Developer’s guide

**CONTENTS**

[1 Development tools 4](#_Toc409180969)

[1.1 Java SE Development Kit 7 4](#_Toc409180970)

[1.2 Eclipse 4](#_Toc409180971)

[1.2.1 Eclipse IDE for Java EE Developers 4](#_Toc409180972)

[1.2.2 Team Foundation Server Plug-in 4](#_Toc409180973)

[1.2.3 Checkstyle Plug-in 6](#_Toc409180974)

[1.2.4 EclEmma Java Code Coverage 6](#_Toc409180975)

[1.3 Apache Tomcat 6](#_Toc409180976)

[1.3.1 Tomcat 7 6](#_Toc409180977)

[1.3.2 PostgreSQL JDBC Driver 7](#_Toc409180978)

[1.4 Ant 7](#_Toc409180979)

[1.4.1 Download 7](#_Toc409180980)

[1.4.2 Setup 7](#_Toc409180981)

[1.4.3 Check installation 9](#_Toc409180982)

[1.5 PowerDesigner 9](#_Toc409180983)

[1.6 pgAdmin III 10](#_Toc409180984)

[1.6.1 Installation 10](#_Toc409180985)

[1.6.2 Add the pgAdmin III install directory to PATH 10](#_Toc409180986)

[2 Development environment setup 11](#_Toc409180987)

[2.1 Get the source code 14](#_Toc409180988)

[2.1.1 Configure the TFS plug-in for Eclipse 14](#_Toc409180989)

[2.1.2 Import the project into Eclipse 16](#_Toc409180990)

[2.2 Set the Checkstyle configuration in Eclipse 17](#_Toc409180991)

[2.3 Setup the development server in Eclipse 18](#_Toc409180992)

[2.4 Set the encoding of Eclipse 20](#_Toc409180993)

[2.5 Connect to the development database 21](#_Toc409180994)

[3 Development guidelines 23](#_Toc409180995)

[3.1 Model 23](#_Toc409180996)

[3.1.1 Domains 23](#_Toc409180997)

[3.1.2 Persistent model 23](#_Toc409180998)

[3.1.3 Non-persistent model 30](#_Toc409180999)

[3.2 Web application 37](#_Toc409181000)

[3.2.1 Adding a new page 37](#_Toc409181001)

[3.2.2 Adding a header to a detail page 62](#_Toc409181002)

[3.2.3 Components 63](#_Toc409181003)

[3.2.4 Adding a pop-in 75](#_Toc409181004)

[3.2.5 Adding a new menu element 80](#_Toc409181005)

[3.2.6 Adding a new role 81](#_Toc409181006)

[3.2.7 Adding a reference list 84](#_Toc409181007)

[3.3 Services 88](#_Toc409181008)

[3.3.1 Creating the interface 89](#_Toc409181009)

[3.3.2 Implementing the interface 91](#_Toc409181010)

[3.3.3 Naming conventions 93](#_Toc409181011)

[3.3.4 Error messages 93](#_Toc409181012)

[3.3.5 Miscellaneous messages 94](#_Toc409181013)

[3.3.6 Collection management (sorting, filtering) 94](#_Toc409181014)

[3.4 Dates 95](#_Toc409181015)

[3.4.1 DateUtil 95](#_Toc409181016)

[3.4.2 DateBuilder 96](#_Toc409181017)

[3.5 Data Access Objects 96](#_Toc409181018)

[3.6 Package Access Objects 97](#_Toc409181019)

[3.6.1 Description 97](#_Toc409181020)

[3.6.2 Creating a PAO object 97](#_Toc409181021)

[3.7 Testing 104](#_Toc409181022)

[3.7.1 Test classes 104](#_Toc409181023)

[3.7.2 Test data 106](#_Toc409181024)

# Development tools

## Java SE Development Kit 7

Create an oracle account and login.

Download the [Java SE Development Kit 7u60](http://www.oracle.com/technetwork/java/javase/downloads/java-archive-downloads-javase7-521261.html#jdk-7u60-oth-JPR) installer and install it (an Oracle account is required).

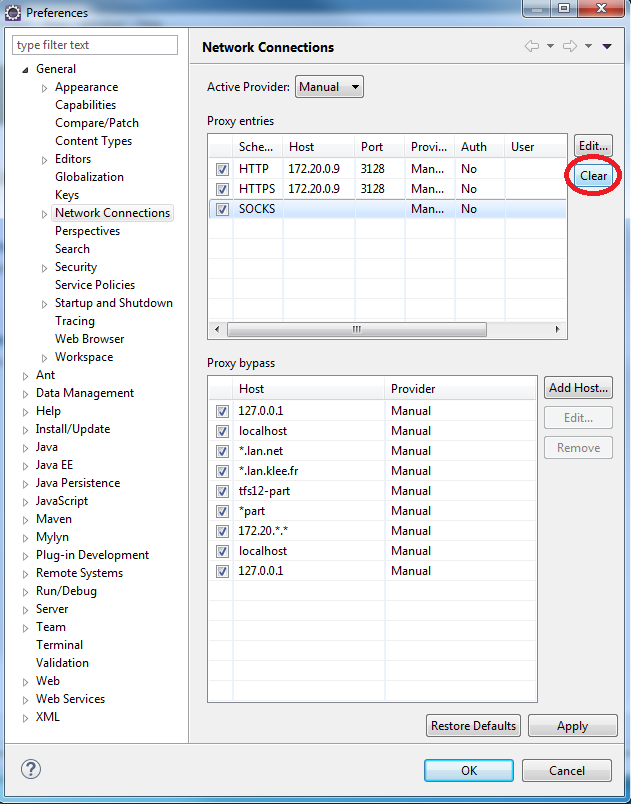
## Eclipse

### Eclipse IDE for Java EE Developers

Get the Luna R package of [Eclipse for JEE developers](https://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/lunar) and unzip it to a directory. Then, just run eclipse.exe to start it.

### Proxy Settings

1. Goto Window -> Preferences -> General -> Network Connections
2. In Proxy entries, edit rows to fit the screen bellow. The SOCKS row must have an empty host (use Clear after having selected it).
3. In Proxy bypass, edit rows to fit the screen bellow.



1. Click Ok

### Team Foundation Server Plug-in

|  |  |
| --- | --- |
| Open Eclise.  In the “Help” menu, click “Install New Software…” |  |
| In the “Install” window, click the “Add…” button. |  |
| In the “Add Repository” window, type the following information:   * Name: TFS Plugin for eclipse * Location: <http://dl.microsoft.com/eclipse/tfs> |  |
| Expand the result “Team Foundation Server Plug-in for Eclipse” and select “TFS plug-in for Eclipse”.  Click “Next” and finish the installation. |  |

### Checkstyle Plug-in

Install the plug in from [its page](http://marketplace.eclipse.org/content/checkstyle-plug#.U_26P_l_tF0) on the Eclipse marketplace.

### EclEmma Java Code Coverage

Install the plug in from [its page](http://marketplace.eclipse.org/content/eclemma-java-code-coverage#.VFstivmG9F0) on the Eclipse marketplace.

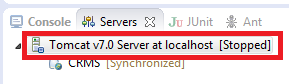
## Apache Tomcat

### Tomcat 7

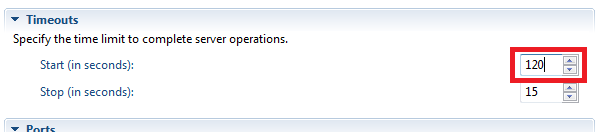
Download the [Tomcat 7.0.53 binary distribution](http://archive.apache.org/dist/tomcat/tomcat-7/v7.0.53/bin/) (apache-tomcat-7.0.53.zip).

Unzip it into a directory (D:\apache\apache-tomcat-7.0.53 for instance).

Increase the timeout : double click the server



Set the Start time to 120 s :



### PostgreSQL JDBC Driver

Download the JDBC4 [PostgreSQL Driver](http://jdbc.postgresql.org/download.html#archived) version 9.1.

Copy the downloaded .jar file to the lib folder in your Apache Tomcat directory (D:\apache\apache-tomcat-7.0.53\lib if you unzipped Tomcat in D:\apache\apache-tomcat-7.0.53).

## Ant

### Download

Get the current release of Ant at <http://ant.apache.org/bindownload.cgi>.

### Setup

|  |  |
| --- | --- |
| Extract the downloaded file into a directory (C:\Program Files\apache-ant-1.9.4 for instance). |  |
| 1. Open the system properties window. 2. Click “Environment variables”. |  |
| The “Environment variables” window appears.  In the “System variables” block, add a variable by clicking “New”. |  |
| 1. Set the variable properties:    * Name: JAVA\_HOME    * Value: the directory of the JDK version 7u60 2. Click “OK” to close the window and return to the “Environment variables” window. 3. Click “New” again in the “System variables” block. |  |
| 1. Set the variable properties:    * Name: ANT\_HOME    * Value: directory you uncompressed Ant to 2. Click “OK” to close the window and return to the “Environment variables” window. |  |
| In the “System variables”, select the variable named “Path” and click “Edit”. |  |
| 1. Add the following expression to the “Path” variable value: %ANT\_HOME%\bin;%JAVA\_HOME%\bin 2. Click “OK” to return to the “Environment variables” window. 3. Click “OK” to validate the changes. |  |

### Check installation

Open a new shell and type:

|  |
| --- |
| **ant -version** |

Type:

|  |
| --- |
| ant |

The result should be:

|  |
| --- |
| Buildfile: build.xml does not exist!  Build failed |

If this does not work ensure your environment variables are set right.

## PowerDesigner

PowerDesigner is the tool we use to edit the data model. It uses floating licenses, which means that only a restricted number of users in the company can use it at the same time. If you only need to view the data model, download PowerDesigner Viewer.

The links to download the tool are located at <http://wiki.klee.lan.net/index.php/Power_Designer>. Download the 16.5 versions.

To see who is using PowerDesigner, check this page: <http://sysam.dev.klee.lan.net/powerdesigner/cgi-bin/status.sh>.

## pgAdmin III

### Installation

Get the latest version of pgAdmin at <http://www.pgadmin.org/download/>, install it and launch it.

### Add the pgAdmin III install directory to PATH

|  |  |
| --- | --- |
| 1. Open the system properties window. 2. Click “Environment variables”. |  |
| In the “System variables”, select the variable named “Path” and click “Edit”. |  |
| 1. Add the absolute path of the pgAdmin III installation directory (the folder that contains the binary “psql.exe”) to the “Path” variable value: C:\Program Files (x86)\pgAdmin III\1.18 2. Click “OK” to return to the “Environment variables” window. 3. Click “OK” to validate the changes. |  |

### (Facultative) Resolve MSVCP120.dll error

Download and install both x86 and x64 versions of MSVC:

<http://www.microsoft.com/en-gb/download/details.aspx?id=40784>

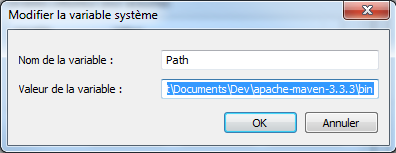
## Maven

### Install Maven

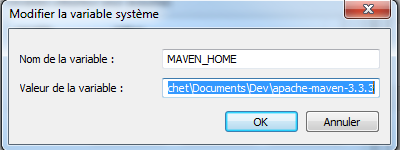
1. Copy [\\part-file01.part.klee.lan.net\PROJETS-PART\SNCF\CRMS - Refonte du SI Restauration ferroviaire\05 - Développement\Developer's guide\extra\settings.xml](file:///\\part-file01.part.klee.lan.net\PROJETS-PART\SNCF\CRMS%20-%20Refonte%20du%20SI%20Restauration%20ferroviaire\05%20-%20Développement\Developer's%20guide\extra\settings.xml%20) file to “C:\Users\username\.m2\ settings.xml”
2. Download and install Maven 3.3.3

<https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html>

1. Add the maven environment variable “C:\Users\username\Documents\Dev\apache-maven-3.3.3\bin” to the end of the “Path” variable.



1. Add the maven environment variable « C:\Users\username\Documents\Dev\apache-maven-3.3.3» to the « MVN\_HOME » variable.



### Check Delivery files (optional)

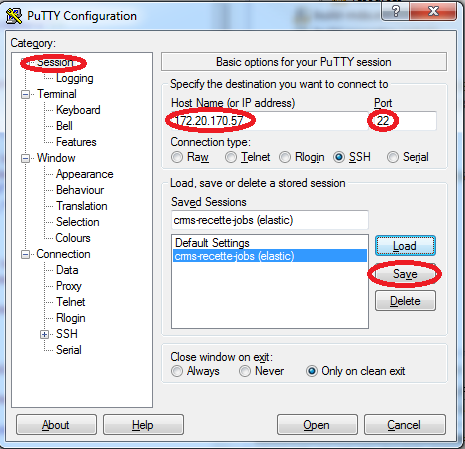
1. Open a console in "D:\Projets\CRMS\workspace\CRMS\” and run “start package.bat”
2. Wait until maven downloads all libraries and compiles (20 min)
3. Check if files « prosper-3.0.0.tar.gz », « prosper-postgresql-3.0.0.tar.gz » and « rso\_app\_jbo\_v3.0.0.tar.gz » exist in the directory “D:\Projets\CRMS\workspace\CRMS\target\delivery-prosper-3.0.0\“

## Additional software

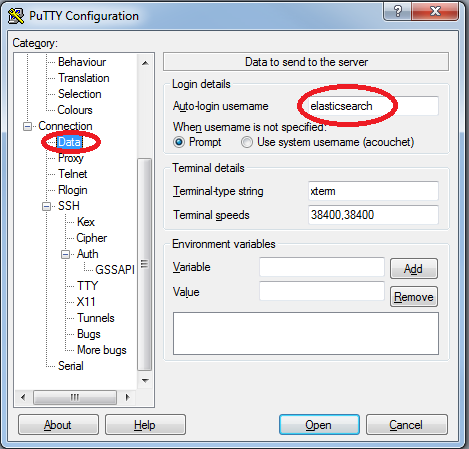
### Putty

Install Putty here [putty-0.65-installer.exe](http://the.earth.li/%7Esgtatham/putty/latest/x86/putty-0.65-installer.exe)

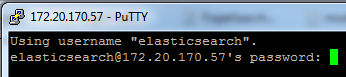
Configure all servers in  [\\part-file01.part.klee.lan.net\PROJETS-PART\SNCF\CRMS - Refonte du SI Restauration ferroviaire\05 - Développement\CRMS\_Machines\_and\_users.docx](%20\\part-file01.part.klee.lan.net\PROJETS-PART\SNCF\CRMS%20-%20Refonte%20du%20SI%20Restauration%20ferroviaire\05%20-%20Développement\CRMS_Machines_and_users.docx)



Step 1 : Create a host



Step 2 : Add the username



Step 3 : Login with the credentials in CRMS\_Machines\_and\_users.docx

Step 4 : Configure all other servers in CRMS\_Machines\_and\_users.docx

### Winscp

Install and install Winscp here [https://winscp.net/eng/download.php#download2](https://winscp.net/eng/download.php%23download2)

# Development environment setup

## Get the source code

### Configure the TFS plug-in for Eclipse

|  |  |
| --- | --- |
| Open Eclipse.  In the menu Window > Show View, click “Other”.  The “Show View” window appears. Select the “Team Foundation Server” -> “Team Explorer” view and click “OK” to show it. |  |
| In the “Team Explorer” window, click the “Connect to Team Projects” button.  The “Add existing Team Project” window appears. |  |
| In the “Add Existing Team Project”, click the “Servers” button”.  The “Add/Remove Team Foundation Server” window appears. |  |
| In the “Add/Remove Team Foundation Server”, click the “Add” button.  The “Add Team Foundation Server” appears. |  |
| In The “Add Team Foundation Server”, set the URL of the server to:  <http://tfs12-part.part.klee.lan.net:8080/tfs>  Click “OK” to add the server, and then close the “Add/Remove Team Foundation Server” window to get back to the “Add Existing Team Project” window. |  |
| In the “Add Existing Team Project” window:   1. In the “Team Project Collections” panel, select “PGC”. 2. In the “Team Projects” panel, select “SNCF\_CRMS”. 3. Click “Finish”. |  |

### Import the project into Eclipse

|  |  |
| --- | --- |
| Open Eclipse.  Open the “Team Explorer View” and click “Source Control Explorer”. |  |
| In the “Source Control Explorer”, expand SNCF\_CRMS > Main, right click the “CRMS” folder and select “Import”.  The “Import Projects from Team Foundation Server” window appears. |  |
| In the “Import Projects from Team Foundation Server” window:   1. Select SNCF\_CRMS > Main > CRMS. 2. Click “Next”. |  |
| In the confirmation Window, click “Finish”. The download of the sources will begin.  When the download is complete, the CSRM project appears in the Eclipse project explorer. |  |

## Set the Checkstyle configuration in Eclipse

|  |  |
| --- | --- |
| Open Eclipse.  In the “Project Explorer” view, right click the project and select “Properties”. |  |
| In the “Properties” window, select the “Checkstyle” item in the left panel.  In the “Checkstyle” panel (on the right), in the “Main” tab:   1. Enable Checkstyle (parameter “Checkstyle active for this project”). 2. Enable simple configuration (parameter “Use simple configuration”). 3. In the “Simple group”, select the check configuration named “CRMS”).   Validate the configuration by clicking “OK”. |  |

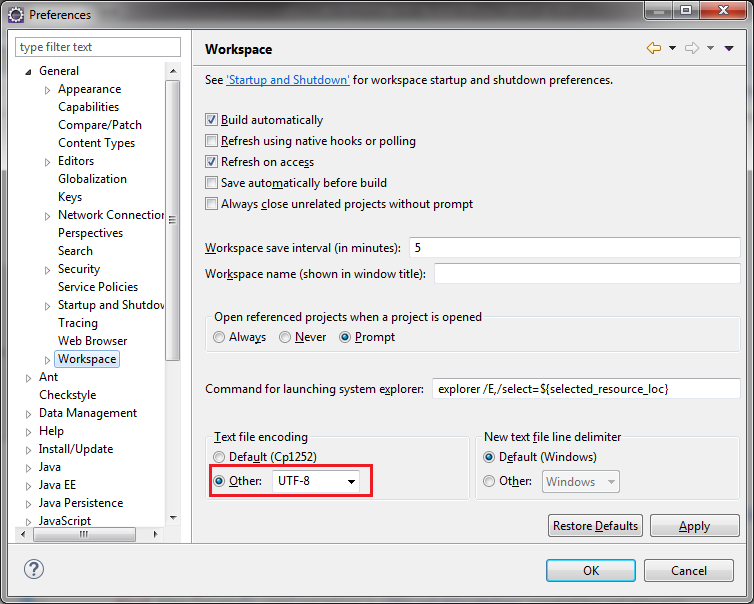
## Setup the development server in Eclipse

|  |  |
| --- | --- |
| Open Eclipse.  Display the “Servers” view  Click the link “No servers available. Click this link to create a new server…”  The “New server” window appears. |  |
| Select “Tomcat v7.0 Server” in the “Apache” folder.  Optionally, define a server name.  Click “Next”. |  |
| Specify the Tomcat installation directory: it is the path of the folder where Tomcat was unzipped to (at step 1.3.1).  Leave the value of the JRE to “Workbench default JRE”.  Click “Next”. |  |
| Add the application from the left list (“Available”) to the right (“Configured”) by selecting it and clicking “Add”.  Click “Finish”. |  |
| The configured server now appears in the “Servers” view. |  |

## Set the encoding of Eclipse

Open Eclipse, and go to Window > Preferences.

In the “Preferences” window, go to General > Workspace. Then, in the “Text encoding” panel, choose “Other” and select “UTF-8”.



## Connect to the development database

Open pgAdmin III.

|  |  |
| --- | --- |
| Click the button “Add a server connection”.  The “New Server Registration” window appears. |  |
| In the “New Server Registration” window, type the following information:   * Name: CRMS-DEV-SQL * Host: CRMS-DEV-SQL.part.klee.lan.net * Port: 5432 * Username: postgres * Password: kleeklee   Click the “OK” button.  The server will now appear in the object browser. |  |

# Development guidelines

## Model

The model is defined in the **\*.oom** file, located in folder **src/main/bdd/modele**.

It is a PowerDesigner Object Oriented model. It contains the definitions of all the persistent entities of the application, and it is used (along with other files, \*.ksp files) to generate the Java classes manipulated in the code as well as the SQL code to create the database structure.

A class defined in the PowerDesigner model will then correspond to a table in the data base, and to a Java class.

### Domains

Editing the model consists in defining classes (tables) and their attributes (columns). Each column in the database has a type, but even though some columns have the same type, they are used to store data of different nature.

For instance, one column may represent a positive amount of money while another one may contain signed (positive or negative) amounts of money. The type of these two columns is NUMERIC, they are stored the same way, but in the case of positive amounts, only positive values should be allowed, while in the case of signed amounts, positive and negative values are allowed, and it would be nice to display their sign in the application.

So there is a difference between the type used to store data and the rules to apply to this data. A domain combines these two aspects: it specifies a storage type (integer, numeric, string, etc.) and rules to apply to the data (formatting, value constraints, etc.).

Domains are defined in two places:

* In the PowerDesigner Object Oriented Model.
* In the file “domain.ksp” in the Java project.

#### Domains in PowerDesigner

TODO

#### Domains in “domain.ksp”

TODO

### Persistent model

The persistent model represents all the Java entities that correspond to database objects:

* A class corresponds to a table.
* A Java attribute corresponds to a column.

The editing of tables and columns in the database (and at the same time the corresponding class/attribute) begins in the PowerDesigner Object Oriented model.

#### Packages

The PowerDesigner model is divided in packages which contain the classes. Each package represents a functional module of the application.

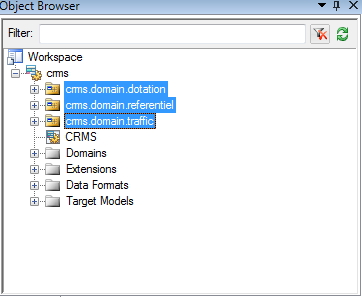


Figure 1 : Three packages in a PowerDesigner Object-Oriented model (OOM)

The name of each package must be formatted as follows:

|  |
| --- |
| <Application alias>.domain.<Functional module name> |

The package code will be computed automatically by PowerDesigner by replacing dots by underscore characters. Do not change it.

|  |
| --- |
| <Application alias>\_domain\_<Functional module name> |

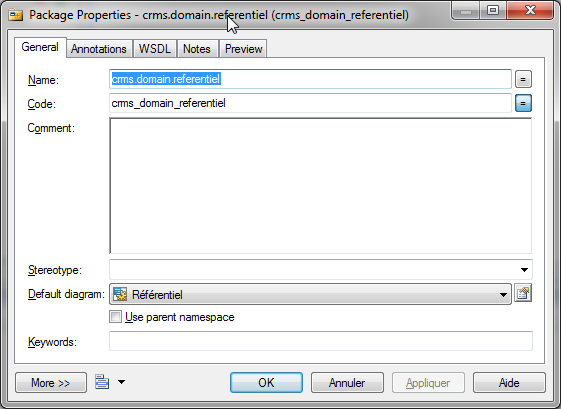


Figure 2 : Name and code of an OOM package

The name of a package defines the folder structure where the Java files will be created by the class generator.

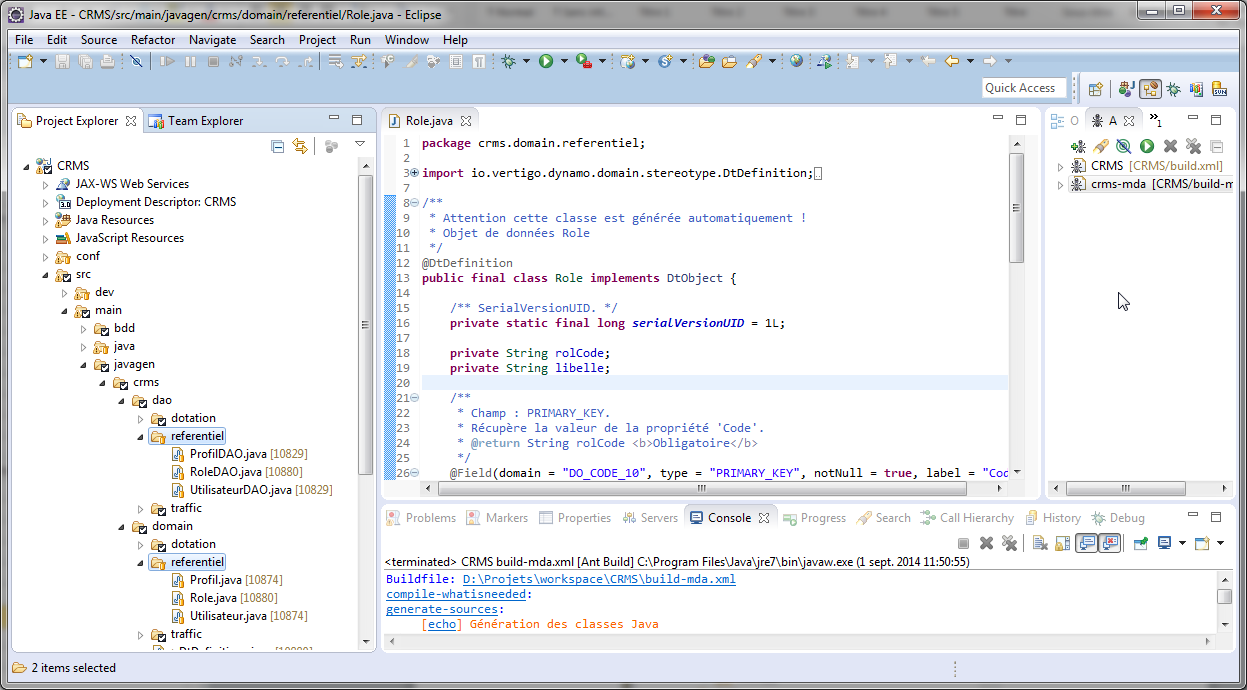


Figure 3 : Folders corresponding to a PowerDesigner package

#### Classes

A class declared in PowerDesigner will correspond to a Java class and to a table in the database. When defining a class, specify the following properties.

##### General properties

* Name: a user friendly name that describes what the class represents.
* Code: a string used to generate the name of the corresponding Java class and the corresponding table in the database. Use capital letters and separate words by underscore characters ‘\_’.

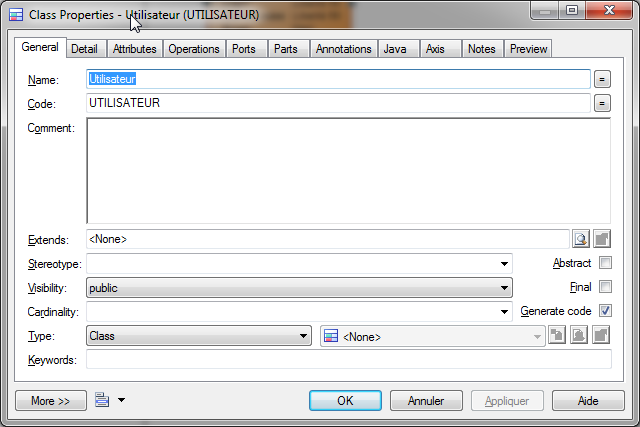


Figure 4 : PowerDesigner class general properties

##### Detail properties

Persistent: make sure that this property is checked.

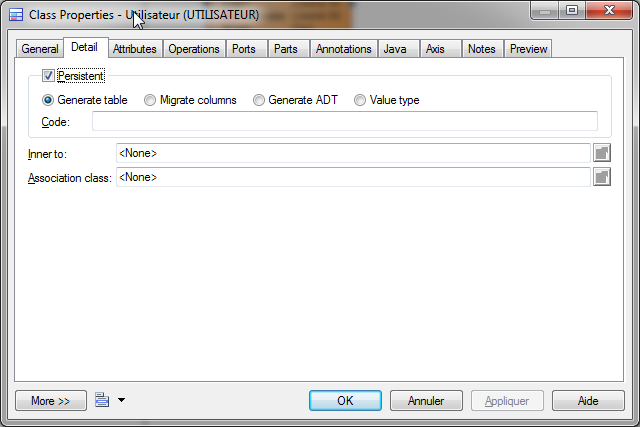


Figure 5 : PowerDesigner class detail properties

##### Attributes

In the “Attributes” tab, add a line for each property of the class. A property will correspond to a Java attribute along with its getter and setter methods, and to a column in the database table corresponding to the edited PowerDesigner class. When adding an attribute to class, specify the following attributes:

* Name: the user-friendly label of the attribute.
* Code: a code used to generate the name of the corresponding Java attribute and the name of the corresponding column in the database. Use capital letters for it. If the attribute is the primary key of the class, its code should be <Table trigram>\_[ID|CODE]. The table trigram consists in three letters uniquely identifying the table. By convention, use ID if the attribute is an integer identifier and use CODE if the attribute is a string identifier.
* PI (for “Primary Identifier”): check this property if the attribute is the primary identifier of the class. If checked, make sure that the attribute’s code’s format is correct (see the previous item). There must be one and only one attribute with the “PI” property checked by class, as it will correspond to the primary key of its class’s corresponding table in the database.
* Mandatory: if the value of this property cannot be null, check this property.
* Domain: the domain indicates what type of data the attribute will store, and what constraints it should meet (limited range, regular expression, etc.). Domains are also defined in PowerDesigner by the user according to the project requirements (see Domains).
* Data Type: The Data Type is automatically defined when selecting the domain of the attribute; you should normally not have to modify it.

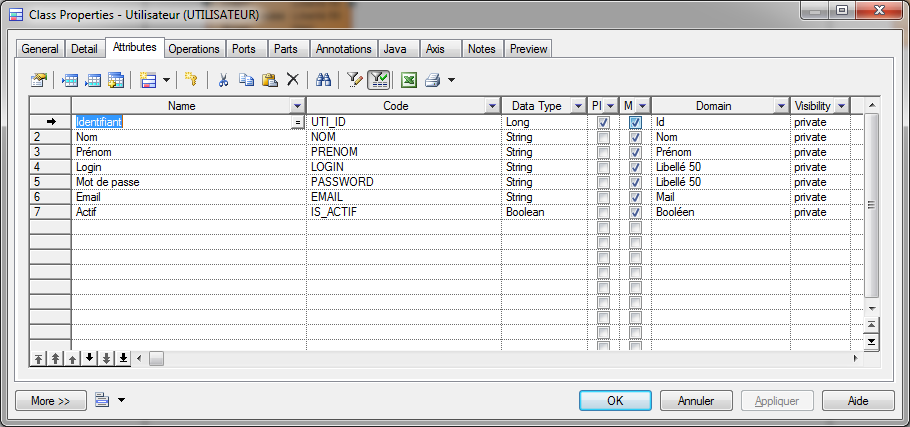


Figure 6 : PowerDesigner class attributes

#### Associations

Many objects refer to other objects. These relationships are modeled by associations in UML and in PowerDesigner, and by foreign keys in SQL.

To add a relationship between two classes:

1. Click the “Association” button in PowerDesigner
2. Click the referring class, maintain the click.
3. Drag and drop the cursor to the referred class. A link appears between the two classes as you release the mouse button.
4. Double-click the link to specify its properties

##### General properties

* Name: a user-friendly name for the association.
* Code: the code used to generate the name of foreign key constraint corresponding to the association. Use capital letters and separate words by underscore characters.

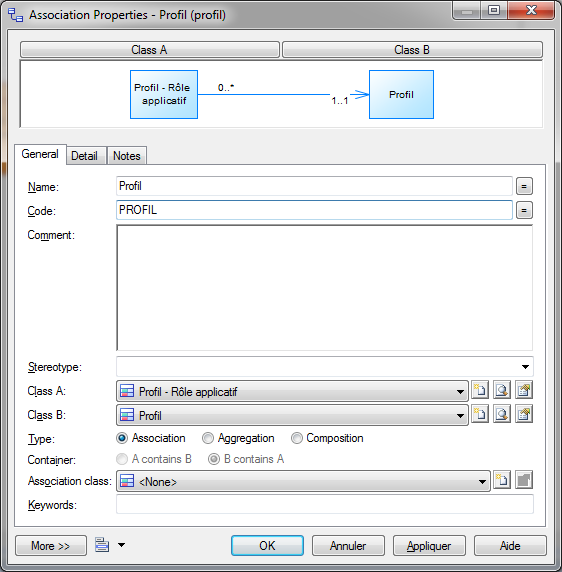


Figure 7 : PowerDesigner association general properties

##### Detail properties

TODO

#### Generating Java classes and SQL scripts

TODO

### Non-persistent model

#### Description

The non-persistent model represents all the Java entities that DO NOT correspond to database objects. You may have to define non-persistent objects every time the persistent objects are not enough for what you want to do. For example, when you need to:

* Load a list of results from the database where each result contains data from several tables (this happens frequently in search pages).
* Code a form whose fields will be dispatched in different tables.

#### Creating a non-persistent class

Non-persistent classes are declared in text-files (whose extension is .ksp). These KSP files are then used by the model generator that will create the Java classes according to the definitions in the KSP files.

So, here are the steps to create a non-persistent class:

1. Find the “model.ksp” file corresponding to the concerned functional module. Create it if it does not exist and reference it in the “generation.kpr” file (otherwise it will not be considered by the model generator). Add the class definition in the “model.ksp” file.
2. Call the model generator.

##### Model.ksp

Non persistent classes are defined in text files named “model.ksp”.

###### Location

There should be a “model.ksp” file for every functional module of the application, located in the directory **/src/main/resources/<project alias>/services/<functional module name>**.

For instance, the CRMS project contains a page to search users that is located in the module “Référentiel”. The non-persistent criteria class used to implement this page should be declared in the file **/src/main/resources/crms/services/referentiel/model.ksp**.

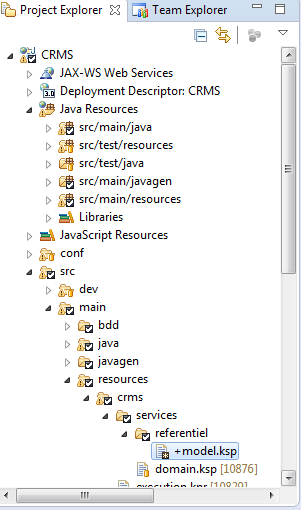


Figure 8 : Location of the model.ksp file for the module "Référentiel"

###### Content

Package declaration

The first line of the “model.ksp” file should be the declaration of the package where the Java classes will be generated. The syntax of this definition is identical to the Java package definition syntax.

In the current example, the first line of the “model.ksp” file located in /src/main/resources/crms/services/referentiel/ is:

|  |
| --- |
| package crms.domain.referentiel |

Class definition

A class definition specifies:

* The name of the class,
* Whether the class is persistent or not,
* The fields of the class.

The definition of classes in the “model.ksp” file is done using a specific syntax:

|  |
| --- |
| create DtDefinition DT\_<CLASS\_NAME> {  persistent: "<true|false>",    <field 1>,  <field 2>,  ...,  <field N>,  }, |

Here is an example of a class definition to generate a Java class named UtilisateurCriteria, containing the following attributes:

* nom,
* prenom,
* isActif,
* proId

|  |
| --- |
| create DtDefinition DT\_UTILISATEUR\_CRITERIA {  persistent: "false",    field NOM {domain: DO\_NOM, label: "Nom", notNull: "false"},  field PRENOM {domain: DO\_PRENOM, label: "Prénom", notNull: "false"},  field IS\_ACTIF {domain: DO\_BOOLEEN, label: "Actif", notNull: "true"},  field PRO\_ID {domain: DO\_ID, label: "Profil", notNull: "false"},  }, |

Figure 9 : Generated Java non-persistent class illustrates the Java class that will be generated according to the example definition.

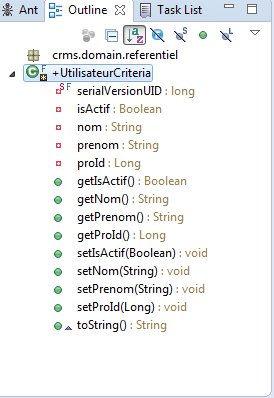


Figure 9 : Generated Java non-persistent class

Name

The name of a class must begin by the prefix “DT\_”. Use capital letters and separate terms with the underscore character ‘\_’. What follows the prefix will be used to name the generated Java class.

In the current example, the class named DT\_UTILISATEUR\_CRITERIA in the KSP file will generate a Java class named “UtilisateurCriteria”.

Persistent

If this property is set to “true”, a SQL statement will be generated to create a table corresponding to this class at the Class generation time. As we are creating non persistent objects, set this property to “false”.

Fields

A field is determined by:

* Its name,
* Its domain,
* Its label,
* Whether it is persistent or not.

The syntax of a field declaration is:

|  |
| --- |
| field <Name> {domain: <Domain>, label: "<Label>", notNull: "<true|false>"} |

Example:

|  |
| --- |
| field PRENOM\_UTILISATEUR {domain: DO\_PRENOM, label: "Prénom", notNull: "false"}, |

Name

Use capital letters and separate terms with the underscore character ‘\_’. The name will be used to name the generated Java attribute.

In the current example, the class named PRENOM\_UTILISATEUR in the KSP file will generate a Java attribute named “prenomUtilisateur”.

Domain

The domain of a field should be one the domains defined in “domain.ksp” (see section Domains in “domain.ksp”)

If the field should be an instance of another class named “OtherClass”. The domain should be:

|  |
| --- |
| DO\_DT\_<OTHER\_CLASS>\_DTO |

If the field should be a list of instances of another class “OtherClass”, the domain should be:

|  |
| --- |
| DO\_DT\_<OTHER\_CLASS>\_DTC |

Label

The value of this property will be used as the default display label of the field in JSP files.

Not null

If the field should not be equal to null, set this property to “true”.

##### Reference in “generation.kpr”

Each “model.ksp” should be referenced in the “generation.kpr” file of the project otherwise it will not be taken into account and the Java classes defined in it will not be generated by the model generator.

The file “generation.kpr” is located in the directory **/src/main/resources/<project alias>/**. Make sure it contains the relative path of the “model.ksp” file containing the definitions of your non-persistent Java classes.

In our example, “generation.kpr” should contain the following line:

|  |
| --- |
| services/referentiel/model.ksp |

##### Class generation

To generate the classes defined in the “model.ksp” file, run the Ant file named “build-mda.xml”.

After the build completes, the Java classes appear in the folder corresponding to the package indicated in the “model.ksp” file.

In our example, the class UtilisateurCriteria will be created in folder src/main/javagen/crms/domain/referentiel.

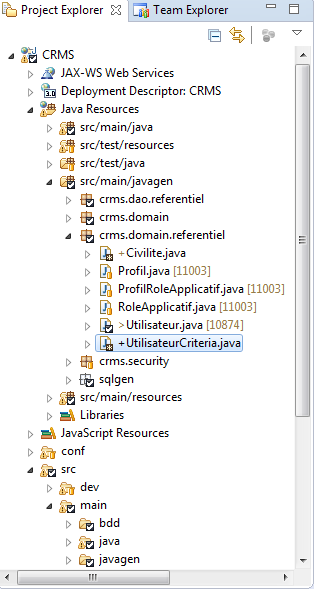


Figure 10 : Location of a generated non-persistent Java class

## Web application

### Adding a new page

#### Principle

There are three main steps in adding a new page:

1. Create a new JSP (Java Server Page) file where the structure of the page will be defined according to the requirements. This file declares which elements (input fields, tables, labels, buttons, etc.) should appear and which actions should be executed in the page.
2. Create the class that will perform the actions of the page, such as loading the data to be displayed, create/update/delete an object, and any other action specified in the requirements of the page. This Java class will be referred to as the “action class” of the page.
3. Map each action of the JSP page to a method of the action class. This mapping is done declaratively in an XML file named struts.xml.

#### Examples

##### List page

This example will demonstrate how to create a new list page from scratch. The page will contain:

* A table listing items (in this case, user profiles) loaded from the database, each row will represent a user profile. The first cell of each row of the table will contain a hyperlink redirecting the user on the detail page of the clicked profile (if the user has the appropriate permission).
* A button that redirects the user to a page where they can create a new user profile.

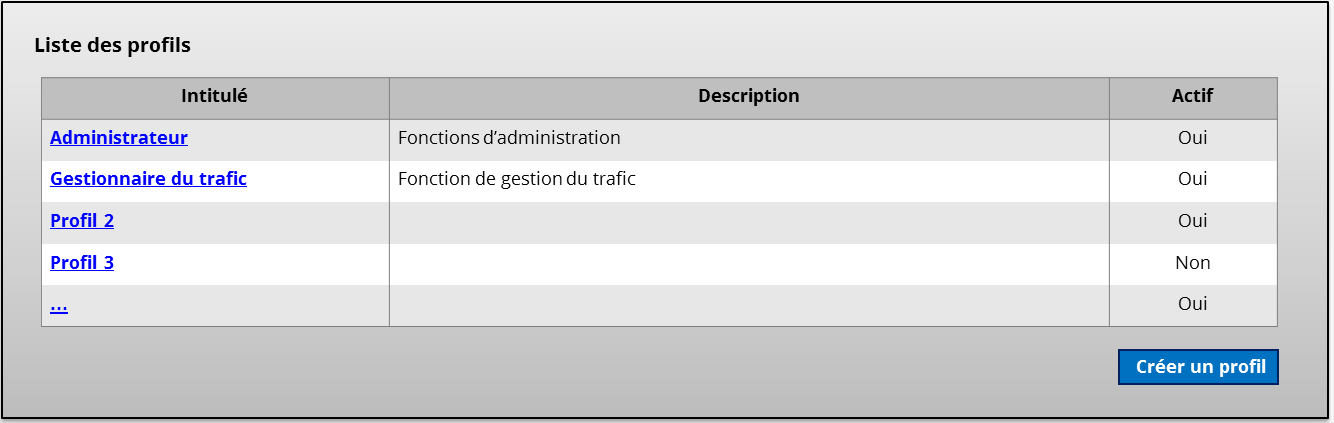


Figure 11 : Mockup of the example list page

###### Creating the JSP file

Location

The first thing to do is to add the new JSP file at the right place. All JSP files should be located in a subfolder of the directory **src/main/webapp/jsp/**. Most of the time, the appropriate subfolder corresponds to the functional module that contains the page, use the user requirements to determine its name.

In our example, the list of user profiles is located in the module called “Référentiel”, so the JSP file is added to the folder **src/main/webapp/jsp/referentiel**.

By convention, the name of a list page respects the following format:

|  |
| --- |
| <type of the listed items>List.jsp |

Here we are creating a list of profiles (“profil” in French), so the JSP page will be named profilList.jsp.

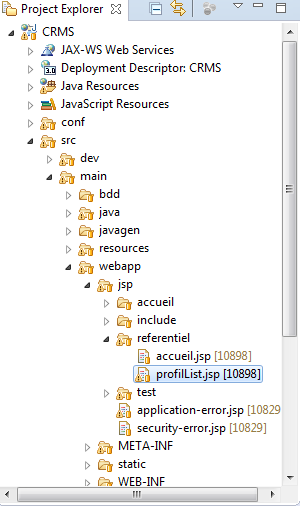


Figure 12 : Location of the JSP file for the profile list page

Contents

Once the JSP file has been created, the next step is to write the page structure inside of it. The JSP file content is composed of four main parts:

1. The declaration of the language of the page and the import of tag libraries.
2. The inclusion of the page header.
3. The use of tags to describe the actual structure of the page.
4. The inclusion of the page footer.



Figure 13 : Structure of the JSP file for a list page

The structure of the page is actually defined in section 3. It basically contains a <s:form> element, and inside of it:

* A table declaration, with at least one column.
* A button bar (with buttons and/or links to other pages for instance).

Table declaration

The list is defined using the “table” tag of the Display Tag Library (which is used here with the prefix “display”, as it can be seen in the import section at the top of the JSP file):

|  |
| --- |
| <display:table name=*"profils"* id=*"profil"*>  ... Columns declaration  </display:table> |

The table element should declare at least these attributes:

* Name: the name of the context item containing the list of objects to be displayed here. This context item will be an attribute of the action class, which means that in our case our action class will declare a context item identified as “profils” (see section Context objects).
* Id: the name that will represent an item in the list. It will be used to access the properties of each item in the columns declaration (see section Columns declaration below).

Columns declaration

Each column of a table is defined by a “column” element (still prefixed with “display” in our example) inside the parent “table” element.

Basic columns

The simplest type of columns is the one where an attribute of the item is mapped to the column. The cells in this column will simply display the value of this attribute. To do this, set the “property” attribute of the column element to the name of the mapped Java attribute. Here, the table list instances of the Java class “Profil” and our column should display the value of their attribute named “description”:

|  |
| --- |
| <display:column property=*"description"* title=*"Description"*></display:column> |

Complex columns

A column can contain more complicated data, such as a hyperlink to another page with parameters depending on the current row.

In the current example, the column must contain a hyperlink to access the detail of a profile. Here is the declaration of the column containing the hyperlink:

|  |
| --- |
| <display:column title=*"Intitulé"*>  <s:url action=*"ProfilDetail"* var=*"profilDetailURL"*>  <s:param name=*"proId"*>${profil.proId}</s:param>  </s:url>  <a href=*"*${profilDetailURL}*"*>${profil.intitule}</a>  </display:column> |

Inside the <display:column> tag:

* An URL is declared and identified as “profilDetailURL”; it contains a parameter named “proId” whose value will be the value of the property “proId” of “profil”. “profil” corresponds to the object mapped to the row.
* The URL identified as “profilDetailURL” is then used to set the href attribute of the <a> element contained in the column. The text of the hyperlink is the value of the property “intitule” of the item corresponding to the row.

Button bar

The button bar simply consists in a div containing the links and buttons. In our example it looks like this:

|  |
| --- |
| <div class=*"button-bar"*>  <div class=*"right"*>  <s:if test=*"* *boutonCreationVisible"*>  <s:a action=*"ProfilDetail"*>Créer un profil</s:a>  </s:if>  </div>  </div> |

* The first div element represents the button bar itself.
* The second div element will be used to position the link on the right side of the page.
* The <s:if> tag is used to test a condition. If this condition is satisfied, the inner elements of this tag will be rendered in the page displayed to the user. The test here consists in evaluating the value of the context object “boutonCreationVisible” which will be equal to true if the user has the role “Création de profil” and false otherwise. This context object will be defined in the action class of the JSP page (described in section Creating the action class).
* The <s:a> tag is used to render a <a> HTML tag that will redirect the user to the JSP page that will be mapped to the action “ProfilDetail” (described in section Mapping actions to methods of the action class).

###### Creating the action class

The action class is a Java class that extends AbstractActionSupport, or maybe there is an abstract action class specific to your project (for the project CRMS, the action classes should implement AbstractCRMSActionSupport).

Location

The action classes are located in a Java package named **<application alias>.controller.<functional module>** inside the directory **/src/main/java/**.

In our example, the application alias is “crms” and the page listing the profiles is in the module named “Référentiel” (according to the user requirements), so the action class is added to the package crms.controller.referentiel.

The name of the action class should respect the following naming convention:

|  |
| --- |
| <Type of the listed items>ListAction.java |

Here we are creating a list of profiles (“profil” in French), so the action class will be named ProfilListAction.java.

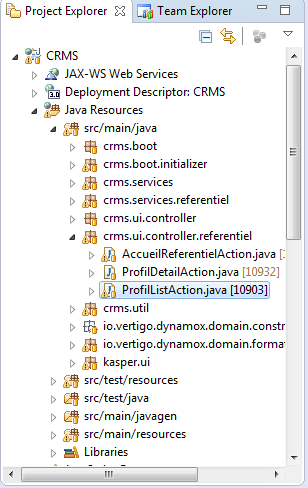


Figure 14 : Location of the action class for the profile list page

Contents

The role of the action class of a list page consists in:

1. Call a service to load the list to display in the page.
2. Storing this list in the context of the page, so that it can be accessed in the JSP page.
3. Performing any action that should be possible in the page according to the user requirements.

Attributes

Service references

As the class will load the list by calling the appropriate method of a service, it needs a reference to this service. The class may contain as many service references as necessary.

In our example, the list of profile is loaded using the method loadProfilList, declared in interface ProfilReadServices, so the service reference attribute of our class ProfilListAction will look like this:

|  |
| --- |
| @Inject  **private** ProfilReadServices profilReadServices; |

* The type of this attribute should be the interface (and NOT the class that implements it) that declares the method that will be actually invoked to load the list.
* The @Inject annotation has to be added to the attribute. It will result in the injection of the interface’s implementation at runtime and allow the proper working of the page.

Context objects

The list loaded by the action class will have to be stored in the context of the page. This is done thanks an attribute of type ContextList<T> (where T is the type of items of the list). In our example:

|  |
| --- |
| **private** **final** ContextList<Profil> profilsRef = **new** ContextList<>("profils", **this**); |

* The first parameter of the ContextList constructor is the context key; this key will be used in the JSP page to refer to the list of items.
* The second parameter is always the current instance of the action class.

We also need a context object to store the Boolean value indicating whether the button “Créer un profil” should be visible or not. The class ContextReft<T> is used to store a single object.

|  |
| --- |
| **private** **final** ContextRef<Boolean> boutonCreationVisibleRef = **new** ContextRef<>("boutonCreationVisible", Boolean.**class**, **this**); |

* The first argument is the context key that will be used in the JSP page to refer to the object.
* The second argument is the class of the stored object.
* The last argument is the current instance of the action class.

Methods

initContext()

This method is the mandatory implementation of the abstract method initContext() defined in the action class parent AbstractActionSupport. It should contain the logic that loads the list of profiles and stores it in the page context, and the code that checks if the button “Créer un profil” should be displayed and store the result in the context.

Here is the implementation of initContext() in our example:

|  |
| --- |
| **protected** **void** initContext() **throws** KSecurityException {  boutonCreationVisibleRef.set(hasRole(Role.***R\_REF\_PRF\_CREA***));  profilsRef.publish(profilServices.loadProfilList());  } |

* The button “Créer un profil” is visible if the current user has the role R\_REF\_PRF\_CREA. This is determined by calling the method hasRole() defined in the class AbstractCRMSActionSupport. The result of this call is then stored in the context object by calling the method set() of the class ContextRef.
* The list of profiles is loaded by calling the method loadProfilList() of the service reference. The result is then stored in the context by calling publish() of the class ContextList.

###### Mapping actions to methods of the action class

In the case of a simple page list, the only action to map is the one that redirects the user to the page list. It is done in the **struts.xml** file.

Instructions

In the file struts.xml, add an <action> element inside the <package> element. Specify the following attributes of the action element:

* Name: an expression matching the action used in JSP files. It can be an exact match, or it can contain the wildcard character ‘\*’ to match different actions using the same pattern. By convention, the format of the name is \*<Type of the listed items>List.
* Class: the fully qualified name of the action class corresponding to the page list to reach.
* Method: by convention, set this property to the wildcard value ‘{1}’.

Then, add a <result> element inside the newly added <action> element:

* Set its name attribute to “none”.
* Set its inner value to the path to the JSP file.

Example

In our case, the mapping for the redirection to the profile list would be the following:

|  |
| --- |
| <action  name=*"\*ProfilList"*  class=*"crms.ui.controller.referentiel.ProfilListAction"*  method=*"{1}"*>  <result name=*"none"*>/jsp/referentiel/profilList.jsp</result>  </action> |

Now, to add a link to the profile list in any JSP file, you can use the following code:

|  |
| --- |
| <s:a action=*"ProfilList"*>Liste des profils</s:a> |

This code will be rendered as the following HTML:

|  |
| --- |
| <a href="/crms/ProfilList.do">Liste des profils</a> |

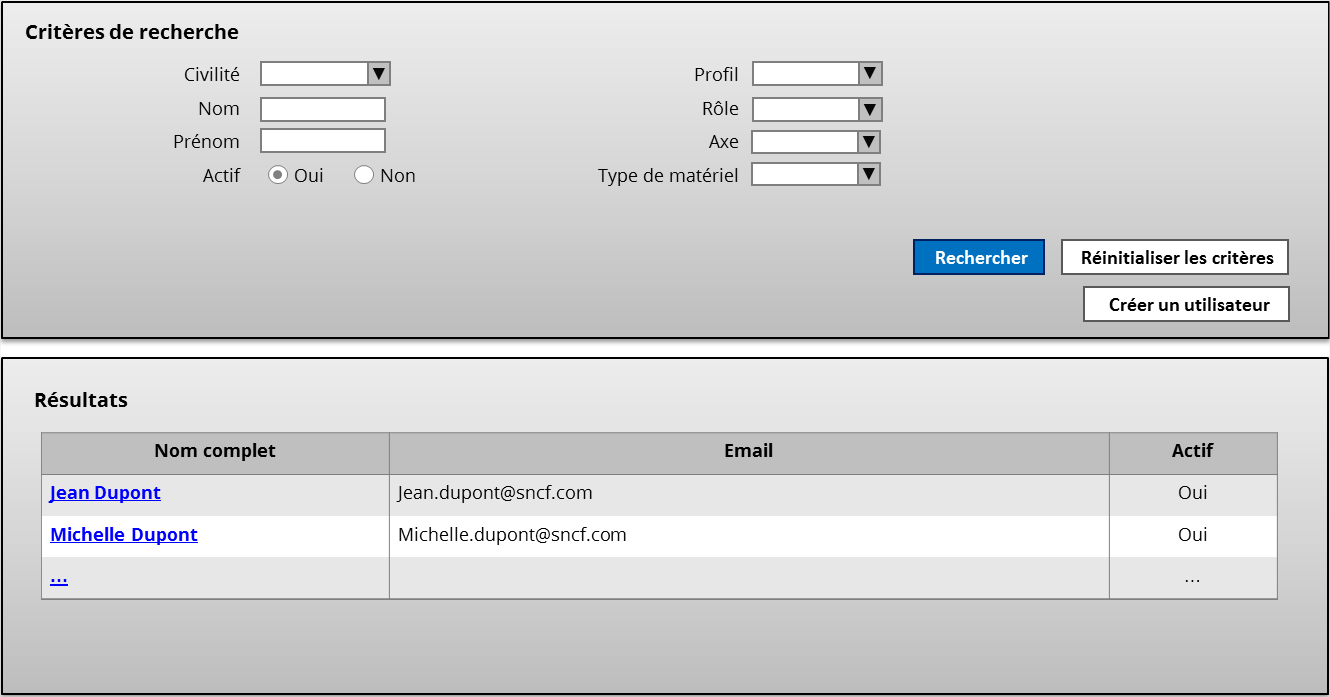
Then, when the user clicks the hyperlink, the steps below will be executed:

1. Struts will extract the action name “ProfilList” from the URL “/crms/ProfilList.do”.
2. Using the struts.xml file, the application will determine that the action name “ProfilList” matches the pattern “\*ProfilList”, so :
   1. The action class to instantiate to handle the request is ProfilListAction.
   2. To find out which method of the ProfilListAction class should be invoked, the application extracts the part of the action name that matches the wildcard character ‘\*’. In this example, the matching part of the action name is the empty string, so there is no method to invoke in the ProfilListAction, only its method initContext will be run, and that is enough because this is where the list is loaded.
   3. As no method has been invoked apart from initContext(), there is no result (i.e. the result is “none”). According to the <result> element in the struts.xml file, the user must be redirected to the proflList.jsp page.

##### Search page

This example will demonstrate how to create new search page from scratch. A typical search page has two blocks:

* A form that contains:
  + The various available criteria.
  + Buttons to: perform the search and reset the criteria
  + Optionally, a link that redirects the user to the item creation page.
* One result block that contains the results of the search.



###### Creating the JSP file

Location

The first thing to do is to add the new JSP file at the right place. All JSP files should be located in a subfolder of the directory **src/main/webapp/jsp/**. Most of the time, the appropriate subfolder corresponds to the functional module that contains the page, use the user requirements to determine its name. In our example, the list of user profiles is located in the module called “Référentiel”, so the JSP file is added to the folder **src/main/webapp/jsp/referentiel**.

By convention, the names of search pages respect the following format:

|  |
| --- |
| <type of the searched items>Search.jsp |

Here we are creating a page to search users (“utilisateur” in French), so the JSP page will be named utilisateurSearch.jsp.

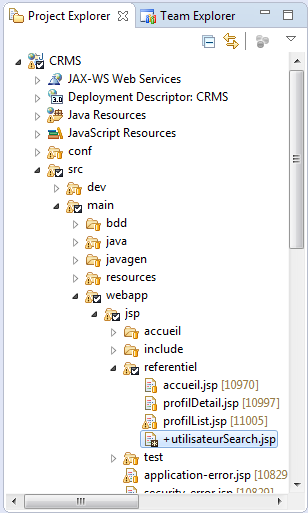


Figure 15 : Location of the JSP file for the user search page

Contents

Once the JSP file has been created, the next step is to write the page structure inside of it. The JSP file content is composed of four main parts:

1. The declaration of the language of the page and the import of tag libraries.
2. The inclusion of the page header.
3. The use of tags to describe the actual structure of the page.
4. The inclusion of the page footer.

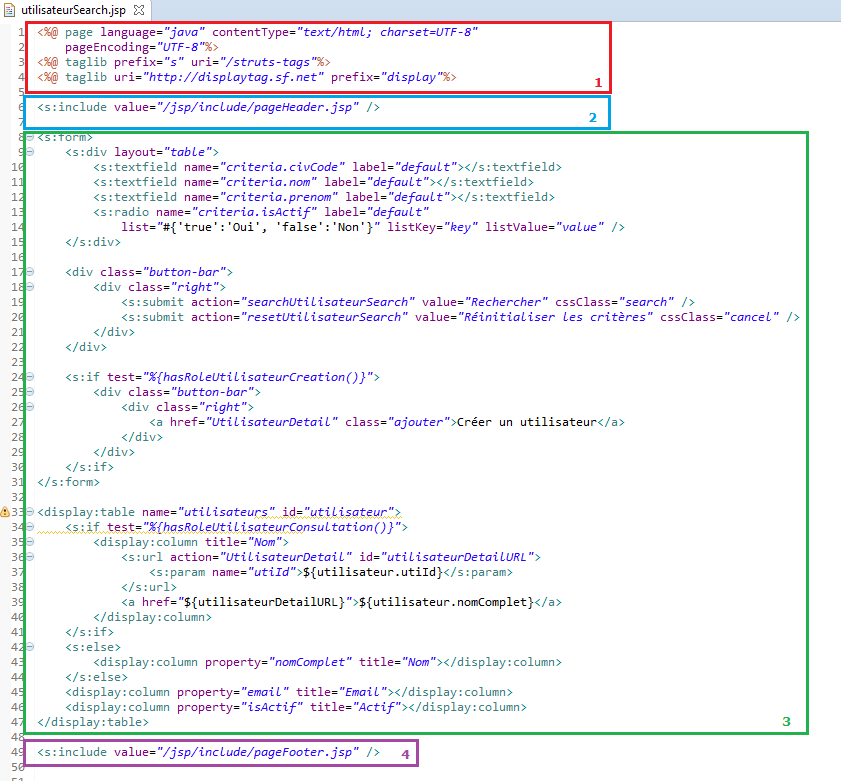


Figure 16 : Structure of the JSP file for a search page

Sections 1, 2 and 4 are the same in every JSP file. What changes is section 3, which contains the actual structure of the page. This section basically contains:

* A <s:form> element representing the block of criteria.
* A <display:table> element representing the list of results.

Form

The <s:form> element of a search page contains:

* The input fields corresponding to all the criteria of the page. They are grouped in a <s:div> element.
* The buttons of the page.

Criteria

The criteria are contained in a <s:div> element. Each criterion is defined using an input tag (textfield, checkbox, select, etc.) which will be mapped to an attribute of a Java object.

|  |
| --- |
| <s:div layout=*"table"*>  <s:textfield name=*"criteria.nom"* label=*"default"*></s:textfield>  <s:textfield name=*"criteria.prenom"* label=*"default"*></s:textfield>  <s:radio name=*"criteria.isActif"* label=*"default"*  list=*"*#{'true':'Oui', 'false':'Non'}*"* listKey=*"key"* listValue=*"value"* />  </s:div> |

Each element inside the <s:div> element represents a criterion. The values of these criteria will be stored in the corresponding attributes of a Java object referred to as “criteria” here.

1. The first criterion is a text field mapped to the attribute “nom” of the object “criteria”.
2. The second one is also a text field and is mapped to the attribute “prenom”.
3. The last one is a radio-button mapped to the attribute “isActif”.

Buttons

A search page allows the user to:

* Perform the search (button).
* Reset the criteria (button).
* Be redirected to the page where they can create a new item of the same type as the searched objects (link).

The buttons and links should be contained in a button bar, which is simply a div with the appropriate class (in our example, the class is “button-bar”, it depends on the project):

|  |
| --- |
| <div class=*"button-bar"*>  <div class=*"right"*>  <s:submit action=*"searchUtilisateurSearch"* value=*"Rechercher"* cssClass=*"search"* type=*"button"* />  <s:submit action=*"resetUtilisateurSearch"* value=*"Réinitialiser les critères"* cssClass=*"cancel"* type=*"button"* />  </div>  </div> |

* The first div element contains a second one, whose class is “right”. This will be used to position the link on the right side of the page thanks to CSS.
* Each <s:submit type=”button”> element is used to render a button that will perform the action specified in its action attribute:
  + The value of the action attribute should be set to match the method to be executed in the Java action class corresponding to the search JSP page. By convention, its value should respect the format: <methodName><Type of the items searched in the page>Search (where methodName is “search” for the search button and “reset” for the reset button).

###### Creating the action class

The action class is a Java class that extends AbstractActionSupport (from the io.vertigo.struts2.core package), or maybe there is an abstract action class specific to your project (for the project CRMS, the action classes should implement AbstractCRMSActionSupport).

Location

The action classes are located in a Java package named **<application alias>.controller.<functional module>** inside the directory **/src/main/java/**.

In our example, the application alias is “crms” and the page to search users is in the module named “Référentiel” (according to the user requirements), so the action class is added to the package crms.controller.referentiel.

The name of the action class should respect the following naming convention:

|  |
| --- |
| <Type of the searched items>SearchAction.java |

Here, the action class will be named UtilisateurSearchAction.

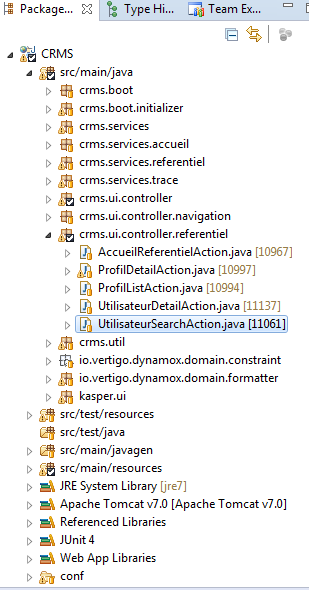


Figure 17 : Location of the action class for the user search page

Contents

The role of the action class of a search page consists in:

1. Initializing the criteria (an instance of a Java class) and putting it in the context of the page, so that it can be accessed in the JSP page.
2. Performing at least these two actions:
   1. Execute the search and put the result list in the context of the page.
   2. Reset the criteria.

Attributes

Criteria key

Search pages should memorize the criteria that were used to perform the last search. To do this, it needs a unique key (a string value) that will used to store the criteria in the session.

A solution is to declare a static final string attribute in the action class and set its value to the name of the action class.

For example, in the action class of the user search page (named UtilisateurSearchAction.java), the search criteria key is defined as follows:

|  |
| --- |
| **private** **static** **final** String ***SEARCH\_CRITERIA\_KEY*** = UtilisateurSearchAction.**class**.getName(); |

Service references

The search consists in a call to a service, so the action class needs a reference to the service.

In our example, the users are searched using the method loadUtilisateurResultListByCriteria, declared in interface UtilisateurServices, so the service reference attribute of our class UtilisateurSearchAction will look like this:

|  |
| --- |
| @Inject  **private** UtilisateurServices utilisateurService; |

* The type of this attribute should be the interface (and NOT the class that implements it) that declares the method that will be actually invoked to search users.
* The @Inject annotation has to be added to the attribute. It will result in the injection of the interface’s implementation at runtime and allow the proper working of the page.

Context objects

Usual search pages need two context objects:

* One to store the criteria.
* A second one to store the result list.

In the current example:

|  |
| --- |
| **private** **final** ContextForm<UtilisateurCriteria> criteriaRef = **new** ContextForm<>("criteria", **this**);  **private** **final** ContextList<UtilisateurResult> utilisateursRef = **new** ContextList<>("utilisateurs", **this**); |

* The generic class ContextForm<T> is used to store a single instance of a Java class. The first parameter of its constructor is the context key that will be used in the JSP page to reference the item; the second parameter is always the current instance of the action class.
* The generic class ContextList<T> is used to store a list of instances of a Java class. . The first parameter of its constructor is the context key that will be used in the JSP page to reference the list; the second parameter is always the current instance of the action class.

Methods

initContext()

This method is the mandatory implementation of the abstract method initContext() defined in the action class parent AbstractActionSupport. It should contain the logic that initializes the criteria and stores it in the page context.

Initializing the criteria consists in creating a new instance of the Java class representing the criteria. However, if the user has already accessed the page and performed a search during their session, the criteria used for their last search has been saved in the user session by the Search method and should be restored.

Here is the implementation of initContext() in our example:

|  |
| --- |
| @Override  **public** **void** initContext() {  **final** DtObject criteria = getSession().getSearchCriteria(***SEARCH\_CRITERIA\_KEY***);  **if** (criteria != **null**) {  **final** UtilisateurCriteria utilisateurCriteria = (UtilisateurCriteria) criteria;  criteriaRef.publish(utilisateurCriteria);  } **else** {  criteriaRef.publish(**new** UtilisateurCriteria());  }  toModeEdit();  } |

* The first thing to do is to find out if there is a criteria in the session by calling getSearchCriteria() and passing the Criteria key to it. If a criteria object was saved in the user session during a previous search, the returned object is not null and can be published in the criteria context object. Otherwise, create a new instance of the appropriate Java class, set its default values and publish it in the criteria context object.
* Switch to edit mode by calling toModeEdit(), otherwise the criteria will not be editable.

Search method

The action class must contain the method that will:

1. Read the criteria typed by the user.
2. Store the criteria in the user session.
3. Call the search method.
4. Store the results of the search in the page context.
5. Return a NONE (a constant which is equal to “none”). This result is discussed in section Mapping actions to methods of the action class.

|  |
| --- |
| **public** String search() {  **final** UtilisateurCriteria criteria = criteriaRef.readDto();  getSession().putSearchCriteria(***SEARCH\_CRITERIA\_KEY***, criteria);  **final** DtList<UtilisateurResult> utilisateurs  = utilisateurService.loadUtilisateurResultListByCriteria(criteria);  **if** (utilisateurs.isEmpty()) {  getUiMessageStack().warning(  "La recherche ne retourne aucun résultat.");  }  utilisateursRef.publish(utilisateurs);  **return** ***NONE***;  } |

Reset criteria method

The reset method typically:

1. Creates a new instance of the criteria class.
2. Stores it in the user session.
3. Stores the new instance in the page context.
4. Returns NONE (a constant which is equal to “none”). This result is discussed in section Mapping actions to methods of the action class.

|  |
| --- |
| **public** String reset() {  **final** UtilisateurCriteria criteria = **new** UtilisateurCriteria();  getSession().putSearchCriteria(***SEARCH\_CRITERIA\_KEY***, criteria);  criteriaRef.publish(criteria);  **return** ***NONE***;  } |

###### Mapping actions to methods of the action class

In the case of a simple page list, the only action to map is the one that redirects the user to the page list. It is done in the **struts.xml** file.

Instructions

In the file struts.xml, add an <action> element inside the <package> element. Specify the following attributes of the action element:

* Name: an expression matching the action used in JSP files. It can be an exact match, or it can contain the wildcard character ‘\*’ to match different actions using the same pattern. By convention, the format of the name is \*<Type of the searched items>Search.
* Class: the fully qualified name of the action class corresponding to the search page to reach.
* Method: by convention, set this property to the wildcard value ‘{1}’.

Then, add a <result> element inside the newly added <action> element:

* Set its name attribute to “none”.
* Set its inner value to the path to the JSP file.

Example

In our case, the mapping for the redirection to the user search page would be the following:

|  |
| --- |
| <action name=*"\*UtilisateurSearch"*  class=*"crms.ui.controller.referentiel.UtilisateurSearchAction"*  method=*"{1}"*>  <result name=*"none"*>/jsp/referentiel/utilisateurSearch.jsp</result>  </action> |

The JSP file utilisateurSearch.jsp contains the button to perform the search:

|  |
| --- |
| <s:submit action=*"searchUtilisateurSearch"* value=*"Rechercher"* type=*"button"* /> |

When the user clicks the button, the steps below will be executed:

1. Struts will extract the action name “searchUtilisateurSearch”.
2. Using the struts.xml file, the application will determine that the action name “searchUtilisateurSearch” matches the pattern “\*UtilisateurSearch”, so :
   1. The action class to instantiate to handle the request is UtilisateurSearchAction.
   2. To find out which method of the UtilisateurSearchAction class should be invoked, the application extracts the part of the action name that matches the wildcard character ‘\*’. In this example, the matching part of the action name is “search”, so the method to invoke in the UtilisateurSearchAction is search() (described in section Search method).
   3. The search method returns the String value “none”. According to the <result> element in the struts.xml file, the user must be redirected to the utilisateurSearch.jsp page. That is why the user remains on the search page after clicking the search button.

##### Detail page

This example will demonstrate how to create a new detail page from scratch. The page will display the detail of a profile, and will contain:

* A form used to display or edit the profile properties.
* An editable list used to select the roles of the profile.

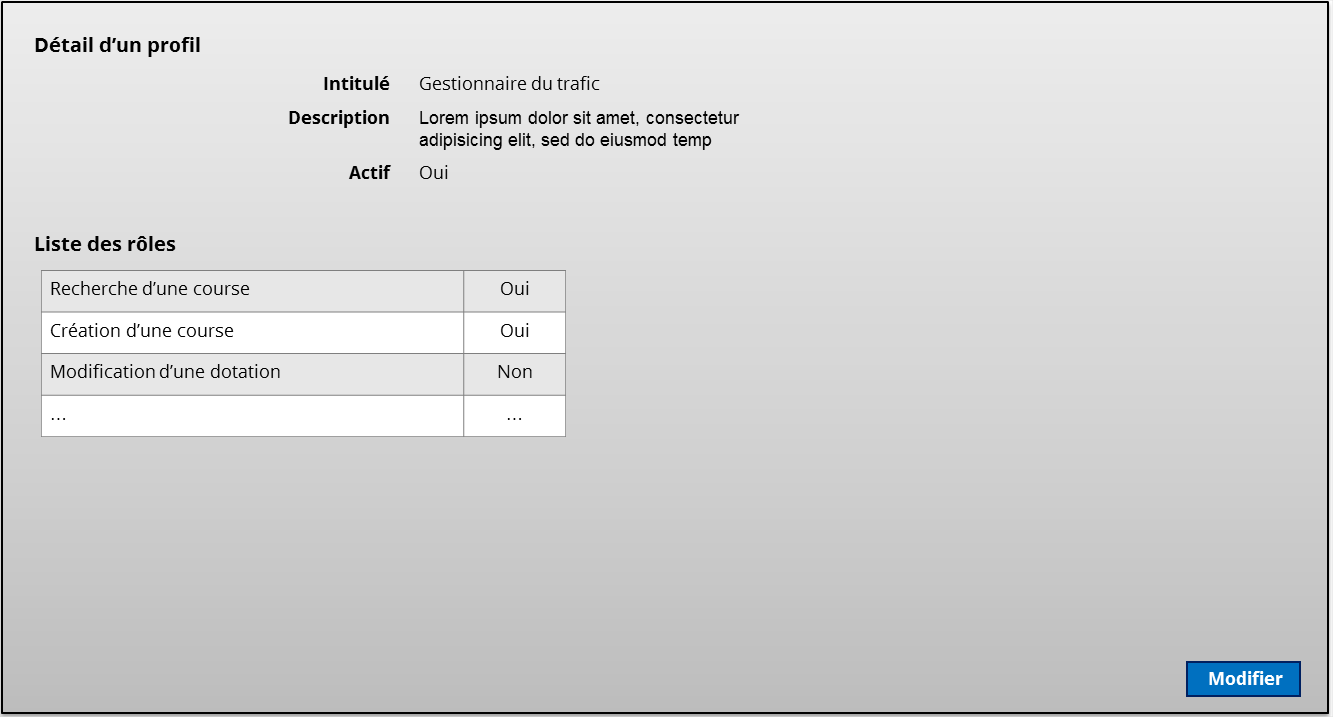


Figure 18 : Mockup of the example detail page (view mode)

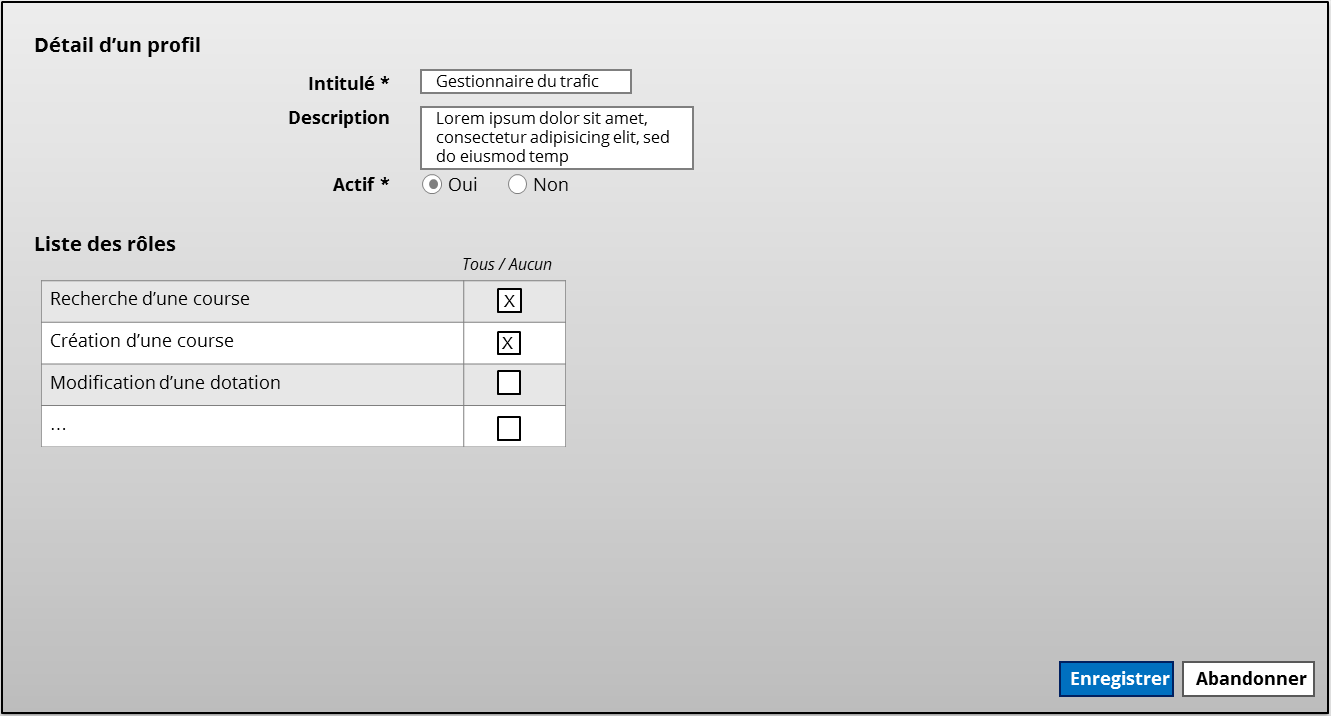


Figure 19 : Mockup of the example detail page (edit or create mode)

###### Creating the JSP file

Location

The first thing to do is to add the new JSP file at the right place. All JSP files should be located in a subfolder of the directory **src/main/webapp/jsp/**. Most of the time, the appropriate subfolder corresponds to the functional module that contains the page, use the user requirements to determine its name.

In our example, the detail page of a profile is located in the module called “Référentiel”, so the JSP file is added to the folder **src/main/webapp/jsp/referentiel**.

By convention, the name of a detail page respects the following format:

|  |
| --- |
| <type of the detailed item>Detail.jsp |

Here we are creating a profile (“profil” in French) detail page, so the JSP page will be named profilDetail.jsp.

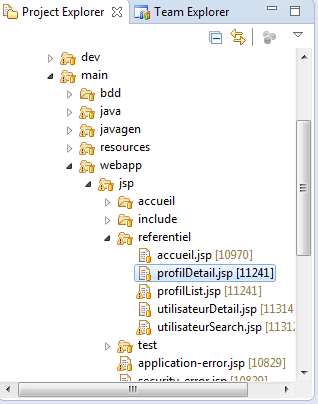


Figure 20 : Location of the JSP file for the profile detail page

Contents

Once the JSP file has been created, the next step is to write the page structure inside of it. The JSP file content is composed of four main parts:

1. The declaration of the language of the page and the import of tag libraries.
2. The inclusion of the page header.
3. The use of tags to describe the actual structure of the page.
4. The inclusion of the page footer.



Figure 21 : Structure of the JSP file for a detail page

Sections 1, 2 and 4 are the same in every JSP file. What changes is section 3, which contains the actual structure of the page. This section basically contains a <s:form> element where the properties of the object will be displayed or edited.

Form

The <s:form> element should contain all the tags to show or edit the properties of the Java objects loaded in the page.

In the current example, the form should contain:

* A <s:div> element to edit the properties “Intitulé”, “Description”, “Actif” of the profile.
* A <display:table> to select the roles of the profile.

Div element

The <s:div> element has its “layout” attribute set to “table”, so that the elements inside of it will be rendered inside an HTML table and thus be displayed on top of each other, with their label on the left and their value on the right.

The elements inside the <s:div> tag depend on the content of the page. They are usually, <s:textfield>, <s:checkbox>, <s:select>, <s:radio>. What is important is to bind each of these elements to a property of a Java object placed in the context of the page in the action class.

Here, the input are respectively mapped to the attributes “intitule”, “description”, “actif” of a Java object named “profil”:

|  |
| --- |
| <s:div layout=*"table"*>  <s:textfield name=*"profil.intitule"* label=*"default"* />  <s:textfield name=*"profil.description"* label=*"default"* />  <s:radio name=*"profil.actif"* label=*"default"*  list=*"*#{'true':'Oui', 'false':'Non'}*"* />  </s:div> |

Table element

The table element here is used to display and select the roles of the profile. See section Row selection in a non-editable list to learn how implement it. The important thing to remember is that the <display:table> element should be mapped to a Java object placed in the context by the action class. Here the table is mapped to a context object named “roles”:

|  |
| --- |
| <display:table name=*"roles"* uid=*"role"*>  ...  </display:table> |

###### Creating the action class

The action class is a Java class that extends AbstractActionSupport (from the io.vertigo.struts2.core package), or maybe there is an abstract action class specific to your project (for the project CRMS, the action classes should implement AbstractCRMSActionSupport).

Location

The action classes are located in a Java package named **<application alias>.controller.<functional module>** inside the directory **/src/main/java/**.

In our example, the application alias is “crms” and the page to display the detail of a profile is in the module named “Référentiel” (according to the user requirements), so the action class is added to the package crms.controller.referentiel.

The name of the action class should respect the following naming convention:

|  |
| --- |
| <Type of the detailed item>DetailAction.java |

Here, the action class will be named ProfilDetailAction.

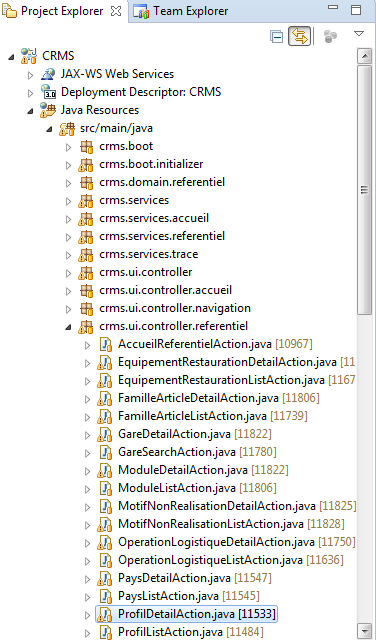


Figure 22 : Location of the action class for the user search page

Contents

The role of the action class of a detail page consists in:

1. Loading the object(s) to display in the context if the page is accessed in view mode.
2. Initialize the object to create and put them in the context if the page is accessed in create mode.
3. Performing any action that should be possible in the page according to the user requirements. They usually include:
   1. Toggling the edit mode of the page.
   2. Performing the saving of the object when user clicks the “Save” button when the page is in create or edit mode.
   3. Cancelling the creation or the modification of the object.
   4. Deleting the object when the user clicks the “Delete” button when the page is in view mode.

Attributes

As the class will load and save object(s) by calling the appropriate methods of a service, it needs a reference to this service. The class may contain as many service references as necessary.

In our example, the detail of a profile is loaded using the method loadProfil, declared in interface ProfilReadServices, and the roles of a profile are loaded using loadRoleApplicatifListByProfil defined in RoleApplicatifServices. Our class ProfilDetailAction will then need the two following service references:

|  |
| --- |
| @Inject  **private** ProfilServices profilServices;  @Inject  **private** RoleApplicatifServices roleApplicatifServices; |

* The type of this attribute should be the interface (and NOT the class that implements it) that declares the method that will be actually invoked to load the list.
* The @Inject annotation has to be added to the attribute. It will result in the injection of the interface’s implementation at runtime and allow the proper working of the page.

Context objects

The following information will have to be stored in the context:

* The profile which is an instance of the Java class “Profil”: it will be stored using ContextForm<T>.
* The list of all the roles using ContextMdl<T> (the list of roles is a reference list; see section Standard reference list to learn about them).

|  |
| --- |
| **private** **final** ContextForm<Profil> profilRef = **new** ContextForm<>("profil", **this**);  **private** **final** ContextMdl<RoleApplicatif> rolesRef = **new** ContextMdl<>("roles", **this**); |

* The first parameter of the ContextList and ContextMdl constructors is the context key; this key will be used in the JSP page to the context objects.
* The second parameter is always the current instance of the action class.

We also need context objects to store the primary key of the profile and the Boolean value indicating whether the button “Modifier” should be visible or not. The class ContextReft<T> is used to store a single object.

|  |
| --- |
| **private** **final** ContextRef<Long> proIdRef = **new** ContextRef<>("proId", Long.**class**, **this**);  **private** **final** ContextRef<Boolean> boutonModifierVisibleRef = **new** ContextRef<>("boutonModifierVisible", Boolean.**class**, **this**); |

* The first argument is the context key that will be used in the JSP page to refer to the object.
* The second argument is the class of the stored object.
* The last argument is the current instance of the action class.

Methods

initContext()

This method should contain the logic to:

* Load or create the objects to display/edit and put them in the page context
* Check if the button “Modifier” should be displayed and store the result in the context.

Here is the implementation of initContext() in our example:

|  |
| --- |
| **protected** **void** initContext(@Named("proId") **final** Option<Long> proId) {  String[] rolesSelectionnes;  **if** (proId.isDefined()) {  proIdRef.set(proId.get());  toModeReadOnly();  **final** Profil profil = profilServices.loadProfil(proIdRef.get());  rolesSelectionnes = initRolesSelectionnes();  profilRef.publish(profil);  } **else** {  toModeCreate();  rolesSelectionnes = **new** String[] {};  profilRef.publish(profilServices.getNewProfil());  }  boutonModifierVisibleRef.set(hasRole(Role.***R\_REF\_PRF\_MODIF***));  rolesRef.publish(RoleApplicatif.**class**, PersistenceManagerInitializer.***TOUS***);  rolesSelectionnesRef.set(rolesSelectionnes);  } |

* The initContext() method can have an indefinite number of parameters annotated with the @Named annotation. These annotated parameters represent the parameters of the URL used to access the page corresponding to the action class. In this example, the page ProfilDetail can have a parameter named “proId”. The type of parameter proId is Option<Long> which means that the URL parameter is optional. The URLs “http://…/ProfilDetail.do?proId=1” and “http://…/ProfilDetail.do” are both valid. The parameter “proId” is optional because the page can be accessed to view an existing profile (the URL then specifies the parameter “proId”) or to create a new profile (the URL does not specify the parameter “proId”). To prevent the access of the page without parameter “proId”, change the type of proId from Option<Long> to Long.
* If the primary key of a profile is specified using the “proId” parameter, it means that the page is being accessed in view mode, so the profile should be loaded from the database. This is done by calling the service loadProfil, the result is stored in the context object “profilRef”.
* If the “proId” parameter is undefined, the page is being accessed in create mode, so a new instance of Profil is created using the service getNewProfil, and stored in the context object “profilRef”.
* The button “Modifier” is visible if the current user has the role R\_REF\_PRF\_MODIF. This is determined by calling the method hasRole() defined in the class AbstractCRMSActionSupport. The result of this call is then stored in the context object by calling the method set() of the class ContextRef.
* The list of roles is loaded and stored in the context by calling the method publish() of the ContextMdl<T> class (see section Standard reference list for more information).

### Adding a header to a detail page

TODO

The header of a detail page usually contains:

* A summary of the object displayed in the page.
* A list of tabs to navigate to other tabs.

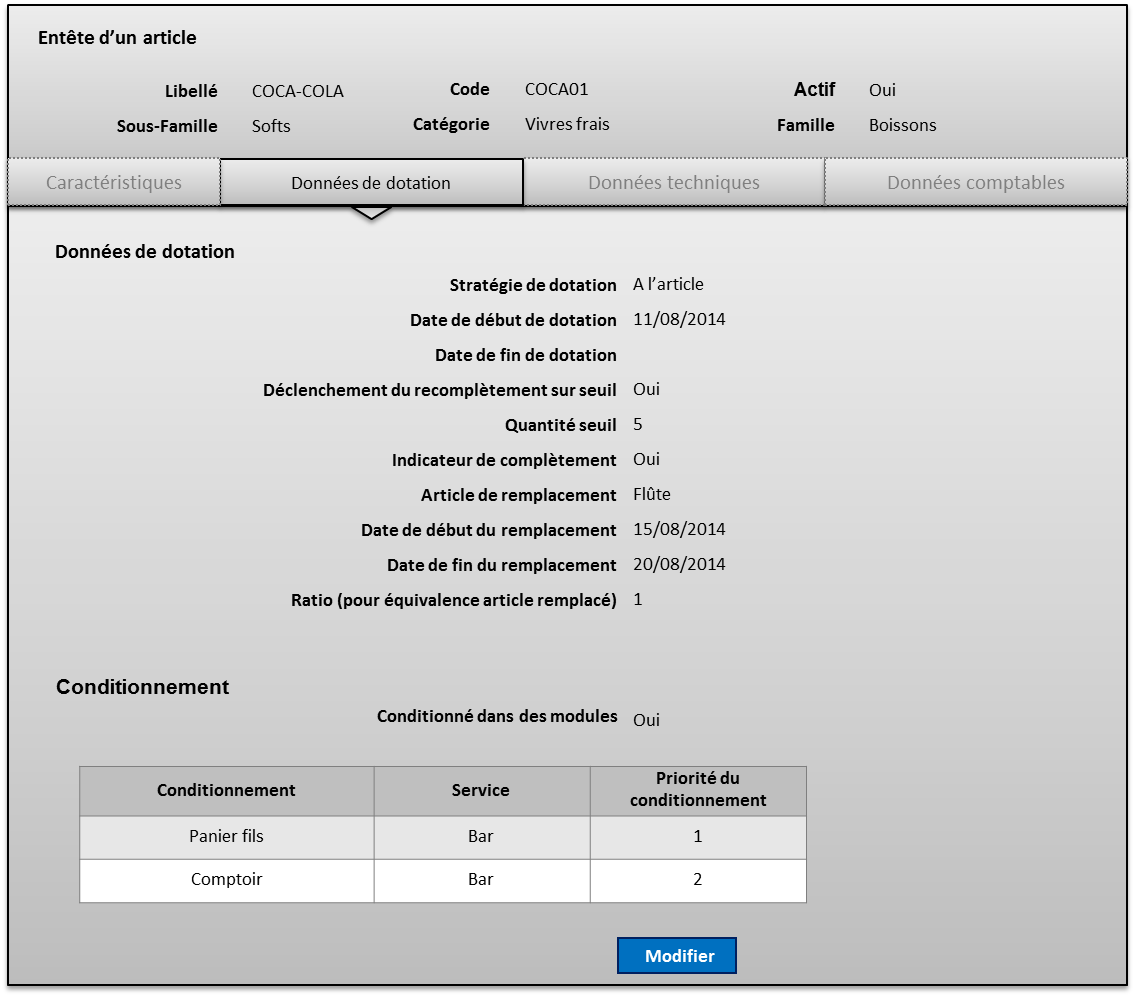


Figure 23 : Example of a detail page with a header

Adding a header to a page consists in:

* Creating the JSP file for the header.
* Creating a helper class that will be used to load the data to display and store it in the context.

TODO

### Components

#### Lists

##### Displaying a formatted Boolean in a column

When the type of a Java attribute mapped to a <display:column> tag is Boolean, you must use a <s:checkbox> inside the column to display the appropriate labels (Yes or No) instead of true or false:

|  |
| --- |
| <display:table name=*"typeEvtList"* id=*"typeEvt"* sort=*"list"* requestURI=*"#"* >  <display:column title=*"Actif"* sortable=*"true"* >  <s:checkbox name=*"actif"* value=*"%{#attr.typeEvt.actif}"* theme=*"simple\_read"*></s:checkbox>  </display:column>  </display:table> |

* Here the table displays a list of “TypeEvenement”. The list of objects is named “typeEvtList” (it is the value of the attribute “name” of the <display:table> tag), and an item of this list is referred to as “typeEvt” (which is the value of the attribute “id” of the <display:table> tag).
* The attribute to map to the only column of the table is TypeEvenement.actif, so we use a <s:checkbox> tag inside the <display:column> element and set its attributes as follows:
  + Value: %{#attr.<id>.<attribute name>} where <id> is “typeEvt” and <attribute name> is “actif”.
  + Theme: “simple\_read” so that the <s:checkbox> is simply rendered as a label displaying the appropriate text (Yes when “actif” equals true and No when “actif” equals” false).

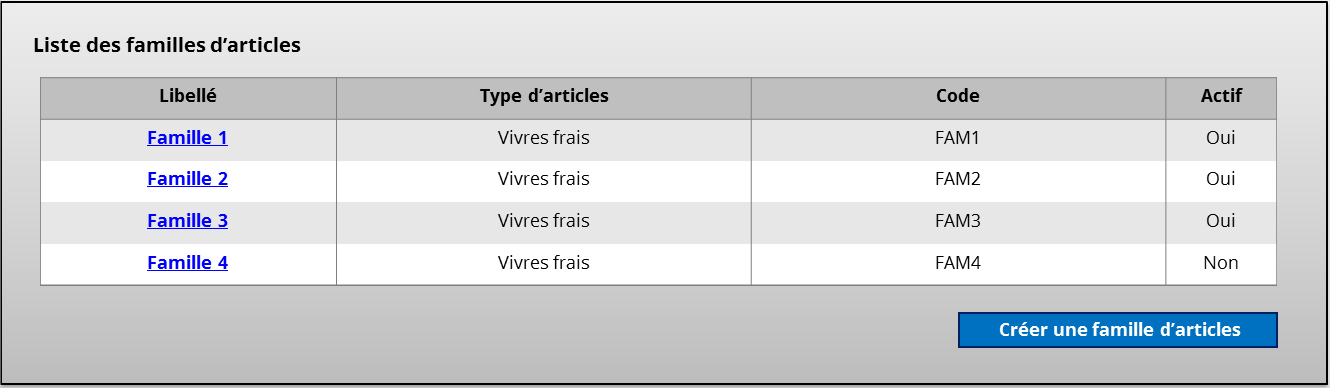
##### Displaying the label of a reference item in a column

The technique explained in this section works only to display items belonging to a reference list.

Use a <s:select> tag inside a <display:column> element to display the label of a reference item in the column.

For instance, the page “Liste des familles d’articles” lists instances of the FamilleArticles class. FamilleArticles contains a reference to the class TypeArticles:

|  |
| --- |
| **public** **final** **class** FamilleArticles **implements** DtObject {  **private** Long tatId; // Id of the referenced TypeArticle.    //... Rest of the class.  } |



The page “Liste des familles d’articles" includes a table with a column named “Type d’articles”. This column displays the label of the referenced TypeArticles of each listed FamilleArticles.

###### JSP file

The following code shows how the column “Type d’articles” has been implemented in the JSP file:

|  |
| --- |
| <display:table name=*"fmArticles"* uid=*"fmArticle"*">  ... First columns.  <display:column title=*"Type d'articles"*>  <s:select list=*"typesArticles"* value=*"%{#attr.fmArticle.tatId}"* /> </display:column>  ... Other columns.  </display:table> |

Table element

The <display:table> tag must specify two attributes:

* Name: this is the name of the context object that contains the list of “Familles d’articles”. This context object is an attribute of the action class. In our case, the action class will declare a context item identified as “fmArticles” (see section Action class).
* Uid: the name that will represent an item of the list in the JSP file. It will be used to access the id of the referenced “type d’articles” of each item.

Select element

The <display:column> tag contains a <s:select> tag that will actually render the label of the “type d’articles”. The <s:select> element must specify two attributes: “list” and “value”.

List

This is the name of the context object that contains the list of “Types d’articles”. This context object is an attribute of the action class. In our case, the action class will declare a context item identified as “typesArticles”.

Value

Value should contain an expression that describes how which property of the listed item will be used as the selected value. This property basically represents the foreign key of the reference item. Its format is:

|  |
| --- |
| %{#attr.<uid>.<foreign key>} |

Where:

* <uid> is the value of the attribute “uid” of the parent <display:table> element. It refers to the item corresponding to the row.
* < foreign key > is the name of the Java attribute corresponding to the foreign key of the reference item to display in the column.

###### Action class

Context objects

Listed items

The context object containing the items to list is declared as usual as a member of the Java action class. It can be a reference list or not.

In the current example, the list of “Familles d’articles” is not a reference list, so the type of the context object is ContextList:

|  |
| --- |
| **private** **final** ContextList<FamilleArticles> fmaListRefs = **new** ContextList<>("fmArticles", **this**); |

Reference list

The action class must also contain the reference list that contains the reference items to display in the column of the table. To store reference lists in the context, we use the ContextMdl class.

In the current example, the reference list is the list of “types d’articles”:

|  |
| --- |
| **private** **final** ContextMdl<TypeArticles> typesArticlesRef = **new** ContextMdl<>("typesArticles", **this**); |

Initialization of the context objects

The list of items and the reference list are loaded in the method initContext() method, as usual.

##### Sorting

Most of the time list should be sortable. When the list is coded using the <display:table> tag, enabling the sorting is done by setting the value of its attribute “sort” to “list” and the value of the attribute “requestURI” to “#”.

Next, for each of the columns that must be sortable, set the following attributes of their corresponding <display:column> tag:

* “sortable”: “true”.
* “sortProperty”: if the column you want to make sortable contains inside elements, you should set this attribute to the name of the property that will be used to sort it.

Example:

|  |
| --- |
| <display:table name=*"profils"* id=*"profil"* sort=*"list"* requestURI=*"#"*>  <display:column title=*"Intitulé"* sortable=*"true"* sortProperty=*"intitule"*>>  <s:url action=*"ProfilDetail"* var=*"profilDetailURL"*>  <s:param name=*"proId"*>${profil.proId}</s:param>  </s:url>  <a href=*"*${profilDetailURL}*"*>${profil.intitule}</a>  </display:column>  <display:column property=*"description"* title=*"Description"* sortable=*"true"*></display:column>  <display:column property=*"actif"* title=*"Actif"* sortable=*"true"*></display:column>  </display:table> |

##### Row selection in a non-editable list

This section explains how to make the rows of a non-editable list editable.

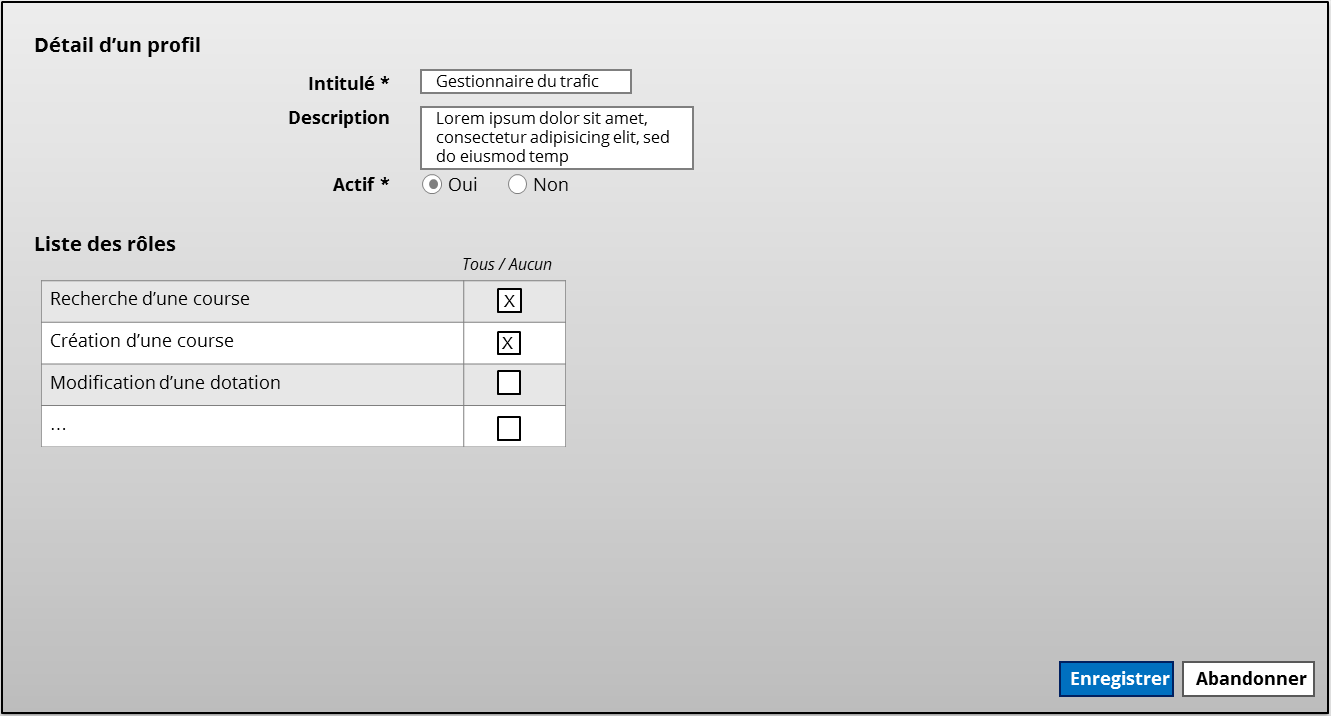


Figure 24 : Example of a non-editable list with selectable rows

The principle will be to use an array containing the primary key of each item mapped to a selected row.

###### JSP file

Because the content of its cells is not editable, the list is done using the <display:table> element of the Display Tag Library.

In our example, the list contains roles. Each of its rows has two columns:

* The label of the role corresponding to the row (here it will be the value of the attribute “libelle”).
* A check box to select the row.

|  |
| --- |
| <display:table name=*"roles"* uid=*"role"*>  <display:column property=*"libelle"*></display:column>  <display:column>  <s:checkbox name=*"rolesSelectionnes"*  fieldValue=*"%{#attr.role.rolCode}"*  theme=*"%{modeReadOnly ? 'simple\_read' : 'simple'}"*/>  </display:column>  </display:table> |

Table element

The <display:table> element must specify two attributes:

* Name: this is the name of the context object that contains the list to display in the table. This context object is an attribute of the action class. In our case, the action class will declare a context item identified as “roles” (see section Action class).
* Uid: the name that will represent an item of the list in the JSP file. It will be used to access the properties of each item in the columns declaration.

The <display:table > element must then contain the definitions of the columns used to display the information, and one column definition containing a <s:checkbox> element.

Check box element

This is the element that will make a row selectable. It must specify three attributes: name, fieldValue and theme.

Name

This is the name of the context object representing the array of the selected rows.

FieldValue

FieldValue should contain an expression that describes which property of the listed item will be stored in the array of the selected rows. Its format is:

|  |
| --- |
| %{#attr.<uid>.<attribute>} |

Where:

* <uid> is the value of the attribute “uid” of the parent <display:table> element. It refers to the item corresponding to the row.
* <attribute> is the name of the Java attribute whose value will be stored in the array of the selected rows.

Theme

The theme attribute defines how the <s:checkbox> element will be rendered.

If it is set to “simple”, the <s:checkbox> will be rendered as a checkbox.

When it is set to “simple\_read”, the <s:checkbox> element will be rendered as a <span> that will contain the label corresponding to the value mapped to the <s:checkbox> element.

In our example, the value of the theme attribute is set to an expression:

|  |
| --- |
| theme=*"%{modeReadOnly ? 'simple\_read' : 'simple'}"* |

This means that, the value theme will depend on the mode of the page. The figures below show the result.

|  |  |
| --- | --- |
| Figure 25 : List with selectable rows when the page is in read-only mode | Figure 26 : List with selectable rows when the page is in mode edit or create |

###### Action class

In the action class, we must:

* Declare the context objects:
  + One that will contain the listed items.
  + One that will contain an array used to keep track of the selected rows.
* Initialize the array of selected rows (when necessary).
* Handle the selected rows.

Context objects

Listed items

The context object containing the items to list is declared as usual as a member of the Java action class. It can be a reference list or not.

In the current example, the list of roles is a reference list, so the type of the context object is ContextMdl:

|  |
| --- |
| **private** **final** ContextMdl<RoleApplicatif> rolesRef = **new** ContextMdl<>("roles", **this**); |

Array of the selected rows

The selected rows are kept in a simple array of string values, which is stored in the context using the ContextRef class:

|  |
| --- |
| **private** **final** ContextRef<String[]> rolesSelectionnesRef = **new** ContextRef<>("rolesSelectionnes", String[].**class**, **this**); |

Initialization of the selected rows

Sometimes, some rows must be checked by default. To make this happen, the array of the selected rows must initially contain the primary key of each item whose corresponding row should be selected. This operation must be performed in the initContext method of the action class.

For example, the code below illustrates how to check the line corresponding to the role “Modification de profil” whose code is “REF\_PRF\_MODIF”:

|  |
| --- |
| **protected** **void** initContext() **throws** KSecurityException {  // Initialization code.  **final** String[] rolesSelectionnes = **new** String[] { "REF\_PRF\_MODIF" };  rolesSelectionnesRef.set(rolesSelectionnes);  // Other initialization code...  } |

Handling of the selected rows

The array containing the keys of the selected rows is retrieved by calling the get() method on the context object.

The code below shows how this is done in the case of our example:

|  |
| --- |
| **final** String[] rolesSelectionnes = rolesSelectionnesRef.get(); |

The variable rolesSelectionnes will contain the codes (i.e. the values of the attribute “rolCode”) of all the roles whose corresponding row is checked.

##### Non editable list

TODO

##### Editable list

A fully editable list is a list where the user can add or remove rows, and edit the content of each cell.

Coding an editable list will consist in:

* Defining the list layout in the JSP file (columns, buttons to add/delete rows).
* Writing the methods to handle the additions and deletions of rows in the Java action class.
* Editing the struts.xml file to declare a parameter named “rowindex” that will be used to keep track of the row to delete.

For example, let’s consider the detail of a VAT regime that contains an editable list of its VAT rates.

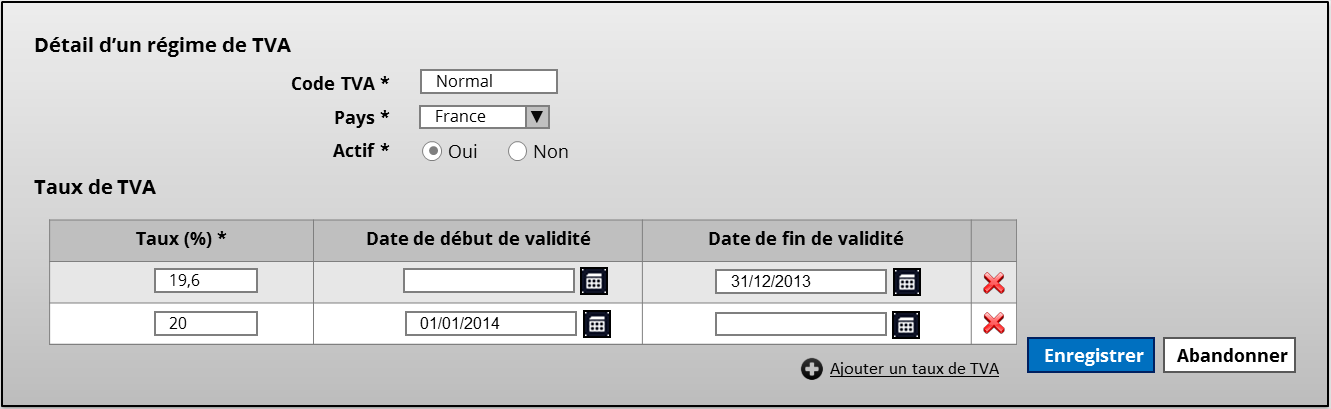


Figure 27 : Detail of VAT regime with an editable list of VAT rates

###### JSP file

Table element

The table is coded using a <display:table> element :

* Its name attribute is set to the name of a context list declared in the action class.
* Its uid attribute is set to an arbitrary value that will be used later in the JSP file to refer to each item of the context list.

In our example, the context list will contain the list of VAT rates.

|  |
| --- |
| <display:table name=*"tauxTvaList"* uid=*"tauxTva"*>  ...  </display:table> |

Column element

The <display:table> element contains a <display:column> element for each column to display.

Each <display:column> element then contains an element to render the appropriate input control (a text field, a date picker, a drop-down list, etc.). To bind an input control to the appropriate Java property, set its name attribute to the following value:

|  |
| --- |
| %{util.contextKey(#attr.<uid>).<property>} |

* <uid> is the value of the uid attribute of the parent <display:table> element.
* <property> is the name of the Java property to bind to the column.

For instance, the code below shows the declaration of the <display:column> elements that will allow the user to edit their content:

|  |
| --- |
| <display:column title=*"Taux (%)"*>  <s:textfield name=*"%{util.contextKey(#attr.tauxTva)}.taux"* />  </display:column>  <display:column title=*"Date de début de validité"*>  <sj:datepicker  name=*"%{util.contextKey(#attr.tauxTva)}.dateDebutValidite"*  changeMonth=*"true"* changeYear=*"true"* displayFormat=*"dd/mm/yy"*  showOn=*"button"* />  </display:column>  <display:column title=*"Date de fin de validité"*>  <sj:datepicker  name=*"%{util.contextKey(#attr.tauxTva)}.dateFinValidite"*  changeMonth=*"true"* changeYear=*"true"* displayFormat=*"dd/mm/yy"*  showOn=*"button"* />  </display:column> |

Delete row button

In an editable list, each row can be removed. This means that in addition to the <display:column> elements representing editable columns, the <display:table> element must contain another <display:column> element where the delete button will be located.

This additional column will only be displayed in create or edit mode, so its corresponding <display:column> element must be surrounded by a <s:if> tag.

|  |
| --- |
| <s:if test=*"%{modeCreate || modeEdit}"*>  ...  </s:if> |

Inside the <s:if> tag resides the declaration of column containing the delete button, as a <s:submit> element:

|  |
| --- |
| <display:column>  <s:submit  action=*"deleteTauxTvaRegimeTvaDetail%{util.indexOf(tauxTvaList, #attr.tauxTva)}"*  value=*"Supprimer"* type=*"input"* />  </display:column> |

The action attribute of the <s:submit> tag should respect the following format:

|  |
| --- |
| <method name><Java action class name>%{util.indexOf(<displayTable name attribute>, #<uid>)} |

* <Method name>: this is the name of the method of the Java action class that will handle the deletion of a row. Here, the method name is “deleteTauxTva”.
* <Java action class name>: the name of the Java action class. “RegimeTvaDetail” in this case.
* <displayTable name attribute>: this is the value of the name attribute of the parent <display:table> element. In our example, the name attribute of the <display:table> is set to “tauxTvaList”.
* <uid>: the value of the uid attribute of the parent <display:table> element. Here, “tauxTva”.

Add row button

The button to add a row is a simple <s:submit> element withing a <s:if> tag to render it in edit or create mode only:

|  |
| --- |
| <s:if test=*"%{modeCreate || modeEdit}"*>  <div class=*"button-bar"*>  <div class=*"right"*>  <s:submit action=*"addTauxTvaRegimeTvaDetail"*  value=*"Ajouter un taux de TVA"* cssClass=*"edit"* />  </div>  </div>  </s:if> |

###### Action class

The action class must declare:

* The context list that will contain the items of the editable list.
* The method to handle the deletion of a row.
* The method to handle the addition of a row.

Context list

The context list that will hold the items of the editable list must be of type ContextListModifiable<T>. In our example:

|  |
| --- |
| **private** **final** ContextListModifiable<TauxTva> tauxTvaListRef = **new** ContextListModifiable<>("tauxTvaList", **this**); |

Delete row method

The method called when the user clicks the “delete” button of a row will retrieve the index of this row and remove the item at this index from the context list.

In order to get the row index, make sure that the parameter “rowindex” is declared in the struts.xml file (see section Struts configuration file).

|  |
| --- |
| **public** String deleteTauxTva() {  **final** **int** index = getRowIndex();  **final** UiListModifiable<TauxTva> tauxTvaList = tauxTvaListRef.getUiListModifiable();  tauxTvaList.remove(index);  **return** ***NONE***;  } |

Add row method

This method will be called when the user clicks the “add” button. It simply adds a new element to the context list mapped to the editable list in the JSP file:

|  |
| --- |
| **public** String addTauxTva() {  **final** UiListModifiable<TauxTva> tauxTvaList = tauxTvaListRef.getUiListModifiable();  tauxTvaList.add(**new** TauxTva());  **return** ***NONE***;  } |

Struts configuration file

To properly retrieve the index of the row that is being deleted when clicking the “delete” button, the <action> element corresponding to the page must declare a parameter named “rowindex”, as the example below shows:

|  |
| --- |
| <action name=*"\*RegimeTvaDetail\*"*  class=*"crms.ui.controller.referentiel.RegimeTvaDetailAction"* method=*"{1}"*>  <param name=*"rowindex"*>{2}</param>  <result name=*"none"*>/jsp/referentiel/regimeTvaDetail.jsp</result>  <result name=*"cancel\_creation"* type=*"redirectAction"*>RegimeTvaList</result>  </action> |

* The name of the action contains two wildcard characters ‘\*’:
  + One as a prefix: it represents the name of the method to invoke in the Java action class.
  + One as a suffix: it will contain the index of the deleted row (see section Delete row button).
* The value of the parameter “rowindex” is set to {2} to indicate that its value is taken from the second wildcard character in the action name.

#### AutoComplete input

AutoCompleted text fields search a reference list using a term entered by the user. The term is used to perform a full-text-search on a particular field of the reference list.

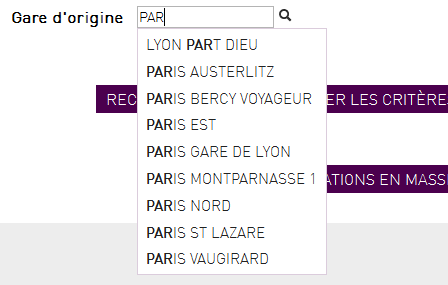


Figure 28 : Example of an autocompleted input

Make sure the JSP file includes the following taglib:

|  |
| --- |
| <%@ taglib uri=*"http://displaytag.sf.net"* prefix=*"display"* %> |

Then, implement the autocompleted text field by using the element <sj:autocompleter>:

|  |
| --- |
| <sj:autocompleter list=*"gares"* label=*"default"* name=*"criteria.garIdOrigine"* listKey=*"garId"* listValue=*"libelle"* headerKey=*""* headerValue=*""* href=*"ListAutocomplete.do"* /> |

Here is a description of the element’s main attributes:

* list: the name of the context variable containing the searched reference list. (See section Adding a reference list)
* name: the attribute of the Java object that will be mapped to the autocompleted text field.
* listKey: the attribute of the objects in the reference list that will be used to assign the selected value.
* listValue: the attribute of the objects in the reference list that will be used to perform the search and to display the results.
* href: the link to the Java action class that will actually perform the search.

#### Date picker

TODO

### Adding a pop-in

A pop-in is a window that overlays the current page. It can be used to as an information window, a confirmation panel, a form to enter data, a search window, etc.

#### Simple pop-in

A simple pop-in basically contains confirmation message or simple forms. If your pop-in must contain a real page, consider section Include a page in a pop-in.

Adding a simple pop-in to a page consists in:

* Writing its structure in the JSP file of the page where the pop-in should appear.
* Writing the Java code in the action class of the page where it should appear.

##### JSP file

A pop-in is a simple <div> element containing everything the pop-in should contain (<s:form>, <display:table>, <p>, etc.).

A pop-in is displayed depending on some conditions, so the <div> element must be surrounded by a <s:if> element that will check this condition and render its content only if this condition is satisfied.

Finally, the inputs under a pop-in should not be editable. To disable them, some JavaScript is necessary.

Here is a simple example of the JSP code to define a pop-in:

|  |
| --- |
| <s:if test=*"displayModal"*>  <div id=*"myPopin"* class=*"modal hide detail"*>  <div class=*"modal-header"*><h4>Pop-in title</h4></div>  <div class=*"modal-body"*>  <s:actionmessage />  <p>Hello world!</p>  </div>  </div>  <script>  $(document).ready(**function**() {  showModal('#myPopin');  });  </script>  </s:if> |

* Test: it is a boolean expression that will be evaluated. If true, the content of the <s:if> element will be rendered. Here name is set to “displayModal”, which refers to a boolean context item which is declared and set in the Action class corresponding to the JSP file.
* Id: the id of the <div> element. This id is will be passed to the JavaScript function showModal() in order to display the pop-in over the page.
* The <script> element only contains the code to call showModal() when the document is ready.

##### Action class

The Java action class corresponding to the JSP file must at least declare the context object that is used in the <s:if> element to determine whether the pop-in should be rendered or not.

|  |
| --- |
| **private** **final** ContextRef<Boolean> displayModalRef = **new** ContextRef<>("displayModal", Boolean.**class**, **this**); |

Because the pop-in is initially hidden, the context object must be initially set to “false”. This is done in the initContext() method of the action class:

|  |
| --- |
| **public** **void** initContext() {  // Page initialization.  displayModalRef.set(Boolean.***FALSE***);  // Other page initialization.  } |

#### Include a page in a pop-in

Sometimes a page should be displayed inside a pop-in.

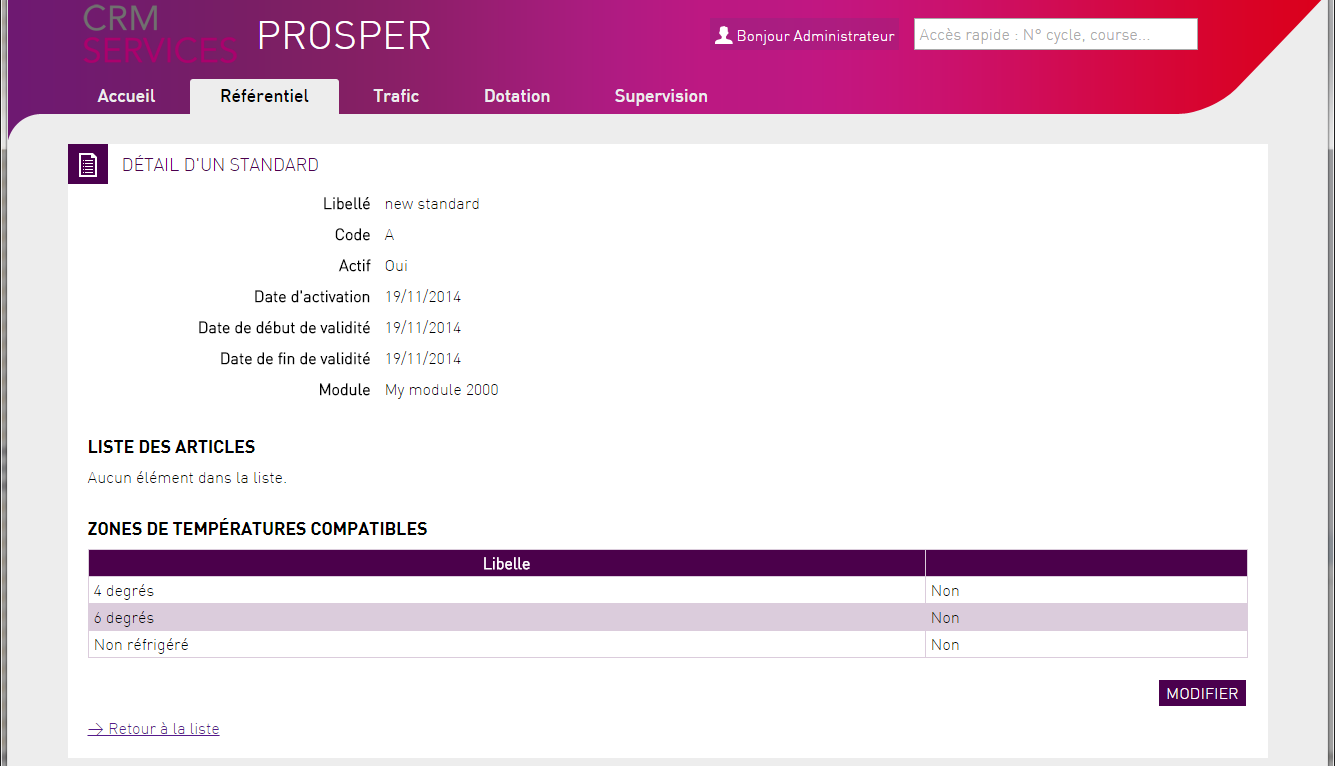


Figure 29 : The page to include in a pop-in

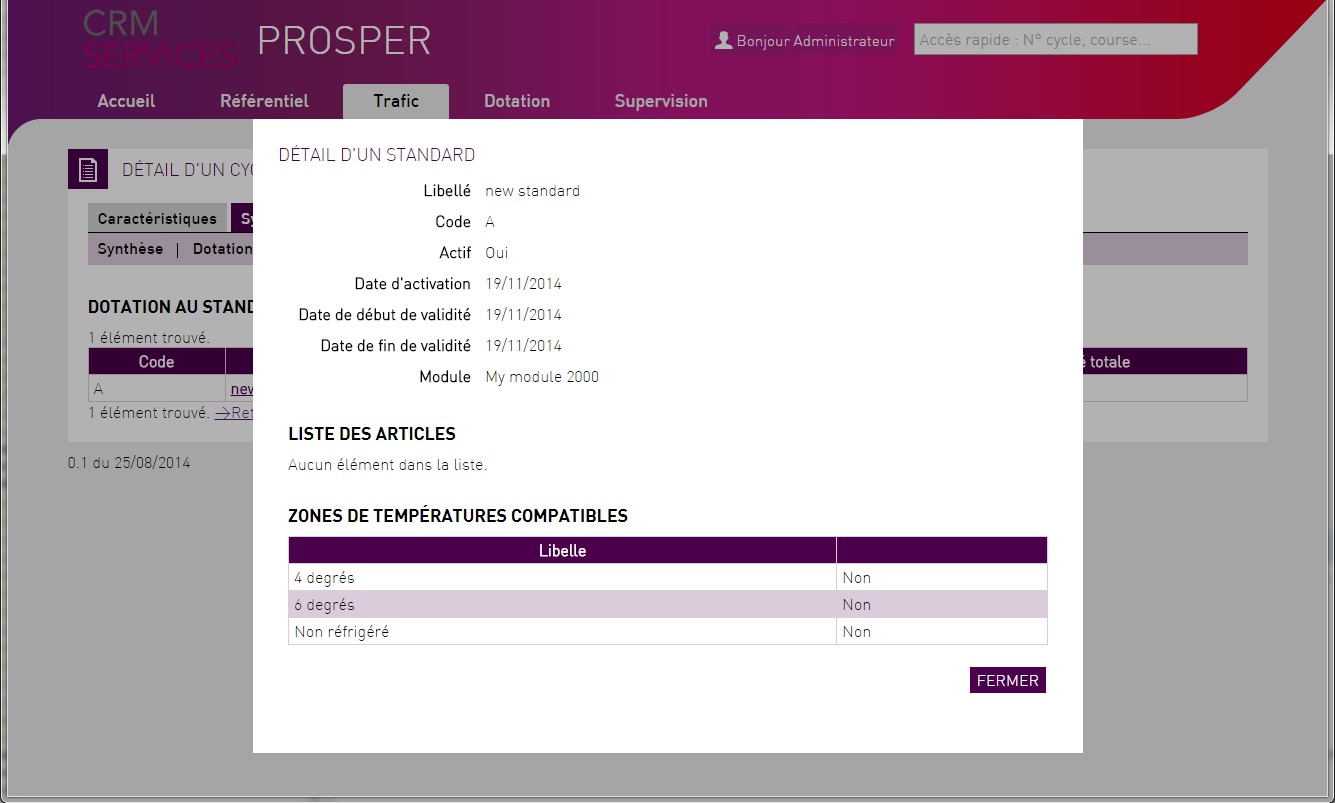


Figure 30 : The same page displayed inside a pop-in

##### In parent page of the pop-in

###### Include popinGeneric.jsp

|  |
| --- |
| <s:include value=*"/jsp/popin/popinGeneric.jsp"*/> |

This JSP file contains the <iframe> element representing the pop-in window and the JavaScript functions that will be used to show/display it.

###### Call showPopin(url) to display the pop-in

Add an event-listener to the elements that should trigger the display of the pop-in.

|  |
| --- |
| <script>  $(document).ready(**function**() {  $("#standard a").on( "click", **function**() {  showPopin("StandardDetail.do?popinMode=true&stdId="+$(**this**).attr('id'))  });  });  </script> |

The parameter of the showPopin function is the URL of the page that will be displayed in it. Note the parameter named “popinMode” that will tell the page to be included that it will be rendered inside a pop-in.

##### In the page displayed in the pop-in

###### Java action class

Add the @AcceptCtxQueryParam annotation to the class

This will allow your page to be initialized with URL parameters when it is in pop-in mode.

|  |
| --- |
| @AcceptCtxQueryParam  **public** **class** StandardDetailAction **extends** AbstractCRMSActionSupport { |

Add a context attribute indicating whether the pop-in mode is enabled

Add a new attribute to your Java action class (StandardDetailAction.java in this case):

|  |
| --- |
| **private** **final** ContextRef<Boolean> popinModeRef = **new** ContextRef<>("popinMode", Boolean.**class**, **this**); |

Add an optional URL parameter to enable the pop-in mode

Add a new optional parameter to the method initContext of your Java action and use it to set the context attribute added previously.

|  |
| --- |
| **protected** **void** initContext(@Named("stdId") **final** Option<Long> stdId, @Named("popinMode") **final** Option<String> popinMode) {    **if** (popinMode.isDefined()) {  popinModeRef.set(Boolean.***TRUE***.toString().equalsIgnoreCase(popinMode.get()));  } **else** {  popinModeRef.set(**false**);  }  // ... |

If necessary, adapt the logic of your page when it is in pop-in mode. Here, the page should be displayed in read-only mode if popinMode is “true”, so we need to add logic to the initContext method.

|  |
| --- |
| **protected** **void** initContext(@Named("stdId") **final** Option<Long> stdId, @Named("popinMode") **final** Option<String> popinMode) {    **if** (popinMode.isDefined()) {  popinModeRef.set(Boolean.***TRUE***.toString().equalsIgnoreCase(popinMode.get()));  } **else** {  popinModeRef.set(**false**);  }    **if** (popinModeRef.get()) {  stdIdRef.set(stdId.get());  toModeReadOnly();  loadData(stdId.get());  } **else** {  // Normal mode.  }  } |

###### JSP file

The JSP file of the page that will be displayed in a pop-in should be adapted so that when it is in pop-in mode:

* The application banner containing all the menus is NOT included.
* Some elements of the page (edit button, back to list link, etc.) are hidden and a button to close the pop-in is added.

This is done using the context attribute that was added in the Java action class.

Include the proper header

|  |
| --- |
| <s:if test=*"popinMode"*>  <s:include value=*"/jsp/include/popinHeader.jsp"* />  </s:if>  <s:else>  <s:include value=*"/jsp/include/pageHeader.jsp"* />  </s:else> |

Include the proper footer

|  |
| --- |
| <s:if test=*"popinMode"*>  <s:include value=*"/jsp/include/popinFooter.jsp"* />  </s:if>  <s:else>  <s:include value=*"/jsp/include/pageFooter.jsp"* />  </s:else> |

Add a button to close the pop-in and hide the non-necessary buttons

|  |
| --- |
| <div class=*"button-bar"*>  <div class=*"right"*>  <s:if test=*"popinMode"*>  <s:submit value=*"Fermer"* cssClass=*"cancel"* type=*"button"*  onclick=*"parent.closePopin()"* />  </s:if>  <s:else>  <s:if test=*"%{modeReadOnly}"*>  <s:if test=*"boutonModifierVisible"*>  <s:submit action=*"editStandardDetail"* value=*"Modifier"*  cssClass=*"edit"* />  </s:if>  </s:if>  <s:else>  <s:submit action=*"saveStandardDetail"* value=*"Enregistrer"*  cssClass=*"save"* />  <s:submit action=*"cancelStandardDetail"* value=*"Abandonner"*  cssClass=*"cancel"* />  </s:else>  </s:else>  </div>  </div> |

The onclick attribute of the button “Fermer” will call the JavaScript function closePopin() of the iframe’s parent. closePopin() is defined in popinGeneric.jsp.

##### Conclusion

The page is now ready to be displayed in a pop-in.

|  |
| --- |
| And of course, make sure that these adaptations did not introduce any bugs in the page by testing it both in normal mode and pop-in mode! |

### Adding a new menu element

Menus are managed in a Java file named Menu.java. It contains an enumeration type where the menu hierarchy of the web application is declared.

To add a new menu element, declare a new value in the enumeration type by specifying:

* Its name (which is the name of the enumeration value).
* Its parent menu element (which is another value of the enumeration type).
* Its label.
* Its associated action.

Here is an example of a Menu.java file:

|  |
| --- |
| **public** **enum** Menu {  /\*\* Menu racine. \*/  ***ROOT\_MENU***(),  /\*\* Menu ACCUEIL. \*/  ***ACCUEIL***(***ROOT\_MENU***, "Accueil", "AccueilTest.do"),  /\*\* Module "Référentiel". \*/  ***REFERENTIEL***(***ROOT\_MENU***, "Référentiel", "AccueilReferentiel.do"),    // Members, constructors and methods of the Menu enum...  /\*\* Liste des profils. \*/  ***LISTE\_PROFILS***(***REFERENTIEL***, "Liste des profils", "ProfilList.do"),    // Members, constructors and methods of the Menu enum...  } |

The menu element LISTE\_PROFILS:

* Name: LISTE\_PROFILS.
* Parent menu element: REFERENTIEL.
* Label: “Liste des profils”.
* Associated action: “ProfilList.do”.

**Remark**

The menu element will never appear if:

* There is no action declared in the struts.xml file that matches the action of the menu element.
* There is a no permission associated to the action in the file or if there is no role containing this permission (the permission(s) and role(s) should be declared in the <application alias>-auth-config.xml file, see section Adding a new role).

### Adding a new role

In the Java web application, roles are specified in the file **<application alias>-auth-config.xml**.

A role is a set of permissions where a permission is the association of a resource (a page, data, or a service) and an operation type (read or write). So, adding a new role consists in:

* Declaring the permissions that the role will contain if they do not exist.
* Declaring the role itself.
* Calling the ant script that will generate an Enum value corresponding to the role, so that it will be accessible in Java code.

#### Defining a permission

A permission associate a resource to an operation, it is defined using the <permission> element. Here is the attributes of the <permission> element:

* Description.
* Id.
* Filter.
* Operation.

For instance, the permission to access the user search page of an application would be declared like this:

|  |
| --- |
| <permission description=*"Accès à l'écran de recherche des utilisateurs"* id=*"PRM\_REF\_UTILISATEUR\_LIST"* filter=*"/PAGE/UtilisateurSearch.do"* operation=*"OP\_READ"* /> |

##### Description

It is the user-friendly description of the permission.

##### Id

This is the identifier of the permission. It will be used to refer to it by the roles which will contain it.

##### Filter

The filter is an expression that identifies the resource; it depends on the resource type:

|  |  |  |
| --- | --- | --- |
| Type | Filter | Example |
| Page | /PAGE/{url} | /PAGE/Accueil.do |
| Data | /DATA/{URI} | /DATA/TProfilPro/[^/]\*/${cptId} |
| Service | /SERVICE/{class.method} | TODO |

##### Operation

OP\_READ, OP\_WRITE, OP\_READ\_WRITE.

TODO

#### Defining the role

A role is a set of permissions, it is defined using a <role> element, whose attributes are:

* Name.
* Description.

Inside the <role> element are <permission> elements, one for each permission of the role:

|  |
| --- |
| <role name=*"R\_REF\_UTI\_CONS"* description=*"Consultation d’utilisateur"*>  <permission ref=*"PRM\_REF\_UTILISATEUR\_LIST"* />  <permission ref=*"PRM\_REF\_UTILISATEUR\_DETAIL"* />  <permission ref=*"PRM\_REF\_UTILISATEUR\_READ"* />  </role> |

##### Name

The name of the role must begin by the prefix “R\_”. After this prefix, use capital letters and separate terms with the underscore character (use the code of the role in the user requirements).

##### Description

The description should be a short, user-friendly explanation of the role: basically the label of the role as defined in the user requirements.

##### Permissions

To add a permission to a role, put a <permission> element inside of its <role> element. The <permission> element has one attribute, “ref”, which must contain the Id of the referenced permission.

Example:

|  |
| --- |
| <permission ref=*"PRM\_REF\_UTILISATEUR\_LIST"* /> |

#### Generating the Java Enum value

In addition to generating the Java classes for the persistent and non-persistent model, SQL script, the Ant script build-mda.xml also generates a Java file named “Role.java” in the package <application alias>.security located in the folder src/main/javagen/.

Role.java contains an enumeration named “Role” which enumerates all the roles declared in <application alias>-auth-config.xml.

Once the role and its permissions are defined in the <application alias>-auth-config.xml file, run the Ant script build-mda.xml. This will add a value to the Java Enum Role with the same name as the role declared in the XML file:

|  |
| --- |
| **public** **enum** Role {  /\*\*  \* R\_REF\_UTI\_CONS.  \*/  ***R\_REF\_UTI\_CONS***,  } |

### Adding a reference list

#### Standard reference list

##### Definition

The definition of reference lists is done in the method init() of the “PersistenceManagerInitializer” class.

As an example, let us define the following reference lists:

* One representing all the items in the table corresponding to the Java class Civilite.
* A second one representing all the items in the table corresponding to the Profil class.
* A third one representing the items in the table corresponding to the Profil class whose column IS\_ACTIF is equal to true.

Here is the code of the init() method in the PersistenceManagerInitializer class:

|  |
| --- |
| **public** **void** init(**final** PersistenceManager persistenceManager) {  *registerMasterData*(persistenceManager, Civilite.**class**);  *registerMasterData*(persistenceManager, Profil.**class**);  *registerMasterData*(persistenceManager, Profil.**class**, "ACTIFS", "isActif", **true**);  } |

* The first reference list is straightforward; it will be used to access the list of all the available civilités.
* The second reference list is also pretty obvious.
* The second one illustrates how to define reference lists to access only a subset of a table. In this example: the profils whose property isActif is equal to true. To do this, the call to registerMasterData() contains three more parameters:
  + “ACTIFS”: this is the code of this reference list for the class Profil.
  + “isActif”: this is the name of the Java attribute of the Profil class that will be used to determine which profils belong to this list.
  + true: the value of the property

##### Sorting and display fields

When adding a reference list concerning a new object, its sorting field and its display field must be specified in the KSP file named “mdm.ksp”. Just add a new instruction to declare the sorting and display fields:

|  |
| --- |
| alter DtDefinition DT\_<REFERENCE\_LIST\_TYPE> {  sortField : "<SORT\_FIELD>"  displayField : "<DISPLAY\_FIELD>"  } |

For instance, the sort field of “Civilite” objects is “libelle”, and so is their display field:

|  |
| --- |
| alter DtDefinition DT\_CIVILITE {  sortField : "LIBELLE"  displayField : "LIBELLE"  } |

##### Usage

###### In a page

Let us consider an example where we want to use the reference list of all the active profiles of the application as the source of a drop-down list in a page. This will consist in:

* Editing the Java action class corresponding to the page to declare and load the reference list.
* Editing the JSP file corresponding to the page to use the reference list as the source of a drop-down list.

Action class

Adding a context object

The first thing to do is to add a context object as an attribute of the Java action class corresponding to the JSP file. In the case of reference list, use the ContextMdl<> class.

|  |
| --- |
| **private** **final** ContextMdl<RoleApplicatif> rolesApplicatifsRef = **new** ContextMdl<>("rolesApplicatifs", **this**); |

* The first parameter of the constructor, here “rolesApplicatifs”, is the name that will be used to refer to the reference list in the JSP file.
* The second parameter is the current instance of the Java action class, as usual.

Publishing the list in the context object

In order to load the reference list and add it to the context object, call the publish() method of the context object in the method initContext() of the action class:

|  |
| --- |
| **public** **void** initContext() {  // Initialization code.  profilsRef.publish(Profil.**class**, "ACTIFS");  // Other initialization code.  } |

The publish method of the ContextMdl<> class takes two parameters:

* The first parameter is the class of the reference list items.
* The second one is the code that identifies it. This code is the same as the one that was used during the Definition of the reference list, in the PersistenceManagerInitializer.

JSP file

The reference list is used as the source of a drop-down list. Here the reference list contains instances of the “Profil” class, whose primary key is represented by the field “proId”

For each item of the reference list, the displayed value will be the value of the field “intitule”. The actual value that will be used in the code will be the value of the field “proId”. The selected value will be used to set the property “proId” of the context object named “criteria”.

|  |
| --- |
| <s:select name=*"criteria.proId"* list=*"profils"* listKey=*"proId"* listValue=*"intitule"* label=*"default"*></s:select> |

###### In a service

TODO

#### Custom reference list

The reference lists described in the section Standard reference list are mapped on a single table, and their sorting and display fields correspond to columns of this table.

It is possible to create reference lists that are mapped to the result of a query and whose sorting and/or display field correspond to a computed field.

As an example, we’re going to create a reference list whose display field contains the concatenation of its label and its code.

To do so, we will:

* Create a new non-persistent class that will contain a field corresponding to the concatenation of the label and the code of the article.
* Create a new StorePlugin that will be used to load all the articles and compute the concatenation of their label and their code.
* Declare the new reference list using both the new non-persistent class and the new StorePlugin.

##### Create a new non-persistence class

We’re going to create a new class named ArticleMdl. By convention, add the suffix “Mdl” to classes that are intended to be used with reference lists.

ArticleMdl will have the following properties:

* artId: the id of the article.
* display: the property that will contain the label and the code of the article formatted as “label (code)”.
* actif: so that we will be able to create a reference list corresponding to the active articles only (i.e. articles where actif = true).

##### Adding the new store plugin

The store plugin will be used to load all the articles; it is a class that extends AbstractStaticDataStorePlugin:

|  |
| --- |
| **public** **class** ArticleMdlStorePlugin **extends** AbstractStaticDataStorePlugin {  **private** **final** ComponentRef<ArticleMdlPAO> articleMdlPAO = ComponentRef.*makeLazyRef*(ArticleMdlPAO.**class**);  @Override  **public** <D **extends** DtObject> D load(**final** URI<D> uri) {  **return** (D) articleMdlPAO.get().loadArticleMdl((Long) uri.getKey());  }  @Override  **public** <D **extends** DtObject> DtList<D> loadList(**final** DtListURI uri) {  **if** (uri **instanceof** DtListURIAll) {  **final** DtList<ArticleMdl> result = articleMdlPAO.get().loadArticleMdlList();  result.setURI(uri); //important pour la mise en cache  **return** (DtList<D>) result;  }  **throw** **new** UnsupportedOperationException();  }  } |

To load the data, the store plugin uses a PAO (see section Creating a PAO object).

The PAO should contain two tasks:

* One that loads all the elements of the reference list.
* One that loads one element using its id.

The tasks should both return objects of the type of the non-persistent class that you created at section Create a new non-persistence class. They should also compute the value of the property that will be searched (here this property is “display”; it will be computed by concatenating the label and the code of the article into a single string):

|  |
| --- |
| create Task TK\_LOAD\_ARTICLE\_MDL {  className : "io.vertigo.dynamox.task.TaskEngineSelect"  request : "  SELECT  ART\_ID,  LIBELLE || ' (' || CODE || ')' AS DISPLAY,  CAR\_CODE,  ACTIF  FROM  ARTICLE  WHERE  ART\_ID = #ART\_ID#  "  attribute ART\_ID {domain : DO\_ID, notNull:"true", inOut :"in"}  attribute ARTICLE\_MDL {domain : DO\_DT\_ARTICLE\_MDL\_DTO, notNull:"true", inOut :"out"}  },  create Task TK\_LOAD\_ARTICLE\_MDL\_LIST {  className : "io.vertigo.dynamox.task.TaskEngineSelect"  request : "  SELECT  ART\_ID,  LIBELLE || ' (' || CODE || ')' AS DISPLAY,  CAR\_CODE,  ACTIF  FROM  ARTICLE  ORDER BY  DISPLAY  "  attribute ARTICLE\_MDL\_LIST {domain : DO\_DT\_ARTICLE\_MDL\_DTC, notNull:"true", inOut :"out"}  }, |

##### Declaring the new reference list

In the class PersistenceManagerInitializer, in the method init, register the new reference list by calling registerMasterData and passing to it the new non-persistent class, and the new store plugin:

|  |
| --- |
| *registerMasterData*(persistenceManager, ArticleMdl.**class**, ***ACTIFS\_CODE***, ***ACTIFS\_FIELD\_NAME***, Boolean.***TRUE***, **new** ArticleMdlStorePlugin(), **false**); |

The principle is similar to the declaration of reference lists using persistent classes, except that we need to provide the store plugin.

WARNING!

If you want to declare several reference lists using the same non-persistent class, you must not pass the new store plugin again:

|  |
| --- |
| *registerMasterData*(persistenceManager, ArticleMdl.**class**, ***ARTICLES\_PREFIX*** + code, ***CAR\_CODE\_FIELD\_NAME***, code); |

##### Usage

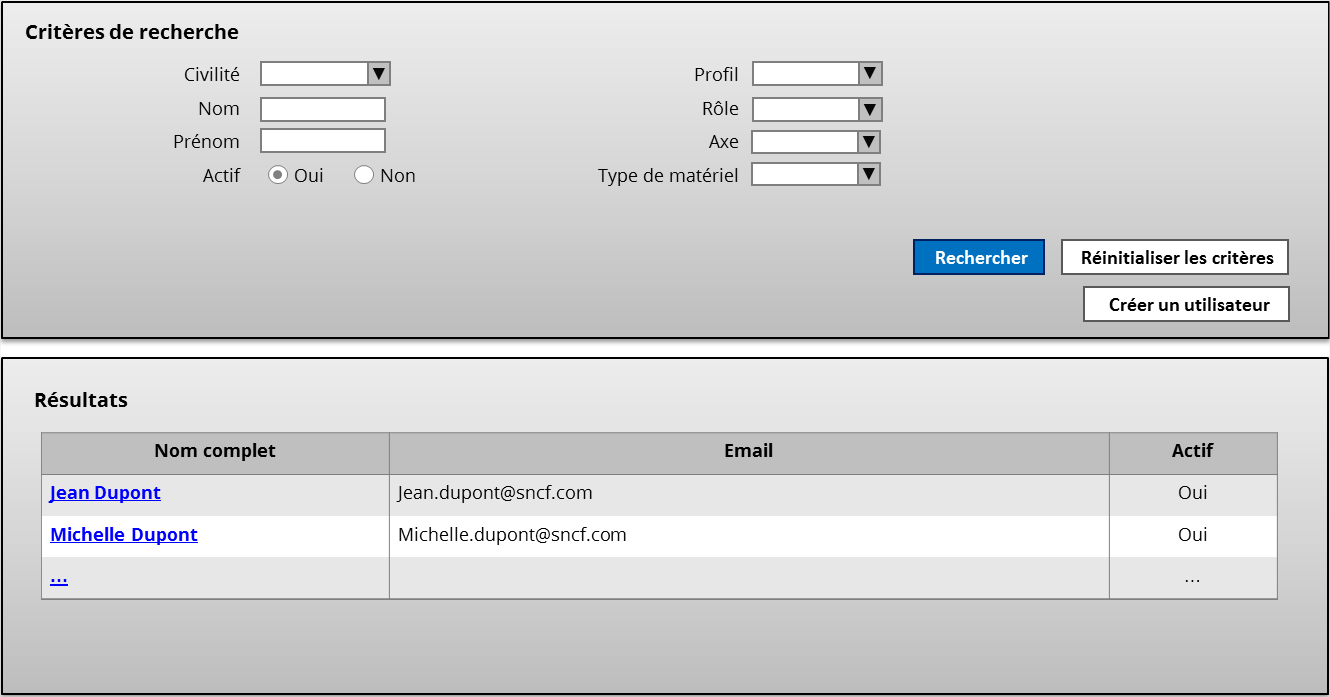
The usage is exactly the same as the usage of standard reference list. See section Usage.

## Services

Services are Java interfaces declaring the methods that will be called by the application to apply/check business rules and select/insert/update/delete data in the database.

A service should be a functionally coherent (relatively to the user requirements) interface, which means that it should contain methods that deal with the same functional module of the application or with the same functional entity. For instance: a service that deals with users and declares methods to search, to create, to delete users.

As an example, we are going to write the service that will be called by the user search page (see section Search page to know how to create the user interface and call the services from it):



The service will initially contain one method that will perform the search by executing a custom SQL query against the database.

### Creating the interface

#### Location

The services are found in a Java package named **<application alias>.services.<functional module>** inside the directory **/src/main/java/**.

In our example, the application alias is “crms” and the user entity (“Utilisateur”) is in the module named “Référentiel” (according to the user requirements), so the service interface is added to the package crms.services.referentiel.

The name of the service interface should respect the following naming convention:

|  |
| --- |
| <Name of the concerned functional module or entity >Services.java |

Here, the interface will be named UtilisateurServices.

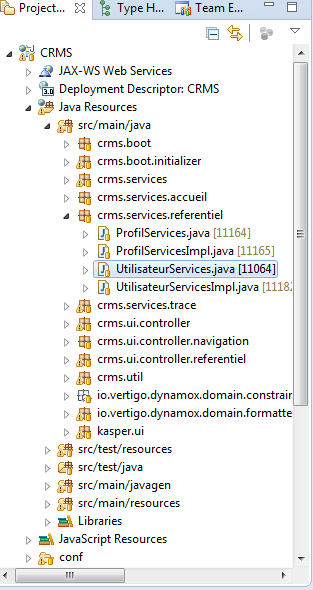


Figure 31 : Location of the service interface which deals with users

#### Content

The interface is an ordinary interface; it should only contain the definitions of the methods.

For the moment, the example interface “UtilisateurServices” only contains the method that will search users:

|  |
| --- |
| **public** **interface** UtilisateurServices **extends** Service {  DtList<UtilisateurResult> loadUtilisateurResultListByCriteria(UtilisateurCriteria criteria);  } |

### Implementing the interface

Implementing a service interface consists in creating a Java class that will implement it. The logic of each method declared in the interface will have to be written to apply/check the business rules described in the user requirements and interact with the database if necessary.

#### Location

The class implementing a service interface should be placed in the same location as the interface. The name of the class should respect the following format:

|  |
| --- |
| <Name of the implemented interface>Impl |

In the current example, the implemented class is UtilisateurServices, which is located in the Java package com.services.referentiel. As a result, it will be named UtilisateurServicesImpl and also added in the package com.services.referentiel.

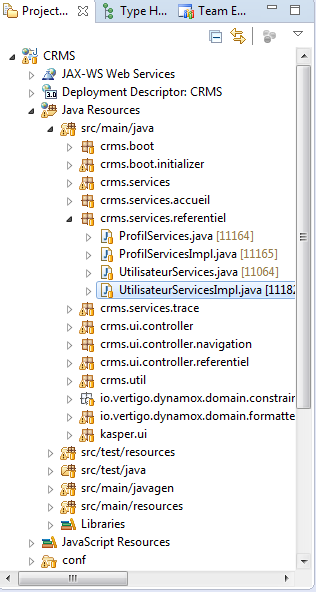


Figure 32 : Location of the implementation of the service interface dealing with users

#### Contents

The class implementing a service interface typically contains:

* A @Transactional annotation to make transactional calls to the database.
* The implementation of the methods declared in the interface.
* References to Data Access Objects (DAO) which are generated Java classes that allow basic create/update/delete/select operations in tables.
* References to Package Access Objects (PAO) which are generated Java classes that contain method to execute custom SQL queries against the database.

##### @Transactional annotation

Each service implementation class MUST have a @Transactional annotation; otherwise the database calls will not be transactional.

|  |
| --- |
| @Transactional  **public** **class** UtilisateurServicesImpl **implements** UtilisateurServices {  // Class implementation  } |

##### Methods implementation

Each method defined in the interface implemented by the class is coded. This is where all the code regarding the business rules should be.

Here, the implementation of UtilisateurServices. loadUtilisateurResultListByCriteria() is straightforward. There are no business rules to apply or check, the only thing to do is to pass the parameter representing the search criteria to the package access object reference:

|  |
| --- |
| **public** DtList<UtilisateurResult> loadUtilisateurResultListByCriteria(**final** UtilisateurCriteria criteria) {  **return** utilisateurPAO.loadUtilisateurResultListByCriteria(criteria);  } |

##### Data Access Object references

If the service implementation class performs simple operations in the database (i.e. inserting/selecting/deleting rows in a single table), then it should contain Data Access Object references as class attributes (see section Data Access Objects to know more about these objects).

A Data Access Object reference attribute is a simple Java class attribute annotated with the @Inject attribute. Example:

|  |
| --- |
| @Inject  **private** UtilisateurDAO utilisateurDAO; |

##### Package Access Object references

When the service implementation class must perform operations in the database that cannot be done using Data Access Object, such as executing custom SQL queries, it should contain Package Access Object references as class attributes (see section Package Access Objects to know how to create such objects).

A Package Access Object reference attribute is a simple Java class attribute annotated with the @Inject attribute. Example:

|  |
| --- |
| @Inject  **private** UtilisateurPAO utilisateurPAO; |

### Naming conventions

In order to have homogeneous code, service should be named according to what they do. The following table describes this convention.

|  |  |  |
| --- | --- | --- |
| Service description | Name | Example |
| Load a list of objects without criteria. | load<Type of the returned objects>List | DtList<Article> loadArticleList(); |
| Load a list of objects corresponding to particular criteria (the criteria is an object) | load<Type of the returned objects>ListByCriteria | DtList<UtilisateurResult> loadUtilisateurResultListByCriteria(UtilisateurCriteria criteria); |
| Load a list of objects corresponding to a criterion (the criterion is the id of an object). | Load<Type of the returned objects>ListBy<Type of the object whose id is the parameter> | DtList<TauxTva> loadTauxTvaListByRegimeTva(long rtvId); |
| Load an object by its id. | Load<Type of the returned object> | loadProfil(long proId); |

### Error messages

#### Database constraints messages

The database is secured with unique constraints that prevent the presence of several rows with the same set of values.

Here is an example of a unique constraint that prevents two articles to have the same code:

|  |
| --- |
| ALTER TABLE ARTICLE ADD CONSTRAINT ARTICLE\_CODE\_UNIQUE UNIQUE (CODE); |

To associate an error message to this constraint, add an entry in the text file named “SqlConstraintMessages.properties”. The name of the entry should be the name of the SQL constraint and its value should be the message to display in the web application if the constraint is violated:

|  |
| --- |
| ARTICLE\_CODE\_UNIQUE=Le code de l'article doit être unique |

### Miscellaneous messages

Inside a service implementation, when a business rule is violated and if an error must be displayed, throw a VUserException to interrupt the process and show the message:

|  |
| --- |
| **if** (roleList.size() == 0) {  **throw** **new** VUserException(**new** MessageText(Resources.***RG\_REF\_PRF\_02\_C***));  } |

The constructor of VUserException takes a MessageText as a parameter. The constructor of MessageText expects MessageKey which identifies the text resource containing the error message.

“Resources” is a Java enumeration that implements the MessageKey interface and that should be located in the same package as the service implementation (if it is not there, create it). The “Resources” enumeration must correspond to a text file named “Resources.properties” located in the same package, that contains the actual error message (here again, create it if it does not exist). Each entry in the “Resources.properties” file should have its corresponding value in the “Resources” Java enumeration.

Example of an entry in “Resources.properties”:

|  |
| --- |
| RG\_REF\_PRF\_02\_C=Au moins un rôle est nécessaire à la création/modification d'un profil |

The corresponding value in the “Resources” Java enumeration:

|  |
| --- |
| **public** **enum** Resources **implements** MessageKey {  /\*\*  \* Au moins un rôle est nécessaire à la création/modification d'un profil.  \*/  ***RG\_REF\_PRF\_02\_C***,  } |

### Collection management (sorting, filtering)

The Vertigo framework comes with tools to manipulate collections and perform actions such as sorting or filtering.

First, inject the CollectionsManager interface in your service implementation class.

|  |
| --- |
| @Inject  **private** CollectionsManager collections; |

Then, to handle DtLists, create a DtListProcessor by calling the method CollectionsManager.createDtListProcessor() .

|  |
| --- |
| **final** DtListProcessor dtListProcessor = collections.createDtListProcessor(); |

The DtListProcessor interface offers methods such as sort, filter, etc.

#### Sorting

The method DtListProcessor.sort() allows to sort a list of DtObjects by a particular field. For instance, to sort a list of TauxTva by starting date (“Date de début de validité”), call sort() on the DtListProcessor to setup the sorting parameters (which field, ascending or descending order, how to handle null values, ignore case or not) and apply the sort by calling apply().

|  |
| --- |
| **final** DtList<TauxTva> sortedList = collections.createDtListProcessor() //  .sort(DtDefinitions.TauxTvaFields.***DATE\_DEBUT\_VALIDITE***.name(), **false**, **false**, **false**) //  .apply(tauxTvaList); |

The sorts can be chained by calling sort() multiple times. The code below illustrates how to sort a list of TauxTva by starting date (“date de début de validité”), ending date (“date de fin de validité”) and rate (“taux”):

|  |
| --- |
| **final** DtList<TauxTva> sortedList = collections.createDtListProcessor() //  .sort(DtDefinitions.TauxTvaFields.***DATE\_DEBUT\_VALIDITE***.name(), **false**, **false**, **false**) //  .sort(DtDefinitions.TauxTvaFields.***DATE\_FIN\_VALIDITE***.name(), **false**, **false**, **false**) //  .sort(DtDefinitions.TauxTvaFields.***TAUX***.name(), **false**, **false**, **false**) //  .apply(tauxTvaList); |

## Dates

### DateUtil

The DateUtil helper class provides static methods to handle dates:

|  |  |
| --- | --- |
| Method | Description |
| newDate() | Returns the current date (without the time component). |
| newDateTime() | Returns the current date and time. |
| daysBetween(Date, Date) | Returns the number of days from the first date parameter to the second. |
| compareDate(Date, Date) | Compares two dates (without time components).  Returns :   * 0 if the dates are equal. * A negative value if the first date is earlier than the second. * A positive value if the first date is later than the second. |
| compareDateTime(Date, Date) | Compares two dates (with time components).  Returns :   * 0 if the dates are equal. * A negative value if the first date is earlier than the second.   A positive value if the first date is later than the second. |
| parse(String, String) | Parses a String value (the first parameter) as a date using a given pattern (the second parameter). |

### DateBuilder

The DateBuilder class provides a way to build dates from a starting date by adding days, months, years, etc. to it.

|  |  |
| --- | --- |
| Method | Description |
| DateBuilder(date) | Constructor. The parameter is the base date that will be used to perform the operations. The date itself will not be modified. |
| build() | Truncate the time component of the date. |
| addSeconds(int) | Add seconds to the date and return the current builder. |
| addMinutes(int) | Add minutes to the date and return the current builder. |
| addHours(int) | Add hours to the date and return the current builder. |
| addDays(int) | Add days to the date and return the current builder. |
| addWeeks(int) | Add weeks to the date and return the current builder. |
| addMonths(int) | Add months to the date and return the current builder. |
| addYears(int) | Add years to the date and return the current builder. |
| toDateTime() | Returns the date of the builder (which is the result of all the operations performed on the date passed as a parameter of the constructor). |

The code below shows how to build a date that is one month earlier than today’s date:

|  |
| --- |
| **final** Date aujourdhui = DateUtil.newDate();  **final** Date dateDebut = **new** DateBuilder(aujourdhui).addMonths(-1).toDateTime(); |

## Data Access Objects

A Data Access Object class is automatically generated by the Ant script build-mda.xml for every class defined the Persistent model.

Each Data Access Object class’s name respects the format <Persistent class name>DAO.java and offers methods to perform basic operations in the class’s corresponding table in the database:

* get(id): load the row whose primary key is equal to id.
* save(dto): insert or update a row.
* getList(criteria, maxRows): load all the rows corresponding to the criteria object. Use parameter maxRows to limit the number of results.

Data Access Objects are to be used in Services only, which means that they should not be called in a page action class.

Data Access Objects do not allow more complex operations in the database, such as selecting data from several tables using joins, or updating rows depending on complex conditions. See Package Access Objects to learn how to execute custom SQL queries.

## Package Access Objects

### Description

Package Access Objects are Java classes that allow the developer to execute their own SQL queries when the simple operations provided by Data Access Objects are not enough.

Each Package Access Object class is automatically generated by the Ant script build-mda.xml, based on the content of a text file whose extension is .ksp.

A Package Access Object class should contain methods that are functionally coherent (relatively to the user requirements), which means that it should contain methods that deal with the same functional module of the application or with the same functional entity.

Here are the steps to create a Package Access Object class that executes custom SQL queries:

1. Create the KSP file that will contain the definition of the task to execute the custom SQL and reference this KSP file in the “execution.kpr” file (otherwise it will not be considered by the model generator). Add the task definition in the KSP file.
2. Call the model generator.

### Creating a PAO object

As an example, let us create a Package Access Object that will offer methods to interact with the table containing the users (“UTILISATEUR”).

#### KSP file

##### Location

The KSP file for a Package Access Object should be added in the folder **/src/main/java/<application alias>/<functional module>**.

In our example, the application alias is “crms” and the user entity (“Utilisateur”) is in the module named “Référentiel” (according to the user requirements), so the KSP file is added in folder **/src/main/java/crms/referentiel**.

The name of the KSP file should respect the following naming convention:

|  |
| --- |
| <name of the concerned functional module or entity >Pao.ksp |

Here, the Package Access Object will interact with the table UTILISATEUR, corresponding to the Java class Utilisateur, so its name will be: utilisateurPao.ksp.

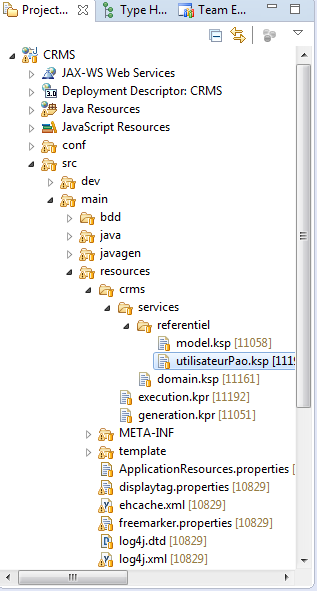


Figure 33 : Location of the KSP file for the Package Access Object dealing with users

##### Content

###### Package declaration

The first line of the KSP file should be the declaration of the package. This will define:

* Where the Java class corresponding to the KSP file will be generated.
* The name of the Java class.

The syntax of this definition is identical to the Java package definition syntax.

In the current example, the first line of the “utilisateurPao.ksp” file located in /src/main/resources/crms/services/referentiel/ is:

|  |
| --- |
| package crms.pao.referentiel.utilisateur |

As a result, the PAO Java class will be named UtilisateurPAO and added in the package crms.pao.referentiel.utilisateur.

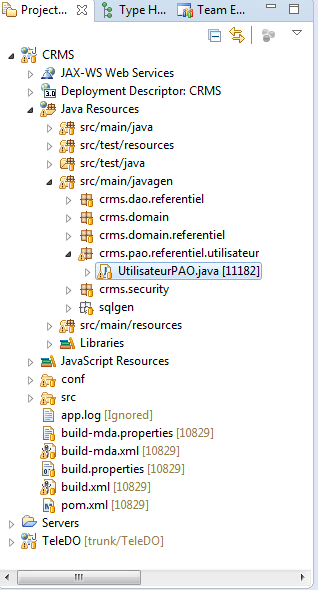


Figure 34 : Location of the generated PAO class

###### Task definition

A task is basically a custom SQL query with input and output parameters. It is defined in the KSP file using a specific syntax. Each task defined in a KSP file will generate a method in the Java class corresponding to the KSP file.

A task is defined by:

* Its name.
* A class name that is determined by the type of SQL query the task executes (select, insert, update, delete).
* The request: the SQL statement to execute.
* Attributes: the input and output parameters of the SQL query.

The declaration of a task is structured as follows:

|  |
| --- |
| create Task <Name> {  className : "<Class name>",  request : "<Request>",  <Attribute 1>,  <Attribute 2>,  ...  <Attribute N>  } |

As an example, the task defined below uses an input parameter as a criterion to execute a select query and returns the list of results as an output parameter:

|  |
| --- |
| create Task TK\_LOAD\_UTILISATEUR\_RESULT\_LIST\_BY\_CRITERIA {  className : "io.vertigo.dynamox.task.TaskEngineSelect",  request : "  SELECT  uti.UTI\_ID,  uti.PRENOM || ' ' || uti.NOM as NOM\_COMPLET,  uti.EMAIL,  uti.IS\_ACTIF  FROM  UTILISATEUR uti  WHERE  1 = 1  <%if (criteria.getCivCode() != null){%>  and uti.CIV\_CODE = #CRITERIA.CIV\_CODE#  <%}%>  <%if (criteria.getNom() != null){%>  and uti.NOM like '%%' || #CRITERIA.NOM# || '%%'  <%}%>  <%if (criteria.getPrenom() != null){%>  and uti.PRENOM like '%%' || #CRITERIA.PRENOM# || '%%'  <%}%>  <%if (criteria.getIsActif() != null){%>  and uti.IS\_ACTIF = #CRITERIA.IS\_ACTIF#  <%}%>  ",  attribute CRITERIA {  domain : DO\_DT\_UTILISATEUR\_CRITERIA\_DTO,  notNull:"true",  inOut :"in"  },  attribute DTC\_UTILISATEUR\_RESULT {  domain : DO\_DT\_UTILISATEUR\_RESULT\_DTC,  notNull:"true",  inOut :"out"  }  } |

Name

The name of a task must begin by the prefix “TK\_”. Use capital letters and separate terms with the underscore character ‘\_’. What follows the prefix will be used to name the method that will be generated in the resulting PAO Java class.

In the current example, the task named TK\_LOAD\_UTILISATEUR\_RESULT\_LIST\_BY\_CRITERIA in the KSP file will generate a Java method named loadUtilisateurResultListByCriteria.

The name of a task should reflect what it does. Here is the format of the name of common tasks:

|  |  |
| --- | --- |
| Task | Name format |
| Search objects by criteria. | TK\_LOAD\_<TYPE OF THE RETURNED OBJECTS>\_LIST\_BY\_CRITERIA |
| Load one object. | TK\_LOAD\_<TYPE OF THE RETURNED OBJECT> |
| Update several objects. | TK\_UPDATE\_<TYPE OF THE UPDATED TABLE>\_LIST |
| Update one object. | TK\_UPDATE\_<TYPE OF THE UPDATED TABLE> |

Class name

The class name depends on the type of SQL operation the task will perform:

|  |  |
| --- | --- |
| Type of SQL query | Class name |
| Select | io.vertigo.dynamox.task.TaskEngineSelect |
| Update | io.vertigo.dynamox.task.TaskEngineProc |
| Insert | io.vertigo.dynamox.task.TaskEngineProc |
| Delete | io.vertigo.dynamox.task.TaskEngineProc |

Request

The request is the SQL code that will be executed by the task. The SQL code may contain Java code between <% %> markups. This specific syntax allows the developers to make some parts of the SQL query conditional.

Accessing the value of an attribute

To access the value of an attribute in the SQL code, surround it with ‘#’ characters.

For example, a task has an attribute named SEARCHED\_TEXT and this is how it is used in its request:

|  |
| --- |
| SELECT  \*  FROM  UTILISATEUR uti  WHERE  uti.NOM = #SEARCHED\_TEXT# |

**Remark:** it is not possible to use the same attribute several times in a request.

Conditional request

Sometimes, some parts of the SQL code should be used only in certain conditions. Surrounding these parts of the query with Java code can make them conditional, which means that they will appear in the actual SQL query only if some conditions are satisfied.

For example, a task has an attribute named “CRITERIA” of type “UtilisateurCriteria” (so it is translated to DO\_DT\_UTILISATEUR\_CRITERIA\_DTO in the KSP syntax):

|  |
| --- |
| attribute CRITERIA {  domain : DO\_DT\_UTILISATEUR\_CRITERIA\_DTO,  notNull:"true",  inOut :"in"  } |

The Java class UtilisateurCriteria contains an attribute named “nom” and the getter method to access it is named “getNom()”.

In the SQL query of the task, we want to select rows from the table UTILISATEUR, and filter these rows using the attribute “nom” of the attribute “CRITERIA” if it is not null. This is done like this in the KSP file:

|  |
| --- |
| SELECT  \*  FROM  UTILISATEUR uti  WHERE  1 = 1  <%if (criteria.getNom() != null) {%>  and uti.NOM like '%%' || #CRITERIA.NOM# || '%%'  <%}%> |

At runtime, thanks to this syntax, the request code will be parsed and evaluated by the framework so that the actual query that will be executed against the database will depend on the value of criteria.getNom():

|  |  |
| --- | --- |
| null | != null |
| SELECT  \*  FROM  UTILISATEUR uti  WHERE  1 = 1 | SELECT  \*  FROM  UTILISATEUR uti  WHERE  1 = 1  and uti.NOM like '%%' || #CRITERIA.NOM# || '%%' |

Attributes

A task attribute is defined by:

* Its name.
* Its domain.
* The fact that it is nullable or not.
* Its type (IN or OUT).

|  |
| --- |
| attribute <Name> {  domain : "<Domain>",  notNull:"<true|false>",  inOut :"<in|out>"  } |

Name

The name of an attributes is used in the Request to access its value.

Domain

The domain of an attribute depends of its type.

|  |  |
| --- | --- |
| Type of the attribute | Domain of the attribute |
| Scalar value (integer, string, boolean, etc.) | One of the domains defined in the “domain.ksp” file (see section Domains). |
| Instance of a Java class | DTO\_DT\_<TYPE OF THE JAVA CLASS>\_DTO.  Example: if the class is UtilisateurCriteria, the domain will be DO\_DT\_UTILISATEUR\_CRITERIA\_DTO |
| Collection of instances of a Java class | DO\_DT\_<TYPE OF THE JAVA CLASS>\_DTC.  Example: if the class is UtilisateurResult, the domain will be  DO\_DT\_UTILISATEUR\_RESULT\_DTC |

Not null

If the attribute can be equal to null, set the property “notNull” to “false”.

If the attribute cannot be null, set the property “notNull” to “true”.

In or out

If the attribute is an input parameter of the task, set the property “inOut” to “in”.

If the attribute is an output parameter of the task, set the property “inOut” to “out”.

#### Reference in “execution.kpr”

Each KSP file should be referenced in the “execution.kpr” file of the project otherwise it will not be taken into account and the PAO Java class defined in it will not be generated by the model generator.

The file “execution.kpr” is located in the directory **/src/main/resources/<project alias>/**. Make sure it contains the relative path of the KSP file containing the definition of your Package Access Object.

In our example, “execution.kpr” should contain the following line:

|  |
| --- |
| services/referentiel/utilisateurPao.ksp |

#### Class generation

To generate the PAO class defined in the KSP file, run the Ant file named “build-mda.xml”.

After the build completes, the Java class appears in the folder corresponding to the package indicated in the KSP file.

In our example, the class UtilisateurPAO will be created in folder src/main/javagen/crms/pao/referential/utilisateur.

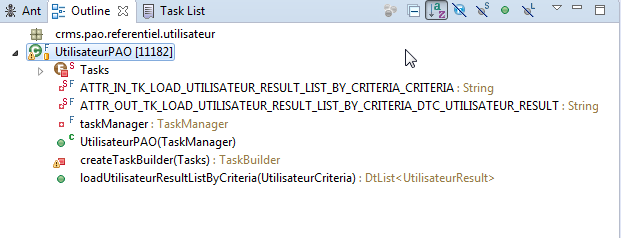


Figure 35 : Generated PAO class

## Testing

To guarantee the quality of the application, we must test our services to verify that they do not crash and that their business rules do what they are expected to do.

### Test classes

Services are tested using the JUnit framework.

|  |
| --- |
| Each method declared in a service should have its own test class. |

#### Location

The test classes are found in a Java package named **<application alias>.services.<functional module>** inside the directory **/src/test/java/**.

For example, the test class for the method saveRegimeTva() declared in the interface RegimeTvaServices (located in the package crms.services.referentiel) should be added into the package crms.services.referentiel, in folder /src/test/java.

The name of each test class must be named as follows:

|  |
| --- |
| <Capitalized named of the tested method>Test |

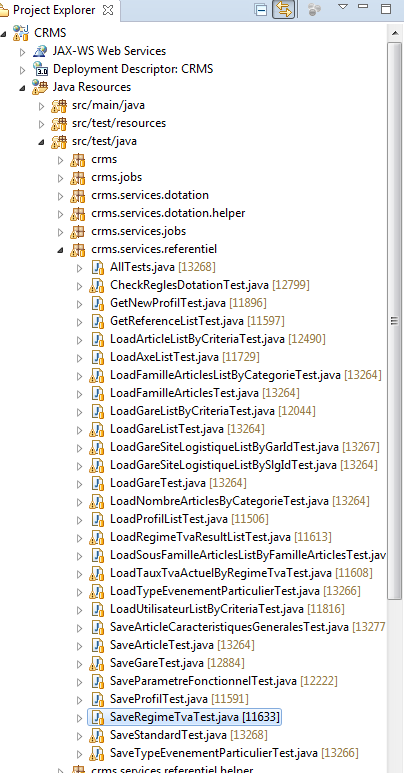


Figure 36 : Location of a test class

#### Parent test class

Tests are run against an empty database. The data is created during the test and everything is roll backed at the end of the test. This behavior is ensured by an abstract test class that rolls back the transaction automatically, regardless of the result of the test.

In project CRMS, this class is AbstractCrmsTestCase. Every test class must extend this class.

|  |
| --- |
| **public** **class** SaveRegimeTvaTest **extends** AbstractCrmsTestCase {  //...  } |

#### Content

TODO

### Test data

Services need data to be tested. To create this data we use helper classes that contain the methods to create the objects we need.

TODO