GenericEditorAPI

Status: **DRAFT**

This page describes the Lenya editor API as it slowly evolves into shape. It might eventually contain all the information you need to make an editor of your choice work with Lenya. Currently, it is incomplete, since it omits the Java usecase handler side of things.

Quick start: how to make your editor work with Lenya

1. Create a module named after your editor (check our modules documentation).
2. Create a Java usecase handler, either by
    a. stealing the code from the editors module, putting it in $YOURMODULE/java and applying necessary changes *(current practice, BAD!)*, or
    b. using it as it is (then you don't need to copy it) *(good practice, not widely used - if you feel the existing stuff is not versatile enough, abstract your needs and put them upstream!)*. (FIXME: is the code from the editors module the best one we have?)
    For inspiration, read our usecase documentation.
3. Create a usecase view: either steal it from the One-Form-Editor (JXTemplate-based, for an out-of-context form: oneform.jx), or follow the TinyMCE approach for an in-place WYSIWYG editor (pipeline-based, check out the sitemap (`<match pattern="tinymce.edit">`) and the XSLT! See below under #WYSIWYG).
4. Register your editor usecase by writing an xpatch file in `$YOURMODULE/config/cocoon-xconf/usecase-edit.xconf`:

```
<xconf xpath="/cocoon/usecases" unless="/cocoon/usecases/component-instance[@name = '$YOURMODULE.edit']">
   <component-instance name="$YOURMODULE.edit" logger="lenya.publication"
     class="org.apache.lenya.cms.$YOURMODULE.YourEditor">
     <transaction policy="pessimistic"/>
     <!-- either the JXTemplate approach: -->
     <view template="modules/$YOURMODULE/usecases/$YOURMODULE.jx" menu="false"/>
     <!-- or the pipeline approach: -->
     <view uri="cocoon://modules/$YOURMODULE/$YOURMODULE.edit" menu="false"/>
     <!-- but not both :-D -->
   </component-instance>
 </xconf>
```

1.#5 Make sure that your view adds the following javascript snippets:

```
<script type="text/javascript" src="/modules/editors/javascript/org.apache.lenya.editors.js"/>
<script type="text/javascript" src="/modules/$YOURMODULE/javascript/youreditor_lenya_glue.js"/>
```

The first is the "API" file that defines the interaction between the usecase and your editor window, and it contains some utility functions you can (should) use for your window management (see org.apache.lenya.editors.js). The second is a file where you place all lenya-specific javascript code. 1.#6 Create some hooks in your editor:

a. "Insert Lenya Link",
b. "Insert Lenya Image", and
c. "Insert Lenya Asset",
either as new buttons, or hooked up as file browsers to some existing dialog.
They should trigger functions in `youreditor_lenya_glue.js` where you can put the code that takes preset values from your editor and opens a usecase window pre-filled with those values (example: oneform_lenya_glue.js)

```
// These arrays are indexed by window name:
var objectData = new Array(); // For each triggered usecase: store its dataset.
var usecaseMap = new Array(); // For each opened window, store the usecase name.

// This is a callback you need to implement:
// The usecase will call it when the user submits the form.
org.apache.lenya.editors.setObjectData = function(objectData, windowName) {
  var currentUsecase = usecaseMap[windowName];
  var snippet = org.apache.lenya.editors.generateContentSnippet(currentUsecase, objectData);
  org.apache.lenya.editors.insertContent(
    document.forms['oneform'].elements['content'],
    snippet
  );
  usecaseMap[windowName] = undefined; // we're done!
  objectData[windowName] = undefined; // we're done!
}

// This is another callback you need to implement:
// The usecase will call it on loading to get its default data.
org.apache.lenya.editors.getObjectData = function(windowName) {
  return objectData[windowName];
}

// This function is called by the editor buttons you created:
function triggerUsecase(usecase) {
  var windowName = org.apache.lenya.editors.generateUniqueWindowName(); // a convenience function
  // Now you need to do clever, editor-specific things to present useful defaults if your user has
  // selected something. For getting at selected text, check out the following utility function.
  // But maybe you can do even more: getting a DOM snippet and parsing attributes, for instance.
  var selectedText = org.apache.lenya.editors.getSelectedText(document.forms[0].elements
['content']);
  switch (usecase) {

    case org.apache.lenya.editors.USECASE_INSERTLINK:
      objectData[windowName] = new org.apache.lenya.editors.ObjectData({
        url   : "", /* Note we're setting the empty string as default, rather than omitting the
field.
                       Undefined fields will cause the usecase to disable the form field (which is
useful
                       if your editor wants to handle part of the settings itself). */
        text  : selectedText,
        title : ""
      });
      break;

    case org.apache.lenya.editors.USECASE_INSERTIMAGE:
      // ...
    case org.apache.lenya.editors.USECASE_INSERTASSET:
      // ...
  }
  // a utility function you should use:
  org.apache.lenya.editors.openUsecaseWindow(usecase, windowName);
  usecaseMap[windowName] = usecase;
    alert("Stored values for new window " + windowName + ":\n"
      + "objectData[windowName] = '" + objectData[windowName] + "'\n"
      + "usecaseMap[windowName] = '" + usecaseMap[windowName] + "'"
  );
}
```

The ObjectData structure is defined by its prototype properties. Check `org.apache.lenya.editors.js`, it's documented.

Using server-side validation

Requirements:

Each editor should be able to handle validation errors from the server gracefully. It must receive the appropriate error messages from the server, display them to the user, and keep the file checked out!

This implies that the transmission of the document data and the receiving of result messages should happen via AJAX. thus it's a task for well after 2.0 is out.

FIXME!

Writing a usecase handler for your editor

FIXME!

Spiffing things up with AJAX

FIXME!

Writing in-place true WYSIWYG editors

If you're keen to provide true WYSIWYG editing, you must delegate the rendering of the page to the publication sitemap, since your editor module cannot know about the correct rendering. This can be accomplished by mounting the publication sitemap into the editor sitemap. You can then take the output of the publication pipeline and apply editor-specific pre-processing (such as adding javascript code or working around editor-specific quirks).

Under no circumstances should you hack any workarounds into global sitemaps or publication resources.

A possible approach (taken from the tinymce module) is this:

```
    <map:pipeline internal-only="yes">

      <!-- when editing, the page should look exactly like the original, and since
           we cannot know anything about the pipelines used for rendering, we must
           delegate the job to the publication's own sitemap. -->

      <map:match pattern="tinymce.delegateToPubSitemap/**">
        <map:mount src="{fallback:{page-envelope:publication-id}:sitemap.xmap}" uri-prefix="tinymce.
delegateToPubSitemap"/>
      </map:match>

      <!-- the usecase framework provides error and info messages. since we bypass
           the jxtemplate view mechanism, we must include them by hand. -->

      <map:match pattern="tinymce.getUsecaseMessages">
        <map:generate type="jx" src="fallback://lenya/modules/usecase/templates/messages.jx"/>
        <map:transform type="i18n">
          <map:param name="locale" value="[request:locale}"/>
        </map:transform>
        <map:transform src="context://lenya/xslt/util/strip_namespaces.xsl"/>
        <map:serialize type="xml"/>
      </map:match>
    </map:pipeline>

    <map:pipeline>

      <!-- this is the view of the usecase (see config/cocoon-xconf/usecases.xconf) -->

      <map:match pattern="tinymce.edit">
        <!-- check if TinyMCE is installed -->
        <map:select type="resource-exists">
          <!-- render page with tinymce inserted -->
          <map:when test="fallback://lenya/modules/tinymce/resources/tinymce/jscripts/tiny_mce/tiny_mce.js">
            <map:aggregate element="tinymceWrapper">
              <map:part src="cocoon:/tinymce.delegateToPubSitemap/authoring{page-envelope:document-url}"/>
              <map:part src="cocoon:/tinymce.getUsecaseMessages"/>
            </map:aggregate>
            <map:transform src="fallback://lenya/modules/tinymce/xslt/page2edit.xsl">
              <map:parameter name="contextPath" value=""/>
              <map:parameter name="continuationId" value="{flow-continuation:id}"/>
              <map:parameter name="usecaseName" value="{request-param:lenya.usecase}"/>
              <map:parameter name="publicationid" value="{page-envelope:publication-id}"/>
            </map:transform>
          </map:when>
          <!-- TinyMCE is not installed - generate info page for the user -->
          <map:otherwise>
            <map:generate src="fallback://lenya/modules/tinymce/resources/misc/download.xml"/>
          </map:otherwise>
        </map:select>
        <map:call resource="style-cms-page"/>
        <map:serialize type="xhtml"/>
      </map:match>

    </map:pipeline>
```

FIXME: This sitemap duplicates the "style-cms-page" resource. It would be great if such resources could be provided just once by a global sitemap and then included, but resources of other sitemaps are not accessible. If anyone has an idea how to factor this out, let me know.