The Usecase Framework

Table of contents

1. Introduction

A usecase in Lenya means a user triggered action. In most cases, a usecase is triggered by a CMS menu option on a specific document of the publication. This document is the object of the usecases' action (such as edit, delete, publish, ...).

There are usecases which are independent of a specific document, such as the `ac.logout` usecase. In that case it does not matter on what document the usecase is triggered. The part of the request which specifies the document is simply ignored by usecases that are document independent.

The CMS menus trigger usecases by setting the `lenya.usecase` request parameter on the current document. If for example the user selects the *Publish* option from the *Workflow* menu, a request will be triggered such as: `GET http://www.server.com/lenya/default/authoring/tutorial.html?&lenya.usecase=publish`

The Lenya `global-sitemap.xmap` will redirect requests with a `lenya.usecase` request parameter to the `$LENYA_WEBAPP/lenya/usecase.xmap` sub-sitemap. From version 1.4 on, the following pipeline in this sitemap is used to recognonize usecases which are implemented in Java using the new 1.4 usecase framework:

```
<map:pipeline>
  <map:match type="registered-usecase">
    <map:mount src="usecases/usecase.xmap" uri-prefix="" check-reload="yes" reload-method="synchron"/>
  </map:match>
</map:pipeline>
```

The `registered-usecase` matcher's default implementation (`org.apache.lenya.cms.cocoon.matching.UsecaseRegistrationMatcher`) will use the Avalon component resolver mechanism to resolve the name of the usecase to a an Avalon component. In case it cannot resolve the usecase to an Avalon component, sitemap processing will continue and the usecase is treated in the traditional way using the `usecase` and `step` matchers (`org.apache.cocoon.matching.WildcardRequestParameterMatcher`). In order for this to work correctly, there should be a `lenya.step` parameter in the request.

If the usecase could be resolved successfully into an Avalon component, processing will continue in the `$LENYA_WEBAPP/lenya/usecases/usecase.xmap` (as opposed to `$LENYA_WEBAPP/lenya/usecase.xmap`) with the new JX and Java based 1.4 usecase framework.

The *usecase framework* in Lenya 1.4 is a simple framework to implement usecases using JX templates and Java. This approach is an "85% solution". It enables the user to implement a big range of common usecases.

Note:
Some special complex usecases might require a custom flowscript, in this case you can't use this framework.

2. Directory Structure

2.1. The Lenya Core

```
$LENYA_WEBAPP
  /lenya/usecases                    usecase-related files
                /usecase.xmap        usecase dispatching sitemap
                /usecases.js         flowscript for usecase control flow
                /admin               Lenya admin usecases
                      /addUser.jx    JX templates for usecase views
                ...                  more Lenya core usecases
```

2.2. Your Publication

```
$PUB_HOME
```

```
/lenya/usecases                    usecase-related files
              /editHeadline.jx     JX templates for usecase views
/java/src/...                      usecase handler classes
```

3. Architecture

A usecase request - denoted by the request parameter `lenya.usecase` - is dispatched by `$LENYA_WEBAPP/lenya/usecases/usecase.xmap`. All usecases are handled by a single flowscript `$LENYA_WEBAPP/lenya/usecases/usecases.js`. This keeps javascript maintenance costs at a minimum.

The flowscript `usecases.js` determines the usecase handler class using the `org.apache.lenya.cms.usecase.UsecaseResolver`. All business code operations are delegated to the usecase handler class.

4. The Contract Between Flowscript And Usecase Handler

The usecase handler class has to implement the interface `org.apache.lenya.cms.usecase.Usecase`. The methods of this interface are called in a certain order when the usecase is invoked:

1. `setSourceURL(String sourceUrl)`
   `setName(String)`

   Initialize the handler.
2. `Passing request parameters`

   In the next step, all request parameters are passed as parameters to the usecase, except the following reserved parameters:

   - `lenya.usecase`
   - `lenya.continutation`
   - `submit`

   If the request parameter value is a string, the value can be accessed inside the usecase handler using `getParameterAsString(name)`. If the request parameter is a part of a multipart request, e.g., in a file upload form, it will be available using `getPart(name)`.
3. `checkPreconditions()`

   This method is called to check the pre-conditions of the usecase. The pre-conditions are checked before the usecase is started, i.e., before the first screen is presented to the user. To denote that a condition does not comply, add an appropriate error message using `addErrorMessage(String)`. If an error message was added, the usecase is not started. Alternatively, you can provide information to the user using `addInfoMessage(String)`. This doesn't prevent the usecase from being executed.
4. `lockInvolvedObjects()`

   This method is called to lock all objects which could be changed during the usecase.

The following methods are called in a loop while the user interacts with the usecase, until a request parameter with the name `submit` was sent.

1. `getView()`

   Requests the next view to be displayed. The view may be `null` if no screen should be presented to the user.
2. `advance()`

   This method is called to advance the usecase after the a user interaction. In contrast to `execute()`, this method is not called when the `<input type="submit" name="submit">` was pressed, but for every other submitting of the form. A typical usecase is the *multiple forms editor* where `advance()` is used to update the document when the user switched to another element.

When the form is submitted using the `<input type="submit" name="submit">` button, the usecase is finished:

1. `checkExecutionConditions()`

   This method is called before the usecase is actually executed. A typical example is the validation of form data.
2. `execute()`

   This method actually executes the final step of the usecase.

When the form is submitted using the `<input type="submit" name="cancel">` button, the usecase is cancelled:

1. `cancel()`

   This method cancels the usecase. The transaction is rolled back.

5. Implementing a Custom Usecase

5.1. Prerequisites

1. Choose a name to identify the usecase, e.g. editHeadline. It is possible to group usecases using "." as delimiter, for instance `article.editHeadline`.

5.2. Add a Menu Item

Note:

This step is necessary if you want to call the usecase from the Lenya menubar.

1. Add the corresponding menu item:

```
<item uc:usecase="article.editHeadline">Edit Headline</item>
```

5.3. Implement the Usecase Handler Class

1. Choose a name for your business logic class, e.g. `org.myproject.lenya.usecases.EditHeadline`.
2. The class must implement the interface `org.apache.lenya.cms.usecase.Usecase`.
3. To simplify development, you can extend one of the following classes:
   - `org.apache.lenya.cms.usecase.AbstractUsecase`
   - `org.apache.lenya.cms.usecase.DocumentUsecase` (only for usecases invoked on document pages)
   - `org.apache.lenya.cms.usecase.SiteUsecase`

   They have built-in support for the unit-of-work pattern (which will evolve into an ACID transaction someday) as well as functionality specific to the area they are supposed to be used with, e.g. the site area.
4. Add the usecase handler class declaration to an XPatch file, e.g. `$PUB_HOME/config/cocoon-xconf/usecases.xconf`:

```
<xconf xpath="/cocoon/usecases"
       unless="/cocoon/usecases/component-instance[@name = 'article.editHeadline']">
  <component-instance name="article.editHeadline"
                      logger="lenya.usecases.editHeadline"
                      class="org.myproject.lenya.usecases.EditHeadline"/>
</xconf>
```

5.4. Implement the View

The view of a usecase is optional. If you omit the view declaration, no screen is presented to the user. The view is declared in the usecase configuration:

```
  <component-instance ...>
    <view template="usecases/article/editHeadline.jx" menu="false">
      <parameter name="title" value="Edit Headline"/>
      <parameter name="..." value="..."/>
    </view>
  </component-instance>
```

The `<view>` element takes an optional `menu` attribute which denotes if the menubar should be visible when the usecase

screen is presented. If omitted, it defaults to `false`.

The `<view>` element can contain an arbitrary number of `<parameter>` elements, each containing a `name` and `value` attribute. These parameters can be accessed in the JX template using `${usecase.getView().getParameter('...')}`.

One option is to implement view for a usecase as a JX template. The location of the JX template is defined using the `<view>` element's `template` attribute (relatively to the `context://lenya` directory). The output of the template has to be a Lenya page:

```
<page:page
  xmlns:jx="http://apache.org/cocoon/templates/jx/1.0"
  xmlns:page="http://apache.org/cocoon/lenya/cms-page/1.0"
  xmlns="http://www.w3.org/1999/xhtml"
  xmlns:i18n="http://apache.org/cocoon/i18n/2.1"
  >

  <page:title>
    <i18n:text><jx:out value="${usecase.getView().getParameter('title')}"/></i18n:text>
  </page:title>
  <page:body>

    <form>
      <input type="hidden" name="lenya.continuation" value="${continuation.id}"/>
      <input type="hidden" name="lenya.usecase" value="${usecase.getName()}"/>

      ...

    </form>

  </page:body>
</page:page>
```

Take care of adding the hidden `lenya.usecase` and `lenya.continuation` fields as shown above.

If you don't want to display a form, but output arbitrary XML, text, an SVG-generated image or something like that, you can use a `uri` attribute instead of the `template` attribute. In this case, the `menu` attribute can be omitted.

```
  <component-instance ...>
    <view uri="cocoon://modules/reports/generateReport.pdf" />
  </component-instance>
```

5.5. Displaying Usecases in Tabs

It is possible to use a tab-based layout to assemble a set of usecases. The admin and site areas are displayed using this style. To add a tab-based usecase to a tab set, two steps are required:

1. Add the tab to the GUI manager configuration:

```
<xconf xpath="/cocoon/gui-manager/tab-group[@name = 'admin']"
       unless="/cocoon/gui-manager/tab-group[@name = 'admin']/tab[@name = 'search']">

  <tab name="search" label="Search" usecase="admin.search"/>

</xconf>
```

2. Add the tab configuration to the usecase view declaration:

```
<component-instance name="admin.search" logger="lenya.admin"
class="org.apache.lenya.cms.usecase.DummyUsecase">
  <view template="usecases/admin/search.jx" menu="true">
    <tab group="admin" name="search"/>
  </view>
  <exit usecase="admin.search"/>
</component-instance>
```

5.6. Dynamically Setting the Exit URL

Like you see above you can specify the "exit" URL of the usecase via the component configuration, however this is sometimes

not enough because you may need to set the exit URL of a usecase dynamically.

Consider the use case is that you have a form and you need two possibilities to exit the usecase.

1. Pressing on "submit" will save and exit to the same page in the authoring area.
2. The other exit point could be a dynamic link based on a `<a href=""/>` where the user get redirected after submitting the form (e.g. to edit the linked document with the BXE (../../../../docs/1_2_x/components/editors/bxe.html) editor).

First you need to add another request parameter (e.g., `TRANSFER_FIELD`) denoting that the user clicked the link. This could be e.g. a hidden field which contains the URL to transfer the request. Further you need to override `Usecase.getTargetURL()` to return the BXE usecase URL if the `TRANSFER_FIELD` parameter is set, and `super.getTargetURL()` otherwise.

```
public String getTargetURL(boolean success) {
    String tmpTransfer = getParameterAsString("TRANSFER_FIELD", null);
    if (!tmpTransfer.equals("") & tmpTransfer != null)
        return tmpTransfer;
    else
        return super.getTargetURL(success);
}
```

6. Overriding Core Usecases in Publications

6.1. Overriding Usecase Handler Classes

The usecase resolver, which is responsible for obtaining the handler class for a usecase, looks first if the current publication overrides the core usecase handler. This can be done by declaring a usecase called `<pub-id>/<usecase-name>`, for instance `mypub/admin.addUser`. To implement a core usecase using a custom handler class, you need to

1. Implement the handler class and put it in `$PUB_HOME/java/src`. In most cases, you will extend the core usecase handler class to inherit the basic functionality.
2. Declare it in an *xpatch* file, for instance `$PUB_HOME/config/cocoon-xconf/usecases.xconf`:

```
<xconf xpath="/cocoon/usecases" unless="/cocoon/usecases/component-instance[@name =
'mypub/admin.addUser']">
  <component-instance name="mypub/admin.addUser"
                      logger="lenya.usecases.editHeadline"
                      class="org.myproject.lenya.usecases.AddUser"/>
</xconf>
```

Now, when the usecase is invoked from inside the publication mypub, the custom handler class will be used.

6.2. Overriding JX Templates

Overriding the JX template of a usecase follows the publication templating (../publication-templating/index.html) principle. You just have to put a JX template with the same name in `$PUB_HOME/lenya/usecases`, for instance `$PUB_HOME/lenya/usecases/admin/addUser.jx`.