

Plugin Developer Guide

Contents

Extending Oxygen XML Editor with Plugins.....	3
Introduction.....	3
General configuration of an Oxygen XML Editor plugin.....	3
Types of plugins.....	4
General Plugin.....	4
Selection Plugin.....	4
Document Plugin.....	5
Custom Protocol Plugin.....	5
Resource Locking Custom Protocol Plugin.....	5
Components Validation Plugin.....	6
Workspace Access Plugin.....	7
Open Redirect Plugin.....	8
Targeted URL Stream Handler Plugin.....	8
Lock Handler Factory Plugin.....	10
How to.....	10
How to Write a CMS Integration Plugin.....	10
How to Write A Custom Protocol Plugin.....	14
Installation.....	15
Example - A Selection Plugin.....	15

Extending Oxygen XML Editor with Plugins

This chapter explains how to write and install a plugin of the Oxygen XML Editor. It treats only the standalone version, as the Eclipse plugin version can be extended with other plugins following the rules of the Eclipse platform.

Introduction

Oxygen XML Editor defines a couple of extension points to allow providing custom functionality via plugins. The plugin support includes the following types of plugins:

- General plugins
- Selection plugins
- Document plugins
- Custom protocol plugins
- Resource locking custom protocol plugins
- Components validation plugins
- Workspace access plugins
- Open redirect plugins

A selection plugin can be applied to both an XML document and a non-XML document. Other types of plugins can be applied only to XML documents.

A components validation plugin and a workspace access plugin are not connected with one document type, they have access to some resources of the application workspace used by all opened documents.

In order to develop a plugin a Java development environment must be installed. Apart from any library that the specific plugin requires, the file `oxygen.jar` is necessary for plugin compilation. Also an Oxygen XML Editor installation is helpful for testing the deployment and plugin the functionality.

General configuration of an Oxygen XML Editor plugin

The Oxygen XML Editor functionality can be extended with plugins that implement a clearly specified API. A plugin includes at least a descriptor file which is an XML file called `plugin.xml` and two Java classes that extend `ro.sync.exml.plugin.Plugin` and `ro.sync.exml.plugin.PluginExtension`. Most plugins work only in the Text mode of the XML editor panel while others work at the workspace level. For extending the Author mode of the XML editor panel, see *Author Developer Guide* for the custom Author actions API.

On the Oxygen XML Editor website there is a [plugin development kit](#) with some sample plugins (source code and compiled code) and the Javadoc API necessary for developing custom plugins.

The minimal implementation of a plugin must provide:

- a Java class that extends the `ro.sync.exml.plugin.Plugin` class
- a Java class that implements the `ro.sync.exml.plugin.PluginExtension` interface
- a plugin descriptor file called `plugin.xml`

A `ro.sync.exml.plugin.PluginDescriptor` object is passed to the constructor of the subclass of the `ro.sync.exml.plugin.Plugin` class. It contains the following data items about the plugin:

- `basedir` - *File* object - the base directory of the plugin.
- `description` - *String* object - the description of the plugin.
- `name` - *String* object - the name of the plugin.
- `vendor` - *String* object - the vendor name of the plugin.
- `version` - *String* object - the plugin version.

The `ro.sync.exml.plugin.PluginDescriptor` fields are filled with information from the plugin descriptor file.

The plugin descriptor is an XML file that defines how the plugin is integrated in Oxygen XML Editor and what libraries are loaded. The structure of the plugin descriptor file is fully described in a DTD grammar located in `OXYGEN_INSTALLATION_FOLDER/plugins/plugin.dtd`. Here is a sample plugin descriptor used by the *Capitalize Lines* sample plugin:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plugin SYSTEM "../plugin.dtd">
<plugin
  name="Capitalize Lines"
  description="Capitalize the first character on each line"
  version="1.0.0"
  vendor="SyncRO"
  class="ro.sync.sample.plugin.caplines.CapLinesPlugin">
  <runtime>
    <library name="lib/caplines.jar"/>
  </runtime>
  <extension type="selectionProcessor"
    class="ro.sync.sample.plugin.caplines.CapLinesPluginExtension"
    keyboardShortcut="ctrl shift EQUALS"/>
</plugin>
```

If your plugin is of type *Selection*, *Document* or *General*, and thus contributes an action either to the contextual menu or to the main menu, then you can assign a keyboard shortcut for it. You can use the `keyboardShortcut` attribute for each `extension` element to specify the desired shortcut.

 **Tip:** To compose string representations of the desired shortcut keys you can go to the Oxygen XML Editor **Menu Shortcut Keys** preferences page, press **Edit** on any action, press the desired key sequence and use the representation which appears in the edit dialog.

Types of plugins

General Plugin

This plugin type allows the developer to invoke custom code and to interact with the application workspace.

This plugin is the most general plugin type. It provides a limited API:

- The interface `GeneralPluginExtension` - Intended for general-purpose plugins, kind of external tools but triggered from the **Plugins** menu. The implementing classes must provide the method `process(GeneralPluginContext)` which must provide the plugin processing. This method takes as a parameter a `GeneralPluginContext` object.
- The class `GeneralPluginContext` - Represents the context in which the general plugin extension does its processing. The method `getPluginWorkspace()` allows you access to the application workspace.

Selection Plugin

A selection plugin can be applied to both an XML document and a non-XML document. It works as follows: the user makes a selection in the editor, displays the contextual menu, and selects from the **Plugins** submenu the item corresponding to the plugin.

This plugin type provides the following API:

- The interface `SelectionPluginExtension` - The context containing the selected text is passed to the extension and the processed result is going to replace the initial selection. The `process(GeneralPluginContext)` method must return a `SelectionPluginResult` object which contains the result of the processing. The `String` value returned by the `SelectionPluginResult` object can include editor variables like `#{caret}` and `#{selection}`.
- The `SelectionPluginContext` object represents the context. It provides four methods:

- `getSelection()` - Returns a `String` that is the current selection of text.
- `getFrame()` - Returns a `Frame` that is the currently editing frame.
- `getPluginWorkspace()` - Returns access to the application workspace.
- `getDocumentURL()` - Returns the URL of the current edited document.

Document Plugin

This plugin type can be applied only to an XML document. It can modify the current document which is received as callback parameter.

The plugin is started by selecting the corresponding menu item from the contextual menu of the XML editor (Text mode), **Plugins** submenu. It provides the following API:

- The interface `DocumentPluginExtension` - Receives the context object containing the current document in order to be processed. The `process(GeneralPluginContext)` method can return a `DocumentPluginResult` object containing a new document.
- The `DocumentPluginContext` object represents the context. It provides three methods:
 - `getDocument()` - Returns a `javax.swing.text.Document` object that represents the current document.
 - `getFrame()` - Returns a `java.awt.Frame` object that represents the currently editing frame.
 - `getPluginWorkspace()` - Returns access to the application workspace.

Custom Protocol Plugin

This type of plugins allows the developer to work with a custom designed protocol for retrieving and storing files.

It provides the following API:

- The interface `URLStreamHandlerPluginExtension` - There is one method that must be implemented:
 - `getURLStreamHandler(String protocol)` - It takes as an argument the name of the protocol and returns a `URLStreamHandler` object, or null if there is no URL handler for the specified protocol.
- With the help of the `URLChooserPluginExtension2` interface, it is possible to write your own dialog that works with the custom protocol. This interface provides two methods:
 - `chooseURLs(StandalonePluginWorkspace workspaceAccess)` - Returns a `URL[]` object that contains the URLs the user decided to open with the custom protocol. You can invoke your own URL chooser dialog here and then return the chosen URLs having your own custom protocol. You have access to the application workspace.
 - `getMenuName()` - Returns a `String` object that is the name of the entry added in the **File** menu.
- With the help of the `URLChooserToolBarExtension` interface, it is possible to provide a toolbar entry which is used for launching the custom URLs chooser from the `URLChooserPluginExtension` implementation. This interface provides two methods:
 - `getToolBarIcon()` - Returns the `javax.swing.Icon` image used on the toolbar.
 - `getToolBarTooltip()` - Returns a `String` that is the tooltip used on the toolbar button.

Resource Locking Custom Protocol Plugin

This plugin type allows the developer to work with a custom designed protocol for retrieving and storing files. It can lock a resource on opening it in Oxygen XML Editor. This type of plugin extends the custom protocol plugin type with resource locking support.

Such a plugin provides the following API:

- The interface `URLStreamHandlerWithLockPluginExtension` - The plugin receives callbacks following the simple protocol for resource locking and unlocking imposed by Oxygen XML Editor.

There are two additional methods that must be implemented:

- `getLockHandler()` - Returns a `LockHandler` implementation class with the implementation of the lock specific methods from the plugin.
- `isSupported(String protocol)` - Returns a `boolean` that is `true` if the plugin accepts to manage locking for a certain URL protocol scheme like `ftp`, `http`, `https`, or `customName`.

Components Validation Plugin

This plugin type allows the developer to make customization of the editor menus, toolbars, and some other components by allowing or filtering them from the user interface.

This plugin provides the following API:

- The interface `ComponentsValidatorPluginExtension` - There is one method that must be implemented:
 - `getComponentsValidator()` - Returns a `ro.sync.exml.ComponentsValidator` implementation class used for validating the menus, toolbars, and their actions.
- The interface `ComponentsValidator` provides methods to filter various features from being added to the application GUI:
 - `validateMenuOrTaggedAction(String[] menuOrActionPath)` - Checks if a menu or a tag action from a menu is allowed and returns a `boolean` value. A tag is used to uniquely identifying an action. The `String[]` argument is the tag of the menu / action and the tags of its parent menus if any.
 - `validateToolbarTaggedAction(String[] toolbarOrAction)` - Checks if an action from a toolbar is allowed and returns a `boolean` value. The `String[]` argument is the tag of the action from a toolbar and the tag of its parent toolbar if any.
 - `validateComponent(String key)` - Checks if the given component is allowed and returns a `boolean` value. The `String` argument is the tag identifying the component. You can remove toolbars entirely using this callback.
 - `validateAccelAction(String category, String tag)` - Checks if the given accelerator action is allowed to appear in the GUI and returns a `boolean` value. An accelerator action can be uniquely identified so it will be removed both from toolbars or menus. The first argument represents the action category, the second is the tag of the action.
 - `validateContentType(String contentType)` - Checks if the given content type is allowed and returns a `boolean` value. The `String` argument represents the content type. You can instruct the application to ignore content types like `text/xsl` or `text/xquery` and the application will no longer be able to recognize them.
 - `validateOptionPane(String optionPaneKey)` - Checks if the given options page can be added in the preferences option tree and returns a `boolean` value. The `String` argument is the option pane key.
 - `validateOption(String optionKey)` - Checks if the given option can be added in the option page and returns a `boolean` value. The `String` argument is the option key. This method is mostly used for internal use and it is not called for each option in a preferences page.
 - `validateLibrary(String library)` - Checks if the given library is allowed to appear listed in the **About** dialog and returns a `boolean` value. The `String` argument is the library. This method is mostly for internal use.
 - `validateNewEditorTemplate(EditorTemplate editorTemplate)` - Checks if the given template for a new editor is allowed and returns a `boolean` value. The `EditorTemplate` argument is the editor template. An `EditorTemplate` is used to create an editor for a given extension. You can thus filter what appears in the **New** dialog list.
 - `isDebuggerperspectiveAllowed()` - Check if the debugger perspective is allowed and returns a `boolean` value.
 - `validateSHMarker(String marker)` - Checks if the given marker is allowed and returns a `boolean` value. The `String` argument represents the syntax highlight marker to be checked. If you decide to filter certain content types, you can also filter the syntax highlight options so that the content type is no longer present in the Preferences options tree.



Tip: The best way to decide what to filter is to observe the values the application passes when these callbacks are called. You have to create an implementation for this interface which lists in the console all values received by each function. Then you can decide on the values to filter and act accordingly.

Workspace Access Plugin

This plugin type allows the developer to contribute actions to the application main menu and toolbars, to create custom views and to interact with the application workspace

Many complex integrations, like integrations with Content Management Systems (CMS) usually requires access to some workspace resources like the toolbar, menus and to the opened XML editors. This type of plugin is also useful because it allows you to make modifications to an opened editor's XML content.

The plugin must implement the interface

[ro.sync.exml.plugin.workspace.WorkspaceAccessPluginExtension](#). The callback method `applicationStarted` of this interface allows access to a parameter of type

[ro.sync.exml.workspace.api.standalone.StandalonePluginWorkspace](#) which in its turn allows for API access to the application workspace.

The interface `StandalonePluginWorkspace` has three methods which can be called in order to customize the toolbars, menus and views:

- `addToolbarComponentsCustomizer` - Contributes to or modifies existing toolbars. You can specify in the associated `plugin.xml` descriptor additional toolbar IDs using the following construct:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plugin SYSTEM "../plugin.dtd">
<plugin name="CustomWorkspaceAccess" .....>
  <runtime>
    .....
  </runtime>

  <extension type="WorkspaceAccess" ...../>
    .....
    <toolbar id="SampleWorkspaceAccessToolbarID" initialSide="NORTH"
initialRow="1"/>
</plugin>
```

The `toolbar` element adds a toolbar in the Oxygen XML Editor interface and allows you to contribute your own plugin specific actions. The following attributes are available:

- `id` - unique identifier of the plugin toolbar;
- `initialSide` - specifies the place where the toolbar is initially displayed. The allowed values are NORTH and SOUTH.
- `initialRow` - specifies the initial row on the specified side where the toolbar is displayed. For example the main menu has an initial row of "0" and the "Edit" toolbar has an initial row of "1".

The [ro.sync.exml.workspace.api.standalone.ToolbarInfo](#) toolbar component information with the specified `id` will be provided to you by the customizer interface. You will thus be able to provide Swing components which will appear on the toolbar when the application starts.

- `addViewComponentCustomizer` - Contributes to or modifies existing views or contributes to the reserved custom view. You can specify in the associated `plugin.xml` descriptor additional view IDs using the following construct:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plugin SYSTEM "../plugin.dtd">
<plugin name="CustomWorkspaceAccess" .....>
  <runtime>
    .....
  </runtime>
```

```

<extension type="WorkspaceAccess" ...../>
.....
<view id="SampleWorkspaceAccessID" initialSide="WEST" initialRow="0"/>
</plugin>

```

The view element adds a view in the Oxygen XML Editor interface and allows you to contribute your own plugin specific UI components. The following attributes are available:

- `id` - unique identifier of the view component.
- `initialSide` - specifies the place where the view is initially displayed. The allowed values are NORTH, SOUTH, EAST and WEST.
- `initialRow` - specifies the initial row on the specified side where the view is displayed. For example the **Project** view has an initial row of 0 and the Outline view has an initial row of 1. Both views are in the WEST part of the workbench.

The `ro.sync.exml.workspace.api.standalone.ViewInfo` view component information with the specified id will be provided to you by the customizer interface. You will thus be able to provide Swing components which will appear on the view when the application starts.

- `addMenuBarCustomizer` - Contributes to or modifies existing menu components.

Access to the opened editors can be done first by getting access to all URLs opened in the workspace using the API method `StandalonePluginWorkspace.getAllEditorLocations(int editingArea)`. There are two available editing areas: the DITA Maps Manager editing area where only DITA Maps are edited and the main editing area. Using the URL of an opened resource you can gain access to it using the `StandalonePluginWorkspace.getEditorAccess(URL location, int editingArea)` API method. A [ro.sync.exml.workspace.api.editor.WSEditor](#) allows then access to the current editing page. Special editing API is supported only for the **Text** ([ro.sync.exml.workspace.api.editor.page.text.WSTextEditorPage](#)) page and the **Author** ([ro.sync.exml.workspace.api.editor.page.author.WSAuthorEditorPage](#)) page.

In order to be notified when editors are opened, selected and closed you can use the API method `StandalonePluginWorkspace.addEditorChangeListener` to add a listener.

Open Redirect Plugin

This type of plugin is useful for opening more than one file with only one open action.

For example when a zip archive or an ODF file or an OOXML file is open in the **Archive Browser** view a plugin of this type can decide to open a file also from the archive in an XML editor panel. This file can be the `document.xml` main file from an OOXML file archive or a specific XML file from a zip archive.

The plugin must implement the interface `OpenRedirectExtension`. It has only one callback: `redirect(URL)` that receives the URL of the file opened by the Oxygen XML Editor user. If the plugin decides to open also other files it must return an array of information objects (`OpenRedirectInformation[]`) that correspond to these files. Such an information object must contain the URL that is opened in a new editor panel and the content type, for example `text/xml`. The content type is used for determining the type of editor panel. A `null` content type allows auto-detection of the file type.

Targeted URL Stream Handler Plugin

This type of plugin can be used when it is necessary to impose custom URL stream handlers for specific URLs.

This plugin extension can handle the following protocols: `http`, `https`, `ftp` or `sftp`, for which Oxygen XML usually provides specific fixed URL stream handlers. If it is set to handle connections for a specific protocol, this extension will be asked to provide the URL stream handler for each opened connection of an URL having that protocol.

To use this type of plugin, you have to implement the `ro.sync.exml.plugin.urlstreamhandler.TargetedURLStreamHandlerPluginExtension` interface, that provides the following methods:

- `boolean canHandleProtocol(String protocol)`

This method checks if the plugin can handle a specific protocol. If this method returns `true` for a specific protocol, the `getURLStreamHandler(URL)` method will be called for each opened connection of an URL having this protocol.

- `URLStreamHandler getURLStreamHandler(URL url)`

This method provides the URL handler for the specified URL and it is called for each opened connection of an URL with a protocol for which the `canHandleProtocol(String)` method returns `true`.

If this method returns `null`, the Oxygen `URLStreamHandler` is used.

To use this type of extension in your plugin, create an extension of `TargetedURLHandler` type in your `plugin.xml` and specify the class that implements `TargetedURLStreamHandlerPluginExtension`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plugin SYSTEM "../plugin.dtd">
<plugin name="CustomTargetedURLStreamHandlerPlugin" .....>
  <runtime>
    .....
  </runtime>

  <extension type="TargetedURLHandler"
class="CustomTargetedURLStreamHandlerPluginExtension"/>
  .....
</plugin>
```

This extension can be useful in situations when connections opened from a specific host must be handled in a particular way. For example, the Oxygen HTTP `URLStreamHandler` may not be compatible for sending and receiving SOAP using the SUN Webservices implementation. In this case you can override the stream handler set by Oxygen for HTTP to use the default SUN `URLStreamHandler` which is more compatible with sending and receiving SOAP requests.

```
public class CustomTargetedURLStreamHandlerPluginExtension
implements TargetedURLStreamHandlerPluginExtension {

  @Override
  public boolean canHandleProtocol(String protocol) {
    boolean handleProtocol = false;
    if ("http".equals(protocol) || "https".equals(protocol)) {
      // This extension handles both HTTP and HTTPS protocols
      handleProtocol = true;
    }
    return handleProtocol;
  }

  @Override
  public URLStreamHandler getURLStreamHandler(URL url) {
    // This method is called only for the URLs with a protocol
    // for which the canHandleProtocol(String) method returns true (HTTP and
    // HTTPS)

    URLStreamHandler handler = null;

    String host = url.getHost();
    String protocol = url.getProtocol();
    if ("some_host".equals(host)) {
      // When there are connections opened from some_host, the SUN HTTP(S)
      // handlers are used
      if ("http".equals(protocol)) {
```

```

        handler = new sun.net.www.protocol.http.Handler();
    } else {
        handler = new sun.net.www.protocol.https.Handler();
    }
}
return handler;
}
}

```

Lock Handler Factory Plugin

This type of extension is used for locking resources from a specific protocol.

It provides the following API:

- The interface `LockHandlerFactoryPluginExtension`.

You need to implement the following two methods:

- `LockHandler getLockHandler()`
Gets the lock handler for the current handled protocol. Might be null if not supported.
- `boolean isLockingSupported(String protocol)`
Checks if a lock handler can be provided for a specific protocol.

To use this type of extension in your plugin, create an extension of `LockHandlerFactory` type in your `plugin.xml` and specify the class implementing `LockHandlerFactoryPluginExtension`:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plugin SYSTEM "../plugin.dtd">
<plugin name="CustomLockHandler" .....>
  <runtime>
    .....
  </runtime>

  <extension type="LockHandlerFactory"
class="LockHandlerFactoryPluginExtensionImpl"/>
  .....
</plugin>

```

How to

Different tutorials about how to implement complex plugins.

How to Write a CMS Integration Plugin

In order to have a complete integration between Oxygen XML Editor and any CMS you usually have to write a plugin which combines two available plugin extensions:

- [Workspace Access](#)
- [Custom protocol](#)

The usual set of requirements for an integration between Oxygen XML Editor and the CMS are the following:

- Contribute to the Oxygen XML Editor toolbars and main menu with your custom **Check Out** and **Check In** actions:
 - **Check Out** triggers your custom dialogs which allow you to browse the remote CMS and choose the resources you want to open;

- **Check In** allows you to send back to the server the modified content.

You can use the **Workspace Access** plugin extension (and provided sample Java code) for all these operations.

- When **Check Out** is called, use the Oxygen XML Editor API to open your custom URLs (URLs created using your custom protocol). It is important to implement and use a **Custom Protocol** extension in order to be notified when the files are opened and saved and to be able to provide to Oxygen XML Editor the content for the relative references the files may contain. Your custom `java.net.URLStreamHandler` implementation checks out the resource content from the server, stores it locally and provides its content. Sample **Check Out** implementation:

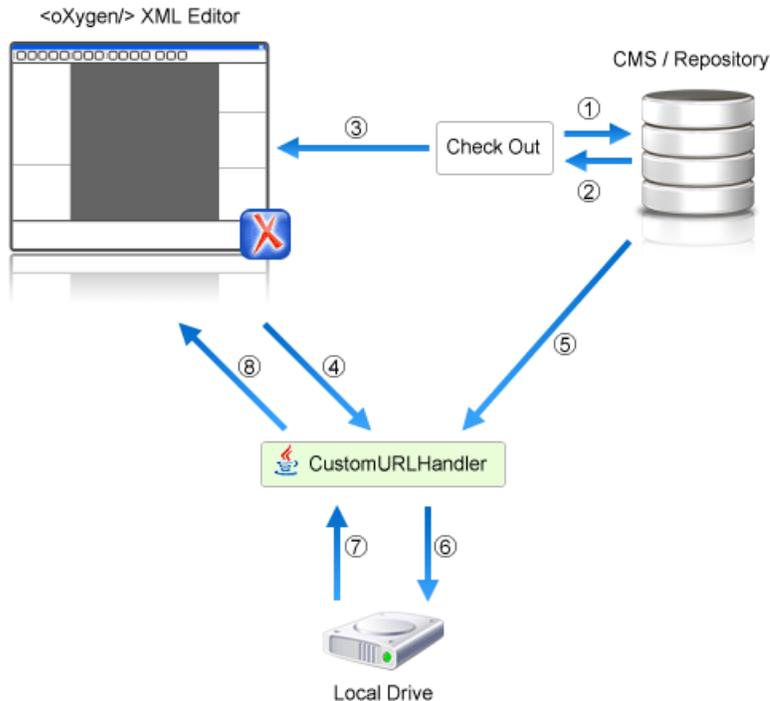
```

/**
 * Sample implementation for the "Check Out" method.
 *
 * @param pluginWorkspaceAccess The plugin workspace access (Workspace
Access plugin).
 * @throws MalformedURLException
 */
private void checkOut(StandalonePluginWorkspace pluginWorkspaceAccess)
throws MalformedURLException {
    //TODO Show the user a custom dialog for browsing the CMS
    //TODO after the user selected the resource create an URL with a custom
protocol
    // which will uniquely map to the resource on the CMS using the URLHandler

    //something like:
    URL customURL = new URL("mycms://host/path/to/file.xml");
    //Ask Oxygen to open the URL
    pluginWorkspaceAccess.open(customURL);
    //Oxygen will then your custom protocol handler to provide the contents
for the resource "mycms://host/path/to/file.xml"
    //Your custom protocol handler will check out the file in a temporary
directory for example and provide the content from it.
    //Oxygen will also pass through your URLHandler if you have any relative
references which need to be opened/obtained.
}

```

Here is a diagram of the **Check Out** process:



Each phase is described below:

1. Browse CMS repository
 2. User chooses a resource
 3. Use API to open custom URL: mycms://path/to/file.xml
 4. Get content of URL: mycms://path/to/file.xml
 5. Get content of resource
 6. Store on disk for faster access
 7. Retrieve content from disk if already checked out
 8. Retrieved content
- Contribute a special **Browse CMS** action to every dialog in Oxygen XML Editor where an URL can be chosen to perform a special action (like the **Insert a DITA Content Reference** action or the **Insert Image** action). Sample code:

```
//Add an additional browse action to all dialogs/places where Oxygen
allows selecting an URL.
pluginWorkspaceAccess.addInputURLChooserCustomizer(new
InputURLChooserCustomizer() {
    public void customizeBrowseActions(List<Action> existingBrowseActions,
    final InputURLChooser chooser) {
        //IMPORTANT, you also need to set a custom icon on the action for
situations when its text is not used for display.
        Action browseCMS = new AbstractAction("CMS") {
            public void actionPerformed(ActionEvent e) {
                URL chosenResource = browseCMSAndChooseResource();
                if (chosenResource != null) {
                    try {
                        //Set the chosen resource in the dialog's combo box chooser.

                        chooser.urlChosen(chosenResource);
                    } catch (MalformedURLException e1) {
                        //
                    }
                }
            }
        }
    }
}
```

```

        }
    };
    existingBrowseActions.add(browseCMS);
}
});

```

When inserting references to other resources using the actions already implemented in Oxygen XML Editor, the reference to the resource is made by default relative to the absolute location of the edited XML file. You can gain control over the way in which the reference is made relative for a specific protocol like:

```

//Add a custom relative reference resolver for your custom protocol.
//Usually when inserting references from one URL to another Oxygen makes
the inserted path relative.
//If your custom protocol needs special relativization techniques then
it should set up a custom relative
//references resolver to be notified when resolving needs to be done.
pluginWorkspaceAccess.addRelativeReferencesResolver(
//Your custom URL protocol for which you already have a custom
URLStreamHandlerPluginExtension set up.
    "mycms",
//The relative references resolver
    new RelativeReferenceResolver() {
public String makeRelative(URL baseUrl, URL childURL) {
//Return the referenced path as absolute for example.
//return childURL.toString();
//Or return null for the default behavior.
return null;
}
});

```

- Write the plugin.xml descriptor. Your plugin combines the two extensions using a single set of libraries. The descriptor would look like:

```

<!DOCTYPE plugin SYSTEM "../plugin.dtd">
<plugin
  name="CustomCMSAccess"
  description="Test"
  version="1.0.0"
  vendor="ACME"
  class="custom.cms.CMSAccessPlugin">
  <runtime>
    <library name="lib/cmsaccess.jar"/>
  </runtime>
  <!--Access to add actions to the main menu and toolbars or to add custom
views.-->
  <!--See the
"ro.sync.sample.plugin.workspace.CustomWorkspaceAccessPluginExtension" Java
sample for more details-->
  <extension type="WorkspaceAccess"
    class="custom.cms.CustomWorkspaceAccessPluginExtension"/>
  <!--The custom URL handler which will communicate with the CMS
implementation-->
  <!--See the
"ro.sync.sample.plugin.workspace.customprotocol.CustomProtocolURLHandlerExtension"
Java sample for more details-->
  <extension type="URLHandler"
    class="custom.cms.CustomProtocolURLHandlerExtension"/>
</plugin>

```

- Create a cmsaccess.jar JAR archive containing your implementation classes.

- Copy your new plugin directory in the `plugins` subfolder of the Oxygen XML Editor install folder and start Oxygen XML Editor.

Class Loading Issues

It is possible that the Java libraries you have specified in the plugin libraries list conflict with the ones already loaded by Oxygen XML Editor. In order to instruct the plugin to prefer its libraries over the ones used by Oxygen XML Editor, you can add the following attribute on the `<plugin>` root element: `classLoaderType="preferReferencedResources"` from the `plugin.xml` descriptor.

A **Late Delegation Class Loader** (the main class loader in Oxygen XML Editor) is a `java.net.URLClassLoader` extension which prefers to search classes in its own libraries list and only if a class is not found there to delegate to the parent class loader.

The main Oxygen XML Editor Class Loader uses as libraries all jars specified in the `OXYGEN_INSTALL_DIR\lib` directory. Its parent class loader is the default JVM Class loader. For each instantiated plugin a separate class loader is created having as parent the Oxygen XML Editor Class Loader.

The plugin class loader can be either a standard `java.net.URLClassLoader` or a `LateDelegationClassLoader` (depending on the attribute `classLoaderType` in the `plugin.xml`). Its parent class loader is always the Oxygen XML Editor `LateDelegationClassLoader`.

If you experience additional problems like the following:

```
java.lang.LinkageError: ClassCastException: attempting to cast
jar:file:/C:/jdk1.6.0_06/jre/lib/rt.jar!/javax/xml/ws/spi/Provider.class to jar:file:/D:/Program
Files/Oxygen XML Editor
12/plugins/wspcaccess/../../xdocs/lib/jaxws/jaxws-api.jar!/javax/xml/ws/spi/Provider.class
at javax.xml.ws.spi.Provider.provider(Provider.java:94) at
javax.xml.ws.Service.<init>(Service.java:56)
.....
```

The cause could be the fact that some classes are instantiated using the context class loader of the current thread. The most straightforward fix is to write your code in a *try/finally* statement:

```
ClassLoader oldClassLoader = Thread.currentThread().getContextClassLoader();
try {
    //This is the implementation of the WorkspaceAccessPluginExtension plugin
interface.
    Thread.currentThread().setContextClassLoader(
        CustomWorkspaceAccessPluginExtension.this.getClass().getClassLoader());

    //WRITE YOUR CODE HERE
} finally {
    Thread.currentThread().setContextClassLoader(oldClassLoader);
}
```

How to Write A Custom Protocol Plugin

For creating a custom protocol plugin, apply the following steps:

1. Write the handler class for your protocol that implements the `java.net.URLStreamHandler` interface. Be careful to provide ways to correct and uncorrect the URLs of your files.
2. Write the plugin class by extending `ro.sync.exml.plugin.Plugin`.
3. Write the plugin extension class that implements the `ro.sync.exml.plugin.urlstreamhandler.URLStreamHandlerPluginExtension` interface.

It is necessary that the plugin extension for the custom protocol implements the `URLStreamHandlerPluginExtension` interface. Without it, you cannot use your plugin, because Oxygen XML Editor is not able to find the protocol handler.

You can choose also to implement the `URLChooserPluginExtension` interface. It allows you to write and display your own customized dialog for selecting resources that are loaded with the custom protocol.

An implementation of the extension `URLHandlerReadOnlyCheckerExtension` allows you to:

- mark a resource as read-only when it is opened
- switch between marking the resource as read-only and read-write while it is edited

It is useful when opening and editing CMS resources.

4. Write the `plugin.xml` descriptor.

Remember to set the name of the plugin class to the one from the second step and the plugin extension class name with the one you have chosen at step 3.

5. Create a `.jar` archive with all these files.

6. Install your new plugin in the `plugins` subfolder of the Oxygen XML Editor install folder.

Installation

In the directory where Oxygen XML Editor is installed there exists a directory called `plugins` that contains all the available plugins. In order for Oxygen XML Editor to use the new functionality you provided, follow the next steps:

1. In the `plugins` folder create a subfolder to store the plugin files.
2. Put in this new folder the plugin descriptor file `plugin.xml`, the Java classes of the plugin and the other files that are referenced in the descriptor file.
3. Restart Oxygen XML Editor.

Example - A Selection Plugin

The following plugin is called `UppercasePlugin` and is an example of *selection plugin*. It is used in Oxygen XML Editor for capitalizing the characters in the current selection. This example consists of two Java classes and the plugin descriptor:

- `UppercasePlugin.java`:

```
package ro.sync.sample.plugin.uppercase;

import ro.sync.exml.plugin.Plugin;
import ro.sync.exml.plugin.PluginDescriptor;

public class UppercasePlugin extends Plugin {
    /**
     * Plugin instance.
     */
    private static UppercasePlugin instance = null;

    /**
     * UppercasePlugin constructor.
     *
     * @param descriptor Plugin descriptor object.
     */
    public UppercasePlugin(PluginDescriptor descriptor) {
        super(descriptor);
    }
}
```

```

        if (instance != null) {
            throw new IllegalStateException("Already instantiated !");
        }
        instance = this;
    }

    /**
     * Get the plugin instance.
     *
     * @return the shared plugin instance.
     */
    public static UppercasePlugin getInstance() {
        return instance;
    }
}

```

- UppercasePluginExtension.java:

```

package ro.sync.sample.plugin.uppercase;

import ro.sync.exml.plugin.selection.SelectionPluginContext;
import ro.sync.exml.plugin.selection.SelectionPluginExtension;
import ro.sync.exml.plugin.selection.SelectionPluginResult;
import ro.sync.exml.plugin.selection.SelectionPluginResultImpl;

public class UppercasePluginExtension implements SelectionPluginExtension {
    /**
     * Convert the text to uppercase.
     *
     * @param context Selection context.
     * @return Uppercase plugin result.
     */
    public SelectionPluginResult process(SelectionPluginContext context) {
        return new SelectionPluginResultImpl(
            context.getSelection().toUpperCase());
    }
}

```

- plugin.xml:

```

<!DOCTYPE plugin SYSTEM "../plugin.dtd">
<plugin
  name="UpperCase"
  description="Convert the selection to uppercase"
  version="1.0.0"
  vendor="SyncRO"
  class="ro.sync.sample.plugin.uppercase.UppercasePlugin">
  <runtime>
    <library name="lib/uppercase.jar"/>
  </runtime>
  <extension type="selectionProcessor"
    class="ro.sync.sample.plugin.uppercase.UppercasePluginExtension"/>
</plugin>

```

Index

D

Develop an <oxygen/> plugin 3, 15
 example - UppercasePlugin 15
 introduction 3

E

Extend Oxygen with plugins 3, 4, 5, 6, 8, 10, 14, 15
 implement plugin 3, 4, 5, 6, 8, 10, 14, 15
 CMS integration plugin 10
 components validation plugin 6
 custom protocol plugin 5, 8, 10

Extend Oxygen with plugins (*continued*)

 implement plugin (*continued*)
 document plugin 5
 general plugin 4
 how to install a plugin 15
 how to write a custom protocol plugin 14
 resource locking custom protocol plugin 5
 selection plugin 4
 requirements 3

W

Workspace Access 7