
  }
```

3. Add the function to a function definition file:

```
 public Decimal addTwoNos(Decimal a, Decimal b)
native org.eko.primeapp.customfunctions.Calculator.add;
```

4. Build and re-deploy the application.

3.2.1.1.2.5 Accessing the Execution Context

When the server creates a model instance, it is created with its model instance data, such as, who and when created the instance, what status it is currently in, etc. and with the contexts of its executable modules, that includes, the parent executable module and any imported modules; all modules are instantiated as module instances of the model instance and start their execution. They create and hold their runtime data and contexts of their process instances; etc.

To inspect the exact structure of a model instance, export a model instance to XML, for example, from the Management perspective of your PDS. Also refer to the `modeling-language documentation`.

> **Important:** The contexts are by default created on the base execution level; (refer to execution levels).

The `execution context` of your custom objects is passed to it as the first parameter. To add objects to the context, use the methods of its namespace, for example, to create maps, sets, and lists:

```
 Set<String> names = new HashSet<>();
//populate names ...
//add to the namespace of the context:
myCurrentContext.getNamespace().createSet(names)
```

3.2.1.1.2.6 Example Functions

3.2.1.1.2.7 POJO function that checks if an Integer is a prime number

- Declaration:

```
 public Boolean isIntPrime(Integer i*)
       native org.whitestein.myapp.customfunctions.PrimeUtils.isPrime;
```

- Implementation:

```
 public class PrimeUtils {
~
  public Boolean isPrime(Decimal d) throws ErrorException {
    int i = d.intValueExact();
    return org.apache.commons.math3.primes.Primes.isPrime(i);
  }
}
```

3.2.1.1.2.8 POJO function that returns the day of the week of the received Date

The weekday is returned as the enumeration literal of the enumeration *functionJava::Weekday*.

- Declaration:

```
public functionJava::Weekday (Date d*)
 native com.example.library.customfunctions.DateUtils.getWeekday;
```

- Implementation:

```
package com.example.library.customfunctions;
~
import java.util.Calendar;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
~
import com.whitestein.lsps.lang.exec.EnumerationImpl;
import com.whitestein.lsps.lang.type.EnumerationType;
~
public class DateUtils {
~
  private final static Map<Integer, EnumerationImpl> wDays =
     new HashMap<>();
~
  static {
    EnumerationType weekdayenum =
        new EnumerationType("functionJava", "Weekday");
~
    wDays.put(Calendar.SUNDAY, new EnumerationImpl(weekdayenum, "Sunday"));
    wDays.put(Calendar.MONDAY, new EnumerationImpl(weekdayenum, "Monday"))
    wDays.put(Calendar.TUESDAY, new EnumerationImpl(weekdayenum, "Tuesday"));
    wDays.put(Calendar.WEDNESDAY, new EnumerationImpl(weekdayenum, "Wednesday"));
    wDays.put(Calendar.THURSDAY, new EnumerationImpl(weekdayenum, "Thursday"));
    wDays.put(Calendar.FRIDAY, new EnumerationImpl(weekdayenum, "Friday"));
    wDays.put(Calendar.SATURDAY, new EnumerationImpl(weekdayenum, "Saturday"));
~
  }
~
  public static EnumerationImpl getWeekday(Date date) {
    Calendar c = Calendar.getInstance();
    c.setTime(date);
~
    return wDays.get(c.get(Calendar.DAY_OF_WEEK));
  }
}
```

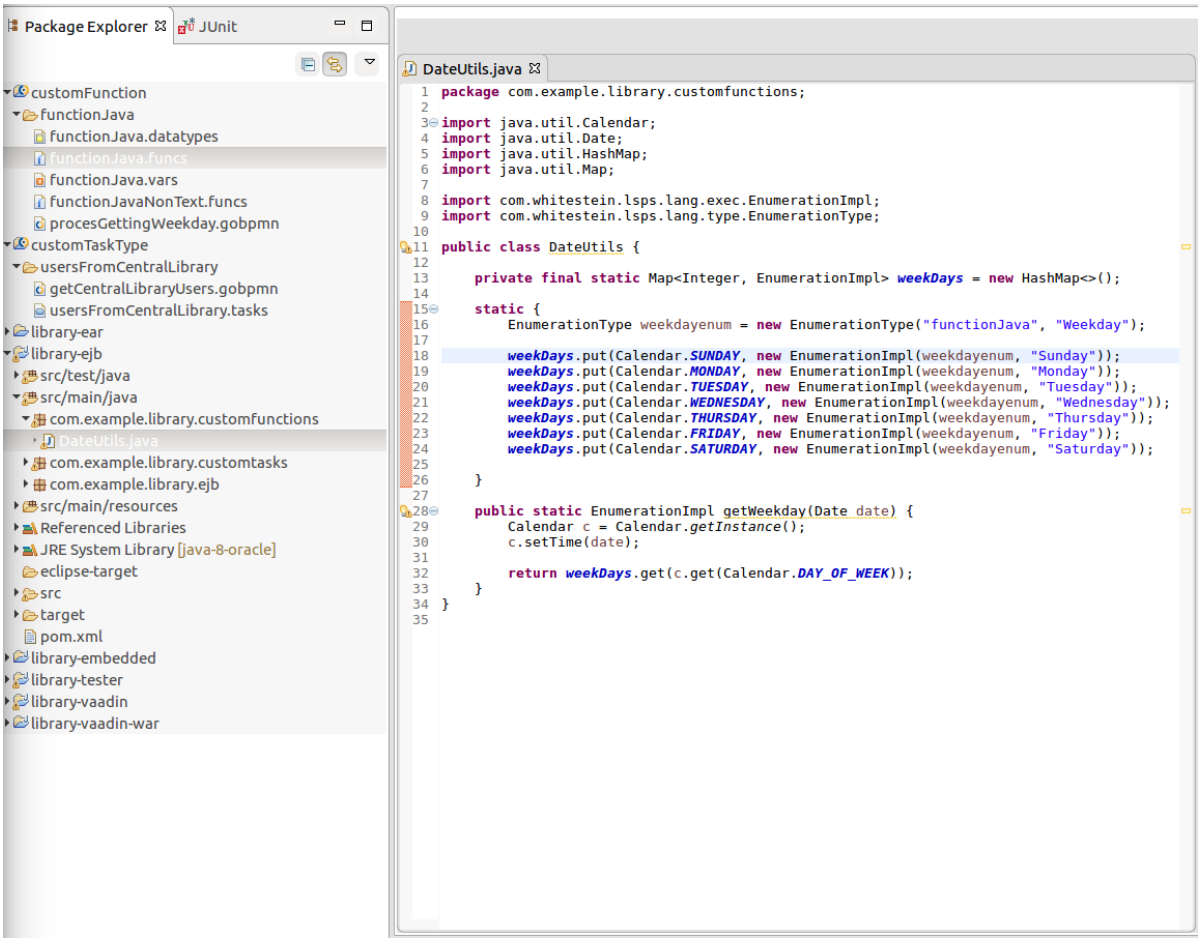


Figure 3.9 Custom function definition and declaration

3.2.1.1.2.9 POJO function that returns a Set with names of Goals in a model instance

- Declaration:

```
public void getGoalNames()
      native org.eko.primeapp.customfunctions.GoalServer.getGoalNames;
```

- Implementation:

```
import com.whitestein.lsps.engine.lang.ExecutionContext;
import com.whitestein.lsps.engine.state.xml.*;
import com.whitestein.lsps.lang.exec.SetHolder;
~
public class GoalServer {
~
  public SetHolder getGoalNames(ExecutionContext ctx) {
~
    Set<String> names = new HashSet();
~
    for (ProcessInstance processInstance : ctx.getModelInstance().getProcessInstances()) {
      for (Goal goalValue : processInstance.getGoals()) {
        for (GOElement goalChild : goalValue.getChildren()) {
          //populate a set of strings
          names.add(goalChild.getName());
~
        }
```

```
      }
    }
    return ctx.getNamespace().createSet(names);
  }
}
```

3.2.1.1.2.10 Stateless-EJB function

- **Interface:**

```
 @Local
public interface PdfTools {
~
  /**
   * Check if the file is valid PDF file.
   *
   * @param context
   * @param binaryPdf
   * @return is provided file of type PDF, true or false
   */
  boolean isValidPdf(ExecutionContext context, BinaryHolder binaryPdf);
~
  /**
   * Extract a creation date from PFD file.
   *
   * @param context
   * @param binaryPdf
   * @return Date when provided PDF was created or null if no or invalid value.
   * @throws IOException if file is not valid PDF file
   */
  Date getCreationDate(ExecutionContext context, BinaryHolder binaryPdf) throws IOException;
~
  /**
   * Read document title from provided PDF file.
   *
   * @param context
   * @param binaryPdf
   * @return Title of the provided PDF or null if no value.
   * @throws IOException if file is not valid PDF file
   */
  String getTitle(ExecutionContext context, BinaryHolder binaryPdf) throws IOException;
}
```

- **Implemenation:**

```
 @Stateless
@PermitAll
@Interceptors({ LspsFunctionInterceptor.class })
public class PdfToolsImpl implements PdfTools {
~
  @Override
  public boolean isValidPdf(ExecutionContext context, BinaryHolder binaryPdf) {
    try {
      PdfReader reader = new PdfReader(binaryPdf.getData());
      reader.close();
      return true;
    } catch (IOException e) {
      //Throws an exception when file is not PDF file
      return false;
    }
  }
~
```

```
  @Override
  public Date getCreationDate(ExecutionContext context, BinaryHolder binaryPdf) throws IOExce
    PdfReader reader = new PdfReader(binaryPdf.getData());
    Map<?, ?> info = reader.getInfo();
    String pdfDateString = (String) info.get("CreationDate"); //PdfName.CREATIONDATE contains
    reader.close();
    if (pdfDateString == null) { // no value in the PDF
      return null;
    }
    Calendar creationDateCalendar = PdfDate.decode(pdfDateString);
    if (creationDateCalendar == null) { // invalid value in the PDF
      return null;
    }
    return creationDateCalendar.getTime();
  }
~
  @Override
  public String getTitle(ExecutionContext context, BinaryHolder binaryPdf) throws IOException
    PdfReader reader = new PdfReader(binaryPdf.getData());
    Map<?, ?> info = reader.getInfo();
    String pdfTitle = (String) info.get("Title");
    reader.close();
    return pdfTitle;
  }
}
```

- **Registration**

```
 @EJB
private PdfTools pdfTools;
~
@Override
protected void registerCustomComponents() {
   register(pdfTools, PdfTools.class);
}
```

3.2.1.2 Creating a Task Type

Every Task in a BPMN Process is of a particular type: The task type determines the Task's logic. Therefore, if you need a task that will perform actions specific to your business or interact with another system, consider implementing a custom task type.

You will need to create the following:

- Declare the task type
- Implement the task type

3.2.1.2.1 Declaring a Task Type

To declare the task type with its implementation and make it accessible for modeling in PDS, do the following:

1. Switch to the Modeling perspective.
2. In your module, create a task type definition: right-click the module, then **New** -> **Task Type Definition**.
3. In the Task Type Editor, click Add under Task Types.

4. Under Task Type Details, enter the task-type name and in the Classname field enter the fully qualified name of the class.

   If you have already implemented the task type, you can copy the qualified name of the task class in its Outline view: right-click the class name and select **Copy Qualified Name**.

5. Select the relevant flags:

   - Public: select to allow access from importing modules
   - Create activity reflection type: select to create a Record that represents the action taken by the task type. The Record is a subtype of the Activity record and Therefore can be set as the *activity* parameter of the Execute task.
   - Deprecated: select to display a validation marker for deprecated elements when a task of this task type is used.

6. In the Parameters list box, define the task parameters if applicable.

   Check *Dynamic* to wrap the parameter value in a non-parametric closure: Such a parameter is processed as `{ -> <parameter_value> }`. The task-type implementation has to process the parameter as a closure.

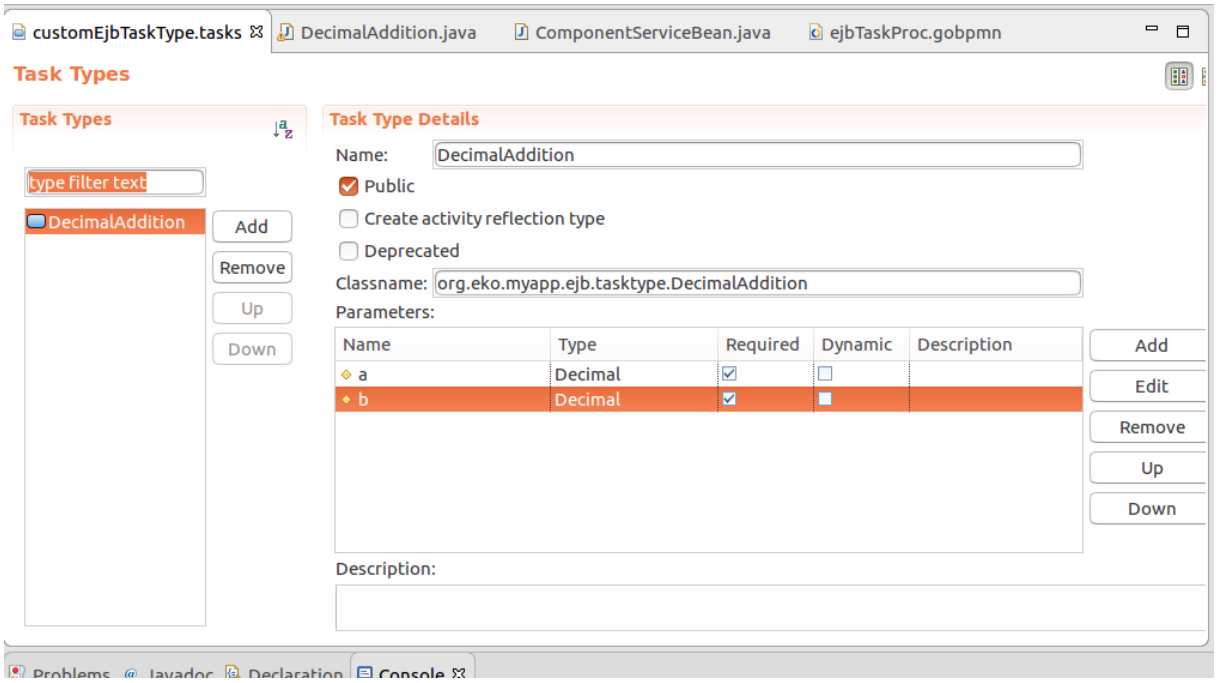


Figure 3.10 Task type declaration

7. If you have not created the implementation yet, generate the class for the task type:

   (a) In the GO-BPMN Explorer view, right-click the GO-BPMN module.

   (b) Go to **Generate** -> **Task Java Sources**.

   (c) In the Task Source Code Generation dialog box:

      i. Select the check boxes of the relevant tasks.

      ii. In the Destination folder text box, specify the destination path, possibly to a package of the <YOUR_APP>-ejb project.

      iii. Click **Finish**.

   The generator does the following:

   - places the task implementation in the correct directory structure.

- generates a class declaration as an implementation of the com.whitestein.lsps.engine.↩ ExecutableTask interface;
- creates a variable for each parameter in the task type and the related setters used by the Execution Engine to access the variables;
- generates the initial structure for the task implementation and documentation.

(d) Implement the task-type class.

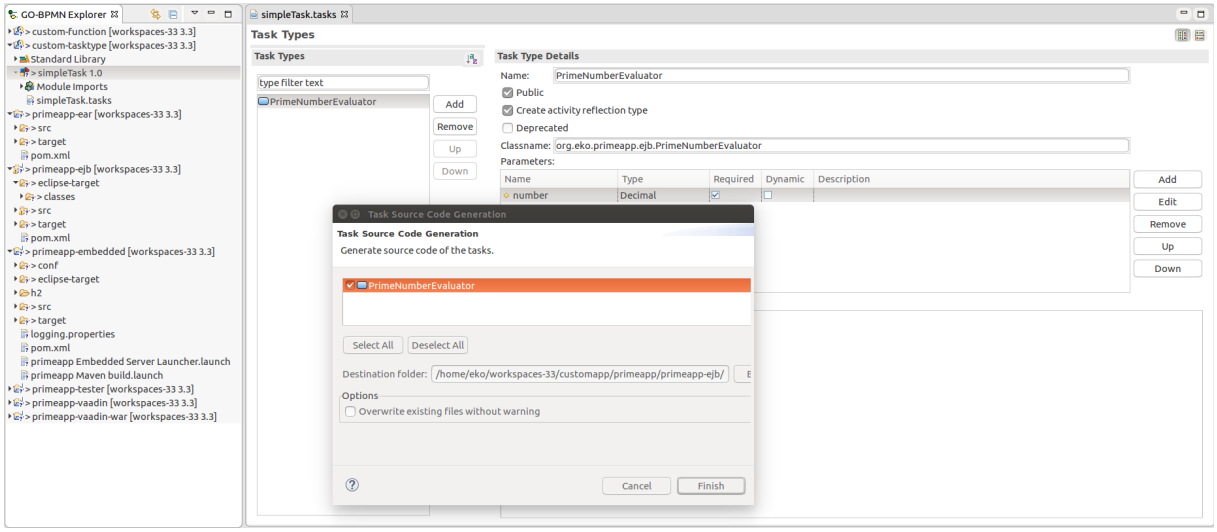


Figure 3.11 Task source code generation

Consider distributing the `module as a library`.

3.2.1.2.2 Implementing a Task Type

The implementation of your task type must implement the `ExecutableTask` interface or its subtype.

> **Note:** To implement an asynchronous task that waits for an event, communicates with a third-party system, or performs demanding operations, refer to Implementing an Asynchronous Task Type.

To implement a custom task type, do the following:

1. Declare the task type in your Module. Consider distributing it as a part of a library.
2. In a workspace with your LSPS Application, switch to the Java perspective.
3. If you have not generated the *Task Java Sources*, create the implementing task-type class:
   (a) Right-click your package, go to **New** -> **Class**.
   (b) In the dialog, set ExecutableTask in the Interface field.

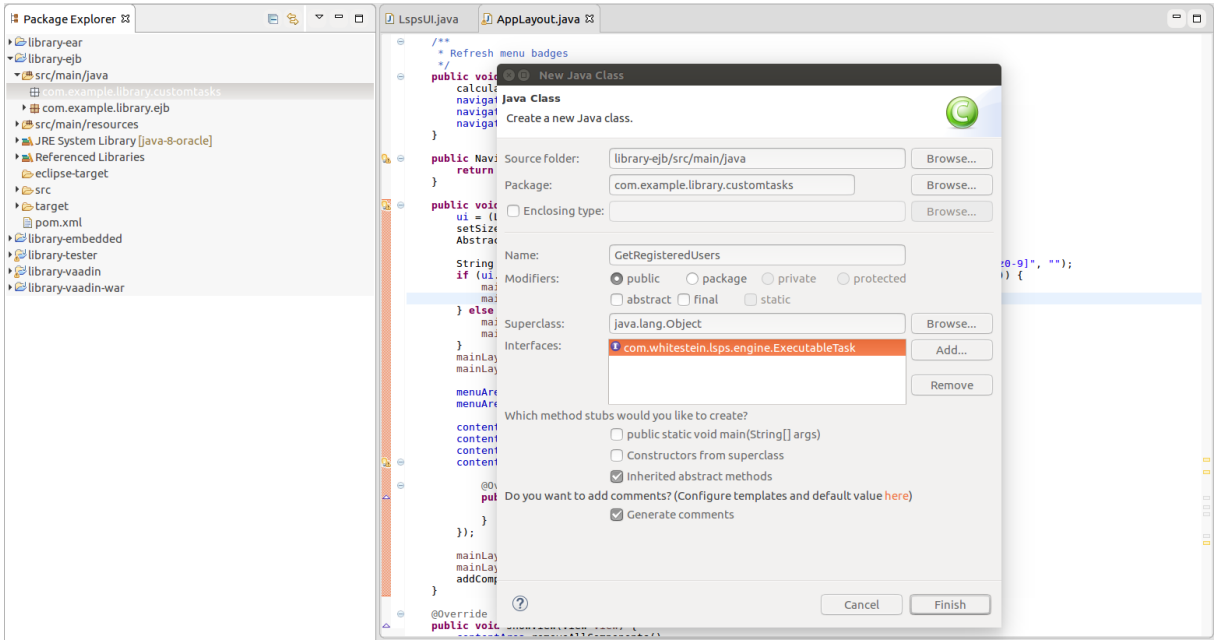


Figure 3.12 Creating class for implementation of the custom task type

4. Implement the methods in the created class:

   - The *start()* runs when the task instance becomes `Alive` The method must return a *Result* value, that is, *Result.FINISHED* or *Result.WAITING_FOR_INPUT*:
     - If the method returns FINISHED, the task becomes `Accomplished` and the Process execution continues (the token leaves the Task).
     - If the method returns WAITING_FOR_INPUT, the token stops: the task becomes `a transaction boundary` and the method `processInput()` will be called when the task instance receives new input.
   - The *processInput()* method must also return a *Result* value: the return value is the same as in *start()*, but while the *start()* method is called only when the task becomes *alive*, the *processInput()* method is called whenever the task instance receives input until it returns `Result.FINISHED`. Then the task instance becomes `accomplished` and the Process execution continues (it releases the token).
   - The *terminate()* method is called when the task instance is terminated abnormally, for example, by a timeout intermediate event attached to the task

     Important: The methods *start()* and *processInput()* must be implemented in such a way that the thread is not blocked.

5. To get the parameters of the task as defined in the Task Type definition, use the *getParameter()* method on the TaskContext. The TaskContext is passed as a parameter to the *start()* method.

6. To access other resources of the model instance, such as, variables, signal queue, etc. use the respective methods of the TaskContext, such as, *getVariableValue()*, *getProcessModel()*, *getProcessInstance()*, *addSignal()*, etc.

Example task implementation

```
public class RecordReturn implements ExecutableTask {
~
  @Override
  public Result processInput(TaskContext arg0, Object arg1) throws ErrorException {
    return null;
  }
```

```
~
  @Override
  public Result start(TaskContext context) throws ErrorException {
    //get the message parameter of the Task which is a Record:
    RecordHolder message = (RecordHolder) context.getParameter("message");
    //save the value of the "text" property of the message Record instance:
    String messageText = (String) message.getProperty("text");
~
    //set the "myProcessVariable" of the process to the messageText:
    context.setVariableValue("myProcessVariable", messageText);
    //finish and release the token:
    return Result.FINISHED;
  }
~
  @Override
  public void terminate(TaskContext context, TerminationReason reason) throws ErrorException {
  }
}
```

If your class is an EJB, register it with the ComponentServiceBean class.

3.2.1.2.3 Implementing an Asynchronous Task Type

Asynchronous tasks represent the `border of a model transaction`: they hold the token but do not block further process execution (other tokens can continue). The task still waits for the result of its implementation and only then finishes.

An asynchronous task type must implement `ExecutableTask` and extend the `AbstractAsynchronous↩ExecutionTask` which is an EJB.

Note that implementations of asynchronous task types must run in a single transaction: if your task-type implementation requires multiple transactions, split it into multiple task types. For such tasks, if the server is restarted during the task execution, the execution is repeated.

> **Important:** If you still decide to use multiple transactions in the implementation of your asynchronous tasks, make sure that the execution is always consistent: consider handling the scenario when the server is restarted while the task is running: since the asynchronous execution of the task is not governed by the task, on server restart, the execution seizes to exist while the task itself becomes running again and waits for the result from the execution. To solve this problem, you can
>
> - define the timeout on the task in your models to handle the case when the task receives no results after server restart: add an `interrupting timer event` on the border of your task with a timeout duration. Mind that such handling can cause premature interruption of the task as a side effect.
> - persist data about the execution phase in the database and check its status regularly from the running task (heartbeat check).

To implement an asynchronous task type, do the following:

1. In the *<YOUR_APP>-ejb project*, create the class for the task type implementation that extends `AbstractAsynchronousExecutionTask` and implements `ExecutableTask`.
2. Make the class an EJB (`AbstractAsynchronousExecutionTask` is implemented as an EJB).
3. Implement the methods:

- `executeAsynchronously()`: checks prior to task execution whether the task should be executed asynchronously; this is useful for tasks that can run both as asynchronous or synchronous;
- `collectDataForExecution()`: provides the data from the task context required for the Java implementation, typically parameters of the task type
- `getImplementationClass()`: returns the class that implements the task type (typically this class)
- `processDataAsynchronously()`: processing logic that returns the result of the actions
- `processExecutionResult()`: processes the result after the asynchronous action

```
 @Stateless
public class GoogleSearchResultStats extends AbstractAsynchronousExecutionTask implements
~
  @Override
  public Serializable collectDataForExecution(TaskContext context) throws ErrorException
    String collectedData = (String) context.getParameter("queryString");
    return collectedData;
  }
~
  @Override
  public boolean executeAsynchronously(TaskContext context) throws ErrorException {
    //the task is always asynchronous:
    return true;
  }
~
  @Override
  public Class<? extends AbstractAsynchronousExecutionTask> getImplementationClass() {
    return GoogleSearchResultStats.class;
  }
~
  @Override
  public Serializable processDataAsynchronously(Serializable data) {
    String result;
    try {
      String keyword = (String) data;
      result = readFromUrl(keyword);
    } catch (IOException e) {
      result = "not found";
    }
    return result;
  }
~
  @Override
  public void processExecutionResult(TaskContext context, Serializable result) throws Err
    System.out.println("Number of results: " + result);
  }
~
  /**
   *
   * @param keyword
   * @return number of results
   * @throws IOException
   */
  public String readFromUrl(String keyword) throws IOException {
    String url = "https://www.google.sk/search?q=" + keyword;
~
    Document document = Jsoup.connect(url).userAgent("Mozilla").timeout(10000).get();
    String question = document.select("#resultStats").text();
    String resultCount = question.split(": ")[1];
~
    return resultCount;
  }
}
```

4. If you want to repeat the execution under some circumstances, throw a runtime exception from the respective method, for example, `LSPSRuntimeException`.
5. Inject and register your class in the ComponentServiceBean.

```
  @EJB(beanName = "GoogleSearchResultStats")
  private ExecutableTask searchResultTask;
~
  @Override
  protected void registerCustomComponents() {
    // parameter 1 is the ejb, parameter 2 is the implementation class as referenced in the t
    register(searchResultTask, GoogleSearchResultStats.class);
```

6. If your class is an EBJ, register it with the ComponentServiceBean class.
7. Build and deploy your application
8. Declare the task type in a module.

3.2.1.2.4 Creating an EJB Task Type

To have a task type implementation that is an EJB, you need to register it with the server via the `register()` method of the `ComponentServiceBean` class.

1. Inject the task EJB in the `ComponentServiceBean` class as the ExecutableTask:

```
  @EJB(beanName = "DecimalAddition")
  private ExecutableTask decimalAddition;
```

2. Register it with the `ComponentServiceBean` class in the `registerCustomComponents()` method.

```
  register(decimalAddition, DecimalAddition.class);
```

3.2.1.2.4.1 Example EJB Task Type

Implementation:

```
 @Stateless
public class DecimalAddition implements ExecutableTask {
~
  @Override
  public Result processInput(TaskContext context, Object input) throws ErrorException {
    return null;
  }
~
  @Override
  public Result start(TaskContext context) throws ErrorException {
    Decimal a = (Decimal) context.getParameter("a");
    Decimal b = (Decimal) context.getParameter("b");
    System.out.println("################### result" + a.add(b));
    return Result.FINISHED;
  }
~
  @Override
  public void terminate(TaskContext context, TerminationReason reason) throws ErrorException {
  }
}
```

Registration:

```
 //Modifications in the ComponentServiceBean class:
  @EJB(beanName = "DecimalAddition")
  private ExecutableTask decimalAddition;
~
    protected void registerCustomComponents() {
    register(decimalAddition, DecimalAddition.class);
  }
```

3.2.1.3 Custom Objects as EJBs

When implementing your Task Types and Function in Java, you can implement them as POJOs or as EJBs: it is recommended to use 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`.

3.2.1.3.1 Registering EJBs

If your custom object needs to inject EJBs, you will need to implement its as EJB and register it with the Execution Engine:

1. Make sure your implementation is annotated as an EJB.
2. In your ejb project, edit the ComponentServiceBean class:
   - 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;
     ```

   - For pure EJBs, add the bean directly.

     ```
     @EJB
      private UserBean userBean;
     ```

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);
     }
     ```

3.2.1.3.2 Handling Exceptions in EJB Functions and Task Types

If a custom a task or function implemented as a session bean fails with an exception on runtime, the current transaction is rolled back and the interpretation fails even if the exception is caught later (for example, by a boundary Error Intermediate Event). Hence make sure to catch all exceptions in your bean.

Use `RuntimeExceptionCatcherInterceptor` as an interceptor in your stateless beans. This interceptor catches all runtime exceptions thrown from a bean's business methods and converts them to ErrorExceptions, which can be handled in your model by the with the `error mechanism` of the GO-BPMN Modeling Language.

3.2.1.4 Using Entities

If you are creating JPA *entities*, make sure to add the class to the mapping file of the persistence unit for its data source.

Example configurations on SDK Embedded Server *persistence.xml*

```
 8<---
    <persistence-unit name="user-unit" transaction-type="JTA">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <jta-data-source>jdbc/USERS_DS</jta-data-source>
        <mapping-file>META-INF/user-entities.xml</mapping-file>
        <validation-mode>NONE</validation-mode>
----->8
```

user-entities.xml

```
 8<---
<?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.sun.com/xml/ns/persis
  version="2.0">
    <entity class="org.eko.orderusersapp.entity.User" />
</entity-mappings>
----->8
```

Entity

```
 @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;
  }
}
```

3.2.2 Custom Form and UI Components

If the components for your UI or Forms do not meet your requirements, you can implement your own components: The process differs depending on whether you are using UI or Forms form implementation.

3.2.2.1 Creating a UI Component

Important: This section deals with implementing custom form components for the *ui* module forms. Information on how to implement a custom *forms* component is available here.

You can implement a custom ui form component either in Java or in the Expression Language: When implementing in the Expression Language, both the implementation and declaration of the component are stored in a custom definition file in a GO-BPMN Module. If you want to use a custom Vaadin implementation, you will need to first implement the component in Java and then create its declaration in a GO-BPMN Module.

Before you implement a custom UI component, make sure to get familiar with execution levels, since ui forms are automatically created on the `screen level` and might use the View Model component to operate on additional screen levels to isolate transient form data.

3.2.2.1.1 UI Component in Java

To define and declare a custom form 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 a dedicated package `<YOUR_APP_PACKAGE>.vaadin.<MY_PACKAGE>` in the `<YOUR_APP>-vaadin` project). The implementing class must meet the following:
   - It *implements the* `com.whitestein.lsps.vaadin.ui.components.UIComponent` interface.

     Important: It is not recommended to extend the form components of the Standard Library since their methods might change.
   - It defines a *constructor with UIComponentData* 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.
   - It *implements the getComponentData() method*, which returns the component data.
   - It *implements 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 Label Component implementation

```
 import com.vaadin.ui.AbstractComponent;
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.util.Variant;
~
public class MyLabel extends Label implements UIComponent {
~
  private final UIComponentData uic;
~
  public MyLabel(UIComponentData uic) {
    this.uic = uic;
  }
~
  @Override
  public UIComponentData getComponentData() {
    return uic;
  }
~
  @Override
    public void refresh() {
```

```
    String suffixText = getProperty("suffix");
    String content = getProperty("content");
    suffixText = getLocalizedString(suffixText);
    content = getLocalizedString(content);
    setValue(content + " " + suffixText);
  }
~
  private String getLocalizedString(String string) {
    return uic.getScreen()
        .getContextHolder()
        .getAppConnector()
        .getLocalizer()
        .getLocalizedString(string, this);
  }
~
  private String getProperty(String propertyName) {
    return Variant //variant allows to set the scope of closure and handle any null value
        .definitionOf(this) //get the record of MyLabel (wrapped in Variant)
        .getPropertyValue(propertyName).closure() //get text property value and cast to c
        .inScope(this)//set the scope of the closure
        .call().string().valueOrNull(); //execute; cast result to string, and get value (
  }
~
  @Override
  public AbstractComponent getWidget() {
    return this;
  }
}
```

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) Enable automatic compilation of the your widget sets: open the `<YOUR_APP>-vaadin-war/pom.xml` file and configure the maven Vaadin plugin.

   (c) In the pom file, uncomment the vaadin-client-compiler dependency.

   (d) Open the LspsUI Java file and modify the @Widgetset annotation, to reference your widget set, for example `@Widgetset("com.whitestein.lsps.vaadin.webapp.MyWidgetSet")`

   (e) Open the `<YOUR_APP>-vaadin-war/pom.xml` and add maven dependency to the Vaadin component jar file.

3. In the Modeling perspective, reflect the component implementation as records:

   (a) Create the component record which extends the UIComponent record which implements the LSPS reflection of the Vaadin component: In the example, we extended Vaadin's Label, which is implemented by the *UIOutputText* and this is reflected as the ui::OutputText: Hence we imported the OutputText record since this reflects the Vaadin Label and created the subrecord UICustomLabel that will reflect our MyLabel class.

   (b) Add any additional fields which the user needs to populate for your component (suffix defined as a closure in our example). Make sure these are handled in your implementation properly.

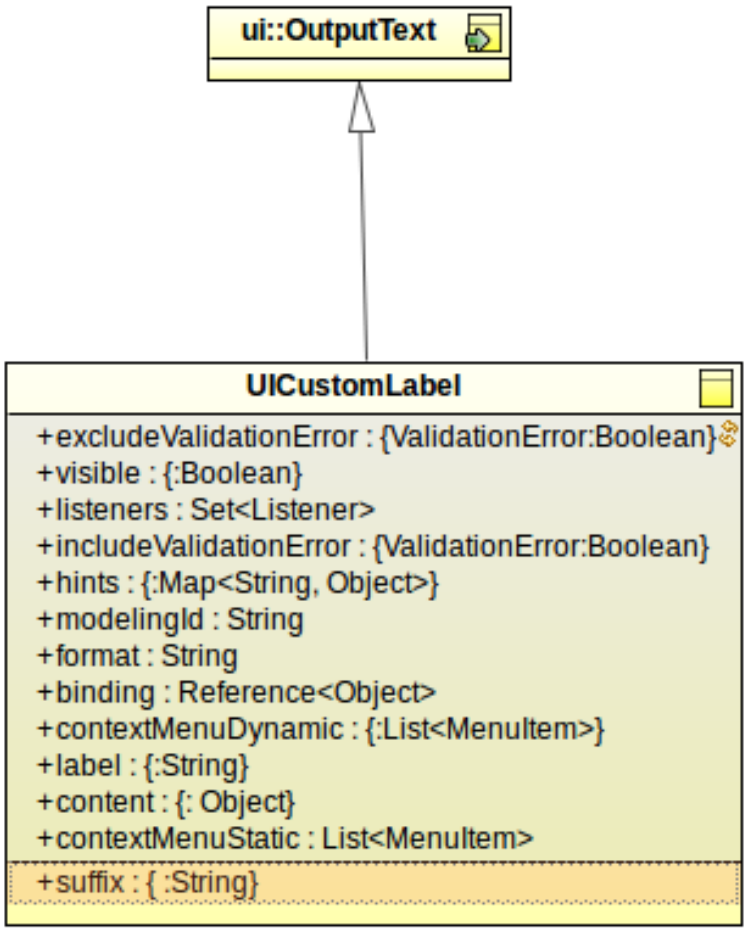


Figure 3.13 Custom component records

(c) 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 name of the component record.

(d) 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 value of the property is displayed in the Form editor

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

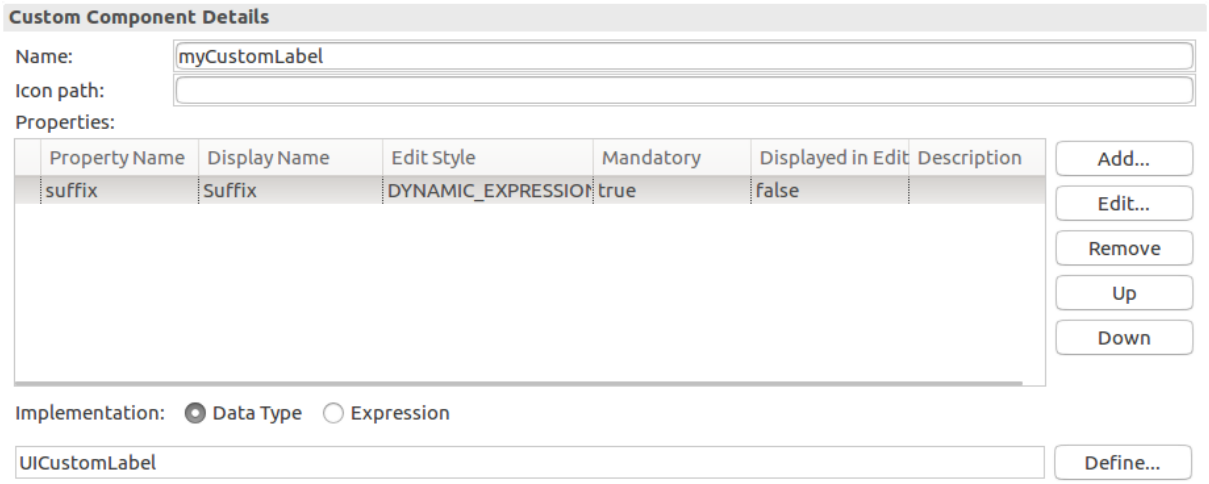


Figure 3.14 A custom component definition with no additional fields

4. Connect the record with its implementation: in the `LspsUIComponentFactory` class, uncomment the `createComponent()` method and modify it to return your component when the respective record is requested.

```
 public class LspsUIComponentFactory extends UIComponentFactoryImpl {
~
  public LspsUIComponentFactory(LspsAppConnector connector) {
    super(connector);
  }
~
  @Override
  protected UIComponent createComponent(UIComponentData componentData) {
    final String type = componentData.getDefinition().getTypeFullName();
    if (type.equals("customUIComponent::UICustomLabel")) {
      return new CustomUiComponent(componentData);
    }
    return super.createComponent(componentData);
  }
}
```

5. Build and deploy your application.

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

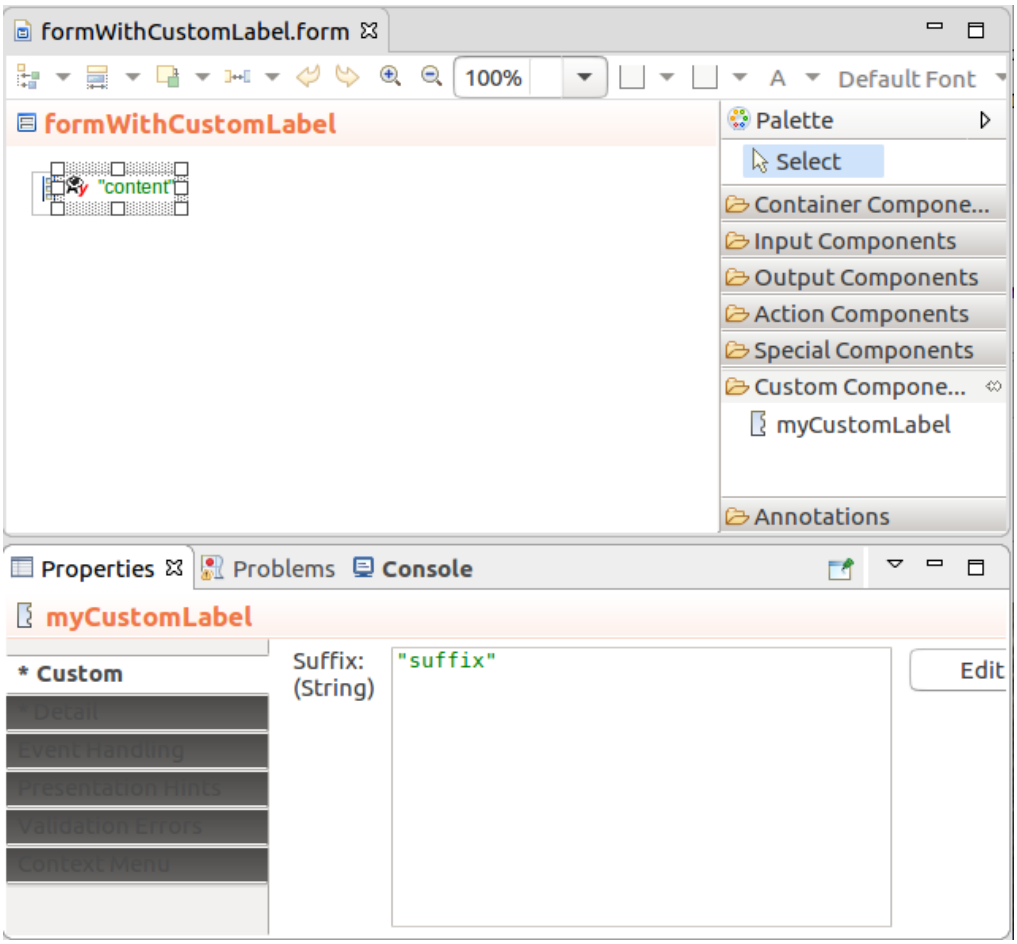


Figure 3.15 Custom component in form definition (inherited properties are in the Detail tab)

3.2.2.1.2 UI Component in the Expression Language

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.

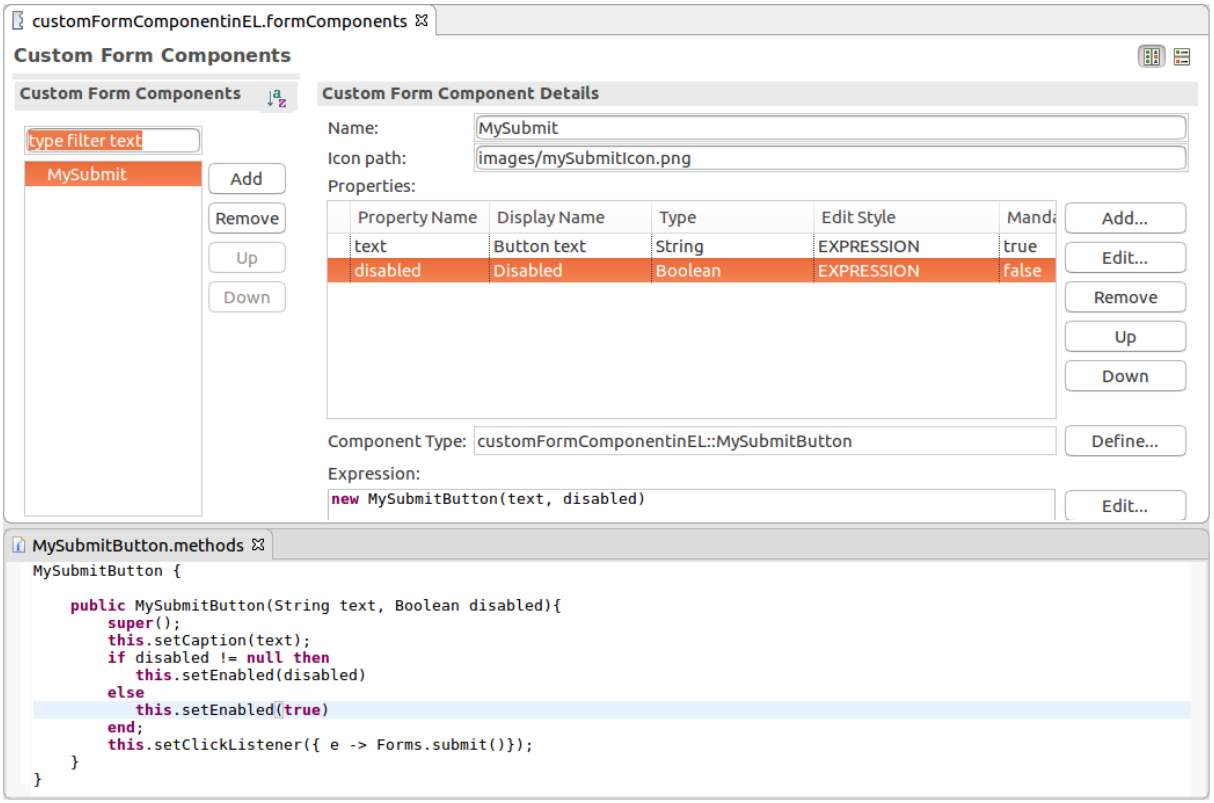


Figure 3.16 A custom component with a parameter

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 checkbos: 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;
```

3.2.2.1.3 UI Component with a Custom Event

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

1. In the <YOUR_APP>-vaadin project, create the class of the 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.
2. In the event class, 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()
        .createRecord("colorpicker::Color");
    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 recordholde
  protected Map<String, ?> getEventProperties(UIComponent component) {
    final Map<String, Object> result =
       new HashMap<String, Object>(super.getEventProperties(component));
    result.put("color", toColor(component, newColor));
    return result;
  }
}
```

3. Implement your custom component: Make sure the UIComponentData is defined as a class variable, so you can use it in the getComponentData method.

   (a) 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 a custom event. It enters the `event queue of the event-processing cycle` when `UIComponents.fireAnd↩ Process()` method is called.

   (a) Register the component in the `LspsAppComponentFactory` class: uncomment the `create↩ Component` 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);
    }
}
```

(b) 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);
```

(c) Define the refresh() method for the component.

```
 @Override
public void refresh() {
  Variant.RecordVariant color = Variant.definitionOf(this).getPropertyValue("color").clos
      .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());
  }
```

(d) 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").cl
        .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;
  }
}
```

4. Create the records for your event.

5. Declare your component in PDS:

   (a) Create a data type model that reflects the components, events, and any related data types defined in your application (in the example, the color picker, color, and color-change listener). Note that the data types 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.

   (b) Create the custom component definition: Use the component record as its implementation.

   (c) 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})}
)
```

3.2.2.2 Creating a Forms Component

Important: This feature is experimental and its API might change in future releases. To create fully supported forms in this LSPS version, use the `ui module of the Standard Library`. If you decide to use this feature, you might want to read about `the differences between forms and ui`.

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

- Declaration of your form component as a Record
- Implementation of your form component
  - in the Expression Language: you will create a form component that will extend an existing form component available in your libraries.
  - in Java as a Vaadin component: you will implement the component as a Vaadin component in your LSPS Application freely and deploy the implementation as part of the LSPS Application. You can implement a custom forms component in Java or in the Expression Language.

In addition to a custom form component, you can also implement a custom Grid renderer: a Grid renderer allows you to define how the data in a cell of a Grid Column is rendered.

3.2.2.2.1 Creating a Custom Form Component in the Expression Language

To create a custom form component based on an existing component, use the expression in a custom component definition: Like this, you can, for example, define a custom component that returns a Vertical Layout with a set of components inside.

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

1. In a module, create a Record which has a subtype of the `forms::FormComponent` record as its parent: Pick a component that is the closest to what you require, and add the additional fields for new properties of the components.

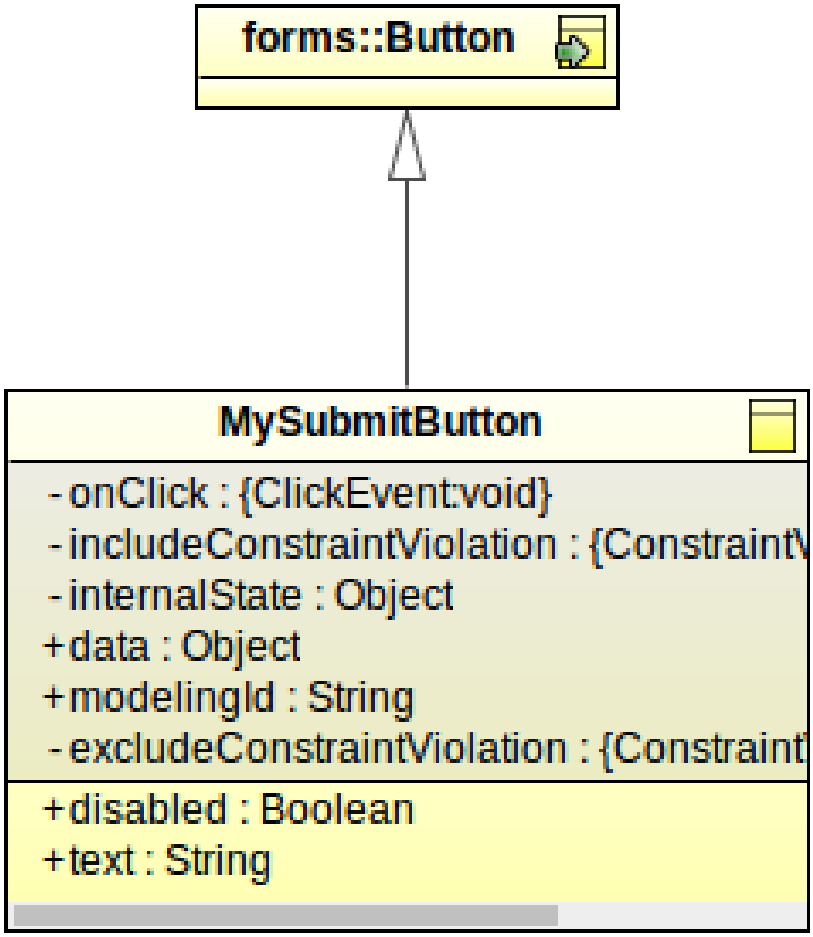


Figure 3.17 Example record of a custom form component

2. Create a custom form component definition file: right-click your Module and go to **New** > **Custom Form Component Definition**.
3. In the *Custom Form Components* section of the editor, click **Add** to create a new form component definition.
4. In Custom Form Component Details define the component properties:
   - Name: name of the form component
   - Icon path: relative workspace path to the icon that should be used in the palette
   - Properties: properties that will be available for editing in the Properties view of the component
     - 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
5. In the *Expression* section enter an expression that returns a custom component.

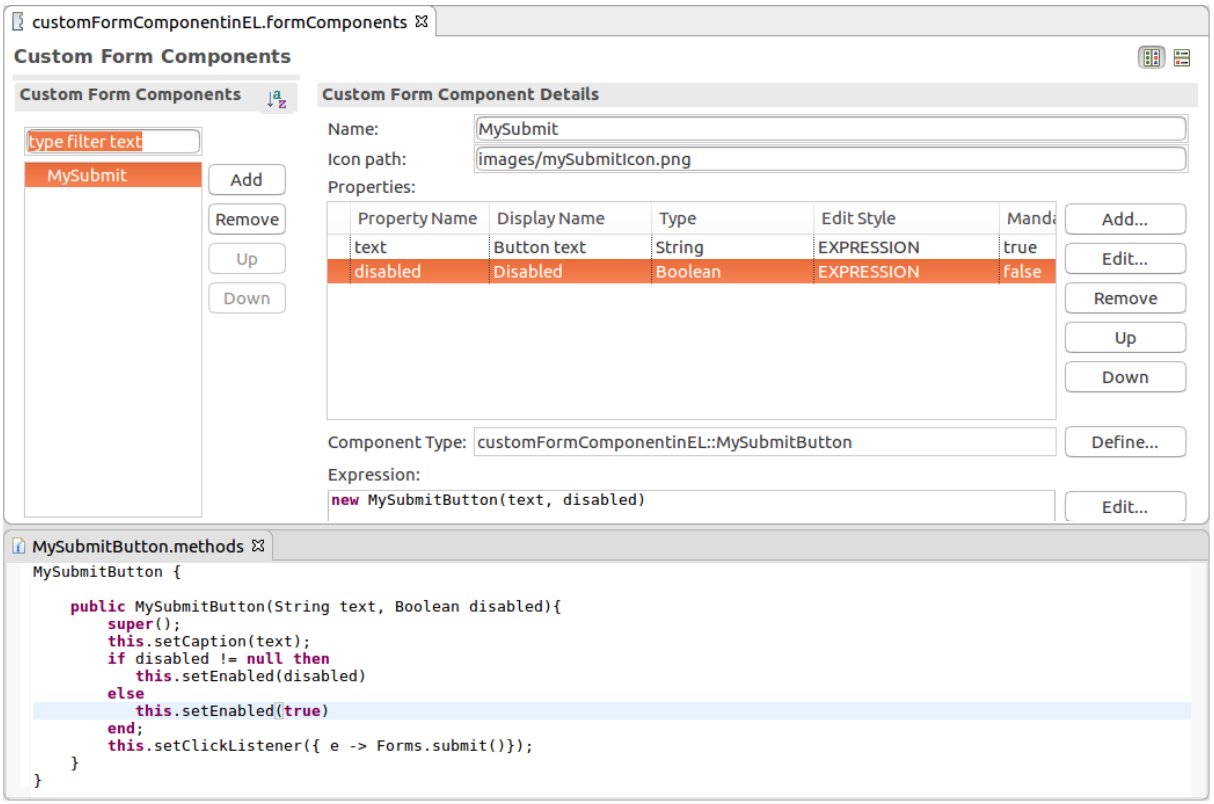


Figure 3.18 A custom component definition and the component record constructor

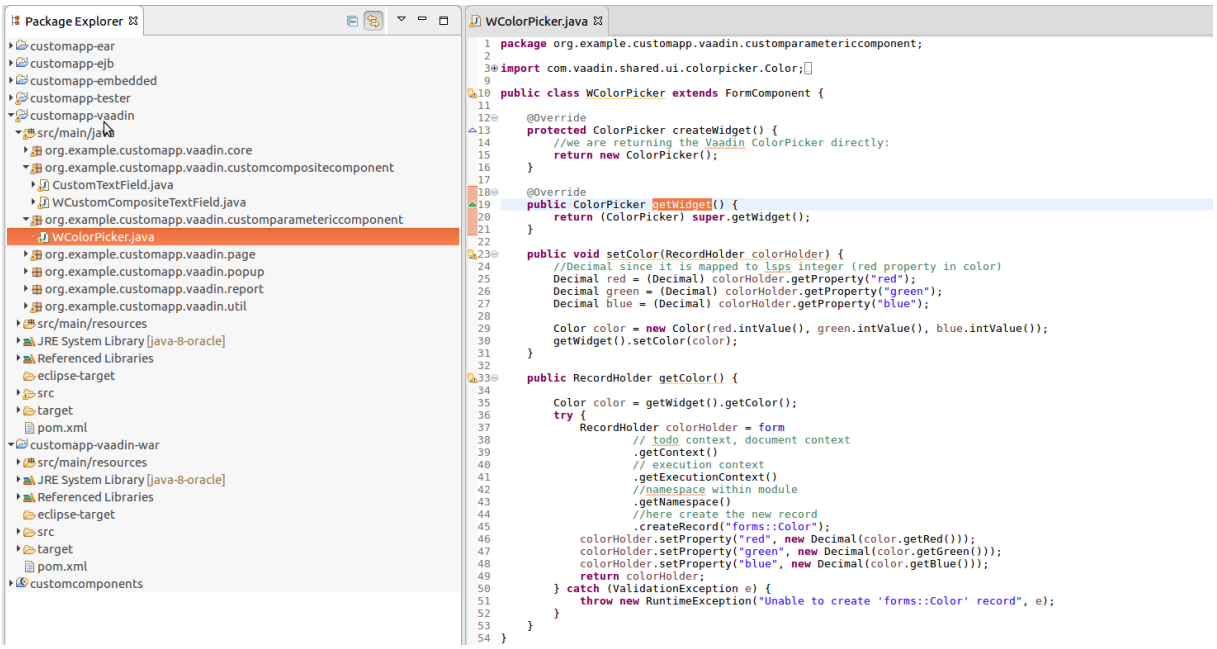
The component is now available in the palette of form definitions editor. Note that if you want to use it in other modules, make sure to they import your Module.

3.2.2.2.2 Creating a Custom Form Component in Java

If an existing form component does not cut it, you can either import an existing Vaadin component or implement your custom Vaadin component from scratch and import your implementation:

1. Switch to Java perspective.
2. If required, create the Vaadin implementation in a package in the <YOUR_APP>-vaadin project.
3. Create a Java class that will connect the LSPS component to its Vaadin implementation:
   - The class must inherit from the FormComponent class or its subclass so you can register it with the component factory.

     You can either inherit directly from the FormComponent or alternatively from the LSPS components, typically prefixed with `W`: For the *forms::TextArea* record, the implementation is *com.whitestein.lsps.vaadin.forms.WTextArea* class. This makes sure your connector class implements all the inherited methods of your component.
   - The class must implement the *createWidget()* method, which returns the Vaadin implementation.
   - Optionally, the class can implement the *getWidget()* method, which calls the getWidget() of the extended LSPS class and casts the returned component to the custom Vaadin component.



Example connector LSPS class

```
 import com.vaadin.shared.ui.colorpicker.Color;
import com.vaadin.ui.ColorPicker;
import com.whitestein.lsps.lang.Decimal;
import com.whitestein.lsps.lang.exception.ValidationException;
import com.whitestein.lsps.lang.exec.RecordHolder;
import com.whitestein.lsps.vaadin.forms.FormComponent;
~
public class WColorPicker extends FormComponent {
~
  @Override
  protected ColorPicker createWidget() {
~
    return new ColorPicker();
  }
~
  @Override
  public ColorPicker getWidget() {
    return (ColorPicker) super.getWidget();
  }
~
  public void setColor(RecordHolder colorHolder) {
    //Decimal since it is mapped to lsps integer (red property in color)
    Decimal red = (Decimal) colorHolder.getProperty("red");
    Decimal green = (Decimal) colorHolder.getProperty("green");
    Decimal blue = (Decimal) colorHolder.getProperty("blue");
~
    Color color = new Color(red.intValue(), green.intValue(), blue.intValue());
    getWidget().setColor(color);
  }
~
  public RecordHolder getColor() {
    Color color = getWidget().getColor();
    RecordHolder colorHolder = getNamespace().createRecord("forms::Color");
    colorHolder.setProperty("red", new Decimal(color.getRed()));
    colorHolder.setProperty("green", new Decimal(color.getGreen()));
    colorHolder.setProperty("blue", new Decimal(color.getBlue()));
    return colorHolder;
  }
}
```

4. Once the Vaadin implementation is ready, create the definition of the custom form component so you can use it in forms definitions, do the following:

   (a) In a GO-BPMN module, create a Record that will represent your Vaadin component:

   - The supertype of the Record must be the **forms::FormComponent** record or its child Record↩ : this will typically be the same component as your Vaadin implementation extends. If your Vaadin component extends the *com.vaadin.ui.CustomComponent* class as in the case of composite components, your record should have *forms::FormComponent* as its super type.
     It is not recommended to add any additional record fields since the record serves to reflect a Vaadin component in LSPS; Vaadin components are intended for presentation, not for business logic.

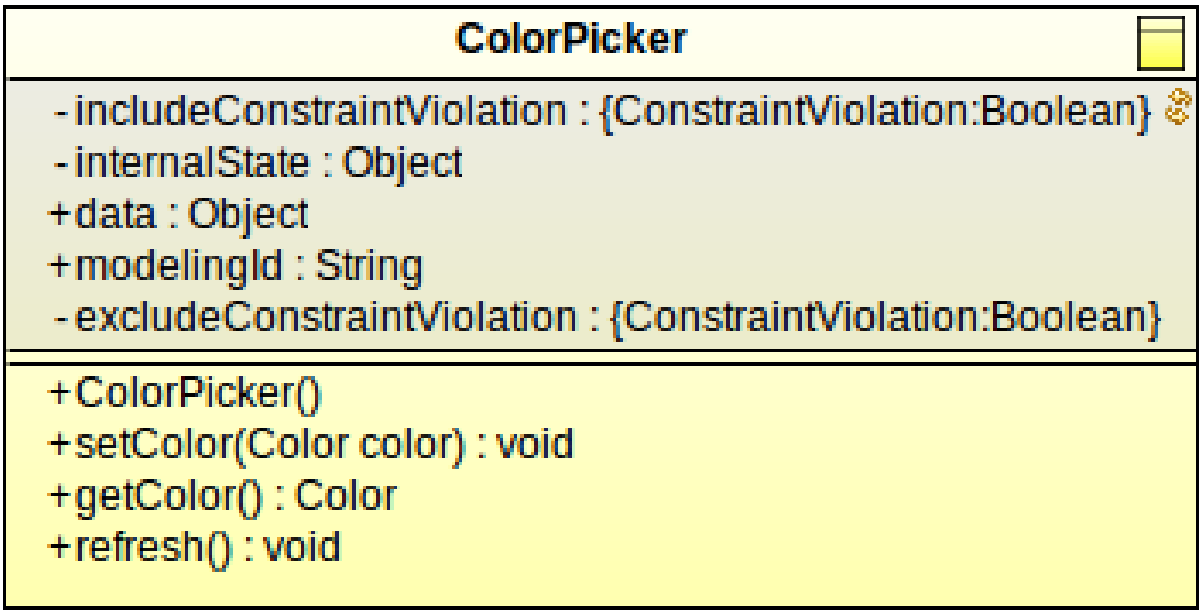


Figure 3.19 Custom component record

   - The Record must implement the methods of its interfaces, and override inherited methods as applicable, and define methods that call their Vaadin implementation: Such methods will use `call()` to call the method on its Vaadin implementation. When implementing a layout component, that is, components that can hold child components, your record should implement the **HasChildren** interface.

```
   ColorPicker {
~
    public void setColor(Color color){
      //calls the setColor(color) method on the Vaadin implementation:
       call("setColor", [color]);
    }
~
    public Color getColor(){
      //calls the getColor(color) method on the Vaadin implementation:
        call("getColor",[]) as Color;
    }
~
    public void refresh() {
      getColor();  // make sure that at least ObjectReference is set as a property
      call(#"refresh", null)
  }
}
```

   (b) To include the component in the palette of the form editor, create a custom component definition in a Custom Form Component definition file in your Module.

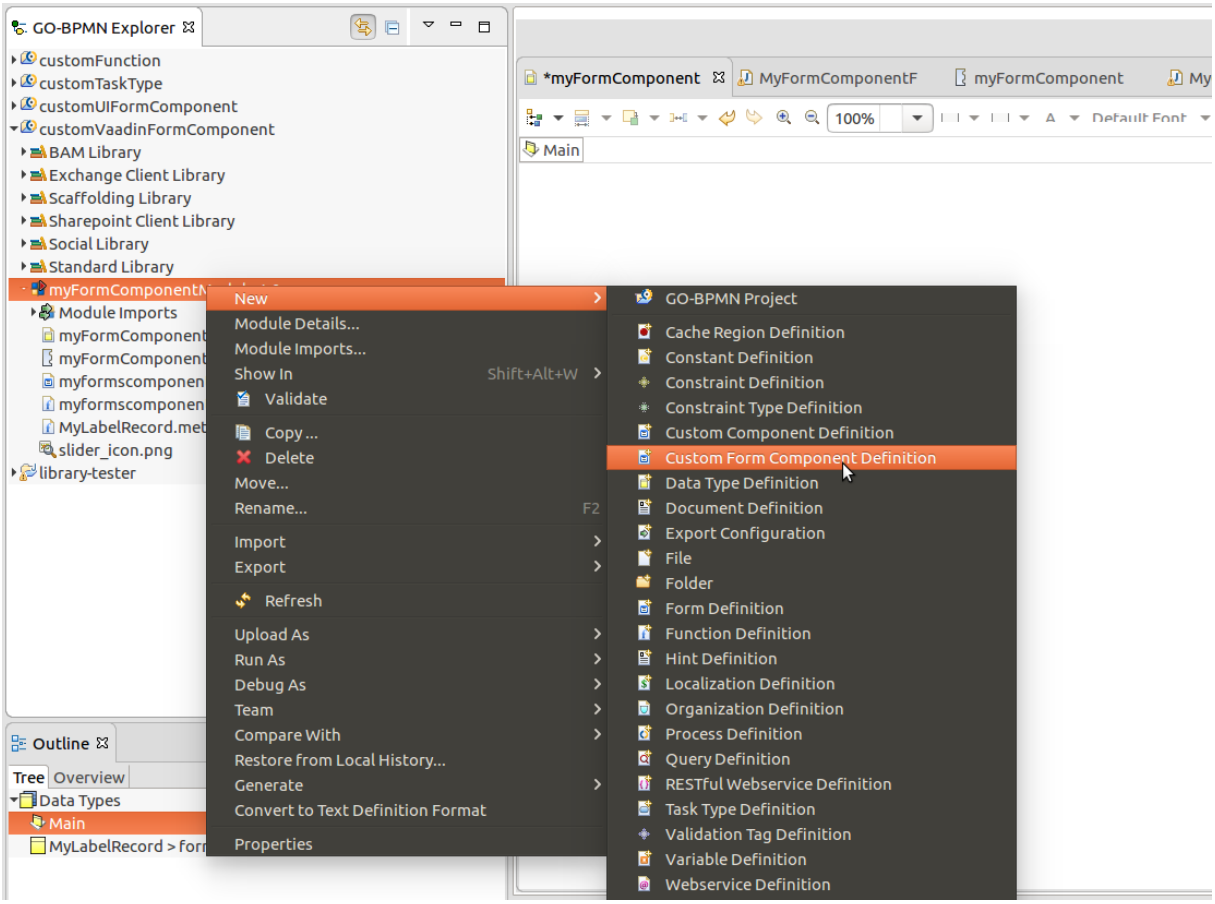


Figure 3.20 Creating custom form component definition file

(c) In the definition file, create a new custom component declaration:

i. Set the *Component Type* to the component record.

ii. Define properties that will be edited in the Properties view of your custom component in the Properties area:

- 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.
  - **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
- Property displayed in editor: if set to true, the value of the property is displayed in the Form editor (only one property can be displayed in the component graphical depiction).

iii. In the Expression field, define an expression that will return the instance of the record, typically the constructor of the record that takes the defined properties, such as `new MyLabel(text)`. Implement handling of the properties in the Expression.

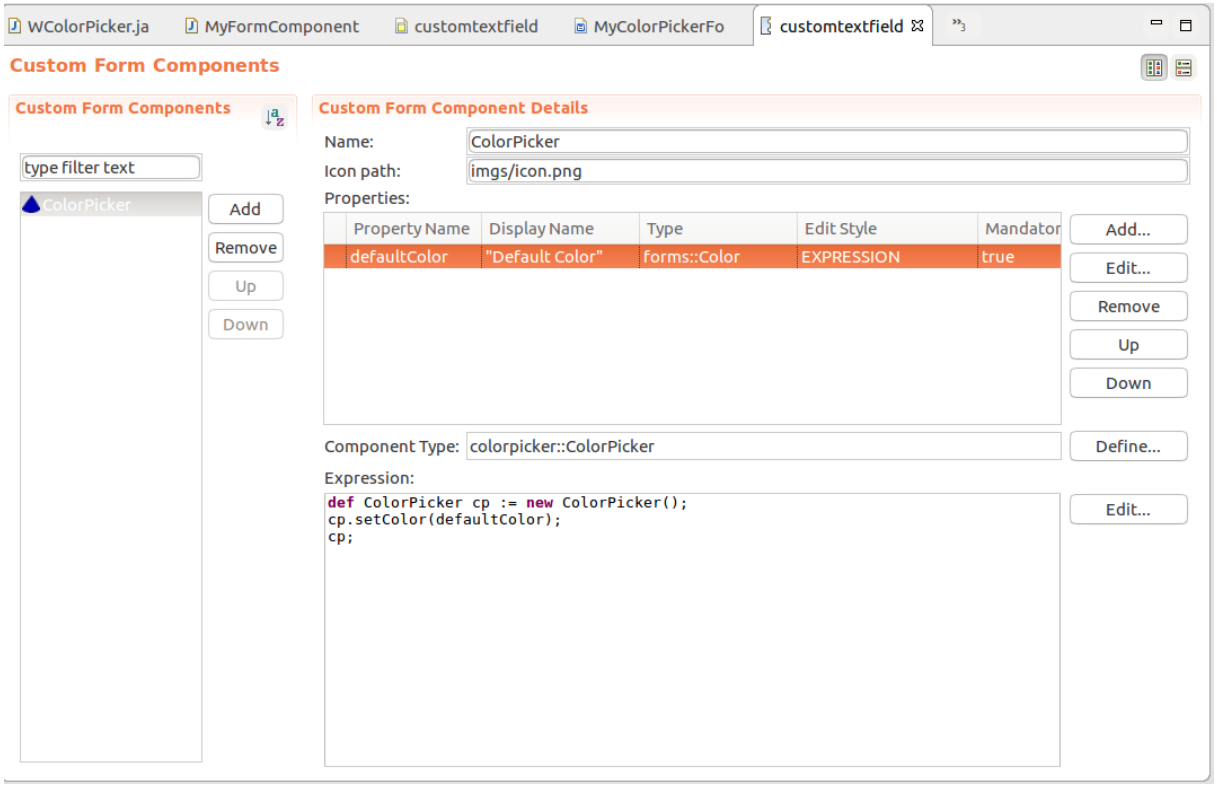


Figure 3.21 Custom form component definition with a property

5. In your LSPS Application, create the LSPS implementation that will connect the LSPS record and its Vaadin implementation and vice versa: Modify the `MyFormComponentFactory` class to return the LSPS implementation when the Record of your component is requested.

```
  @Override
public FormComponent create(Variant.RecordVariant def) {
  final String type = def.getTypeFullName();
  //modified code (returns the WColorPicker for the record ColorPicker):
  if (type.equals("colorpicker::ColorPicker")) {
    return new WColorPicker();
  }
  return super.create(def);
}
```

6. Rebuild and deploy your application.

You can now use the custom components in your forms and distribute it to other users as part of a library.

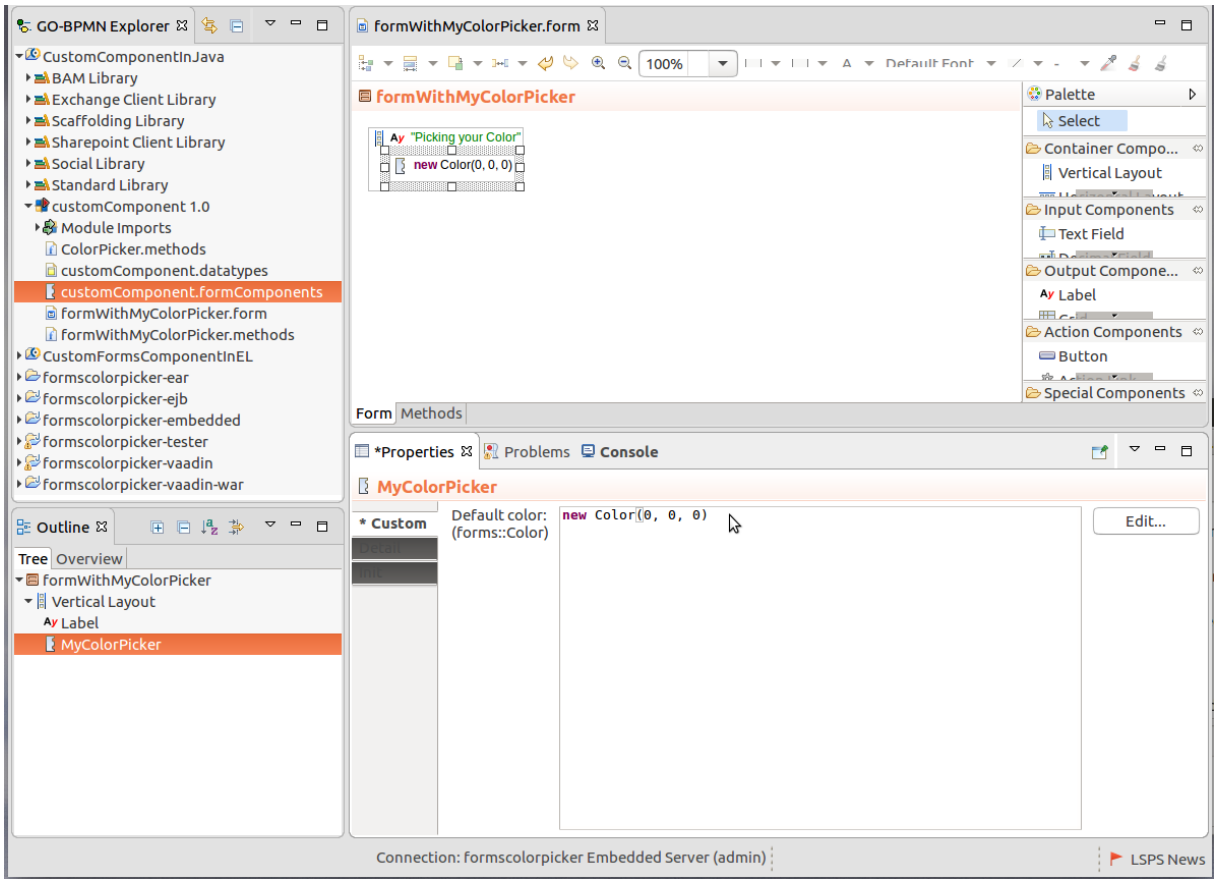


Figure 3.22 Custom component in a form definition

3.2.2.2.1 Saving of a Custom Form Component

In the LSPS Application, the user can save a to-do or document with a ui form for later editing. To include data of your custom component when a to-do or a document is saved, you need to include the data on save and recover it when required:

1. To save presentation data which is not part of the component record, do the following:

   (a) In your connector class, in the example, WColorPicker, create constant-property variables for each property values to be saved.

   ```
   private static final String STATE_CURRENT_COLOR = "ColorPicker_currentColor";
   ```

   (b) Override `writeInternalState()` so it saves these properties in the internal state of the component record.

   ```
   @Override
   protected void writeInternalState(Map<String, Object> state) {
     super.writeInternalState(state);
     state.put(STATE_CURRENT_COLOR, getWidget().getColor());
   }
   ```

   (c) Override `restoreInternalState()` so it applies the properties on restore:

   ```
   @Override
   protected void restoreInternalState(Map<String, Object> state) {
     super.restoreInternalState(state);
     Color color = (Color) state.get(STATE_CURRENT_COLOR);
     if (color != null) {
   ```

```
      getWidget().setColor(color);
    }
  }
```

2. If applicable, to save the child components of the custom component, do the following:

   (a) Add a list of the components to `writeInternalState()`:

```
 @Override
protected void writeInternalState(Map<String, Object> state) {
  super.writeInternalState(state);
~
  writeChildComponents(state);
}
protected void writeChildComponents(Map<String, Object> state) {
  state.put(STATE_COMPONENTS, getComponents());
}
public ListHolder getComponents() {
  final List<Object> children = new ArrayList<>(getWidget().getComponentCount());
  for (FormComponent child : getChildren()) {
    children.add(form.getDef(child).get());
  }
  return form.getContext().getExecutionContext().getNamespace().createList(children);
}
```

   (b) Create empty instances of the child components in `restoreInternalState()` so the saved data has a component which it can populate.

```
 protected void restoreInternalState(Map<String, Object> state) {
  super.restoreInternalState(state);
~
  restoreChildComponents(state);
~
}
protected void restoreChildComponents(Map<String, Object> state) {
  final LspsContextHolder context = form.getContext();
~
  ListHolder children = (ListHolder) state.get(STATE_COMPONENTS);
~
  for (Object child : children) {
    RecordHolder recordHolder = (RecordHolder) child;
    Variant.RecordVariant record = Variant.wrap(recordHolder, context).record();
    form.createComponent(record);
~
    addComponent(recordHolder);
  }
}
```

3.2.2.2.3 Creating a Custom Grid Renderer

Columns of the Grid component define a renderer which is used to render the data in each row of the Column. While a variety of renderers are available by default, you can define your own renderer if necessary.

To implement your custom Grid renderer, do the following:

1. In a GO-BPMN module, create a record that will represent your renderer: The supertype of the record must be the **forms::Renderer** record or its child record.

Figure 3.23 Custom renderer record

2. Switch to Java perspective to work with the Java part of the implementation:

   (a) Create the Vaadin component implementation in <YOURAPP>.vaadin.util.<RENDERER> in the <YOUR_APP>-vaadin project. In the example, we are using the already available ProgressBar↩ Renderer.

   (b) Create a Java class that will connect the LSPS renderer to its Vaadin renderer. The class must extend your Vaadin renderer.

   Example renderer class:

```
 public class WProgressBarRenderer extends ProgressBarRenderer {
~
  private final WGrid grid;
~
  public WProgressBarRenderer(WGrid owner, RecordVariant rendererDef) {
    grid = owner;
    rendererDef.checkSubtypeOf("gridModule::ProgressBarRenderer");
  }
}
```

3. In your Application User Interface, modify the `createRenderer()` method of `MyFormComponent↩ Factory` class to return the LSPS implementation when the renderer record is requested:

```
 public Renderer<?> createRenderer(WGrid owner, Variant.RecordVariant rendererDef) {
  //add the if with the renderer record and the call to return the progress bar via the conne
  if (rendererDef.getTypeFullName().equals("gridModule::ProgressBarRenderer")) {
    return createProgressBarRenderer(owner, rendererDef);
  } else {
    return super.createRenderer(owner, rendererDef);
  }
}
~
protected Renderer<?> createProgressBarRenderer(WGrid owner, Variant.RecordVariant rendererDe
  return new WProgressBarRenderer(owner, rendererDef);
}
```

4. If your renderer passes parameters to its Vaadin counterpart, make sure the data types of the parameters are compatible. You can check the compatibility of data types in Data Type Mapping in the Expression Language documentation. If the data types of the parameters passed from LSPS to the renderer implementation and vice versa are not compatible, do the following:

   (a) Implement the converter as a class that implements the Vaadin *Converter* interface. for your renderer.

```
 //This is example implementation of a converter of
public class DoubleToDecimalConverter implements Converter<Double, Decimal> {
~
  /**
   * serialVersionUID
   */
  private static final long serialVersionUID = 1L;
~
```

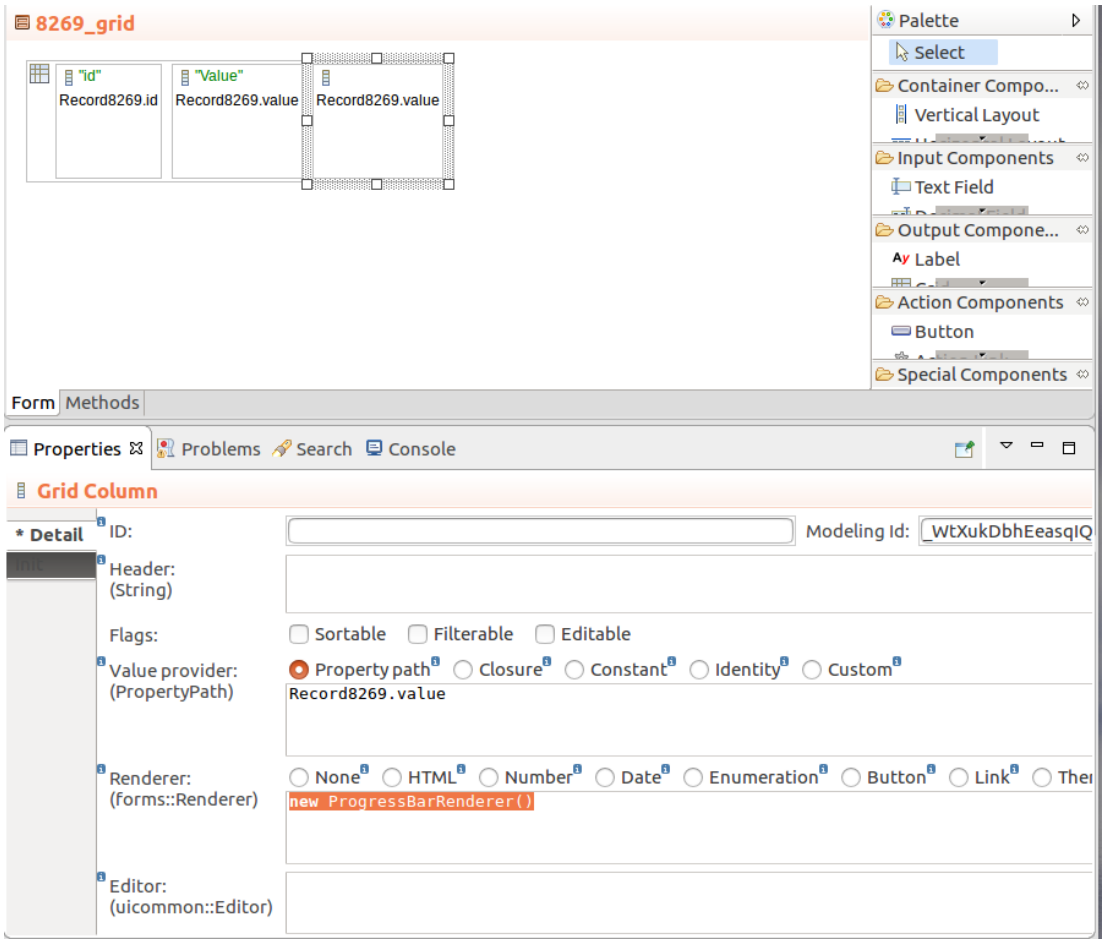
```
  @Override
  public Decimal convertToModel(Double value, Class<? extends Decimal> targetType, Locale
    return value == null ? null : new Decimal(value);
  }
~
  @Override
  public Double convertToPresentation(Decimal value, Class<? extends Double> targetType,
    return value == null ? null : new Double(value.toString());
  }
~
  @Override
  public Class<Decimal> getModelType() {
    return Decimal.class;
  }
~
  @Override
  public Class<Double> getPresentationType() {
    return Double.class;
  }
}
```

(b) Implement the `createConverterForRenderer()` method in the *MyFormComponentFactory()* class.

```
 @Override
public Converter<?, ?> createConverterForRenderer(WGrid owner, Variant.RecordVariant rend
  if (rendererDef.getTypeFullName().equals("gridModule::ProgressBarRenderer")) {
    //return new StringToDecimalConverter();
    return new DoubleToDecimalConverter();
  } else {
    return super.createConverterForRenderer(owner, rendererDef);
  }
}
```

5. Rebuild and deploy your application as required.

You can now use the custom renderer in your grid columns. Consider distributing the renderer as part of a Library.



3.3 Working with a Model

At some point you will want to work with data of model instances from your application. When manipulating a model instance, make sure to take into consideration:

- the `execution context`
- and the execution level of your object's context.

3.3.1 Execution Levels

An execution context can exist on different execution levels so one context element can have different values in context versions on different levels.

This mechanism serves to separate possibly transient data from the "real" data: the real data exist in the contexts on the **base level** or `0` level, the execution level where model instances are created. Consequently, for example, changes on shared records in contexts on this level are reflected instantly in the database.

From a custom object, you can create contexts on sublevels: on the sublevel context you work with copies of the contexts and their data. Once happy with the changes, you can merge your changes into its super contexts. Sublevels are designated with an additional digit added to the original level digit separated by a colon, for example `0:1`, `0:1:1`, `0:1:2` etc.

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

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

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()`.

3.3.1.1 Creating an Execution Level

To create an execution level, call `com.whitestein.lsps.engine.state.xml.EvaluationLevelUtils.nextSublevel(String level, ModelInstance modelInstance)`.

Note that the level does not contain any data when created: the data of non-base levels are loaded when you request an entity that is not present on that level. The system attempts to load it from the immediate parent context and, if not available, it continues to request higher context levels until it reaches the base level. It is *when you change the context data* that a context on a non-base level stores data.

3.3.1.2 Merging an Execution Level

To merge all changes from a non-base context into the context on the parent level, use the `com.whitestein.lsps.engine.lang.EvaluationLevelMerger.mergeLevel(String level)`.

On merge, the system checks for data conflicts. For example, if a variable is changed in two contexts on the second level (the one above the base level) and both context are merged to the base-level context, 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 one context and property P2 of the same record R is changed in another context no conflict is detected during merge.

3.3.1.3 Cleaning an Execution Level

To clean changes in a level, call one of the *com.whitestein.lsps.engine.state.xml.EvaluationLevelUtils* methods:

- `cleanDataOfEvaluationLevel(ModelInstance modelInstance, String level)`
- `cleanDataOfLevelAndSublevels(ModelInstance, String)`.

Note that the `cleanDataOfLevelAndSublevels(ModelInstance, String)` method cleans changes in the given level and in all child levels.

- To remove a context from a level, use one of the methods:
  - `removeDataOfEvaluationLevel(ModelInstance, String level)`
  - `removeDataFromLevelAndSublevels(ModelInstance, String level)`: removes all entities that belong to the execution level and its sub-levels

3.3.1.4 Checking for Changes on an Execution Level

To check if there are changes in the non-base levels, call `com.whitestein.lsps.engine.state.xml.↩ ModelInstance.isDirty()`.

3.3.2 Creating a Record

To create a record instance in the execution context of your custom object, use the `createRecord()` method of the namespace.

```
 //custom form component implementation (the class extends com.whitestein.lsps.vaadin.forms.FormCo
RecordHolder colorHolder = getNamespace().createRecord("forms::Color");
```

```
 //custom task and function:
factory.createRecord(
  "GoogleCalendar::GoogleCalendarEvent",
  new HashMap<String, Object>() {
    { put("title", event.getTitle().getPlainText());
      put("content", event.getTextContent().getContent().getPlainText());
      put("date", new Date(event.getTimes().get(0).getStartTime().getValue()));
    }
  }
);
```

3.3.2.1 Generating Classes and Interfaces for Records

To work more efficiently with record instances and their types in the LSPS Application, generate the Java wrapper classes for the types and use these classes rather than using the `RecordHolder` class to work with Records.

Example of difference in coding

```
 // original access via context:
RecordHolder record = ctx.getNamespace().createRecord("core::ConstraintViolation");
record.setProperty("message", msg);
~
// better with generated Java class for records:
ConstraintViolationRecord better = new ConstraintViolationRecord(record);
better.setMessage(msg);
```

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

- subpackage for each module
- class for records in their respective subpackage
- interface for record classes
- RecordWrapperFactoryImpl for individual modules

To generate Java classes and interfaces for records, do the following:

1. Select the record and configure the properties of the generation:

(a) Open the *Java* tab in the Properties view.

(b) To generate a class, select the **Generate Java class** option and the target Java class name.

(c) Optionally, set the *Java record type constant name* that will hold the path to the record.

2. Open the Properties view of individual record fields and on the *Java* tab, set the properties of the methods generated for the field.

3. Right-click the module or project with the records and go to **Generate** > **Record Java Sources** and, in the dialog, define the export properties:

   - Source folder: target src folder in a Java project
   - Package: target package name

   The Java sources are generated into a subpage with the name of the module.

   - 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
   - Generate interfaces: select to generate also interface for the record classes
   - Additional record types: other records that should be included in the operation
     The parameter is primarily intended for inclusion of library records.
   - Selected modules: the system generates Java classes for the selected Modules and any dependencies

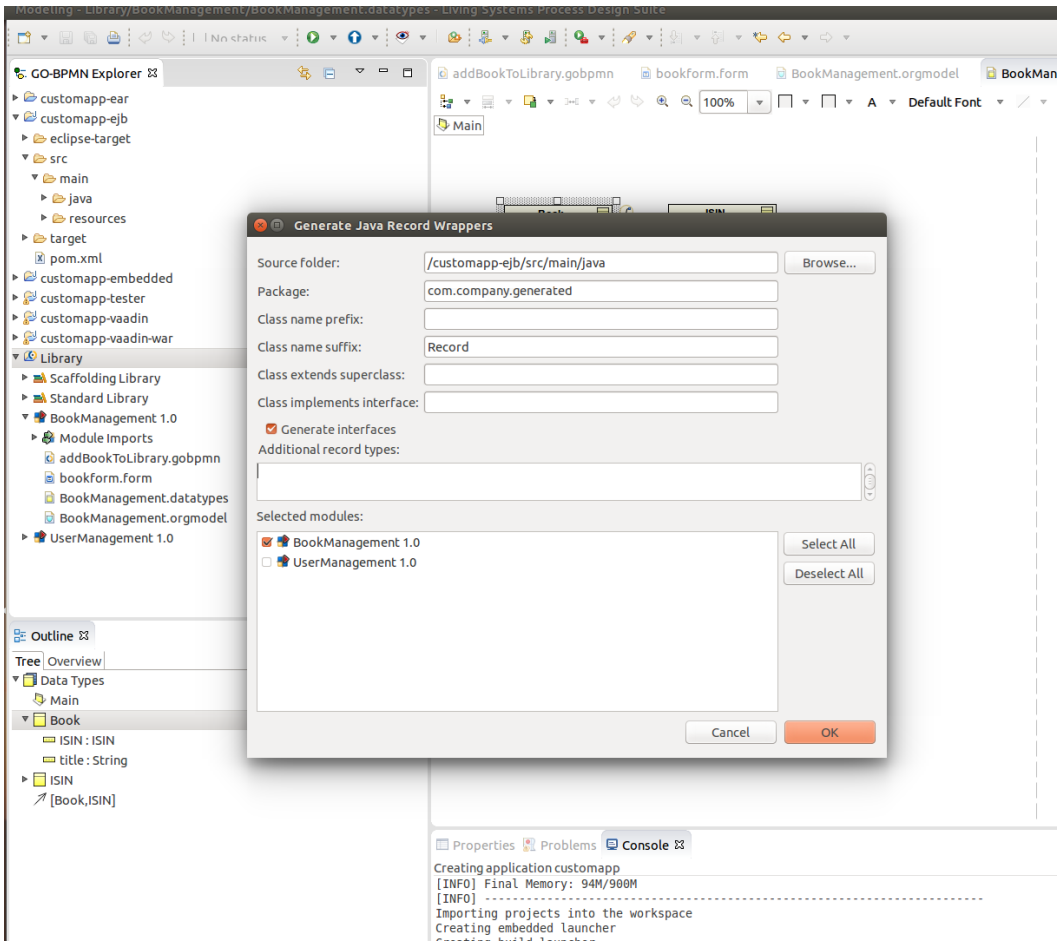


Figure 3.24 Generating Java record wrappers

4. Refresh the project with the target src directory.

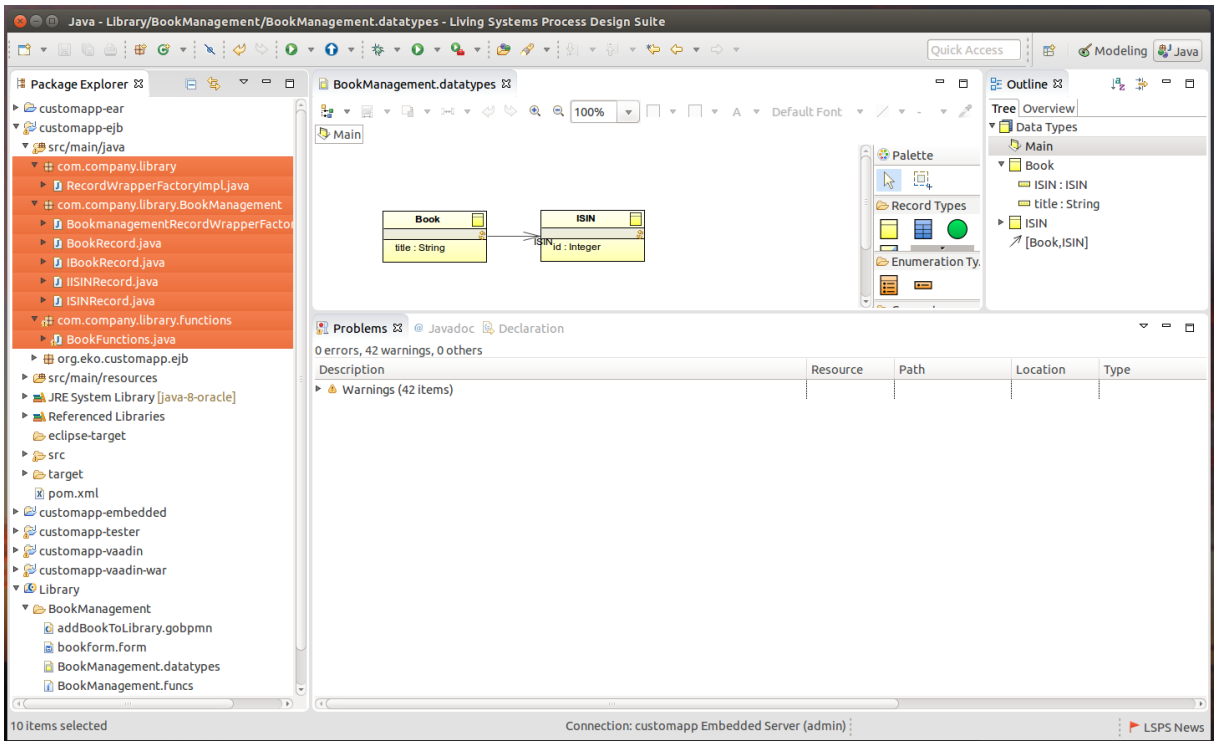


Figure 3.25 Generated Java record wrappers

Implementation of a custom function and its definition

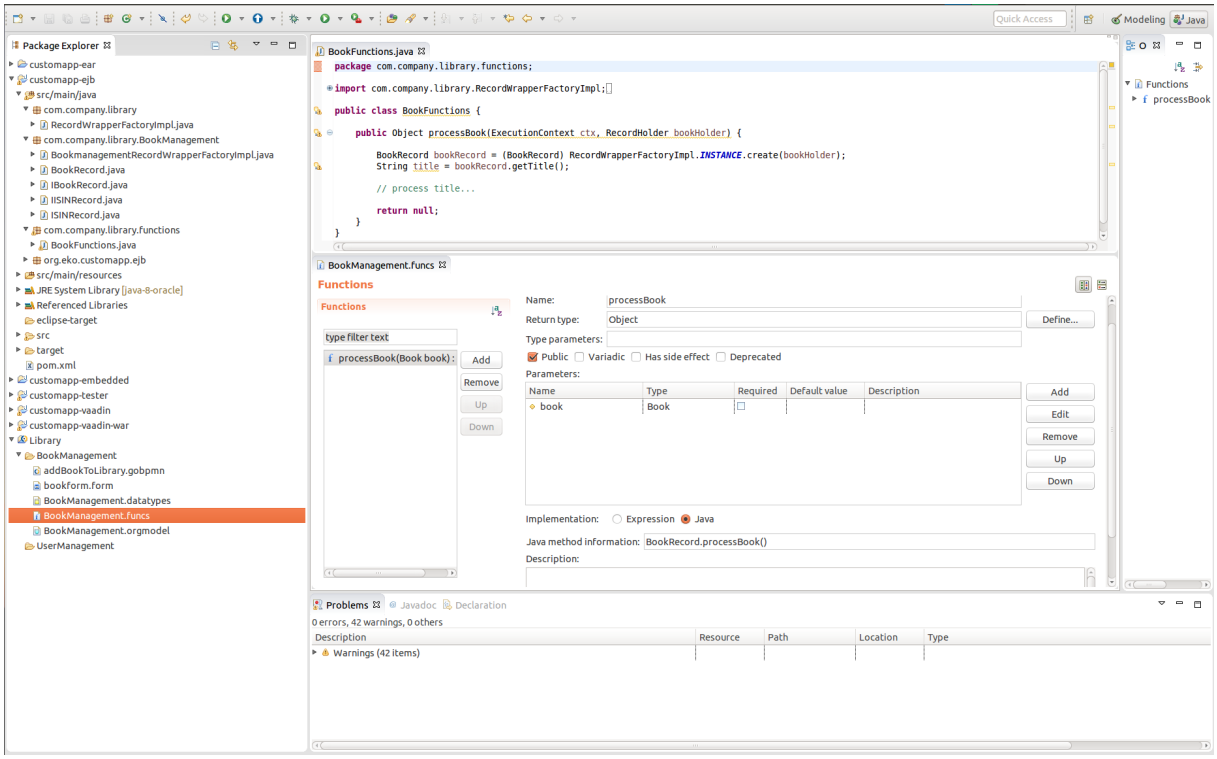


Figure 3.26 Custom Function implementation and definition

3.3.2.2 Checking a Record Constraint

To work with constraints on a record, use the following methods:

- **getConstraints** 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** returns a validation tag with the given qualified name (may be simple or ∗ full).

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

3.3.3 Throwing a Signal

A Signal is a special object used for communication within a model instance or with another model instance.

You can produce signals in your custom object using the server API. To catch signals, use the `Signal Start Events` or `Catch Signal Intermediate Events`.

> **Note:** You can also use `signal modeling elements` to work with signals in your models and signal-related standard-library functions, such as `sendSignal()`.

You can work with signals as follows:

- To send a *synchronous signal to the parent model instance of the custom object*, use the *addSignal()* call of the object's context. **Sending a signal to the current model instance from a custom task type**

```
//start method of a custom task
 @Override
 public Result start(TaskContext context) throws ErrorException {
   Decimal d = (Decimal) context.getParameter("number");
   Integer i = d.intValueExact();
   //adding the signal to the task type context:
   context.addSignal(
      "prime check output:"
      + System.out.println(org.apache.commons.math3.primes.Primes.isPrime(i))
   );
   return Result.FINISHED;
 }
```

- To send a *signal to another model instance*:

  1. Define and register your object as an EJB.
  2. Inject *CommunicationService* into your bean object.
  3. Send a signal with the `sendSync()` or `sendAsynch()` method of the CommunicationService bean. **Example sendSync() call from the start method of a custom task type**

```
@EJB
 private CommunicationService communicationService;
 ...
 @Override
 public Result start(TaskContext context) throws ErrorException {
   long modelInstanceId = 71000;
   String signal = "signal";
   SignalMessage signalMessage = new SignalMessage(
       //sender of the signal:
       Identifier.ofModelInstance(context.getModelInstance().getId()),
       //receiver of the signal:
       Identifier.ofModelInstance(71000), new ObjectValue(signal));
   try {
     //sending the signal:
     communicationService.sendSync(signalMessage);
   } catch (ModelInstanceNotFoundException | InvalidModelInstanceStateException e) {
     e.printStackTrace();
   }
   return Result.FINISHED;
 }
```

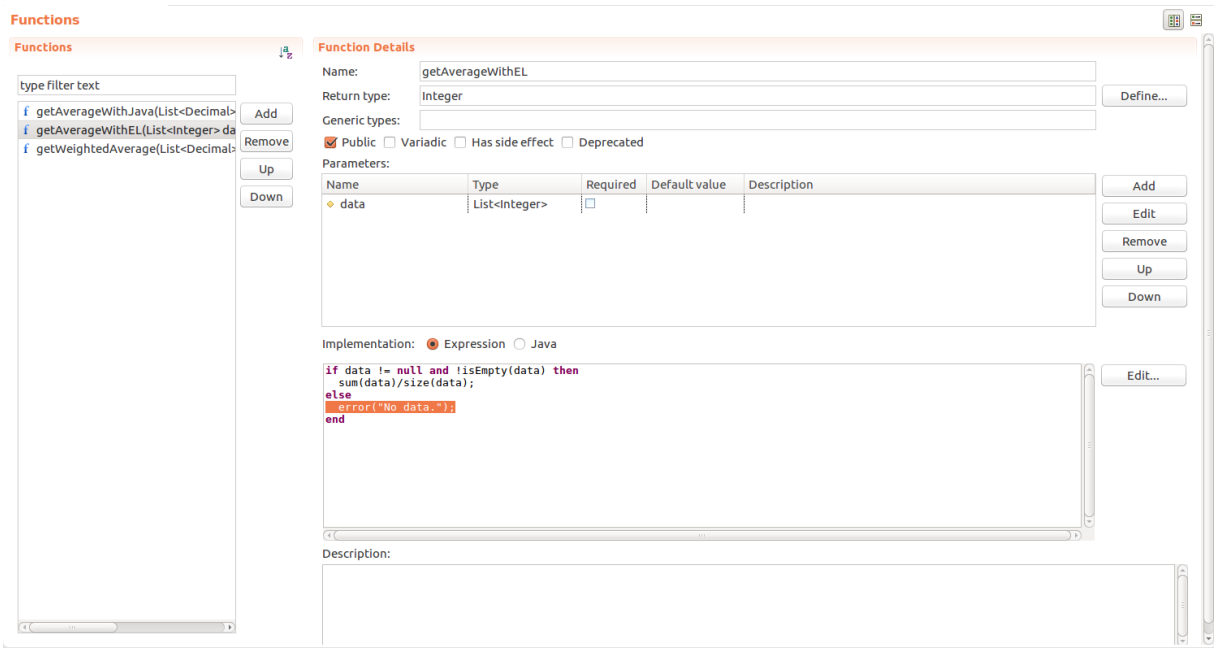
3.3.4 Throwing an Error

You can throw `errors` directly from your custom objects.

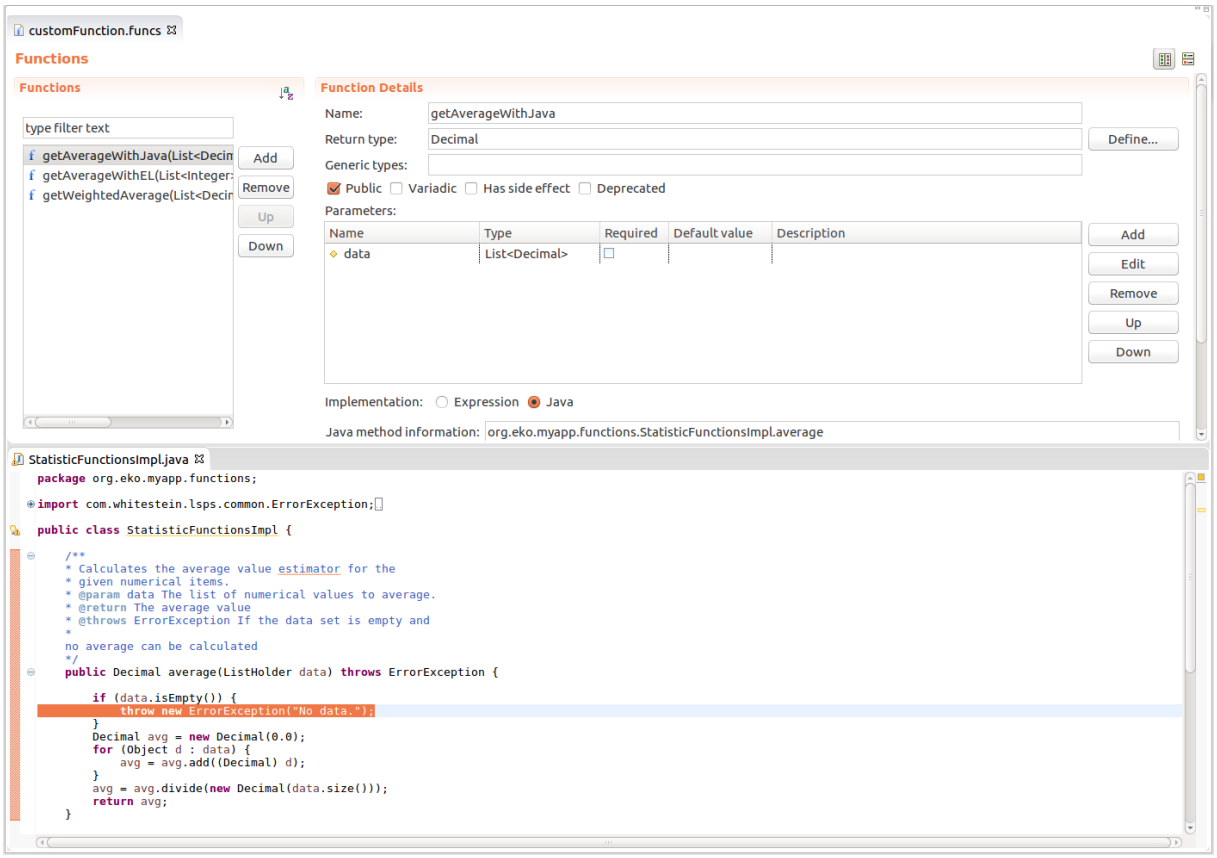
To catch error, use the `Error Start Events` or `Error Intermediate Events`.

To throw an error, do the following:

- In implementation in the Expression Language, use the `error(<error_code>)` function.



- In implementation in Java, throw `com.whitestein.lsps.common.ErrorException`.



3.3.5 Creating Hooks on Model Execution

To perform an action always when an instance of a model is created, started, or finished, do the following:

1. Create a stateless bean that implements the ModelInstancePlugin interface.

```
public class MyModelInstancePlugin implements ModelInstancePlugin {
  ~
  @Override
  public void onCreate(CreateCommand cmd, ModelInstanceEntity modelInstance) {
    System.out.println(">>>>>>>>>>>>>>>>>> This is on create hook <<<<<<<<<<<<<<<");
  }
...
```

2. Register the bean with the ComponentServiceBean:

```
 @EJB(beanName = "MyModelInstancePlugin")
private ModelInstancePlugin myMIPlugin;
~
@Override
protected void registerCustomComponents() {
  register(myMIPlugin, ModelInstancePlugin.class);
}
```

3.3.6 Invoking the Command-Line Console

To use the command-line tool from your application, do the following:

1. Include the `mconsolecl` dependency in your pom.xml:

```
<dependency>
   <groupId>com.whitestein.lsps.mconsolecl</groupId>
   <artifactId>lsps-mconsole-cl</artifactId>
   <version>${project.version}</version>
</dependency>
```

2. Call the main method with the cli command as its String[] argument.

```
com.whitestein.lsps.mconsolecl.Main.main(new String[]{"arg 1", "arg 2"});
```

For details about the console are available in the Management documentation.

3.4 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.

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

> **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*: the custom listener will get the data for the field or related shared record.

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

3.4.1 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.

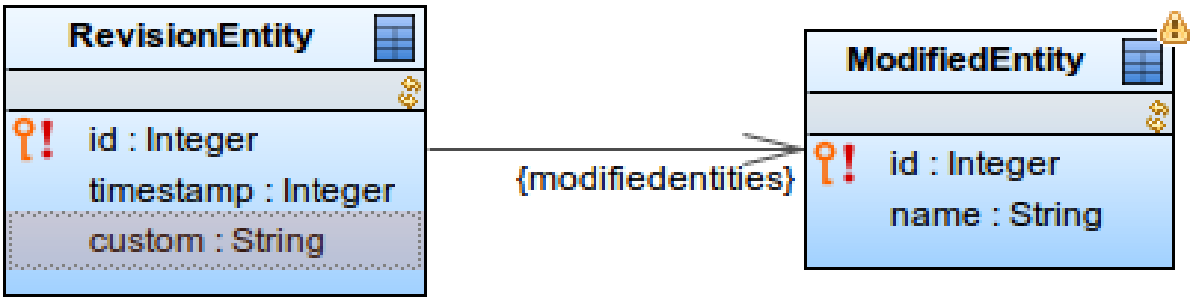


Figure 3.27 Revision entity record with the custom field

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.

3.4.2 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.

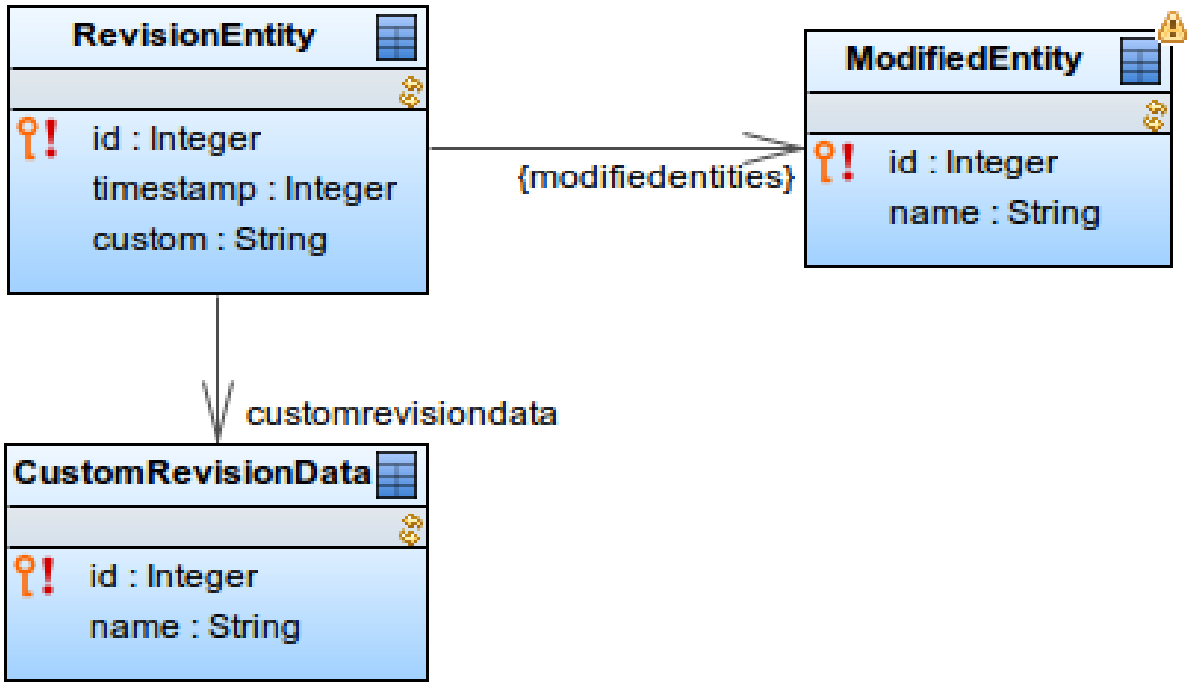


Figure 3.28 Revision entity record with the custom field and a related record

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.

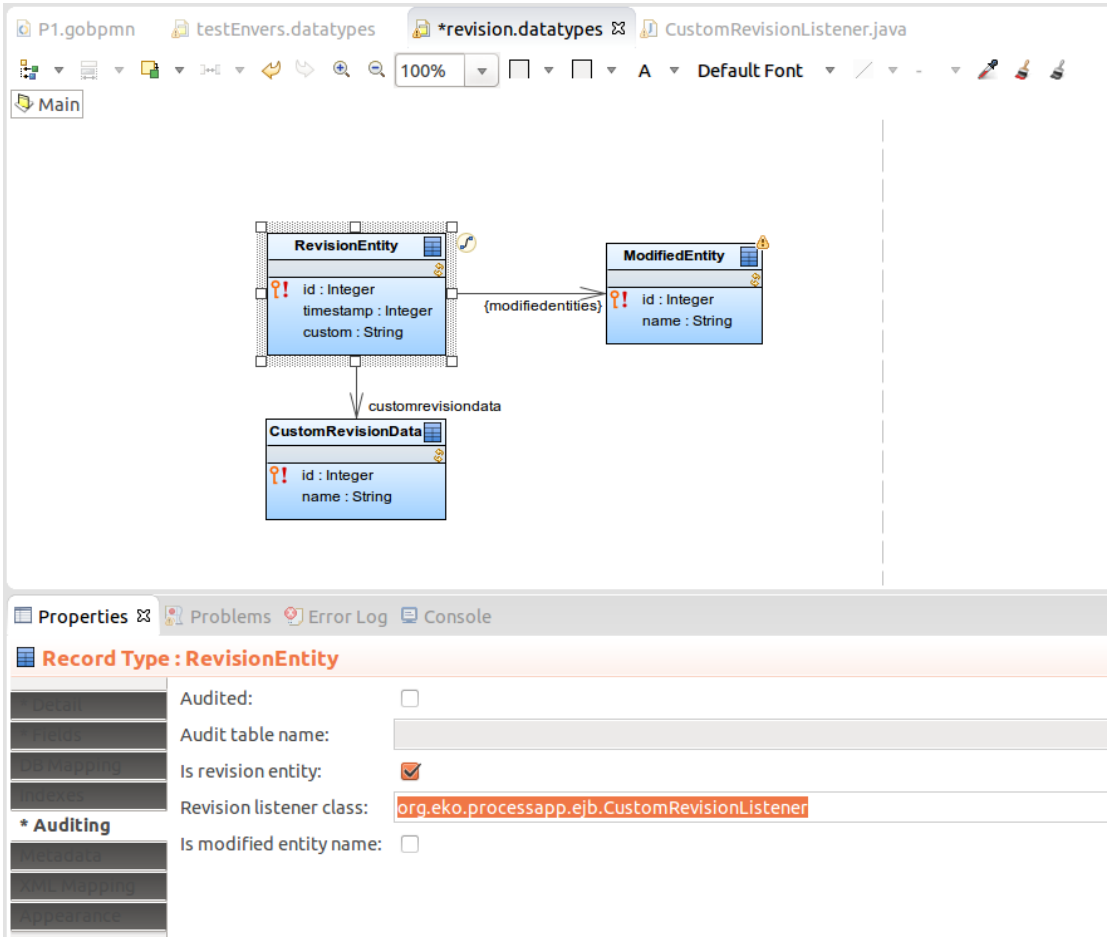


Figure 3.29 Revision Entity shared Record implemented by a custom RevisionListener class

3.4.3 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 4

Build

The LSPS Application build is a standard maven build and as such you can manage all its dependencies.

Once you have adapted the build, you can test it on SDK Embedded Server. For deployment to other servers, build the application EAR.

4.1 Building and Deploying LSPS Application during Development

To simplify the deployment during development of the LSPS Application, the SDK comes with SDK Embedded Server and its launcher: the launcher is created when you generate the LSPS Application along with the Maven build configuration for the application build. The launcher runs 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.

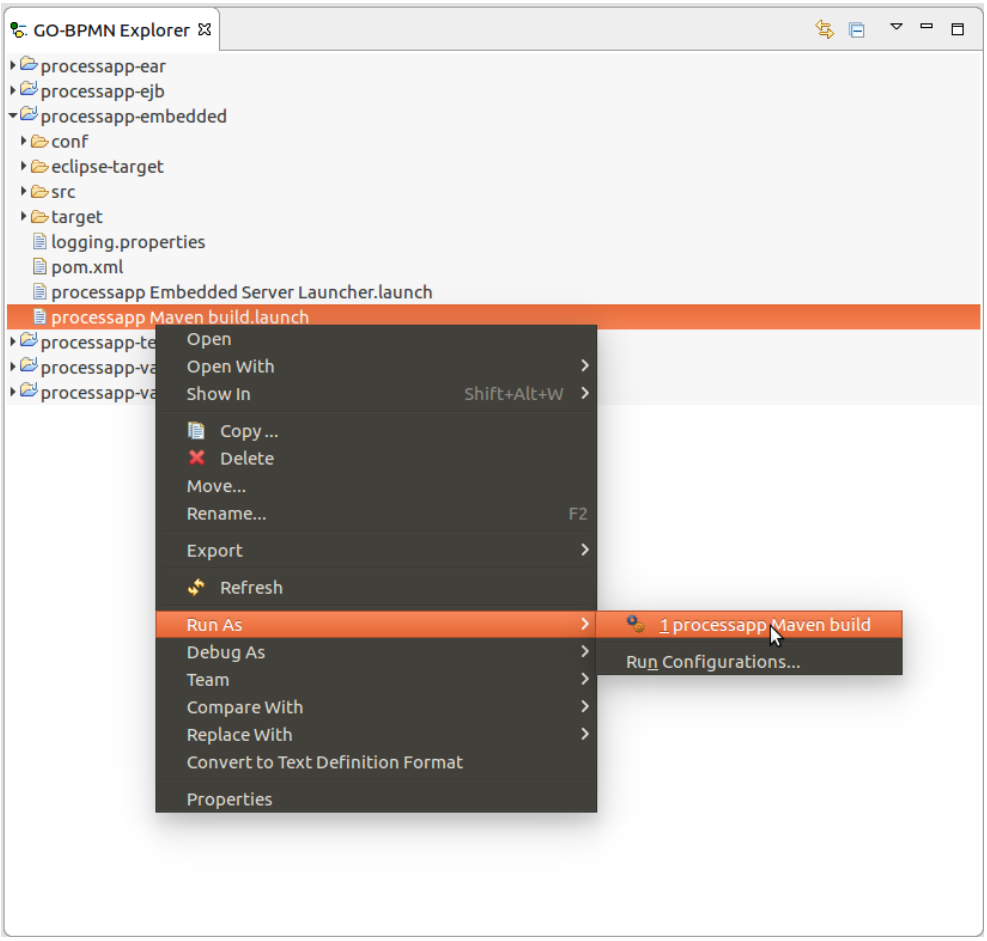
> **Important:** SDK Embedded Server is a different server from PDS Embedded Server.

You can run the server with the application in normal or debug mode:

- When launched in debug mode, changes in the source code of the application are reflected immediately.
- When launched in normal mode, changes are reflected after re-build the application, and restart of the Embedded server.

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

1. Build the application: Right-click the maven build launcher configuration in the `<YOUR_APP>-embedded` project and select **Run As** > <**YOUR_APP**> **Maven Build**.



Note: Next time build the application from the Run menu or from the drop-down of the Run icon  on the toolbar.

2. Run SDK Embedded Server with the application to check your customizations: Right-click the Embedded Server Launcher configuration in the <YOUR_APP>-embedded project and select **Run As** > <**YOUR**↩ **_APP**> **Embedded Server Launcher**.

Note: Next time run the application server from the Run menu or from the drop-down of the Run icon  on the toolbar.

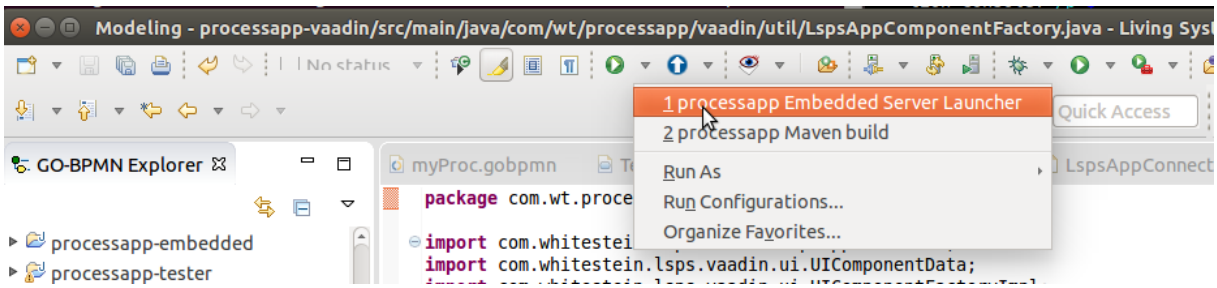


Figure 4.1 Running the application with the generated launcher

4.2 Building the LSPS Application EAR

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`. The tester package includes various checks, including a check of the ear content against the dependencies in the *jboss-deployment-structure.xml*.

> **Important:** When preparing LSPS Application EAR for production environment, disable the form preview feature from the application: Create a custom application navigator class that extends DefaultAppNavigator and override its addAllViews() method:
>
> ```
> public class AppAppNavigator extends DefaultAppNavigator {
> public AppAppNavigator(UI ui, ViewDisplay display) {
>   super(ui, display);
> }
> @Override
> protected void addAllViews() {
>   addView(todoListViewId(), todoListViewClass());
>   addView(documentsViewId(), documentsViewClass());
>   addView(runModelViewId(), runModelViewClass());
>   addView(appSettingsViewId(), appSettingsViewClass());
>   addView(todoViewId(), todoViewClass());
>   addView(documentViewId(), documentViewClass());
>   //remove this:
>   //addView(formPreviewId(), formPreviewViewClass());
> }
> }
> ```
>
> Make your LspsUI class (typically AppLspsUI), use this navigator: override the createNavigator method:
>
> ```
> @Override
> protected void createNavigator(ViewDisplay display) {
>   Navigator navigator = new AppAppNavigator(getUI(), display);
>   navigator.addViewChangeListener(new PageTitleFromAppView());
> }
> ```

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

4.3 Dependency Management

4.3.1 Adding a Module in the Build

To export a module into a deployable zip file as part of your Maven build use *ModelExporter*: ModelExporter exports the module with all imported modules and their dependencies including the Standard Library Modules.

To integrate the export in your Maven build include the plug-in in your `pom.xml`:

```
<plugin>
 <groupId>org.codehaus.mojo</groupId>
 <artifactId>exec-maven-plugin</artifactId>
 <version>1.2.1</version>
 <executions>
   <execution>
```

```
      <goals>
        <goal>java</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <mainClass>com.whitestein.lsps.export.ModelExporter</mainClass>
    <arguments>
      <argument>d:\CustomerModule</argument>
      <argument>processes\Main</argument>
      <argument>d:\CustomerProject\target\stdlib</argument>
      <argument>d:\result.zip</argument>
    </arguments>
  </configuration>
</plugin>
```

Your can then use the resulting zip file as input for the uploadModel in your `pom.xml`:

```
 <plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <version>1.2.1</version>
  <executions>
    <execution>
      <id>uploadModel</id>
      <goals>
        <goal>java</goal>
      </goals>
      <phase>deploy</phase>
      <configuration>
        <mainClass>com.whitestein.lsps.mconsolecl.Main</mainClass>
        <arguments>
          <argument>modelUpload</argument>
          <argument>-h</argument>
          <argument>${modelUpload.host}</argument>
          <argument>-u</argument>
          <argument>${modelUpload.user}</argument>
          <argument>-p</argument>
          <argument>${modelUpload.password}</argument>
          <!-- The output file of the ModelExporter run:-->
          <argument>-m</argument>
          <argument>d:\result.zip</argument>
          <argument>--dbUpdateStrategy</argument>
          <argument>${modelUpload.dbUpdateStrategy}</argument>
        </arguments>
      </configuration>
    </execution>
  </executions>
</plugin>
```

4.3.2 Adding Dependencies

If you plan to use libraries that are not imported by default, add them to the application `pom.xml` as dependencies, so they are included by maven automatically, and compile the application:

1. Add the library as a dependency to *pom.xml* of the respective project, typically the pom.xml
   - in the <YOUR_APP>-ejb project when extending the LSPS Server with custom functions or task types

- in the <YOUR_APP>-vaadin-war when implementing custom form or ui components
- in the <YOUR_APP>-vaadin when implementing or extending Application User Interface features and components

2. Define the dependency metadata with the dependency version in `<dependencyManagement>` of the main pom.xml.
3. On the command line, go to the application root directory and rebuild the application: `mvn clean eclipse:clean eclipse:eclipse install lsps:updateClasspath -DskipTests`.
4. Refresh the resources in PDS: select the resources in GO-BPMN Explorer, right-click the selection, and click Refresh.

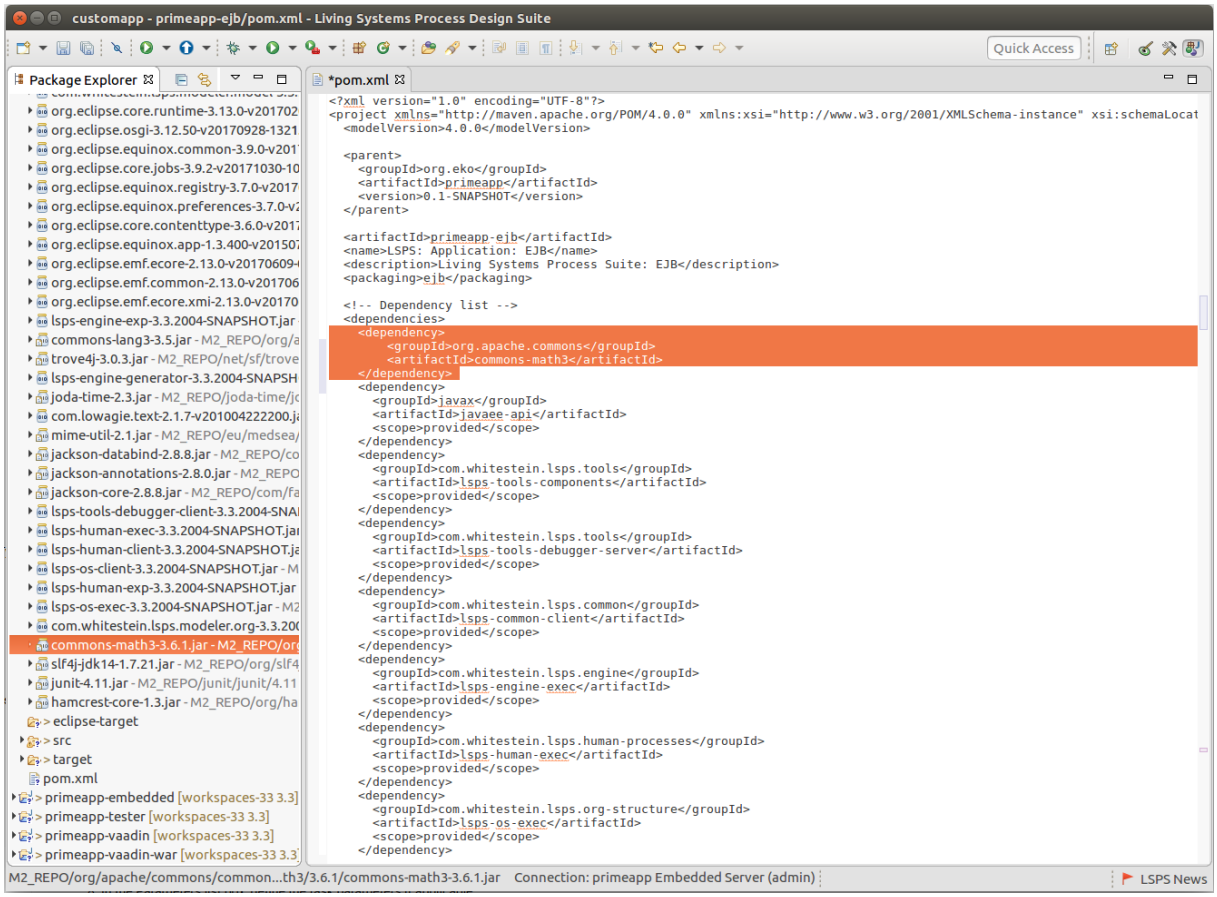


Figure 4.2 New dependency in the application pom.xml

4.3.3 Removing Dependencies

To remove a webapp you do not need, such as Monitoring, from dependencies, do the following:

- for production environments, remove the dependency form the respective pom.xml file, in the case of Monitoring from the pom.xml of the ear project.
- for SDK Embedded Server, remove the dependency form the respective pom.xml file and the application line from the `LSPSLauncher` class in the embedded project.

Chapter 5

Tests

The LSPS Server API provides supports for JUnit testing of your models.

The generated *LSPS Application* contains the testing project with sample JUnit tests with the JUnit API provided by 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 of front-end applications
- **com.whitestein.lsps.test.web.components**: API for testing of 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. Also make sure you use a browser that is compatible with the underlying TestBench version.

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

5.1 Prerequisites

1. Before you create tests of the GUI of the LSPS Process Application, do the following (If you don't require such clicker tests, skip this step):
   - Purchase the Vaadin TestBench license and copy the license to your home directory (on Windows to `c:\Users\<USERNAME>\`).
   - Enable the modeling IDs so you can identify form components on runtime: run the LSPS Server with the `-Dcom.whitestein.lsps.vaadin.ui.debug=true` property.

   To add the property to SDK Embedded Server, in the Java perspective of PDS, go to **Run** > **Run Configurations**; select the configuration under **Java Application** and add the property on the *Arguments* tab to *VM arguments*).
2. If you plan to run the tests 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.

3. Set up Google Chrome as the testing browser:

   (a) Download the ChromeDriver for your OS and copy it to `<YOUR_APP>/tester`.

   (b) In *SampleUIIT* or *Sample*, change the *UIDefaultAppTester* to return ChromeDriver:

```
 //original content with firefox driver:
//protected UIDefaultAppTester app = new UIDefaultAppTester(new FirefoxDriver(), VAADIN_A
// modified content with chrome driver:
protected UIDefaultAppTester app = new UIDefaultAppTester(new ChromeDriver(), VAADIN_APP_
```

5.2 Running JUnit Tests

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

1. Synchronize maven and eclipse (run `mvn eclipse:eclipse`) and add the artifacts to maven repo (run `mvn clean install`).
2. Refresh the GO-BPMN Explorer.
3. Run or connect PDS to your LSPS Server (typically, you will run SDK Embedded Server with your application using the launcher).
4. Run the test:
   - In PDS, right-click the *tester* project or the Java test class, then **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`.

5.3 Creating JUnit Tests

To create a JUnit test, do the following:

1. Open the workspace with the source of the application. In the generated LSPS Application, JUnit testing resources are stored in the `<YOUR_APP>-tester` project:
   - `SampleModelIT.java`: a sample test class that uploads a model, creates its instance and evaluates an expression in the context of the model instance
   - `SampleUIIT.java`: a sample test class that test the UI
   - `test.properties`: relative path to the tested model and to the Standard Library Modules (Standard Library resources are dependencies of the LSPS test classes)
   - `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
```

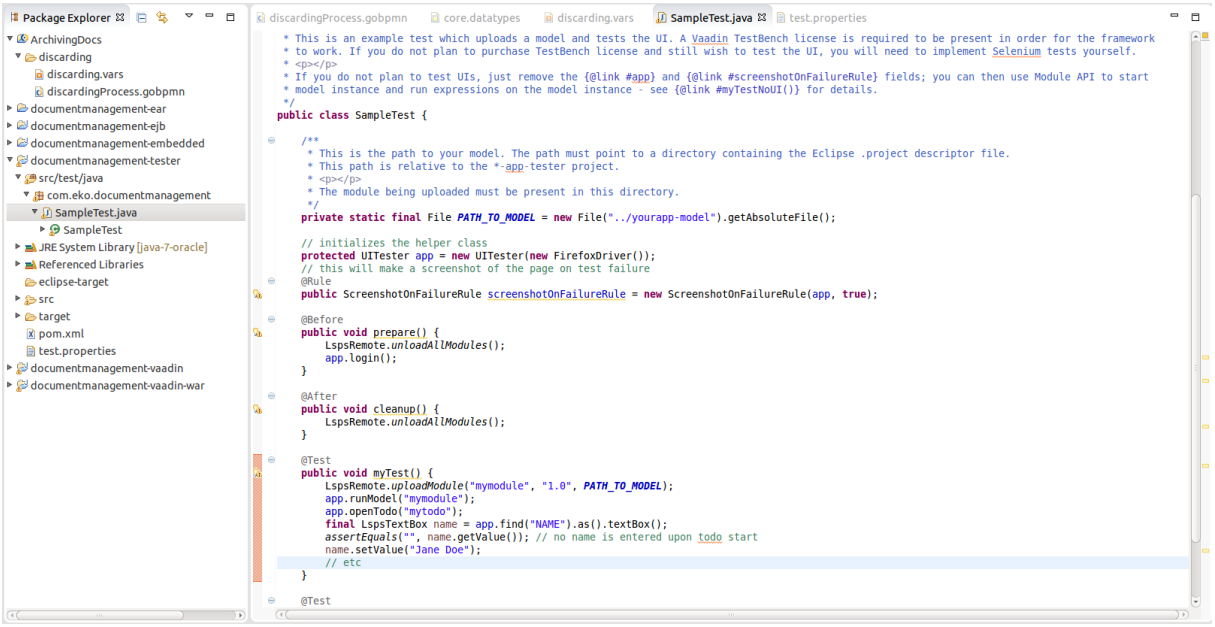


Figure 5.1 SampleTest class in the default LSPS Application

2. Modify or create a new testing class:
   (a) Open the application `tester` project.
   (b) Expand the `src` package and open the `SampleModelTests.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 which are based on the methods of the *com.whitestein.lsps.test* classes:
   For tests that do not require GUI testing, perform `LspsRemote` calls. When testing the GUI, create a UITester instance.
   (c) Edit the sample test file.
   From tests, you can access only the model context: Data from child contexts, such as Sub-Process variables, are not available for testing.
3. Edit the `test.properties` file to point to the location with the project with your model if applicable. Optionally, provide paths to libraries.
4. 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`.
5. Refresh the GO-BPMN Explorer.

5.4 Testing Record Values

When testing record values, consider returning preferably value of simple types from tested expressions.

Instead of:

```
 String patientName = (String) modelInstance.execute(getJohnDoe().name).toObject();
assertEquals("John", patientName);
String patientSurname = (String) modelInstance.execute(getJohnDoe().surname).toObject();
```

move the logic to the expression:

```
 Boolean arePatientDetailsCorrect = (Boolean) modelInstance.execute(
    "def Patient john := getJohnDoe(); john.name==\"John" and john.surname=="Doe"").toObject();
)
assertEquals(true, arePatientDetailsCorrect)
```

Alternatively, you can store the returned object as RecordValue:

```
 RecordValue rec = (RecordValue) modelInstance.execute("getMyRecordById(1)").toObject();
```

Chapter 6

Integration

6.1 Implementing a Custom Person Management

This section contains instructions on how to provide custom implementation of the LSPS person management modules with custom authorization, authentication, and related operations.

> **Important:** If you want to add authentication against another directory service, such as LDAP and Active Directory, add the respective login module settings to the configuration of your application server (refer to the documentation of your application server). Make sure the users from your directory service exist in LSPS Application (for example, create a cronjob that will synchronize the users).

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 your Custom Application User Interface to authenticate and authorize, you need to provide your implementation of the person management services in a custom `pm-<DIRECTORY>-exec.jar` file. You can find the source code of an example LDAP implementation here.

To set your application to use a custom directory service, do the following:

1. In your application, create the `pm-<DIRECTORY>-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. Make sure that whenever you create a person in your directory service, the respective LSPS person is created as well.
5. Rebuild and deploy your application.

6.2 Adding an MXBean

To define an MXBean so its is accessible from JMX monitoring tools, add the interface and the implementation classes to the `<YOUR_APP>-ejb` project.

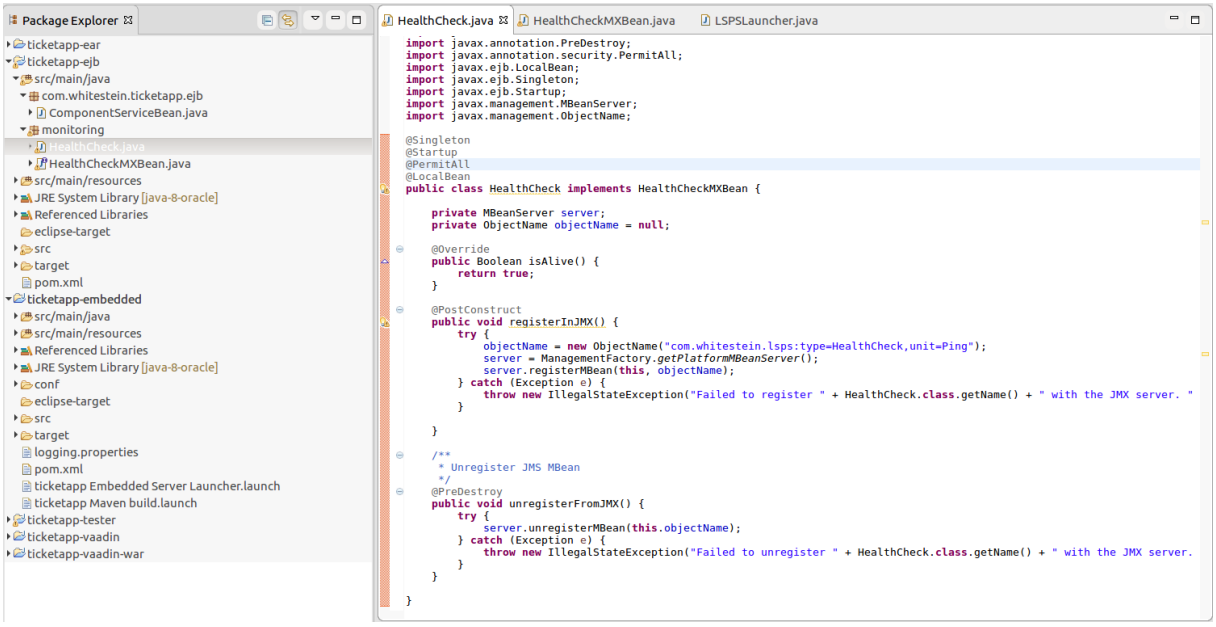


Figure 6.1 MXBean class in the LSPS Application

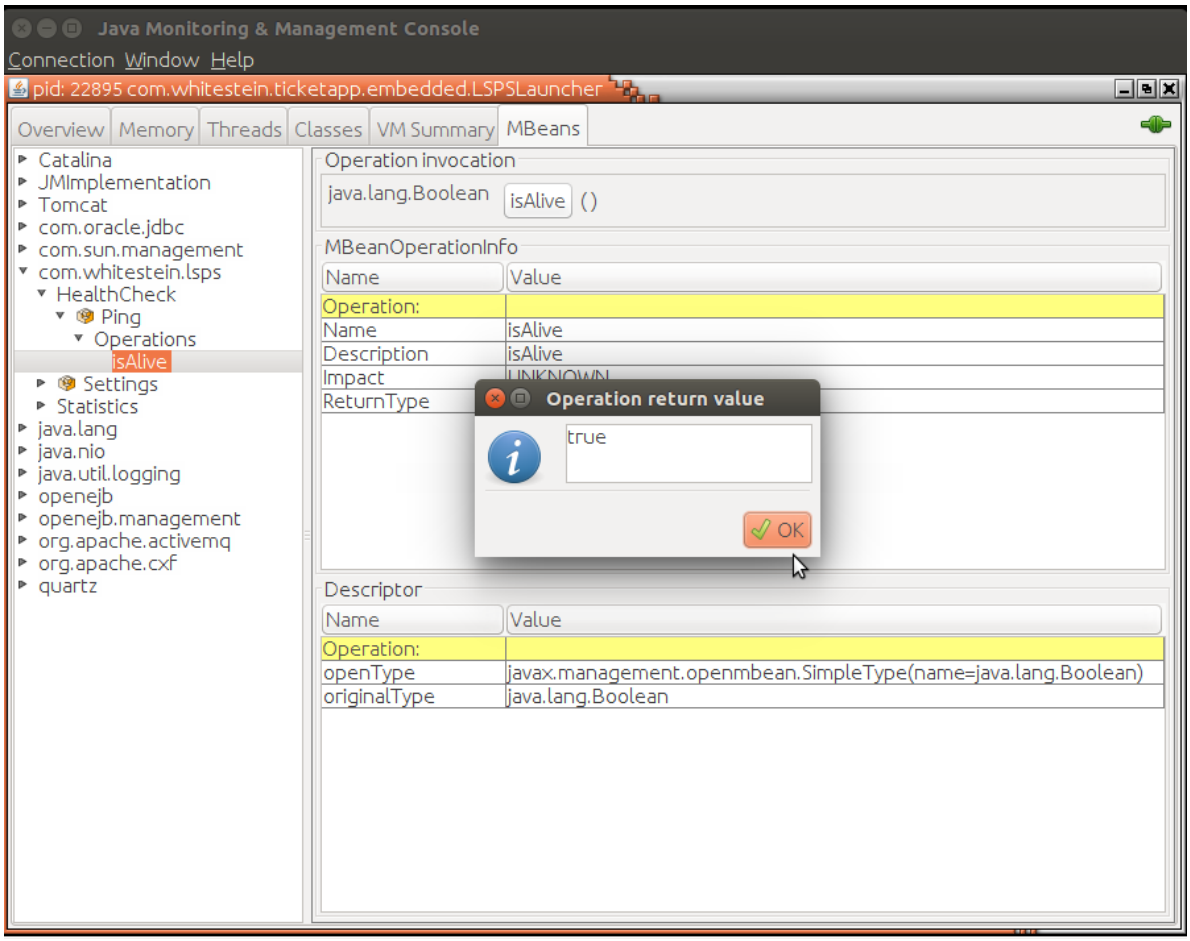


Figure 6.2 MXBean accessed from JConsole

6.3 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.

Make sure you have the following ready:

- You have set up a database or you have access to a database.
- You have the LSPS Application with the related resources open in PDS.
- You have set up the data source on the application server with the LSPS application (refer to the application server documentation).

  For example, to configure a data source so it is accessible from SDK Embedded Server, in `<YOUR_APP>-embedded/conf/conf/openejb.xml`, define its data source configuration (You need to restart SDK Embedded Server for the changes to be applied):

```
  ...
    JdbcUrl jdbc:h2:tcp://localhost/./h2/h2;MVCC=TRUE;LOCK_TIMEOUT=60000
    Username lsps
    Password lsps
#   DefaultTransactionIsolation = READ_COMMITTED
```

```
</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>
```

To work with data from another data source, do the following:

1. Create the entity for the data:

   (a) 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;
  }
}
```

   (b) 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
               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.eh
      <property name="net.sf.ehcache.configurationResourceName" value="META-INF/lsps-ehca
      <!-- JBoss specific parameters -->
      <property name="jboss.as.jpa.providerModule" value="application" />
      <property name="jboss.as.jpa.adapterClass" value="com.whitestein.lsps.common.hibern
    </properties>
  </persistence-unit>
</persistence>
```

   (c) 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://
                  version="2.0">
  <entity class="org.eko.orderusersapp.entity.User" />
</entity-mappings>
```

   (d) 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.JmsCacheMana
 <defaultCache eternal="true" maxElementsInMemory="0" overflowToDisk="false" >
   <cacheEventListenerFactory class="com.whitestein.lsps.common.ehcache.JmsCacheReplicat
 </defaultCache>
 <cache name="org.hibernate.cache.internal.StandardQueryCache" maxBytesLocalHeap="100000
   <cacheEventListenerFactory class="com.whitestein.lsps.common.ehcache.JmsCacheReplicat
 </cache>
 <cache name="org.hibernate.cache.spi.UpdateTimestampsCache" maxElementsInMemory="1000"
   <cacheEventListenerFactory class="com.whitestein.lsps.common.ehcache.JmsCacheReplicat
 </cache>
</ehcache>
```

2. Register the EJB so you can use the entity via an EJB in LSPS modules:

   (a) 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();
    }
}
```

   (b) Register the EJB in the ComponentServiceBean class:

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

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

Chapter 7

Preparing Updates and Upgrades

You can update and upgrade the following:

- Modules: upload new versions of modules
- LSPS Application: deploy a newer version of the LSPS Application
- LSPS:
  - upgrade to a patch version of LSPS
  - upgrade to a minor or major version of LSPS

7.1 Preparing Module Update

When new versions of modules are ready and their previous versions are used in production, you need to handle their running model instances:

- If modules are not restartable, you need to perform `model update` on all running model instances. Mind that the process can be complex and is generally discouraged.
- If modules introduce changes to data models with shared records, you need to update the database tables that hold the data first (you can generate `database schema update scripts for the new model versions from the Management perspective of PDS`, and modify them as necessary).

The artifacts to distribute when updating modules are

- new modules and models exported with GO-BPMN export
- model update definition files
- database-schema update scripts

7.2 Update of the LSPS Application

If you have changed *only* the code of the LSPS Application, it is enough to build the new EAR and deploy it.

If you need to update modules as well, make sure

- to update modules, running model instances, and underlying database tables;
- the application does not introduce any backwards-incompatible changes; For example, if you have removed a custom function from your application and the function is being used by a running model instance, the update of the model instance will fail.

The artifacts necessary for update of LSPS Application with backwards-compatible changes are

- LSPS Application EAR If providing updated of modules as well:
- optionally:
  - new modules and models exported with GO-BPMN export
  - model update definition files
  - database-schema update scripts

7.2.1 Preparing LSPS Upgrade to a New Patch Version

If you want to upgrade to a new patch version of LSPS, To upgrade the entire LSPS stack to a new patch version (x.y.z), you need to do the following:

1. Install the new PDS with SDK.
2. In the root pom.xml, update the LSPS version:
   (a) Update the parent version

```
<parent>
 <groupId>com.whitestein.lsps</groupId>
 <artifactId>lsps</artifactId>
 <version><NEW_VERSION></version>
</parent>
```

   (b) Update the lsps version

```
<properties>
 <lsps.version><NEW_VERSION></lsps.version>
</properties>
```

   (c) In the `openejb.xml` of the >*YOUR_APP*<*-embedded*, remove the *h2* directory and change the path to the database to `JdbcUrl jdbc:h2:tcp://localhost/./h2/h2;MVCC=TRUE;LOCK_TIMEOUT=60000`
3. Build the application.
4. Deploy the EAR to your application server.

Note: If you perform an update to a minor or major version in this way the application might lack new features.

The artifacts to distribute:

- LSPS Application EAR

7.2.2 Preparing LSPS Upgrade to a New Minor or Major Version

To upgrade the entire LSPS stack to a new minor (x.y) of major version (x) of LSPS, do the following:

1. Install and run the new PDS with SDK.
2. Generate a new LSPS Application.
3. Commit the initial state of the application to separates your changes from the initial state of the application.
4. Apply the commits with customizations from your application: when under git, create a patch from the commits that customized the LSPS Application since it was generated and apply it on the newly generated LSPS Application.
5. Build the application.
6. Prepare upgraded modules:
   (a) Import the modules into the workspace.
   (b) For non-restartable models, prepare `model update definitions`;
   (c) If the modules change the data model of persisted data (shared records and their relationships), prepare also the database-schema update scripts.
   (d) `Export the updated models with GO-BPMN Export`.

The artifacts to distribute when upgrading LSPS are

- LSPS Application EAR
- modules and models exported with GO-BPMN export
- model update definition files
- database schema script for business data when applicable