WHITESTEIN
Technologies

Living Systems® Process Suite

Customizing the LSPS Application

Living Systems Process Suite Documentation

3.6
Mon Nov 1 2021

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

Contents

Chapter 1

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.

The resources required for the developement are provided by the LSPS Repository: it provides resources so you can 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.

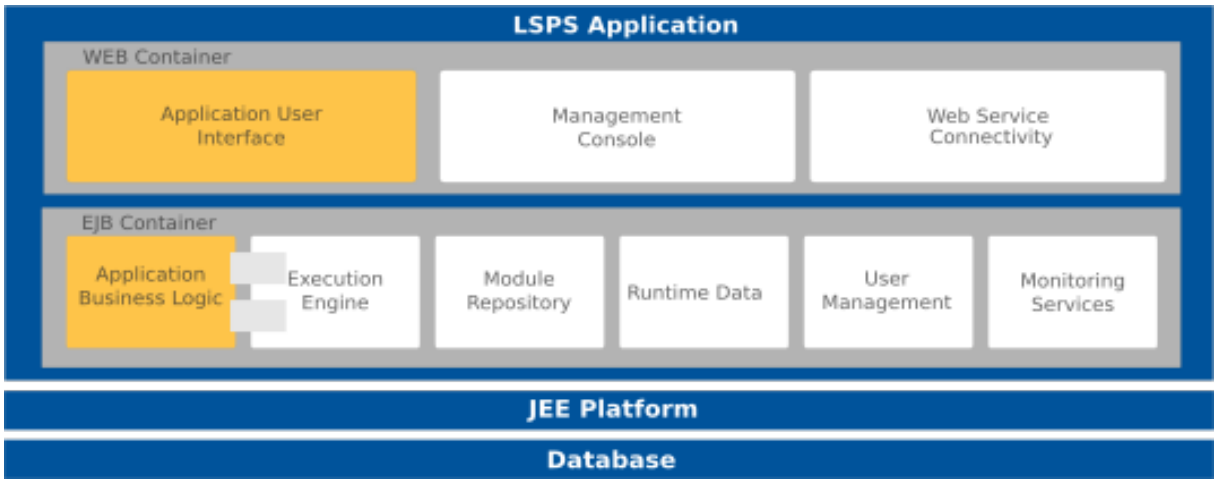


Figure 1.1 LSPS Application architecture with customizable parts highlighted in orange

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, either 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
- `model`: project for your business model
- `model-exporter`: project for `export of models` from the *model* project to deployable zip files
- `tester`: JUnit testing resources
- `vaadin`: web application resources with JavaScript and Java classes
- `vaadin-war`: project for the WAR archive with the login page resources and presentation resources including Vaadin themes.

2.1 Generating the LSPS Application

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

- directly from Designer: this will add launchers for the Maven build and the SDK Embedded Server to the menu toolbar; the application is generated directly in the workspace.
- from the command line: the application sources are generated in the given location. Once generated, you can import the application to a workspace as required.

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

- You have installed Maven.
- You have added Maven to the classpath, for example:

```
export M2_HOME="/Users/lsps/tools/apache-maven-3.6.3"
export PATH=${PATH}:${M2_HOME}/bin
```

- You have installed the LSPS repository into the system repository (~/.m2/repository)
- Your workspace has the *M2_REPO* classpath variable with the path to the LSPS Maven repository defined.

You can check the path to the LSPS Maven repository in `<Designer_HOME>/tools/settings.xml`. To create and set the `M2_REPO` variable, go to **Window** > **Preferences**; then **Java** > **Build Path** > **Classpath Variables**; click **Add** and define the `M2_REPO` variable.

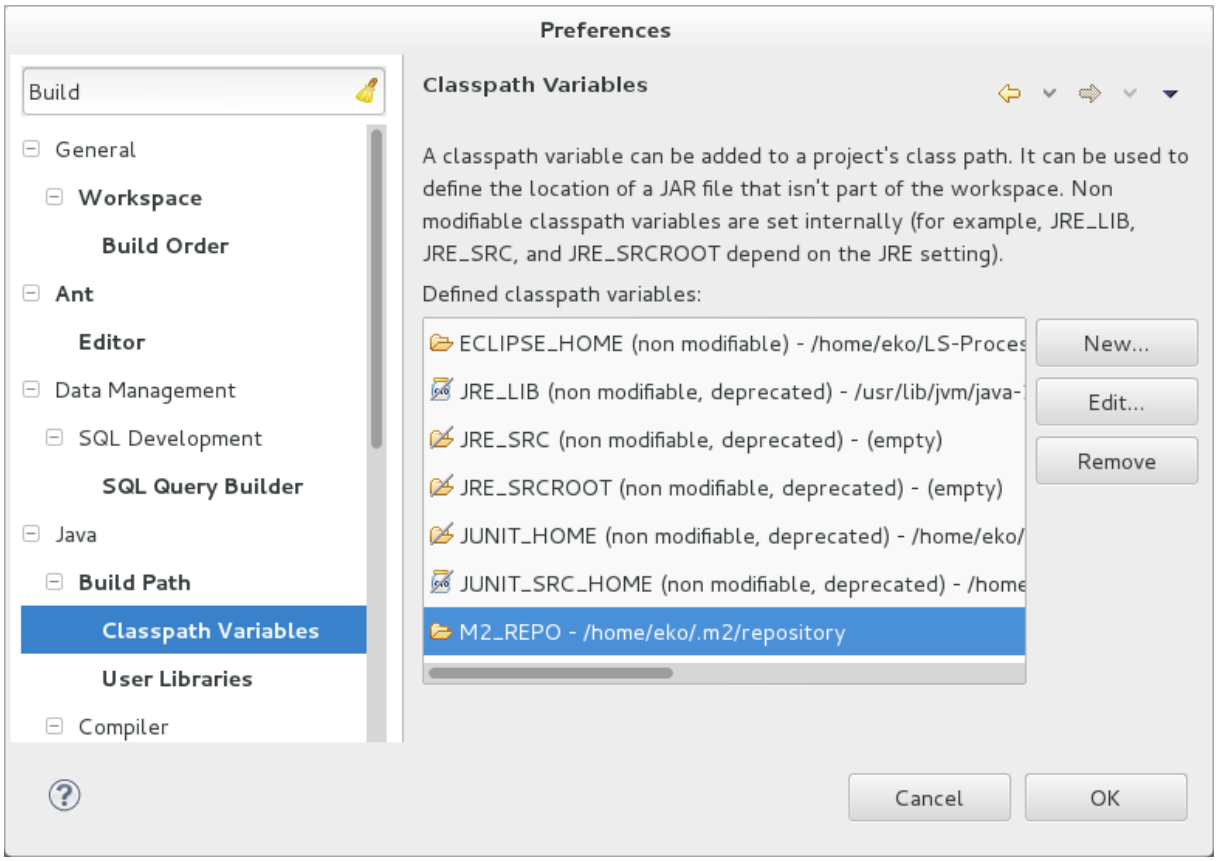


Figure 2.1 M2_REPO variable set up

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

2.1.1 Generating the LSPS Application from Designer

The Designer allows you to generate the LSPS Application–sources of the Application User Interface that expose its API, an SDK Embedded Server, build and launch configurations, and the structure for your business model based on the resources of the LSPS Maven repository: You can modify the sources and check the changes using the SDK Embedded Server,

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

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

1. Extract the lsps-repo zip file to your system repository orto a location pointed to by the M2_REPO variable", for example, `unzip lsps-repo-<VERSION>.zip ~/.m2/`

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:

   - Maven properties: name, version, namespace, base java package name
   - Web properties: context root of the application URL
   - Git properties: where to set up a git repository over the application and create an initial git commit with .gitignore and related sources (git must be installed on your computer)

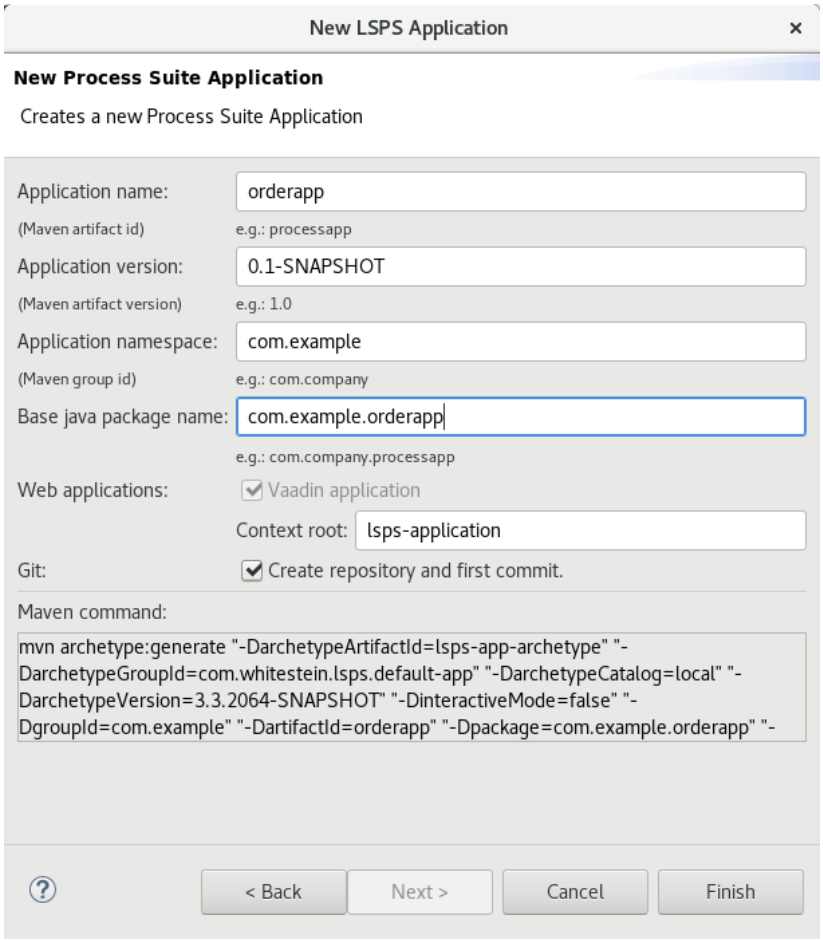


Figure 2.2 Application details

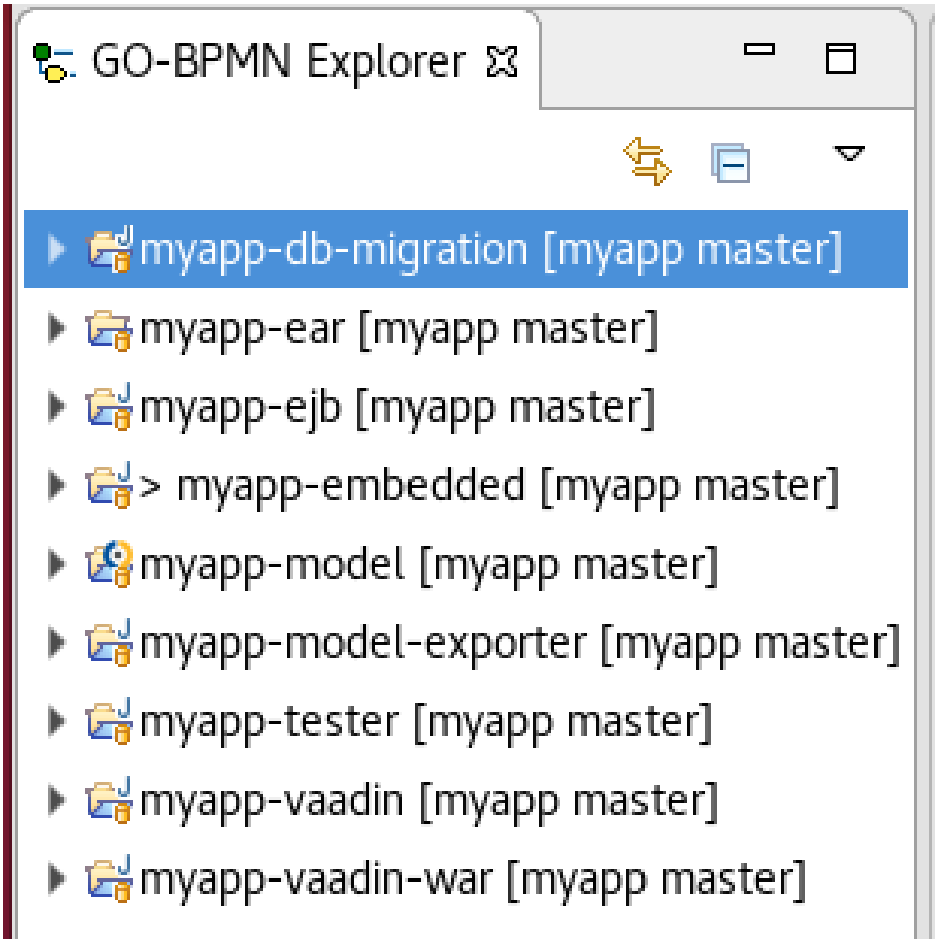


Figure 2.3 Generated application sources

Now you can customize the application and easily re-build it and check out the outcome on SDK Embedded Server.

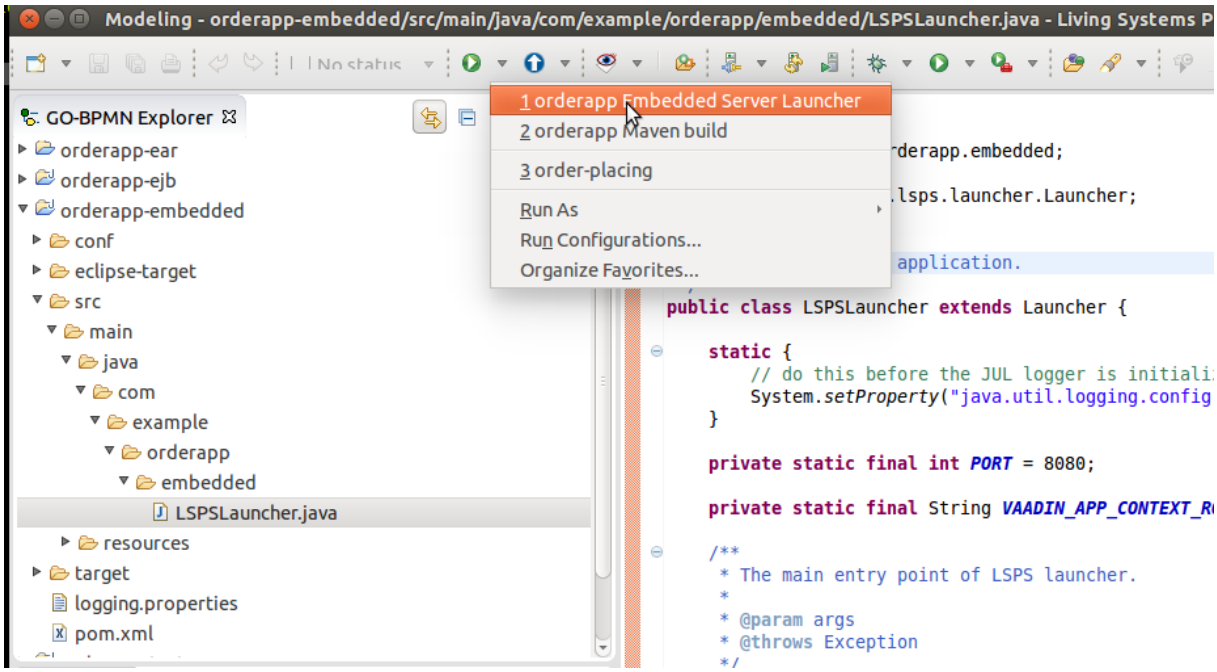


Figure 2.4 Launching SDK Embedded Server with the LSPS Application

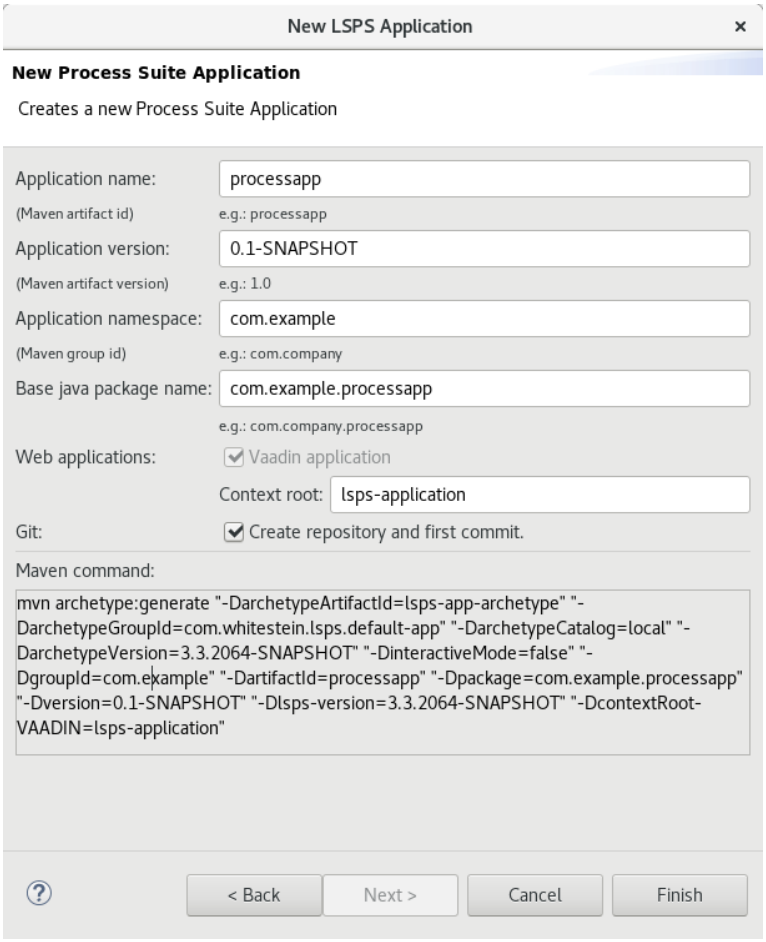
2.1.2 Generating the LSPS Application from the Command Line

1. Install Maven.

2. Create a directory for the application and models.

3. Generate the `lsps-app-archetype` maven artifact.

   To get an example of the maven command, open Designer, 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 either in your local repo or in the LSPS maven repo depending on what maven repository location you defined during Designer installation.



Example command that generates the default LSPS Application

```
$ ~/LSPS-Enterprise-3.3/tools/mvn.sh archetype:generate "-DarchetypeArtifactId=lsps-app-arch
```

4. Put the directory under version control:

   (a) Initialize the repository, for a git repository, enter the directory, and run `git init`.

   (b) Consider adding `.project` to the .gitignore file in the application directory: The directories are generated by maven and differ depending on the environment. Note that is *not* the case for `.project` directories in modules: these are not generated by maven and must be under version control.

   (c) Create an initial commit, for git, `git add *` and `git commit -m "init"`:

   In the future, when migrating to a newer version of the application, you will apply all commits starting from the next commit on the new application version: this will allow you to use a new version of the application with your customization changes added on top (refer to Preparing Upgrade of the Minor or Major Version).

5. Build the application: run `$ ./mvn.sh clean install eclipse:eclipse`.

6. Import the application to Designer workspace.

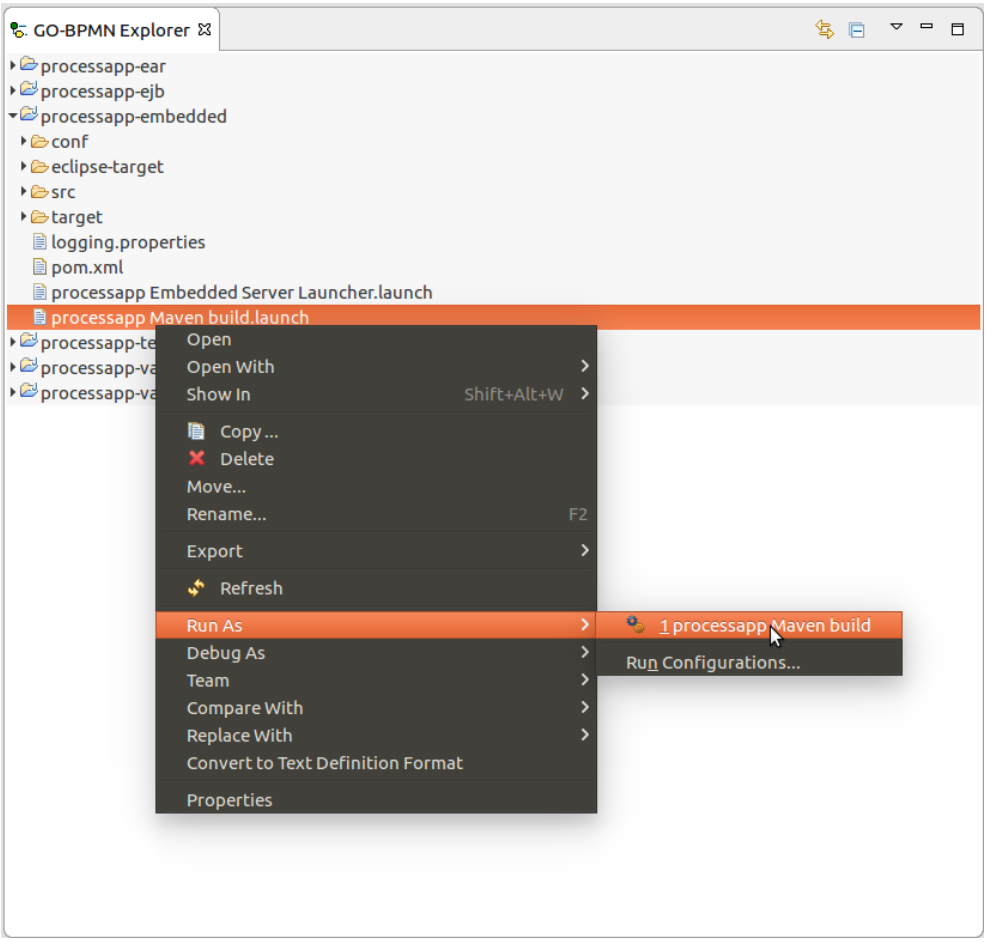
2.2 Importing an LSPS Application to Designer Workspace

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 Designer.
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 Designer, not from the command line, you can build the application with 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.

2.3 Running the Application with the SDK Embedded Server

When you generate the LSPS Application, apart from the application, an embedded server, *SDK Embedded Server*, with the application deployed and a launcher is generated in the *<APP>-embedded* project. The server uses an embedded h2 database.

When you run the server using the launcher class, the server starts and if there is no h2 database, a new h2 database is created in the project.

The server is intended for testing purposes only. For information on how to set up production servers, refer to the deployment documentation

2.3.1 Configuring the Mail Server of the 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:

```
<Resource id="mail/LSPS_MAIL"  type="javax.mail.Session">
  mail.transport.protocol=smtp
  mail.smtp.host=<YOUR_SMTP>
  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`

2.3.2 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 mssql data source

   ```
   ...
     <Resource id="jdbc/LSPS_DS" type="javax.sql.DataSource">
       JdbcDriver com.microsoft.sqlserver.jdbc.SQLServerDriver
       JdbcUrl jdbc:sqlserver://localhost:1433;DatabaseName=lsps
       Username lsps
       Password lsps
     </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.

2.3.2.1 Deleting the Embedded H2 Database of SDK Embedded Server

To delete the embedded 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

While you can create a new theme from scratch, it is recommended to extend an existing theme. To create a custom theme for your application based on an existing theme, 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 a directory with a theme (`lsps-valo`, `lsps-dark` or `lsps-blue`).
   (c) Rename the directory to the theme name.
   (d) In `<YOUR_THEME>/styles.scss`, change the theme class name to your theme name: Make sure the directory name and the theme name are identical.

```
/*
changed .lsps-valo to .my-theme
*/
.my-theme {
  @include addons;
  @include lsps-valo-base;
  @include theme-app;
}
```

2. In `<YOUR_APP>-vaadin/src/main/java/org/eko/processapp/core/AppUIProvider.java`, override the `getThemeManager()` method so it uses the `ThemeManager` constructor with all the themes.

```
@Override
  protected ThemeManager getThemeManager() {
    ImmutableSet<String> themes =
              ImmutableSet.of("lsps-dark", "lsps-valo", "lsps-blue", "my-theme");
    //my-theme passed as default:
    return new ThemeManager("my-theme", themes);
  }
```

   The default themes have a `style.scss` file that imports the relevant scss files. It is the included scss files you need to change. If possible, introduce your changes to the `sass/_variables.scss` file before you create or modifying other scss files.

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

3. Consider setting up the Sass compiler so you can preview changes to your theme without rebuilding the entire application: Once set up, you simply introduce your change to the scss files and run the saas compiler with its run configuration from Designer.

   Important: Make sure that the name of the directory with your theme and the style class defined in its *style.scss* file have the same value.

3.1.1.2 Modifying a Theme

It is recommended to modify your theme primarily using the exposed variables in the `<YOUR_THEME>/sass/_variables.scss` file.

Example _variables.scss

```
$v-background-color: hsl(100, 100%, 100%);
$valo-menu-background-color: #FFFFFF;
/* logo path is relative to theme root */
$l-application-logo: url('img/logo.png');
```

If you need to add an scss rule, 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;
}
--------->8
```

4. Recompile the theme with the SassCompiler.
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.

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`

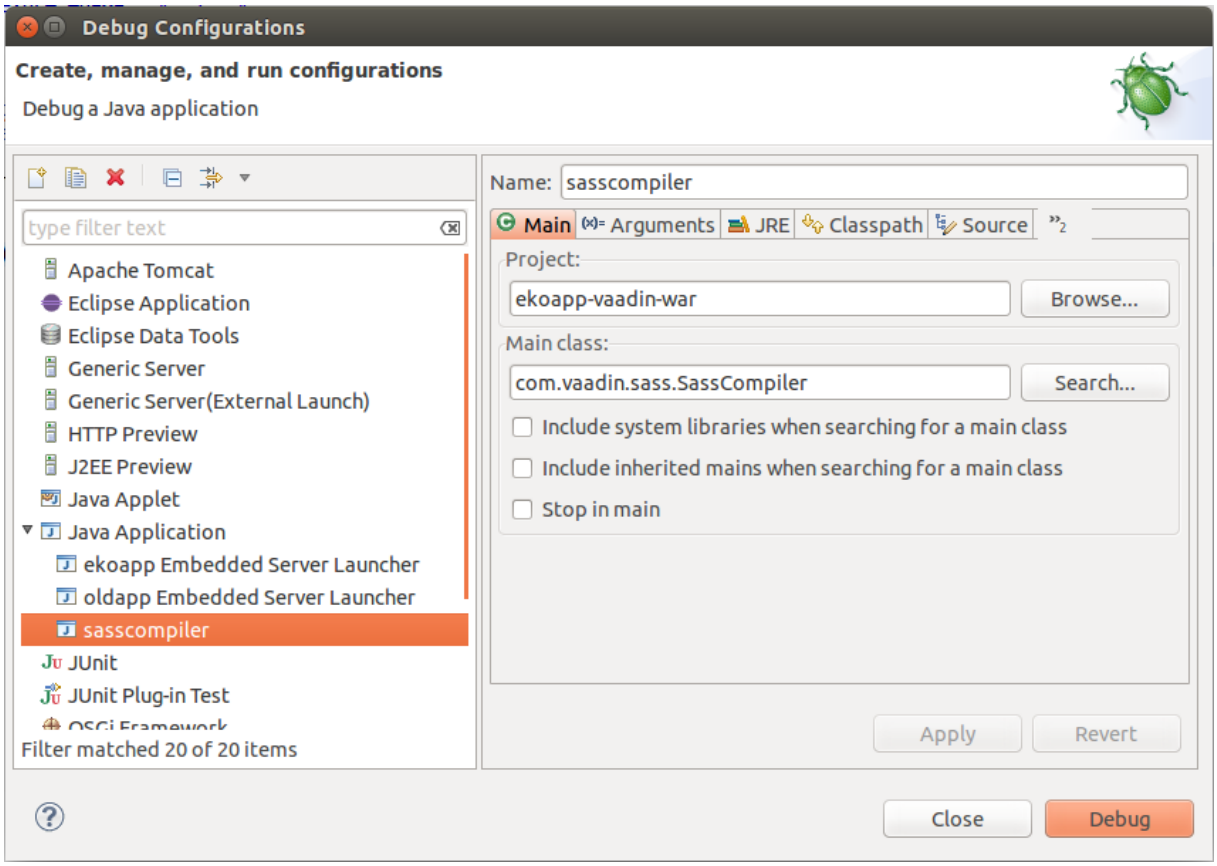


Figure 3.1 Run configuration for Sass compiler

(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. Test the configuration:

   (a) Create an scss file and import it in the theme's styles.scss.

   (b) Introduce your changes to the scss file and see them applied:

   (c) Run the Saas compiler configuration.

   (d) 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 AppUIProvider class in the *<YOUR_APP>.core* package of the `<YOUR_APP>-vaadin` project and override the `getThemeManager()` method so it returns a ThemeManager with the theme.

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

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: check the LspsUIProvider for the relevant method; 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 Switching between Vaadin 7 and Vaadin 8 UI Implementations

In your application, you can use either the Vaadin 7 or Vaadin 8 implementation of `UI form components`. By default, the Vaadin 8 implementation is used.

To change the UI implementation, open the `DefaultLspsAppConnector` class or your custom class if you extended it, and adjust the `getComponentFactory()` method so it returns the required UI factory implementation:

- `UIComponentFactoryV7Impl` to retain the Vaadin 7 implementation

- `UIComponentFactoryV8Impl` to use Vaadin 8 implementation

The `implementations are not fully backwards compatible` and switching between them requires thorough testing.

> **Important:** Currently not all components are implemented in Vaadin 8 and even if you are using Vaadin 8 UI factory, Vaadin 7 implementations will be used if no Vaadin 8 implementation is available.

```
@ Override
public UIComponentFactory getComponentFactory() {
return new UIComponentFactoryV7Impl(this);
}
```

To enable the Vaadin 8 implementation on the fly, add the following cookie from the console of your browser `document.cookie="switch_ui_to_vaadin_v8=true"` and reload.

3.1.3 Customizing Content

With the API of the *Application User Interface* web application, you can fully customize the content of the application; you add new components as well as modify or remove the existing ones.

The API allows you to extend applications classes and override their methods so you can keep what is already provided or implement the classes anew.

It is exposed in the <YOUR_APP>.core and <YOUR_APP>.connectors package of the <YOUR_APP>-vaadin project:

- The main class of the Application User Interface is the `AppLspsUI` class in the core package. It represents the root component of the application.
- The components that remain the same throughout the application in your browser are governed by the *AppLayout* class: The class holds the menu to the Views of Documents, To-Dos, and Models, etc. By extending the class, you can implement your own custom menu or add custom layout parts that are always visible, such as, headers.
- You can create custom views if required.

3.1.3.1 Customizing AppLspsUI

The *AppLspsUI* class holds the root component of the gui and provides functions required by the entire application, including ThemeManager and Navigator. Its *init()* method creates the content root layout and the connector between LSPS forms and your application, called the AppConnector, and calls the *initLayout()* method which assembles the screen content.

Generally, it is not recommended to reimplement the AppLspsUi from scratch since the class is responsible for connection to the LSPS Server, authentication and other basic functionalities. You will rather override its methods so it uses your customized resources, such as a custom navigator.

If you still decide to implement LspsUI anew, make sure it meets the following requirements:

- It returns an instance of LspsUIBase.
- It declares the LSPS widget set for the Vaadin servlet.

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

- It provides an implementation of LspsAppConnector, interface that defines the binding contract between the LSPS vaadin renderer and the rest of the application.
- It provides an implementation of LspsFormConnector, interface for a connector of forms and views.
- I t uses JAAS for user authentication: The setting is available in the `login.jsp` and in the `WEB-INF/web.xml` of the webapp project.

3.1.3.2 Customizing AppAppLayout

AppAppLayout is responsible for the menu and other components that are always visible.

Note: If you are reimplementing the AppLayout, your implementation must

- implement *com.whitestein.lsps.human.app.ui.AppLayout* and
- extend a vaadin Component.

You can extend this class to add a new component to it, such as a footer or header: modify the AppAppLayout class:

1. In your AppLayout class, by default the AppAppLayout, override the attach() method. Call the attach method of the parent AppLayout and then use the `com.vaadin.ui.CssLayout.addComponent()` method to add your component.

```
 private Component header;
 @Override public void attach() {
~
  super.attach();
  if (header == null) {
    header = new AppHeader();
    addComponent(header, 0);
  }
}
```

2. In the <YOUR_APP>.core package of the vaadin project, create a class with the Vaadin component. Consider extending the *CustomComponent*.

3. Implement the component: consider adding a style class to the component with the `addStyleName()` call so you can add the style to scss.

   Example header implementation

```
 public class AppHeader extends CustomComponent {
~
  public AppHeader() {
~
    //csslayout renders as a div:
    CssLayout hl = new CssLayout();
    setCompositionRoot(hl);
~
    hl.addComponent(new Label("My Company"));
    hl.addComponent(new Label("My Company Address"));
    hl.addComponent(getLogo());
    //add style to header:
    addStyleName("my-custom-header");
  }
~
  protected Image getLogo() {
    String basepath = VaadinService.getCurrent()
        .getBaseDirectory().getAbsolutePath();
    //Image as a file resource
    FileResource resource = new FileResource(new File(basepath +
        "/WEB-INF/images/lsps_logo.png"));
    //Show the image in the application
    Image image = new Image("Image from file", resource);
    return image;
~
  }
}
```

3.1.3.3 Adding a Custom View

A View is used as the content of the AppLayout, which is different for different URLs: this is the mechanism used to display To-Dos, Documents, and Run Model content. To create a custom view, you need to create a class with the GUI and register the URL of the view with the navigator.

> **Important:** Before you create a custom view, consider using Documents: with documents and only adding a link to the document to the navigation menu using the addDocumentsMenuItem() call. Like this, your page is delivered with your model, not as part of the application, and created with a graphical form editor.

To create a new view and register its URL with the navigator, do the following:

1. In the <YOU_APP>.core package, create your View class that extends *DefaultAppView*. If this is inconvenient or impossible, extend View or Component. Implement the view constructor.

```
public class MyAppView extends DefaultAppView {
~
    public MyAppView() {
      AppInjector.injector.inject(this);
~
      layout = new VerticalLayout();
      layout.setSpacing(false);
      layout.setMargin(false);
      layout.setSizeFull();
      layout.addComponent(new Label("Hello"));
      setCompositionRoot(layout);
      setSizeFull();
~
      title = new Label("My View");
    }
  }
}
```

2. Create and integrate your custom Navigator class:
   (a) Make the class extend the *DefaultAppNavigator* class and name it *AppAppNavigator*.
   (b) In the *AppLspsUI* class, modify the *createNavigator()* method to use your navigator class.

```
    @Override protected void createNavigator(ViewDisplay display) {
      //create navigator with your custom navigator:
      Navigator navigator = new AppAppNavigator(getUI(), display);
      navigator.addViewChangeListener(new PageTitleFromAppView());
~
    }
```

   (c) Override the *getNavigator()* method to return your navigator as well.

```
    @Override public AppAppNavigator getNavigator() {
       return (AppAppNavigator) super.getNavigator();
     }
```

3. Make the navigator resolve URLs with the ID of your view to the class of with your View:
   (a) Define the id of your view (it will be used in the URL to indicate that we want to navigate to the view)

```
public String myViewId() {
  return "myview";
}
```

   (b) Make it return the class of your view, when it is requested.

```
protected Class<? extends AppView> myAppViewClass() {
  return MyAppView.class;
}
```

(c) Override the *addAllViews()* method and add the view with the ID and class to the Navigator.

```
@Override protected void addAllViews() {
  super.addAllViews();
  addView(myViewId(), myAppViewClass());
}
```

4. Consider adding a component that will link to your view, for example an item to the main menu.

This is also the way how the create a custom Welcome page, which is displayed when you click the logo with the difference that you need to override the *openHomePage* method:

```
@Override public void openHomePage() {
  navigateTo(getNavigator().myViewId());
  // navigateTo(getNavigator().todoListViewId());
}
```

3.1.3.4 Customizing the Navigation Menu

To add or remove items in the navigation menu, do the following:

1. Make your AppLayout use a custom MainMenu:

   (a) Open your AppLayout implementation (the **AppAppLayout** by default) in the core package of the vaadin project.

   (b) Adapt the createMainMenu() method so it returns your MainMenu implementation.

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

2. In the <YOU_APP>.core package of the <YOUR_APP>-vaadin project, create the implementation of the custom MainMenu class that extends the *DefaultMainMenu* class, in the example, `AppMainMenu`.

3. To allow closing of the menu, when the user taps out of the menu on their mobile device, create the constructor with the `closeOnTapAreaId` parameter that calls the DefaultMainMenu constructor.

```
public AppMainMenu(String closeOnTapAreaId) {
  super(closeOnTapAreaId);
}
```

4. Copy the implementation of the *DefaultMainMenu.getNavigator()* method:

```
private AppNavigator getNavigator() {
  LspsUI ui = (LspsUI) UI.getCurrent();
  return ui.getNavigator();
}
```

5. Copy the implementation of the *DefaultMainMenu.createMenu()* method to the custom *MainMenu* implementation (*AppMainMenu*) and adapt it:

   - To *remove items from the menu*, comment out the `if` statement with the respective navigation menu item.
   - To *add your items to the menu*, do the following:

     (a) Create a private method that adds the Menu Item:

```
private void addMyViewMenuItem(NavigationMenu navigationMenu) {
   navigationMenu.addViewMenuItem("nav.myview", getNavigator().myViewId(), VaadinIcon
 }
 //you need to add a getter for a Navigator:
 //here we use a custom AppNavigator to get the View id:
 private AppAppNavigator getNavigator() {
   LspsUI ui = (LspsUI) UI.getCurrent();
   return (AppAppNavigator) ui.getNavigator();
 }
```

(b) Add the custom menu item to the *createMenu()* method.

```
addMyViewMenuItem(navigationMenu);
```

When adding 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)

(c) Add the localization property to the *localization.properties* file in the <YOUR_APP>-vaadin project.

```
nav.myview = My View
```

Example custom MainMenu implementation

```
public class AppMainMenu extends DefaultMainMenu {
~
  public AppMainMenu(String closeOnTapAreaId) {
    super(closeOnTapAreaId);
  }
~
  @Override
  @SuppressWarnings("unused")
  protected NavigationMenu createMenu() {
    NavigationMenu navigationMenu = new NavigationMenu(new UserMenu());
~
    LspsUI ui = (LspsUI) UI.getCurrent();
    UserInfo user = ui.getUser();
~
    //removing To-Do List:
    /*if (user.hasRight(HumanRights.READ_ALL_TODO) || user.hasRight(HumanRights.READ_OWN_TODO)) {
      addTodoListMenuItem(navigationMenu);
    }*/
    if (user.hasRight(HumanRights.ACCESS_DOCUMENTS)) {
      addDocumentsMenuItem(navigationMenu);
    }
    if (user.hasRight(EngineRights.READ_MODEL) && user.hasRight(EngineRights.CREATE_MODEL_INSTANCE
      addRunModelMenuItem(navigationMenu);
    }
    //adding custom navigation item:
    if (user.hasRight(EngineRights.READ_MODEL) && user.hasRight(EngineRights.CREATE_MODEL_INSTANCE
      addMyMenuItem(navigationMenu);
    }
    if (hasRightToOpenDocument("custom_todo_list_ui::todoItemsList")) {
      navigationMenu.addDocumentItem("nav.mytodoitems", "custom_todo_list_ui::todoItemsList", null
    }
    //link to custom view:
    addMyViewMenuItem(navigationMenu);
    return navigationMenu;
  }
~
  private void addMyMenuItem(NavigationMenu navigationMenu) {
    navigationMenu.addViewMenuItem("nav.myview", getNavigator().myViewId(), VaadinIcons.LIST, null
  }
~
```

```
  private AppNavigator getNavigator() {
    LspsUI ui = (LspsUI) UI.getCurrent();
    return ui.getNavigator();
  }
}
```

3.1.3.5 Setting the Home Page

To set the page which is loaded after the user has logged in, modify the `openHomePage()` method of your *LspsUI* class, by default the *AppLspsUI* 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);
}
```

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 {
      background-color: white;
      background-image: url("../img/logo_splash.png");
      background-position: center bottom;
      background-repeat: no-repeat;
      background-size: 300px auto;
   }
   &.login-page #loginHeader {
      background: none !important;
   }
```

```
    &.login-page #username {
       background: yellow !important;
    }
    &.login-page #password {
       background: blue !important;
    }
    &.login-page .loginform table {
        background: none;
    }
    &.login-page .login-button {
        background: red;
        margin-right: 47px;
        color: grey;
    }
}
```

3.1.3.6 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.

3.1.3.7 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.

3.1.3.8 Customizing the Welcome Page

To customize the Welcome Page of your application, create a new custom view and set the view as your application's home page.

3.1.3.9 Adding a Locale

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

1. Create a properties file with the name localization_<LANGUAGE_CODE>.properties with the translations in <YOUR_APP>-vaadin/src/main/resources/com/whitestein/lsps/vaadin/webapp/‘. The language code is based on the java.util.Locale class.

   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 language option to the language picker on the Settings screen:

   (a) In the <YOUR_APP>.core package or the vaadin project, create the *AppAppSettingsView* class that extends the *AppSettingsView* class and add the locale to the *allLanguages()* method.

```
@Override
protected Collection<StringOption> allLanguages() {
  ArrayList<StringOption> langs = new ArrayList<>(super.allLanguages());
  langs.add(new StringOption("it_IT", "Italiano"));
  return langs;
}
```

(b) Extend the Navigator so the user is navigated to your SettingView: create a Navigator that returns the implementation of the Settings view you have just created: override the appSettingsViewClass so it returns your AppAppSettingsView.

```
public class AppAppNavigator extends DefaultAppNavigator {
~
  public AppAppNavigator(UI ui, ViewDisplay display) {
    super(ui, display);
  }
~
  @Override
  protected Class<? extends AppView> appSettingsViewClass() {
    return AppAppSettingsView.class;
  }
}
```

(c) Extend your AppLspsUI to use your Navigator.

```
@Override
protected void createNavigator(ViewDisplay display) {
  Navigator navigator = new AppAppNavigator(getUI(), display);
  navigator.addViewChangeListener(new PageTitleFromAppView());
}
```

3.1.3.10 Customizing the Settings View

To customize the Application Setting section of the Settings view, you need to override `createOther↩SettingsSidebar` method of the *AppSettingsView* class.

If you need to to remove or change the User Settings part or the Substitution Setting, implement a custom Setting view.

To customize the Settings view, do the following:

1. In the vaadin project in the *core* package, make *AppLspsUI* to use a custom vaadin Navigator:

   (a) Override the *createNavigator()* method of the *AppLspsUI* class.

   ```
   @Override
   protected void createNavigator(ViewDisplay display) {
     Navigator navigator = new AppAppNavigator(getUI(), display);
   }
   ```

   (b) Create the *AppAppNavigator* class and override the *appSettingsViewClass()* method so it returns your *AppAppSettingsView* class.

   ```
   public class AppAppNavigator extends DefaultAppNavigator {
   ~
     public AppAppNavigator(UI ui, ViewDisplay display) {
       super(ui, display);
     }
   ~
     @Override
     protected Class<? extends AppView> appSettingsViewClass() {
       //not yet existing class:
       return AppAppSettingsView.class;
     }
   }
   ```

2. Create the settings view class that extends *AppSettingsView* and override the *createOtherSettingsSidebar()* method: make it return the relevant setting components. Remove the respective *addSettingComponent()* calls and add new ones as required.

```
@Override
  protected SettingComponent createOtherSettingsSidebar() {
    VerticalSettingsComponent result = new VerticalSettingsComponent();
    result.addSettingComponent(createLanguageSettings());
    //removing theme settings:
    //result.addSettingComponent(createThemeSettings());
    result.addSettingComponent(createLayoutsSettings());
    //adding custom setting:
    result.addSettingComponent(createSoundSettingComponent());
~
    return result;
  }
  private SettingComponent createSoundSettingComponent() {
    //not yet existing class:
     return new SoundSettingsComponent();
  }
```

3. Implement the classes of any new settings components you added. Make the class extend the *Custom↩ Component* class and implement *SettingComponent*:

   (a) Create the setting content as a Vaadin component.

   (b) Make the component the root of the of the SettingComponent with the `setCompositionRoot()` call.

```
public class SoundSettingsComponent extends CustomComponent implements SettingComponent {
~
  private final LspsUI ui;
~
  private Boolean soundSetting;
  private CheckBox soundSettingCheck;
  private Cookie soundCookie;
  private String NAME_COOKIE = "l-sound-cookie";
~
  public SoundSettingsComponent() {
    ui = LspsUI.getCurrent();
~
    setCaption("Sound");
~
    soundSetting = getCurrentSoundSetting();
    soundSettingCheck = new CheckBox();
~
    ...
    setCompositionRoot(soundSettingCheck);
  }
```

4. If required, implement the `validate()` and `save()` method:

   - *validate* is called when the user clicks the **Save** button. If it returns `false` the settings are not saved.
   - The *save()* method is called after the `validate()` call returns `true`.

```
@Override
public boolean validate() {
  if (getCurrentSoundSetting()) {
    //adds a validation message to the component:
    soundSettingCheck.setComponentError(new UserError("This setting cannot be true."));
    return false;
  } else {
    return true;
  }
}
```

```
~
  @Override
  public void save() {
    Cookie soundCookie = new Cookie(NAME_COOKIE, soundSetting.toString());
    VaadinService.getCurrentResponse().addCookie(soundCookie);
  }
```

3.1.3.11 Importing JavaScript

Important: We strongly recommend to use JavaScript sparingly and exclusively **with the aim to change the presentation layer** of your application. It is not commended to manipulate your business data with JavaScript.

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 implementation of your custom application, by default the `AppLspsUI`class, as follows:

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

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

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

Expression Language Type	Class
Set	com.whitestein.lsps.lang.exec.SetHolder
Type	com.whitestein.lsps.lang.type.Type
Map	com.whitestein.lsps.lang.exec.MapHolder
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

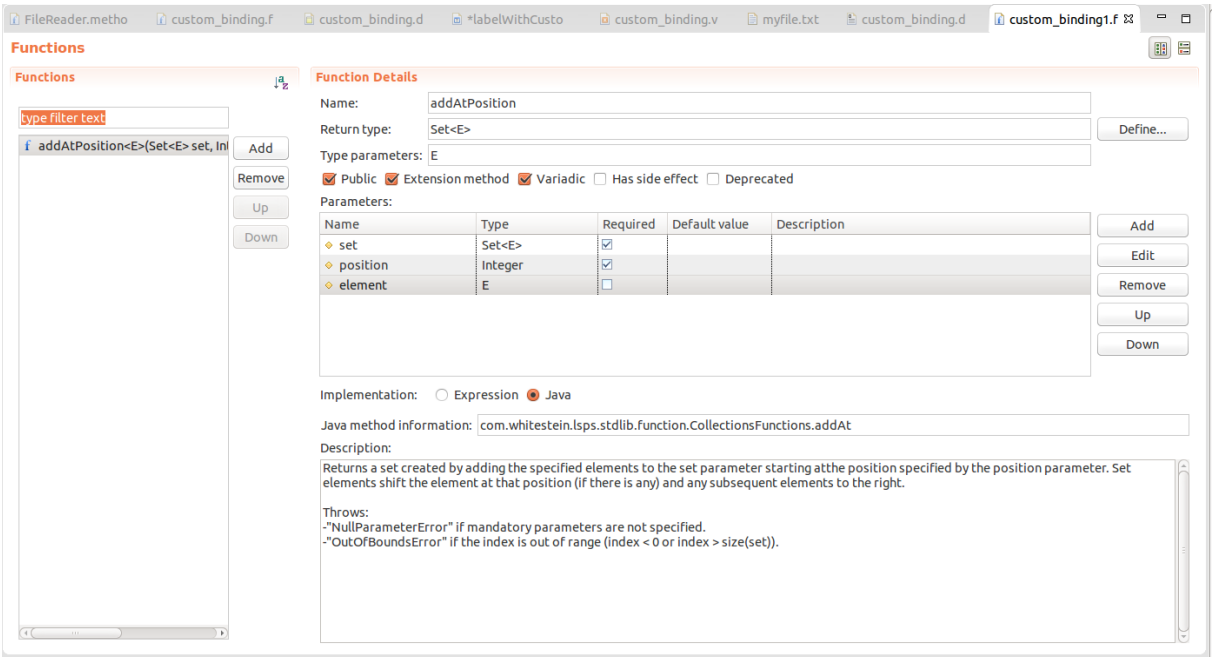


Figure 3.2 Function Editor with a function definition

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

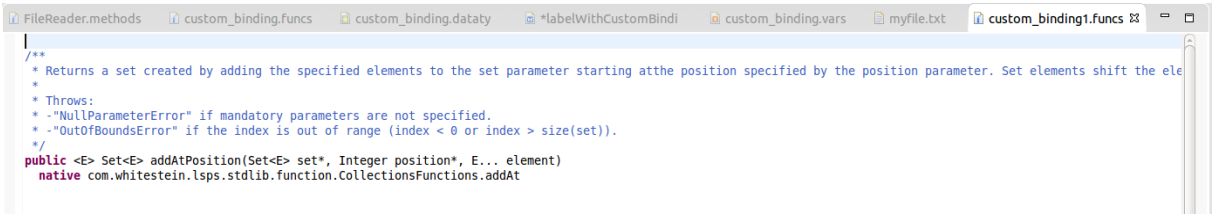


Figure 3.3 Text function editor with a function definition

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.

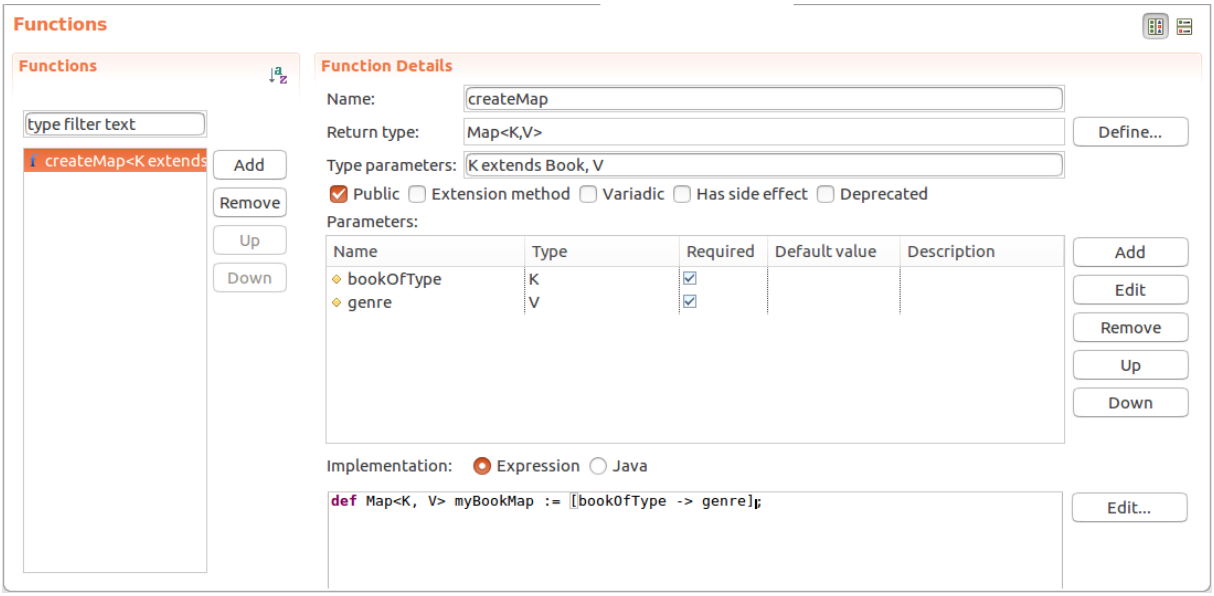


Figure 3.4 Example function with type parameters

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.

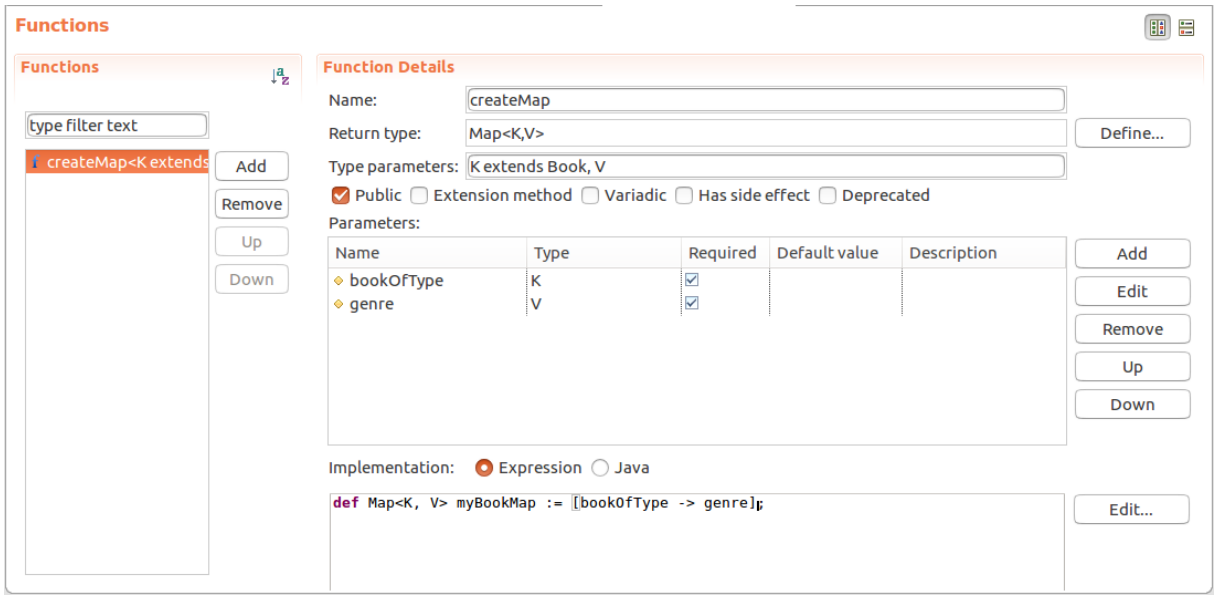


Figure 3.5 Function declaration and definition in the Expression Language

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 Designer. 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));
  }
}
```

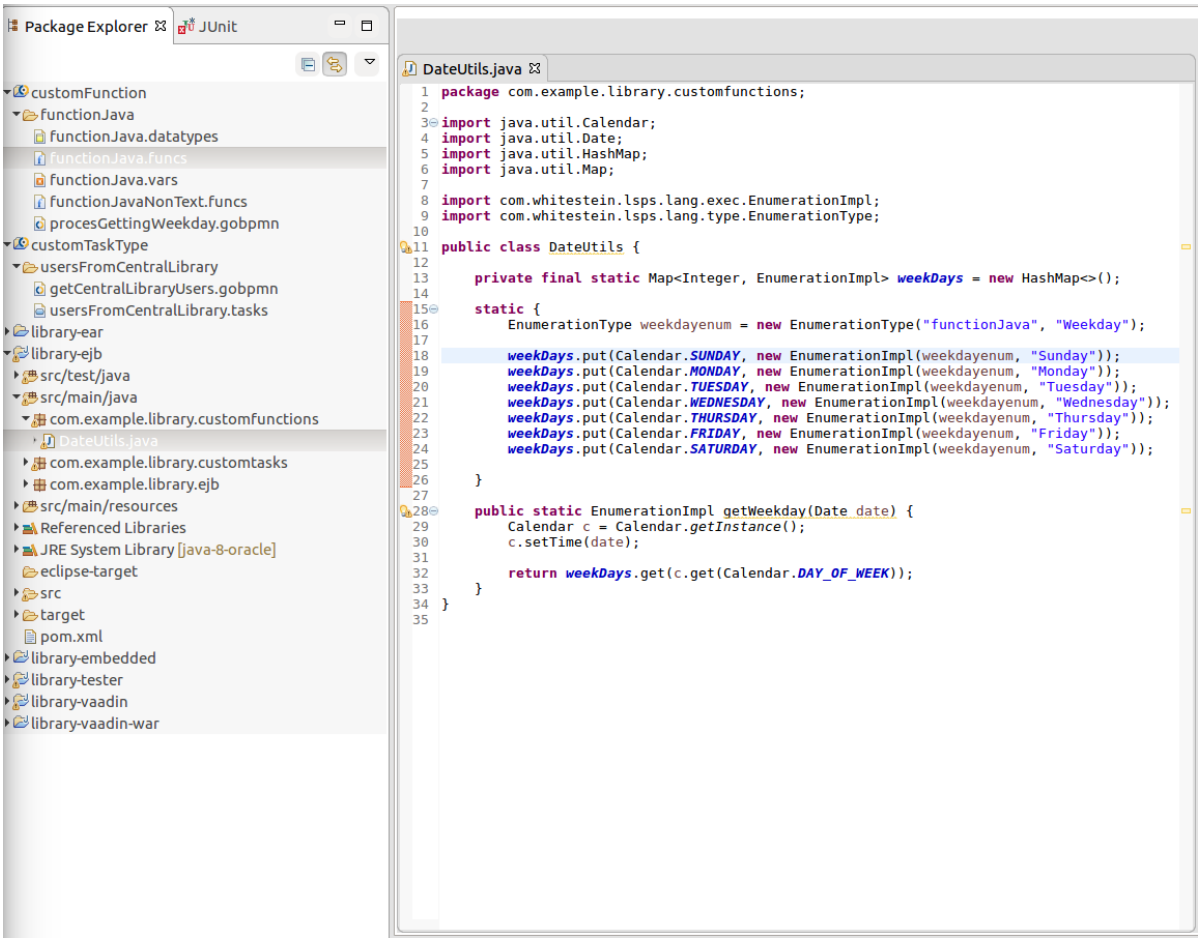


Figure 3.6 Custom function definition and declaration

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 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.

3.2.1.2.1 Task Execution

When a token enters a task, a task instance is created and becomes `Alive`: At this point, the *start()* method of the task is executed. The method can return either with `Result.FINISHED` or `Result.WAITING_FOR_INP↩UT`:

- If the method returns FINISHED, the task becomes `Accomplished`.
- If the method returns WAITING_FOR_INPUT, the task remains *Alive* it becomes a transaction boundary of the current `model transaction`. When such a task receives a message from the *CommunicationService*, its `processInput()` method is executed. If it returns `Result.FINISHED`, the task becomes Accomplished; if it returns `Result.WAITING_FOR_INPUT`, it remains Alive and waits for the next message.

When a task instance is terminated abnormally, for example, by a timeout intermediate event attached to the task, the *terminate()* method is called.

To create a custom task type, you will need to do the following:

- Declare the task type
- Implement the task type

3.2.1.2.2 Declaring a Task Type

To declare the task type and make it accessible in Designer, 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.
4. Under Task Type Details, enter the task-type name and in the *Classname* field enter the fully qualified name of the class. Mind that the task class must be in the `ejb` package of your application.

   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.

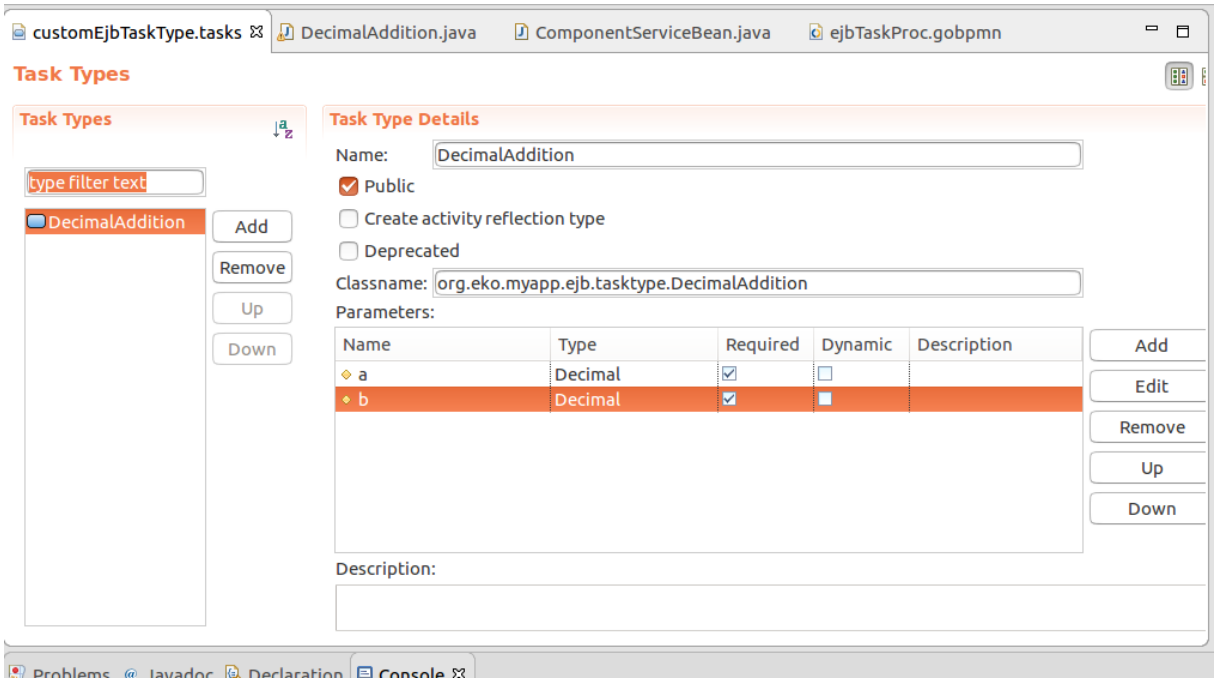


Figure 3.7 Task type declaration

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**.

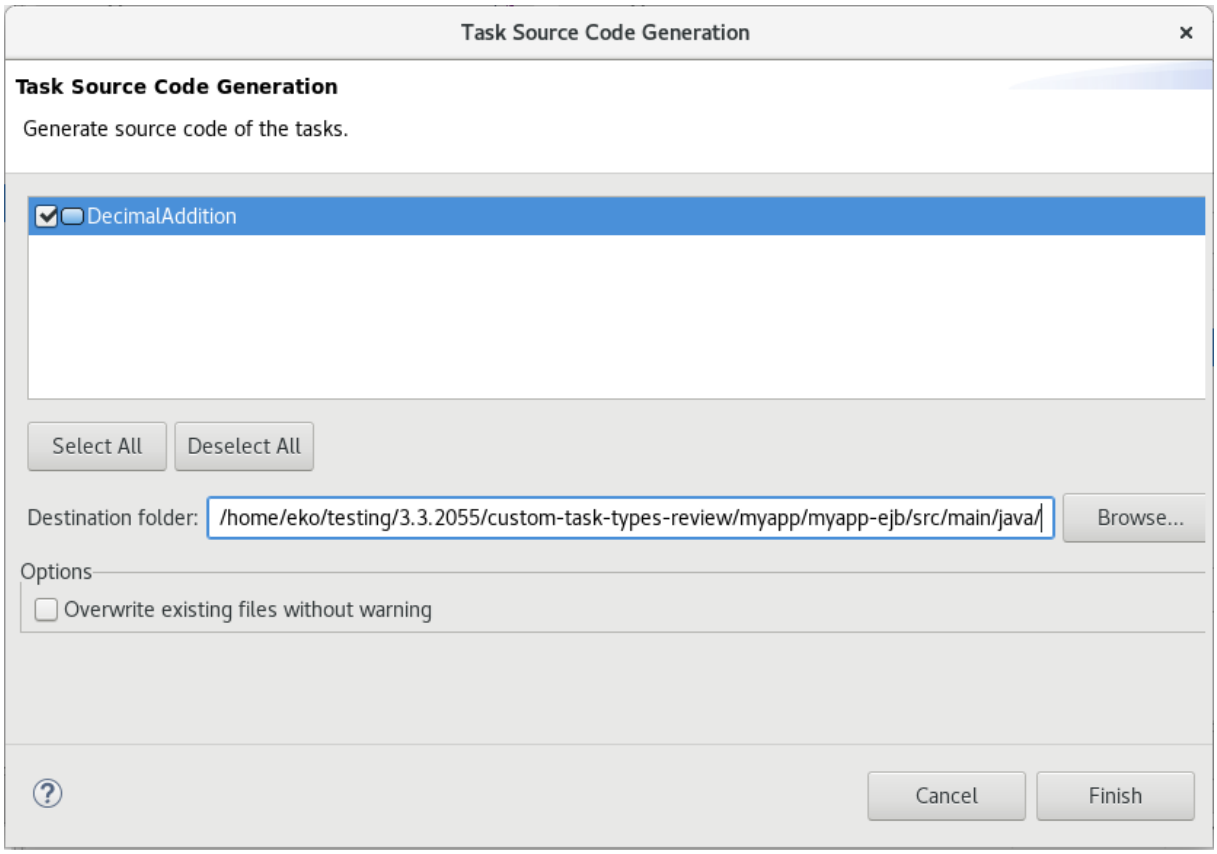


Figure 3.8 Task source code generation

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.

Consider distributing the `module as a library`.

3.2.1.2.3 Implementing a Task Type

You can implement a task type that:

- executes some logic and immediately finishes its execution
- or waits for a message to continue its execution,
- or is truly asynchronous.

3.2.1.2.3.1 Implementing a Simple Task Type

To implement a custom task type that starts and finishes in the same transaction and does not wait for any event, 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.

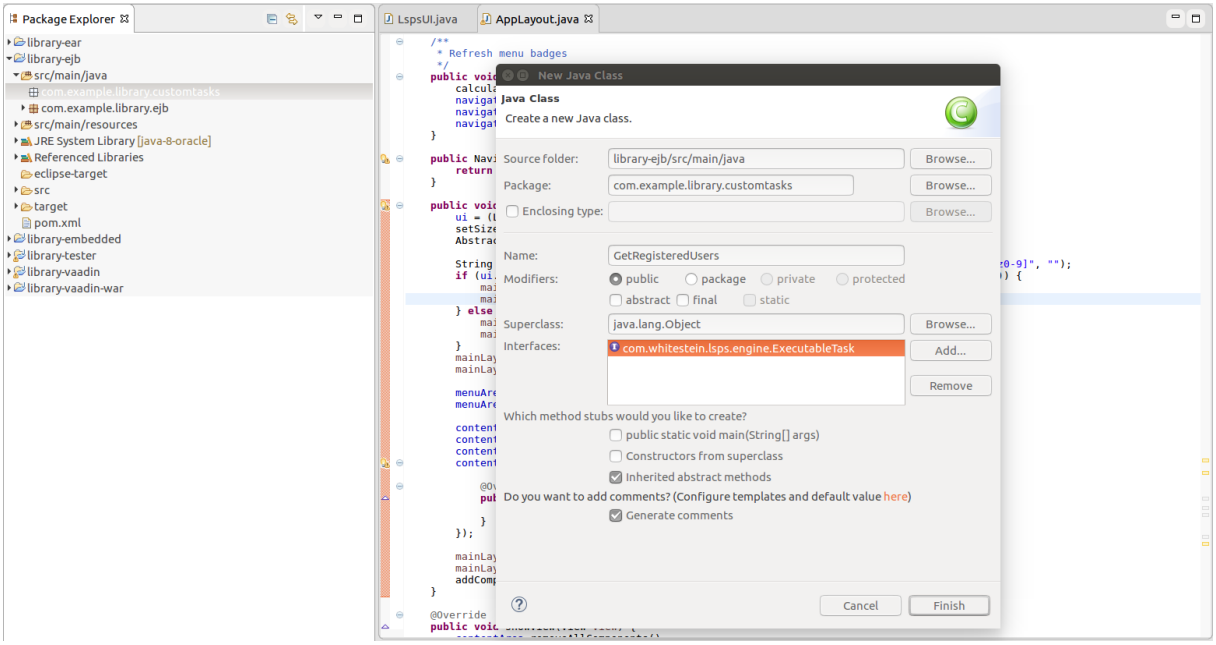


Figure 3.9 Creating class for implementation of the custom task type

4. Implement the logic in the `start()` method and make it return `Result.FINISHED`

   Important: The methods *start()* 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 the first parameter of 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.
7. If the implementing class is an EJB, register it with the ComponentServiceBean class.
8. Build and deploy the application.

You can now use the task in processes: make sure the task declaration is available to the process, for example, by the means of module import.

Simple 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 value of the param parameter of type String:
    String message = (String) context.getParameter("param");
~
    //to get the message parameter 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(QID.create("procVar"), message);
    //finish and release the token:
    return Result.FINISHED;
~
  }
~
  @Override
  public void terminate(TaskContext context, TerminationReason reason) throws ErrorException {
  }
}
```

3.2.1.2.3.2 Implementing a Task Type That Waits for a Message

To implement a custom task type that waits for an event and potentially continues in a new transaction, 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.
4. Make the task class a stateless EJB and add the `@PermitAll` annotation.
5. Implement the logic in the start() method and make it return `Result.FINISHED` and `Result.WAITI↩NG_FOR_INPUT` as appropriate.
6. Implement the `processInput()` method, which is executed when the task receives a message. The message is sent via the CommunicationService, which requires the task to be implemented as an EJB.
7. Implement the *terminate()* method, which is called when the task instance is terminated abnormally.

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

```
@Stateless
@PermitAll
public class WaitForInput implements ExecutableTask {
~
  @Override
```

```
  public Result start(TaskContext context) throws ErrorException {
~
    //implement the logic of the start method.
    return Result.WAITING_FOR_INPUT;
  }
~
  @Override
  public Result processInput(TaskContext context, Object input) throws ErrorException {
~
    return Result.FINISHED;
  }
~
  @Override
  public void terminate(TaskContext context, TerminationReason reason) throws ErrorException
  }
}
```

8. Register the EJB in the ComponentServiceBean:

```
@EJB(beanName = "WaitForInput")
private ExecutableTask waitForInput;
@Override
protected void registerCustomComponents() {
  register(waitForInput, WaitForInput.class);
}
```

9. Send the message to the task using the CommunicationService (when the task receives the message, it runs the `processInput()` method):

   (a) Store the id of the task so it can be accessed by the task or function, for example, from the `start()` method, or as a shared record from the module. You will need the id to address the message.

```
 @Override
  public Result start(TaskContext context) throws ErrorException {
~
    //saves the task id in process variable taskId
    //(taskId is modeled in the process):
    context.setVariableValue(QID.create("taskId"), new Decimal(context.getTaskId()));
    return Result.WAITING_FOR_INPUT;
  }
```

   (b) Declare a task or function that will send the message that will wake up the Task that finished previously with Result.WAITING_FOR_INPUT: the task will run the `processInput()` method.

```
 //example function declaration
//passing id of the target task as parameter:
public void wakeUpTask(Integer taskId)
   native org.eko.taskapp.ejb.WakeUpFunctions.wakeUpWaitingTask;
```

   (c) Implement the task or function as an EJB that calls the CommunicationService to send a message to the Task:

   - For the function, implement the function interface, function as an EJB, and register it with ComponentServiceBean.
   - For the task, implement the task as an EJB implementing ExecutableTask, and register it with ComponentServiceBean.

10. From the model, call the function or run the task that sends the message to trigger the `processInput()` method of the task.

Note: In the example, the `wakeUpTask()` call used within the same model instance (as the task context parameter, you get the context tree of the model instance, through the current process instance, to the task. If you want to call the task from another model instance, make the id of the model instance with the task accessible, for example, via database.

Example interface of the function

```
 @Local
public interface WakeUpFunctions {
  void wakeUpWaitTask(ExecutionContext context, Decimal taskId)
      throws ModelInstanceNotFoundException, InvalidModelInstanceStateException, ErrorException;
}
```

Example bean that implements the interface of the function

```
 @Stateless
@PermitAll
public class WakeUpFunctionsBean implements WakeUpFunctions {
  @EJB
  private CommunicationService communicationService;
~
  @Override
  public void wakeUpWaitTask(ExecutionContext context, Decimal taskId) throws ModelInstanceNotFoun
~
    //message from the current model instance to the task passed in the param:
    Message myMessage = new Message(Identifier.ofModelInstance(context.getModelInstance().getId())
        Identifier.ofTask(context.getModelInstance().getId(), taskId.longValue()));
    communicationService.sendAsync(myMessage);
  }
}
```

Example registration of the EJB

```
 @EJB(beanName = "WaitForInputTask")
private ExecutableTask waitForInput;
@EJB
private WakeUpFunctions wakeUpFunctions;
@Override
protected void registerCustomComponents() {
  register(waitForInput, WaitForInputTask.class);
  register(wakeUpFunctions, WakeUpFunctions.class);
}
```

Example task implementation that sends a message to the waiting task

```
 @Stateless
@PermitAll
public class WakeTask implements ExecutableTask {
  @EJB
  private CommunicationService communicationService;
  @Override
  public Result start(TaskContext context) throws ErrorException {
    context.getVariableValue(QID.create("taskId"));
    Message myMessage = new Message(
        Identifier.ofModelInstance(
            context.getModelInstance().getId()),
        Identifier.ofTask(
            context.getModelInstance().getId(), ((Decimal) context.getVariableValue(QID.create("ta
    communicationService.sendAsync(myMessage);
    return Result.FINISHED;
  }
...
```

3.2.1.2.4 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.

3.2.1.2.5 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);
```

3.2.1.2.5.1 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:
  (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 implemented in the Expression Language, both the implementation and declaration of the component are stored in a custom definition file in a GO-BPMN Module.
- When implementing in Java, you will:
  - create the component in Java and deploy it as part of your LSPS Application
  - 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` with the aim to isolate the data in the form so that a form can be discarded without leaving the data in an inconsistent state.

3.2.2.1.1 UI Component in Java

This section primarily describes how to implement a custom `UI component in Vaadin 8` component.

> **Important:** UI Vaadin 8 implementation is a new forms implementation and is not compatible with the previously used `UI Vaadin 7` implementation. A list of differences between the implementations is available in the `here`. To keep using the UI Vaadin 7 implementation, set UIComponentFactoryV7Impl as the UIComponentFactory. Where applicable, an admonition with instructions on how to implement UI Vaadin 7 when you are using UIComponentFactoryV7Impl.

To define and declare a custom form component, do the following:

1. If you require Vaadin components which are not available out-of-the-box, 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 the maven dependency to the Vaadin component jar file.
2. Create the custom component class in your application in a dedicated package in the `<YOUR_APP>-vaadin` project):
   The implementing class must meet the following requirements:
   - It implements `com.whitestein.lsps.vaadin.ui.components.UIComponent`.
   - It extends a class that implements the `com.vaadin.ui.<Component>` class. Do not extend the form components of the Standard Library since their methods might change.
     For UI Vaadin 7, import and extend the v7 versions of the components, for example, use `com.vaadin.v7.ui.Label` instead of `com.vaadin.ui.Label`.
   - It defines a constructor with the input parameter *UIComponentData*, a wrapper of the component record: the constructor sets the UIComponentData values.
   - 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. If you want to return multiple Vaadin components as a single LSPS custom component, return it from this method.
   - It *implements the `refresh()` method*.
     An example Label Component implementation in UI Vaadin 8

```
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;
  }
}
```

3. In the Modeling perspective, create the component record:

   (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 and created the subrecord UICustomLabel that will reflect our MyLabel class.

   Important: Make sure your module imports the `ui` module of the Standard Library.

   (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.

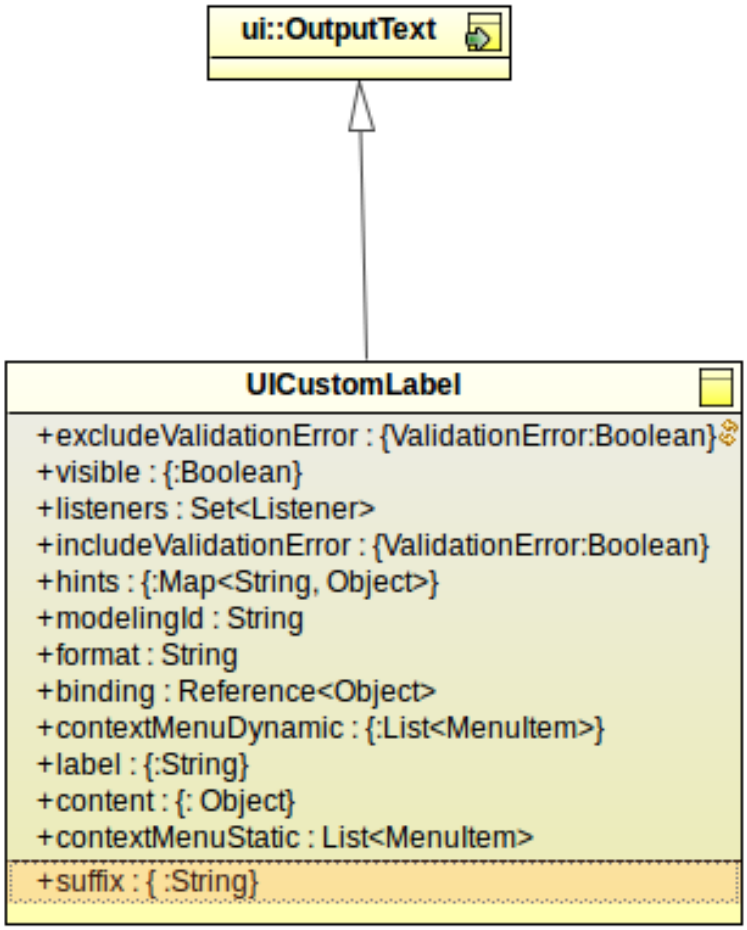


Figure 3.10 Custom component records

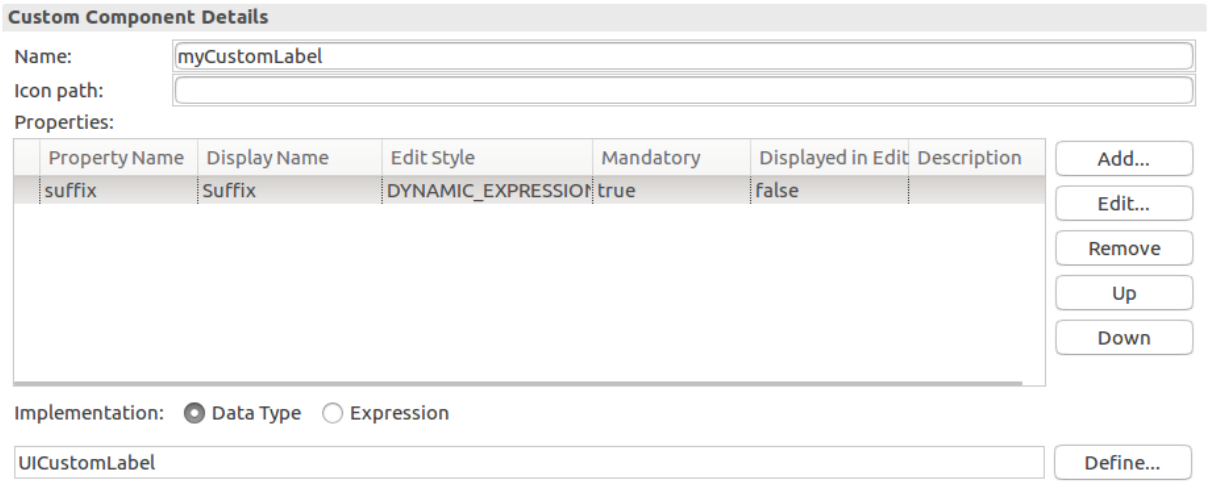
4. In a *custom component definition file* declare the component:

   (a) Set the Implementation property of the custom component definition to Data Type and enter the name of the component record.

   (b) In the Properties area, define the component properties that will be available for editing in the Properties view (in the case of the example Label, it is only the suffix property). For each property define the following:

   - 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.



5. Resolve the record of your form component to the implementation in your LSPS Application:

   (a) In the `connectors` package of the `vaadin` project, create a factory class that extends `UI↩ ComponentFactoryV8Impl`.

   For UI Vaadin 7, extend `UIComponentFactoryV7Impl`

   (b) Override the createComponent() method so it returns your component when the record from your form definition is requested.

```
public class MyUIComponentFactoryV8Impl extends UIComponentFactoryV8Impl {
~
  public MyUIComponentFactoryV8Impl(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);
  }
}
```

   (c) In the `DefaultLspsAppConnector` class in the core package of the vaadin project, return your factory in `getComponentFactory()`.

```
public class DefaultLspsAppConnector extends LspsAppConnectorImpl {
~
  private static final long serialVersionUID = 6600639435905976895L;
~
  public DefaultLspsAppConnector(LspsUI ui) {
    super(ui);
  }
~
  @Override
  public UIComponentFactory getComponentFactory() {
    //returns the custom ui factory:
    return new MyUIComponentFactoryV8Impl(this);
  }
```

6. Build and deploy your application.

You can now use the custom component in your ui definition.

Don't forget to upload the module with your ui component record and declaration along with the LSPS Application. Consider distributing the components as part of a `library`.

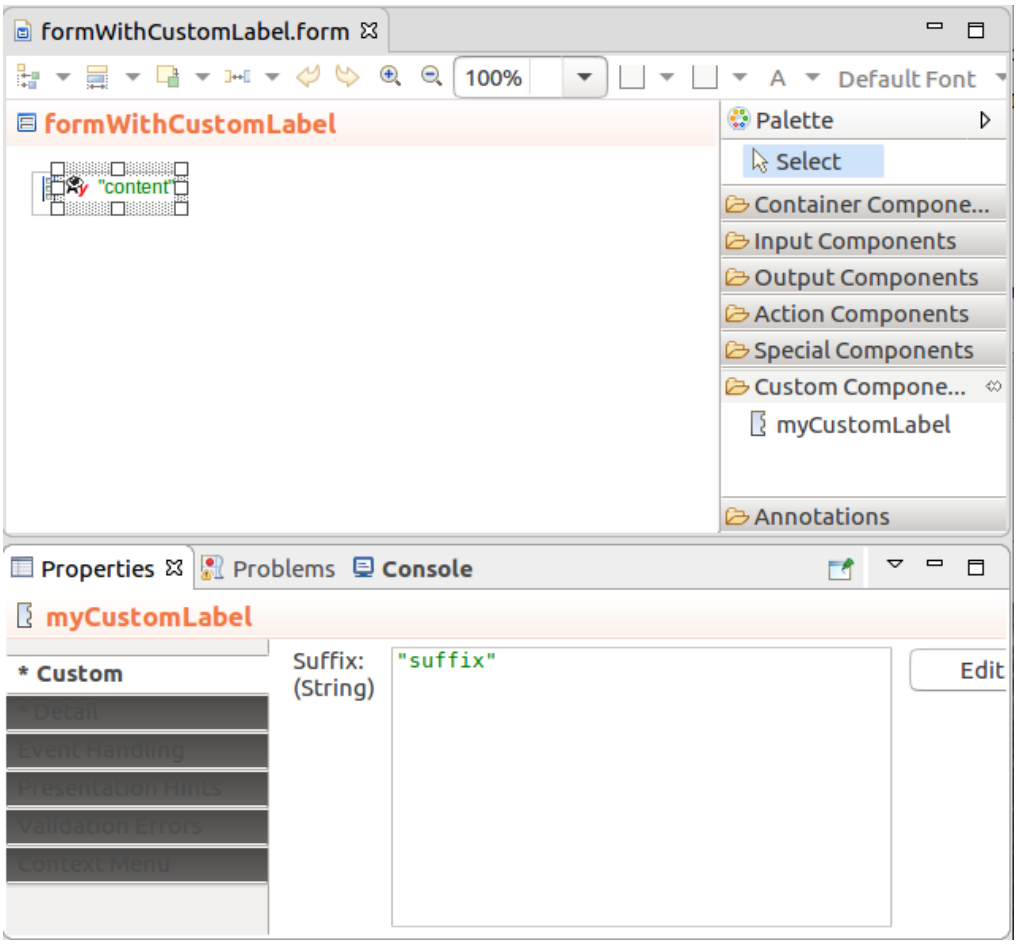


Figure 3.11 Custom component in form definition (inherited properties are in the Detail tab)

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.

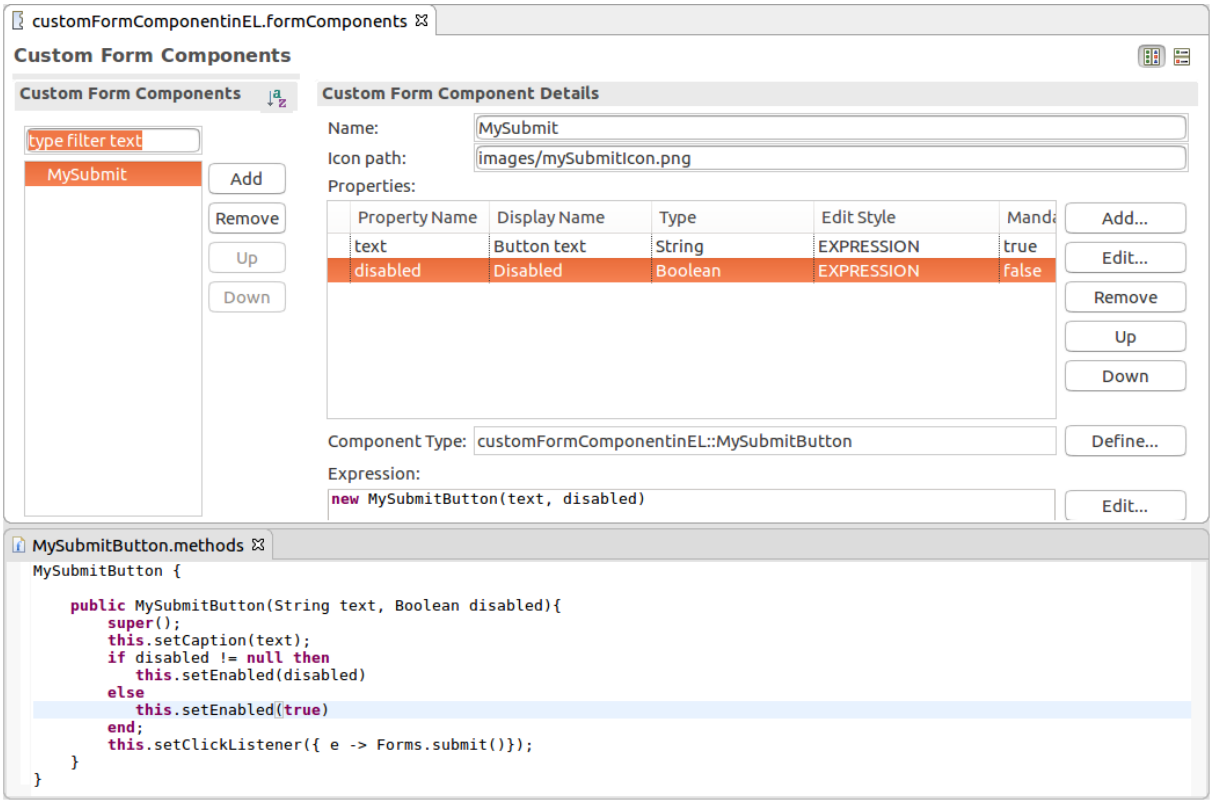


Figure 3.12 A custom component with a parameter

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:
   - Make it extend `UIEvent`.
   - Pass the data type as the second parameter of the event constructor.
     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. UIComponentData includes such data about the component as its parent component, its depth from the root component, etc.
   (a) Create the custom listener.
   The listener should override the method that creates the event on the component, in the example `colorChanged(ColorChangeEvent e)` so the event is transformed into a custom event. It enters the `event queue of the event-processing cycle` when `UIComponents.fireAndProcess()` method is called.
   (b) Register the component in the `LspsAppComponentFactory` class: uncomment the `createComponent` method and add the constructor call for your component that is called when the respective Record is requested.

```
package org.eko.ekoapp.vaadin.util;
~
import org.eko.ekoapp.vaadin.components.UIText;
~
import com.whitestein.lsps.vaadin.LspsAppConnector;
import com.whitestein.lsps.vaadin.ui.UIComponentData;
import com.whitestein.lsps.vaadin.ui.UIComponentFactoryImpl;
import com.whitestein.lsps.vaadin.ui.components.UIComponent;
~
public class MyComponentFactory extends UIComponentFactoryImpl {
~
    public MyComponentFactory(LspsAppConnector connector)
            throws NullPointerException {
        super(connector);
    }
~
    @Override
    protected UIComponent createComponent(UIComponentData componentData) {
        String type = componentData.getComponentDefinition().getType()
                .getFullName();
        if (type.equals("customComponentModule::TextComponentRecord")) {
            return new UIText(componentData);
        }
        return super.createComponent(componentData);
    }
}
```

(c) 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);
```

(d) 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());
  }
```

(e) 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 Designer:

   (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

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.

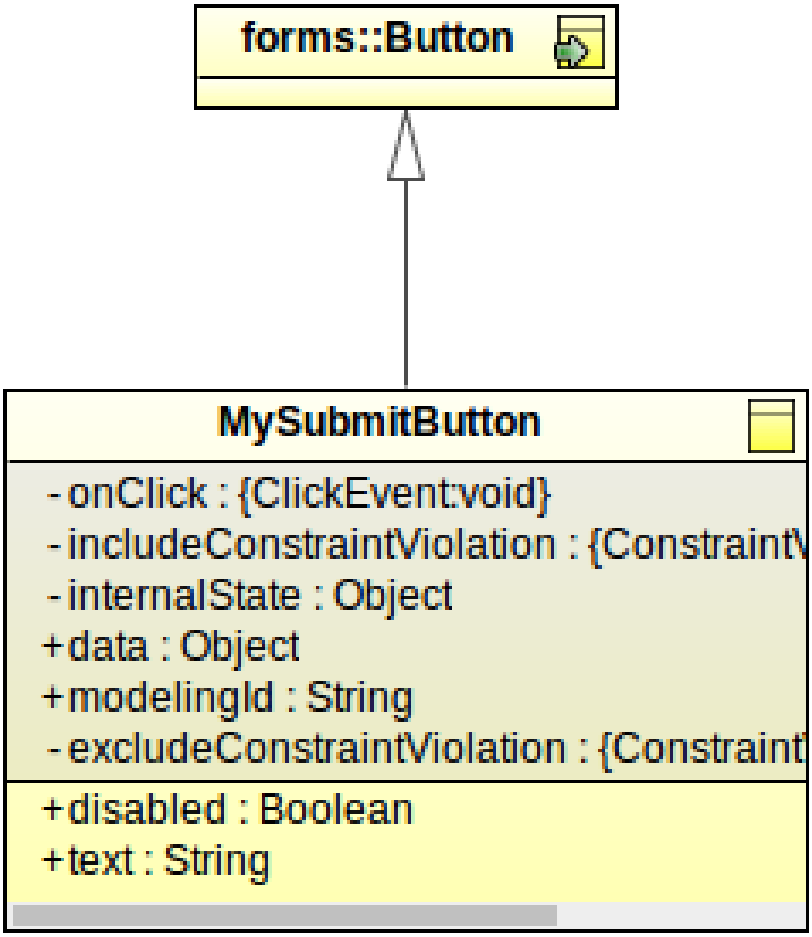


Figure 3.13 Example record of a custom form component

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.

6. Close and reopen any forms for the component to appear in the palette.

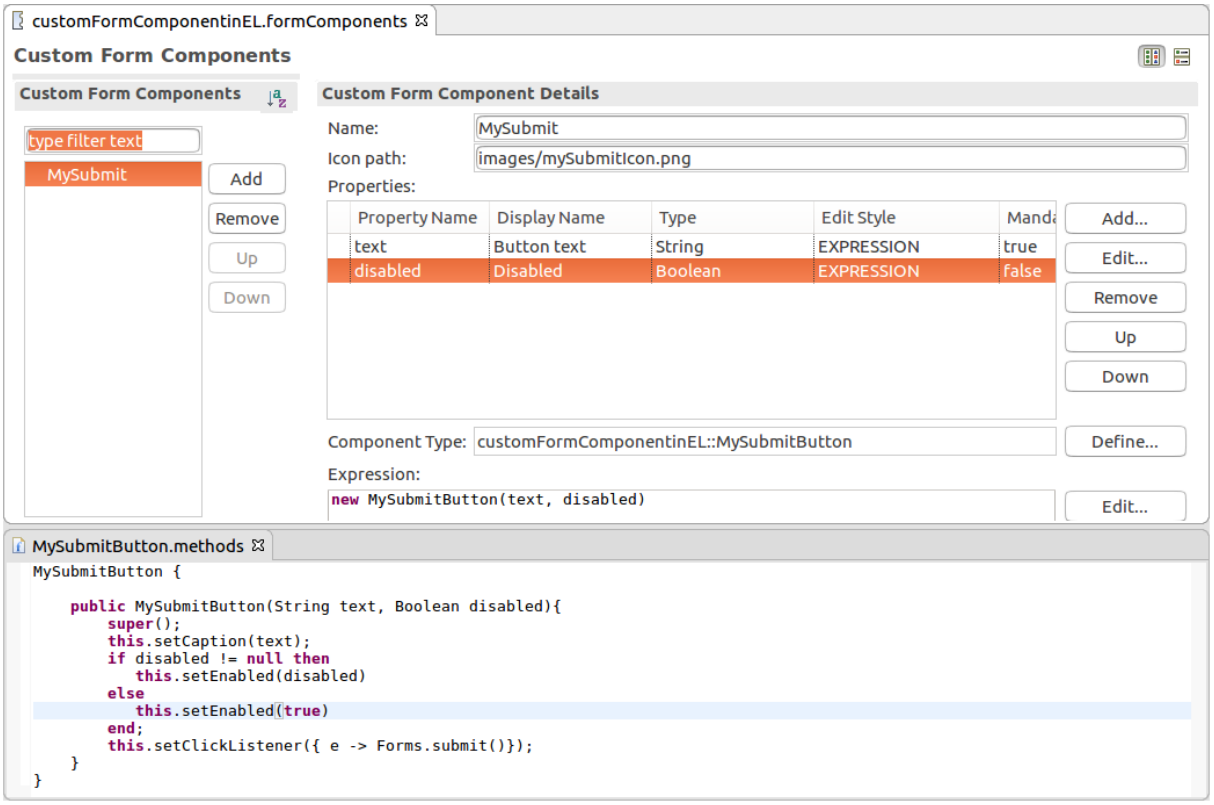


Figure 3.14 A custom component definition and the component record constructor

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 doesn't 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. Create the Vaadin implementation in a package in the <YOUR_APP>-vaadin project.

   The implementation might be provided by Vaadin already. This is the scenario used in the example: creating the custom component for the Vaadin's Colorpicker class.
3. In a new package in <YOUR_APP>-vaadin, create a Java class that will connect the LSPS component to its Vaadin implementation:

   - If the component makes use of LSPS mechanisms, the class must extend the `FormComponent` class or its subclass.

     You can either inherit from LSPS components, typically prefixed with `W`: For the *forms::TextArea* record, the implementation is *com.whitestein.lsps.vaadin.forms.WTextArea* class; for generic input components, it is `WInputComponentWithValue`, etc.

     Important: Do not use components of Vaadin 7 located in the com.vaadin.v7.ui package. These are included for compatibility purposes and will be removed.

- 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().setValue(color);
  }
~
  public RecordHolder getColor() {
    Color color = getWidget().getValue();
    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 component returned by your `createWidget()` method. If your Vaadin component extends the *com.vaadin.ui.CustomComponent* class as it is the case of composite components, your record should have *forms::FormComponent* as its super type.
     It is not recommended to add record fields to the record since the record serves to reflect a Vaadin component in LSPS and Vaadin components are intended for presentation, not for business logic.
   - 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.

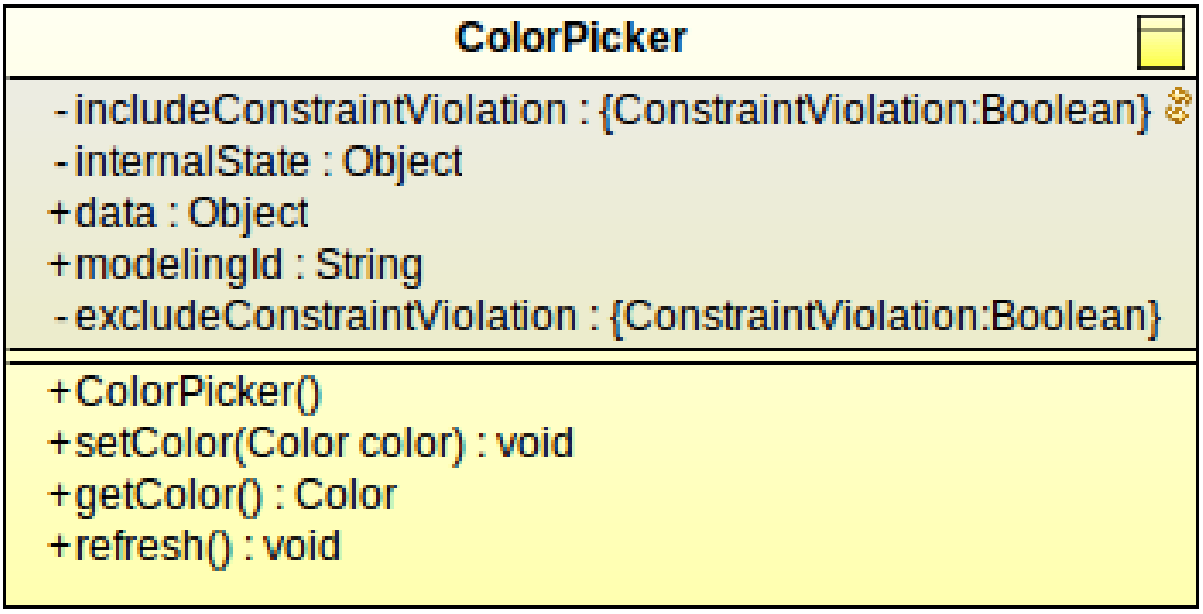


Figure 3.15 Custom component record with FormComponent supertype; when you enter the super type, refresh the record to display the inherited fields

```
   ColorPicker {
~
    public void setColor(Color color){
      //calls the setColor(color) method on the connecting class:
       call("setColor", [color]);
    }
~
    public Color getColor(){
      //calls the getColor(color) method on the connecting class:
        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.

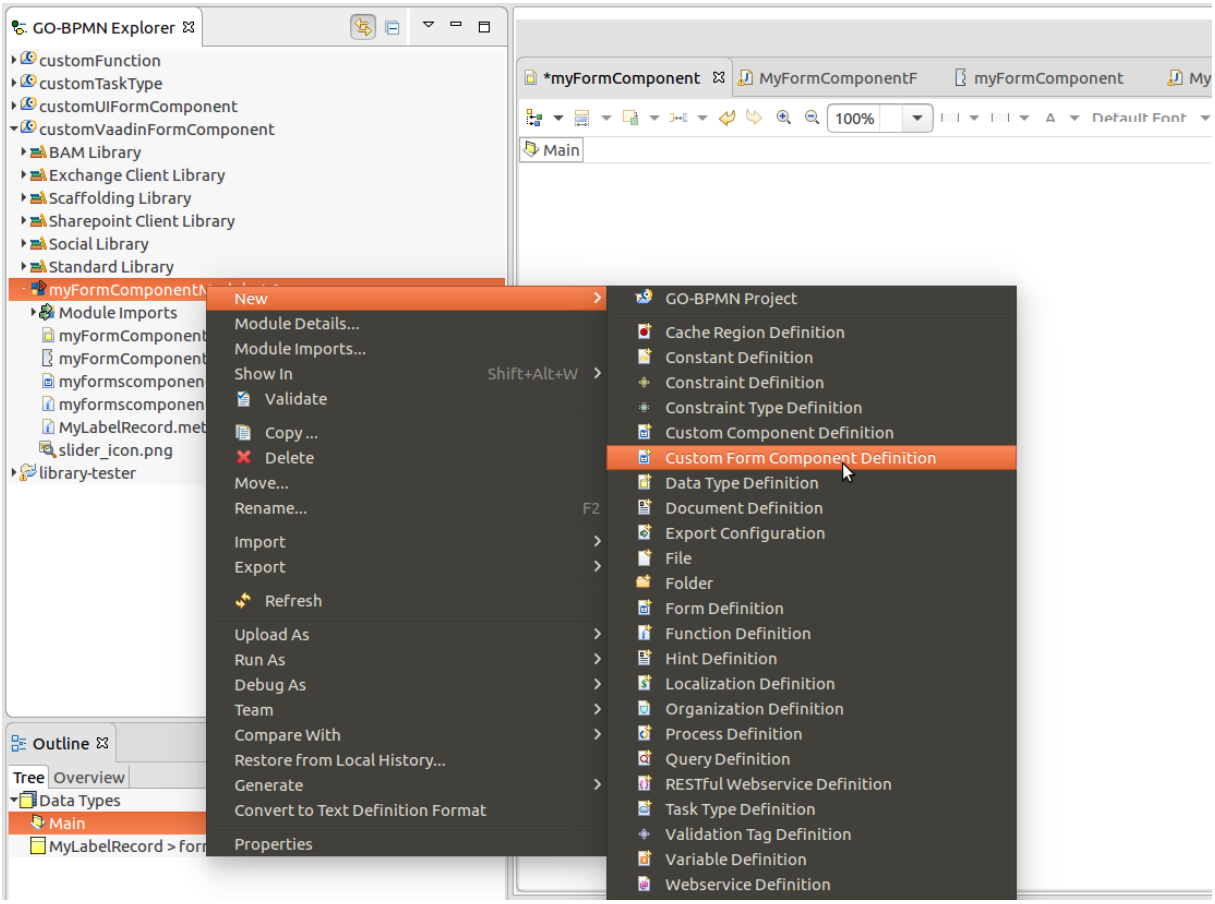


Figure 3.16 Creating custom form component definition file

(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.

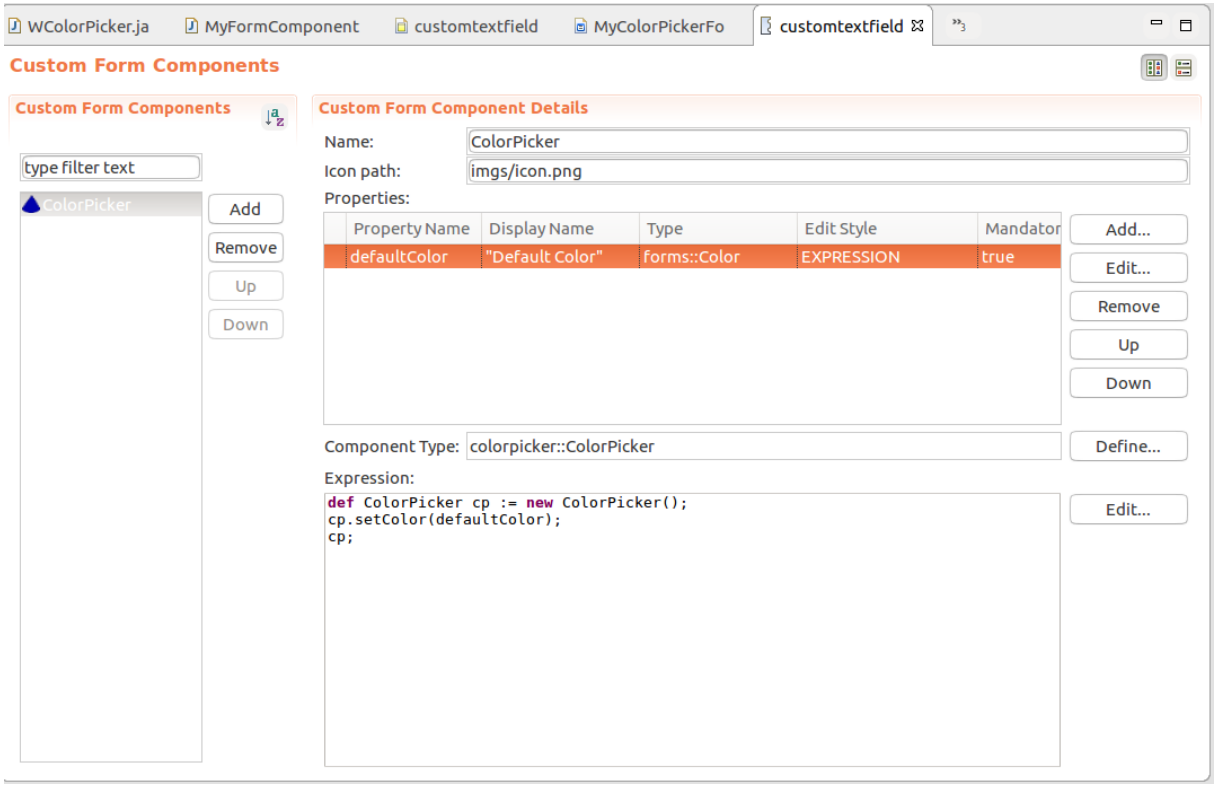


Figure 3.17 Custom form component definition with a property

5. In the `LspsFormComponentFactory` class of your LSPS Application (connectors package), connect the LSPS record to its Vaadin implementation: Modify the `create()` method that returns FormComponent to return your 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.

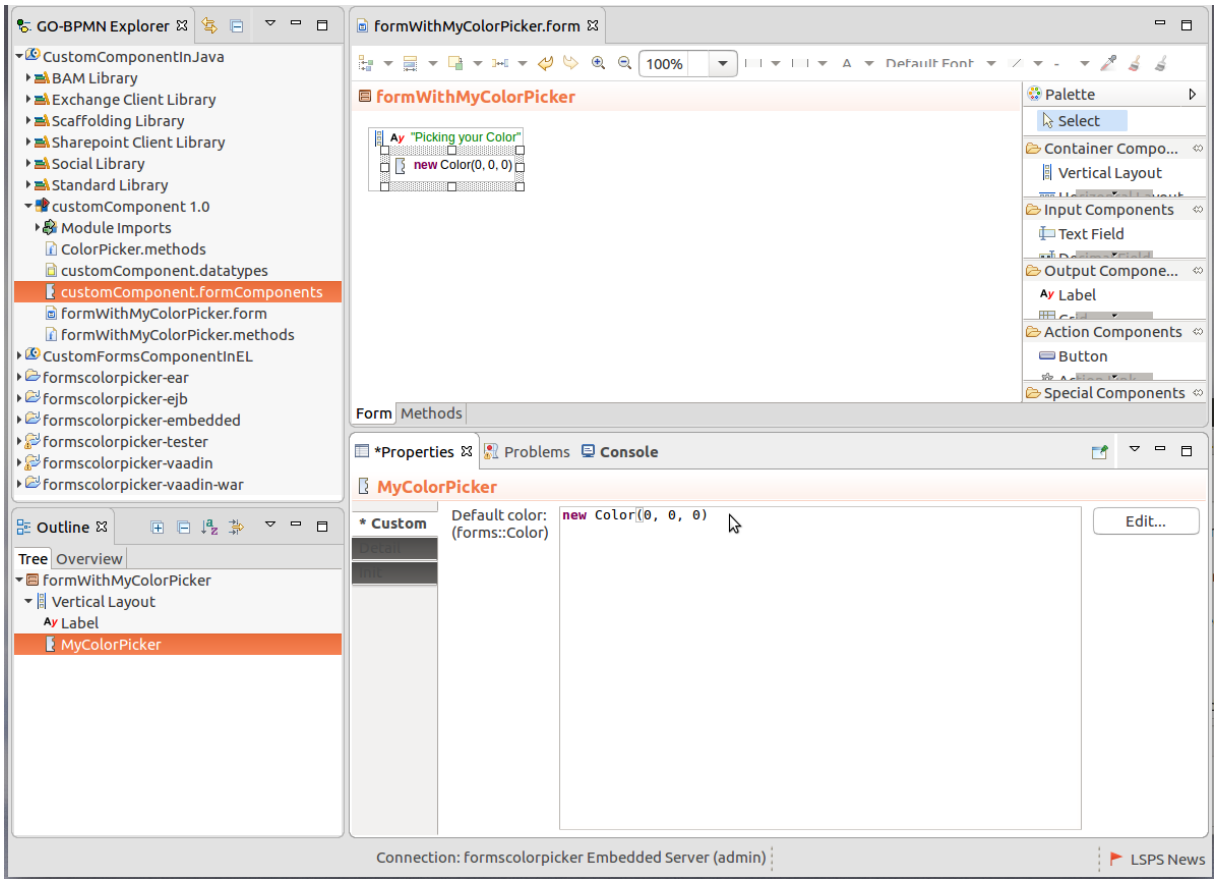


Figure 3.18 Custom component in a form definition

3.2.2.2.2.1 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().getValue());
   }
   ```

   (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 3.19 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) {
  final String type = rendererDef.getTypeFullName();
  //return the renderer in the if block for the record that reflects the renderer:
  if (type.equals("gridModule::ProgressBarRenderer")) {
  //add the if with the renderer record and the call to return the progress bar via the conne
    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
   */
```

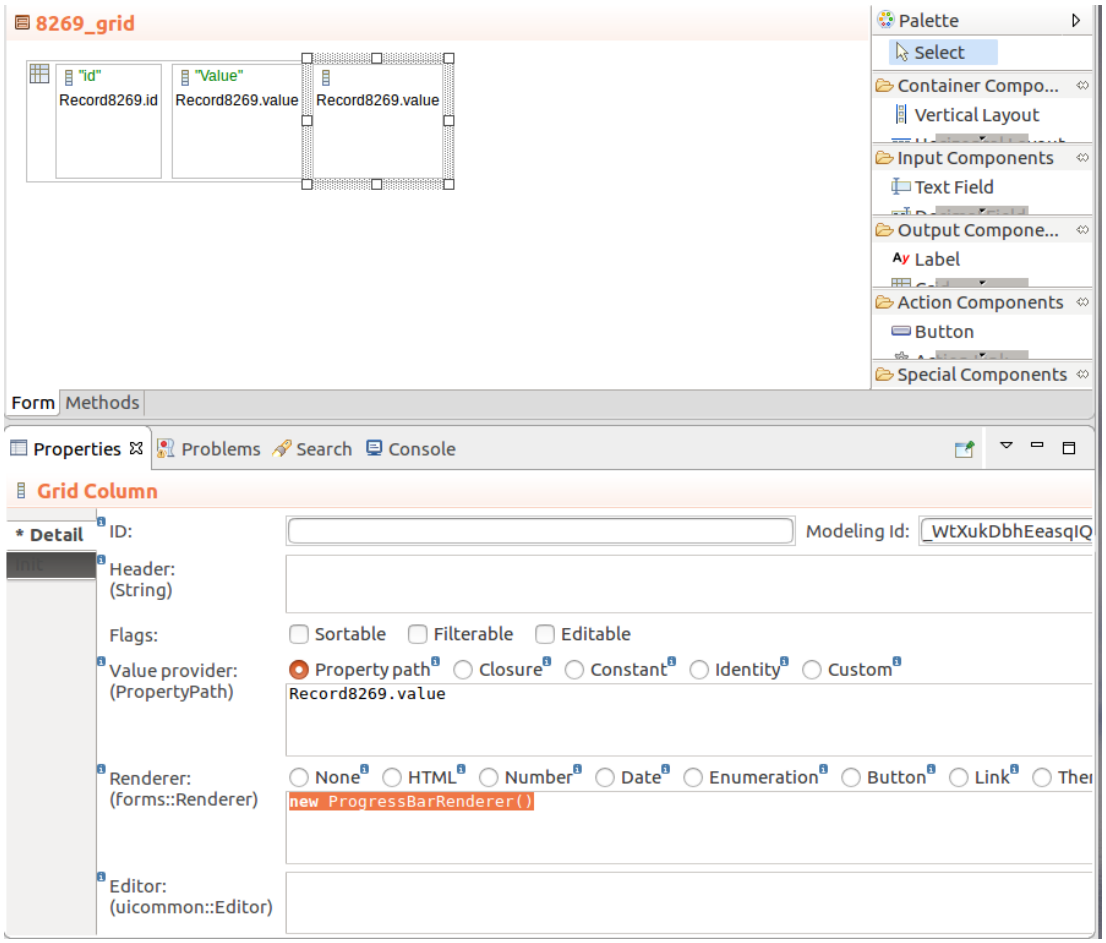
```
  private static final long serialVersionUID = 1L;
~
  @Override
  public Decimal convertToModel(Double value, Class<? extends Decimal> targetType, Locale
    return value == null ? null : new Decimal(value);
  }
~
  @Override
  public Double convertToPresentation(Decimal value, Class<? extends Double> 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 *LspsFormComponentFactory()* class.

```
@Override
public Converter<?, ?> createConverterForRenderer(WGrid owner, Variant.RecordVariant rend
  if (type.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.FormComponent):
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 Java Classes and Interfaces for Data Types

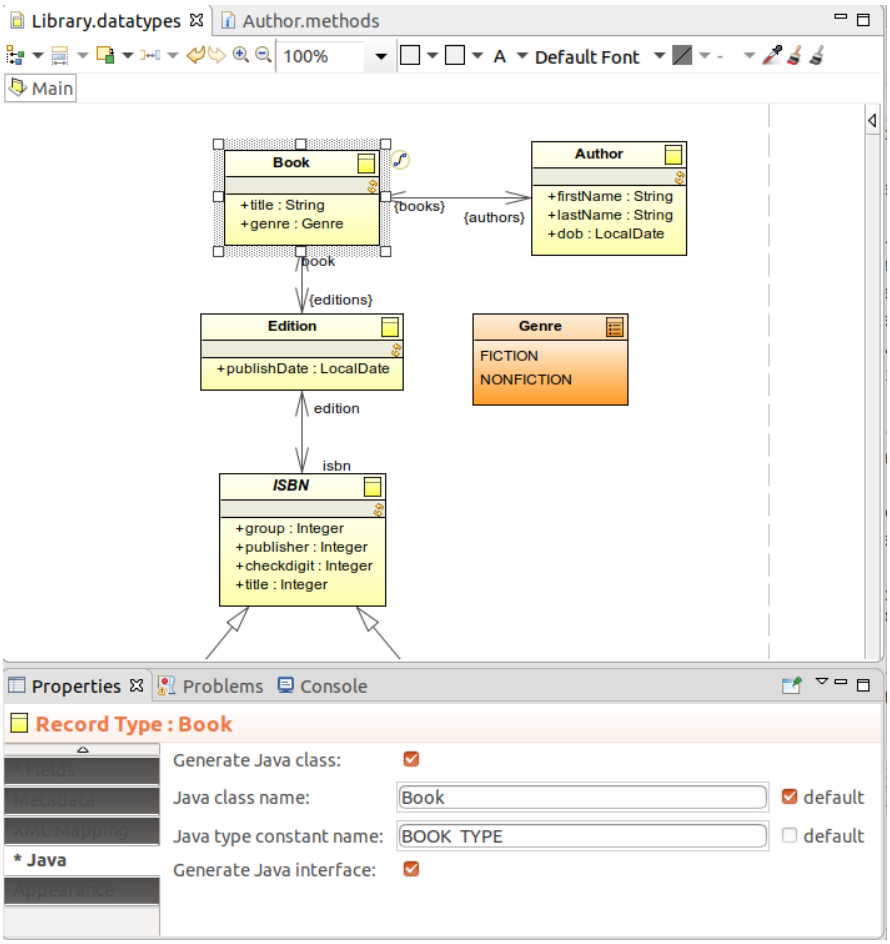
To work more efficiently with elements of your data type model and record instances in the LSPS Application, generate the Java wrapper classes and interfaces for the types and use these classes rather than using the RecordHolder class to work with Records.

Example of difference in coding

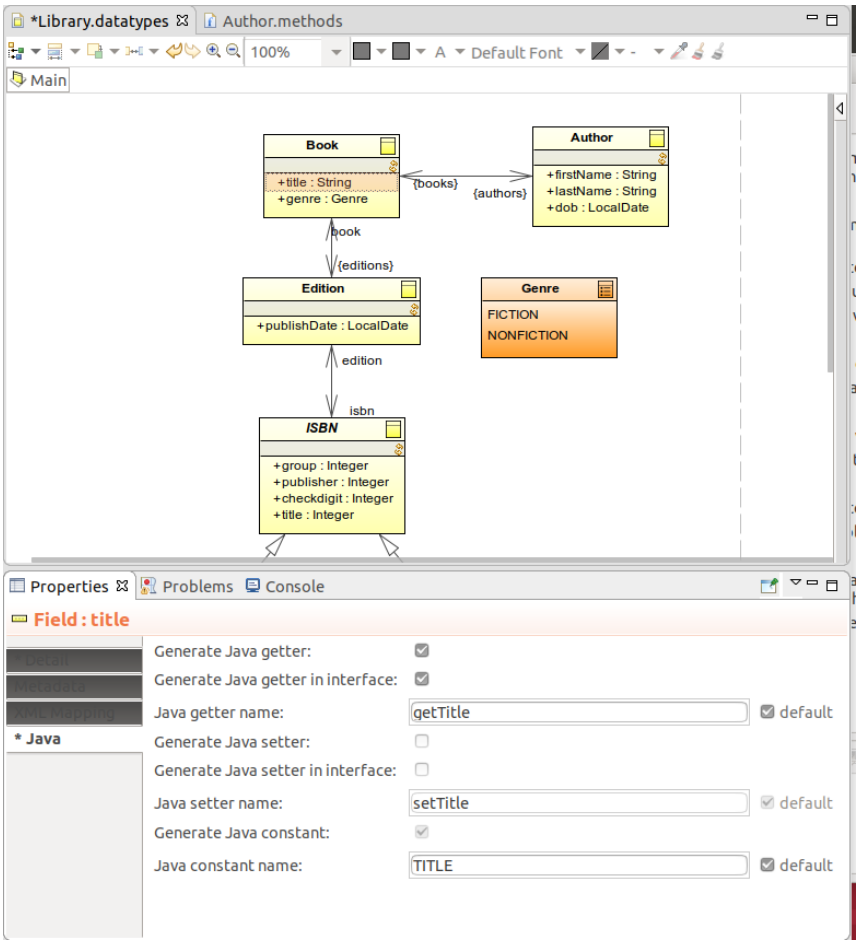
```
// original access via context:
RecordHolder record = ctx.getNamespace().createRecord("core::ConstraintViolation");
record.setProperty("message", msg);
~
// better with generated Java class for records:
ConstraintViolationRecord better = new ConstraintViolationRecord(record);
better.setMessage(msg);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

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

1. On the data-type element, define the properties of its class and interface to be generated:
   (a) Select the record, interface, or enumeration.
   (b) Open the *Java* tab in the Properties view:
   - To generate a class, select the **Generate Java class** option. Optionally, just the output class name and the constant name with the path to the record.
   - To generate an interface, select the **Generate Java interface**.



2. Open the Properties view of individual record fields and on the *Java* tab, select the methods you want to include in the generated class and interface. Note that methods definitions of Records are ignored: only getters and setters of record fields are generated as per this configuration.



3. Save the changes.

4. Generate the classes and interfaces:

   (a) Right-click a module or project with the records and go to **Generate** > **Record Java Sources**.

   (b) In the dialog, define the properties:

   (c) Select the modules, for which to generate the Java sources.

   (d) For each selected module, define the properties for the classes and interfaces:

   - Source folder: target src folder in a Java project
   - Package: target package name
   - 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
   - Interface extends interface: common interface extended by the generated interfaces
   - Additional record types: other records that should be included in the operation
     The parameter is primarily intended for inclusion of library records.

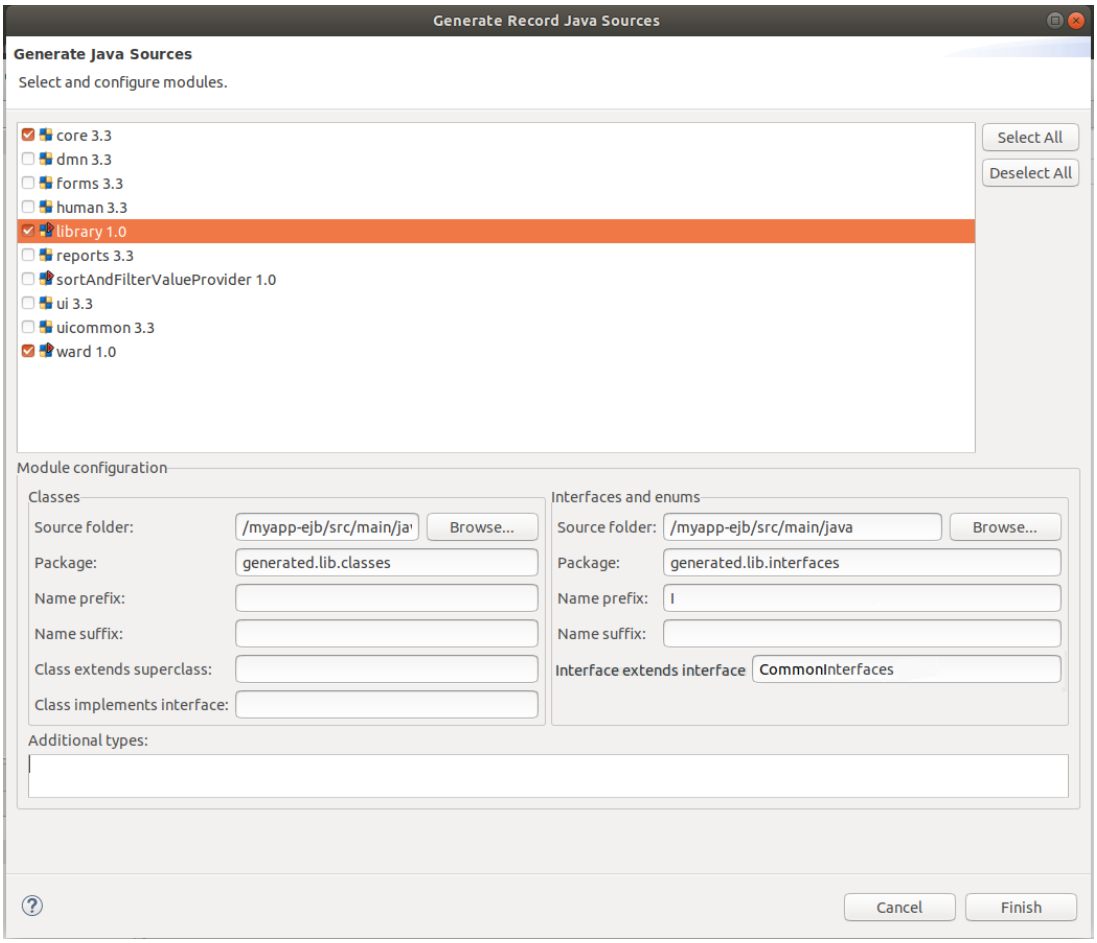


Figure 3.20 Generating Java record wrappers

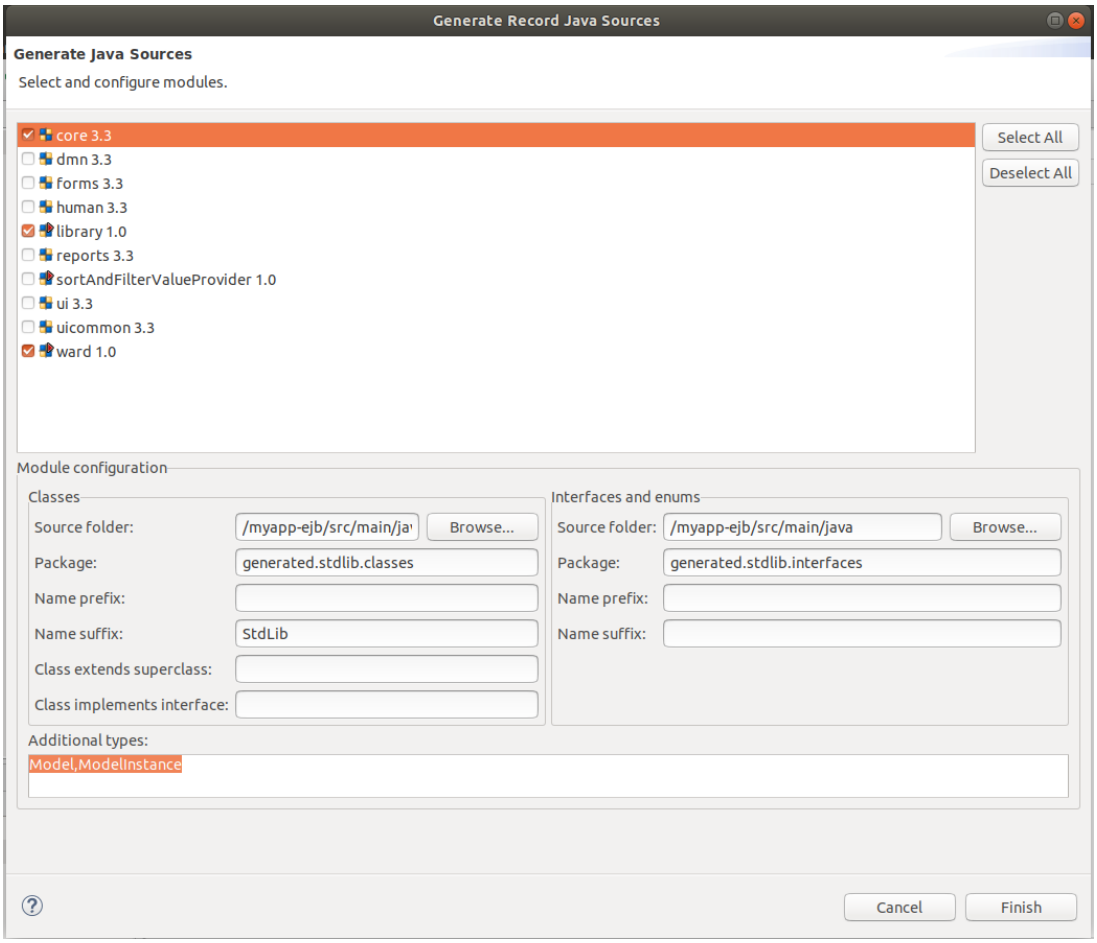


Figure 3.21 Generating Java record wrappers for a library

5. Refresh the project with the target src directory.

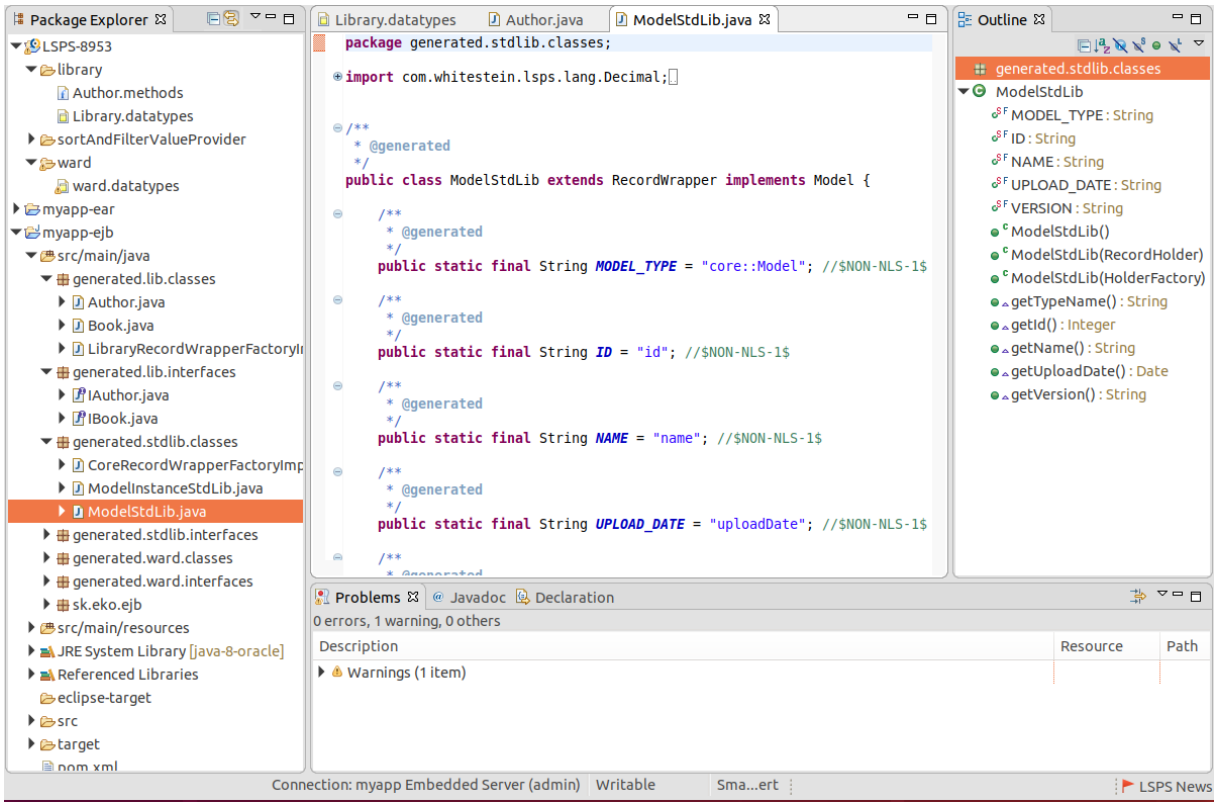


Figure 3.22 Generated Java record wrappers

Implementation of a custom function and its definition

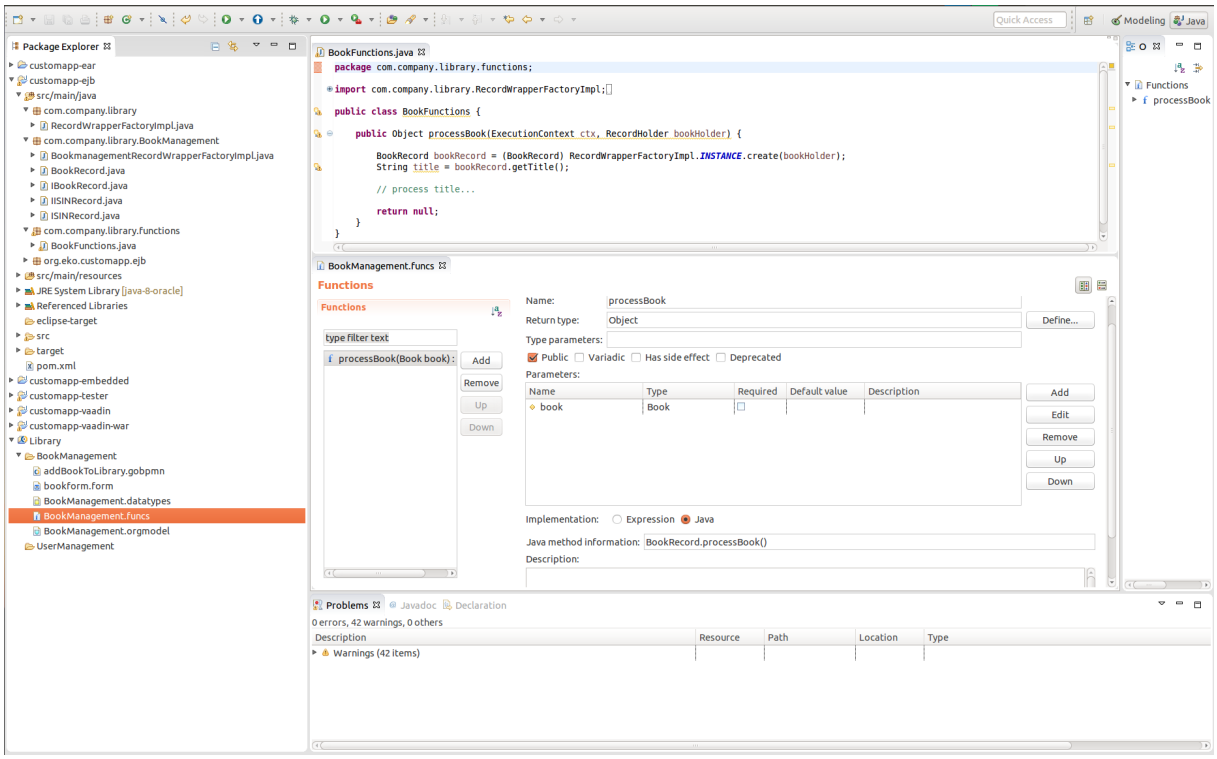
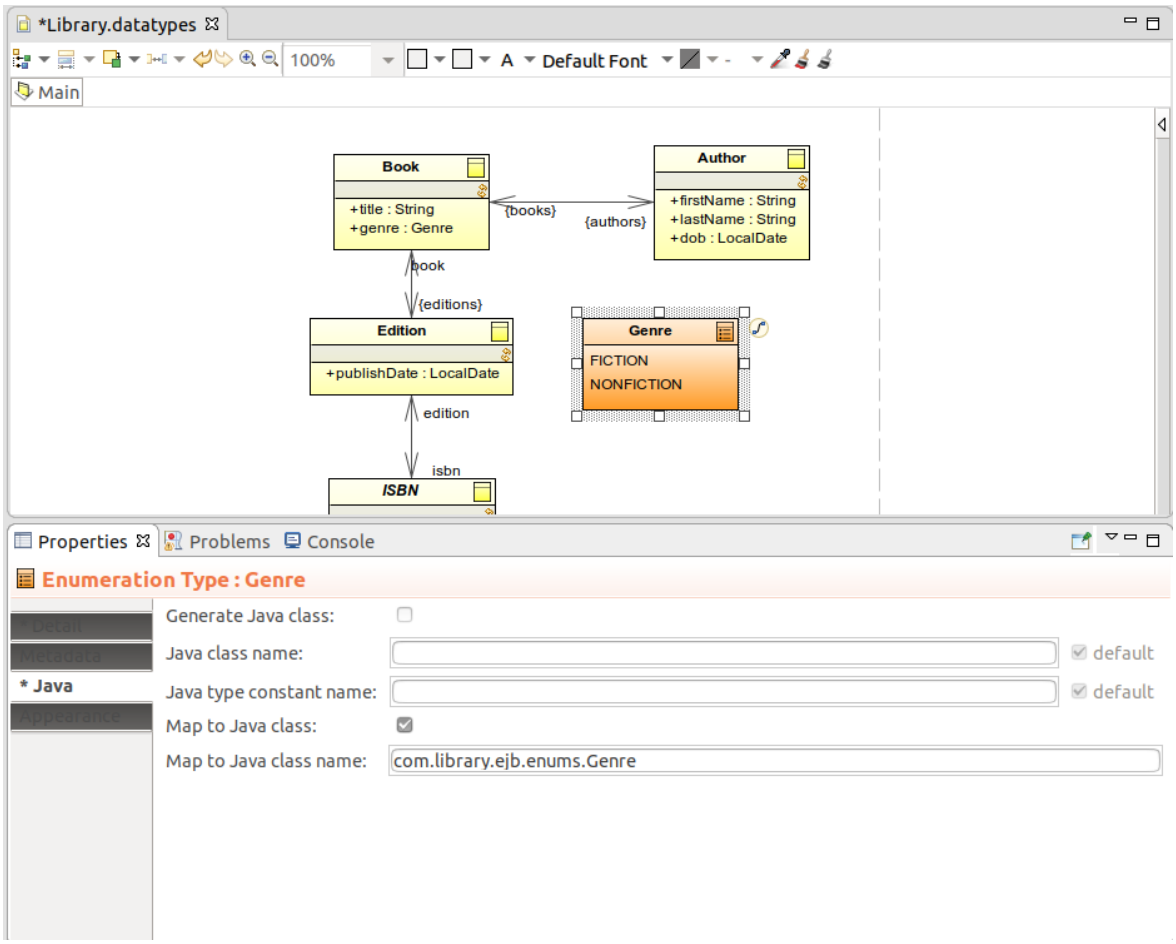


Figure 3.23 Custom Function implementation and definition

3.3.2.2 Mapping Enumerations to Existing Java Classes

If you have existing Java implementations of your Enumerations, you can map them to the Enumerations in your module to prevent them from being generated when you generate the Java classes of your records:

1. Select the Enumeration.
2. Open the *Java* tab in the Properties view.
3. Select *Map to Java class name* and enter the class name of the implementation.



3.3.2.3 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;
}
```

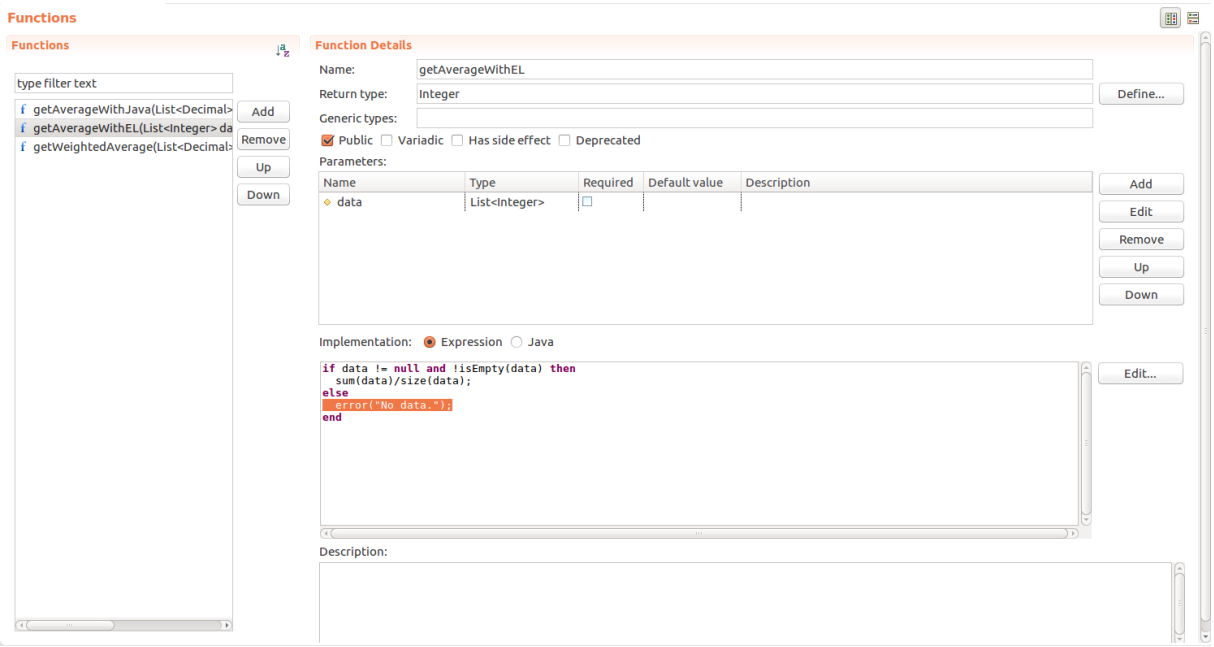
3.3.4 Throwing an Error

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

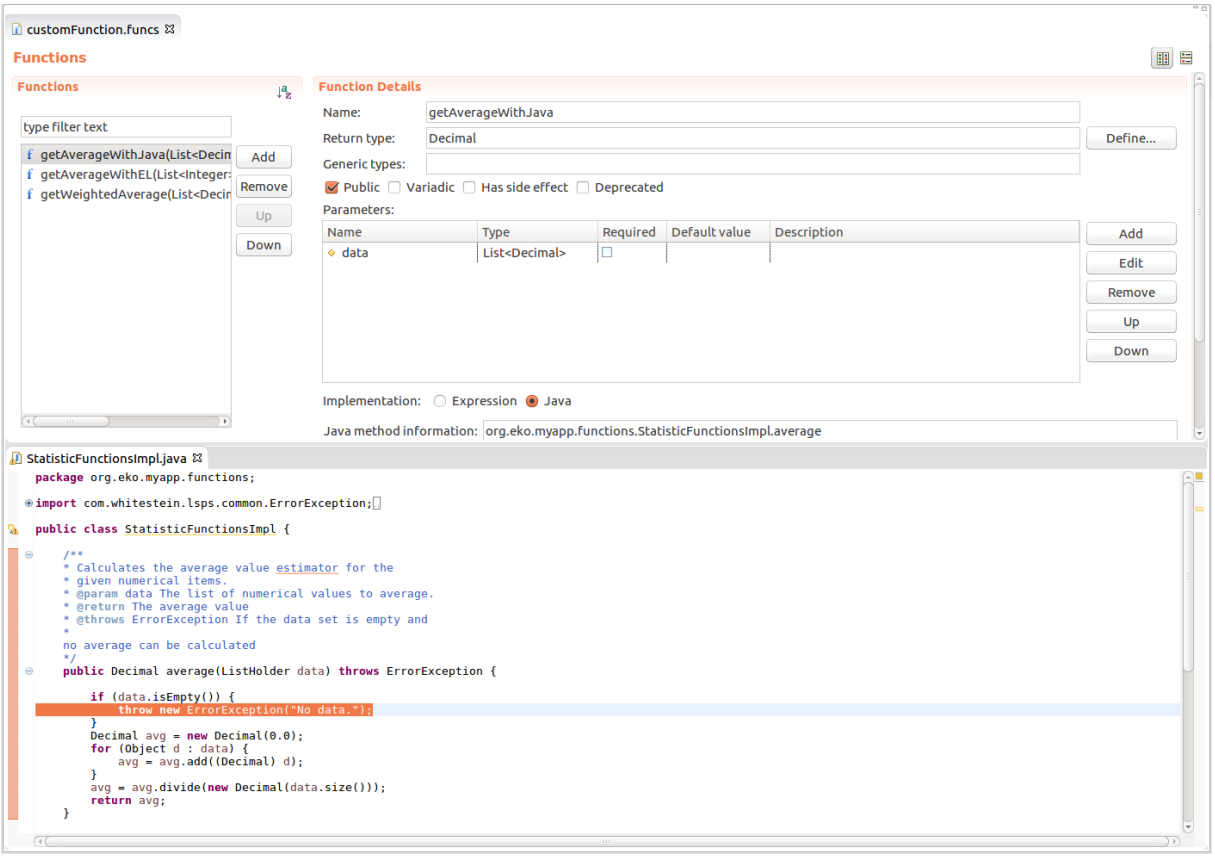
To catch error, use the `Error Start Events` or `Error Intermediate Events`.

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

To use the command-line tool from a Java program, do the following:

1. Include the dependency to the command line Management Console 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.

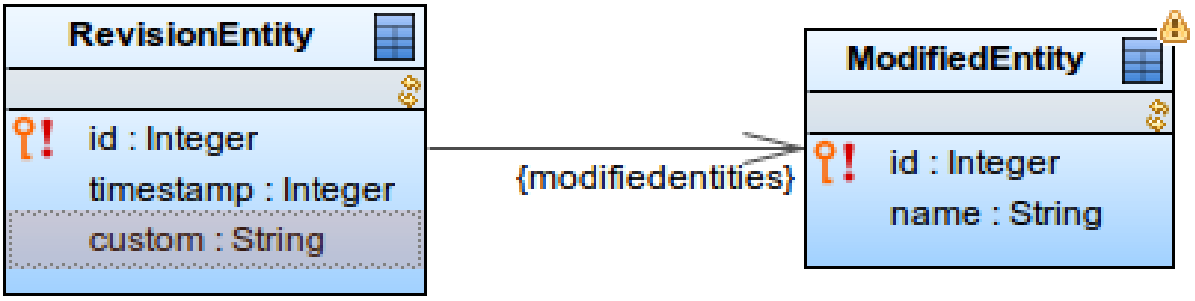


Figure 3.24 Revision entity record with the custom field

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.

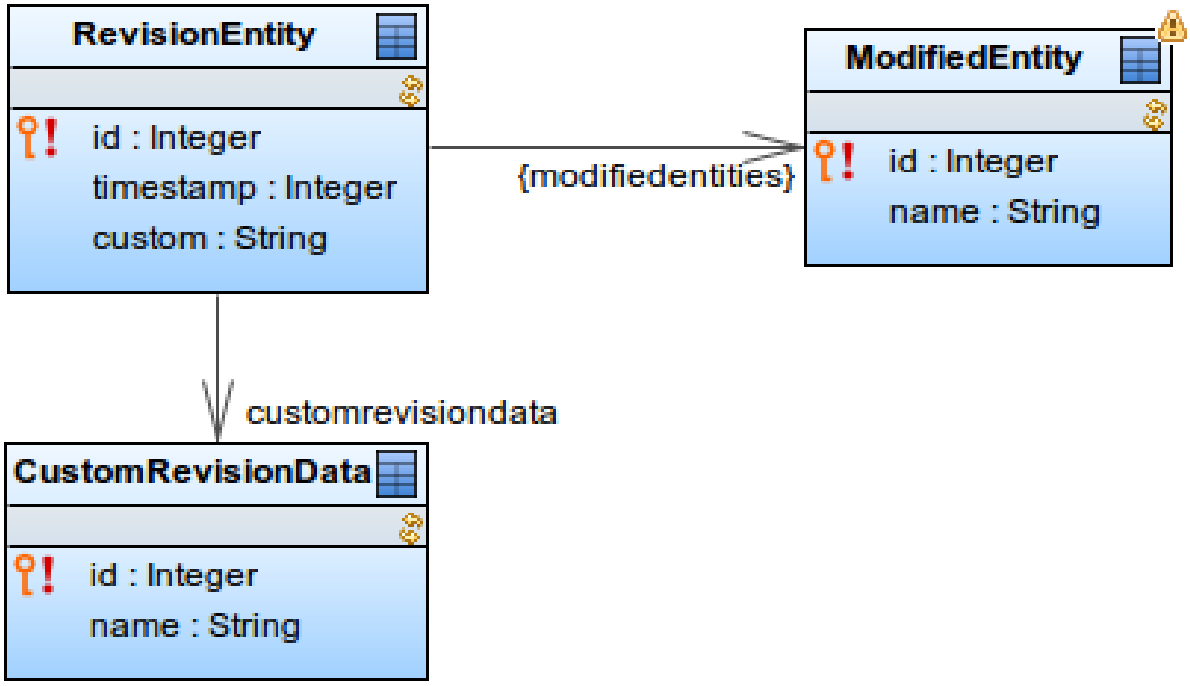


Figure 3.25 Revision entity record with the custom field and a related record

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.

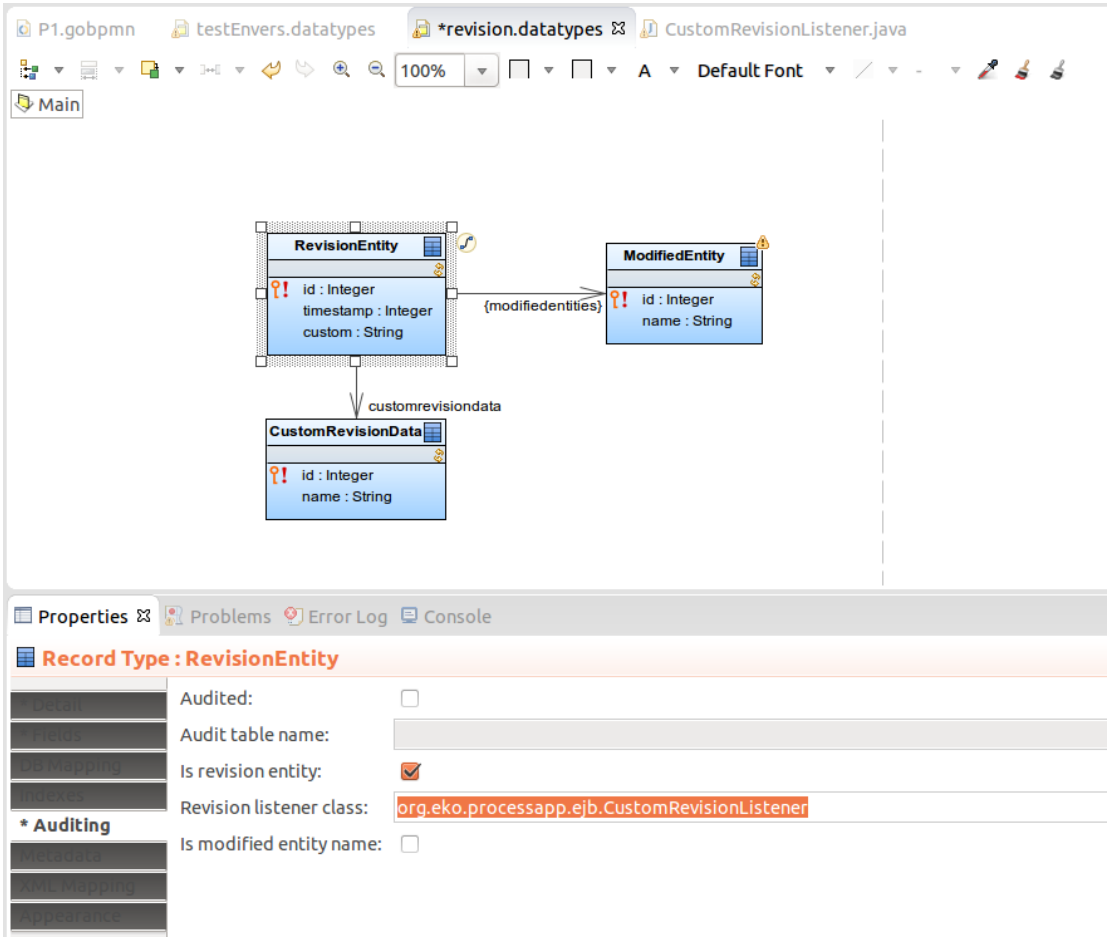


Figure 3.26 Revision Entity shared Record implemented by a custom RevisionListener class

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.

Also, you can expand it to create zip files with your modules and migration tool with database-migration scripts.

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 LSPS Application for 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.

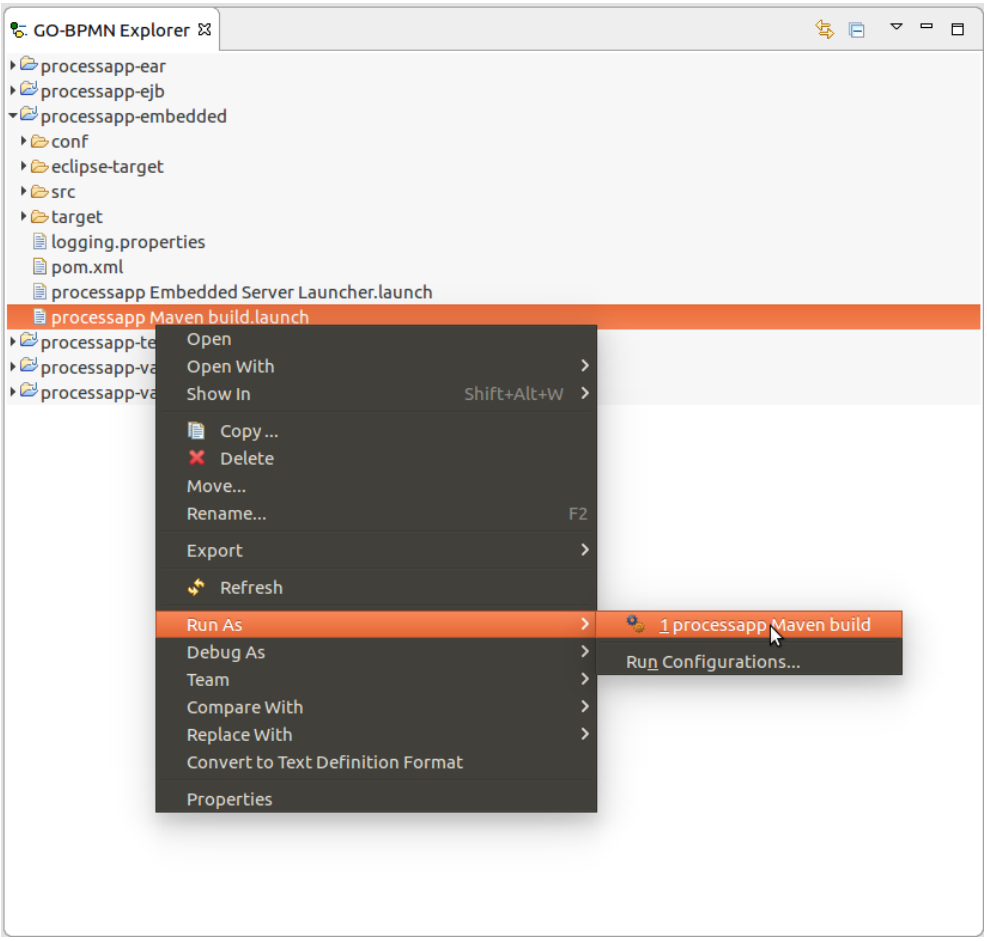
> **Important:** SDK Embedded Server is yet another server that runs locally just like Designer Embedded Server.

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.

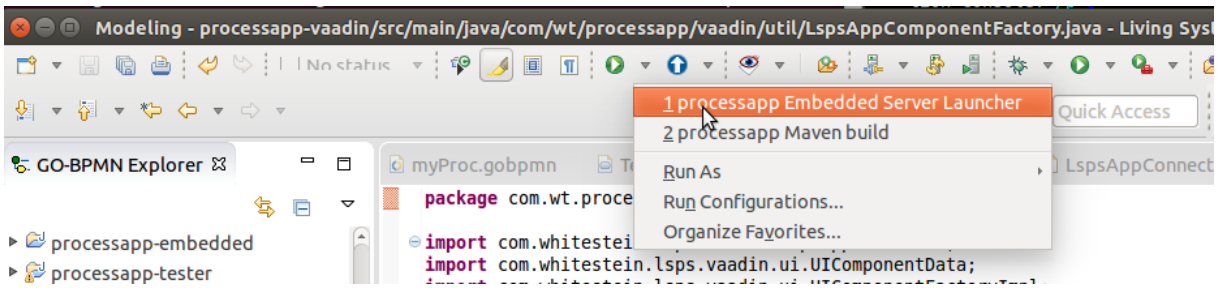


Figure 4.1 Running the application with the generated launcher

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 Adding a Module to 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>
```

4.4 Building Database Migration Tool with Migration Scripts

To generate a tool for migration of your business database as part of the build, do the following:

1. Create a Java project `<YOUR_APP>-db-migration` in your application directory and add it to modules of the root pom.xml: `mvn archetype:generate -DgroupId=<YOUR_APP_PACKAGE> -DartifactId=<YOUR_APP>-db-migration -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false`
2. Create the build pom.xml in the project:

- Set root pom.xml as parent.
- Set the group id to `com.whitestein.lsps.db-migration`, artifact Id to `<YOUR_APP>-db-migration`and packaging to `jar`.
- Define dependencies to your database driver, flyway-core, lsps-db-migration, and commons-io.
- Define the build with the resources in `src/main/scripts` and the *mainClass* property set to `<YOUR_APP_PACKAGE>.db.migration.<YOUR_APP_NAME>DbMigration`
- Add the `filters` configuration.

3. Rebuild the application and generate eclipse resources: run `mvn clean install eclipse:eclipse`.
4. Create the resources directory `mkdir -p src/main/scripts/db/migration/mysql`.
5. Import the db-migration project to your Designer workspace.
6. Create a class that extends the DBMigration class in a migration package, for example, `<YOU_APP>.db.migration` and set the schema table name.

```
import com.whitestein.lsps.dbmigration.DbMigration;
~
public class MyBusinessDataMigration extends DbMigration {
  //define the name of the schema version table:
  private static final String SCHEMA_VERSION_TABLE = "SCHEMA_VERSION_TABLE";
~
  public static void main(String[] args) {
    new MyBusinessDataMigration().migrate(args);
  }
~
  protected String getSchemaVersionTableName() {
    return SCHEMA_VERSION_TABLE;
  }
}
```

7. Add migration scripts to the resources directory, in the example `src/main/scripts/db/migration/mysql` : make sure the names of the scripts follow the convention `<DB_TYPE>_V<VERSION>_<SCRIPT_NO>__<COMMENT>.sql`, for example, `MYSQL_V1_0__Initialize.sql`.

   DB_TYPE values can be `H2`, `DB2`, `ORACLE`, `SQLSERVER`, or `MYSQL`

   When creating migration scripts for your model data, consider `generate update scripts` and modify these as required.

8. Build the project and test the migration:

```
pa/pa-db-migration$ mvn clean install
...
pa/pa-db-migration$ java -jar ./target/pa-db-migration-0.1-SNAPSHOT-full.jar \
-dbUrl jdbc:mysql://localhost:3306/lsps
```

Example pom.xml for the db-migration tool

```
<?xml version="1.0"?>
<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>sk.eko</groupId>
    <artifactId>pa</artifactId>
    <version>0.1-SNAPSHOT</version>
  </parent>
~
  <groupId>com.whitestein.lsps.db-migration</groupId>
  <artifactId>pa-db-migration</artifactId>
```

```
  <name>Pa: Database Migration</name>
  <description>Pa: Database Migration</description>
  <packaging>jar</packaging>
~
  <dependencies>
    <dependency>
      <groupId>org.flywaydb</groupId>
      <artifactId>flyway-core</artifactId>
      <version>3.2.1</version>
    </dependency>
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
    </dependency>
    <dependency>
      <groupId>com.whitestein.lsps.db-migration</groupId>
      <artifactId>lsps-db-migration</artifactId>
    </dependency>
    <dependency>
      <groupId>commons-io</groupId>
      <artifactId>commons-io</artifactId>
      <version>1.4</version>
    </dependency>
  </dependencies>
~
    <build>
    <resources>
      <resource>
        <directory>src/main/scripts</directory>
      </resource>
    </resources>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>3.2.1</version>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
            <configuration>
              <filters>
                  <filter>
                      <artifact>*:*</artifact>
                      <excludes>
                          <exclude>META-INF/*.SF</exclude>
                          <exclude>META-INF/*.DSA</exclude>
                          <exclude>META-INF/*.RSA</exclude>
                      </excludes>
                  </filter>
              </filters>
~
              <outputFile>${basedir}/target/${project.artifactId}-${project.version}-full.jar</out
              <transformers>
                <transformer implementation="org.apache.maven.plugins.shade.resource.ManifestResou
                  <mainClass>sk.eko.pa.PaDBMigration</mainClass>
                </transformer>
              </transformers>
~
            </configuration>
          </execution>
```

```
        </executions>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <configuration>
          <skipTests>true</skipTests>
        </configuration>
      </plugin>
    </plugins>
  </build>
~
  <profiles>
    <profile>
      <id>migration.tests</id>
      <activation>
        <property>
          <name>migration.tests</name>
        </property>
      </activation>
      <build>
        <plugins>
          <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <configuration>
              <skipTests>false</skipTests>
            </configuration>
          </plugin>
        </plugins>
      </build>
    </profile>
  </profiles>
~
</project>
```

4.5 Dependency Management

4.5.1 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 Designer: select the resources in GO-BPMN Explorer, right-click the selection, and click Refresh.

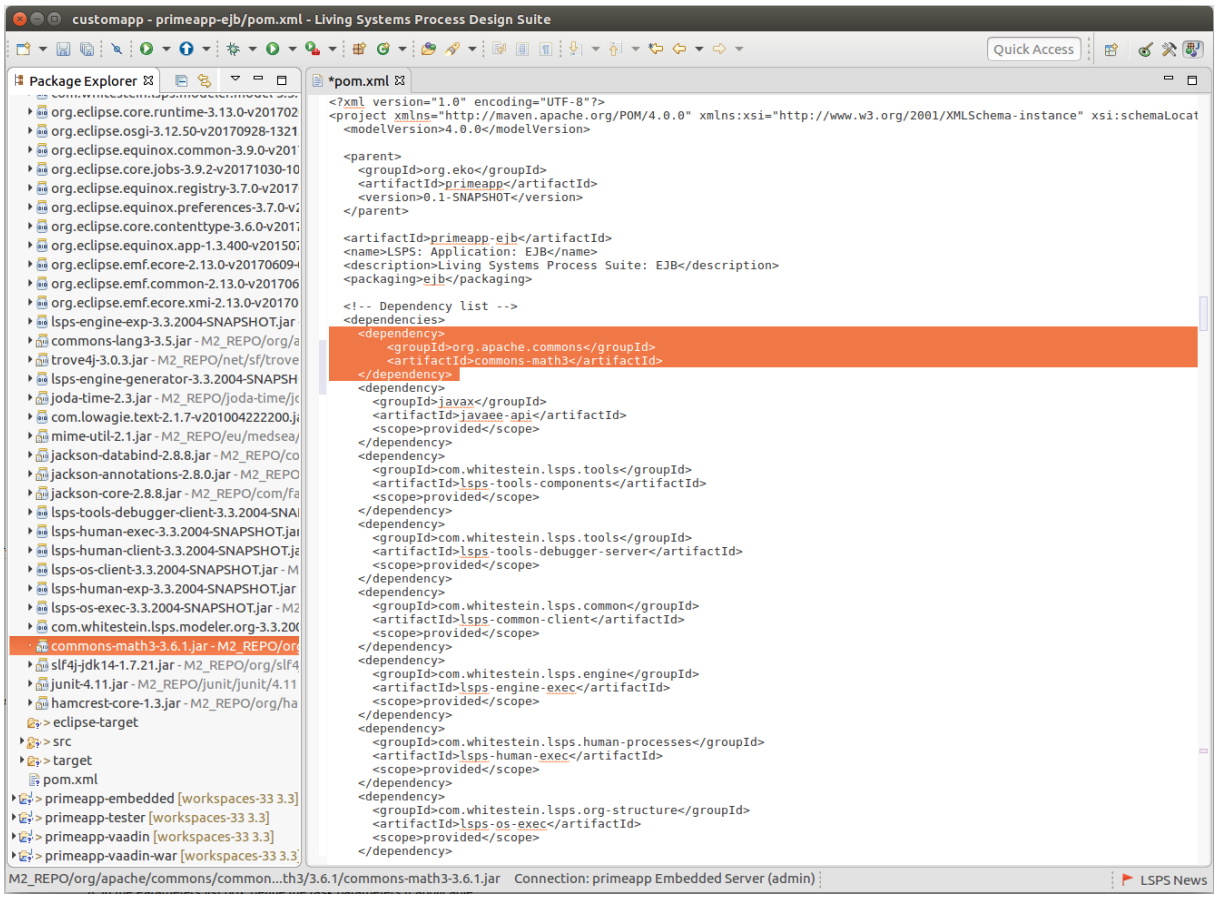


Figure 4.2 New dependency in the application pom.xml

4.5.2 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 Application* contains the testing project with sample JUnit tests.

The API is provided by the packages in *com.whitestein.lsps.test*.

The API for web tests (clicker GUI tests) is provided in the *com.whitestein.lsps.test.web* packages

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

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

5.1 JUnit Tests

The LSPS Application provides API for testing of modules as well as for clicker tests of the GUI and forms, both ui and forms.

5.1.1 Creating JUnit Tests for Modules

The generated LSPS Application comes with example JUnit tests `SampleModelIT.java`, with a test class that uploads a model, creates its instance and evaluates an expression in the context of the model instance

To create your own JUnit test, do the following:

1. Open the workspace with the source of the application.
2. In the generated LSPS Application, adjust the JUnit testing resources if applicable:
   - `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
     ```

3. Modify or create a new testing class in the `src/test/java`:
    - To establish and manage the connection to the LSPS Server, use the `LspsRemote` methods.
    - The testing classes are JUnit 4 tests and hence require annotations, such as @Before. You can access only the model context: Data from child contexts, such as Sub-Process variables, are not available.

```
public class MyTestsIT {
~
  private LspsRemote remote;
  private String moduleName;
  private File moduleFile;
  private Module module;
~
  @Before
  public void lspsRemote() {
    remote = LspsRemote.ADMIN; //create("admin", "admin", "http://localhost:8080");
    moduleName = "myTestModule";
    moduleFile = new File("src/modules").getAbsoluteFile();
~
    //Since uploadModule is an expensive operation,
    //we check if the module are uploaded already before upload:
    ModuleCriteria criteria = new ModuleCriteria();
    criteria.setNamePattern(moduleName);
    if (remote.findModules(criteria).isEmpty()) {
      remote.unloadAllModules();
      module = remote.uploadModule(moduleName, "1.0", moduleFile);
    } else {
      module = remote.getLatestModule(moduleName);
    }
  }
~
  @Test
  public void myTestNoUI() throws Exception {
    ModelInstance mi = module.startModelInstance();
    String logMessage = mi.getLogs().get(0).getDescription();
    assertTrue(logMessage.equals("My message"));
  }
}
```

5.1.2 Testing Record Values in JUnit Tests

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();
```

5.2 Creating JUnit Tests for GUI and Forms

The generated LSPS Application comes with example JUnit tests for both implementations of forms in the *tester* project:

- `SampleUIIT.java`: a sample test class that tests the UI form
- `SampleFormsIT.java`: a sample test class that test the Forms forms

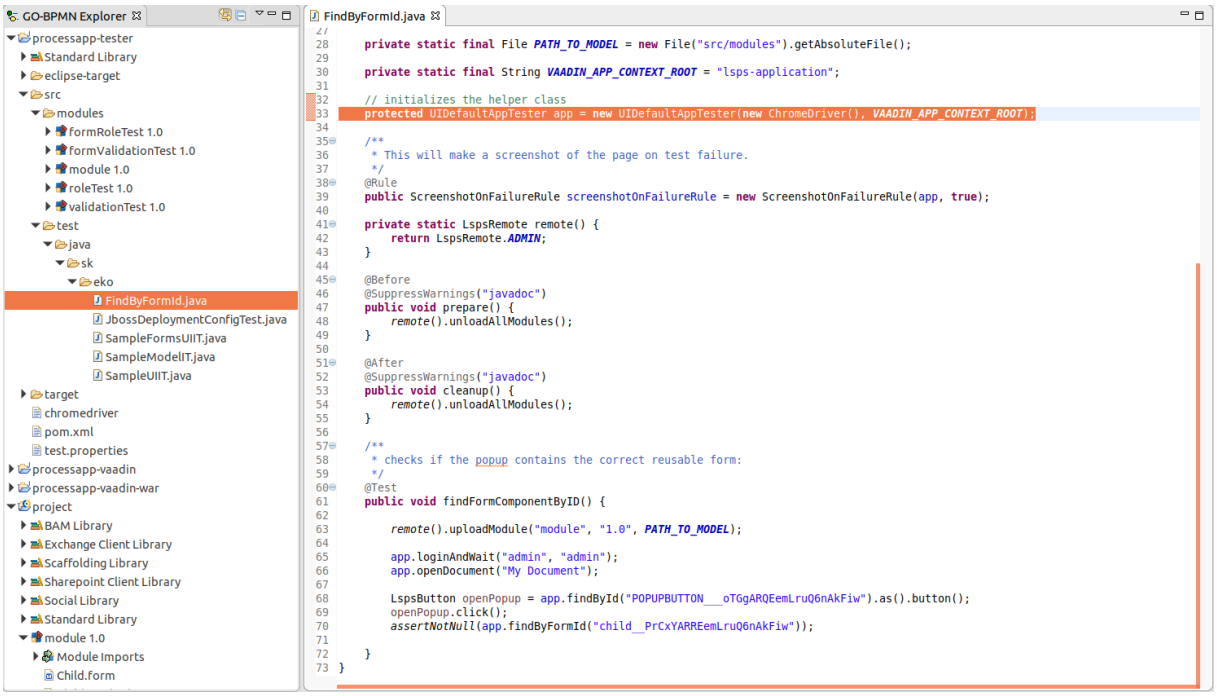
Before creating and running your clicker tests to test the GUI and forms in the LSPS Process Application, do the following:

- 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 by their modeling ID on runtime: run the LSPS Server with the `-Dcom.whitestein.lsps.vaadin.ui.debug=true` property.

  To add the property to the SDK Embedded Server, in the Java perspective of Designer, go to **Run** > **Run Configurations**; select the configuration under **Java Application** and add the property on the *Arguments* tab to *Program Arguments*).

To create your own JUnit test, do the following:

1. Download the ChromeDriver for your OS.
2. Copy the *extracted* chrome driver to `<YOUR_APP>/tester`.



   Alternatively, run your server with the `testbench_chromedriver` property with the path to the ChromeDriver.

3. Modify (*SampleUIIT* or *SampleFormsIT*) or create a testing class in the `src/test/java`.
4. Modify or create a UIDefaultAppTester member that will govern the browser: with the ChromeDriver.

```
//setting chrome driver for JUnit tests:
protected UIDefaultAppTester app = new UIDefaultAppTester(SetupUtils.createChromeDriver(),
       VAADIN_APP_CONTEXT_ROOT);

//setting firefox driver for JUnit tests:
//protected UIDefaultAppTester app = new UIDefaultAppTester(SetupUtils.createFirefoxDriver(),
       VAADIN_APP_CONTEXT_ROOT);
```

When testing in Firefox, make sure to use Firefox Portable. *not newer than version 46* (released in 2016) and run your server with the `testbench_firefox` property set to the path to your Firefox, for example, `-Dtestbench_firefox="C:\Users\John\Firefox_46\"`

- To establish and manage the connection to the LSPS Server, use the `LspsRemote` methods.
- The testing classes are JUnit 4 tests and hence require annotations, such as @Before.

5.2.1 Searching for Form Components in JUnit Tests

To get a form component, use of the following methods on the application tester (IUDefaultAppTester) to search in the entire form tree or on a component to search in the component subtree:

- `findByLspsId()` with the ID of the component you set in the ID property
- `findByFormId()` with the modeling ID of the component
- `findById()` with the HTML ID of the component: This is identical to the modeling id assuming the component is rendered only once.

Mind that the methods return a UIComponent object: to cast it to a component type, call the `as()` method; for example, `app.findByLspsId("my_vl").as().verticalLayout();`

5.2.2 Testing a Forms::Reusable Form in JUnit Tests

To work with the content inserted by a reusable form component, get the root of the tree with the `findByForm↩Id()` method on the application helper class.

```
  @Test
  public void findFormComponentByID() {
~
    remote().uploadModule("module", "1.0", PATH_TO_MODEL);
~
    app.loginAndWait("admin", "admin");
    app.openDocument("My Document");
~
    //passing the modeling id of the Reusable Form component
    //defined in the calling form:
    assertNotNull(app.findByFormId("child__PrCxYARREemLruQ6nAkFiw"));
~
  }
```

5.3 Running JUnit Tests

If you plan to run the tests on a remote server, edit the JVM parameters:

1. Go to Run > Run Configurations
2. 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.

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

1. 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`.
2. Refresh the GO-BPMN Explorer.
3. Run or connect Designer to your LSPS Server (typically, you will run the SDK Embedded Server with your application using the launcher).
4. Run the test:
   - In Designer, 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`.
   - 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.

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.

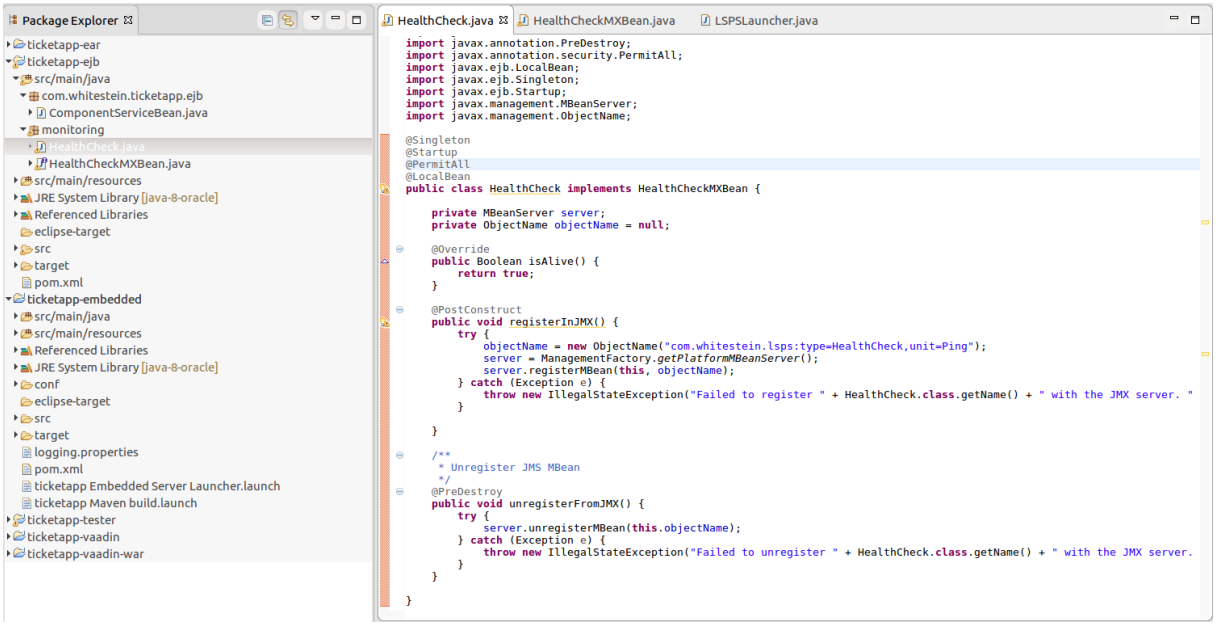


Figure 6.1 MXBean class in the LSPS Application

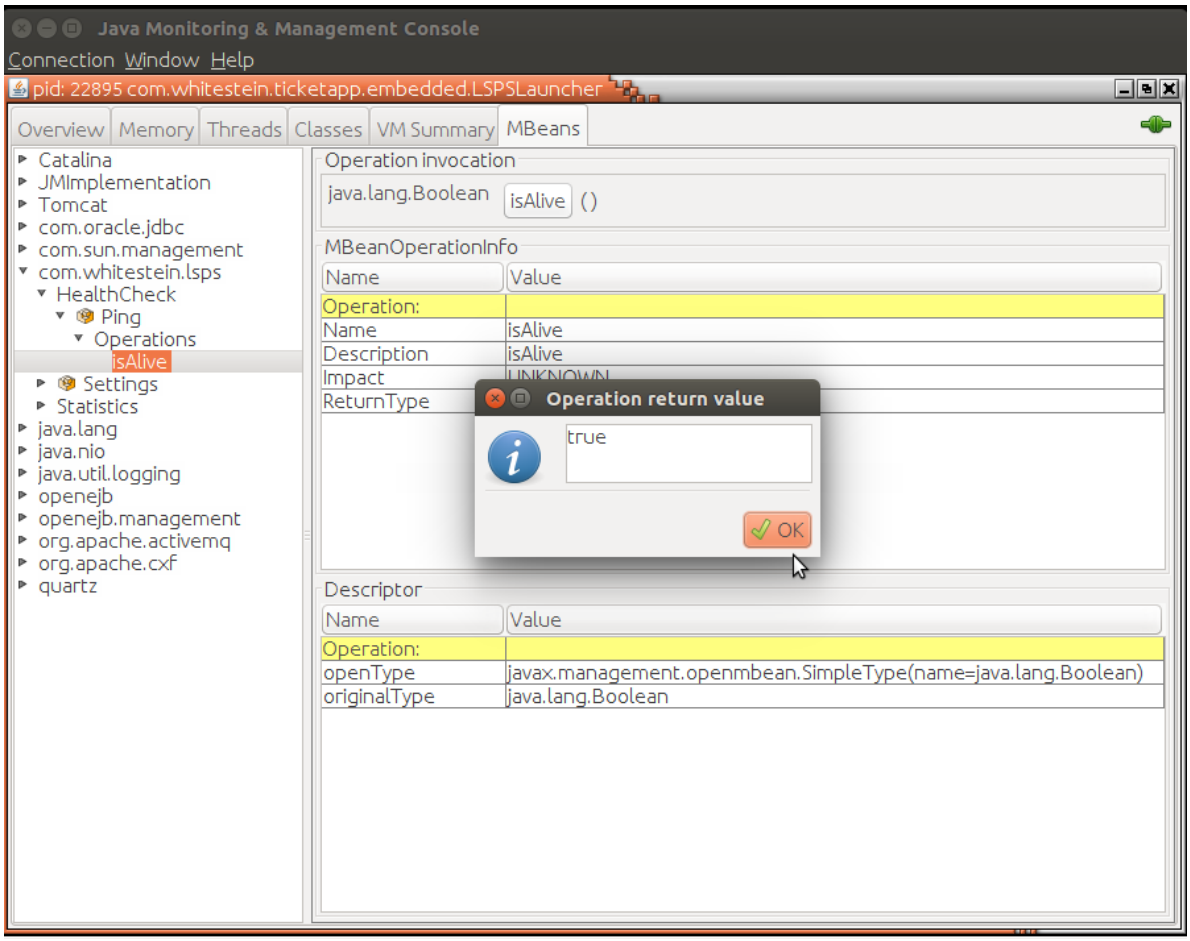


Figure 6.2 MXBean accessed from JConsole

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 Designer.
- 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 Designer`, and modify them as necessary).

The artifacts to distribute when updating modules are

- modules exported into deployable zip files

  Deployable zip files of the modules in the *<YOUR_APP>-model* projects are created as part of the maven build in the *<YOUR_APP>-model-exporter* project. `You can create them also manually`.
- model update definition files
- database-schema update scripts

7.2 Update of the Custom 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 backward-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 backward-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 Upgrade of the Patch Version

> **Important:** This upgrade procedure does not add any new features of the newer version to your application. To upgrade fully and get all the new features, follow the procedure for upgrades to minor and major versions.

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 Designer and LSPS Repository.
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 *<YOUR_APP>-embedded* project, remove the *h2* directory (this
   (d) If present, remove the `createConverterForRenderer()` method in *<-vaadin/src/main/java/sk/eko/formstestapp/c LspsFormComponentFactory.java>*. is a testing database and will be recreated on the launch of the embedded server).
3. In Window > Preferences, check the value of the `M2_REPO` classpath variable under Java > Build Path > Classpath Variables: make sure it points to the correct repository.
4. Build the application.
5. Deploy the EAR to your application server.

The artifacts to distribute:

- LSPS Application EAR

7.2.2 Preparing Upgrade of the 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 Designer with the LSPS Maven repository.
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. You can check the `release notes` for any backward-incompatible changes. Also check the summary of LSPS Application changes. in the new LSPS Application. You can check the `release notes` for any backward-incompatible changes.
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

7.2.3 Important Points of Considerations when Migrating LSPS Application 3.2 to 3.3

The LSPS Application 3.3 has been refactored:

- Classes that generally do not require modifications have been removed and are now provided as a library.
- Classes that are intended to be modified have been removed and exposed via their `App<CLASSNAME>` versions:
  - If you have previously modified the AppLayout class, you will now modify the AppAppLayout class;
  - If you have previously modified the LspsUi class, you will now modify the AppLspsUi class;
- The application uses Vaadin 8. To use the previous component implementation in Vaadin 7, do the following:
  - In `LspsUIComponentFactory` change the extended class from `UIComponentFactoryImpl` to `UIComponentFactoryV7Impl`
  - In your components, change the imported vaadin packages to its v7 version, for example, change `com.vaadin.ui.utils` to `com.vaadin.v7.ui`.

  Consider migrating your forms to the `UI over Vaadin 8`.

Unless you move the content from the original classes to the new classes, your custom LSPS Application will work as expected but will not benefit from the new features of the application.