Author Developer Guide

Contents

Author Developer Guide

The Author editor of was designed to bridge the gap between the XML source editing and a friendly user interface. The main achievement is the fact that the Author combines the power of source editing with the intuitive interface of a text editor.

This guide is targeted at advanced authors who want to customize the Author editing environment and is included both as a chapter in the user manual and as a separate document in *the Author SDK*.

Figure 1: Author Visual Editor

Although comes with already configured frameworks for DocBook, DITA, TEI, XHTML, you might need to create a customization of the editor to handle other types of documents. The common use case is when your organization holds a collection of XML document types used to define the structure of internal documents and they need to be visually edited by people with no experience in working with XML files.

There are several ways to customize the editor:

1. Create a CSS file defining styles for the XML elements the user will work with, and create XML files that refer the CSS through an `xml-stylesheet` processing instruction.
2. Fully configure a document type association. This involves putting together the CSSs, the XML schemes, actions, menus, etc, bundling them and distributing an archive. The CSS and the GUI elements are settings of the Author. The other settings like the templates, catalogs, transformation scenarios are general settings and are enabled whenever the association is active, no matter the editing mode (Text, Grid or Author).

Both approaches will be discussed in the following sections.

Simple Customization Tutorial

The most important elements of a document type customization are represented by an XML Schema to define the XML structure, the CSS to render the information and the XML instance template which links the first two together.

XML Schema

Let's consider the following XML Schema, `test_report.xsd` defining a report with results of a testing session. The report consists of a title, few lines describing the test suite that was run and a list of test results, each with a name and a boolean value indicating if the test passed or failed.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="report">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="title"/>
                <xs:element ref="description"/>
                <xs:element ref="results"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
    <xs:element name="title" type="xs:string"/>
    <xs:element name="description">
        <xs:complexType>
            <xs:sequence maxOccurs="unbounded">
                <xs:element name="line">
                    <xs:complexType mixed="true">
                        <xs:sequence minOccurs="0"
                            maxOccurs="unbounded">
                            <xs:element name="important"
                              type="xs:string"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="results">
        <xs:complexType>
            <xs:sequence maxOccurs="unbounded">
                <xs:element name="entry">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="test_name"
                              type="xs:string"/>
                            <xs:element name="passed"
                               type="xs:boolean"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

The use-case is that several users are testing a system and must send report results to a content management system. The Author customization should provide a visual editor for this kind of documents.

CSS Stylesheet

A set of rules must be defined for describing how the XML document is to be rendered into the Author. This is done using Cascading Style Sheets or CSS on short. CSS is a language used to describe how an HTML or XML document should be formatted by a browser. CSS is widely used in the majority of websites.

The elements from an XML document are displayed in the layout as a series of boxes. Some of the boxes contain text and may flow one after the other, from left to right. These are called in-line boxes. There are also other type of boxes that flow one below the other, like paragraphs. These are called block boxes.

For example consider the way a traditional text editor arranges the text. A paragraph is a block, because it contains a vertical list of lines. The lines are also blocks. But any block that contains inline boxes is arranging its children in a horizontal flow. That is why the paragraph lines are also blocks, but the traditional "bold" and "italic" sections are represented as inline boxes.

The CSS allows us to specify that some elements are displayed as tables. In CSS a table is a complex structure and consists of rows and cells. The "table" element must have children that have "table-row" style. Similarly, the "row" elements must contain elements with "table-cell" style.

To make it easy to understand, the following section describes the way each element from the above schema is formatted using a CSS file. Please note that this is just one from an infinite number of possibilities of formatting the content.

report

This element is the root element of the report document. It should be rendered as a box that contains all other elements. To achieve this the display type is set to **block**. Additionally some margins are set for it. The CSS rule that matches this element is:

```
report{
    display:block;
    margin:1em;
}
```

title

The title of the report. Usually titles have a larger font. The **block** display should also be used - the next elements will be placed below it, and change its font to double the size of the normal text.

```
title {
    display:block;
    font-size:2em;
}
```

description

This element contains several lines of text describing the report. The lines of text are displayed one below the other, so the description will have the same **block** display. To make it standout the background color is changed.

```
description {
    display:block;
    background-color:#EEEEFF;
    color:black;
}
```

line

A line of text in the description. A specific aspect is not defined for it, just indicate that the display should be **block**.

```
line {
    display:block;
}
```

important

The `important` element defines important text from the description. Because it can be mixed with text, its display property must be set to **inline**. To make it easier to spot, the text will be emphasized.

```
important {
    display:inline;
    font-weight:bold;
}
```

results

The `results` element shows the list of test_names and the result for each one. To make it easier to read, it is displayed as a **table** with a green border and margins.

```
results{
    display:table;
    margin:2em;
    border:1px solid green;
}
```

entry

An item in the results element. The results are displayed as a table so the entry is a row in the table. Thus, the display is **table-row**.

```
entry {
    display:table-row;
}
```

test_name, passed

The name of the individual test, and its result. They are cells in the results table with display set to **table-cell**. Padding and a border are added to emphasize the table grid.

```
test_name, passed{
    display:table-cell;
    border:1px solid green;
    padding:20px;
}
```

```
passed{
    font-weight:bold;
}
```

The full content of the CSS file `test_report.css` is:

```
report {
    display:block;
    margin:1em;
}

description {
    display:block;
    background-color:#EEEEFF;
    color:black;
}

line {
    display:block;
}

important {
    display:inline;
    font-weight:bold;
}

title {
    display:block;
    font-size:2em;
}

results{
    display:table;
    margin:2em;
    border:1px solid green;
}

entry {
    display:table-row;
}

test_name, passed{
    display:table-cell;
    border:1px solid green;
    padding:20px;
}

passed{
    font-weight:bold;
}
```

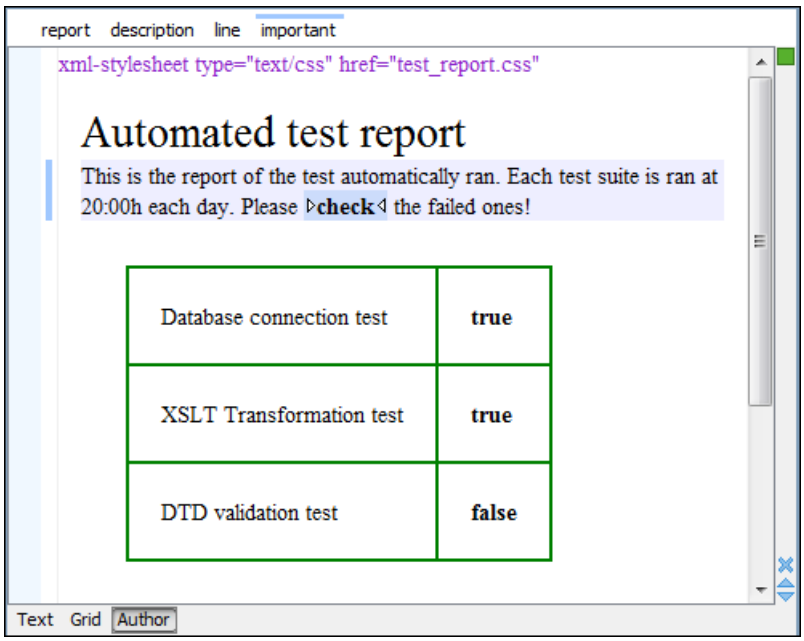


Figure 2: A report opened in the Author

Note:

You can edit attributes in-place in the Author mode using *form-based controls*.

The XML Instance Template

Based on the XML Schema and the CSS file the Author can help the content author in loading, editing and validating the test reports. An XML file template must be created, a kind of skeleton, that the users can use as a starting point for creating new test reports. The template must be generic enough and refer the XML Schema file and the CSS stylesheet.

This is an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="test_report.css"?>
<report xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="test_report.xsd">
  <title>Automated test report</title>
  <description>
    <line>This is the report of the test automatically ran. Each test suite is ran at 20:00h each
      day. Please <important>check</important> the failed ones!</line>
  </description>
  <results>
    <entry>
      <test_name>Database connection test</test_name>
      <passed>true</passed>
    </entry>
    <entry>
      <test_name>XSLT Transformation test</test_name>
      <passed>true</passed>
    </entry>
    <entry>
      <test_name>DTD validation test</test_name>
      <passed>false</passed>
    </entry>
  </results>
</report>
```

The processing instruction `xml-stylesheet` associates the CSS stylesheet to the XML file. The href pseudo attribute contains the URI reference to the stylesheet file. In our case the CSS is in the same directory as the XML file.

The next step is to place the XSD file and the CSS file on a web server and modify the template to use the HTTP URLs, like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css"
        href="http://www.mysite.com/reports/test_report.css"?>
<report xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation=
        "http://www.mysite.com/reports/test_report.xsd">
    <title>Test report title</title>
    <description>
.......
```

The alternative is to create an archive containing the `test_report.xml`, `test_report.css` and `test_report.xsd` and send it to the content authors.

Advanced Customization Tutorial - Document Type Associations

is highly customizable. Practically you can associate an entire class of documents (grouped logically by some common features like namespace, root element name or filename) to a bundle consisting of CSS stylesheets, validation schemas, catalog files, new files templates, transformation scenarios and even custom actions. The bundle is called *document type* and the association is called *Document Type Association* or, more generically, *framework*.

In this tutorial a **Document Type Association** will be created for a set of documents. As an example a light documentation framework (similar to DocBook) will be created, then complete customization of the Author editor will be set up.

Note: The samples used in this tutorial can be found in the *Example Files Listings*.

Note: The complete source code can be found in the Simple Documentation Framework project, included in the *Oxygen Author SDK zip* available for download *on the website*.

Note: The Javadoc documentation of the Author API used in the example files is *available on the website*. Also it can be downloaded as a *zip archive from the website*.

Document Type

A *document type* or *framework* is associated to an XML file according to a set of rules. It includes also many settings that improve editing in the Author mode for the category of XML files it applies for. These settings include:

- a default grammar used for validation and content completion in both Author mode and Text mode;
- CSS stylesheet(s) for rendering XML documents in Author mode;
- user actions invoked from toolbar or menu in Author mode;
- predefined scenarios used for transformation of the class of XML documents defined by the document type;
- XML catalogs;
- directories with file templates;
- user-defined extensions for customizing the interaction with the content author in Author mode.

Note: The Author mode allows WYSIWYG-like visual editing of XML documents and is available only in Editor and Author.

The tagless editor comes with some predefined document types already configured when the application is installed on the computer. These document types describe well-known XML frameworks largely used today for authoring XML documents. Editing a document which conforms to one of these types is as easy as opening it or creating it from one of the predefined document templates which also come with .

Author Settings

You can add a new *Document Type Association* or edit the properties of an existing one from the **Options** > **Preferences** > **Document Type Association** option pane. All the changes can be made into the *Document type* edit dialog.

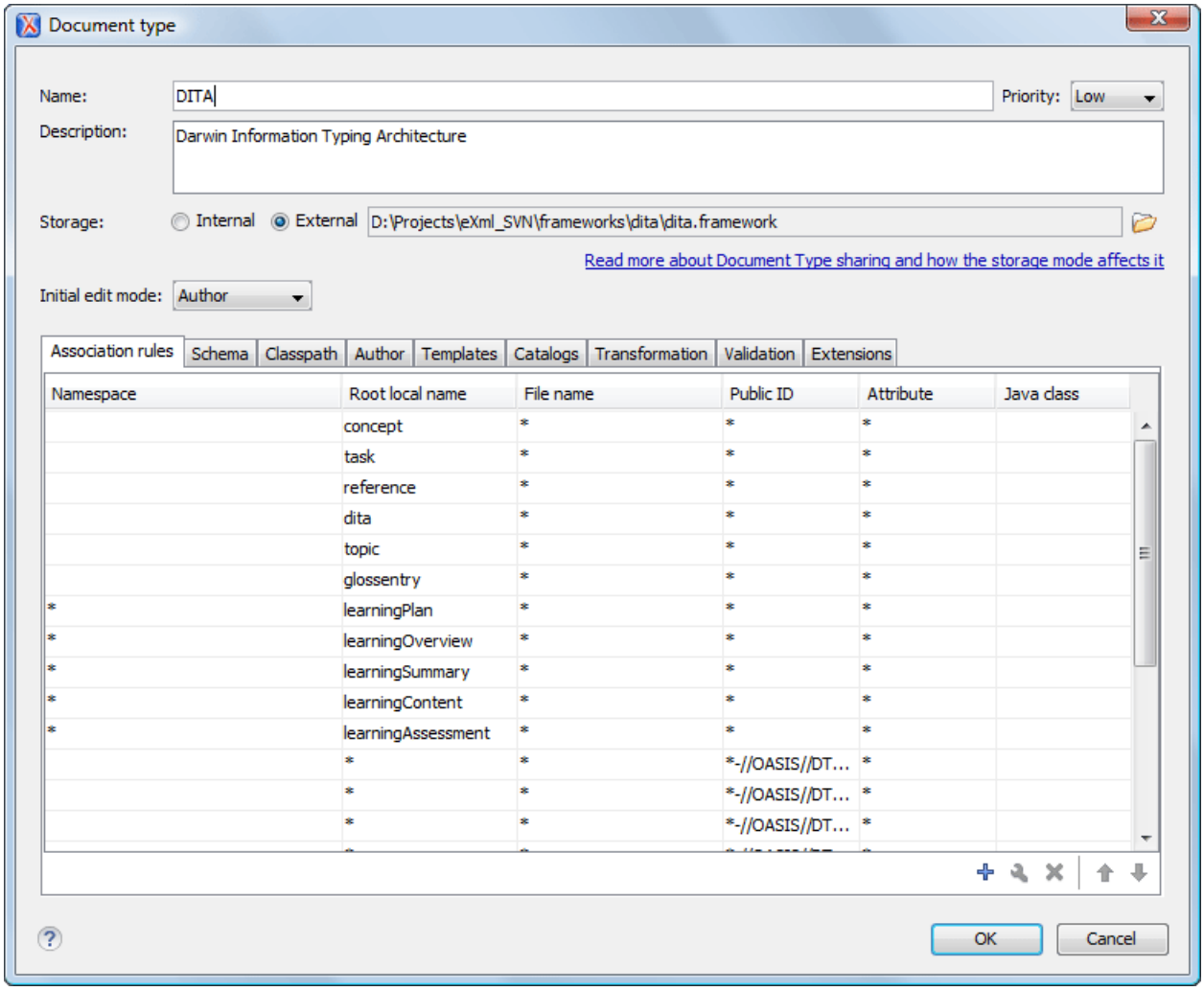


Figure 3: The Document Type

You can specify the following properties for a document type:

- **Name** - The name of the document type.
- **Priority** - When multiple document types match the same document, the priority determines the order in which they are applied. It can be one of: Lowest, Low, Normal, High, Highest. The predefined document types that are already configured when the application is installed on the computer have the default Low priority.

  Note: The frameworks having the same priority are alphabetically sorted.

- **Description** - The document type description displayed as a tooltip in the Document Type Association table.
- **Storage** - The location where the document type is saved. If you select the **External** storage, the document type is saved in the specified file with a mandatory `framework` extension, located in a subfolder of the current `frameworks` directory. If you select the **Internal** storage option, the document type data is saved in the current `.xpr` project file (for Project-level Document Type Association Options) or in the internal options (for Global-level Document Type Association Options). You can change the Document Type Association Options level in the Document Type Association panel.
- **Initial edit mode** - Allows you to select the initial editing mode (**Editor specific**, **Text**, **Grid** and **Design** (available only for the W3C XML Schema editor)) for this document type. If the **Editor specific** option is selected, the initial edit mode is determined depending on the editor type. You can find the mapping between editors and edit modes in the **Edit modes** preferences page. You can decide to impose an initial mode for opening files which match the association rules of the document type. For example if the files are usually edited in the *Author* mode you can set it in the **Initial edit mode** combo box.

Note: You can also customize the initial mode for a document type in the **Edit modes** preferences page. To open this page, go to **Options** > **Preferences** > **Editor** > **Edit modes** .

You can specify the association rules used for determining a document type for an opened XML document. A rule can define one or more conditions. All conditions need to be fulfilled in order for a specific rule to be chosen. Conditions can specify:

- **Namespace** - The namespace of the document that matches the document type.
- **Root local name of document** - The local name of the document that matches the document type.
- **File name** - The file name (including the extension) of the document that matches the document type.
- **Public ID** (for DTDs) - The PUBLIC identifier of the document that matches the document type.
- **Attribute** - This field allows you to associate a document type depending on a certain value of the attribute in the root.
- **Java class** - Name of Java class that is called for finding if the document type should be used for an XML document. Java class must implement `ro.sync.ecss.extensions.api.DocumentTypeCustomRuleMatcher` interface from *Author API*.

In the **Schema** tab, you can specify the type and URI of schema used for validation and content completion of all documents from the document type, when there is no schema detected in the document.

You can choose one of the following schema types:

- DTD;
- Relax NG schema (XML syntax);
- Relax NG schema (XML syntax) + Schematron;
- Relax NG schema (compact syntax);
- XML Schema;
- XML Schema + Schematron rules;
- NVDL schema.

Configuring Actions, Menus and Toolbars

The Author toolbars and menus can be changed to provide a productive editing experience for the content authors. You can create a set of actions that are specific to a document type.

In the example with the `sdf` framework, you created the stylesheet and the validation schema. Now let's add some actions to insert a `section` and a `table`. To add a new action, follow the procedure:

1. Open the **Options Dialog**, and select the **Document Types Association** option pane.
2. In the lower part of the **Document Type Association** dialog, click on the **Author** tab, then select the **Actions** label.
3. To add a new action click on the **+ Add** button.

Creating the Insert Section Action

This section shows all the steps needed to define the Insert Section action. We assume the icon files  (`Section16.png`) for the menu item and  (`Section20.png`) for the toolbar, are already available. Although you could use the same icon size for both menu and toolbar, usually the icons from the toolbars are larger than the ones found in the menus. These files should be placed in the `frameworks / sdf` directory.

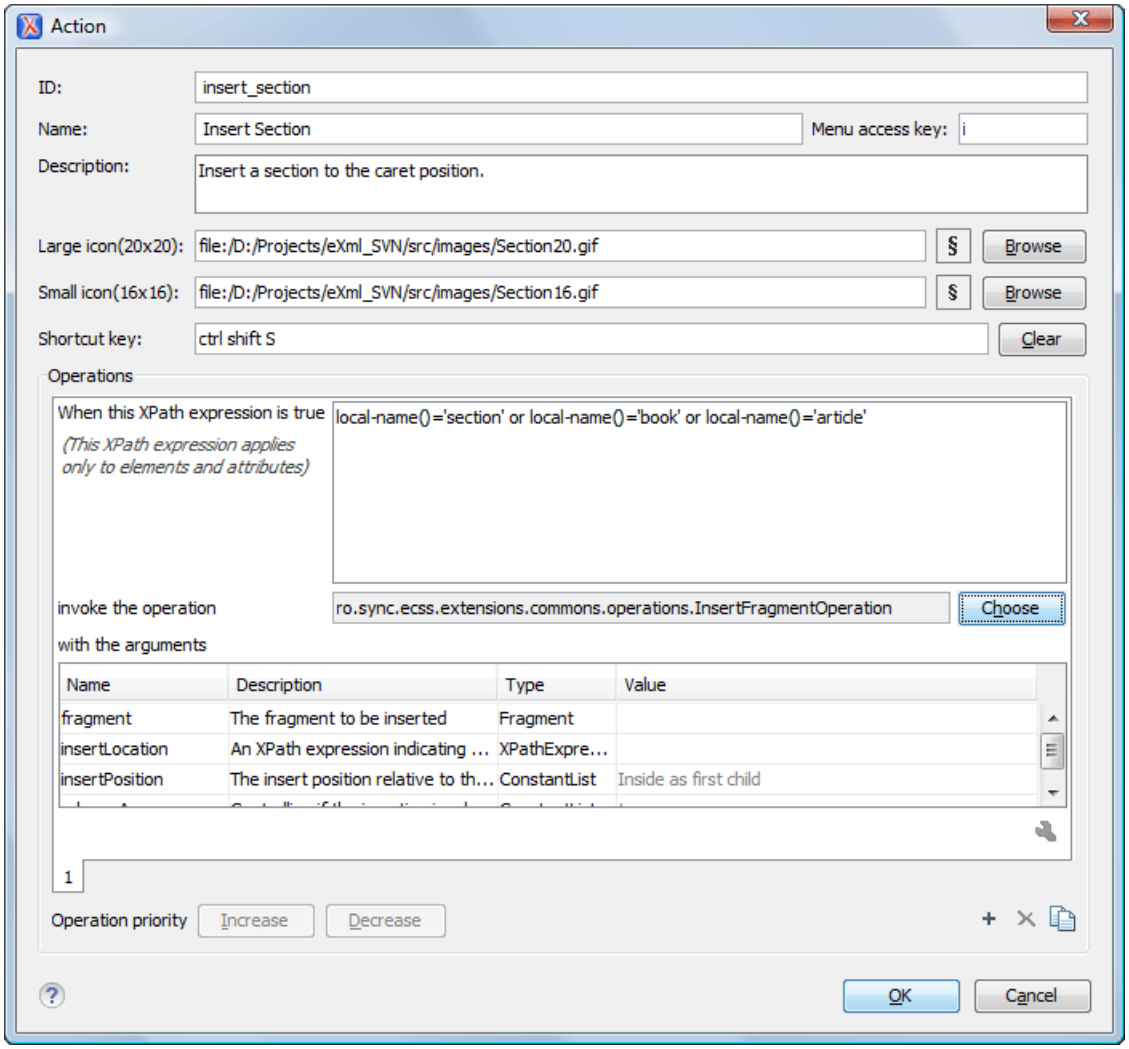


Figure 4: The Action Edit Dialog

1. Set the **ID** field to **insert_section**. This is an unique action identifier.
2. Set the **Name** field to **Insert Section**. This will be the action's name, displayed as a tooltip when the action is placed in the toolbar, or as the menu item name.
3. Set the **Menu access key** to **i**. On Windows, the menu items can be accessed using (ALT + letter) combination, when the menu is visible. The letter is visually represented by underlining the first letter from the menu item name having the same value.
4. Set the **Description** field to **Insert a section at caret position**.
5. Set the **Large icon (20x20)** field to **${frameworks} / sdf / Section20.png**. A good practice is to store the image files inside the framework directory and use *editor variable* `${frameworks}` to make the image relative to the framework location.

   If the images are bundled in a jar archive together with some Java operations implementation for instance, it might be convenient for you to refer the images not by the file name, but by their relative path location in the class-path.

   If the image file `Section20.png` is located in the **images** directory inside the jar archive, you can refer to it by using **/images/Section20.png**. The jar file must be added into the **Classpath** list.
6. Set the **Small icon (16x16)** field to **${frameworks} / sdf / Section16.png**.
7. Click the text field next to **Shortcut key** and set it to **Ctrl+Shift+S**. This will be the key combination to trigger the action using the keyboard only.

The shortcut is enabled only by *adding the action to the main menu of the Author mode* which contains all the actions that the author will have in a menu for the current document type.

8. At this time the action has no functionality added to it. Next you must define how this action operates. An action can have multiple operation modes, each of them activated by the evaluation of an XPath version 2.0 expression. The first enabled action mode will be executed when the action is triggered by the user. The scope of the XPath expression must be only element nodes and attribute nodes of the edited document, otherwise the expression will not return a match and will not fire the action. For this example we'll suppose you want allow the action to add a section only if the current element is either a `book`, `article` or another `section`.
   a) Set the XPath expression field to:

```
local-name()='section' or local-name()='book' or
 local-name()='article'
```

   b) Set the **invoke operation** field to `InsertFragmentOperation` built-in operation, designed to insert an XML fragment at caret position. This belongs to a set of built-in operations, a complete list of which can be found in the *Author Default Operations* section. This set can be expanded with your own Java operation implementations.
   c) Configure the arguments section as follows:

```
<section xmlns=
"http://www.oxygenxml.com/sample/documentation">
     <title/>
</section>
```

   `insertLocation` - leave it empty. This means the location will be at the caret position.

   `insertPosition` - select "**Inside**".

The Insert Table Action

You will create an action that inserts into the document a table with three rows and three columns. The first row is the table header. Similarly to the insert section action, you will use the `InsertFragmentOperation`.

Place the icon files for the menu item and for the toolbar in the `frameworks / sdf` directory.

1. Set **ID** field to **insert_table**.
2. Set **Name** field to **Insert table**.
3. Set **Menu access key** field to **t**.
4. Set **Description** field to **Adds a section element**.
5. Set **Toolbar icon** to **${frameworks} / sdf / toolbarIcon.png**.
6. Set **Menu icon** to **${frameworks} / sdf / menuIcon.png**.
7. Set **Shortcut key** to **Ctrl+Shift+T**.
8. Set up the action's functionality:
   a) Set **XPath expression** field to `true()`.

   `true()` is equivalent with leaving this field empty.

   b) Set **Invoke operation** to use **InvokeFragmentOperation** built-in operation that inserts an XML fragment to the caret position.
   c) Configure operation's arguments as follows:

   fragment - set it to:

```
<table xmlns=
"http://www.oxygenxml.com/sample/documentation">
  <header><td/><td/><td/></header>
  <tr><td/><td/><td/></tr>
  <tr><td/><td/><td/></tr>
</table>
```

`insertLocation` - to add tables at the end of the section use the following code:

```
ancestor::section/*[last()]
```

`insertPosition` - Select **After**.

Configuring the Toolbars

Now that you have defined the ***Insert Section*** action and the ***Insert Table*** action, you can add them to the toolbar. You can configure additional toolbars on which to add your custom actions.

1. Open the Document Type edit dialog for the **SDF** framework and select on the **Author** tab. Next click on the **Toolbar** label.

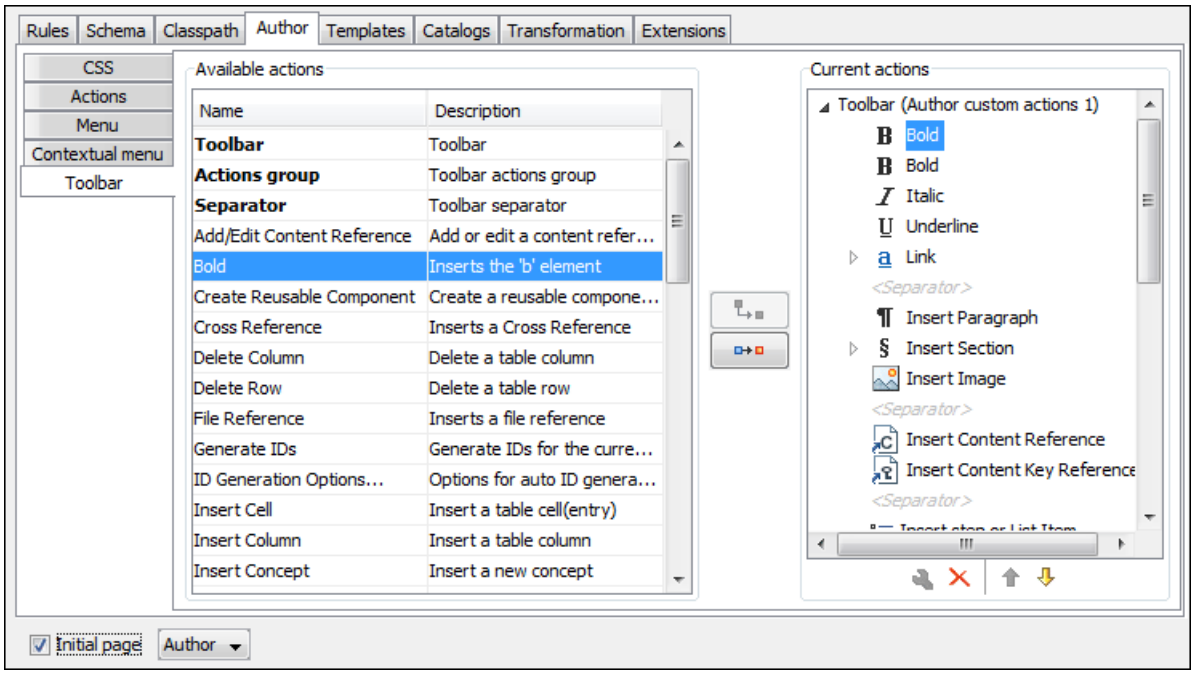


Figure 5: Configuring the Toolbar

The panel is divided in two sections: the left side contains a list of actions, while the right one contains an action tree, displaying the list of actions added in the toolbar. The special entry called *Separator* allows you to visually separate the actions in the toolbar.

2. Select the **Insert section** action in the left panel section and the **Toolbar** label in the right panel section, then press the **Add as child** button.
3. Select the **Insert table** action in the left panel section and the **Insert section** in the right panel section. Press the **Add as sibling** button.
4. When opening a **Simple Documentation Framework** test document in Author mode, the toolbar below will be displayed at the top of the editor.

Figure 6: Author Custom Actions Toolbar

Tip: If you have many custom toolbar actions or want to group actions according to their category you can add additional toolbars with custom names and split the actions to better suit your purpose.

Tip: If your toolbar is not showing when switching to the **Author** mode, check if the toolbar was accidentally hidden: display the contextual menu by clicking in the upper part of application window, and check if the entry labeled **Author custom actions 1** is checked.

Configuring the Main Menu

Defined actions can be grouped into customized menus in the menu bar.

1. Open the Document Type dialog for the **SDF** framework and click on the **Author** tab.
2. Click on the **Menu** label. In the left side you have the list of actions and some special entries:
   - **Submenu** - Creates a submenu. You can nest an unlimited number of menus.
   - **Separator** - Creates a separator into a menu. This way you can logically separate the menu entries.
3. The right side of the panel displays the menu tree with **Menu** entry as root. To change its name click on this label to select it, then press the **Edit** button. Enter **SD Framework** as name, and **D** as menu access key.
4. Select the **Submenu** label in the left panel section and the **SD Framework** label in the right panel section, then press the **Add as child** button. Change the submenu name to **Table**, using the **Edit** button.
5. Select the **Insert section** action in the left panel section and the **Table** label in the right panel section, then press the **Add as sibling** button.
6. Now select the **Insert table** action in the left panel section and the **Table** in the right panel section. Press the **Add as child** button.

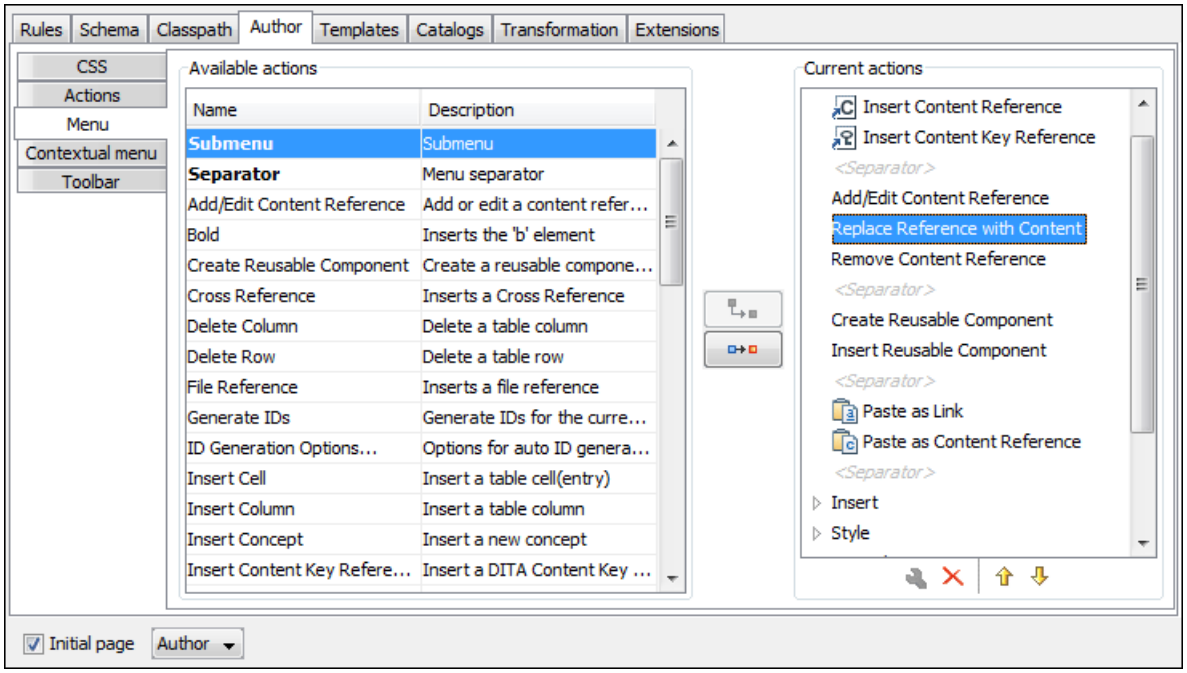


Figure 7: Configuring the Menu

When opening a **Simple Documentation Framework** test document in Author mode, the menu you created is displayed in the editor menu bar, between the **Tools** and the **Document** menus. The upper part of the menu contains generic Author actions (common to all document types) and the two actions created previously (with **Insert table** under the **Table** submenu).

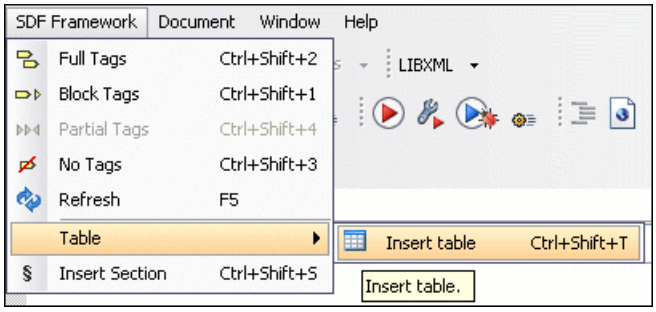


Figure 8: Author Menu

Configuring the Contextual Menu

The contextual menu is shown when you right click (**ctrl + mouse click** on Mac) in the Author editing area. In fact you are configuring the bottom part of the menu, since the top part is reserved for a list of generic actions like Copy, Paste, Undo, etc.

1. Open the Document Type dialog for the **SDF** framework and click on the **Author** tab. Next click on the **Contextual Menu** label.
2. Follow the same steps as explained in the *Configuring the Main Menu*, except changing the menu name because the contextual menu does not have a name.

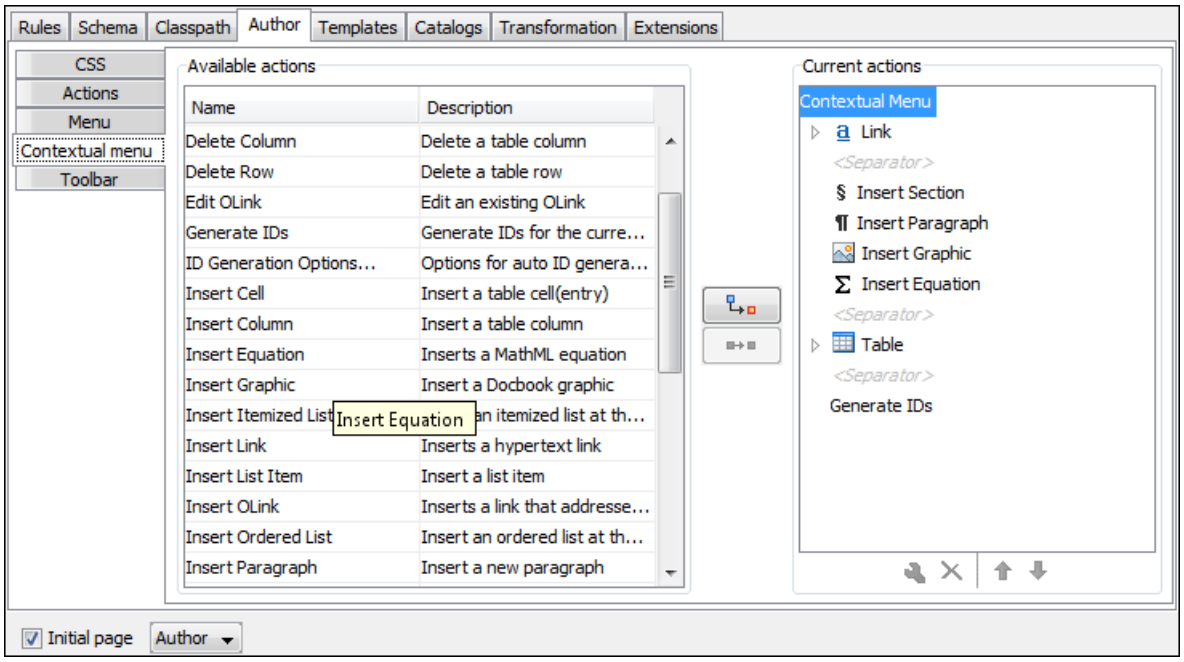


Figure 9: Configuring the Contextual Menu

To test it, open the test file, and open the contextual menu. In the lower part there is shown the **Table** sub-menu and the **Insert section** action.

Customize Content Completion

You can customize the content of the following **Author** controls, adding items (which, when invoked, perform custom actions) or filtering the default contributed ones:

- Content Completion window;
- **Elements** view;
- **Element Insert** menus (from the **Outline** view or breadcrumb contextual menus).

You can use the content completion customization support in the *Simple Documentation Framework* following the next steps:

1. Open the **Document type** edit dialog for the **SDF** framework and select the **Author** tab. Next click on the **Content Completion** tab.

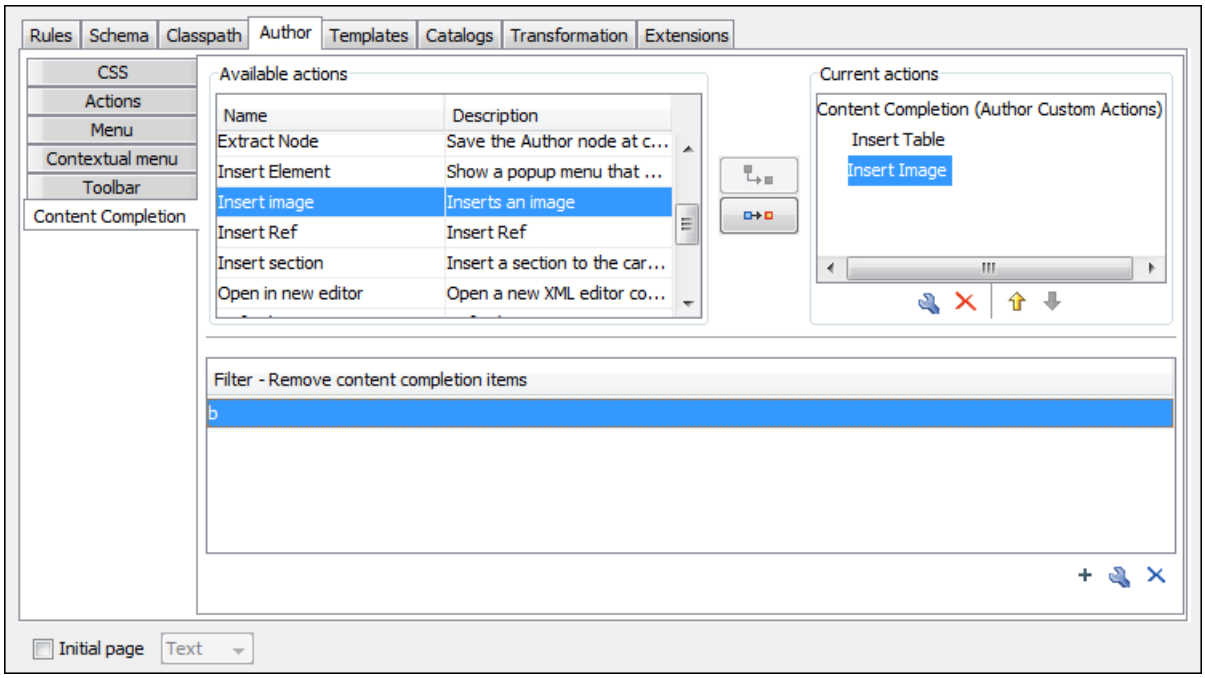


Figure 10: Customize Content Completion

The top side of the **Content Completion** section contains the list with all the actions defined within the simple documentation framework and the list of actions that you decided to include in the **Content Completion** items lists. The bottom side contains the list with all the items that you decided to remove from the **Content Completion** items lists.

2. If you want to add a custom action to the list of current **Content Completion** items, select the action item from the **Available actions** list and press the **Add as child** or **Add as sibling** button to include it in the **Current actions** list. The following dialog appears, giving you the possibility to select where to provide the selected action:

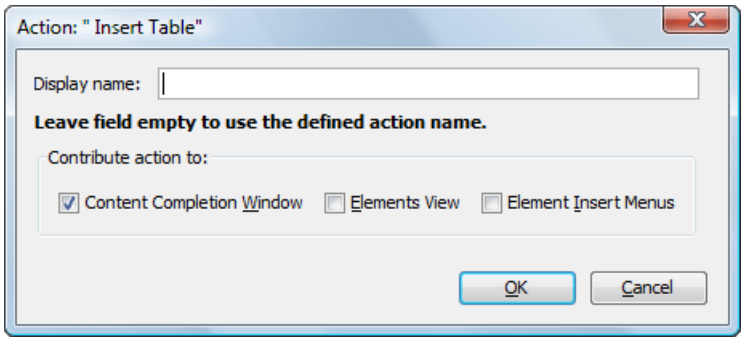


Figure 11: Insert action dialog

3. If you want to exclude a certain item from the **Content Completion** items list, you can use the **Add** button from the **Filter - Remove content completion items** list. The following dialog is displayed, allowing you to input the item name and to choose the controls that filter it.

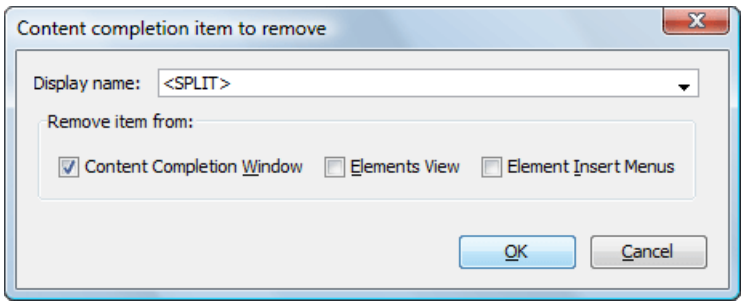


Figure 12: Remove item dialog

Author Default Operations

Below are listed all the operations and their arguments.

`InsertFragmentOperation`

Inserts an XML fragment at the current cursor position. The selection - if there is will be inserted in the current context of the cursor position meaning that if the curre declarations then the inserted fragment must use the same declarations. The inser pasted to the cursor position, but the namespace declarations of the fragment will namespace declarations of the XML document. For more details about the list of *InsertFragmentOperation operation* on page 21.

`InsertOrReplaceFragmentOperation`

Similar to `InsertFragmentOperation`, except it removes the selected cont

`InsertOrReplaceTextOperation`

Inserts a text at current position removing the selected content, if any.

`text`	The text section to insert.

`SurroundWithFragmentOperation`

Surrounds the selected content with a text fragment. Since the fragment can have will be always placed in the first leaf element. If there is no selection, the operatio caret position. For more details about the list of parameters go to *The arguments o* page 22.

`SurroundWithTextOperation`

This operation has two arguments (two text values) that will be inserted before ar no selected content, the two sections will be inserted at the caret position. The arg

`header`	The text that will be placed before the selectio
`footer`	The text that will be placed after the selection

`InsertEquationOperation`

Inserts a fragment containing a MathML equation at caret offset. The operation a

`fragment`	The XML fragment containing the MathML cor

`InsertXIncludeOperation`

Insert an XInclude element at caret offset.

`ChangeAttributeOperation`

This operation allows adding/modifying/removing an attribute. You can use this o modify the value for a certain attribute on a specific XML element. The argumen

`name`	The attribute local name.
`namespace`	The attribute namespace.
`elementLocation`	The XPath location that identifie
`value`	The new value for the attribute. If empty or null t
`editAttribute`	If an in-place editor exists for this attribute, it will a and start editing.

`removeIfEmpty` The possible values are **true** and **false**. True mean
if an empty value is provided. The default behavi

`UnwrapTagsOperation` This operation allows removing the element tags either from the current element or
location. The argument of the operation is:

`unwrapElementLocation` An XPath expression indicating the elem
the element at caret is unwrapped.

`ToggleSurroundWithElementOperation` This operation allows wrapping and unwrapping content in a specific element with
toggle actions like highlighting text as bold, italic, or underline. The arguments o

`element` The element to wrap content (or unwrap).

`schemaAware` This argument applies only on the surround with
insertion is schema aware or not.

`XSLTOperation` and `XQueryOperation` Applies an XSLT or XQuery script on a source element and then replaces or insert

This operation has the following parameters:

`sourceLocation` An XPath expression indicating the element that the script w
the element at the caret position will be used.

There may be situations in which you want to look at an an
decisions in the script based on this. In order to do this you
ancestor node (for example `/`) then declare a parameter calle
and use it to re-position in the current element like:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/T
  xpath-default-namespace="http://docbook.org/ns/docboo
 exclude-result-prefixes="saxon">
    <!-- This is an XPath location which will be sent b
    <xsl:param name="currentElementLocation"/>

    <xsl:template match="/">
        <!-- Evaluate the XPath of the current element
       <xsl:apply-templates select="saxon:eval(saxon:ex

    </xsl:template>

    <xsl:template match="para">
        <!-- And the context is again inside the curren
 from the entire XML -->
        <xsl:variable name="keyImage"
select="//imagedata[@fileref='images/lake.jpeg']/ancestor

                <xref linkend="{$keyImage}" role="key_i
xmlns="http://docbook.org/ns/docbook">
                    <xsl:value-of select="$currentEleme
        </xref>
    </xsl:template>
</xsl:stylesheet>
```

`targetLocation` An XPath expression indicating the insert locatio
If it is not defined then the insert location will be

`script` The script content (XSLT or XQuery). The base system ID for this
include/import reference will be resolved relative to the `.framew`
definition.

For example if your script has the following value:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transfo
    <xsl:import href="xslt_operation.xsl"/>
</xsl:stylesheet>
```

then the imported `xslt_operation.xsl` needs to be located i

`action`	The insert action relative to the node determined be: Replace, At caret position, Before, After, Insi
`caretPosition`	The position of the caret after the action is execu First editable position, End or After. If not specifi outputting in the XSLT script a **${caret}** editor v
`expandEditorVariables`	Parameter controlling the expansion of editor vari Expansion is enabled by default.

Author operations can take parameters that might contain the following editor variables:

- **${caret}** - The position where the caret is inserted. This variable can be used in a code template , in **Author** operations, or in a selection plugin.
- **${selection}** - The XML content of the current selection in the editor panel. This variable can be used in a code template and Author operations, or in a selection plugin.
- **${ask('message', type, ('real_value1':'rendered_value1'; 'real_value2':'rendered_value2'; ...), 'default_value')}** - To prompt for values at runtime, use the *ask('message', type, ('real_value1':'rendered_value1'; 'real_value2':'rendered_value2'; ...), 'default-value'')* editor variable. The following parameters can be set:
  - `'message'` - the displayed message. Note the quotes that enclose the message.
  - `type` - optional parameter. Can have one of the following values:
    - `url` - input is considered an URL. checks that the URL is valid before passing it to the transformation;
    - `password` - input characters are hidden;
    - `generic` - the input is treated as generic text that requires no special handling;
    - `relative_url` - input is considered an URL. tries to make the URL relative to that of the document you are editing.

      Note: You can use the `$ask` editor variable in file templates. In this case, keeps an absolute URL.

    - `combobox` - displays a dialog that contains a non-editable combo-box;
    - `editable_combobox` - displays a dialog that contains an editable combo-box;
    - `radio` - displays a dialog that contains radio buttons;
  - `'default-value'` - optional parameter. Provides a default value in the input text box.

    Examples:

    - `${ask('message')}` - Only the message displayed for the user is specified.
    - `${ask('message', generic, 'default')}` - `'message'` is displayed, the type is not specified (the default is string), the default value is `'default'`.
    - `${ask('message', password)}` - `'message'` is displayed, the characters typed are masked with a circle symbol.
    - `${ask('message', password, 'default')}` - same as before, the default value is `'default'`.
    - `${ask('message', url)}` - `'message'` is displayed, the parameter type is URL.
    - `${ask('message', url, 'default')}` - same as before, the default value is `'default'`.

- **${timeStamp}** - Time stamp, that is the current time in Unix format. It can be used for example to save transformation results in different output files on each transform.
- **${uuid}** - Universally unique identifier.
- **${id}** - Application-level unique identifier.
- **${cfn}** - Current file name without extension and without parent folder.
- **${cfne}** - Current file name with extension.
- **${cf}** - Current file as file path, that is the absolute file path of the current edited document.

- **${cfd}** - Current file folder as file path, that is the path of the current edited document up to the name of the parent folder.
- **${frameworksDir}** - The path (as file path) of the `frameworks` subfolder of the installation folder.
- **${pd}** - Current project folder as file path.
- **${oxygenInstallDir}** - installation folder as file path.
- **${homeDir}** - The path (as file path) of the user home folder.
- **${pn}** - Current project name.
- **${env(VAR_NAME)}** - Value of the *VAR_NAME* environment variable. The environment variables are managed by the operating system. If you are looking for Java System Properties, use the **${system(var.name)}** editor variable.
- **${system(var.name)}** - Value of the *var.name* Java system property. The Java system properties can be specified in the command line arguments of the Java runtime as `-Dvar.name=var.value`. If you are looking for operating system environment variables, use the**${env(VAR_NAME)}** editor variable instead.
- **${date(pattern)}** - Current date. Follows the given pattern. Example: yyyy-MM-dd.

The arguments of `InsertFragmentOperation` operation

fragment

The value for this argument is a text. This is parsed by Author as it was already in the document at the caret position. You can use entity references declared in the document and it is namespace aware. The fragment may have multiple roots.

Note:

You can even use namespace prefixes that are not declared in the inserted fragment, if they are declared in the document where the insertion is done. For the sake of clarity, you should always prefix and declare namespaces in the inserted fragment!

Note:

If the fragment contains namespace declarations that are identical to those found in the document, the namespace declaration attributes will be removed from elements contained by the inserted fragment.

There are two possible scenarios:

1. **Prefixes that are not bound explicitly**

   For instance, the fragment:

   ```
   <x:item id="dty2"/>
   &ent;
   <x:item id="dty3"/>
   ```

   Can be correctly inserted in the document: ('|' marks the insertion point):

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <!DOCTYPE x:root [
       <!ENTITY ent "entity">
   ]>

   <x:root xmlns:x="nsp">
     |
   </x:root>
   ```

   Result:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <!DOCTYPE x:root [
       <!ENTITY ent "entity">
   ]>
   <x:root xmlns:x="nsp">
       <x:item id="dty2"/>
       &ent;
       <x:item id="dty3"/>
   </x:root>
   ```

2. Default namespaces

If there is a default namespace declared in the document and the document fragment does not declare a namespace, the elements from the fragment are considered to be in **no namespace**.

For instance the fragment:

```
<item id="dty2"/>
<item id="dty3"/>
```

Inserted in the document:

```
<?xml version="1.0" encoding="UTF-8"?>
<root xmlns="nsp">
|
</root>
```

Gives the result document:

```
<?xml version="1.0" encoding="UTF-8"?>
<root xmlns="nsp">
    <item xmlns="" id="dty2"/>
    <item xmlns="" id="dty3"/>
</root>
```

`insertLocation` An XPath expression that is relative to the current node. It selects the reference node for the fragment insertion.

`insertPosition` One of the three constants: "**Inside**", "**After**", or "**Before**" , showing where the insertion is made relative to the reference node selected by the `insertLocation`. "**Inside**" has the meaning of the first child of the reference node.

`goToNextEditablePosition` After inserting the fragment, the first editable position is detected and the caret is placed at that location. It handles any in-place editors used to edit attributes. It will be ignored if the fragment specifies a caret position using the caret editor variable. The possible values of this action are **true** and **false**.

The arguments of `SurroundWithFragmentOperation`

The Author operation `SurroundWithFragmentOperation` has only one argument:

- fragment -

  The XML fragment that will surround the selection. For example let's consider the fragment:

```
<F>
   <A></A>
   <B>
     <C></C>
   </B>
</F>
```

  and the document:

```
<doc>
  <X></X>
  <Y></Y>
  <Z></Z>
<doc>
```

  Considering the selected content to be surrounded is the sequence of elements `X` and `Y`, then the result is:

```
<doc>
   <F>
      <A>
         <X></X>
         <Y></Y>
      </A>
      <B>
```

```
        <C></C>
      </B>
   </F>
  <Z></Z>
<doc>
```

Because the element A was the first leaf in the fragment, it received the selected content. The fragment was then inserted in the place of the selection.

How to Add a Custom Action to an Existing Document Type

This task explains how to add a custom Author operation to an existing document type.

1. Download the Author SDK toolkit:*http://www.oxygenxml.com/developer.html#XML_Editor_Authoring_SDK*
2. Create a Java project with a custom implementation of *ro.sync.ecss.extensions.api.AuthorOperation* which performs your custom operation and updates the Author mode using our API like: `AuthorAccess.getDocumentController().insertXMLFragment`.
3. Pack the operation class inside a Java *jar* library.
4. Copy the *jar* library to the `OXYGEN_INSTALL_DIR/frameworks/framework_dir` directory.
5. Go to Oxygen **Preferences** > **Document Type Association** page and edit the document type (you need write access to the `OXYGEN_INSTALLATION_DIR`).
   a) In the **Classpath** tab, add a new entry like: `${frameworks}/docbook/customAction.jar`.
   b) In the **Author** tab, add a new action which uses your custom operation.
   c) Mount the action to the toolbars or menus.
6. Share the modifications with your colleagues. The files which should be shared are your `customAction.jar` library and the `.framework` configuration file from the `OXYGEN_INSTALL_DIR/frameworks/framework_dir` directory.

Java API - Extending Author Functionality through Java

Author has a built-in set of operations covering the insertion of text and XML fragments (see the *Author Default Operations*) and the execution of XPath expressions on the current document edited in Author mode. However, there are situations in which you need to extend this set. For instance if you need to enter an element whose attributes should be edited by the user through a graphical user interface. Or the users must send the selected element content or even the whole document to a server, for some kind of processing or the content authors must extract pieces of information from a server and insert it directly into the edited XML document. Or you need to apply an XPath expression on the current Author document and process the nodes of the result nodeset.

The following sections contain the Java programming interface (API) available to the developers. You will need the *Oxygen Author SDK* available *on the website* which includes the source code of the Author operations in the predefined document types and the full documentation in Javadoc format of the public API available for the developer of Author custom actions.

The next Java examples are making use of AWT classes. If you are developing extensions for the XML Editor plugin for Eclipse you will have to use their SWT counterparts.

It is assumed you already read the *Configuring Actions, Menus, Toolbar* section and you are familiar with the Author customization. You can find the XML schema, CSS and XML sample in the *Example Files Listings*.

Attention:

Make sure the Java classes of your custom Author operations are compiled with the same Java version used by . Otherwise the classes may not be loaded by the Java virtual machine. For example if you run with a Java 1.6 virtual machine but the Java classes of your custom Author operations are compiled with a Java 1.7 virtual machine then the custom operations cannot be loaded and used by the Java 1.6 virtual machine.

Example 1. Step by Step Example. Simple Use of a Dialog from an Author Operation.

Let's start adding functionality for inserting images in the **Simple Documentation Framework** (shortly SDF). The images are represented by the `image` element. The location of the image file is represented by the value of the `href`

attribute. In the Java implementation you will show a dialog with a text field, in which the user can enter a full URL, or he can browse for a local file.

1. Create a new Java project, in your IDE of choice. Create the `lib` folder in the project folder. Copy the `oxygen.jar` file from the `{oXygen_installation_directory}/lib` folder into the newly created `lib` folder. `oxygen.jar` contains the Java interfaces you have to implement and the API needed to access the Author features.
2. Create the `simple.documentation.framework.InsertImageOperation` class that implements the *`ro.sync.ecss.extensions.api.AuthorOperation`* interface. This interface defines three methods: `doOperation`, `getArguments` and `getDescription`

   A short description of these methods follows:

   - The `doOperation` method is invoked when the action is performed either by pressing the toolbar button, by selecting the menu item or by pressing the shortcut key. The arguments taken by this methods can be one of the following combinations:
     - an object of type *`ro.sync.ecss.extensions.api.AuthorAccess`* and a map
     - argument names and values
   - The `getArguments` method is used by when the action is configured. It returns the list of arguments (name and type) that are accepted by the operation.
   - The `getDescription` method is used by when the operation is configured. It returns a description of the operation.

   Here is the implementation of these three methods:

```
/**
 * Performs the operation.
 */
public void doOperation(
               AuthorAccess authorAccess,
               ArgumentsMap arguments)
 throws IllegalArgumentException,
                 AuthorOperationException {

 JFrame oxygenFrame = (JFrame) authorAccess.getParentFrame();
 String href = displayURLDialog(oxygenFrame);
 if (href.length() != 0) {
     // Creates the image XML fragment.
     String imageFragment =
        "<image xmlns='http://www.oxygenxml.com/sample/documentation' href='"
        + href + "'/>";

     // Inserts this fragment at the caret position.
     int caretPosition = authorAccess.getCaretOffset();
     authorAccess.insertXMLFragment(imageFragment, caretPosition);
 }
}

/**
 * Has no arguments.
 *
 * @return null.
 */
public ArgumentDescriptor[] getArguments() {
 return null;
}

/**
 * @return A description of the operation.
 */
public String getDescription() {
 return "Inserts an image element. Asks the user for a URL reference.";
}
```

Note: The complete source code can be found in the Simple Documentation Framework project, included in the *Oxygen Author SDK zip* available for download *on the website*.

Important:

Make sure you always specify the namespace of the inserted fragments.

```
<image xmlns='http://www.oxygenxml.com/sample/documentation'
  href='path/to/image.png'/>
```

3. Package the compiled class into a jar file. An example of an ANT script that packages the `classes` folder content into a jar archive named `sdf.jar` is listed below:

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="project" default="dist">
    <target name="dist">
   <jar destfile="sdf.jar" basedir="classes">
    <fileset dir="classes">
     <include name="**/*"/>
     </fileset>
   </jar>
    </target>
</project>
```

4. Copy the `sdf.jar` file into the `frameworks / sdf` folder.
5. Add the `sdf.jar` to the Author class path. To do this, open the **Options** > **Preferences** > **Document Type Association** dialog, select **SDF** and press the **Edit** button.
6. Select the **Classpath** tab in the lower part of the dialog and press the + **Add** button . In the displayed dialog enter the location of the jar file, relative to the `frameworks` folder.
7. Let's create now the action which will use the defined operation. Click on the **Actions** label. Copy the icon files for the menu item and for the toolbar in the `frameworks / sdf` folder.
8. Define the action's properties:
   - Set **ID** to **insert_image**.
   - Set **Name** to **Insert image**.
   - Set **Menu access key** to letter **i**.
   - Set **Toolbar action** to **${frameworks}/sdf/toolbarImage.png**.
   - Set **Menu icon** to **${frameworks}/sdf/menuImage.png**.
   - Set **Shortcut key** to **Ctrl+Shift+i**.
9. Now let's set up the operation. You want to add images only if the current element is a `section`, `book` or `article`.
   - Set the value of **XPath expression** to

     ```
     local-name()='section' or local-name()='book'
      or local-name()='article'
     ```

   - Set the **Invoke operation** field to `simple.documentation.framework.InsertImageOperation`.
10. Add the action to the toolbar, using the **Toolbar** panel.

To test the action, you can open the *`sdf_sample.xml`* sample, then place the caret inside a `section` between two para elements for instance. Press the button associated with the action from the toolbar. In the dialog select an image URL and press **OK**. The image is inserted into the document.

Example 2. Operations with Arguments. Report from Database Operation.

In this example you will create an operation that connects to a relational database and executes an SQL statement. The result should be inserted in the edited XML document as a `table`. To make the operation fully configurable, it will have arguments for the *database connection string*, the *user name*, the *password* and the *SQL expression*.

1. Create a new Java project in your preferred IDE. Create the `lib` folder in the Java project directory and copy the `oxygen.jar` file from the `{oXygen_installation_directory}/lib` directory.

2. Create the class `simple.documentation.framework.QueryDatabaseOperation`. This class must implements the *`ro.sync.ecss.extensions.api.AuthorOperation`* interface.

```
import ro.sync.ecss.extensions.api.ArgumentDescriptor;
import ro.sync.ecss.extensions.api.ArgumentsMap;
import ro.sync.ecss.extensions.api.AuthorAccess;
import ro.sync.ecss.extensions.api.AuthorOperation;
import ro.sync.ecss.extensions.api.AuthorOperationException;

public class QueryDatabaseOperation implements AuthorOperation{
```

3. Now define the operation's arguments. For each of them you will use a `String` constant representing the argument name:

```
private static final String ARG_JDBC_DRIVER ="jdbc_driver";
private static final String ARG_USER ="user";
private static final String ARG_PASSWORD ="password";
private static final String ARG_SQL ="sql";
private static final String ARG_CONNECTION ="connection";
```

4. You must describe each of the argument name and type. To do this implement the `getArguments` method which will return an array of argument descriptors:

```
public ArgumentDescriptor[] getArguments() {
  ArgumentDescriptor args[] = new ArgumentDescriptor[] {
    new ArgumentDescriptor(
      ARG_JDBC_DRIVER,
      ArgumentDescriptor.TYPE_STRING,
      "The name of the Java class that is the JDBC driver."),
    new ArgumentDescriptor(
      ARG_CONNECTION,
      ArgumentDescriptor.TYPE_STRING,
      "The database URL connection string."),
    new ArgumentDescriptor(
      ARG_USER,
      ArgumentDescriptor.TYPE_STRING,
      "The name of the database user."),
    new ArgumentDescriptor(
      ARG_PASSWORD,
      ArgumentDescriptor.TYPE_STRING,
      "The database password."),
    new ArgumentDescriptor(
      ARG_SQL,
      ArgumentDescriptor.TYPE_STRING,
      "The SQL statement to be executed.")
  };
  return args;
 }
```

These names, types and descriptions will be listed in the **Arguments** table when the operation is configured.

5. When the operation is invoked, the implementation of the `doOperation` method extracts the arguments, forwards them to the method that connects to the database and generates the XML fragment. The XML fragment is then inserted at the caret position.

```
public void doOperation(AuthorAccess authorAccess, ArgumentsMap map)
   throws IllegalArgumentException, AuthorOperationException {

  // Collects the arguments.
  String jdbcDriver =
   (String)map.getArgumentValue(ARG_JDBC_DRIVER);
  String connection =
   (String)map.getArgumentValue(ARG_CONNECTION);
  String user =
   (String)map.getArgumentValue(ARG_USER);
  String password =
   (String)map.getArgumentValue(ARG_PASSWORD);
  String sql =
   (String)map.getArgumentValue(ARG_SQL);

  int caretPosition = authorAccess.getCaretOffset();
  try {
   authorAccess.insertXMLFragment(
     getFragment(jdbcDriver, connection, user, password, sql),
     caretPosition);
  } catch (SQLException e) {
   throw new AuthorOperationException(
     "The operation failed due to the following database error: "
     + e.getMessage(), e);
```

```
  } catch (ClassNotFoundException e) {
   throw new AuthorOperationException(
     "The JDBC database driver was not found. Tried to load ' "
     + jdbcDriver + "'", e);
  }
 }
```

6. The `getFragment` method loads the JDBC driver, connects to the database and extracts the data. The result is a `table` element from the `http://www.oxygenxml.com/sample/documentation` namespace. The `header` element contains the names of the SQL columns. All the text from the XML fragment is escaped. This means that the '<' and '&' characters are replaced with the '<' and '&' character entities to ensure the fragment is well-formed.

```
private String getFragment(
  String jdbcDriver,
  String connectionURL,
  String user,
  String password,
  String sql) throws
   SQLException,
   ClassNotFoundException {

      Properties pr = new Properties();
      pr.put("characterEncoding", "UTF8");
      pr.put("useUnicode", "TRUE");
      pr.put("user", user);
      pr.put("password", password);

      // Loads the database driver.
      Class.forName(jdbcDriver);
      // Opens the connection
      Connection connection =
         DriverManager.getConnection(connectionURL, pr);
      java.sql.Statement statement =
         connection.createStatement();
      ResultSet resultSet =
         statement.executeQuery(sql);

      StringBuffer fragmentBuffer = new StringBuffer();
      fragmentBuffer.append(
        "<table xmlns=" +
        "'http://www.oxygenxml.com/sample/documentation'>");

      //
      // Creates the table header.
      //
      fragmentBuffer.append("<header>");
      ResultSetMetaData metaData = resultSet.getMetaData();
      int columnCount = metaData.getColumnCount();
      for (int i = 1; i <= columnCount; i++) {
          fragmentBuffer.append("<td>");
          fragmentBuffer.append(
            xmlEscape(metaData.getColumnName(i)));
          fragmentBuffer.append("</td>");
      }
      fragmentBuffer.append("</header>");

      //
      // Creates the table content.
      //
      while (resultSet.next()) {
          fragmentBuffer.append("<tr>");
          for (int i = 1; i <= columnCount; i++) {
              fragmentBuffer.append("<td>");
              fragmentBuffer.append(
                xmlEscape(resultSet.getObject(i)));
              fragmentBuffer.append("</td>");
          }
          fragmentBuffer.append("</tr>");
      }

      fragmentBuffer.append("</table>");

      // Cleanup
      resultSet.close();
      statement.close();
      connection.close();
      return fragmentBuffer.toString();
}
```

Note: The complete source code can be found in the Simple Documentation Framework project, included in the *Oxygen Author SDK zip* available for download *on the website*.

7. Package the compiled class into a jar file.
8. Copy the jar file and the JDBC driver files into the `frameworks / sdf` directory.
9. Add the jars to the Author class path. For this, Open the options Document Type Dialog, select **SDF** and press the **Edit** button. Select the **Classpath** tab in the lower part of the dialog.
10. Click on the **Actions** label. The action properties are:
    - Set **ID** to **clients_report**.
    - Set **Name** to **Clients Report**.
    - Set **Menu access key** to letter **r**.
    - Set **Description** to **Connects to the database and collects the list of clients**.
    - Set **Toolbar icon** to **${frameworks}/sdf/TableDB20.png** (image  `TableDB20.png` is already stored in the `frameworks / sdf` folder).
    - Leave empty the **Menu icon**.
    - Set **shortcut key** to **Ctrl+Shift+C**.
11. The action will work only if the current element is a **section**. Set up the operation as follows:
    - Set **XPath expression** to:

      ```
      local-name()='section'
      ```

    - Use the Java operation defined earlier to set the **Invoke operation** field. Press the **Choose** button, then select `simple.documentation.framework.QueryDatabaseOperation`. Once selected, the list of arguments is displayed. In the figure below the first argument, *jdbc_driver*, represents the class name of the MySQL JDBC driver. The connection string has the URL syntax : *jdbc://<database_host>:<database_port>/<database_name>*.

      The SQL expression used in the example follows, but it can be any valid SELECT expression which can be applied to the database:

      ```
      SELECT userID, email FROM users
      ```

12. Add the action to the toolbar, using the **Toolbar** panel.

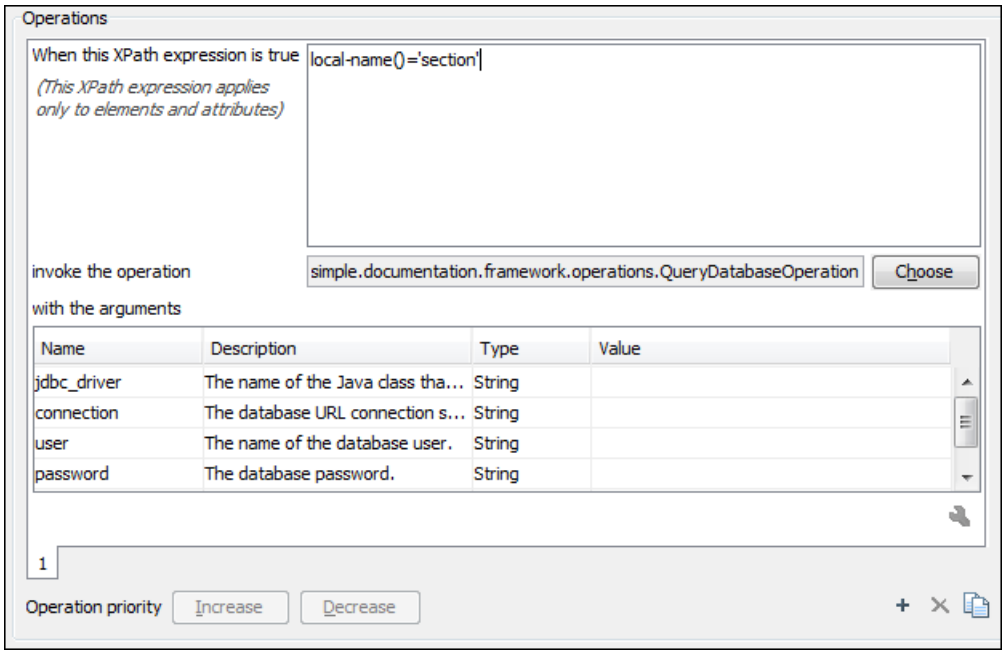


Figure 13: Java Operation Arguments Setup

To test the action you can open the *sdf_sample.xml* sample place the caret inside a `section` between two para elements for instance. Press the **Create Report** button from the toolbar. You can see below the toolbar with the action button and sample table inserted by the **Clients Report** action.

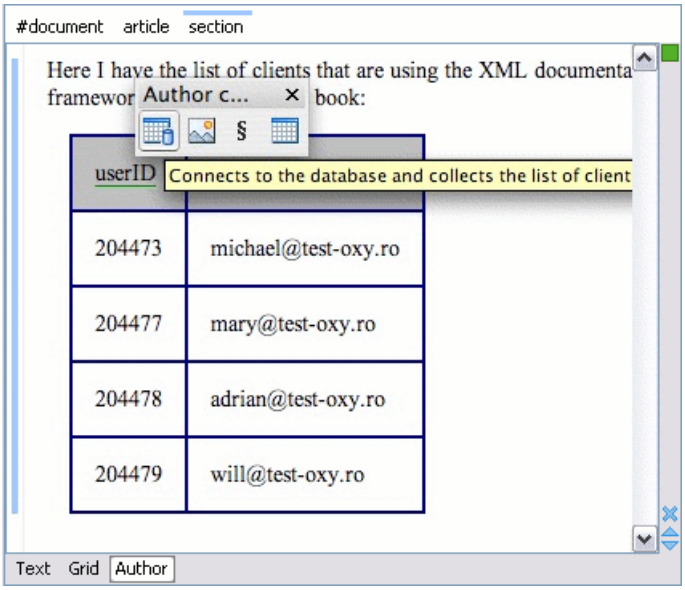


Figure 14: Table Content Extracted from the Database

Editing attributes in-place using form controls

To edit attributes in the **Author** mode, use the in-place attributes editing dialog.

The *oxy_editor* CSS extension function allows you to edit attribute and element text values directly in the Author mode using form-based controls. Various implementations are available out of the box: *combo boxes*, *checkboxes*, *text fields*, *pop-ups*, *buttons* which invoke custom Author actions or *URL choosers*. You can also implement custom editors for your specific needs.

As a working example, the bundled samples project contains a file called `personal.xml` which allows editing attributes in-place using some of these default implementations.

Localizing Frameworks

supports framework localization (translating framework actions, buttons, and menu entries to different languages). Thus you can develop and distribute a framework to users that speak different languages without changing the distributed framework. Changing the language used in Oxygen (in **Options** > **Preferences** > **Global** > **Language** Global preferences page) is enough to set the right language for each framework.

To localize the content of a framework, create a `translation.xml` file which contains all the translation (key, value) mappings. The `translation.xml` has the following format:

```
<translation>
    <languageList>
        <language description="English" lang="en_US"/>
        <language description="German" lang="de_DE"/>
        <language description="French" lang="fr_FR"/>
    </languageList>
    <key value="list">
        <comment>List menu item name.</comment>
        <val lang="en_US">List</val>
        <val lang="de_DE">Liste</val>
        <val lang="fr_FR">Liste</val>
    </key>
.....................
</translation>
```

matches the GUI language with the language set in the `translation.xml` file. In case this language is not found, the first available language declared in the `languagelist` tag for the corresponding framework is used.

Add the directory where this file is located to the **Classpath** list corresponding to the edited document type.

After you create this file, you are able to use the keys defined in it to customize the name and description of:

- framework actions;
- menu entries;
- contextual menus;
- toolbar.

For example, if you want to localize the bold action go to **Options** > **Preferences** > **Document Type Association**. Open the **Document type** dialog, go to **Author** > **Actions**, and rename the bold action to `${i18n(translation_key)}`. Actions with a name format different than `${i18n(translation_key)}` are not localized. `Translation_key` corresponds to the key from the translation.xml file.

Now open the `translation.xml` file and edit the translation entry if it exists or create one if it does not exist. This example presents an entry in the translation.xml file:

```
<key value="translation_key">
        <comment>Bold action name.</comment>
        <val lang="en_US">Bold</val>
        <val lang="de_DE">Bold</val>
        <val lang="fr_FR">Bold</val>
    </key>
```

To use a description from the `translation.xml` file in the Java code used by your custom framework, use the new `ro.sync.ecss.extensions.api.AuthorAccess.getAuthorResourceBundle()` API method to request for a certain key the associated value. In this way all the dialogs that you present from your custom operations can have labels translated in different languages.

Note: You can enter any language you want in the `languagelist` tag and any number of keys.

The `translation.xml` file for the DocBook framework is located here:`[OXYGEN_INSTALL_DIR]/frameworks/docbook/i18n/translation.xml`. In the **Classpath** list corresponding to the Docbook document type the following entry was added: `${framework}/i18n/`.

In **Options** > **Preferences** > **Document Type Association** > **Author** > **Actions**, you can see how the DocBook actions are defined to use these keys for their name and description. If you look in the Java class `ro.sync.ecss.extensions.docbook.table.SADocbookTableCustomizerDialog` available in the Author SDK, you can see how the new `ro.sync.ecss.extensions.api.AuthorResourceBundle` API is used to retrieve localized descriptions for different keys.

How to deploy a framework as an add-on

To deploy a framework as an add-on, follow the next steps:

1. Pack the framework directory as a `ZIP` file;
2. Create a descriptor file. You can use a template that provides. To use this template, go to **File** > **New** and create an descriptor file;
3. Copy the `ZIP` file and the descriptor file to an HTTP server. The URL to this location serves as the **Update Site** URL.

Creating the Basic Association

Let us go through an example of creating a document type and editing an XML document of this type. We will call our document type **Simple Documentation Framework**.

First Step - XML Schema

Our documentation framework will be very simple. The documents will be either `articles` or `books`, both composed of `sections`. The `sections` may contain `titles`, `paragraphs`, `figures`, `tables` and other `sections`. To complete the picture, each section will include a `def` element from another namespace.

The first schema file:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.oxygenxml.com/sample/documentation"
    xmlns:doc="http://www.oxygenxml.com/sample/documentation"
    xmlns:abs="http://www.oxygenxml.com/sample/documentation/abstracts"
    elementFormDefault="qualified">

    <xs:import namespace=
    "http://www.oxygenxml.com/sample/documentation/abstracts"
     schemaLocation=
    "abs.xsd"/>
```

The namespace of the documents will be `http://www.oxygenxml.com/sample/documentation`. The namespace of the `def` element is `http://www.oxygenxml.com/sample/documentation/abstracts`.

Now let's define the structure of the sections. They all start with a title, then have the optional `def` element then either a sequence of other sections, or a mixture of paragraphs, images and tables.

```
<xs:element name="book" type="doc:sectionType"/>
<xs:element name="article" type="doc:sectionType"/>
<xs:element name="section" type="doc:sectionType"/>

<xs:complexType name="sectionType">
    <xs:sequence>
        <xs:element name="title" type="xs:string"/>
        <xs:element ref="abs:def" minOccurs="0"/>
        <xs:choice>
            <xs:sequence>
                <xs:element ref="doc:section" maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:choice maxOccurs="unbounded">
                <xs:element ref="doc:para"/>
                <xs:element ref="doc:image"/>
                <xs:element ref="doc:table"/>
            </xs:choice>
        </xs:choice>
    </xs:sequence>
</xs:complexType>
```

The paragraph contains text and other styling markup, such as bold (`b`) and italic (`i`) elements.

```
<xs:element name="para" type="doc:paragraphType"/>

<xs:complexType name="paragraphType" mixed="true">
    <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="b"/>
        <xs:element name="i"/>
    </xs:choice>
</xs:complexType>
```

The `image` element has an attribute with a reference to the file containing image data.

```
<xs:element name="image">
    <xs:complexType>
        <xs:attribute name="href" type="xs:anyURI" use="required"/>
    </xs:complexType>
</xs:element>
```

The `table` contains a header row and then a sequence of rows (`tr` elements) each of them containing the cells. Each cell has the same content as the paragraphs.

```
 <xs:element name="table">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="header">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="td" maxOccurs="unbounded"
```

```
                        type="doc:paragraphType"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
        <xs:element name="tr" maxOccurs="unbounded">
            <xs:complexType>
                <xs:sequence>
                    <xs:element name="td" type="doc:tdType"
                        maxOccurs="unbounded"/>
                </xs:sequence>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>


<xs:complexType name="tdType">
    <xs:complexContent>
        <xs:extension base="doc:paragraphType">
            <xs:attribute name="row_span" type="xs:integer"/>
            <xs:attribute name="column_span" type="xs:integer"/>
        </xs:extension>
    </xs:complexContent>
</xs:complexType>
```

The `def` element is defined as a text only element in the imported schema `abs.xsd`:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace=
     "http://www.oxygenxml.com/sample/documentation/abstracts">
    <xs:element name="def" type="xs:string"/>
</xs:schema>
```

Now the XML data structure will be styled.

Schema Settings

In *the dialog for editing the document type properties*, in the bottom section there are a series of tabs. The first one refers to the schema that is used for validation of the documents that match the defined **Association Rules**.

Important:

If the document refers a schema, using for instance a `DOCTYPE` declaration or a `xsi:schemaLocation` attribute, the schema from the document type association will not be used when validating.

Schema Type	Select from the combo box the value **XML Schema**.
Schema URI	Enter the value `${frameworks}/sdf/schema/sdf.xsd`. We should use the `${frameworks}` editor variable in the schema URI path instead of a full path in order to be valid for different installations.

Important:

The `${frameworks}` variable is expanded at the validation time into the absolute location of the directory containing the frameworks.

Second Step - The CSS

If you read the *Simple Customization Tutorial* then you already have some basic notions about creating simple styles. The example document contains elements from different namespaces, so you will use CSS Level 3 extensions supported by the Author layout engine to associate specific properties with that element.

Defining the General Layout

Now the basic layout of the rendered documents is created.

Elements that are stacked one on top of the other are: `book`, `article`, `section`, `title`, `figure`, `table`, `image`. These elements are marked as having `block` style for display. Elements that are placed one after the other in a flowing sequence are: `b`, `i`. These will have `inline` display.

```
/* Vertical flow */
book,
section,
para,
title,
image,
ref {
    display:block;
}

/* Horizontal flow */
b,i {
    display:inline;
}
```

Important:

Having `block` display children in an `inline` display parent, makes Author change the style of the parent to `block` display.

Styling the `section` Element

The title of any section must be bold and smaller than the title of the parent section. To create this effect a sequence of CSS rules must be created. The `*` operator matches any element, it can be used to match titles having progressive depths in the document.

```
title{
    font-size: 2.4em;
    font-weight:bold;
}
* * title{
    font-size: 2.0em;
}
* * * title{
    font-size: 1.6em;
}
* * * * title{
    font-size: 1.2em;
}
```

It's useful to have before the title a constant text, indicating that it refers to a section. This text can include also the current section number. The `:before` and `:after` pseudo elements will be used, plus the CSS counters.

First declare a counter named `sect` for each `book` or `article`. The counter is set to zero at the beginning of each such element:

```
book,
article{
    counter-reset:sect;
}
```

The `sect` counter is incremented with each `section`, that is a direct child of a `book` or an `article` element.

```
book > section,
article > section{
    counter-increment:sect;
}
```

The "static" text that will prefix the section title is composed of the constant "Section ", followed by the decimal value of the `sect` counter and a dot.

```
book > section > title:before,
article > section > title:before{
    content: "Section " counter(sect) ". ";
}
```

To make the documents easy to read, you add a margin to the sections. In this way the higher nesting level, the larger the left side indent. The margin is expressed relatively to the parent bounds:

```
section{
    margin-left:1em;
    margin-top:1em;
}
```

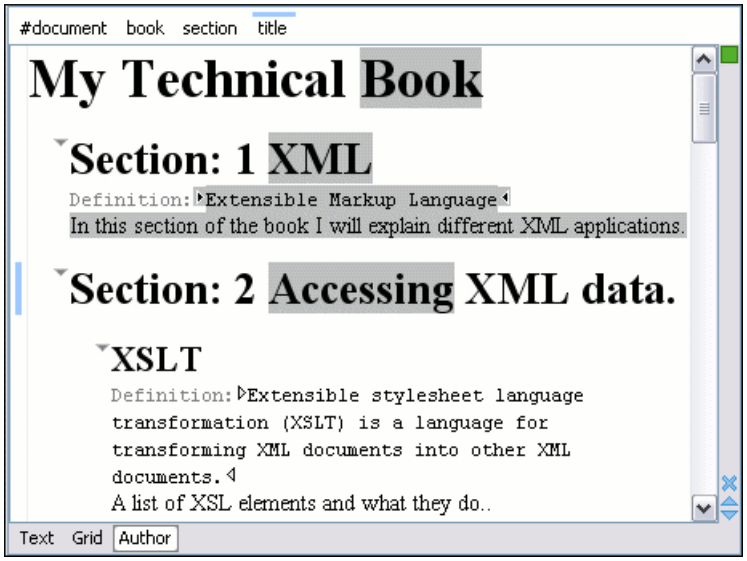


Figure 15: A sample of nested sections and their titles.

In the above screenshot you can see a sample XML document rendered by the CSS stylesheet. The selection "avoids" the text that is generated by the CSS "content" property. This happens because the CSS generated text is not present in the XML document and is just a visual aid.

Styling the Inline Elements

The "bold" style is obtained by using the `font-weight` CSS property with the value `bold`, while the "italic" style is specified by the `font-style` property:

```
b {
    font-weight:bold;
}

i {
    font-style:italic;
}
```

Styling Images

The CSS 2.1 does not specify how an element can be rendered as an image. To overpass this limitation, Author supports a CSS Level 3 extension allowing to load image data from an URL. The URL of the image must be specified by one of the element attributes and it is resolved through the catalogs specified in .

```
image{
    display:block;
    content: attr(href, url);
    margin-left:2em;
}
```

Our `image` element has the required attribute `href` of type `xs:anyURI`. The `href` attribute contains an image location so the rendered content is obtained by using the function:

```
attr(href, url)
```

Important:

The first argument is the name of the attribute pointing to the image file. The second argument of the `attr` function specifies the type of the content. If the type has the `url` value, then identifies the content as being an image. If the type is missing, then the content will be the text representing the attribute value.

Important:

Author handles both absolute and relative specified URLs. If the image has an *absolute* URL location (e.g: "http://www.oasis-open.org/images/standards/oasis_standard.jpg") then it is loaded directly from this location. If the image URL is *relative* specified to the XML document (e.g: "images/my_screenshot.jpg") then the location is obtained by adding this value to the location of the edited XML document.

An image can also be referenced by the name of a DTD entity which specifies the location of the image file. For example if the document declares an entity **graphic** which points to a JPEG image file:

```
<!ENTITY graphic SYSTEM "depo/keyboard_shortcut.jpg" NDATA JPEG>
```

and the image is referenced in the XML document by specifying the name of the entity as the value of an attribute:

```
<mediaobject>
    <imageobject>
        <imagedata entityref="graphic" scale="50"/>
    </imageobject>
</mediaobject>
```

The CSS should use the functions `url`, `attr` and `unparsed-entity-uri` for displaying the image in the Author mode:

```
imagedata[entityref]{
    content: url(unparsed-entity-uri(attr(entityref)));
}
```

To take into account the value of the `width` attribute of the `imagedata` and use it for resizing the image, the CSS can define the following rule:

```
imagedata[width]{
  width:attr(width, length);
}
```

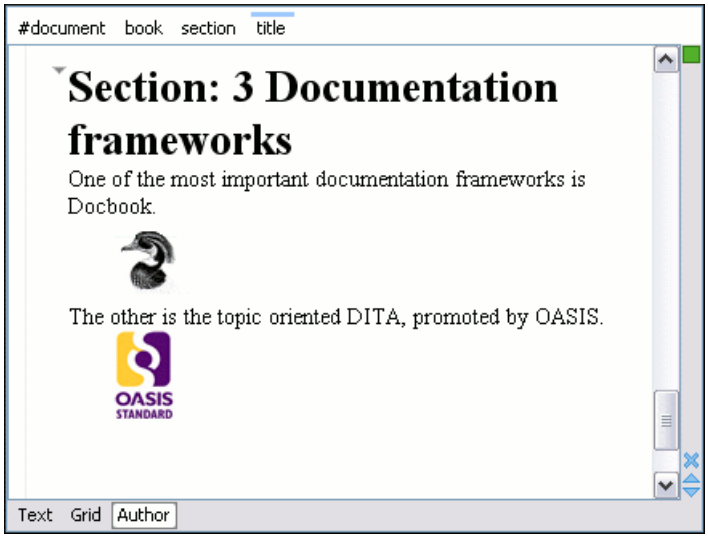


Figure 16: Samples of images in Author

Testing the Document Type Association

To test the new Document Type create an XML instance that is conforming with the *Simple Documentation Framework* association rules. You will not specify an XML Schema location directly in the document, using an `xsi:schemaLocation` attribute; will detect instead its associated document type and use the specified schema.

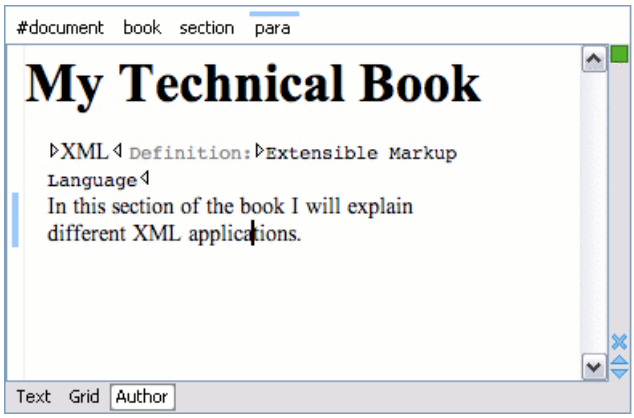
```
<book xmlns="http://www.oxygenxml.com/sample/documentation"
      xmlns:abs="http://www.oxygenxml.com/sample/documentation/abstracts">

    <title>My Technical Book</title>
    <section>
        <title>XML</title>
        <abs:def>Extensible Markup Language</abs:def>
        <para>In this section of the book I will
           explain different XML applications.</para>
    </section>
</book>
```

When trying to validate the document there should be no errors. Now modify the `title` to `title2`. Validate again. This time there should be one error:

```
  cvc-complex-type.2.4.a: Invalid content was found starting with element
  'title2'. One of '{"http://www.oxygenxml.com/sample/documentation":title}'
  is expected.
```

Undo the tag name change. Press on the **Author** button at the bottom of the editing area. should load the CSS from the document type association and create a layout similar to this:



Organizing the Framework Files

First create a new folder called `sdf` (from "Simple Documentation Framework") in `{oXygen_installation_directory}/frameworks`. This folder will be used to store all files related to the documentation framework. The following folder structure will be created:

```
oxygen
  frameworks
     sdf
       schema
       css
```

Important:

The `frameworks` directory is the container where all the oXygen framework customizations are located.

Each subdirectory contains files related to a specific type of XML documents: schemas, catalogs, stylesheets, CSSs, etc.

Distributing a framework means delivering a framework directory.

Important:

It is assumed that you have the right to create files and folder inside the oXygen installation directory. If you do not have this right, you will have to install another copy of the program in a folder you have access to, the home directory for instance, or your desktop. You can download the "all platforms" distribution from the oXygen website and extract it in the chosen folder.

To test your framework distribution you will need to copy it in the `frameworks` directory of the newly installed application and start oXygen by running the provided start-up script files.

You should copy the created schema files `abs.xsd` and `sdf.xsd`, `sdf.xsd` being the master schema, to the `schema` directory and the CSS file `sdf.css` to the `css` directory.

Packaging and Deploying

Using a file explorer, go to the `frameworks` directory. Select the `sdf` directory and make an archive from it. Move it to another installation (eventually on another computer). Extract it in the `frameworks` directory. Start and test the association as explained above.

If you create multiple document type associations and you have a complex directory structure it might be easy from the deployment point of view to use an All Platforms distribution. Add your framework files to it, repackage it and send it to the content authors.

Attention:

When deploying your customized sdf directory please make sure that your sdf directory contains the `sdf.framework` file (that is the file defined as External Storage in Document Type Association dialog shall always be stored inside the sdf directory). If your external storage points somewhere else will not be able to update the Document Type Association options automatically on the deployed computers.

Configuring New File Templates

You will create a set of document templates that the content authors will use as starting points for creating new *Simple Document Framework* books and articles.

Each of the Document Type Associations can point to a directory usually named `templates` containing the file templates. All files found here are considered templates for the respective document type. The template name is taken from the file name, and the template type is detected from the file extension.

1. Create the templates directory into the `frameworks / sdf` directory. The directory tree for the documentation framework is now:

```
oxygen
    frameworks
        sdf
            schema
            css
            templates
```

2. Now let's create in this `templates` directory two files, one for the *book* template and another for the *article* template.

   The `Book.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<book xmlns="http://www.oxygenxml.com/sample/documentation"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:abs="http://www.oxygenxml.com/sample/documentation/abstracts">
    <title>Book Template Title</title>
    <section>
        <title>Section Title</title>
        <abs:def/>
        <para>This content is copyrighted:</para>
        <table>
            <header>
                <td>Company</td>
                <td>Date</td>
            </header>
            <tr>
                <td/>
```

```
                <td/>
            </tr>
        </table>
    </section>
    </book>
```

The `Article.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<article
    xmlns="http://www.oxygenxml.com/sample/documentation"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <title></title>
    <section>
        <title></title>
        <para></para>
        <para></para>
    </section>
</article>
```

You can also use *editor variables* in the template files' content and they will be expanded when the files are opened.

3. Open the Document Type dialog for the **SDF** framework and click on the **Templates** tab. Enter in the **Templates directory** text field the value `${frameworksDir} / sdf / templates`. As you already seen before, it is recommended that all the file references made from a Document Type Association to be relative to the `${frameworksDir}` directory. Binding a Document Type Association to an absolute file (e. g.: "C:\some_dir\templates") makes the association difficult to share between users.
4. To test the templates settings, press the **File**/**New** menu item to display the **New** dialog. The names of the two templates are prefixed with the name of the Document Type Association, in our case **SDF**. Selecting one of them should create a new XML file with the content specified in the template file.

Editor Variables

An editor variable is a shorthand notation for context-dependent information, like a file or folder path, a time-stamp or a date. It is used in the definition of a command (for example the input URL of a transformation, the output file path of a transformation, the command line of an external tool) to make a command or a parameter generic and reusable with other input files. When the same command is applied to different files, the notation is expanded at the execution of the command so that the same command has different effects depending on the actual file.

You can use the following editor variables in commands of external engines or other external tools, in transformation scenarios, and in validation scenarios:

- **${oxygenHome}** - installation folder as URL.
- **${oxygenInstallDir}** - installation folder as file path.
- **${frameworks}** - The path (as URL) of the `frameworks` subfolder of the install folder.
- **${frameworksDir}** - The path (as file path) of the `frameworks` subfolder of the installation folder.
- **${home}** - The path (as URL) of the user home folder.
- **${homeDir}** - The path (as file path) of the user home folder.
- **${pdu}** - Current project folder as URL.
- **${pd}** - Current project folder as file path.
- **${pn}** - Current project name.
- **${cfdu}** - Current file folder as URL, that is the path of the current edited document up to the name of the parent folder, represented as a URL.
- **${cfd}** - Current file folder as file path, that is the path of the current edited document up to the name of the parent folder.
- **${cfn}** - Current file name without extension and without parent folder.
- **${cfne}** - Current file name with extension.
- **${cf}** - Current file as file path, that is the absolute file path of the current edited document.
- **${cfu}** - The path of the current file as a URL.
- **${af}** - The file path of the zip archive that includes the current transformed file (this variable is available only in a transformation scenario).

- **${afu}** - The URL of the zip archive that includes the current transformed file (this variable is available only in a transformation scenario).
- **${afd}** - The directory of the zip archive that includes the current transformed file (this variable is available only in a transformation scenario).
- **${afdu}** - The URL of the directory of the zip archive that includes the current transformed file (this variable is available only in a transformation scenario).
- **${afn}** - The file name (without parent directory and without file extension) of the zip archive that includes the current transformed file (this variable is available only in a transformation scenario).
- **${afne}** - The file name (with file extension, for example `.zip` or `.epub`, but without parent directory) of the zip archive that includes the current transformed file (this variable is available only in a transformation scenario).
- **${currentFileURL}** - Current file as URL, that is the absolute file path of the current edited document represented as URL.
- **${ps}** - Path separator, that is the separator which can be used on the current platform (Windows, Mac OS X, Linux) between library files specified in the class path.
- **${timeStamp}** - Time stamp, that is the current time in Unix format. It can be used for example to save transformation results in different output files on each transform.
- **${caret}** - The position where the caret is inserted. This variable can be used in a code template , in **Author** operations, or in a selection plugin.
- **${selection}** - The XML content of the current selection in the editor panel. This variable can be used in a code template and Author operations, or in a selection plugin.
- **${id}** - Application-level unique identifier.
- **${uuid}** - Universally unique identifier.
- **${env(VAR_NAME)}** - Value of the *VAR_NAME* environment variable. The environment variables are managed by the operating system. If you are looking for Java System Properties, use the **${system(var.name)}** editor variable.
- **${ask('message', type, ('real_value1':'rendered_value1'; 'real_value2':'rendered_value2'; ...), 'default_value')}** - To prompt for values at runtime, use the *ask('message', type, ('real_value1':'rendered_value1'; 'real_value2':'rendered_value2'; ...), 'default-value')* editor variable. The following parameters can be set:
  - `'message'` - the displayed message. Note the quotes that enclose the message.
  - `type` - optional parameter. Can have one of the following values:
    - `url` - input is considered an URL. checks that the URL is valid before passing it to the transformation;
    - `password` - input characters are hidden;
    - `generic` - the input is treated as generic text that requires no special handling;
    - `relative_url` - input is considered an URL. tries to make the URL relative to that of the document you are editing.

      Note: You can use the `$ask` editor variable in file templates. In this case, keeps an absolute URL.

    - `combobox` - displays a dialog that contains a non-editable combo-box;
    - `editable_combobox` - displays a dialog that contains an editable combo-box;
    - `radio` - displays a dialog that contains radio buttons;
  - `'default-value'` - optional parameter. Provides a default value in the input text box.

  Examples:

  - `${ask('message')}` - Only the message displayed for the user is specified.
  - `${ask('message', generic, 'default')}` - `'message'` is displayed, the type is not specified (the default is string), the default value is `'default'`.
  - `${ask('message', password)}` - `'message'` is displayed, the characters typed are masked with a circle symbol.
  - `${ask('message', password, 'default')}` - same as before, the default value is `'default'`.
  - `${ask('message', url)}` - `'message'` is displayed, the parameter type is URL.
  - `${ask('message', url, 'default')}` - same as before, the default value is `'default'`.

- **${system(var.name)}** - Value of the *var.name* Java system property. The Java system properties can be specified in the command line arguments of the Java runtime as `-Dvar.name=var.value`. If you are looking for operating system environment variables, use the**${env(VAR_NAME)}** editor variable instead.
- **${date(pattern)}** - Current date. Follows the given pattern. Example: yyyy-MM-dd.
- **${dbgXML}** - The path to the current Debugger source selection (for tools started from the XSLT/XQuery Debugger).
- **${dbgXSL}** - The path to the current Debugger stylesheet selection (for tools started from the XSLT/XQuery Debugger).
- **${tsf}** - The transformation result file.
- **${ps}** - The path separator which can be used on different operating systems between libraries specified in the class path.
- **${dsu}** - The path of the detected schema as a URL.
- **${ds}** - The path of the detected schema as a local file path.
- **${cp}** - Current page number.
- **${tp}** - Total number of pages in the document.

Custom Editor Variables

An editor variable can be created by the user and included in any user defined expression where a built-in editor variable is also allowed. For example a custom editor variable may be necessary for configuring the command line of an external tool, the working directory of a custom validator, the command line of a custom XSLT engine, a custom FO processor, etc. All the custom editor variables are listed together with the built-in editor variables, for example when editing the working folder or the command line of an external tool or of a custom validator , the working directory, etc.

Creating a custom editor variable is very simple: just specify the name that will be used in user defined expressions, the value that will replace the variable name at runtime and a textual description for the user of that variable.

You can configure the custom editor variables in the **Preferences** page.

Configuring XML Catalogs

In the XML sample file for **SDF** you did not use a xsi:schemaLocation attribute, but instead you let the editor use the schema from the association. However there are cases in which you must refer for instance the location of a schema file from a remote web location and an Internet connection may not be available. In such cases an XML catalog may be used to map the web location to a local file system entry. The following procedure presents an example of using an XML catalogs, by modifying our `sdf.xsd` XML Schema file from the *Example Files Listings*.

1. Create a catalog file that will help the parser locate the schema for validating the XML document. The file must map the location of the schema to a local version of the schema.

   Create a new XML file called `catalog.xml` and save it into the `{oXygen_installation_directory} / frameworks / sdf` directory. The content of the file should be:

```
<?xml version="1.0"?>
<catalog xmlns="urn:oasis:names:tc:entity:xmlns:xml:catalog">
    <uri name="http://www.oxygenxml.com/SDF/abs.xsd"
            uri="schema/abs.xsd"/>
    <uri name="http://www.oxygenxml.com/SDF/abs.xsd"
                uri="schema/abs.xsd"/>
</catalog>
```

2. Add catalog files to your Document Type Association using the Catalogs tab from the Document Type dialog.

To test the catalog settings, restart and try to validate a new sample **Simple Documentation Framework** document. There should be no errors.

The `sdf.xsd` schema that validates the document refers the other file `abs.xsd` through an import element:

```
<xs:import namespace=
 "http://www.oxygenxml.com/sample/documentation/abstracts"
 schemaLocation="http://www.oxygenxml.com/SDF/abs.xsd"/>
```

The `schemaLocation` attribute references the `abs.xsd` file:

```
xsi:schemaLocation="http://www.oxygenxml.com/sample/documentation/abstracts
    http://www.oxygenxml.com/SDF/abs.xsd"/>
```

The catalog mapping is:

```
http://www.oxygenxml.com/SDF/abs.xsd -> schema/abs.xsd
```

This means that all the references to http://www.oxygenxml.com/SDF/abs.xsd must be resolved to the `abs.xsd` file located in the `schema` directory. The URI element is used by URI resolvers, for example for resolving a URI reference used in an XSLT stylesheet.

Configuring Transformation Scenarios

When distributing a framework to the users, it is a good idea to have the transformation scenarios already configured. This would help the content authors publish their work in different formats. Being contained in the **Document Type Association** the scenarios can be distributed along with the actions, menus, toolbars, catalogs, etc.

These are the steps that allow you to create a transformation scenario for your framework.

1. Create a `xsl` folder inside the `frameworks / sdf` folder.

   The folder structure for the documentation framework should be:

   ```
   oxygen
     frameworks
       sdf
         schema
         css
         templates
         xsl
   ```

2. Create the `sdf.xsl` file in the `xsl` folder. The complete content of the `sdf.xsl` file is found in the *Example Files Listings*.
3. Open the **Options**/**Preferences**/**Document Type Associations**. Open the **Document Type** dialog for the **SDF** framework then choose the **Transformation** tab. Click the **New** button.

   In the **Edit Scenario** dialog, fill the following fields:

   - Fill in the **Name** field with *SDF to HTML*. This will be the name of your transformation scenario.
   - Set the **XSL URL** field to `${frameworks}/sdf/xsl/sdf.xsl`.
   - Set the **Transformer** to *Saxon 9B*.

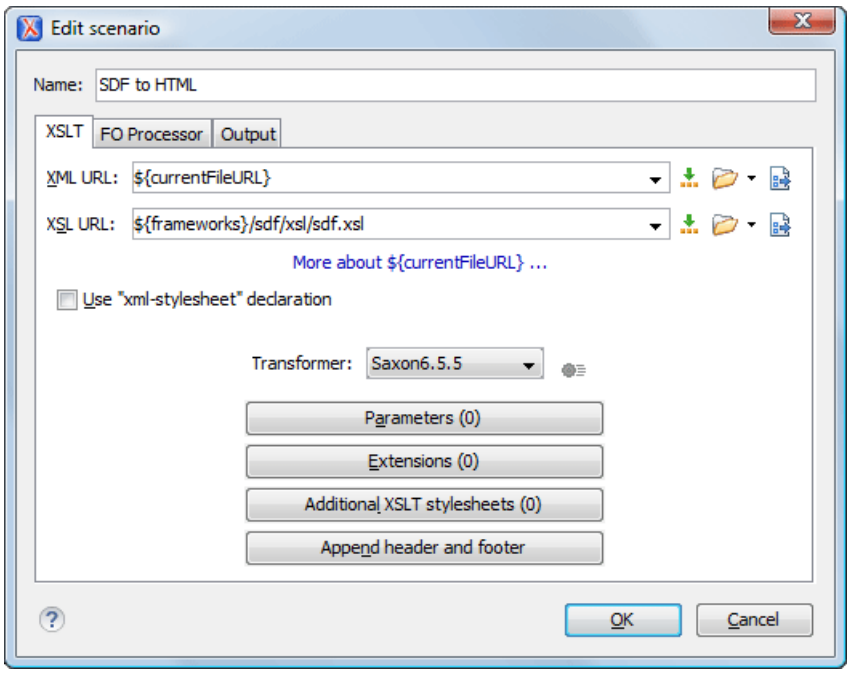


Figure 17: Configuring a transformation scenario

4. Change to the **Output** tab. Configure the fields as follows:
   - Set the **Save as** field to `${cfd}/${cfn}.html`. This means the transformation output file will have the name of the XML file and the *html* extension and will be stored in the same folder.
   - Enable the **Open in Browser/System Application** option.

     Note: If you already set the **Default Internet browser** option in the **Global** preferences page, it takes precedence over the default system application settings.

   - Enable the **Saved file** option.

Now the scenario is listed in the **Transformation** tab:

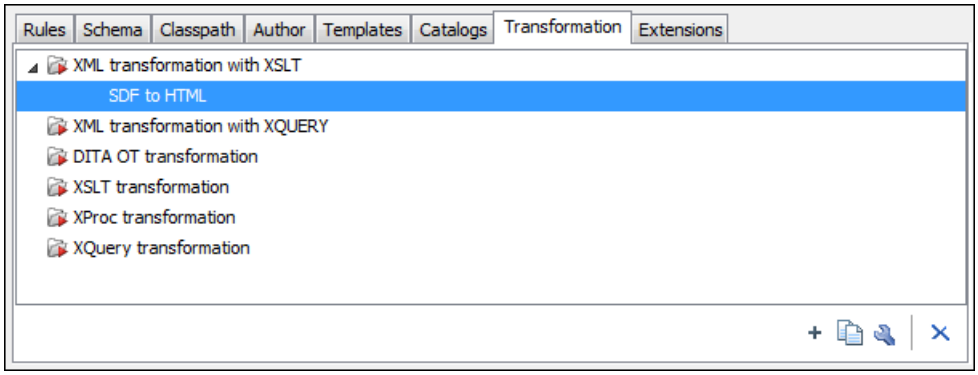


Figure 18: The transformation tab

To test the transformation scenario you just created, open the **SDF** XML sample from the *Example Files Listings*. Click the **Apply Transformation Scenario(s)** button to display the **Configure Transformation Scenario(s)** dialog. Its scenario list contains the scenario you defined earlier *SDF to HTML*. Click it then choose **Transform now**. The HTML file should be saved in the same folder as the XML file and displayed in the browser.

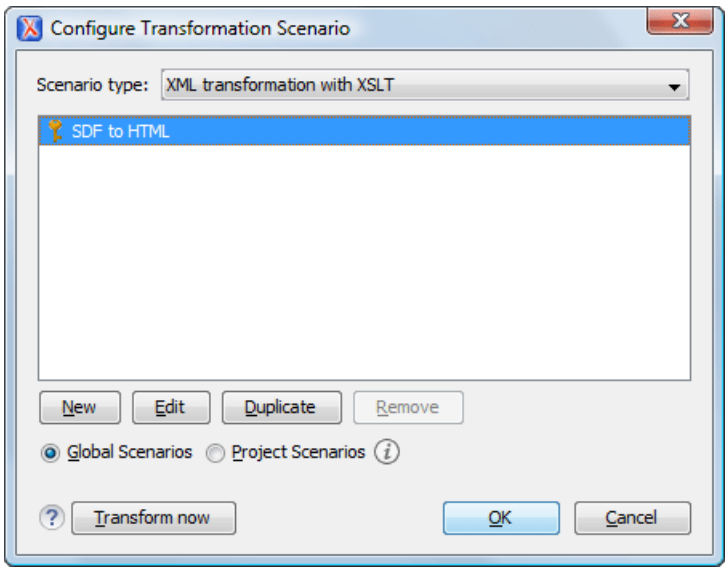


Figure 19: Selecting the predefined scenario

Configuring Validation Scenarios

You can distribute a framework with a series of already configured validation scenarios. Also, this provides enhanced validation support allowing you to use multiple grammars to check the document. For example, you can use Schematron rules to impose guidelines, otherwise impossible to enforce using conventional validation.

To associate a validation scenario with a specific framework, follow these steps:

1. Open the **Options**/**Preferences**/**Document Type Associations**. Open the **Document Type** dialog for the **SDF** framework, then choose the **Validation** tab. This tab holds a list of document types for which you can define validation scenarios. To set one of the validation scenarios as default for a specific document type, select it and press / **Toggle default**.
2. Press the **New** button to add a new scenario.
3. Press the **Add** button to add a new validation unit with default settings.
   The dialog that lists all validation units of the scenario is opened.

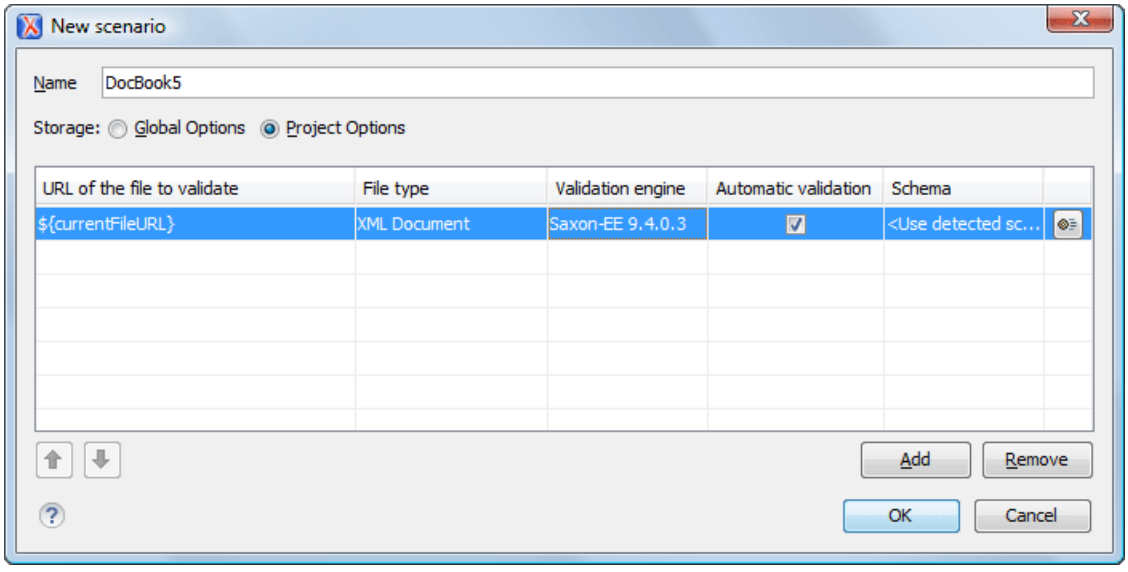


Figure 20: Add / Edit a Validation Unit

The table holds the following information:

- **Storage** - allows you to create a scenario at project level, or as global;
- **URL of the file to validate** - the URL of the main module which includes the current module. It is also the entry module of the validation process when the current one is validated;
- **File type** - the type of the document validated in the current validation unit. automatically selects the file type depending on the value of the **URL of the file to validate** field;
- **Validation engine** - one of the engines available in for validation of the type of document to which the current module belongs. **Default engine** is the default setting and means that the default engine executes the validation. This engine is set in **Preferences** pages for the type of the current document (XML document, XML Schema, XSLT stylesheet, XQuery file, and others) instead of a validation scenario;
- **Automatic validation** - if this option is checked, then the validation operation defined by this row of the table is applied also by the automatic validation feature. If the **Automatic validation** feature is disabled in Preferences then this option does not take effect as the Preference setting has higher priority;
- **Schema** - active when you set the **File type** to **XML Document**;
- **Settings** - contains an action that allows you to set a schema, when validating XML documents, or a list of extensions when validating XSL or XQuery documents.

4. Edit the URL of the main validation module.

   Specify the URL of the main module:

   - browsing for a local, remote, or archived file;
   - using an *editor variable* or a *custom editor variable*, available in the following pop-up menu, opened after pressing the button:

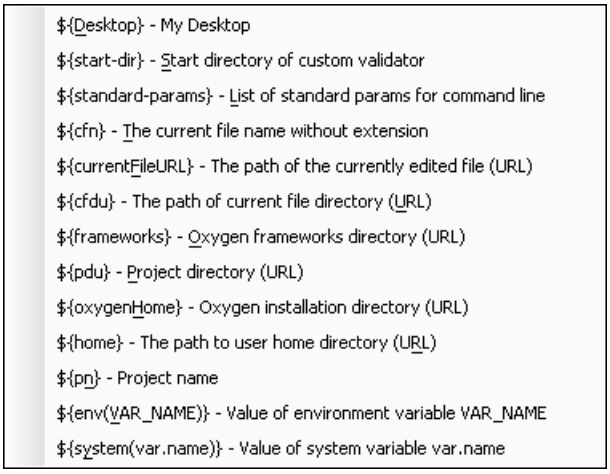


Figure 21: Insert an Editor Variable

5. Select the type of the validated document.

   Note that it determines the list of possible validation engines.

6. Select the validation engine.
7. Select the **Automatic validation** option if you want to validate the current unit when automatic validation feature is turned on in Preferences.
8. Choose what schema is used during validation: the one detected after parsing the document or a custom one.

Configuring Extensions

You can add extensions to your Document Type Association using the **Extensions** tab from the Document Type dialog.

Configuring an Extensions Bundle

Starting with 10.3 version a single bundle was introduced acting as a provider for all other extensions. The individual extensions can still be set and if present they take precedence over the single provider, but this practice is being discouraged

and the single provider should be used instead. To set individual extensions go to **Options** > **Preferences** > **Document Type Association**, double-click a document type and go to the extension tab.

The extensions bundle is represented by the *ro.sync.ecss.extensions.api.ExtensionsBundle* class. The provided implementation of the `ExtensionsBundle` is instantiated when the rules of the Document Type Association defined for the custom framework match a document opened in the editor. Therefor references to objects which need to be persistent throughout the application running session must not be kept in the bundle because the next detection event can result in creating another `ExtensionsBundle` instance.

Note: The Javadoc documentation of the Author API used in the example files is *available on the website*. Also it can be downloaded as a *zip archive from the website*.

1. Create a new Java project, in your IDE. Create the `lib` folder in the Java project folder and copy in it the `oxygen.jar` file from the `{oXygen_installation_directory}/lib` folder.
2. Create the class `simple.documentation.framework.SDFExtensionsBundle` which must extend the abstract class `ro.sync.ecss.extensions.api.ExtensionsBundle`.

```
public class SDFExtensionsBundle extends ExtensionsBundle {
```

3. A **Document Type ID** and a short description should be defined first by implementing the methods `getDocumentTypeID` and `getDescription`. The Document Type ID is used to uniquely identify the current framework. Such an ID must be provided especially if options related to the framework need to be persistently stored and retrieved between sessions.

```
public String getDocumentTypeID() {
     return "Simple.Document.Framework.document.type";
}

public String getDescription() {
     return "A custom extensions bundle used for the Simple Document" +
                    "Framework document type";
}
```

4. In order to be notified about the activation of the custom Author extension in relation with an opened document an *ro.sync.ecss.extensions.api.AuthorExtensionStateListener* should be implemented. The **activation** and **deactivation** events received by this listener should be used to perform custom initializations and to register / remove listeners like *ro.sync.ecss.extensions.api.AuthorListener*, *ro.sync.ecss.extensions.api.AuthorMouseListener* or *ro.sync.ecss.extensions.api.AuthorCaretListener*. The custom author extension state listener should be provided by implementing the method `createAuthorExtensionStateListener`.

```
public AuthorExtensionStateListener createAuthorExtensionStateListener() {
     return new SDFAuthorExtensionStateListener();
}
```

   The `AuthorExtensionStateListener` is instantiated and notified about the activation of the framework when the rules of the Document Type Association match a document opened in the Author editor mode. The listener is notified about the deactivation when another framework is activated for the same document, the user switches to another mode or the editor is closed. A complete description and implementation of an `ro.sync.ecss.extensions.api.AuthorExtensionStateListener` can be found in the *Implementing an Author Extension State Listener*.

   If Schema Aware mode is active in Oxygen, all actions that can generate invalid content will be redirected toward the *ro.sync.ecss.extensions.api.AuthorSchemaAwareEditingHandler*. The handler can either resolve a specific case, let the default implementation take place or reject the edit entirely by throwing an *ro.sync.ecss.extensions.api.InvalidEditException*. The actions that are forwarded to this handler include typing, delete or paste.

   See the *Implementing an Author Schema Aware Editing Handler* section for more details about this handler.

5. Customizations of the content completion proposals are permitted by creating a schema manager filter extension. The interface that declares the methods used for content completion proposals filtering is

ro.sync.contentcompletion.xml.SchemaManagerFilter. The filter can be applied on elements, attributes or on their values. Responsible for creating the content completion filter is the method `createSchemaManagerFilter`. A new `SchemaManagerFilter` will be created each time a document matches the rules defined by the Document Type Association which contains the filter declaration.

```
public SchemaManagerFilter createSchemaManagerFilter() {
     return new SDFSchemaManagerFilter();
}
```

A detailed presentation of the schema manager filter can be found in *Configuring a Content completion handler* section.

6. The Author supports link based navigation between documents and document sections. Therefore, if the document contains elements defined as links to other elements, for example links based on the **id** attributes, the extension should provide the means to find the referred content. To do this an implementation of the *ro.sync.ecss.extensions.api.link.ElementLocatorProvider* interface should be returned by the `createElementLocatorProvider` method. Each time an element pointed by a link needs to be located the method is invoked.

```
public ElementLocatorProvider createElementLocatorProvider() {
     return new DefaultElementLocatorProvider();
}
```

The section that explains how to implement an element locator provider is *Configuring a Link target element finder*.

7. The drag and drop functionality can be extended by implementing the *ro.sync.exml.editor.xmleditor.pageauthor.AuthorDnDListener* interface. Relevant methods from the listener are invoked when the mouse is dragged, moved over, or exits the Author editor mode, when the drop action changes, and when the drop occurs. Each method receives the `DropTargetEvent` containing information about the drag and drop operation. The drag and drop extensions are available on Author mode for both Eclipse plugin and standalone application. The Text mode corresponding listener is available only for Eclipse plugin. The methods corresponding to each implementation are: `createAuthorAWTDndListener`, `createTextSWTDndListener` and `createAuthorSWTDndListener`.

```
public AuthorDnDListener createAuthorAWTDndListener() {
     return new SDFAuthorDndListener();
}
```

For more details about the Author drag and drop listeners see the *Configuring a custom Drag and Drop listener* section.

8. Another extension which can be included in the bundle is the reference resolver. In our case the references are represented by the **ref** element and the attribute indicating the referred resource is **location**. To be able to obtain the content of the referred resources you will have to implement a Java extension class which implements the *ro.sync.ecss.extensions.api.AuthorReferenceResolver*. The method responsible for creating the custom references resolver is `createAuthorReferenceResolver`. The method is called each time a document opened in an Author editor mode matches the Document Type Association where the extensions bundle is defined. The instantiated references resolver object is kept and used until another extensions bundle corresponding to another Document Type is activated as result of the detection process.

```
public AuthorReferenceResolver createAuthorReferenceResolver() {
     return new ReferencesResolver();
}
```

A more detailed description of the references resolver can be found in the *Configuring a References Resolver* section.

9. To be able to dynamically customize the default CSS styles for a certain *ro.sync.ecss.extensions.api.node.AuthorNode* an implementation of the *ro.sync.ecss.extensions.api.StylesFilter* can be provided. The extensions bundle method responsible for creating the `StylesFilter` is `createAuthorStylesFilter`. The method is called each time a document opened in an Author editor mode matches the document type association where the extensions bundle is defined.

The instantiated filter object is kept and used until another extensions bundle corresponding to another Document Type is activated as a result of the detection process.

```
public StylesFilter createAuthorStylesFilter() {
     return new SDFStylesFilter();
}
```

See the *Configuring CSS styles filter* section for more details about the styles filter extension.

10. In order to edit data in custom tabular format implementations of the *`ro.sync.ecss.extensions.api.AuthorTableCellSpanProvider`* and *`the ro.sync.ecss.extensions.api.AuthorTableColumnWidthProvider`* interfaces should be provided. The two methods from the `ExtensionsBundle` specifying these two extension points are `createAuthorTableCellSpanProvider` and `createAuthorTableColumnWidthProvider`.

```
public AuthorTableCellSpanProvider createAuthorTableCellSpanProvider() {
     return new TableCellSpanProvider();
}

public AuthorTableColumnWidthProvider
         createAuthorTableColumnWidthProvider() {
     return new TableColumnWidthProvider();
}
```

The two table information providers are not reused for different tables. The methods are called for each table in the document so new instances should be provided every time. Read more about the cell span and column width information providers in *Configuring a Table Cell Span Provider* and *Configuring a Table Column Width Provider* sections.

If the functionality related to one of the previous extension point does not need to be modified then the developed *`ro.sync.ecss.extensions.api.ExtensionsBundle`* should not override the corresponding method and leave the default base implementation to return **null**.

11. An XML vocabulary can contain links to different areas of a document. In case the document contains elements defined as link you can choose to present a more relevant text description for each link. To do this an implementation of the *`ro.sync.ecss.extensions.api.link.LinkTextResolver`* interface should be returned by the `createLinkTextResolver` method. This implementation is used each time *the oxy_link-text() function* is encountered in the CSS styles associated with an element.

```
public LinkTextResolver createLinkTextResolver() {
  return new DitaLinkTextResolver();
}
```

offers built in implementations for DITA and DocBook:
`ro.sync.ecss.extensions.dita.link.DitaLinkTextResolver`
`ro.sync.ecss.extensions.docbook.link.DocbookLinkTextResolver`

12. Pack the compiled class into a jar file.
13. Copy the jar file into the `frameworks / sdf` directory.
14. Add the jar file to the Author class path.
15. Register the Java class by clicking on the **Extensions** tab. Press the **Choose** button and select from the displayed dialog the name of the class: `SDFExtensionsBundle`.

Note: The complete source code can be found in the Simple Documentation Framework project, included in the *Oxygen Author SDK zip* available for download *on the website*.

Customize Profiling Conditions

For each document type, you can configure the phrase-type elements that wrap the profiled content by setting a custom *`ro.sync.ecss.extensions.api.ProfilingConditionalTextProvider`*. This configuration is set by default for DITA and Docbook frameworks.

Preserve Style and Format on Copy and Paste from External Applications

Styled content can be inserted in the Author editor by copying or dragging it from:

- Office-type applications (**Microsoft Word** and **Microsoft Excel**, **OpenOffice.org Writer** and **OpenOffice.org Calc**);
- web browsers (like **Mozilla Firefox** or **Microsoft Internet Explorer**);
- the **Data Source Explorer** view (where resources are available from WebDAV or CMS servers).

The styles and general layout of the copied content like: sections with headings, tables, list items, bold, and italic text, hyperlinks, are preserved by the paste operation by transforming them to the equivalent XML markup of the target document type. This is available by default in the following *predefined document types*: *DITA*, *DocBook 4*, *DocBook 5*, *TEI 4*, *TEI 5*, *XHTML*.

For other document types the default behavior of the paste operation is to keep only the text content without the styling but it can be customized by setting an XSLT stylesheet in that document type. The XSLT stylesheet must accept as input an XHTML flavor of the copied content and transform it to the equivalent XML markup that is appropriate for the target document type of the paste operation. The stylesheet is *set up* by implementing the `getImporterStylesheetFileName` method of an instance object of *the* `AuthorExternalObjectInsertionHandler` *class* which is returned by the `createExternalObjectInsertionHandler` method of *the* `ExtensionsBundle` *instance* of the target document type.

Implementing an Author Extension State Listener

The `ro.sync.ecss.extensions.api.AuthorExtensionStateListener` implementation is notified when the Author extension where the listener is defined is activated or deactivated in the Document Type detection process.

Note: The Javadoc documentation of the Author API used in the example files is *available on the website*. Also it can be downloaded as a *zip archive from the website*.

```
import ro.sync.ecss.extensions.api.AuthorAccess;
import ro.sync.ecss.extensions.api.AuthorExtensionStateListener;

public class SDFAuthorExtensionStateListener implements
  AuthorExtensionStateListener {
  private AuthorListener sdfAuthorDocumentListener;
  private AuthorMouseListener sdfMouseListener;
  private AuthorCaretListener sdfCaretListener;
  private OptionListener sdfOptionListener;
```

The **activation** event received by this listener when the rules of the Document Type Association match a document opened in the Author editor mode, should be used to perform custom initializations and to register listeners like `ro.sync.ecss.extensions.api.AuthorListener`, `ro.sync.ecss.extensions.api.AuthorMouseListener` or `ro.sync.ecss.extensions.api.AuthorCaretListener`.

```
 public void activated(AuthorAccess authorAccess) {
   // Get the value of the option.
   String option = authorAccess.getOptionsStorage().getOption(
               "sdf.custom.option.key", "");
   // Use the option for some initializations...

   // Add an option listener.
   authorAccess.getOptionsStorage().addOptionListener(sdfOptionListener);

   // Add author document listeners.
   sdfAuthorDocumentListener = new SDFAuthorListener();
   authorAccess.getDocumentController().addAuthorListener(
               sdfAuthorDocumentListener);

   // Add mouse listener.
   sdfMouseListener = new SDFAuthorMouseListener();
   authorAccess.getEditorAccess().addAuthorMouseListener(sdfMouseListener);

   // Add caret listener.
   sdfCaretListener = new SDFAuthorCaretListener();
   authorAccess.getEditorAccess().addAuthorCaretListener(sdfCaretListener);
```

```
    // Other custom initializations...
  }
```

The authorAccess parameter received by the `activated` method can be used to gain access to Author specific actions and informations related to components like the editor, document, workspace, tables, or the change tracking manager.

If options specific to the custom developed Author extension need to be stored or retrieved, a reference to the *`ro.sync.ecss.extensions.api.OptionsStorage`* can be obtained by calling the `getOptionsStorage` method from the author access. The same object can be used to register *`ro.sync.ecss.extensions.api.OptionListener`* listeners. An option listener is registered in relation with an option **key** and will be notified about the value changes of that option.

An `AuthorListener` can be used if events related to the Author document modifications are of interest. The listener can be added to the *`ro.sync.ecss.extensions.api.AuthorDocumentController`*. A reference to the document controller is returned by the `getDocumentController` method from the author access. The document controller can also be used to perform operations involving document modifications.

To provide access to Author editor component related functionality and information, the author access has a reference to the *`ro.sync.ecss.extensions.api.access.AuthorEditorAccess`* that can be obtained when calling the `getEditorAccess` method. At this level `AuthorMouseListener` and `AuthorCaretListener` can be added which will be notified about mouse and caret events occurring in the Author editor mode.

The **deactivation** event is received when another framework is activated for the same document, the user switches to another editor mode or the editor is closed. The `deactivate` method is typically used to unregister the listeners previously added on the `activate` method and to perform other actions. For example, options related to the deactivated author extension can be saved at this point.

```
  public void deactivated(AuthorAccess authorAccess) {
    // Store the option.
    authorAccess.getOptionsStorage().setOption(
                "sdf.custom.option.key", optionValue);

    // Remove the option listener.
    authorAccess.getOptionsStorage().removeOptionListener(sdfOptionListener);

    // Remove document listeners.
    authorAccess.getDocumentController().removeAuthorListener(
                sdfAuthorDocumentListener);

    // Remove mouse listener.
    authorAccess.getEditorAccess().removeAuthorMouseListener(sdfMouseListener);

    // Remove caret listener.
    authorAccess.getEditorAccess().removeAuthorCaretListener(sdfCaretListener);

    // Other actions...

  }
```

Implementing an Author Schema Aware Editing Handler

To implement your own handler for actions like typing, deleting, or pasting, provide an implementation of *`ro.sync.ecss.extensions.api.AuthorSchemaAwareEditingHandler`*. For this handler to be called, the **Schema Aware Editing** option must be set to **On**,or **Custom**. The handler can either resolve a specific case, let the default implementation take place, or reject the edit entirely by throwing an `InvalidEditException`.

Note: The Javadoc documentation of the Author API used in the example files is *available on the website*. Also it can be downloaded as a *zip archive from the website*.

```
package simple.documentation.framework.extensions;

/**
 * Specific editing support for SDF documents.
 * Handles typing and paste events inside section and tables.
 */
public class SDFSchemaAwareEditingHandler implements AuthorSchemaAwareEditingHandler {
```

Typing events can be handled using the `handleTyping` method. For example, the `SDFSchemaAwareEditingHandler` checks if the schema is not a learned one, was loaded successfully and **Smart Paste** is active. If these conditions are met, the event will be handled.

```
/**
 * @see ro.sync.ecss.extensions.api.AuthorSchemaAwareEditingHandler#handleTyping(int, char,
ro.sync.ecss.extensions.api.AuthorAccess)
 */
public boolean handleTyping(int offset, char ch, AuthorAccess authorAccess)
throws InvalidEditException {
  boolean handleTyping = false;
  AuthorSchemaManager authorSchemaManager = authorAccess.getDocumentController().getAuthorSchemaManager();
  if (!authorSchemaManager.isLearnSchema() &&
      !authorSchemaManager.hasLoadingErrors() &&
      authorSchemaManager.getAuthorSchemaAwareOptions().isEnableSmartTyping()) {
    try {
      AuthorDocumentFragment characterFragment =
        authorAccess.getDocumentController().createNewDocumentTextFragment(String.valueOf(ch));
     handleTyping = handleInsertionEvent(offset, new AuthorDocumentFragment[] {characterFragment}, authorAccess);

    } catch (AuthorOperationException e) {
      throw new InvalidEditException(e.getMessage(), "Invalid typing event: " + e.getMessage(), e, false);
    }
  }
  return handleTyping;
}
```

Implementing the `AuthorSchemaAwareEditingHandler` gives the possibility to handle other events like: the keyboard delete event at the given offset (using Delete or Backspace keys), delete element tags, delete selection, join elements or paste fragment.

Note: The complete source code can be found in the Simple Documentation Framework project, included in the *Oxygen Author SDK zip* available for download *on the website*.

Configuring a Content Completion Handler

You can filter or contribute to items offered for content completion by implementing the *`ro.sync.contentcompletion.xml.SchemaManagerFilter`* interface.

Note: The Javadoc documentation of the Author API used in the example files is *available on the website*. Also it can be downloaded as a *zip archive from the website*.

```
import java.util.List;

import ro.sync.contentcompletion.xml.CIAttribute;
import ro.sync.contentcompletion.xml.CIElement;
import ro.sync.contentcompletion.xml.CIValue;
import ro.sync.contentcompletion.xml.Context;
import ro.sync.contentcompletion.xml.SchemaManagerFilter;
import ro.sync.contentcompletion.xml.WhatAttributesCanGoHereContext;
import ro.sync.contentcompletion.xml.WhatElementsCanGoHereContext;
import ro.sync.contentcompletion.xml.WhatPossibleValuesHasAttributeContext;

public class SDFSchemaManagerFilter implements SchemaManagerFilter {
```

You can implement the various callbacks of the interface either by returning the default values given by or by contributing to the list of proposals. The filter can be applied on elements, attributes or on their values. Attributes filtering can be implemented using the `filterAttributes` method and changing the default content completion list of *`ro.sync.contentcompletion.xml.CIAttribute`* for the element provided by the current *`ro.sync.contentcompletion.xml.WhatAttributesCanGoHereContext`* context. For example, the `SDFSchemaManagerFilter` checks if the element from the current context is the `table` element and adds the `frame` attribute to the `table` list of attributes.

```
/**
 * Filter attributes of the "table" element.
 */
public List<CIAttribute> filterAttributes(List<CIAttribute> attributes,
    WhatAttributesCanGoHereContext context) {
  // If the element from the current context is the 'table' element add the
  // attribute named 'frame' to the list of default content completion proposals
  if (context != null) {
    ContextElement contextElement = context.getParentElement();
    if ("table".equals(contextElement.getQName())) {
```

```
      CIAttribute frameAttribute = new CIAttribute();
      frameAttribute.setName("frame");
      frameAttribute.setRequired(false);
      frameAttribute.setFixed(false);
      frameAttribute.setDefaultValue("void");
      if (attributes == null) {
        attributes = new ArrayList<CIAttribute>();
      }
      attributes.add(frameAttribute);
    }
  }
  return attributes;
}
```

The elements that can be inserted in a specific context can be filtered using the `filterElements` method. The `SDFSchemaManagerFilter` uses this method to replace the `td` child element with the `th` element when `header` is the current context element.

```
public List<CIElement> filterElements(List<CIElement> elements,
    WhatElementsCanGoHereContext context) {
  // If the element from the current context is the 'header' element remove the
  // 'td' element from the list of content completion proposals and add the
  // 'th' element.
  if (context != null) {
    Stack<ContextElement> elementStack = context.getElementStack();
    if (elementStack != null) {
      ContextElement contextElement = context.getElementStack().peek();
      if ("header".equals(contextElement.getQName())) {
        if (elements != null) {
          for (Iterator<CIElement> iterator = elements.iterator(); iterator.hasNext();) {
            CIElement element = iterator.next();
            // Remove the 'td' element
            if ("td".equals(element.getQName())) {
              elements.remove(element);
              break;
            }
          }
        } else {
          elements = new ArrayList<CIElement>();
        }
        // Insert the 'th' element in the list of content completion proposals
        CIElement thElement = new SDFElement();
        thElement.setName("th");
        elements.add(thElement);
      }
    }
  } else {
    // If the given context is null then the given list of content completion elements contains
    // global elements.
  }
  return elements;
}
```

The elements or attributes values can be filtered using the `filterElementValues` or `filterAttributeValues` methods.

Note: The complete source code can be found in the Simple Documentation Framework project, included in the *Oxygen Author SDK zip* available for download *on the website*.

Configuring a Link target element finder

The link target reference finder represents the support for finding references from links which indicate specific elements inside an XML document. This support will only be available if a schema is associated with the document type.

If you do not define a custom link target reference finder, the `DefaultElementLocatorProvider` implementation will be used by default. The interface which should be implemented for a custom link target reference finder is *`ro.sync.ecss.extensions.api.link.ElementLocatorProvider`*. As an alternative, the *`ro.sync.ecss.extensions.commons.DefaultElementLocatorProvider`* implementation can also be extended.

The used `ElementLocatorProvider` will be queried for an *`ElementLocator`* when a link location must be determined (when a link is clicked). Then, to find the corresponding (linked) element, the obtained `ElementLocator` will be queried for each element from the document.

Note: The Javadoc documentation of the Author API used in the example files is *available on the website*. Also it can be downloaded as a *zip archive from the website*.

The `DefaultElementLocatorProvider` implementation

The `DefaultElementLocatorProvider` implementation offers support for the most common types of links:

- links based on ID attribute values
- XPointer element() scheme

The method `getElementLocator` determines what `ElementLocator` should be used. In the default implementation it checks if the link is an XPointer element() scheme otherwise it assumes it is an ID. A non-null `IDTypeVerifier` will always be provided if a schema is associated with the document type.

The `link` string argument is the "anchor" part of the of the URL which is composed from the value of the link property specified for the link element in the CSS.

```
public ElementLocator getElementLocator(IDTypeVerifier idVerifier,
                String link) {
  ElementLocator elementLocator = null;
  try {
    if(link.startsWith("element(")){
      // xpointer element() scheme
      elementLocator = new XPointerElementLocator(idVerifier, link);
    } else {
      // Locate link element by ID
      elementLocator = new IDElementLocator(idVerifier, link);
    }
  } catch (ElementLocatorException e) {
    logger.warn("Exception when create element locator for link: "
        + link + ". Cause: " + e, e);
  }
  return elementLocator;
}
```

The `XPointerElementLocator` implementation

`XPointerElementLocator` is an implementation of the abstract class *`ro.sync.ecss.extensions.api.link.ElementLocator`* for links that have one of the following XPointer element() scheme patterns:

element(*elementID*)	Locate the element with the specified id.
element(*/1/2/3*)	A child sequence appearing alone identifies an element by means of stepwise navigation, which is directed by a sequence of integers separated by slashes (/); each integer n locates the nth child element of the previously located element.
element(*elementID/3/4*)	A child sequence appearing after a *NCName* identifies an element by means of stepwise navigation, starting from the element located by the given name.

The constructor separates the id/integers which are delimited by slashes(/) into a sequence of identifiers (an XPointer path). It will also check that the link has one of the supported patterns of the XPointer element() scheme.

```
public XPointerElementLocator(IDTypeVerifier idVerifier, String link)
                    throws ElementLocatorException {
  super(link);
  this.idVerifier = idVerifier;

  link = link.substring("element(".length(), link.length() - 1);

  StringTokenizer stringTokenizer = new StringTokenizer(link, "/", false);
  xpointerPath = new String[stringTokenizer.countTokens()];
  int i = 0;
  while (stringTokenizer.hasMoreTokens()) {
    xpointerPath[i] = stringTokenizer.nextToken();
    boolean invalidFormat = false;

    // Empty xpointer component is not supported
    if(xpointerPath[i].length() == 0){
      invalidFormat = true;
    }

    if(i > 0){
      try {
        Integer.parseInt(xpointerPath[i]);
      } catch (NumberFormatException e) {
        invalidFormat = true;
```

```
      }
    }

    if(invalidFormat){
      throw new ElementLocatorException(
        "Only the element() scheme is supported when locating XPointer links."
        + "Supported formats: element(elementID), element(/1/2/3),
              element(elemID/2/3/4).");
    }
    i++;
  }

  if(Character.isDigit(xpointerPath[0].charAt(0))){
    // This is the case when xpointer have the following pattern /1/5/7
    xpointerPathDepth = xpointerPath.length;
  } else {
    // This is the case when xpointer starts with an element ID
    xpointerPathDepth = -1;
    startWithElementID  = true;
  }
}
```

The method `startElement` will be invoked at the beginning of every element in the XML document(even when the element is empty). The arguments it takes are

uri	The namespace URI, or the empty string if the element has no namespace URI or if namespace processing is disabled.
localName	Local name of the element.
qName	Qualified name of the element.
atts	Attributes attached to the element. If there are no attributes, this argument will be empty.

The method returns `true` if the processed element is found to be the one indicated by the link.

The `XPointerElementLocator` implementation of the `startElement` will update the depth of the current element and keep the index of the element in its parent. If the `xpointerPath` starts with an element ID then the current element ID is verified to match the specified ID. If this is the case the depth of the XPointer is updated taking into account the depth of the current element.

If the XPointer path depth is the same as the current element depth then the kept indices of the current element path are compared to the indices in the XPointer path. If all of them match then the element has been found.

```
public boolean startElement(String uri, String localName,
        String name, Attr[] atts) {
  boolean linkLocated = false;
  // Increase current element document depth
  startElementDepth ++;

  if (endElementDepth != startElementDepth) {
    // The current element is the first child of the parent
    currentElementIndexStack.push(new Integer(1));
  } else {
    // Another element in the parent element
    currentElementIndexStack.push(new Integer(lastIndexInParent + 1));
  }

  if (startWithElementID) {
    // This the case when xpointer path starts with an element ID.
    String xpointerElement = xpointerPath[0];
    for (int i = 0; i < atts.length; i++) {
      if(xpointerElement.equals(atts[i].getValue())){
        if(idVerifier.hasIDType(
            localName, uri, atts[i].getQName(), atts[i].getNamespace())){
          xpointerPathDepth = startElementDepth + xpointerPath.length - 1;
          break;
        }
      }
    }
  }

  if (xpointerPathDepth == startElementDepth){
    // check if xpointer path matches with the current element path
    linkLocated = true;
    try {
      int xpointerIdx = xpointerPath.length - 1;
      int stackIdx = currentElementIndexStack.size() - 1;
      int stopIdx = startWithElementID ? 1 : 0;
```

```
      while (xpointerIdx >= stopIdx && stackIdx >= 0) {
        int xpointerIndex = Integer.parseInt(xpointerPath[xpointerIdx]);
        int currentElementIndex =
          ((Integer)currentElementIndexStack.get(stackIdx)).intValue();
        if(xpointerIndex != currentElementIndex) {
          linkLocated = false;
          break;
        }

        xpointerIdx--;
        stackIdx--;
      }

    } catch (NumberFormatException e) {
      logger.warn(e,e);
    }
  }
  return linkLocated;
}
```

The method `endElement` will be invoked at the end of every element in the XML document (even when the element is empty).

The `XPointerElementLocator` implementation of the `endElement` updates the depth of the current element path and the index of the element in its parent.

```
public void endElement(String uri, String localName, String name) {
  endElementDepth = startElementDepth;
  startElementDepth --;
  lastIndexInParent = ((Integer)currentElementIndexStack.pop()).intValue();
}
```

The `IDElementLocator` implementation

The `IDElementLocator` is an implementation of the abstract class *`ro.sync.ecss.extensions.api.link.ElementLocator`* for links that use an **id**.

The constructor only assigns field values and the method `endElement` is empty for this implementation.

The method `startElement` checks each of the element's attribute values and when one matches the link, it considers the element found if one of the following conditions is satisfied:

- the qualified name of the attribute is `xml:id`
- the attribute type is ID

The attribute type is checked with the help of the method `IDTypeVerifier.hasIDType`.

```
public boolean startElement(String uri, String localName,
        String name, Attr[] atts) {
  boolean elementFound = false;
  for (int i = 0; i < atts.length; i++) {
    if (link.equals(atts[i].getValue())) {
      if("xml:id".equals(atts[i].getQName())) {
        // xml:id attribute
        elementFound = true;
      } else {
        // check if attribute has ID type
        String attrLocalName =
          ExtensionUtil.getLocalName(atts[i].getQName());
        String attrUri = atts[i].getNamespace();
        if (idVerifier.hasIDType(localName, uri, attrLocalName, attrUri)) {
          elementFound = true;
        }
      }
    }
  }

  return elementFound;
}
```

Creating a customized link target reference finder

If you need to create a custom link target reference finder you can do so by creating the class which will implement the *`ro.sync.ecss.extensions.api.link.ElementLocatorProvider`* interface. As an alternative, your class could extend *`ro.sync.ecss.extensions.commons.DefaultElementLocatorProvider`*, the default implementation.

Note: The complete source code of the *ro.sync.ecss.extensions.commons.DefaultElementLocatorProvider*, *ro.sync.ecss.extensions.commons.IDElementLocator* or *ro.sync.ecss.extensions.commons.XPointerElementLocator* can be found in the Oxygen Default Frameworks project, included in the *Oxygen Author SDK zip* available for download *on the website*.

Configuring a custom Drag and Drop listener

Sometimes it is useful to perform various operations when certain objects are dropped from outside sources in the editing area. You can choose from three interfaces to implement depending on whether you are using the framework with the Eclipse plugin or the standalone version of the application or if you want to add the handler for the Text or Author modes.

Note: The Javadoc documentation of the Author API used in the example files is *available on the website*. Also it can be downloaded as a *zip archive from the website*.

Table 1: Interfaces for the DnD listener

Interface	Description
`ro.sync.exml.editor.xmleditor.pageauthor.AuthorDnDListener`	Receives callbacks from the standalone application for Drag And Drop in Author mode.
`com.oxygenxml.editor.editors.author.AuthorDnDListener`	Receives callbacks from the Eclipse plugin for Drag And Drop in Author mode.
`com.oxygenxml.editor.editors.TextDnDListener`	Receives callbacks from the Eclipse plugin for Drag And Drop in Text mode.

Configuring a References Resolver

You need to provide a handler for resolving references and obtain the content they refer. In our case the element which has references is **ref** and the attribute indicating the referred resource is **location**. You will have to implement a Java extension class for obtaining the referred resources.

Note: The Javadoc documentation of the Author API used in the example files is *available on the website*. Also it can be downloaded as a *zip archive from the website*.

1. Create the class `simple.documentation.framework.ReferencesResolver`. This class must implement the *ro.sync.ecss.extensions.api.AuthorReferenceResolver* interface.

```
import ro.sync.ecss.extensions.api.AuthorReferenceResolver;
import ro.sync.ecss.extensions.api.AuthorAccess;
import ro.sync.ecss.extensions.api.node.AttrValue;
import ro.sync.ecss.extensions.api.node.AuthorElement;
import ro.sync.ecss.extensions.api.node.AuthorNode;

public class ReferencesResolver
      implements AuthorReferenceResolver {
```

2. The `hasReferences` method verifies if the handler considers the node to have references. It takes as argument an `AuthorNode` that represents the node which will be verified. The method will return `true` if the node is considered to have references. In our case, to be a reference the node must be an element with the name *ref* and it must have an attribute named *location*.

```
public boolean hasReferences(AuthorNode node) {
  boolean hasReferences = false;
  if (node.getType() == AuthorNode.NODE_TYPE_ELEMENT) {
    AuthorElement element = (AuthorElement) node;
    if ("ref".equals(element.getLocalName())) {
      AttrValue attrValue = element.getAttribute("location");
      hasReferences = attrValue != null;
    }
  }
```

```
  return hasReferences;
}
```

3. The method `getDisplayName` returns the display name of the node that contains the expanded referred content. It takes as argument an `AuthorNode` that represents the node for which the display name is needed. The referred content engine will ask this `AuthorReferenceResolver` implementation what is the display name for each node which is considered a reference. In our case the display name is the value of the *location* attribute from the *ref* element.

```
public String getDisplayName(AuthorNode node) {
  String displayName = "ref-fragment";
  if (node.getType() == AuthorNode.NODE_TYPE_ELEMENT) {
    AuthorElement element = (AuthorElement) node;
    if ("ref".equals(element.getLocalName())) {
      AttrValue attrValue = element.getAttribute("location");
      if (attrValue != null) {
        displayName = attrValue.getValue();
      }
    }
  }
  return displayName;
}
```

4. The method `resolveReference` resolves the reference of the node and returns a `SAXSource` with the parser and the parser's input source. It takes as arguments an `AuthorNode` that represents the node for which the reference needs resolving, the *systemID* of the node, the `AuthorAccess` with access methods to the Author data model and a SAX `EntityResolver` which resolves resources that are already opened in another editor or resolve resources through the XML catalog. In the implementation you need to resolve the reference relative to the *systemID*, and create a parser and an input source over the resolved reference.

```
public SAXSource resolveReference(
    AuthorNode node,
    String systemID,
    AuthorAccess authorAccess,
    EntityResolver  entityResolver) {
  SAXSource saxSource = null;

  if (node.getType() == AuthorNode.NODE_TYPE_ELEMENT) {
    AuthorElement element = (AuthorElement) node;
    if ("ref".equals(element.getLocalName())) {
      AttrValue attrValue = element.getAttribute("location");
      if (attrValue != null) {
        String attrStringVal = attrValue.getValue();
        try {
          URL absoluteUrl = new URL(new URL(systemID),
              authorAccess.correctURL(attrStringVal));

          InputSource inputSource = entityResolver.resolveEntity(null,
              absoluteUrl.toString());
          if(inputSource == null) {
            inputSource = new InputSource(absoluteUrl.toString());
          }

          XMLReader xmlReader = authorAccess.newNonValidatingXMLReader();
          xmlReader.setEntityResolver(entityResolver);

          saxSource = new SAXSource(xmlReader, inputSource);
        } catch (MalformedURLException e) {
          logger.error(e, e);
        } catch (SAXException e) {
          logger.error(e, e);
        } catch (IOException e) {
          logger.error(e, e);
        }
      }
    }
  }

  return saxSource;
}
```

5. The method `getReferenceUniqueID` should return an unique identifier for the node reference. The unique identifier is used to avoid resolving the references recursively. The method takes as argument an `AuthorNode` that

represents the node with the reference. In the implementation the unique identifier is the value of the *location* attribute from the *ref* element.

```
public String getDisplayName(AuthorNode node) {
  String displayName = "ref-fragment";
  if (node.getType() == AuthorNode.NODE_TYPE_ELEMENT) {
    AuthorElement element = (AuthorElement) node;
    if ("ref".equals(element.getLocalName())) {
      AttrValue attrValue = element.getAttribute("location");
      if (attrValue != null) {
        displayName = attrValue.getValue();
      }
    }
  }
  return displayName;
}
```

6. The method `getReferenceSystemID`should return the *systemID* of the referred content. It takes as arguments an `AuthorNode` that represents the node with the reference and the `AuthorAccess` with access methods to the Author data model. In the implementation you use the value of the *location* attribute from the *ref* element and resolve it relatively to the XML base URL of the node.

```
public String getReferenceSystemID(AuthorNode node,
                                   AuthorAccess authorAccess) {
  String systemID = null;
  if (node.getType() == AuthorNode.NODE_TYPE_ELEMENT) {
    AuthorElement element = (AuthorElement) node;
    if ("ref".equals(element.getLocalName())) {
      AttrValue attrValue = element.getAttribute("location");
      if (attrValue != null) {
        String attrStringVal = attrValue.getValue();
        try {
          URL absoluteUrl = new URL(node.getXMLBaseURL(),
              authorAccess.correctURL(attrStringVal));
          systemID = absoluteUrl.toString();
        } catch (MalformedURLException e) {
          logger.error(e, e);
        }
      }
    }
  }
  return systemID;
}
```

Note: The complete source code can be found in the Simple Documentation Framework project, included in the *Oxygen Author SDK zip* available for download *on the website*.

In the listing below, the XML document contains the **ref** element:

```
<ref location="referred.xml">Reference</ref>
```

When no reference resolver is specified, the reference has the following layout:

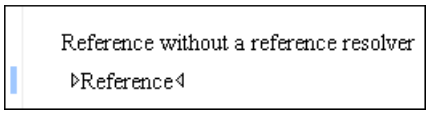


Figure 22: Reference with no specified reference resolver

When the above implementation is configured, the reference has the expected layout:

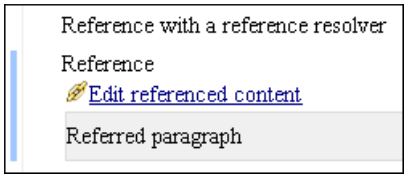


Figure 23: Reference with reference resolver

Configuring CSS Styles Filter

You can modify the CSS styles for each *ro.sync.ecss.extensions.api.node.AuthorNode* rendered in the Author mode using an implementation of *ro.sync.ecss.extensions.api.StylesFilter*. You can implement the various callbacks of the interface either by returning the default value given by or by contributing to the value. The received styles `ro.sync.ecss.css.Styles` can be processed and values can be overwritten with your own. For example you can override the `KEY_BACKGROUND_COLOR` style to return your own implementation of *ro.sync.exml.view.graphics.Color* or override the `KEY_FONT` style to return your own implementation of *ro.sync.exml.view.graphics.Font*.

Note: The Javadoc documentation of the Author API used in the example files is *available on the website*. Also it can be downloaded as a *zip archive from the website*.

For instance in our simple document example the filter can change the value of the `KEY_FONT` property for the `table` element:

```
package simple.documentation.framework;

import ro.sync.ecss.css.Styles;
import ro.sync.ecss.extensions.api.StylesFilter;
import ro.sync.ecss.extensions.api.node.AuthorNode;
import ro.sync.exml.view.graphics.Font;

public class SDFStylesFilter implements StylesFilter {

     public Styles filter(Styles styles, AuthorNode authorNode) {
         if (AuthorNode.NODE_TYPE_ELEMENT == authorNode.getType()
           && "table".equals(authorNode.getName())) {
           styles.setProperty(Styles.KEY_FONT, new Font(null, Font.BOLD, 12));
         }
         return styles;
     }
}
```

Configuring a Table Column Width Provider

In the documentation framework the `table` element as well as the table columns can have specified widths. In order for these widths to be considered by Author we need to provide the means to determine them. As explained in the *Styling the Table Element* section which describes the CSS properties needed for defining a table, if you use the table element attribute **width** can determine the table width automatically. In this example the table has `col` elements with **width** attributes that are not recognized by default. You will need to implement a Java extension class to determine the column widths.

Note: The Javadoc documentation of the Author API used in the example files is *available on the website*. Also it can be downloaded as a *zip archive from the website*.

1. Create the class `simple.documentation.framework.TableColumnWidthProvider`. This class must implement the *ro.sync.ecss.extensions.api.AuthorTableColumnWidthProvider* interface.

```
import ro.sync.ecss.extensions.api.AuthorAccess;
import ro.sync.ecss.extensions.api.AuthorOperationException;
import ro.sync.ecss.extensions.api.AuthorTableColumnWidthProvider;
import ro.sync.ecss.extensions.api.WidthRepresentation;
import ro.sync.ecss.extensions.api.node.AuthorElement;

public class TableColumnWidthProvider
      implements AuthorTableColumnWidthProvider {
```

2. Method `init` is taking as argument an *ro.sync.ecss.extensions.api.node.AuthorElement* that represents the XML `table` element. In our case the column widths are specified in `col` elements from the `table` element. In such cases you must collect the span information by analyzing the `table` element.

```
 public void init(AuthorElement tableElement) {
  this.tableElement = tableElement;
  AuthorElement[] colChildren = tableElement.getElementsByLocalName("customcol");
  if (colChildren != null && colChildren.length > 0) {
   for (int i = 0; i < colChildren.length; i++) {
    AuthorElement colChild = colChildren[i];
    if (i == 0) {
```

```
    colsStartOffset = colChild.getStartOffset();
   }
   if (i == colChildren.length - 1) {
    colsEndOffset = colChild.getEndOffset();
   }
   // Determine the 'width' for this col.
   AttrValue colWidthAttribute = colChild.getAttribute("width");
   String colWidth = null;
   if (colWidthAttribute != null) {
    colWidth = colWidthAttribute.getValue();
    // Add WidthRepresentation objects for the columns this 'customcol' specification
    // spans over.
    colWidthSpecs.add(new WidthRepresentation(colWidth, true));
   }
  }
 }
}
```

3. The method `isTableAcceptingWidth` should check if the table cells are `td`.

```
public boolean isTableAcceptingWidth(String tableCellsTagName) {
  return "td".equals(tableCellsTagName);
}
```

4. The method `isTableAndColumnsResizable` should check if the table cells are `td`. This method determines if the table and its columns can be resized by dragging the edge of a column.

```
public boolean isTableAndColumnsResizable(String tableCellsTagName) {
  return "td".equals(tableCellsTagName);
}
```

5. Methods `getTableWidth` and `getCellWidth` are used to determine the table and column width. The table layout engine will ask this *`ro.sync.ecss.extensions.api.AuthorTableColumnWidthProvider`* implementation what is the table width for each table element and the cell width for each cell element from the table that was marked as cell in the CSS using the property `display:table-cell`. The implementation is simple and just parses the value of the **width** attribute. The methods must return `null` for the tables / cells that do not have a specified width.

```
public WidthRepresentation getTableWidth(String tableCellsTagName) {
 WidthRepresentation toReturn = null;
 if (tableElement != null && "td".equals(tableCellsTagName)) {
  AttrValue widthAttr = tableElement.getAttribute("width");
  if (widthAttr != null) {
   String width = widthAttr.getValue();
   if (width != null) {
    toReturn = new WidthRepresentation(width, true);
   }
  }
 }
 return toReturn;
}
```

```
public List<WidthRepresentation> getCellWidth(AuthorElement cellElement, int colNumberStart,
  int colSpan) {
 List<WidthRepresentation> toReturn = null;
 int size = colWidthSpecs.size();
 if (size >= colNumberStart && size >= colNumberStart + colSpan) {
  toReturn = new ArrayList<WidthRepresentation>(colSpan);
  for (int i = colNumberStart; i < colNumberStart + colSpan; i ++) {
   // Add the column widths
   toReturn.add(colWidthSpecs.get(i));
  }
 }
 return toReturn;
}
```

6. Methods `commitTableWidthModification` and `commitColumnWidthModifications` are used to commit changes made to the width of the table or its columns when using the mouse drag gestures.

```
public void commitTableWidthModification(AuthorDocumentController authorDocumentController,
  int newTableWidth, String tableCellsTagName) throws AuthorOperationException {
 if ("td".equals(tableCellsTagName)) {
  if (newTableWidth > 0) {
   if (tableElement != null) {
    String newWidth = String.valueOf(newTableWidth);

    authorDocumentController.setAttribute(
```

```
        "width",
        new AttrValue(newWidth),
        tableElement);
     } else {
      throw new AuthorOperationException("Cannot find the element representing the table.");
     }
    }
   }
  }
```

```
public void commitColumnWidthModifications(AuthorDocumentController authorDocumentController,
   WidthRepresentation[] colWidths, String tableCellsTagName) throws AuthorOperationException {
  if ("td".equals(tableCellsTagName)) {
   if (colWidths != null && tableElement != null) {
    if (colsStartOffset >= 0 && colsEndOffset >= 0 && colsStartOffset < colsEndOffset) {
     authorDocumentController.delete(colsStartOffset,
       colsEndOffset);
    }
    String xmlFragment = createXMLFragment(colWidths);
    int offset = -1;
    AuthorElement[] header = tableElement.getElementsByLocalName("header");
    if (header != null && header.length > 0) {
     // Insert the cols elements before the 'header' element
     offset = header[0].getStartOffset();
    }
    if (offset == -1) {
     throw new AuthorOperationException("No valid offset to insert the columns width specification.");
    }
    authorDocumentController.insertXMLFragment(xmlFragment, offset);
   }
  }
 }

 private String createXMLFragment(WidthRepresentation[] widthRepresentations) {
  StringBuffer fragment = new StringBuffer();
  String ns = tableElement.getNamespace();
  for (int i = 0; i < widthRepresentations.length; i++) {
   WidthRepresentation width = widthRepresentations[i];
   fragment.append("<customcol");
   String strRepresentation = width.getWidthRepresentation();
   if (strRepresentation != null) {
    fragment.append(" width=\"" + width.getWidthRepresentation() + "\"");
   }
   if (ns != null && ns.length() > 0) {
    fragment.append(" xmlns=\"" + ns + "\"");
   }
   fragment.append("/>");
  }
  return fragment.toString();
 }
```

7. The following three methods are used to determine what type of column width specifications the table column width provider support. In our case all types of specifications are allowed:

```
 public boolean isAcceptingFixedColumnWidths(String tableCellsTagName) {
  return true;
 }

 public boolean isAcceptingPercentageColumnWidths(String tableCellsTagName) {
  return true;
 }

 public boolean isAcceptingProportionalColumnWidths(String tableCellsTagName) {
  return true;
 }
```

Note: The complete source code can be found in the Simple Documentation Framework project, included in the *Oxygen Author SDK zip* available for download *on the website*.

In the listing below, the XML document contains the table element:

```
<table width="300">
    <customcol width="50.0px"/>
    <customcol width="1*"/>
    <customcol width="2*"/>
    <customcol width="20%"/>
    <header>
        <td>C1</td>
        <td>C2</td>
        <td>C3</td>
```

```
        <td>C4</td>
    </header>
    <tr>
        <td>cs=1, rs=1</td>
        <td>cs=1, rs=1</td>
        <td row_span="2">cs=1, rs=2</td>
        <td row_span="3">cs=1, rs=3</td>
    </tr>
    <tr>
        <td>cs=1, rs=1</td>
        <td>cs=1, rs=1</td>
    </tr>
    <tr>
        <td column_span="3">cs=3, rs=1</td>
    </tr>
</table>
```

When no table column width provider is specified, the table has the following layout:

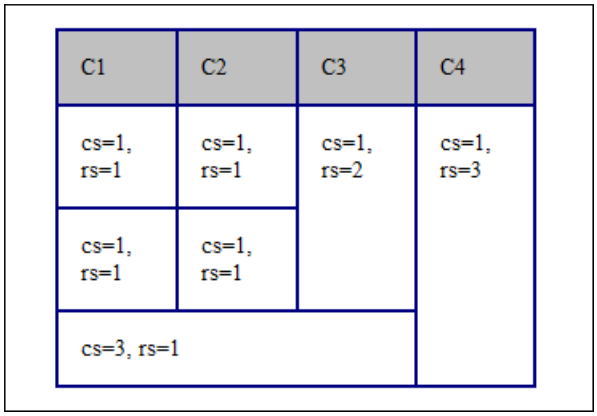


Figure 24: Table layout when no column width provider is specified

When the above implementation is configured, the table has the correct layout:

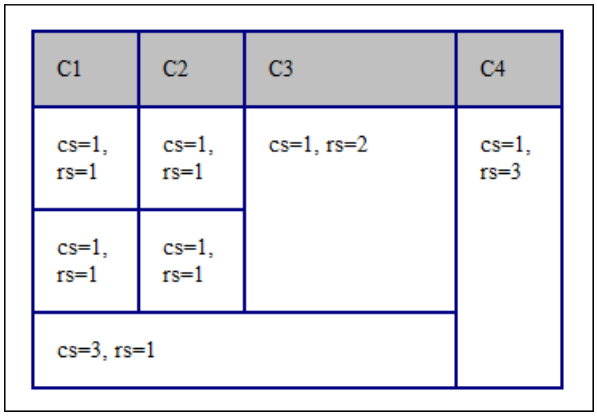


Figure 25: Columns with custom widths

Configuring a Table Cell Span Provider

In the documentation framework the `table` element can have cells that span over multiple columns and rows. As explained in the *Styling the Table Element* section which describes the CSS properties needed for defining a table, you need to indicate a method to determine the cell spanning. If you use the cell element attributes **rowspan** and **colspan** or **rows** and **cols**, can determine the cell spanning automatically. In our example the `td` element uses the attributes **row_span** and **column_span** that are not recognized by default. You will need to implement a Java extension class for defining the cell spanning.

Note: The Javadoc documentation of the Author API used in the example files is *available on the website*. Also it can be downloaded as a *zip archive from the website*.

1. Create the class `simple.documentation.framework.TableCellSpanProvider`. This class must implement the *`ro.sync.ecss.extensions.api.AuthorTableCellSpanProvider`* interface.

```
import ro.sync.ecss.extensions.api.AuthorTableCellSpanProvider;
import ro.sync.ecss.extensions.api.node.AttrValue;
import ro.sync.ecss.extensions.api.node.AuthorElement;

public class TableCellSpanProvider
      implements AuthorTableCellSpanProvider {
```

2. The `init` method is taking as argument the *`ro.sync.ecss.extensions.api.node.AuthorElement`* that represents the XML `table` element. In our case the cell span is specified for each of the cells so you leave this method empty. However there are cases like the table CALS model when the cell spanning is specified in the `table` element. In such cases you must collect the span information by analyzing the `table` element.

```
public void init(AuthorElement table) {
}
```

3. The `getColSpan` method is taking as argument the table cell. The table layout engine will ask this `AuthorTableSpanSupport` implementation what is the column span and the row span for each XML element from the table that was marked as cell in the CSS using the property `display:table-cell`. The implementation is simple and just parses the value of **column_span** attribute. The method must return `null` for all the cells that do not change the span specification.

```
public Integer getColSpan(AuthorElement cell) {
  Integer colSpan = null;

  AttrValue attrValue = cell.getAttribute("column_span");
  if(attrValue != null) {
    // The attribute was found.
    String cs = attrValue.getValue();
    if(cs != null) {
      try {
        colSpan = new Integer(cs);
      } catch (NumberFormatException ex) {
        // The attribute value was not a number.
      }
    }
  }
  return colSpan;
}
```

4. The row span is determined in a similar manner:

```
public Integer getRowSpan(AuthorElement cell) {
  Integer rowSpan = null;

  AttrValue attrValue = cell.getAttribute("row_span");
  if(attrValue != null) {
    // The attribute was found.
    String rs = attrValue.getValue();
    if(rs != null) {
      try {
        rowSpan = new Integer(rs);
      } catch (NumberFormatException ex) {
        // The attribute value was not a number.
      }
    }
  }
  return rowSpan;
}
```

5. The method `hasColumnSpecifications` always returns `true` considering column specifications always available.

```
public boolean hasColumnSpecifications(AuthorElement tableElement) {
  return true;
}
```

Note: The complete source code can be found in the Simple Documentation Framework project, included in the *Oxygen Author SDK zip* available for download *on the website*.

6. In the listing below, the XML document contains the table element:

```
<table>
    <header>
        <td>C1</td>
        <td>C2</td>
        <td>C3</td>
        <td>C4</td>
    </header>
    <tr>
        <td>cs=1, rs=1</td>
        <td column_span="2" row_span="2">cs=2, rs=2</td>
        <td row_span="3">cs=1, rs=3</td>
    </tr>
    <tr>
        <td>cs=1, rs=1</td>
    </tr>
    <tr>
        <td column_span="3">cs=3, rs=1</td>
    </tr>
</table>
```

When no table cell span provider is specified, the table has the following layout:

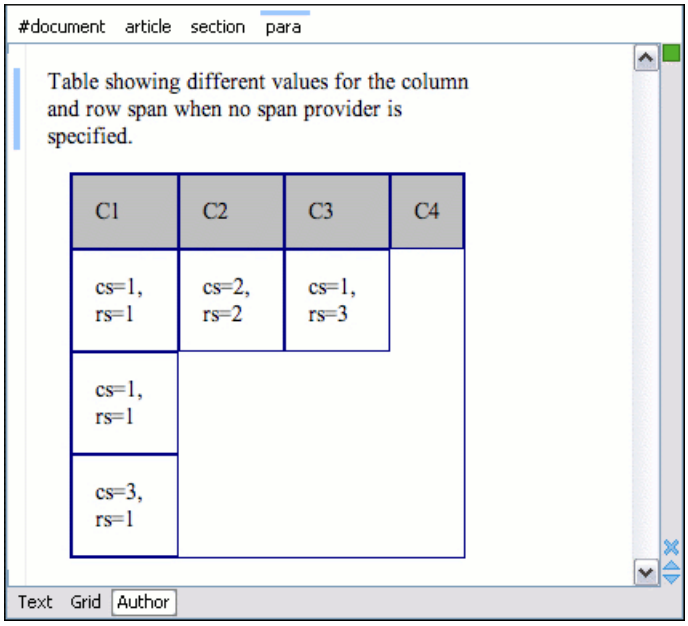


Figure 26: Table layout when no cell span provider is specified

When the above implementation is configured, the table has the correct layout:

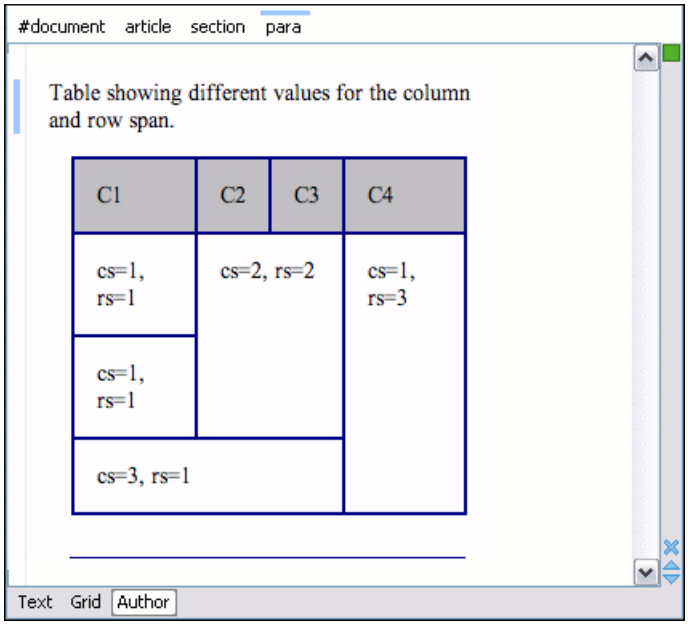


Figure 27: Cells spanning multiple rows and columns.

Configuring an Unique Attributes Recognizer

The *`ro.sync.ecss.extensions.api.UniqueAttributesRecognizer`* interface can be implemented if you want to provide for your framework the following features:

Note: The Javadoc documentation of the Author API used in the example files is *available on the website*. Also it can be downloaded as a *zip archive from the website*.

- **Automatic ID generation** - You can automatically generate unique IDs for newly inserted elements. Implementations are already available for the DITA and Docbook frameworks. The following methods can be implemented to accomplish this: `assignUniqueIDs(int startOffset, int endOffset)`, `isAutoIDGenerationActive()`
- **Avoiding copying unique attributes when "Split" is called inside an element** - You can split the current block element by pressing the "Enter" key and then choosing "Split". This is a very useful way to create new paragraphs, for example. All attributes are by default copied on the new element but if those attributes are IDs you sometimes want to avoid creating validation errors in the editor. Implementing the following method, you can decide whether an attribute should be copied or not during the split: `boolean copyAttributeOnSplit(String attrQName, AuthorElement element)`

  Tip:

  The *`ro.sync.ecss.extensions.commons.id.DefaultUniqueAttributesRecognizer`* class is an implementation of the interface which can be extended by your customization to provide easy assignation of IDs in your framework. You can also check out the DITA and Docbook implementations of `ro.sync.ecss.extensions.api.UniqueAttributesRecognizer` to see how they were implemented and connected to the extensions bundle.

Configuring an XML Node Renderer Customizer

You can use this API extension to customize the way an XML node is rendered in the **Author Outline** view, **Author** breadcrumb navigation bar, **Text** mode **Outline** view, content completion assistant window or **DITA Maps Manager** view.

Note: uses XMLNodeRendererCustomizer implementations for the following frameworks: DITA, DITAMap, Docbook 4, Docbook 5, TEI P4, TEI P5, XHTML, XSLT, and XML Schema.

Note: The Javadoc documentation of the Author API used in the example files is *available on the website*. Also it can be downloaded as a *zip archive from the website*.

There are two methods to provide an implementation of `ro.sync.exml.workspace.api.node.customizer.XMLNodeRendererCustomizer`:

- as a part of a bundle - returning it from the `createXMLNodeCustomizer()` method of the *`ExtensionsBundle`* associated with your document type in the **Document type** dialog, **Extensions** tab, **Extensions bundle** field.
- as an individual extension - associated with your document type in the **Document type** dialog, **Extensions** tab, **Individual extensions** section, **XML node renderer customizer** field.

Styling the `table` Element.

There are standard CSS properties used to indicate what elements are tables, table rows and table cells. What CSS is missing is the possibility to indicate the cell spanning. Author offers support for adding an extension to solve this problem. This will be presented in the next chapters.

The table in this example is a simple one. The header must be formatted in a different way than the ordinary rows, so it will have a background color.

```
table{
    display:table;
    border:1px solid navy;
    margin:1em;
    max-width:1000px;
    min-width:150px;
}

table[width]{
  width:attr(width, length);
}

tr, header{
    display:table-row;
}

header{
    background-color: silver;
    color:inherit
}

td{
  display:table-cell;
  border:1px solid navy;
  padding:1em;
}
```

Because in the schema the `td` tag has the attributes **row_span** and **column_span** that are not automatically recognized by Author, a Java extension will be implemented which will provide information about the cell spanning. See the section *Configuring a Table Cell Span Provider*.

Because the column widths are specified by the attributes **width** of the elements `customcol` that are not automatically recognized by Author, it is necessary to implement a Java extension which will provide information about the column widths. See the section *Configuring a Table Column Width Provider*.

Sample project reference

Note: The complete source code can be found in the Simple Documentation Framework project, included in the *Oxygen Author SDK zip* available for download *on the website*.

Note: The complete source code of the *`ro.sync.ecss.extensions.commons.DefaultElementLocatorProvider`*, *`ro.sync.ecss.extensions.commons.IDElementLocator`* or *`ro.sync.ecss.extensions.commons.XPointerElementLocator`* can be found in the Oxygen Default Frameworks project, included in the *Oxygen Author SDK zip* available for download *on the website*.

Note: The Javadoc documentation of the Author API used in the example files is *available on the website*. Also it can be downloaded as a *zip archive from the website*.

Customizing the Default CSS of a Document Type

The easiest way of customizing the default CSS stylesheet of a document type is to create a new CSS stylesheet in the same folder as the customized one, import the customized CSS stylesheet and set the new stylesheet as the default CSS of the document type. For example let us customize the default CSS for DITA documents by changing the background color of the *task* and *topic* elements to red.

1. First you create a new CSS stylesheet called `my_dita.css` in the folder `${frameworks}/dita/css_classed` where the default stylesheet called `dita.css` is located. `${frameworks}` is the subfolder `frameworks` of the Editor. The new stylesheet `my_dita.css` contains:

```
@import "dita.css";

task, topic{
    background-color:red;
}
```

2. To set the new stylesheet as the default CSS stylesheet for DITA documents first open the Document Type Association preferences panel from menu **Options** > **Preferences** > **Document Type Association**. Select the DITA document type and start editing it by pressing the Edit button. In the Author tab of the document type edit dialog change the URI of the default CSS stylesheet from `${frameworks}/dita/css_classed/dita.css` to `${frameworks}/dita/css_classed/my_dita.css`.

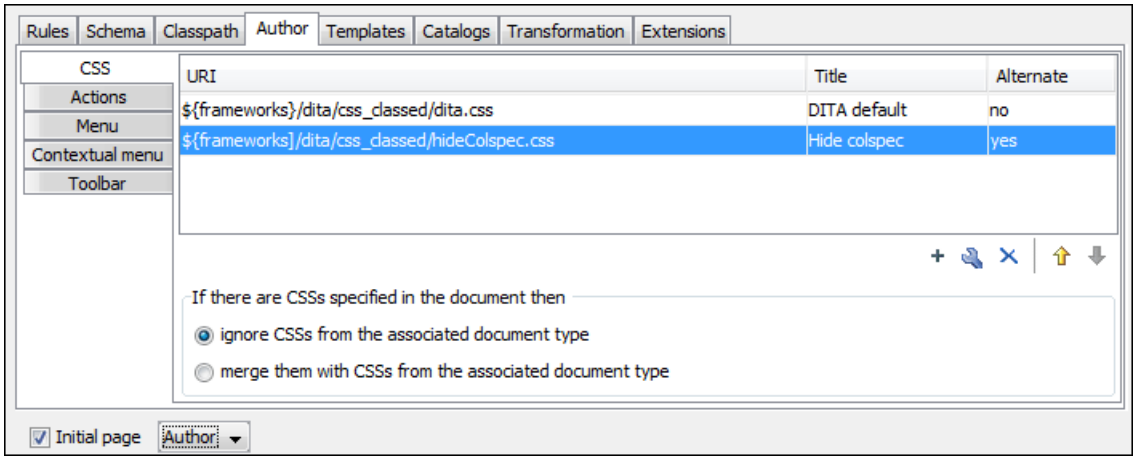


Figure 28: Set the location of the default CSS stylesheet

3. Press OK in all the dialogs to validate the changes. Now you can start editing DITA documents based on the new CSS stylesheet. You can edit the new CSS stylesheet itself at any time and see the effects on rendering DITA XML documents in the Author mode by running the *Refresh* action available on the Author toolbar and on the DITA menu.

Document Type Sharing

Oxygen has support for allowing you to share the customizations for a specific XML type by creating your own *Document Type* in the *Document Type Association* preferences page.

A document type can be shared between authors in two ways:

- Save it externally in a separate framework folder in the `OXYGEN_INSTALL_DIR/frameworks` directory.

  Important: In order for this approach to work you will need to have the application installed to a folder with full write access.

  Please follow the following steps:

  1. Create a new directory in the `OXYGEN_INSTALL_DIR/frameworks` for your new framework. This directory will contain resources for your framework (CSS files, new file templates, schemas used for validation, catalogs). See the **Docbook** framework structure from the `OXYGEN_INSTALL_DIR/frameworks/docbook` as an example.

2. Create your new custom document type and save it external in the newly created framework directory (with a name like `custom.framework`).
3. Configure the custom document type according to your needs, take special care to make all file references relative to the `OXYGEN_INSTALL_DIR/frameworks` directory by using the `${frameworks}` editor variable. The *Author Developer Guide* contains all details necessary for creating and configuring a new document type.
4. If everything went fine then you should have a new configuration file saved in: `OXYGEN_INSTALL_DIR/frameworks/your_framework_dir/custom.framework` after the Preferences are saved.
5. You can then share the new framework directory with other users (have them copy it to their `OXYGEN_INSTALL_DIR/frameworks` directory) and the new document type will be available in the list of Document Types when the application is started.

Adding Custom Persistent Highlights

The Author API allows you to create or remove custom persistent highlights, set their properties, and customize their appearance. They get serialized in the XML document as processing instructions, having the following format:

```
<?oxy_custom_start prop1="val1"....?> xml content <?oxy_custom_end?>
```

The functionality is available in the `AuthorPersistentHighlighter` class, accessible through `AuthorEditorAccess#getPersistentHighlighter()` method. For more information, see JavaDoc online at: *http://www.oxygenxml.com/InstData/Editor/SDK/javadoc/index.html*

CSS Support in Author

Author editing mode supports most CSS 2.1 selectors, a lot of CSS 2.1 properties and a few CSS 3 selectors. Also some custom functions and properties that extend the W3C CSS specification and are useful for URL and string manipulation are available to the developer who creates an Author editing framework.

Supported CSS Selectors

Expression	Name	Description / Example
*	Universal selector	Matches any element
E	Type selector	Matches any E element (i. e. an element with the local name E)
E F	Descendant selector	Matches any F element that is a descendant of an E element.
E > F	Child selectors	Matches any F element that is a child of an element E.
E:first-child	The :first-child pseudo-class	Matches element E when E is the first child of its parent.
E:lang(c)	The :lang() pseudo-class	Matches element of type E if it is in (human) language c (the document language specifies how language is determined).
E + F	Adjacent selector	Matches any F element immediately preceded by a sibling element E.
E[foo]	Attribute selector	Matches any E element with the "foo" attribute set (whatever the value).

Expression	Name	Description / Example
`E[foo="warning"]`	Attribute selector	Matches any `E` element whose `"foo"` attribute value is exactly equal to `"warning"`.
`E[foo~="warning"]`	Attribute selector	Matches any `E` element whose `"foo"` attribute value is a list of space-separated values, one of which is exactly equal to `"warning"`.
`E[lang|="en"]`	Attribute selector	Matches any `E` element whose `"lang"` attribute has a hyphen-separated list of values beginning (from the left) with `"en"`.
`E:before and E:after`	Pseudo elements	The `':before'` and `':after'` pseudo-elements can be used to insert generated content before or after an element's content.
`E[att^="val"]`	CSS 3 attribute selector	An E element whose `att` attribute value begins exactly with the string `val`.
`E[att$="val"]`	CSS 3 attribute selector	An E element whose `att` attribute value ends exactly with the string `val`.
`E[att*="val"]`	CSS 3 attribute selector	An E element whose `att` attribute value contains the substring `val`.
`E:root`	CSS 3 pseudo-class	Matches the root element of the document. In HTML, the root element is always the HTML element.
`E:empty`	CSS 3 pseudo element	An E element which has no text or child elements.

CSS 2.1 Features

This section enumerates the CSS 2.1 features that supports.

CSS 2.1 Properties

validates all CSS 2.1 properties, but does not render in Author mode *aural* and *paged* categories properties, as well as some of the values of the *visual* category, listed below under the **Ignored Values** column.

Name	Rendered Values	Ignored Values
`'background-attachment'`		ALL
`'background-color'`	`<color> | inherit`	`transparent`
`'background-image'`	`<uri> | none | inherit`	

Name	Rendered Values	Ignored Values
'background-position'	top | right | bottom | left | center	<percentage> | <length>
'background-repeat'	repeat | repeat-x | repeat-y | no-repeat | inherit	
'background'		ALL
'border-collapse'		ALL
'border-color'	<color> | inherit	transparent
'border-spacing'		ALL
'border-style'	<border-style> | inherit	
'border-top' 'border-right' 'border-bottom' 'border-left'	[<border-width> || <border-style> || 'border-top-color'] | inherit	
'border-top-color' 'border-right-color' 'border-bottom-color' 'border-left-color'	<color> | inherit	transparent
'border-top-style' 'border-right-style' 'border-bottom-style' 'border-left-style'	<border-style> | inherit	
'border-top-width' 'border-right-width' 'border-bottom-width' 'border-left-width'	<border-width> | inherit	
'border-width'	<border-width> | inherit	
'border'	[<border-width> || <border-style> || 'border-top-color'] | inherit	
'bottom'		ALL
'caption-side'		ALL
'clear'		ALL
'clip'		ALL
'color'	<color> | inherit	
'content'	normal | none | [<string> | <URI> | <counter> | attr(<identifier>) | open-quote | close-quote]+ | inherit	no-open-quote | no-close-quote
'counter-increment'	[<identifier> <integer> ?]+ | none | inherit	

Name	Rendered Values	Ignored Values
'counter-reset'	[<identifier> <integer> ?]+ | none | inherit	
'cursor'		ALL
'direction'	ltr	rtl | inherit
'display'	inline | block | list-item | table | table-row-group | table-header-group | table-footer-group | table-row | table-column-group | table-column | table-cell | table-caption | none | inherit	run-in | inline-block | inline-table - considered block
'empty-cells'	show | hide | inherit	
'float'		ALL
'font-family'	[[<family-name> | <generic-family>] [, <family-name> | <generic-family>]*] | inherit	
'font-size'	<absolute-size> | <relative-size> | <length> | <percentage> | inherit	
'font-style'	normal | italic | oblique | inherit	
'font-variant'		ALL
'font-weight'	normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900 | inherit	
'font'	[['font-style' || 'font-weight']? 'font-size' [/ 'line-height']? 'font-family'] | inherit	'font-variant' 'line-height' caption | icon | menu | message-box | small-caption | status-bar
'height'		ALL
'left'		ALL
'letter-spacing'		ALL
'line-height'	normal | <number> | <length> | <percentage> | inherit	
'list-style-image'		ALL
'list-style-position'		ALL

Name	**Rendered Values**	**Ignored Values**
'list-style-type'	disc | circle | square | decimal | lower-roman | upper-roman | lower-latin | upper-latin | lower-alpha | upper-alpha | -oxy-lower-cyrillic-ru | -oxy-lower-cyrillic-uk | -oxy-upper-cyrillic-ru | -oxy-upper-cyrillic-uk | box | diamond | check | hyphen | none | inherit	lower-greek | armenian | georgian
'list-style'	['list-style-type'] | inherit	'list-style-position' || 'list-style-image'
'margin-right' 'margin-left'	<margin-width> | inherit | auto	
'margin-top' 'margin-bottom'	<margin-width> | inherit	
'margin'	<margin-width> | inherit | auto	
'max-height'		ALL
'max-width'	<length> | <percentage> | none | inherit - supported for inline, block-level, and replaced elements, e.g. images, tables, table cells.	
'min-height'		ALL
'min-width'	<length> | <percentage> | inherit - supported for inline, block-level, and replaced elements, e. g. images, tables, table cells.	
'outline-color'		ALL
'outline-style'		ALL
'outline-width'		ALL
'outline'		ALL
'overflow'		ALL
'padding-top' 'padding-right' 'padding-bottom' 'padding-left'	<padding-width> | inherit	
'padding'	<padding-width> | inherit	
'position'		ALL
'quotes'		ALL
'right'		ALL
'table-layout'	auto	fixed | inherit

Name	Rendered Values	Ignored Values
'text-align'	left | right | center | inherit	justify
'text-decoration'	none | [underline || overline || line-through] | inherit	blink
'text-indent'		ALL
'text-transform'	ALL	
'top'		ALL
'unicode-bidi'		ALL
'vertical-align'	baseline | sub | super | top | text-top | middle | bottom | text-bottom | inherit	<percentage> | <length>
'visibility'	visible | hidden | inherit | -oxy-collapse-text	collapse
'white-space'	normal | pre | nowrap | pre-wrap | pre-line	
'width'	<length> | <percentage> | auto | inherit - supported for inline, block-level, and replaced elements, e.g. images, tables, table cells.	
'word-spacing'		ALL
'z-index'		ALL

CSS 3 Features

This section enumerates the CSS 3 features that supports.

CSS 3 Selectors

Namespace Selectors

In the CSS 2.1 standard the element selectors are ignoring the namespaces of the elements they are matching. Only the local name of the elements are considered in the selector matching process.

Author uses a different approach similar to the CSS Level 3 specification. If the element name from the CSS selector is not preceded by a namespace prefix it is considered to match an element with the same local name as the selector value and ANY namespace, otherwise the element must match both the local name and the namespace.

In CSS up to version 2.1 the name tokens from selectors are matching all elements from ANY namespace that have the same local name. Example:

```
<x:b xmlns:x="ns_x"/>
<y:b xmlns:y="ns_y"/>
```

Are both matched by the rule:

```
b {font-weight:bold}
```

Starting with CSS Level 3 you can create selectors that are namespace aware.

Defining both prefixed namespaces and the default namespace

Given the namespace declarations:

```
@namespace sync "http://sync.example.org";
@namespace "http://example.com/foo";
```

In a context where the default namespace applies:

sync|A	represents the name A in the `http://sync.example.org` namespace.
|B	represents the name B that belongs to NO NAMESPACE.
***|C**	represents the name C in ANY namespace, including NO NAMESPACE.
D	represents the name D in the `http://example.com/foo` namespace.

Defining only prefixed namespaces

Given the namespace declaration:

```
@namespace sync "http://sync.example.org";
```

Then:

sync|A	represents the name A in the `http://sync.example.org` namespace.
|B	represents the name B that belongs to NO NAMESPACE.
***|C**	represents the name C in ANY namespace, including NO NAMESPACE.
D	represents the name D in ANY namespace, including NO NAMESPACE.

Transparent Colors

CSS3 supports RGBA colors. The RGBA declaration allows you to set opacity (via the Alpha channel) as part of the color value. A value of `0` corresponds to a completely transparent color, while a value of `1` corresponds to a completely opaque color. To specify a value, you can use either a *real* number between `0` and `1`, or a percent.

RGBA color

```
personnel:before {
    display:block;
    padding: 1em;
    font-size: 1.8em;
    content: "Employees";
    font-weight: bold;
    color:#EEEEEE;
    background-color: rgba(50, 50, 50, 0.6);
}
```

The `attr()` Function: Properties Values Collected from the Edited Document.

In CSS Level 2.1 you may collect attribute values and use them as content *only* for the pseudo-elements. For instance the :before pseudo-element can be used to insert some content before an element. This is valid in CSS 2.1:

```
title:before{
  content: "Title id=(" attr(id) ")";
}
```

If the `title` element from the XML document is:

```
<title id="title12">My title.</title>
```

Then the title will be displayed as:

```
                          Title id=(title12) My title.
```

In Author the use of `attr()` function is available not only for the `content` property, but also for any other property. This is similar to the CSS Level 3 working draft: *http://www.w3.org/TR/2006/WD-css3-values-20060919/#functional*. The arguments of the function are:

`attr( attribute_name , attribute_type , default_value )`

`attribute_name` The attribute name. This argument is required.

`attribute_type` The attribute type. This argument is optional. If it is missing, argument's type is considered `string`. This argument indicates what is the meaning of the attribute value and helps to perform conversions of this value. Author accepts one of the following types:

`color`	The value represents a color. The attribute may specify a color in different formats. Author supports colors specified either by name: `red`, `blue`, `green`, etc. or as an RGB hexadecimal value `#FFEEFF`.
`url`	The value is an URL pointing to a media object. Author supports only images. The attribute value can be a complete URL, or a relative one to the XML document. Please note that this URL is also resolved through the catalog resolver.
`integer`	The value must be interpreted as an integer.
`number`	The value must be interpreted as a float number.
`length`	The value must be interpreted as an integer.
`percentage`	The value must be interpreted relative to another value (length, size) expressed in percents.
`em`	The value must be interpreted as a size. 1 em is equal to the *font-size* of the relevant font.
`ex`	The value must be interpreted as a size. 1 ex is equal to the *height* of the **x** character of the relevant font.
`px`	The value must be interpreted as a size expressed in pixels relative to the viewing device.
`mm`	The value must be interpreted as a size expressed in millimeters.
`cm`	The value must be interpreted as a size expressed in centimeters.
`in`	The value must be interpreted as a size expressed in inches. 1 inch is equal to 2.54 centimeters.
`pt`	The value must be interpreted as a size expressed in points. The points used by CSS2 are equal to 1/72th of an inch.
`pc`	The value must be interpreted as a size expressed in picas. 1 pica is equal to 12 points.

`default_value` This argument specifies a value that is used by default if the attribute value is missing. This argument is optional.

Usage samples for the attr() function

Consider the following XML instance:

```
<sample>
    <para bg_color="#AAAAFF">Blue paragraph.</para>
    <para bg_color="red">Red paragraph.</para>
    <para bg_color="red" font_size="2">Red paragraph with large font.</para>
    <para bg_color="#00AA00" font_size="0.8" space="4">
        Green paragraph with small font and margin.</para>
</sample>
```

The `para` elements have `bg_color` attributes with RGB color values like #AAAAFF. You can use the `attr()` function to change the elements appearance in the editor based on the value of this attribute:

```
background-color:attr(bg_color, color);
```

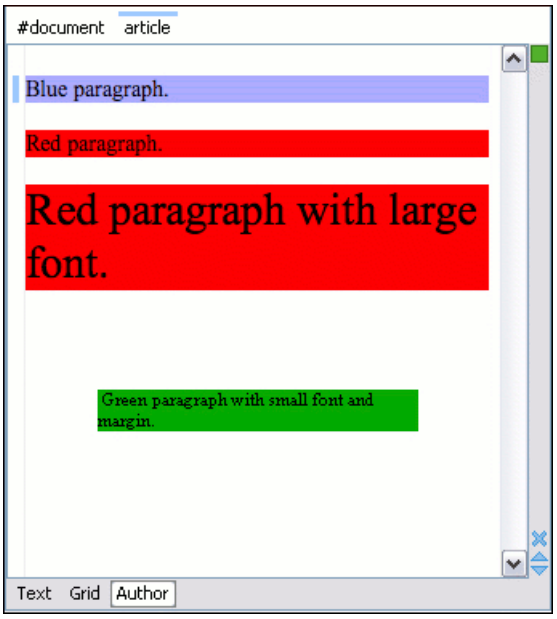
The attribute `font_size` represents the font size in *em* units. You can use this value to change the style of the element:

```
font-size:attr(font_size, em);
```

The complete CSS rule is:

```
para{
 display:block;
 background-color:attr(bg_color, color);
 font-size:attr(font_size, em);
 margin:attr(space, em);
}
```

The document is rendered as:



The @font-face at-rule

allows you to use custom fonts in the Author mode by specifying them in the CSS using the `@font-face` media type. Only the `src` and `font-family` CSS properties can be used for this media type.

```
@font-face{
    font-family:"Baroque Script";
```

```
    /*The location of the loaded TTF font must be relative to the CSS*/
    src:url("BaroqueScript.ttf");
}
```

Styling Elements from other Namespace

In the CSS Level 1, 2, and 2.1 there is no way to specify if an element X from the namespace Y should be presented differently from the element X from the namespace Z. In the upcoming CSS Level 3, it is possible to differentiate elements by their namespaces. Author supports this CSS Level 3 functionality. For more information see the *Namespace Selectors* section.

To match the `def` element its namespace will be declared, bind it to the *abs* prefix, and then write a CSS rule:

```
@namespace abs "http://www.oxygenxml.com/sample/documentation/abstracts";

abs|def{
    font-family:monospace;
    font-size:smaller;
}
abs|def:before{
    content:"Definition:";
    color:gray;
}
```

Additional Custom Selectors

Author provides support for selecting additional types of nodes. These custom selectors apply to: *document*, *doctype sections*, *processing-instructions*, *comments*, *CDATA sections*, *reference sections*, and *entities*.

Note: The custom selectors are presented in the default CSS for Author mode and all of their properties are marked with an *!important* flag. For this reason, you have to set the *!important* flag on each property of the custom selectors from your CSS to be applicable.

In order for the custom selectors to work in your CSSs, declare the Author extensions namespace at the beginning of the stylesheet documents:

```
@namespace oxy url('http://www.oxygenxml.com/extensions/author');
```

- The *oxy*|*document* selector matches the entire document:

```
oxy|document {
    display:block !important;
}
```

- The following example changes the rendering of doctype sections:

```
oxy|doctype {
    display:block !important;
    color:blue !important;
    background-color:transparent !important;
}
```

- To match the processing instructions, you can use the *oxy*|*processing-instruction* selector:

```
oxy|processing-instruction {
    display:block !important;
    color:purple !important;
    background-color:transparent !important;
}
```

- The XML comments display in Author mode can be changed using the *oxy*|*comment* selector:

```
oxy|comment {
    display:block !important;
    color:green !important;
    background-color:transparent !important;
}
```

- The *oxy|cdata* selector matches CDATA sections:

```
oxy|cdata{
    display:block !important;
    color:gray !important;
    background-color:transparent !important;
}
```

- The *oxy|entity* selector matches the entities content:

```
oxy|entity {
    display:morph !important;
    editable:false !important;
    color:orange !important;
    background-color:transparent !important;
}
```

- The *references to entities*, *XInclude*, and *DITA conrefs* are expanded by default in Author mode and the referred content is displayed. The referred resources are loaded and displayed inside the element or entity that refers them.
  - You can use the *reference* property to customize the way these references are rendered in Author mode:

```
oxy|reference {
  border:1px solid gray !important;
}
```

  In Author mode, content is highlighted when parts of text contain:

  - comments;
  - changes and *Track Changes* was active when the content was modified.

  If this content is referred, the Author mode does not display the highlighted areas in the new context. If you want to mark the existence of this comments and changes you can use the *oxy|reference[comments]*, *oxy|reference[changeTracking]*, and *oxy|reference[changeTracking][comments]* selectors.

  Note: Two artificial attributes (*comments* and *changeTracking*) are set on the reference node, containing information about the number of comments and track changes in the content.

  - The following example represents the customization of the reference fragments that contain comments:

```
oxy|reference[comments]:before {
  content: "Comments: " attr(comments) !important;
}
```

  - To match reference fragments based on the fact that they contain change tracking inside, use the *oxy|reference[changeTracking]* selector.

```
oxy|reference[changeTracking]:before {
  content: "Change tracking: " attr(changeTracking) !important;
}
```

  - Here is an example of how you can set a custom color to the reference containing both track changes and comments:

```
oxy|reference[changeTracking][comments]:before {
  content: "Change tracking: " attr(changeTracking) " and comments: " attr(comments) !important;
}
```

A sample document rendered using these rules:

Oxygen CSS Extensions

CSS stylesheets provide support mainly for displaying documents. When editing documents some extensions of the W3C CSS specification are useful, for example:

- property for marking foldable elements in large files
- enforcing a display mode for the XML tags regardless of the current mode selected by the author user
- construct a URL from a relative path location
- string processing functions

Media Type oxygen

The style sheets can specify how a document is to be presented on different media: on the screen, on paper, speech synthesizer, etc. You can specify that some of the features of your CSS stylesheet should be taken into account only in the Author and ignored in the rest. This can be accomplished by using the media type oxygen.

For instance using the following CSS:

```
b{
 font-weight:bold;
 display:inline;
}

@media oxygen{
 b{
  text-decoration:underline;
 }
}
```

would make a text bold if the document was opened in a web browser that does not recognize `@media oxygen` and bold and underlined in Author.

You can use this media type to group specific CSS features and also to hide them when opening the documents with other viewers.

Folding Elements: `-oxy-foldable`, `-oxy-not-foldable-child` and `-oxy-folded` properties

Author allows you to declare some elements to be *foldable* (collapsible). This is especially useful when working with large documents organized in logical blocks, editing a large DocBook article or book for instance. marks the foldable content with a small blue triangle. When you hover with your mouse pointer over this marker, a dotted line borders the collapsible content. The following contextual actions are available:

- **(Ctrl+NumPad+/)** > **Document** > **Folding** > **Close Other Folds (Ctrl+NumPad+/)** **Close Other Folds (Ctrl+NumPad+/)** - Folds all the elements except the current element.
- **Document** > **Folding** > **Collapse Child Folds (Ctrl+Decimal) (Ctrl+NumPad+-) ((Cmd+NumPad+- on Mac OS))** **Collapse Child Folds (Ctrl+Decimal)** - Folds the elements indented with one level inside the current element.
- **Document** > **Folding** > **Expand Child Folds (Ctrl+NumPad++) ((Cmd+NumPad++))** **Expand Child Folds (Ctrl+Equals)**- Unfolds all child elements of the currently selected element.
- **Document** > **Folding** > **Expand All (Ctrl+NumPad+*) ((Cmd+NumPad+* on Mac OS))** **Expand All (Ctrl+NumPad+*)** - Unfolds all elements in the current document.
- **Document** > **Folding** > **Toggle Fold (Alt+Shift+Y) ((Cmd+Alt+Y on Mac OS))** **Toggle Fold** - Toggles the state of the current fold.

To define the element whose content can be folded by the user, you must use the property: `-oxy-foldable:true;`. To define the elements that are folded by default, use the `-oxy-folded:true` property.

Note: The `-oxy-folded` property works in conjunction with the `-oxy-foldable` property. Thus, the `folded` property is ignored if the `-oxy-foldable` property is not set on the same element.

When collapsing an element, it is useful to keep some of its content visible, like a short description of the collapsed region. The property `-oxy-not-foldable-child` is used to identify the child elements that are kept visible. It accepts as value an element name or a list of comma separated element names. If the element is marked as *foldable* (`-oxy-foldable:true;`) but it doesn't have the property `-oxy-not-foldable-child` or none of the specified non-foldable children exists, then the element is still foldable. In this case the element kept visible when folded will be the `before` pseudo-element.

Note: Deprecated properties `foldable`, `not-foldable-child`, and `folded` are also supported.

Folding DocBook Elements

All the elements below can have a `title` child element and are considered to be logical sections. You mark them as being *foldable* leaving the `title` element visible.

```
set,
book,
part,
reference,
chapter,
preface,
article,
sect1,
sect2,
sect3,
sect4,
section,
appendix,
figure,
example,
table {
    -oxy-foldable:true;
    -oxy-not-foldable-child: title;
}
```

Placeholders for empty elements: `-oxy-show-placeholder` and `-oxy-placeholder-content` properties

Author displays the element name as pseudo-content for empty elements, if the *http://www.oxygenxml.com/doc/ug-editor/topics/preferences-author.html***Show placeholders for empty elements** option is enabled and there is no before or after content set is CSS for this type of element.

To control the displayed pseudo-content for empty elements, you can use the `-oxy-placeholder-content` CSS property.

The `-oxy-show-placeholder` property allows you to decide whether the placeholder must be shown. The possible values are:

- `always` - Always display placeholders.
- `default` - Always display placeholders if before or after content are not set is CSS.
- `inherit` - The placeholders are displayed according to **Show placeholders for empty elements** option (if `before` and `after` content is not declared).

Note: Deprecated properties `show-placeholder` and `placeholder-content` are also supported.

Read-only elements: `-oxy-editable` property

If you want to inhibit editing a certain element content, you can set the `-oxy-editable` (deprecated property `editable` is also supported) CSS property to `false`.

Display Elements: `-oxy-morph` value

Author allows you to specify that an element has a `-oxy-morph` display type (deprecated `morph` property is also supported), meaning that the element is inline if all its children are inline.

The `whitespace` property: `-oxy-trim-when-ws-only` value

Author allows you to set the `whitespace` property to `-oxy-trim-when-ws-only`, meaning that the leading and trailing whitespaces are removed.

The `visibility` property: -oxy-collapse-text

Author allows you to set the value of the `visibility` property to `-oxy-collapse-text`, meaning that the text content of that element is not rendered.

The `list-style-type` Cyrillic values

Author allows you to set the value of the `list-style-type` property to `-oxy-lower-cyrillic-ru`, `-oxy-lower-cyrillic-uk`, `-oxy-upper-cyrillic-ru` or `-oxy-upper-cyrillic-uk`, meaning that you can have Russian and Ukrainian counters.

Link Elements

Author allows you to declare some elements to be *links*. This is especially useful when working with many documents which refer each other. The links allow for an easy way to get from one document to another. Clicking on the link marker will open the referred resource in an editor.

To define the element which should be considered a link, you must use the property `link` on the before or after pseudo element. The value of the property indicates the location of the linked resource. Since links are usually indicated by the value of an attribute in most cases it will have a value similar to `attr(href)`

Docbook Link Elements

All the elements below are defined to be links on the before pseudo element and their value is defined by the value of an attribute.

```
*[href]:before{
    link:attr(href);
    content: "Click " attr(href) " for opening" ;
}

ulink[url]:before{
    link:attr(url);
    content: "Click to open: " attr(url);
}

olink[targetdoc]:before{
    link: attr(targetdoc);
    content: "Click to open: " attr(targetdoc);
}
```

Display Tag Markers

Author allows you to choose whether tag markers of an element should never be presented or the current display mode should be respected. This is especially useful when working with `:before` and `:after` pseudo-elements in which case the element range is already visually defined so the tag markers are redundant.

The property is named `-oxy-display-tags`, with the following possible values:

- *none* - Tags markers must not be presented regardless of the current Display mode.
- *default* - The tag markers will be created depending on the current Display mode.
- *inherit* - The value of the property is inherited from an ancestor element.

```
-oxy-display-tags
    Value: none | default | inherit
    Initial: default
    Applies to: all nodes(comments, elements, CDATA, etc)
    Inherited: false
    Media: all
```

Docbook Para elements

In this example the **para** element from Docbook is using an `:before` and `:after` element so you don't want its tag markers to be visible.

```
para:before{
    content: "{";
}

para:after{
    content: "}";
}

para{
    -oxy-display-tags: none;
    display:block;
    margin: 0.5em 0;
}
```

Custom CSS Functions

The visual Author editing mode supports also a wide range of custom CSS extension functions.

The `oxy_local-name()` Function

This function evaluates the local name of the current node. It does not have any arguments.

The `oxy_name()` Function

This function evaluates the qualified name of the current node. It does not have any arguments.

The `oxy_url()` function

This function extends the standard CSS **url()** function, by allowing you to specify additional relative path components (parameters **loc_1** to **loc_n**). uses all these parameters to construct an absolute location.

```
oxy_url(  location  , loc_1 , loc_2 )
```

location	The location as string. If not absolute, will be solved relative to the CSS file URL.
loc_1 ... loc_n	Relative location path components as string. (optional)

The following function:

```
oxy_url('http://www.oxygenxml.com/css/test.css', '../dir1/', 'dir2/dir3/',
'../../dir4/dir5/test.xml')
```

returns

```
'http://www.oxygenxml.com/dir1/dir4/dir5/test.xml'
```

The `oxy_base-uri()` Function

This function evaluates the base URL in the context of the current node. It does not have any arguments and takes into account the `xml:base` context of the current node. See the *XML Base specification* for more details.

The `oxy_parent-url()` Function

This function evaluates the parent URL of an URL received as string.

```
oxy_parent-url ( URL )
```

URL	The URL as string.

The `oxy_capitalize()` Function

This function capitalizes the first letter of the text received as argument.

```
oxy_capitalize (  text  )
```

text	The text for which the first letter will be capitalized.

The `oxy_uppercase()` Function

This function transforms to upper case the text received as argument.

```
oxy_uppercase (  text  )
```

text	The text to be capitalized.

The `oxy_lowercase()` Function

This function transforms to lower case the text received as argument.

```
oxy_lowercase (  text  )
```

text	The text to be lower cased.

The `oxy_concat()` Function

This function concatenates the received string arguments.

```
oxy_concat (  str_1  ,  str_2  )
```

str_1 ... str_n	The string arguments to be concatenated.

The `oxy_replace()` Function

This function has two signatures:

- `oxy_replace ( text , target , replacement )`

  This function replaces each substring of the text that matches the literal target string with the specified literal replacement string.

text	The text in which the replace will occur.
target	The target string to be replaced.
replacement	The string replacement.

- `oxy_replace ( text , target , replacement , isRegExp )`

  This function replaces each substring of the text that matches the target string with the specified replacement string.

text	The text in which the replace will occur.
target	The target string to be replaced.
replacement	The string replacement.
isRegExp	If *true* the target and replacement arguments are considered regular expressions in PERL syntax, if *false* they are considered literal strings.

The `oxy_unparsed-entity-uri()` Function

This function returns the URI value of an unparsed entity name.

`oxy_unparsed-entity-uri ( unparsedEntityName )`

unparsedEntityName	The name of an unparsed entity defined in the DTD.

This function can be useful to display images which are referred with unparsed entity names.

CSS for displaying the image in Author for an *`imagedata`* with *`entityref`* to an unparsed entity

```
imagedata[entityref]{
content: oxy_url(oxy_unparsed-entity-uri(attr(entityref)));
}
```

The `oxy_attributes()` Function

This function concatenates the attributes for an element and returns the serialization.

`oxy_attributes ( )`

oxy_attributes()

For the following XML fragment:`<element att1="x" xmlns:a="2" x="&quot;"/>` the `oxy_attributes()` function will return `att1="x" xmlns:a="2" x="""`.

The `oxy_substring()` Function

This function has two signatures:

- `oxy_substring ( text , startOffset )`

  Returns a new string that is a substring of the original **text** string. It begins with the character at the specified index and extends to the end of **text** string.

text	The original string.
startOffset	The beginning index, inclusive

- `substring ( text , startOffset , endOffset )`

  Returns a new string that is a substring of the original **text** string. The substring begins at the specified **startOffset** and extends to the character at index **endOffset** - 1.

text	The original string.
startOffset	The beginning index, inclusive
endOffset	The ending index, exclusive.

  `oxy_substring('abcd', 1)` returns the string `'bcd'`.

  `oxy_substring('abcd', 4)` returns an empty string.

  `oxy_substring('abcd', 1, 3)` returns the string `'bc'`.

The `oxy_indexof()` Function

This function has two signatures:

- `oxy_indexof ( text , toFind )`

  Returns the index within **text** string of the first occurrence of the **toFind** substring.

text	Text to search in.
toFind	The searched substring.

- `oxy_indexof ( text , toFind , fromOffset )`

  Returns the index within **text** string of the first occurrence of the **toFind** substring. The search starts from **fromOffset** index.

text	Text to search in.
toFind	The searched substring.
fromOffset	The index from which to start the search.

  `oxy_indexof('abcd', 'bc')` returns 1.

  `oxy_indexof('abcdbc', 'bc', 2)` returns 4.

The `oxy_lastindexof()` Function

This function has two signatures:

- `oxy_lastindexof ( text , toFind )`

  Returns the index within **text** string of the rightmost occurrence of the **toFind** substring.

text	Text to search in.
toFind	The searched substring.

- `oxy_lastindexof ( text , toFind , fromOffset )`

  The search starts from **fromOffset** index. Returns the index within **text** string of the last occurrence of the **toFind** substring, searching backwards starting from the **fromOffset** index.

text	Text to search in.
toFind	The searched substring.
fromOffset	The index from which to start the search backwards.

`oxy_lastindexof('abcdbc', 'bc')` returns 4.

`oxy_lastindexof('abcdbccdbc', 'bc', 2)` returns 1.

The `oxy_xpath()` Function

This function has one signature:

- `oxy_xpath ( expression )`

  Evaluates the given XPath expression and returns the result.

  Note: The entities are ignored when the XPath expressions are evaluated.

  Note: The evaluation is performed using Saxon 9.

  expression An XPath 2.0 expression to be evaluated.

The following example counts the number of words from a paragraph and displays the result in front of it:

```
para:before{
  content:
    concat("|Number of words:",
           oxy_xpath(
                "count(tokenize(normalize-space(string-join(text(), '')), ' '))"
           ), "| ");
}
```

The oxy_editor() Function

The `oxy_editor` function allows you to edit attributes or simple element values in the Author mode using standard form controls like combo boxes, text fields, pop-ups, URL choosers or to implement your own custom form controls and renderers.

The oxy_editor() function can appear in the `content` property of a CSS rule. The function's arguments are *property name* - *property value* pairs:

```
myElement {
    content: oxy_editor(
        type, combo,
        edit, "@my_attr"
    )
}
```

The form control allows you to either edit the value of an attribute or the text content of an element. This is specified using the `edit` property. This property can have the following values:

- **@attribute_name** - specifies that the presented/edited value is the value of an attribute;
- **#text** - specifies that the presented/edited value is the simple text value of an element.

  Note: You can set the value of the `visibility` property to `-oxy-collapse-text` to render the text only in the form control that the oxy_editor function specifies.

You can use a number of built-in form controls through the `type` property. The following values are recognized:

- ***text*** - a text field with optional content completion capabilities is used to present and edit a value;
- ***combo*** - a combo-box is used to present and edit a value;
- ***check*** - a single check box or multiple check boxes are used to present and edit a value;
- ***popupSelection*** - a pop-up with single/multiple selection is used as form control;
- ***button*** - a button that invokes an author action is used as form control;
- ***urlChooser*** - a text field with a browse button is used as form control.

- ***datePicker*** - a text field with a calendar browser button is used as form control.

In case the built-in form controls are not enough, you can implement custom form controls in Java and specify them using the following properties:

- **rendererClassName** - the name of the class that draws the edited value. It must be an implementation of `ro.sync.ecss.extensions.api.editor.InplaceRenderer`. The renderer has to be a SWING implementation and can be used both in the standalone and Eclipse distributions;
- **swingEditorClassName** - you can use this property for the standalone (**Swing**-based) distribution to specify the name of the class used for editing. It is a **Swing** implementation of `ro.sync.ecss.extensions.api.editor.InplaceEditor`;
- **swtEditorClassName** - you can use this property for the Eclipse plug-in distribution to specify the name of the class used for editing. It is a **SWT** implementation of the `ro.sync.ecss.extensions.api.editor.InplaceEditor`.

If the custom form control is intended to work in the standalone distribution, the declaration of **swtEditorClassName** is not required. The *renderer* (the class that draws the value) and the *editor* (the class that edits the value) have different properties because you can present a value in one way and edit it in another way.

The custom form controls can use any of the predefined properties of the `oxy_editor` function, as well as specified custom properties. This is an example of how to specify a custom form control:

```
myElement {
    content: oxy_editor(
        rendererClassName, "com.custom.editors.CustomRenderer",
        swingEditorClassName, "com.custom.editors.SwingCustomEditor",
        swtEditorClassName, "com.custom.editors.SwtCustomEditor",
        edit, "@my_attr"
        customProperty1, "customValue1",
        customProperty2, "customValue2"
    )
}
```

Note: Add these custom **Java** implementations in the *classpath* of the document type associated with the document you are editing. To get you started the Java sources for the `SimpleURLChooserEditor` are available in the Author SDK.

The `oxy_editor` function can receive other functions as parameters for obtaining complex behaviors.

The following example shows how the combo box editor can obtain its values from the current XML file by calling the `oxy_xpath` function:

```
link:before{
      content: "Managed by:"
        oxy_editor(
            type, combo,
            edit, "@manager",
            values, oxy_xpath('string-join(//@id , ",") '));
```

The Text Field Form Control

A text field with optional content completion capabilities is used to present and edit the value of an attribute or an element. This type of form control supports the following properties:

- **type** - this property specifies the built-in form control you are using. For the Text form control, its value has to be `text`;
- **columns** - controls the width of the form control. The unit size is the width of the **w** character;
- **fontInherit** - this value specifies whether the form control inherits its font from its parent element. The values of this property can be **true**, or **false**. To make the `pop-up` form control inherit its font from its parent element, set the **fontInherit** property to **true**;
- **values** - specifies the values that populate the content completion list of proposals. In case these values are not specified, they are collected from the associated schema;
- **tooltips** - specifies a tooltip for the form control. The values of this property are a list of tooltips separated by commas. In case you want the tooltip to display a comma, use the `${comma}` variable;

- **color** - specifies the foreground color of the form control. In case the value of the `color` property is `inherit`, the form control has the same color as the element in which it is inserted.

Text Field Form Control

```
element {
    content: "Label: "
        oxy_editor(
            type, text,
            edit, "@my_attr",
            values, "value1, value2"
            columns, 40);
}
```

The `oxy_editor` function acts as a proxy that allows you to insert any of the supported form controls. Alternatively, you can use the `oxy_textfield` dedicated function.

```
element {
    content: "Label: "
        oxy_textfield(
            edit, "@my_attr",
            values, "value1, value2"
            columns, 40);
}
```

The Combo Box Form Control

A combo box is used to present and edit the value of an attribute or an element. This type of form control supports the following properties:

- **type** - this property specifies the built-in form control you are using. For the Combo box form control, its value has to be `combo`;
- **columns** - controls the width of the form control. The unit size is the width of the **w** character;
- **editable** - this property accepts the **true** and **false** values. The **true** value generates an editable combo-box that allows you to insert other values than the proposed ones. The **false** value generates a combo-box that only accepts the proposed values;
- **tooltips** - specifies a tooltip for the form control. The values of this property are a list of tooltips separated by commas. In case you want the tooltip to display a comma, use the `${comma}` variable;
- **values** - specifies the values that populate the content completion list of proposals. In case these values are not specified, they are collected from the associated schema;
- **fontInherit** - this value specifies whether the form control inherits its font from its parent element. The values of this property can be **true**, or **false**. To make the `pop-up` form control inherit its font from its parent element, set the **fontInherit** property to **true**;
- **labels** - this property must have the same number of items as the **values** property. Each item provides a literal description of the items listed in the **values** property;
- **color** - specifies the foreground color of the form control. In case the value of the `color` property is `inherit`, the form control has the same color as the element in which it is inserted.

Combo Box Form Control

```
comboBox:before {
    content: "A combo box that edits an attribute value. The possible values are provided from
 CSS:"
        oxy_editor(
            type, combo,
            edit, "@attribute",
            editable, true,
            values, "value1, value2, value3",
            labels, "Value no1, Value no2, Value no3");
}
```

The `oxy_editor` function acts as a proxy that allows you to insert any of the supported form controls. Alternatively, you can use the `oxy_combobox` dedicated function.

```
comboBox:before {
    content: "A combo box that edits an attribute value. The possible values are provided from
```

```
 CSS:"
        oxy_combobox(
            edit, "@attribute",
            editable, true,
            values, "value1, value2, value3",
            labels, "Value no1, Value no2, Value no3");
}
```

The Check Box Form Control

A single check-box or multiple check-boxes are used to present and edit the value on an attribute or element. This type of form control supports the following properties:

- **type** - this property specifies the built-in form control you are using. For the Check Box form control, its value has to be `check`;
- **resultSeparator** - in case multiple check-boxes are used, the separator is used to compose the final result;
- **tooltips** - specifies a tooltip for the form control. The values of this property are a list of tooltips separated by commas. In case you want the tooltip to display a comma, use the `${comma}` variable;
- **values** - specifies the values that are committed when the check-boxes are selected;
- **fontInherit** - this value specifies whether the form control inherits its font from its parent element. The values of this property can be **true**, or **false**. To make the `pop-up` form control inherit its font from its parent element, set the **fontInherit** property to **true**;
- **uncheckedValues** - specifies the values that are committed when the check-boxes are not selected;
- **labels** - specifies the label associated with each button. In case this property is not specified, the **values** property is used as label;
- **columns** - specifies the number of columns of the layout;
- **color** - specifies the foreground color of the form control. In case the value of the `color` property is `inherit`, the form control has the same color as the element in which it is inserted.

Single Check-box Form Control

```
checkBox[attribute]:before {
    content: "A check box editor that edits a two valued attribute (On/Off).
              The values are specified in the CSS:"
oxy_editor(
    type, check,
        edit, "@attribute",
        values, "On",
        uncheckedValues, "Off",
        labels, "On/Off");
```

Multiple Check-boxes Form Control

```
multipleCheckBox[attribute]:before {
    content: "Multiple checkboxes editor that edits an attribute value.
              Depending whether the check-box is selected a different value is committed:"
        oxy_editor(
        type, check,
        edit, "@attribute",
        values, "true, yes, on",
        uncheckedValues, "false, no, off",
        resultSeparator, ","
        labels, "Present, Working, Started");
```

The `oxy_editor` function acts as a proxy that allows you to insert any of the supported form controls. Alternatively, you can use the `oxy_checkbox` dedicated function.

```
multipleCheckBox[attribute]:before {
    content: "Multiple checkboxes editor that edits an attribute value.
              Depending whether the check-box is selected a different value is committed:"
        oxy_checkbox(
        edit, "@attribute",
        values, "true, yes, on",
        uncheckedValues, "false, no, off",
        resultSeparator, ","
        labels, "Present, Working, Started");
```

The Pop-up Form Control

A pop-up with single or multiple selection is used as a form control. This type of form control supports the following properties:

- **type** - this property specifies the built-in form control you are using. For the Pop-up form control, its value has to be `popupSelection`;
- **rows** - this property specifies the number of rows that the form control presents;

  👉 **Note:** In case the value of the **rows** property is not specifies, the default value of *12* is used.

- **color** - specifies the foreground color of the form control. In case the value of the `color` property is `inherit`, the form control has the same color as the element in which it is inserted;
- **tooltips** - specifies a tooltip for the form control. The values of this property are a list of tooltips separated by commas. In case you want the tooltip to display a comma, use the `${comma}` variable;
- **values** - specifies the values that are committed when the check-boxes are selected;
- **resultSeparator** - if multiple values are allowed, the separator is used to compose the final result;

  👉 **Note:** The value of the **resultSeparator** property cannot exceed one character.

- **selectionMode** - specifies whether the form control allows the selection of a single value or of multiple values. The predefined values of this property are **single** and **multiple**;
- **labels** - specifies the label associated with each entry used for presentation. In case this property is not specified, the **values** property is used as label;
- **columns** - controls the width of the form control. The unit size is the width of the **w** character. This property is used for the visual representation of the form control;
- **rendererSort** - allows you to sort the values rendered in a pop-up form control;
- **rendererSeparator** - defines a separator used when multiple values are rendered;
- **fontInherit** - this value specifies whether the form control inherits its font from its parent element. The values of this property can be **true**, or **false**. To make the `pop-up` form control inherit its font from its parent element, set the **fontInherit** property to **true**.

  👉 **Tip:** In the below example, the value of the **fontInherit** property is **true**, meaning that the pop-up form control inherits the font size of `30px` from the **font-size** property.

Pop-up Form Control

```
popupWithMultipleSelection:before {
    content: " This editor edits an attribute value. The possible values are     specified
    inside the CSS: "
        oxy_editor(
            type, popupSelection,
            edit, "@attribute",
            values, "value1, value2, value3, value4, value5",
            labels, "Value no1, Value no2, Value no3, Value no4, Value no5",
            resultSeparator, "|",
            columns, 10,
            selectionMode, "multiple",
            fontInherit, true);
    font-size:30px;
}
```

The `oxy_editor` function acts as a proxy that allows you to insert any of the supported form controls. Alternatively, you can use the `oxy_popup` dedicated function.

```
popupWithMultipleSelection:before {
    content: " This editor edits an attribute value. The possible values are     specified
    inside the CSS: "
        oxy_popup(
            edit, "@attribute",
            values, "value1, value2, value3, value4, value5",
            labels, "Value no1, Value no2, Value no3, Value no4, Value no5",
            resultSeparator, "|",
            columns, 10,
```

```
            selectionMode, "multiple",
            fontInherit, true);
    font-size:30px;
}
```

The Button Form Control

This form control contributes a button that invokes a *custom Author action* (defined in the associated Document Type) using its defined ID. The following properties are supported:

- **type** - this property specifies the built-in form control you are using. For the Button form control the value of the **type** property is `button`;
- **fontInherit** - this value specifies whether the form control inherits its font from its parent element. The values of this property can be **true**, or **false**. To make the `pop-up` form control inherit its font from its parent element, set the **fontInherit** property to **true**;
- **color** - specifies the foreground color of the form control. In case the value of the `color` property is `inherit`, the form control has the same color as the element in which it is inserted;
- **actionID** - the ID of the action specified in *Author actions*, that is invoked when you click the button;

Note: The element that contains the `Button` form control represents the context where the action is invoked.

- **transparent** - flattens the aspect of the button form control, removing its border and background.

Button Form Control

```
button:before {
    content: "Label:"
        oxy_editor(
            type, button,
            /* This action is declared in the document type associated with the XML document.
 */
            actionID, "insert.popupWithMultipleSelection");
}
```

The `oxy_editor` function acts as a proxy that allows you to insert any of the supported form controls. Alternatively, you can use the `oxy_button` dedicated function.

```
button:before {
    content: "Label:"
        oxy_button(
            /* This action is declared in the document type associated with the XML document.
 */
            actionID, "insert.popupWithMultipleSelection");
}
```

The URL Chooser Form Control

A field that allows you to select local and remote resources is used as a form control. The inserted reference will be made relative to the current opened editor's URL. This type of editor supports the following properties:

- **type** - this property specifies the built-in editor you are using. For the URL Chooser editor, its value has to be `urlChooser`;
- **columns** - controls the width of the form control. The unit size is the width of the **w** character;
- **color** - specifies the foreground color of the form control. In case the value of the `color` property is `inherit`, the form control has the same color as the element in which it is inserted;
- **fontInherit** - this value specifies whether the form control inherits its font from its parent element. The values of this property can be **true**, or **false**. To make the `pop-up` form control inherit its font from its parent element, set the **fontInherit** property to **true**.

URL Chooser Form Control

```
urlChooser[file]:before {
    content: "An URL chooser editor that allows browsing for a URL. The selected URL is made
relative to the currently edited file:"
```

```
        oxy_editor(
        type, urlChooser,
        edit, "@file",
        columns 25);
}
```

The `oxy_editor` function acts as a proxy that allows you to insert any of the supported form controls. Alternatively, you can use the `oxy_urlChooser` dedicated function.

```
urlChooser[file]:before {
    content: "An URL chooser editor that allows browsing for a URL. The selected URL is made
relative to the currently edited file:"
        oxy_urlChooser(
        edit, "@file",
        columns 25);
}
```

The Date Picker Form Control

A text field with a calendar browser is used as a form control. The browse button shows a date chooser allowing you to easily choose a certain date. This type of form control supports the following properties:

- **type** - this property specifies the built-in editor you are using. For the date picker form control, its value has to be `datePicker`;
- **columns** - controls the width of the form control. The unit size is the width of the **w** character;
- **color** - specifies the foreground color of the form control. In case the value of the `color` property is `inherit`, the form control has the same color as the element in which it is inserted;
- **format** - this property specifies the format of the inserted date. The pattern value must be a valid Java date (or date-time) format. If missing, the type of the date is determined from the associated schema.

Date Picker Form Control

```
date {
    content:
        oxy_label(text, "Date time attribute with format defined in CSS: ", width, 300px)
        oxy_editor(
            type, datePicker,
            columns, 16,
            edit, "@attribute",
            format, "yyyy-MM-dd");
}
```

The `oxy_editor` function acts as a proxy that allows you to insert any of the supported form controls. Alternatively, you can use the `oxy_datePicker` dedicated function.

```
date {
    content:
        oxy_label(text, "Date time attribute with format defined in CSS: ", width, 300px)
        oxy_datePicker(
            columns, 16,
            edit, "@attribute",
            format, "yyyy-MM-dd");
}
```

The oxy_label() Function

This function can be used in conjunction with the CSS `content` property to change the style of generated text. The arguments of the function are *property name - property value* pairs. The following properties are supported:

- **text** - specifies the generated content;
- **width** - specifies the horizontal space reserved for the content. The value of this property has the same format as the value of the CSS `width` property;
- **color** - specifies the foreground color of the form control. In case the value of the `color` property is `inherit`, the form control has the same color as the element in which it is inserted;
- **background-color** - specifies the background color of the form control. In case the value of the `background-color` property is `inherit`, the form control has the same color as the element in which it is inserted;

- **text-align** - specifies the alignment of the generated content in its reserved space. This property has three possible values:
  - **left**;
  - **right**;
  - **center**.

```
element {
  content:
    oxy_label(label, "Label1:", width, 20em, text-align, center)
    oxy_label(label, "Label2:", width, 100px, text-align, right)
}
```

You can use the *oxy_editor()* and oxy_label() functions together to create a form control based layout.

The oxy_link-text() Function

You can use the oxy_link-text() function on the CSS `content` property to obtain a text description from the source of a reference. By default, the oxy_link-text() function resolves DITA and DocBook references. For further details about how you can also extend this functionality to other frameworks, go to *Configuring an Extensions Bundle*.

DITA Support

For DITA, the oxy_link-text() function resolves the `xref` element and the elements that have a `keyref` attribute. The text description is the same as the one presented in the final output for those elements.

DocBook Support

For DocBook, the oxy_link-text() function resolves the `xref` element that defines a link in the same document. The text description is the same as the one presented in the final output for those elements.

For the following XML and associated CSS fragments the `oxy_link-text()` function is resolved to the value of the `xreflabel` attribute.

```
<para><code id="para.id" xreflabel="The reference label">my code</code></para>
<para><xref linkend="para.id"/></para>
```

```
xref {
    content: oxy_link-text();
}
```

Arithmetic Functions

You can use any of the arithmetic functions implemented in the `java.lang.Math` class: *http://download.oracle.com/javase/6/docs/api/java/lang/Math.html*.

In addition to that, the following functions are available:

Syntax	Details
oxy_add(`param1`, ... , `paramN`, `'returnType'`)	Adds the values of all parameters from `param1` to `paramN`.
oxy_subtract(`param1`, ..., `paramN`, `'returnType'`)	Subtracts the values of parameters `param2` to `paramN` from `param1`.
oxy_multiply(`param1`, ..., `paramN`, `'returnType'`)	Multiplies the values of parameters from `param1` to `paramN`.
oxy_divide(`param1`, `param2`, `'returnType'`)	Performs the division of `param1` to `param2`.
oxy_modulo(`param1`, `param2`, `'returnType'`)	Returns the reminder of the division of `param1` to `param2`.

Note: The `returnType` can be `'integer'`, `'number'`, or any of the supported CSS measuring types.

Example Files Listings - The Simple Documentation Framework Files

This section lists the files used in the customization tutorials: the XML Schema, CSS files, XML files, XSLT stylesheets.

XML Schema files

`sdf.xsd`

This sample file can also be found in the *Author SDK distribution* in the `"oxygenAuthorSDK\samples\Simple Documentation Framework - SDF\framework\schema"` directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.oxygenxml.com/sample/documentation"
    xmlns:doc="http://www.oxygenxml.com/sample/documentation"
    xmlns:abs="http://www.oxygenxml.com/sample/documentation/abstracts"
    elementFormDefault="qualified">

    <xs:import
        namespace="http://www.oxygenxml.com/sample/documentation/abstracts"
        schemaLocation="abs.xsd"/>

    <xs:element name="book" type="doc:sectionType"/>
    <xs:element name="article" type="doc:sectionType"/>
    <xs:element name="section" type="doc:sectionType"/>

    <xs:complexType name="sectionType">
        <xs:sequence>
            <xs:element name="title" type="xs:string"/>
            <xs:element ref="abs:def" minOccurs="0"/>
            <xs:choice>
                <xs:sequence>
                    <xs:element ref="doc:section"
                        maxOccurs="unbounded"/>
                </xs:sequence>
                <xs:choice maxOccurs="unbounded">
                    <xs:element ref="doc:para"/>
                    <xs:element ref="doc:ref"/>
                    <xs:element ref="doc:image"/>
                    <xs:element ref="doc:table"/>
                </xs:choice>
            </xs:choice>
        </xs:sequence>
    </xs:complexType>

    <xs:element name="para" type="doc:paragraphType"/>

    <xs:complexType name="paragraphType" mixed="true">
        <xs:choice minOccurs="0" maxOccurs="unbounded">
            <xs:element name="b"/>
            <xs:element name="i"/>
            <xs:element name="link"/>
        </xs:choice>
    </xs:complexType>

    <xs:element name="ref">
        <xs:complexType>
            <xs:attribute name="location" type="xs:anyURI"
                use="required"/>
        </xs:complexType>
    </xs:element>

    <xs:element name="image">
        <xs:complexType>
            <xs:attribute name="href" type="xs:anyURI"
                use="required"/>
        </xs:complexType>
    </xs:element>

    <xs:element name="table">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="customcol" maxOccurs="unbounded">
                    <xs:complexType>
                        <xs:attribute name="width" type="xs:string"/>
                    </xs:complexType>
                </xs:element>
                <xs:element name="header">
```

```
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="td"
                                maxOccurs="unbounded"
                                type="doc:paragraphType"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="tr" maxOccurs="unbounded">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="td"
                                type="doc:tdType"
                                maxOccurs="unbounded"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
            <xs:attribute name="width" type="xs:string"/>
        </xs:complexType>
    </xs:element>


    <xs:complexType name="tdType">
        <xs:complexContent>
            <xs:extension base="doc:paragraphType">
                <xs:attribute name="row_span"
                    type="xs:integer"/>
                <xs:attribute name="column_span"
                    type="xs:integer"/>
            </xs:extension>
        </xs:complexContent>
    </xs:complexType>
</xs:schema>
```

`abs.xsd`

This sample file can also be found in the *Author SDK distribution* in the `"oxygenAuthorSDK\samples\Simple Documentation Framework - SDF\framework\schema"` directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace=
 "http://www.oxygenxml.com/sample/documentation/abstracts">
    <xs:element name="def" type="xs:string"/>
</xs:schema>
```

CSS Files

`sdf.css`

This sample file can also be found in the *Author SDK distribution* in the `oxygenAuthorSDK\samples\Simple Documentation Framework - SDF\framework\css` directory.

```
/* Element from another namespace */
@namespace abs "http://www.oxygenxml.com/sample/documentation/abstracts";

abs|def{
    font-family:monospace;
    font-size:smaller;
}
abs|def:before{
    content:"Definition:";
    color:gray;
}

/* Vertical flow */
book,
section,
para,
title,
image,
ref {
    display:block;
}

/* Horizontal flow */
b,i {
    display:inline;
}
```

```
section{
    margin-left:1em;
    margin-top:1em;
}

section{
    -oxy-foldable:true;
    -oxy-not-foldable-child: title;
}

link[href]:before{
    display:inline;
    link:attr(href);
    content: "Click to open: " attr(href);
}

/* Title rendering*/
title{
    font-size: 2.4em;
    font-weight:bold;
}

* * title{
    font-size: 2.0em;
}
* * * title{
    font-size: 1.6em;
}
* * * * title{
    font-size: 1.2em;
}

book,
article{
    counter-reset:sect;
}
book > section,
article > section{
    counter-increment:sect;
}
book > section > title:before,
article > section > title:before{
    content: "Section: " counter(sect) " ";
}

/* Inlines rendering*/
b {
    font-weight:bold;
}

i {
    font-style:italic;
}

/*Table rendering */
table{
    display:table;
    border:1px solid navy;
    margin:1em;
    max-width:1000px;
    min-width:150px;
}

table[width]{
  width:attr(width, length);
}

tr, header{
    display:table-row;
}

header{
    background-color: silver;
    color:inherit
}

td{
  display:table-cell;
  border:1px solid navy;
  padding:1em;
}

image{
    display:block;
    content: attr(href, url);
    margin-left:2em;
}
```

XML Files

`sdf_sample.xml`

This sample file can also be found in the *Author SDK distribution* in the `"oxygenAuthorSDK\samples\Simple Documentation Framework - SDF\framework"` directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<book xmlns="http://www.oxygenxml.com/sample/documentation"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:abs="http://www.oxygenxml.com/sample/documentation/abstracts">
    <title>My Technical Book</title>
    <section>
        <title>XML</title>
        <abs:def>Extensible Markup Language</abs:def>
        <para>In this section of the book I will explain
            different XML applications.</para>
    </section>
    <section>
        <title>Accessing XML data.</title>
        <section>
            <title>XSLT</title>
            <abs:def>Extensible stylesheet language
                transformation (XSLT) is a language for
                transforming XML documents into other XML
                documents.</abs:def>
            <para>A list of XSL elements and what they do..</para>
            <table>
                <header>
                    <td>XSLT Elements</td>
                    <td>Description</td>
                </header>
                <tr>
                    <td>
                        <b>xsl:stylesheet</b>
                    </td>
                    <td>The <i>xsl:stylesheet</i> element is
                        always the top-level element of an
                        XSL stylesheet. The name
                            <i>xsl:transform</i> may be used
                        as a synonym.</td>
                </tr>
                <tr>
                    <td>
                        <b>xsl:template</b>
                    </td>
                    <td>The <i>xsl:template</i> element has
                        an optional mode attribute. If this
                        is present, the template will only
                        be matched when the same mode is
                        used in the invoking
                            <i>xsl:apply-templates</i>
                        element.</td>
                </tr>
                <tr>
                    <td>
                        <b>for-each</b>
                    </td>
                    <td>The xsl:for-each element causes
                        iteration over the nodes selected by
                        a node-set expression.</td>
                </tr>
                <tr>
                    <td column_span="2">End of the list</td>
                </tr>
            </table>
        </section>
        <section>
            <title>XPath</title>
            <abs:def>XPath (XML Path Language) is a terse
                (non-XML) syntax for addressing portions of
                an XML document. </abs:def>
            <para>Some of the XPath functions.</para>
            <table>
                <header>
                    <td>Function</td>
                    <td>Description</td>
                </header>
                <tr>
                    <td>format-number</td>
                    <td>The <i>format-number</i> function
                        converts its first argument to a
                        string using the format pattern
                        string specified by the second
                        argument and the decimal-format
                        named by the third argument, or the
                        default decimal-format, if there is
```

```
                        no third argument</td>
                </tr>
                <tr>
                    <td>current</td>
                    <td>The <i>current</i> function returns
                        a node-set that has the current node
                        as its only member.</td>
                </tr>
                <tr>
                    <td>generate-id</td>
                    <td>The <i>generate-id</i> function
                        returns a string that uniquely
                        identifies the node in the argument
                        node-set that is first in document
                        order.</td>
                </tr>
            </table>
        </section>
    </section>
    <section>
        <title>Documentation frameworks</title>
        <para>One of the most important documentation
            frameworks is Docbook.</para>
        <image
            href="http://www.xmlhack.com/images/docbook.png"/>
        <para>The other is the topic oriented DITA, promoted
            by OASIS.</para>
        <image
href="http://www.oasis-open.org/images/standards/oasis_standard.jpg"
        />
    </section>
</book>
```

XSL Files

`sdf.xsl`

This sample file can also be found in the *Author SDK distribution* in the `"oxygenAuthorSDK\samples\Simple Documentation Framework - SDF\framework\xsl"` directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="2.0"
    xpath-default-namespace=
    "http://www.oxygenxml.com/sample/documentation">

    <xsl:template match="/">
        <html><xsl:apply-templates/></html>
    </xsl:template>

    <xsl:template match="section">
        <xsl:apply-templates/>
    </xsl:template>

    <xsl:template match="image">
        <img src="{@href}"/>
    </xsl:template>

    <xsl:template match="para">
        <p>
            <xsl:apply-templates/>
        </p>
    </xsl:template>

    <xsl:template match="abs:def"
        xmlns:abs=
        "http://www.oxygenxml.com/sample/documentation/abstracts">
        <p>
            <u><xsl:apply-templates/></u>
        </p>
    </xsl:template>

    <xsl:template match="title">
        <h1><xsl:apply-templates/></h1>
    </xsl:template>

    <xsl:template match="b">
        <b><xsl:apply-templates/></b>
    </xsl:template>

    <xsl:template match="i">
        <i><xsl:apply-templates/></i>
    </xsl:template>
```

```
    <xsl:template match="table">
        <table frame="box" border="1px">
            <xsl:apply-templates/>
        </table>
    </xsl:template>

    <xsl:template match="header">
        <tr>
            <xsl:apply-templates/>
        </tr>
    </xsl:template>

    <xsl:template match="tr">
        <tr>
            <xsl:apply-templates/>
        </tr>
    </xsl:template>

    <xsl:template match="td">
        <td>
            <xsl:apply-templates/>
        </td>
    </xsl:template>

    <xsl:template match="header/header/td">
        <th>
            <xsl:apply-templates/>
        </th>
    </xsl:template>

</xsl:stylesheet>
```

Author Component

The Author Component was designed as a separate product to provide the functionality of the standard Author mode. Recently (in version 14.2), the component API was extended to also allow multiple edit modes like **Text** and **Grid**. The component can be embedded either in a third-party standalone Java application or customized as a Java Web Applet to provide WYSIWYG-like XML editing directly in your web browser of choice.

The Author Component Startup Project for Java/Swing integrations is available online on the website: *http://www.oxygenxml.com/demo/AuthorDemoApplet/author-component-sample.zip*

Licensing

The licensing terms and conditions for the Author Component are defined in the **<oXygen/> XML Editor SDK License Agreement**. You can contact our support team at **support@oxygenxml.com** to obtain it as well as more information about licensing. You may also obtain a free of charge evaluation license key for development purposes. Any development work using the Author Component is also subject to the terms of the SDK agreement.

There are two main categories of Author Component integrations:

1. Integrations for internal use.

   You develop an application which embeds the Author Component to be used internally (in your company or by you). You can buy and use *oXygen XML Author standard licenses* (either user-based or floating) to enable the Author Component in your application.

2. Integrations for external use.

   You create using the Author Component an application that you distribute to other users, outside your company (with a CMS for example). In this case you need to contact us to apply for a Value Added Reseller (VAR) partnership.

From a technical point of view the Author Component provides the Java API to:

- Inject floating license server details in the Java code. The following link provides details about how to configure a floating license servlet or server:*http://www.oxygenxml.com/license_server.html*.

```
AuthorComponentFactory.getInstance().init(frameworkZips, optionsZipURL, codeBase, appletID,
    //The servlet URL
    "http://www.host.com/servlet",
    //The HTTP credentials user name
```

```
    "userName",
    //The HTTP credentials password
    "password");
```

- Inject the licensing information key (for example the evaluation license key) directly in the component's Java code.

```
AuthorComponentFactory.getInstance().init(
   frameworkZips, optionsZipURL, codeBase, appletID,
   //The license key if it is a fixed license.
   licenseKey);
```

- Display the license registration dialog to the end user. This is the default behavior in case a `null` license key is set using the API, this transfers the licensing responsibility to the end-user. The user can license an Author component using standard Editor/Author license keys. The license key will be saved to the local user's disk and on subsequent runs the user will not be asked anymore.

```
AuthorComponentFactory.getInstance().init(
   frameworkZips, optionsZipURL, codeBase, appletID,
   //Null license key, will ask the user.
   null);
```

Installation Requirements

Running the Author component as a Java applet requires:

- Oracle (Sun) Java JRE version 1.6 update 10 or newer;
- At least 100 MB disk space and 100MB free memory;
- The applet needs to be signed with a valid certificate and will request full access to the user machine, in order to store customization data (like options and framework files);
- A table of supported browsers can be found here:*Supported browsers and operating systems* on page 103.

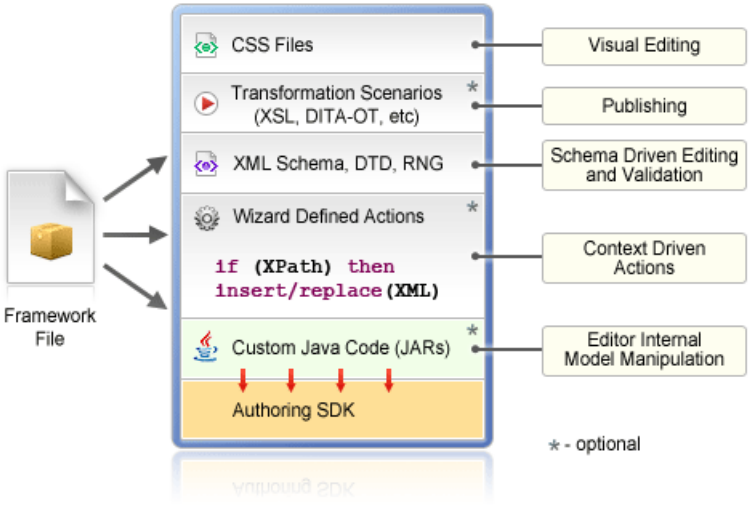
Running the Author component embedded in a third-party Java/Swing application requires:

- Oracle (Sun) Java JRE version 1.6 or newer;
- At least 100 MB disk space and 100MB free memory;

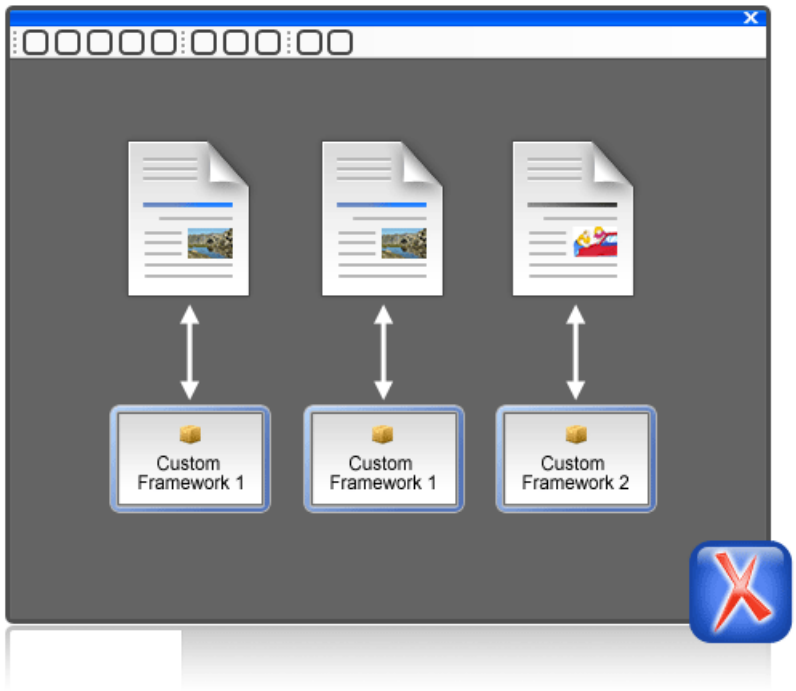
Customization

For a special type of XML, you can create a custom framework (which also works in an Oxygen standalone version). already has frameworks for editing DocBook, DITA, TEI, and so on. Their sources are available in *the Author SDK*. This custom framework is then packed in a zip archive and used to deploy the component.

The following diagram shows the components of a custom framework.



More than one framework can coexist in the same component and can be used at the same time for editing XML documents.



You can add on your custom toolbar all actions available in the standalone application for editing in the Author mode. You can also add custom actions defined in the framework customized for each XML type.

The Author component can also provide the *Outline*, *Model*, *Elements* and *Attributes* views which can be added to your own developed containers.

Packing a fixed set of options

The Author Component shares a common internal architecture with the standalone application although it does not have a **Preferences** dialog. But the Author Component Applet can be configured to use a fixed set of user options on startup.

The sample project contains a resource called `APPLET_PROJECT/resources/options.zip.jar`. The JAR contains a ZIP archive which contains a file called `options.xml`. Such an XML file can be obtained by exporting to an XML format from a standalone application.

To create an *options file* in the standalone application, follow these steps:

- make sure the options that you want to set are not stored at project level;
- set the values you want to impose as defaults in the *Preferences pages*;
- select **Options** > **Export Global Options**.

Deployment

The Author Component Java API allows you to use it in your Java application or as a Java applet. The JavaDoc for the API can be found in the *sample project* in the `lib/apiSrc.zip` archive. The sample project also comes with Java sources (`ro/sync/ecss/samples/AuthorComponentSample.java`) demonstrating how the component is created, licensed and used in a Java application.

Web Deployment

The Author Component can be deployed as a Java Applet using the new Applet with JNLP Java technology, available in Oracle (Sun) Java JRE version 1.6 update 10 or newer.

The *sample project* demonstrates how the Author component can be distributed as an applet.

Here are the main steps you need to follow in order to deploy the Author component as a Java Applet:

- Unpack the sample project archive and look for Java sources of the sample Applet implementation. They can be customized to fit your requirements.
- The `default.properties` configuration file must first be edited to specify your custom certificate information used to sign the applet libraries. You also have to specify the code base from where the applet will be downloaded.
- You can look inside the `author-component-dita.html` and `author-component-dita.js` sample Web resources to see how the applet is embedded in the page and how it can be controlled using Javascript (to set and get XML content from it).
- The sample Applet `author-component-dita.jnlp` JNLP file can be edited to add more libraries. The packed frameworks and options are delivered using the JNLP file as JAR archives:

```
<jar href="resources/frameworks.zip.jar"/>
<jar href="resources/options.zip.jar"/>
```

- The sample frameworks and options JAR archives can be found in the `resources` directory.
- Use the `build.xml` ANT build file to pack the component. The resulting applet distribution is copied in the `dist` directory. From this on, you can copy the applet files on your web server.

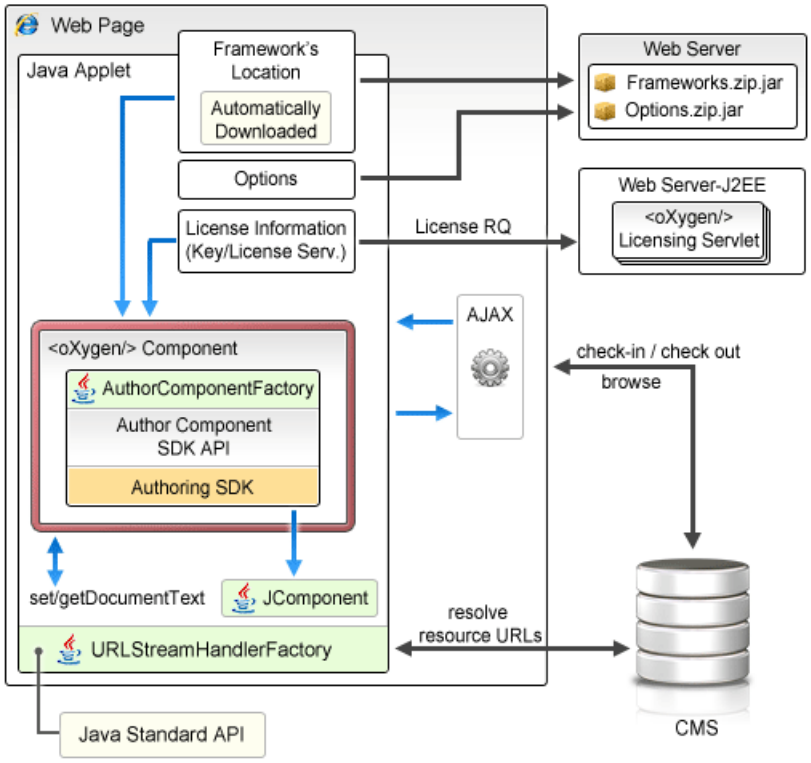


Figure 29: Author Component deployed as a Java applet

Generate a Testing Certificate For Signing an Applet

All jar files of an applet deployed on a remote Web server must be signed with the same certificate before the applet is deployed. The following steps describe how to generate a test certificate for signing the jar files. We will use the tool called **keytool** which is included in the Oracle's Java Development Kit.

1. Create a keystore with a RSA encryption key.

   Invoke the following in a command line terminal:

   ```
   keytool -genkey -alias myAlias -keystore keystore.pkcs -storetype PKCS12
   -keyalg RSA -keysize 2048 -dname "cn=your name here, ou=organization unit
   name,  o=organization name, c=US"
   ```

   This command creates a keystore file called `keystore.pkcs`. The certificate attributes are specified in the *dname* parameter: common name of the certificate, organization unit name (for example *Purchasing* or *Sales Department*), organization name, country.

2. Generate a self-signed certificate.

   Invoke the following in a command line terminal:

   ```
   keytool -selfcert -alias myAlias -keystore keystore.pkcs -storetype PKCS12
   ```

3. Optionally display the certificate details in a human readable form.

   First, the certificate must be exported to a separate file with the following command:

   ```
   keytool -export -alias myAlias -keystore keystore.pkcs -storetype PKCS12 -file certfile.cer
   ```

   The certificate details are displayed with the command:

   ```
   keytool -printcert -file certfile.cer
   ```

4. Edit the `default.properties` file and fill-in the parameters that hold the path to `keystore.pkcs` file (`keystore` parameter), keystore type (`storetype` parameter, with `JSK` or `PKCS12` as possible values), alias (`alias` parameter) and password (`password` parameter).
5. Sign the jar files using the certificate by running the `sign` Ant task available in *the applet project*.

Supported browsers and operating systems

The applet was tested for compatibility with the following browsers:

	IE 7	IE 8 (32bit)	IE 9 (64bit)	Firefox 3	Firefox 4	Firefox 6	Safari 5	Chrome	Opera
Windows XP	Passed	Passed	-	Passed	Passed	Passed	-	Passed	Passed
Vista	Failed	Passed	Failed	Passed	Passed	Passed	Failed	Passed	Passed
Windows 7	-	-	Passed	Passed	Passed	Passed	-	Passed	Passed
Mac OS X 10.6	-	-	-	Failed	Passed	Passed	Passed	Failed	Passed
Mac OS X Lion	-	-	-	Failed	Passed	Passed	Passed	Failed	Passed
Linux Ubuntu 10	-	-	-	Failed	-	Passed	-	Failed	Passed

Communication between the Web Page and Java Applet

Using the Java 1.6 *LiveConnect* technology, applets can communicate with Javascript code which runs in the Web Page. Javascript code can call an applet's Java methods and from the Java code you can invoke Javascript code from the web page.

> You are not limited to displaying only Swing dialogs from the applet. From an applet's operations you can invoke Javascript API which shows a web page and then obtains the data which has been filled by the user.

Troubleshooting

When the applet fails to start:

1. Make sure that your web browser really runs the next generation Java plug-in and not the legacy Java plug-in.

   For Windows and Mac OSX the procedure is straight forward. Some steps are given below for installing the Java plug-in on Linux.

   Manual Installation and Registration of Java Plugin for Linux:
   http://www.oracle.com/technetwork/java/javase/manual-plugin-install-linux-136395.html
2. Refresh the web page.
3. Remove the Java Webstart cache from the local drive and try again.
   - On Windows this folder is located in: `%APPDATA%\LocalLow\Sun\Java\Deployment\cache`;
   - On Mac OSX this folder is located in: `/Users/user_name/Library/Caches/Java/cache`;
   - On Linux this folder is located in: `/home/user/.java/deployment/cache`.
4. Remove the Author Applet Frameworks cache from the local drive and try again:
   - On Windows Vista or 7 this folder is located in: `%APPDATA%\Roaming\com.oxygenxml.author.component`;
   - On Windows XP this folder is located in: `%APPDATA%\com.oxygenxml.author.component`;

- On Mac OSX this folder is located in: `/Users/user_name/Library/Preferences/com.oxygenxml.author.component`;
- On Linux this folder is located in: `/home/user/.com.oxygenxml.author.component`.

5. Problems sometimes occur after upgrading the web browser and/or the JavaTM runtime. Redeploy the applet on the server by running ANT in your Author Component project. However, doing this does not always fix the problem, which often lies in the web browser and/or in the Java plug-in itself.
6. Sometimes when the HTTP connection is slow on first time uses the JVM would simply shut down while the jars were being pushed to the local cache (i.e., first time uses). This shut down typically occurs while handling `oxygen.jar`. One of the reasons could be that some browsers (Firefox for example) implement some form of "Plugin hang detector" See *https://developer.mozilla.org/en/Plugins/Out_of_process_plugins/The_plugin_hang_detector*.

Enable JavaWebstart logging on your computer to get additional debug information:

1. Open a console and run `javaws -viewer`;
2. In the **Advanced** tab, expand the **Debugging** category and check all boxes.
3. Expand the **Java console** category and choose **Show console**.
4. Save settings.
5. After running the applet, you will find the log files in:

- On Windows this folder is located in: `%APPDATA%\LocalLow\Sun\Java\Deployment\log`;
- On Mac OSX this folder is located in: `/Users/user_name/Library/Caches/Java/log`;
- On Linux this folder is located in: `/home/user/.java/deployment/log`.

Avoiding Resource Caching

A Java plugin installed in a web browser caches access to all HTTP resources that the applet uses. This is useful in order to avoid downloading all the libraries each time the applet is run. However, this may have undesired side-effects when the applet presents resources loaded via HTTP. If such a resource is modified on the server and the browser window is refreshed, you might end-up with the old content of the resource presented in the applet.

To avoid such a behaviour, you need to edit the `ro.sync.ecss.samples.AuthorComponentSampleApplet` class and set a custom `URLStreamHandlerFactory` implementation. A sample usage is already available in the class, but it is commented-out for increased flexibility:

```
//THIS IS THE WAY IN WHICH YOU CAN REGISTER YOUR OWN PROTOCOL HANDLER TO THE JVM.
//THEN YOU CAN OPEN YOUR CUSTOM URLs IN THE APPLET AND THE APPLET WILL USE YOUR HANDLER
URL.setURLStreamHandlerFactory(new URLStreamHandlerFactory() {
 public URLStreamHandler createURLStreamHandler(String protocol) {
  if("http".equals(protocol) || "https".equals(protocol)) {
   return new URLStreamHandler() {
    @Override
    protected URLConnection openConnection(URL u) throws IOException {
     URLConnection connection = new HttpURLConnection(u, null);
     if(!u.toString().endsWith(".jar")) {
      //Do not cache HTTP resources other than JARS
      //By default the Java HTTP connection caches content for
      //all URLs so if one URL is modified and then re-loaded in the
      //applet the applet will show the old content.
      connection.setDefaultUseCaches(false);
     }
     return connection;
    }
   };
  }
  return null;
 }
});
```

Adding MathML support in the Author Component Web Applet

By default the Author Component Web Applet project does not come with the libraries necessary for viewing and editing MathML equations in the Author page. You can view and edit MathML equations either by adding support for *JEuclid* or by adding support for *MathFlow*.

Adding MathML support using JEuclid

In the `author-component-dita.jnlp` JNLP file, refer additional libraries necessary for the `JEuclid` library to parse MathML equations:

```
<jar href="lib/jcip-annotations.jar"/>
<jar href="lib/jeuclid-core.jar"/>
<jar href="lib/batik-all-1.7.jar"/>
<jar href="lib/commons-io-1.3.1.jar"/>
<jar href="lib/commons-logging-1.0.4.jar"/>
<jar href="lib/xmlgraphics-commons-1.4.jar"/>
```

Copy these additional libraries to the component project `lib` directory from an `OXYGEN_INSTALLATION_DIRECTORY/lib` directory.

To edit specialized DITA Composite with MathML content, include the entire `OXYGEN_INSTALLATION_DIRECTORY/frameworks/mathml2` Mathml2 framework directory in the frameworks bundled with the component `frameworks.zip.jar`. This directory is used to solve references to MathML DTDs.

Adding MathML support using MathFlow

In the `author-component-dita.jnlp` JNLP file, refer additional libraries necessary for the `MathFlow` library to parse MathML equations:

```
<jar href="lib/MFComposer.jar"/>
<jar href="lib/MFExtraSymFonts.jar"/>
<jar href="lib/MFSimpleEditor.jar"/>
<jar href="lib/MFStructureEditor.jar"/>
<jar href="lib/MFStyleEditor.jar"/>
```

Copy these additional libraries from the `MathFlow` SDK.

In addition, you must obtain fixed `MathFlow` license keys for editing and composing MathML equations and register them using these API methods: `AuthorComponentFactory.setMathFlowFixedLicenseKeyForEditor` and `AuthorComponentFactory.setMathFlowFixedLicenseKeyForComposer`.

To edit specialized DITA Composite with MathML content, include the entire OXYGEN_INSTALLATION_DIRECTORY/frameworks/mathml2 Mathml2 framework directory in the frameworks bundled with the component frameworks.zip.jar. This directory is used to solve references to MathML DTDs.

Using Plugins with the Author Component

To bundle Workspace Access plugins, that are developed for standalone application with the Author Component, follow these steps:

- The content that is bundled to form the `frameworks.zip.jar` must contain the additional plugin directories, besides the framework directories. The content must also contain a `plugin.dtd` file.

  Note:

  Copy the `plugin.dtd` file from an `OXYGEN_INSTALL_DIR\plugins` folder.
- In the class which instantiates the `AuthorComponentFactory`, for example the `ro.sync.ecss.samples.AuthorComponentSample` class, call the methods `AuthorComponentFactory.getPluginToolbarCustomizers()`, `AuthorComponentFactory.getPluginViewCustomizers()` and `AuthorComponentFactory.getMenubarCustomizers()`, obtain the customizers which have been added by the plugins and call them to obtain the custom swing components that they contribute. There is a commented-out example for this in the `AuthorComponentSample.reconfigureActionsToolbar()` method for adding the toolbar from the **Acrolinx** plugin.

Important: As the Author Component is just a subset of the entire application, there is no guarantee that all the functionality of the plugin works.

Frequently asked questions

Installation and licensing

1. What hosting options are available for applet delivery and licensing services (i.e., Apache, IIS, etc.)?

For applet delivery any web server. We currently use Apache to deploy the sample on our site. For the floating license server you would need a J2EE server, like Tomcat if you want to restrict the access to the licenses.

If you do not need the access restrictions that are possible with a J2EE server you can simplify the deployment of the floating license server by using the standalone version of this server. The standalone license server is a simple Java application that communicates with Author Component by TCP/IP connections.

2. Are there any client requirements beyond the Java VM and (browser) Java Plug-In Technology?

Oracle (formerly Sun) Java JRE version 1.6 update 10 or newer. At least 200 MB disk space and 200MB free memory would be necessary for the Author Applet component.

3. Are there any other client requirements or concerns that could make deployment troublesome (i.e., browser security settings, client-side firewalls and AV engines, etc.)?

The applet is signed and will request access to the user machine, in order to store customization data (frameworks). The applet needs to be signed by you with a valid certificate.

4. How sensitive is the applet to the automatic Java VM updates, which are typically on by default (i.e., could automatic updates potentially "break" the run-time)?

The component should work well with newer Java versions but we cannot guarantee this.

5. How and when are "project" related files deployed to the client (i.e., applet code, DTD, styling files, customizations, etc.)?

Framework files are downloaded on the first load of the applet. Subsequent loads will re-use the cached customization files and will be much faster.

6. For on-line demo (http://www.oxygenxml.com/demo/AuthorDemoApplet/author-component-dita.html), noted a significant wait during initial startup. Any other mechanisms to enhance startup time?

See explanation above.

7. Does the XML Author component support multiple documents being open simultaneously? What are the licensing ramifications?

A single `AuthorComponentFactory` instance can create multiple `EditorComponentProvider` editors which can then be added and managed by the developer who is customizing the component in a Swing `JTabbedPane`. A single license (floating or user-based) is enough for this.

If you need to run multiple Java Applets or distinct Java processes using the Author component, the current floating license model allows for now only two concurrent components from the same computer when using the license servlet. An additional started component will take an extra license seat.

Another licensing technique would be to embed the license key in one of the jar libraries used by the applet. But you would need to implement your own way of determining how many users are editing using the Author applet.

8. Is there any internet traffic during an editing session (user actively working on the content, on the client side, in the XML Author component))?

No.

Functionality

1. How and when are saves performed back to the hosting server?

What you can see on our web site is just an example of the Author component (which is a Java Swing component) used in an Applet.

This applet is just for demonstration purposes. It's source can be at most a starting point for a customization. You should implement, sign and deploy your custom applet implementation.

The save operation could be implemented either in Javascript by requesting the XML content from the Applet or in Java directly working with the Author component. You would be responsible to send the content back to the CMS.

2. Is there a particular XML document size (or range) when the Author applet would start to exhibit performance problems?

   The applet has a total amount of used memory specified in the JNLP JavaWebstart configuration file which can be increased if necessary. By default it is 156 Mb. It should work comfortably with documents of 1-3 megabytes.

3. What graphic formats can be directly rendered in the XML Author component?

   GIF, JPEG, PNG, BMP and SVG.

4. Can links be embedded to retrieve (from the server) and "play" other types of digital assets, such as audio or video files?

   You could add listeners to intercept clicks and open the clicked links. This would require a good knowledge of the Author SDK. The Author component can only render static images (no GIF animations).

5. Does the XML Author component provide methods for uploading ancillary files (new graphics, for instance) to the hosting server?

   No.

6. Does the XML Author component provide any type of autosave functionality?

   By default no but you could customize the applet that contains the author component to save its content periodically to a file on disk.

7. Assuming multiple documents can be edited simultaneously, can content be copied, cut and pasted from one XML Author component "instance" to another?

   Yes.

8. Does the XML Author component support pasting content from external sources (such as a web page or a Microsoft Word document and, if so, to what extent?

   If no customizations are available the content is pasted as simple text. We provide customizations for the major frameworks (DITA, Docbook, TEI, etc) which use a conversion XSLT stylesheet to convert HTML content from clipboard to the target XML.

9. Can UTF-8 characters (such as Greeks, mathematical symbols, etc.) be inserted and rendered?

   Any UTF-8 character can be inserted and rendered as long as the font used for editing supports rendering the characters. The font can be changed by the developers but not by the users. When using a logical font (which by default is *Serif* for the Author component) the JVM will know how to map all characters to glyphs. There is no character map available but you could implement one

Customization

1. Please describe, in very general terms, the menus, toolbars, context menu options, "helper panes", etc. that are available for the XML Author component "out of the box".

   You can mount on your custom toolbar all actions available in the standalone Oxygen application for editing in the Author page. This includes custom actions defined in the framework customized for each XML type.

   The Author component also can provide the *Outline*, *Model*, *Elements* and *Attributes* views which can be added to your own panels (see sample applet).

2. Please describe, in general terms, the actions, project resources (e.g., DTD/Schema for validation purposes, CSS/XSL for styling, etc.) and typical level of effort that would be required to deploy a XML Author component solution for a customer with a proprietary DTD.

   The Author internal engine uses CSS to render XML.

   For a special type of XML you can create a custom framework (which also works in an Oxygen standalone version) which would also contain default schemas and custom actions. A simple framework would probably need 2-3 weeks development time. For a complex framework with many custom actions it could take a couple of months. Oxygen already has frameworks for editing Docbook, DITA, TEI, etc. Sources for them are available in *the Author SDK*.

   More than one framework can coexist in the same Oxygen instance (the desktop standalone version or the applet version) and can be used at the same time for editing XML documents.

3. Many customers desire a very simplistic interface for contributors (with little or no XML expertise) but a more robust XML editing environment for editors (or other users with more advanced XML savviness). How well does the XML Author component support varying degrees of user interface complexity and capability?

   - *Showing/hiding menus, toolbars, helpers, etc.*

     All the UI parts from the Author component are assembled by you. You could provide two applet implementations: one for advanced/power users and one for technical writers.

   - *Forcing behaviors (i.e., ensuring change tracking is on and preventing it from being shut down)*

     You could avoid placing the change tracking toolbar actions in the custom applet. You could also use API to turn change tracking ON when the content has been loaded.

   - *Preventing access to "privileged" editor processes (i.e., accept/reject changes)*

     You can remove the change tracking actions completely in a custom applet implementation. Including the ones from the contextual menu.

   - *Presenting and/or describing XML constructs (i.e., tags) in "plain-English"*

     Using our API you can customize what the Outline or Breadcrumb presents for each XML tag. You can also customize the in-place content completion list.

   - *Presenting a small subset of the overall XML tag set (rather than the full tag set) for use by contributors (i.e., allowing an author to only insert Heading, Para and inline emphasis) Could varying "interfaces", with different mixes these capabilities and customizations, be developed and pushed to the user based on a "role" or a similar construct?*

     The API allows for a content completion filter which also affects the *Elements* view.

4. Does the XML Author component's API provide access to the XML document, for manipulation purposes, using common XML syntax such as DOM, XPath, etc.?

   Yes, using the Author API.

5. Can custom dialogs be developed and launched to collect information in a "form" (with scripting behind to push tag the collection information and embed it in the XML document?

   Yes.

6. Can project resources, customizations, etc. be readily shared between the desktop and component versions of your XML Author product line?

   A framework developed for the Desktop Oxygen application can then be bundled with an Author component in a custom applet. For example the Author demo applet from our web site is DITA-aware using the same framework as the Oxygen standalone distribution.

   A custom version of the applet that includes one or more customized frameworks and user options can be built and deployed for non-technical authors by a technical savvy user using a built-in tool of Oxygen. All the authors that load the deployed applet from the same server location will share the same frameworks and options.

A custom editing solution can deploy one or more frameworks that can be used at the same time.

Creating and Running Automated Tests

If you have developed complex custom plugins and/or document types the best way to test your implementation and insure that further changes will not interfere with the current behavior is to make automated tests for your customization.

An installation standalone (Author or Editor) comes with a main `oxygen.jar` library located in the `OXYGEN_INSTALLATION_DIRECTORY`. That JAR library contains a base class for testing developer customizations named `ro.sync.exml.workspace.api.PluginWorkspaceTCBase`.

Please see below some steps in order to develop `JUnit` tests for your customizations using the **Eclipse** workbench:

1. Create a new Eclipse Java project and copy to it the entire contents of the `OXYGEN_INSTALLATION_DIRECTORY`.
2. Add to the **Java Build Path**->**Libraries** tab all JAR libraries present in the `OXYGEN_INSTALLATION_DIRECTORY/lib` directory. Make sure that the main JAR library `oxygen.jar` or `oxygenAuthor.jar` is the first one in the Java classpath by moving it up in the **Order and Export** tab.
3. Download and add to the Java build path the additional `JUnit` libraries *`jfcunit.jar`* and *`junit.jar`*.
4. Create a new Java class which extends `ro.sync.exml.workspace.api.PluginWorkspaceTCBase`.
5. Pass on to the constructor of the super class the following parameters:
   - `File frameworksFolder` The file path to the frameworks directory. It can point to a custom frameworks directory where the custom framework resides.
   - `File pluginsFolder` The file path to the plugins directory. It can point to a custom plugins directory where the custom plugins resides.
   - `String licenseKey` The license key used to license the test class.
6. Create test methods which use the API in the base class to open XML files and perform different actions on them. Your test class could look something like:

```
public class MyTestClass extends PluginWorkspaceTCBase {

 /**
  * Constructor.
  */
 public MyTestClass() throws Exception {
  super(new File("frameworks"), new File("plugins"),
        "------START-LICENSE-KEY------\n" +
    "\n" +
    "Registration_Name=Developer\n" +
    "\n" +
    "Company=\n" +
    "\n" +
    "Category=Enterprise\n" +
    "\n" +
    "Component=XML-Editor, XSLT-Debugger, Saxon-SA\n" +
    "\n" +
    "Version=14\n" +
    "\n" +
    "Number_of_Licenses=1\n" +
    "\n" +
    "Date=09-04-2012\n" +
    "\n" +
    "Trial=31\n" +
    "\n" +
    "SGN=MCwCFGNoEGJSeiC3XCYIyalvjzHhGhhqAhRNRDpEu8RIWb8icCJO7HqfVP4++A\\=\\=\n" +
    "\n" +
  "-------END-LICENSE-KEY-------");
 }

  /**
    * <p><b>Description:</b> TC for opening a file and using the bold operation</p>
    * <p><b>Bug ID:</b> EXM-20417</p>
    *
    * @author radu_coravu
    *
    * @throws Exception
    */
   public void testOpenFileAndBoldEXM_20417() throws Exception {
     WSEditor ed = open(new File("D:/projects/eXml/test/authorExtensions/dita/sampleSmall.xml").toURL());
     //Move caret
     moveCaretRelativeTo("Context", 1, false);
```

```
    //Insert <b>
    invokeAuthorExtensionActionForID("bold");
    assertEquals("<?xml version=\"1.0\" encoding=\"utf-8\"?>\n" +
      "<!DOCTYPE task PUBLIC \"-//OASIS//DTD DITA Task//EN\"
\"http://docs.oasis-open.org/dita/v1.1/OS/dtd/task.dtd\">\n" +
      "<task id=\"taskId\">\n" +
      "    <title>Task <b>title</b></title>\n" +
      "    <prolog/>\n" +
      "    <taskbody>\n" +
      "        <context>\n" +
      "            <p>Context for the current task</p>\n" +
      "        </context>\n" +
      "        <steps>\n" +
      "            <step>\n" +
      "                <cmd>Task step.</cmd>\n" +
      "            </step>\n" +
      "        </steps>\n" +
      "    </taskbody>\n" +
      "</task>\n" +
      "", getCurrentEditorXMLContent());
  }
}
```

API Frequently Asked Questions (API FAQ)

This section contains answers to common questions regarding the customisations using the *Author SDK*, *Author Component*, or *Plugins*.

For additional questions, *contact us*. The preferred approach is via email because API questions must be analysed thoroughly. We also provide code snippets in case they are required.

To stay up-to-date with the latest API changes, discuss issues and ask for solutions from other developers working with the SDK, register to the *oXygen-SDK mailing list*.

Difference Between a Document Type (Framework) and a Plugin Extension

Question

What is the difference between a Document Type (Framework) and a Plugin Extension?

Answer

Two ways of customising the application are possible:

1. Implementing a plugin.

   A plugin serves a general purpose and influences any type of XML file that you open in .

   For the Plugins API, Javadoc, samples, and documentation, go to *http://www.oxygenxml.com/oxygen_sdk.html#Developer_Plugins*

2. Creating or modifying the document type which is associated to your specific XML vocabulary.

   This document type is used to provide custom actions for your type of XML files and to mount them on the toolbar, menus, and contextual menus.

   For example, if the application end users are editing DITA, all the toolbar actions which are specific for DITA are provided by the DITA Document Type. If you look in the Oxygen Preferences->"Document Type Association" there is a "DITA" document type.

   If you *edit that document type* in you will see that it has an `Author` tab in which it defines all custom DITA actions and adds them to the toolbars, main menus, contextual menus.

   We have a special chapter in our user manual which explains how such document types are constructed and modified:

   http://www.oxygenxml.com/doc/ug-oxygen/index.html?q=/doc/ug-oxygen/topics/author-devel-guide-intro.html

   If you look on disk in the:

   ```
   OXYGEN_INSTALL_DIR\frameworks\dita
   ```

   folder there is a file called `dita.framework`. That file gets updated when you edit a document type from the Oxygen Preferences. Then you can share that updated file with all users.

   The same folder contains some JAR libraries. These libraries contain custom complex Java operations which are called when the user presses certain toolbar actions.

   If you want to add a custom action this topic explains how:

   http://www.oxygenxml.com/doc/ug-oxygen/index.html?q=/doc/ug-oxygen/tasks/addCustomActionHowTo.html

   We have an Author SDK which contains the Java sources from all the DITA Java customizations:

   http://www.oxygenxml.com/oxygen_sdk.html#XML_Editor_Authoring_SDK

Dynamically Modify the Content Inserted by the Writer

Question

Is there a way to insert typographic quotation marks instead of double quotes?

Answer

By using the API you can set a document filter to change the text that is inserted in the Author document. You can use this method to change the insertion of double quotes with the typographic quotes.

Here is some sample code:

```
authorAccess.getDocumentController().setDocumentFilter(new AuthorDocumentFilter() {
  /**
   * @see
ro.sync.ecss.extensions.api.AuthorDocumentFilter#insertText(ro.sync.ecss.extensions.api.AuthorDocumentFilterBypass,
 int, java.lang.String)
   */
  @Override
  public void insertText(AuthorDocumentFilterBypass filterBypass, int offset, String toInsert) {
    if(toInsert.length() == 1 && "\"".equals(toInsert)) {
      //User typed a quote but he actually needs a smart quote.
      //So we either have to add \u201E (start smart quote)
      //Or we add \u201C (end smart quote)
      //Depending on whether we already have a start smart quote inserted in the current paragraph.

      try {
        AuthorNode currentNode = authorAccess.getDocumentController().getNodeAtOffset(offset);
        int startofTextInCurrentNode = currentNode.getStartOffset();
        if(offset > startofTextInCurrentNode) {
          Segment seg = new Segment();
          authorAccess.getDocumentController().getChars(startofTextInCurrentNode, offset -
startofTextInCurrentNode, seg);
          String previosTextInNode = seg.toString();
          boolean insertStartQuote = true;
          for (int i = previosTextInNode.length() - 1; i >= 0; i--) {
            char ch = previosTextInNode.charAt(i);
            if('\u201C' == ch) {
              //Found end of smart quote, so yes, we should insert a start one
              break;
            } else if('\u201E' == ch) {
              //Found start quote, so we should insert an end one.
              insertStartQuote = false;
              break;
            }
          }

          if(insertStartQuote) {
            toInsert = "\u201E";
          } else {
            toInsert = "\u201C";
          }
        }
      } catch (BadLocationException e) {
        e.printStackTrace();
      }
    }
    System.err.println("INSERT TEXT |" + toInsert + "|");
    super.insertText(filterBypass, offset, toInsert);
  }
});
```

You can find the online Javadoc for `AuthorDocumentFilter` API here:
http://www.oxygenxml.com/InstData/Editor/SDK/javadoc/ro/sync/ecss/extensions/api/AuthorDocumentFilter.html

An alternative to using a document filtering is the use of a `ro.sync.ecss.extensions.api.AuthorSchemaAwareEditingHandlerAdapter` which has clear callbacks indicating the source from where the API is called (Paste, Drag and Drop, Typing).

Split Paragraph on Enter (Instead of Showing Content Completion List)

Question

How to split the paragraph on Enter instead of showing the content completion list?

Answer

To obtain this behaviour, *edit your Document Type* and in the `Author` tab, `Actions` tab, add your own split action. This action must have the **Enter** shortcut key associated and must trigger your own custom operation which handles the split.

So, when you press **Enter**, your Java operation is invoked and it will be your responsibility to split the paragraph using the current API (probably creating a document fragment from the caret offset to the end of the paragraph, removing the content and then inserting the created fragment after the paragraph).

This solution has as a drawback. hides the content completion window when you press **Enter**. If you want to show allowed child elements at that certain offset, implement your own content proposals window using the `ro.sync.ecss.extensions.api.AuthorSchemaManager` API to use information from the associated schema.

Impose Custom Options for Writers

Question

How to enable **Track Changes** at startup?

Answer

There are two ways to enable **Track Changes** for every document that you open:

1. You could customise the default options which are used by your writers and set the Track Changes `Initial State` option to `Always On.`
2. Use the API to toggle the Track Changes state after a document is opened in Author mode:

```
// Check the current state of Track Changes
boolean trackChangesOn = authorAccess.getReviewController().isTrackingChanges();
if (!trackChangesOn) {
  // Set Track Changes state to On
  authorAccess.getReviewController().toggleTrackChanges();
}
```

Highlight Content

Question

How can we add custom highlights to the Author document content?

Answer

There are two types of highlights you can add:

1. Not Persistent Highlights. Such highlights are removed when the document is closed and then re-opened.

   You can use the following API method:

```
ro.sync.exml.workspace.api.editor.page.author.WSAuthorEditorPageBase.getHighlighter()
```

to obtain an *AuthorHighlighter* which allows you to add a highlight between certain offsets with a certain painter.

For example you can use this support to implement your custom spell checker.

2. Persistent Highlights. Such highlights are saved in the XML content as processing instructions.

   You can use the following API method:

   ```
   ro.sync.exml.workspace.api.editor.page.author.WSAuthorEditorPageBase.getPersistentHighlighter()
   ```

   to obtain an *AuthorPersistentHighlighter* which allows you to add a persistent highlight between certain offsets and containing certain custom properties and render it with a certain painter.

   For example you can use this support to implement your own way of adding review comments.

How Do I Add My Custom Actions to the Contextual Menu?

The API methods `WSAuthorEditorPageBase.addPopUpMenuCustomizer` and `WSTextEditorPage.addPopUpMenuCustomizer` allow you to customize the contextual menu shown either in the Author or in the Text modes. The API is available both in the standalone application and in the Eclipse plugin.

Here's an elegant way to add from your Eclipse plugin extension actions to the Author page:

1. Create a pop-up menu customizer implementation:

```
import org.eclipse.jface.action.ContributionManager;
import org.eclipse.ui.PlatformUI;
import org.eclipse.ui.menus.IMenuService;
import ro.sync.ecss.extensions.api.AuthorAccess;
import ro.sync.ecss.extensions.api.structure.AuthorPopupMenuCustomizer;
/**
* This class is used to create the possibility to attach certain
* menuContributions to the {@link ContributionManager}, which is used for the
* popup menu in the Author Page of the Oxygen Editor.<br />
* You just need to use the org.eclipse.ui.menus extension and add a
* menuContribution with the locationURI: <b>menu:oxygen.authorpage</b>
*/
public class OxygenAuthorPagePopupMenuCustomizer implements
        AuthorPopupMenuCustomizer {

    @Override
    public void customizePopUpMenu(Object menuManagerObj,
            AuthorAccess authoraccess) {
        if (menuManagerObj instanceof ContributionManager) {
            ContributionManager contributionManager = (ContributionManager) menuManagerObj;
            IMenuService menuService = (IMenuService) PlatformUI.getWorkbench()
                    .getActiveWorkbenchWindow().getService(IMenuService.class);

            menuService.populateContributionManager(contributionManager,
                    "menu:oxygen.authorpage");
            contributionManager.update(true);
        }
    }
}
```

2. Add a workbench listener and add the pop-up customizer when an editor is opened in the Author page:

```
Workbench.getInstance().getActiveWorkbenchWindow().getPartService().addPartListener(
    new IPartListener() {
      @Override
      public void partOpened(IWorkbenchPart part) {
        if(part instanceof ro.sync.exml.workspace.api.editor.WSEditor) {
          WSEditorPage currentPage = ((WSEditor)part).getCurrentPage();
          if(currentPage instanceof WSAuthorEditorPage) {
           ((WSAuthorEditorPage)currentPage).addPopUpMenuCustomizer(new OxygenAuthorPagePopupMenuCustomizer());

          }
        }
      }
      ........
    });
```

3. Implement the extension point in your `plugin.xml`:

```
<extension
       point="org.eclipse.ui.menus">
    <menuContribution
          allPopups="false"
          locationURI="menu:oxygen.authorpage">
       <command
             commandId="eu.doccenter.kgu.client.tagging.removeTaggingFromOxygen"
             style="push">
       </command>
    </menuContribution>
</extension>
```

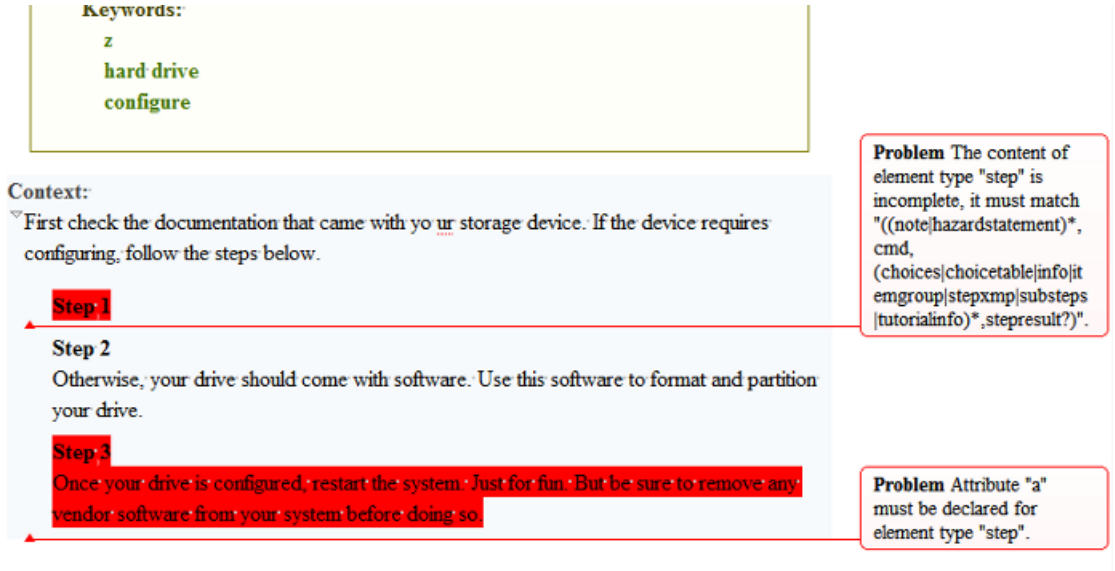
Adding Custom Callouts

Question

I'd like to highlight validation errors, instead of underlining them, for example changing the text background color to light red (or yellow).Also I like to let oxygen write a note about the error type into the author view directly at the error position, like " [value "text" not allowed for attribute "type"] ". Is this possible using the API?

Answer

The Plugins API allows setting a `ValidationProblemsFilter` which gets notified when automatic validation errors are available. Then you can map each of the problems to an offset range in the Author page using the API `WSTextBasedEditorPage.getStartEndOffsets(DocumentPositionedInfo)`. For each of those offsets you can add either persistent or non-persistent highlights. If you add persistent highlights you can also customize callouts to appear for each of them, the downside is that they need to be removed before the document gets saved. The end result would look something like:



Here is a small working example:

```
    /**
    * Plugin extension - workspace access extension.
    */
    public class CustomWorkspaceAccessPluginExtension implements WorkspaceAccessPluginExtension {

      /**
       * @see
ro.sync.exml.plugin.workspace.WorkspaceAccessPluginExtension#applicationStarted(ro.sync.exml.workspace.api.standalone.StandalonePluginWorkspace)
       */
      public void applicationStarted(final StandalonePluginWorkspace pluginWorkspaceAccess) {
        pluginWorkspaceAccess.addEditorChangeListener(new WSEditorChangeListener() {
```

```
          /**
           * @see ro.sync.exml.workspace.api.listeners.WSEditorChangeListener#editorOpened(java.net.URL)
           */
          @Override
          public void editorOpened(URL editorLocation) {
            final WSEditor currentEditor = pluginWorkspaceAccess.getEditorAccess(editorLocation,
StandalonePluginWorkspace.MAIN_EDITING_AREA);
            WSEditorPage currentPage = currentEditor.getCurrentPage();
            if(currentPage instanceof WSAuthorEditorPage) {
              final WSAuthorEditorPage currentAuthorPage = (WSAuthorEditorPage)currentPage;
            currentAuthorPage.getPersistentHighlighter().setHighlightRenderer(new PersistentHighlightRenderer()
 {
                @Override
                public String getTooltip(AuthorPersistentHighlight highlight) {
                  return highlight.getClonedProperties().get("message");
                }
                @Override
                public HighlightPainter getHighlightPainter(AuthorPersistentHighlight highlight) {
                  //Depending on severity could have different color.
                  ColorHighlightPainter painter = new ColorHighlightPainter(Color.COLOR_RED, -1, -1);
                  painter.setBgColor(Color.COLOR_RED);
                  return painter;
                }
              });
              currentAuthorPage.getReviewController()
                .getAuthorCalloutsController().setCalloutsRenderingInformationProvider(
                    new CalloutsRenderingInformationProvider() {
                @Override
                public boolean shouldRenderAsCallout(AuthorPersistentHighlight highlight) {
                  //All custom highlights are ours
                  return true;
                }
                @Override
                public AuthorCalloutRenderingInformation getCalloutRenderingInformation(
                    final AuthorPersistentHighlight highlight) {
                  return new AuthorCalloutRenderingInformation() {
                    @Override
                    public long getTimestamp() {
                      //Not interesting
                      return -1;
                    }
                    @Override
                    public String getContentFromTarget(int limit) {
                      return "";
                    }
                    @Override
                    public String getComment(int limit) {
                      return highlight.getClonedProperties().get("message");
                    }
                    @Override
                    public Color getColor() {
                      return Color.COLOR_RED;
                    }
                    @Override
                    public String getCalloutType() {
                      return "Problem";
                    }
                    @Override
                    public String getAuthor() {
                      return "";
                    }
                    @Override
                    public Map<String, String> getAdditionalData() {
                      return null;
                    }
                  };
                }
              });
              currentEditor.addValidationProblemsFilter(new ValidationProblemsFilter() {
                List<int[]> lastStartEndOffsets = new ArrayList<int[]>();
                /**
                 * @see
ro.sync.exml.workspace.api.editor.validation.ValidationProblemsFilter#filterValidationProblems(ro.sync.exml.workspace.api.editor.validation.ValidationProblems)

                 */
                @Override
                public void filterValidationProblems(ValidationProblems validationProblems) {
                  List<int[]> startEndOffsets = new ArrayList<int[]>();
                  List<DocumentPositionedInfo> problemsList = validationProblems.getProblemsList();
                  if(problemsList != null) {
                    for (int i = 0; i < problemsList.size(); i++) {
                      try {
                        startEndOffsets.add(currentAuthorPage.getStartEndOffsets(problemsList.get(i)));
                      } catch (BadLocationException e) {
                        e.printStackTrace();
                      }
                    }
                  }
                    if(lastStartEndOffsets.size() != startEndOffsets.size()) {
```

```
                    //Continue
                  } else {
                    boolean equal = true;
                    for (int i = 0; i < startEndOffsets.size(); i++) {
                      int[] o1 = startEndOffsets.get(i);
                      int[] o2 = lastStartEndOffsets.get(i);
                      if(o1 == null && o2 == null) {
                        //Continue
                      } else  if(o1 != null && o2 != null
                          && o1[0] == o2[0] && o1[1] == o2[1]){
                        //Continue
                      } else {
                        equal = false;
                        break;
                      }
                    }
                    if(equal) {
                      //Same list of problems already displayed.
                      return;
                    }
                  }
                  //Keep last used offsets.
                  lastStartEndOffsets = startEndOffsets;
                try {
                  if(! SwingUtilities.isEventDispatchThread()) {
                    SwingUtilities.invokeAndWait(new Runnable() {
                      @Override
                      public void run() {
                        //First remove all custom highlights.
                        currentAuthorPage.getPersistentHighlighter().removeAllHighlights();
                      }
                    });
                  }
                } catch (InterruptedException e1) {
                  e1.printStackTrace();
                } catch (InvocationTargetException e1) {
                  e1.printStackTrace();
                }
                if(problemsList != null) {
                  for (int i = 0; i < problemsList.size(); i++) {
                    //A reported problem (could be warning, could be error).
                    DocumentPositionedInfo dpi = problemsList.get(i);
                    try {
                      final int[] currentOffsets = startEndOffsets.get(i);
                      if(currentOffsets != null) {
                        //These are offsets in the Author content.
                        final LinkedHashMap<String, String> highlightProps = new LinkedHashMap<String,
String>();
                        highlightProps.put("message", dpi.getMessage());
                        highlightProps.put("severity", dpi.getSeverityAsString());
                        if(! SwingUtilities.isEventDispatchThread()) {
                          SwingUtilities.invokeAndWait(new Runnable() {
                            @Override
                            public void run() {
                              currentAuthorPage.getPersistentHighlighter().addHighlight(
                                  currentOffsets[0], currentOffsets[1] - 1, highlightProps);
                            }
                          });
                        }
                      }
                    } catch (InterruptedException e) {
                      e.printStackTrace();
                    } catch (InvocationTargetException e) {
                      e.printStackTrace();
                    }
                  }
                }
              }
            });
            currentEditor.addEditorListener(new WSEditorListener() {
              /**
               * @see ro.sync.exml.workspace.api.listeners.WSEditorListener#editorAboutToBeSavedVeto(int)
               */
              @Override
              public boolean editorAboutToBeSavedVeto(int operationType) {
                try {
                  if(! SwingUtilities.isEventDispatchThread()) {
                    SwingUtilities.invokeAndWait(new Runnable() {
                      @Override
                      public void run() {
                        //Remove all persistent highlights before saving
                        currentAuthorPage.getPersistentHighlighter().removeAllHighlights();
                      }
                    });
                  }
                } catch (InterruptedException e) {
                  e.printStackTrace();
                } catch (InvocationTargetException e) {
                  e.printStackTrace();
```

```
                }
                return true;
              }
            });
          }
        }
      }, StandalonePluginWorkspace.MAIN_EDITING_AREA);
    }


    /**
     * @see ro.sync.exml.plugin.workspace.WorkspaceAccessPluginExtension#applicationClosing()
     */
    public boolean applicationClosing() {
      return true;
    }
  }
```

Change the DOCTYPE of an Opened XML Document

Question

How to change the DOCTYPE of a document opened in the Author mode?

Answer

The following API:

```
ro.sync.ecss.extensions.api.AuthorDocumentController.getDoctype()
```

allows you to get the `DOCTYPE` of the current XML file opened in the Author page.

There is also an API method available which would allow you to set the `DOCTYPE` back to the XML:

```
ro.sync.ecss.extensions.api.AuthorDocumentController.setDoctype(AuthorDocumentType)
```

Here is an example of how this solution would work:

```
AuthorDocumentType dt = new AuthorDocumentType("article", "testSystemID", "testPublicID",
            "<!DOCTYPE article PUBLIC \"testPublicID\" \"testSystemID\">");
docController.setDoctype(dt);
```

Basically you could take the entire content from the existing DOCTYPE,

```
ro.sync.ecss.extensions.api.AuthorDocumentType.getContent()
```

modify it to your needs, and create another `AuthorDocumentType` object with the new content and with the same public, system IDs.

For example you could use this API is you want to add unparsed entities in the XML `DOCTYPE`.

Customize the Default Application Icons for Toolbars/Menus

Question

How can we change the default icons used for the application built-in actions?

Answer

If you look inside the main JAR library `OXYGEN_INSTALL_DIR\lib\oxygen.jar` or `OXYGEN_INSTALL_DIR\lib\author.jar` it contains an `images` folder in which all the images which we use for our buttons, menus, and toolbars exist.

In order to overwrite them with your own creations:

1. In the `OXYGEN_INSTALL_DIR\lib` directory create a folder called `endorsed`;
2. In the `endorsed` folder create another folder called `images`;
3. Add your own images in the `images` folder.

You can use this mechanism to overwrite any kind of resource located in the main Oxygen JAR library. The folder structure in the endorsed directory and in the main Oxygen JAR must be identical.

Disable Context-Sensitive Menu Items for Custom Author Actions

Question

Is there a way to disable menu items for custom Author actions depending on the cursor context?

Answer

By default Oxygen does not toggle the enabled/disabled states for actions based on whether the activation XPath expressions for that certain Author action are fulfilled. This is done because the actions can be many and evaluating XPath expression on each caret move can lead to performance problems. But if you have your own `ro.sync.ecss.extensions.api.ExtensionsBundle` implementation you can overwrite the method:

```
ro.sync.ecss.extensions.api.ExtensionsBundle.createAuthorExtensionStateListener()
```

and when the extension state listener gets activated you can use the API like:

```
/**
 * @see
ro.sync.ecss.extensions.api.AuthorExtensionStateListener#activated(ro.sync.ecss.extensions.api.AuthorAccess)
 */
public void activated(final AuthorAccess authorAccess) {

  //Add a caret listener to enable/disable extension actions:
  authorAccess.getEditorAccess().addAuthorCaretListener(new AuthorCaretListener() {
    @Override
    public void caretMoved(AuthorCaretEvent caretEvent) {
      try {
        Map<String, Object> authorExtensionActions =
authorAccess.getEditorAccess().getActionsProvider().getAuthorExtensionActions();
        //Get the action used to insert a paragraph. It's ID is "paragraph"
        AbstractAction insertParagraph = (AbstractAction) authorExtensionActions.get("paragraph");
        //Evaluate an XPath expression in the context of the current node in which the caret is located
        Object[] evaluateXPath = authorAccess.getDocumentController().evaluateXPath(".[ancestor-or-self::p]",
false, false, false, false);
        if(evaluateXPath != null && evaluateXPath.length > 0 && evaluateXPath[0] != null) {
          //We are inside a paragraph, disable the action.
          insertParagraph.setEnabled(false);
        } else {
          //Enable the action
          insertParagraph.setEnabled(true);
        }
      } catch (AuthorOperationException e) {
        e.printStackTrace();
      }
    }
  });
```

When the extension is deactivated you should remove the caret listener in order to avoid adding multiple caret listeners which perform the same functionality.

Dynamic Open File in Distributed via JavaWebStart

Question

How can we dynamically open a file in an distributed via JWS?

Answer

The JWS packager ANT build file which comes with Oxygen signs by default the JNLP file (this means that a copy of it is included in the main JAR library) in this step:

```
<copy file="${outputDir}/${packageName}/${productName}.jnlp" tofile="${home}/JNLP-INF/APPLICATION.JNLP"/>
```

Signing the JNLP file indeed means that it is impossible to automatically generate a JNLP file containing some dynamic arguments.

But the JNLP does not need to be signed. Indeed the user probably receives this information when launching the application but at least in this way you should be able to dynamically generate a JNLP file via a PHP script based on the URL which was clicked by the user.

The generated JNLP would then take as argument the URL which needs to be opened when Oxygen starts.

Maybe a different approach (more complicated though) would be to have the JNLP file signed and always refer as a URL argument a location like this:

```
http://path/to/server/redirectEditedURL.php
```

When the URL gets clicked on the client side you would also call a PHP script on the server side which would update the redirect location for `redirectEditedURL.php` to point to the clicked XML resource. Then the opened Oxygen would try to connect to the redirect PHP and be redirected to open the XML.

Change the Default Track Changes (Review) Author Name

Question

How can we change the default author name used for Track Changes in the Author Component?

Answer

The Track Changes (Review) Author name is determined in the following order:

1. **API** - The review user name can be imposed through the following API:

   ```
   ro.sync.ecss.extensions.api.AuthorReviewController.setReviewerAuthorName(String)
   ```

2. **Options** - If the author name was not imposed from the API, it is determined from the `Author` option set from the following `Preferences` page: `Editor / Edit modes / Author / Review`.
3. **System properties** - If the author name was not imposed from the API or from the application options then the following system property is used:

   ```
   System.getProperty("user.name")
   ```

So, to impose the Track Changes author, use one of the following approaches:

1. Use the API to impose the reviewer Author name. Here is the online Javadoc of this method: *http://www.oxygenxml.com/InstData/Editor/SDK/javadoc/ro/sync/ecss/extensions/api/AuthorReviewController.html#setReviewerAuthorName(java.lang.String)*
2. Customise the default options and set a specific value for the reviewer Author name option.
3. Set the value of `user.name` system property when the applet is initialising and before any document is loaded.

Multiple Rendering Modes for the Same Author Document

Question

How can we add multiple buttons, each showing different visualisation mode of the same Author document (by associating additional/different CSS style sheet)?

Answer

In the toolbar of the Author mode there is a drop-down button which contains alternative CSS styles for the same document. To add an alternative CSS stylesheet got o `Preferences->Document Type Association` page, select the document type associated with your documents and press `Edit`. In the `Document Type` dialog that appears go to "Author" tab, "CSS" tab and add there references to alternate CSS stylesheets.

For example, one of the alternate CSSs that we offer for `DITA` document type is located here:

```
OXYGEN_INSTALL_DIR/frameworks/dita/css_classed/hideColspec.css
```

If you open it, you will see that it imports the main CSS and then adds selectors of its own.

Obtain a DOM Element from an `AuthorNode` or `AuthorElement`

Question

Can a DOM Element be obtained from an `AuthorNode` or an `AuthorElement`?

Answer

No, a DOM Element cannot be obtained from an `AuthorNode` or an `AuthorElement`. The `AuthorNode` structure is also hierarchical but the difference is that all the text content is kept in a single text buffer instead of having individual text nodes.

We have an image in the Javadoc which explains the situation:*http://www.oxygenxml.com/InstData/Editor/SDK/javadoc/ro/sync/ecss/extensions/api/node/AuthorDocumentFragment.html*

Print Document Within the Author Component

Question

Can a document be printed within the Author Component?

Answer

You can use the following API method to either print the Author document content to the printer or to show the Print Preview dialog, depending on the `preview` parameter value:

```
AuthorComponentProvider.print(boolean preview)
```

Here is the online Javadoc for this method:
http://www.oxygenxml.com/InstData/Editor/SDK/javadoc/ro/sync/ecss/extensions/api/component/AuthorComponentProvider.html#print(boolean)

Running XSLT or XQuery Transformations

Question

Can I run XSL 2.0 transformation with Saxon EE using the oXygen SDK?

Answer

The API class `ro.sync.exml.workspace.api.util.XMLUtilAccess` allows you to create an XSLT Transformer which implements the JAXP interface `javax.xml.transform.Transformer`. Then this type of transformer can be used to transform XML. Here's just an example of transforming when you have an `AuthorAccess` API available:

```
 InputSource is = new org.xml.sax.InputSource(URLUtil.correct(new File("test/personal.xsl")).toString());
 xslSrc = new SAXSource(is);
 javax.xml.transform.Transformer transformer = authorAccess.getXMLUtilAccess().createXSLTTransformer(xslSrc,
null, AuthorXMLUtilAccess.TRANSFORMER_SAXON_ENTERPRISE_EDITION);
 transformer.transform(new StreamSource(new File("test/personal.xml")), new StreamResult(new
File("test/personal.html")));
```

If you want to create the transformer from the plugins side, you can use this method instead: `ro.sync.exml.workspace.api.PluginWorkspace.getXMLUtilAccess()`.

Use Different Rendering Styles for Entity References, Comments or Processing Instructions

Question

Is there a way to display entity references in the Author mode without the distinct gray background and tag markers?

Answer

There is a built-in CSS stylesheet in the Oxygen libraries which is used when styling content in the Author mode, no matter what CSS you use. This CSS has the following content:

```
@namespace oxy url('http://www.oxygenxml.com/extensions/author');
@namespace xi "http://www.w3.org/2001/XInclude";
@namespace xlink "http://www.w3.org/1999/xlink";
@namespace svg "http://www.w3.org/2000/svg";
@namespace mml "http://www.w3.org/1998/Math/MathML";

oxy|document {
    display:block !important;
}

oxy|cdata {
    display:morph !important;
    white-space:pre-wrap !important;
    border-width:0px !important;
    margin:0px !important;
    padding: 0px !important;
}

oxy|processing-instruction {
    display:block !important;
    color: rgb(139, 38, 201) !important;
    white-space:pre-wrap !important;
    border-width:0px !important;
    margin:0px !important;
    padding: 0px !important;
}

oxy|comment {
    display:morph !important;
    color: rgb(0, 100, 0) !important;
    background-color:rgb(255, 255, 210) !important;
    white-space:pre-wrap !important;
    border-width:0px !important;
    margin:0px !important;
```

```
    padding: 0px !important;
}

oxy|reference:before,
oxy|entity[href]:before{
  link: attr(href) !important;
  text-decoration: underline !important;
  color: navy !important;

  margin: 2px !important;
  padding: 0px !important;
}

oxy|reference:before {
  display: morph !important;
  content: url(../images/editContent.gif) !important;
}

oxy|entity[href]:before{
  display: morph !important;
  content: url(../images/editContent.gif) !important;
}

oxy|reference,
oxy|entity {
    editable:false !important;
    background-color: rgb(240, 240, 240) !important;
    margin:0px !important;
    padding: 0px !important;
}

oxy|reference {
    display:morph !important;
}

oxy|entity {
    display:morph !important;
}

oxy|entity[href] {
  border: 1px solid rgb(175, 175, 175) !important;
  padding: 0.2em !important;
}

xi|include {
    display:block !important;
    margin-bottom: 0.5em !important;
    padding: 2px !important;
}
xi|include:before,
xi|include:after{
    display:inline !important;
    background-color:inherit !important;
    color:#444444 !important;
    font-weight:bold !important;
}

xi|include:before {
    content:url(../images/link.gif) attr(href)  !important;
    link: attr(href) !important;
}
xi|include[xpointer]:before {
    content:url(../images/link.gif) attr(href) " " attr(xpointer) !important;
    link: oxy_concat(attr(href), "#", attr(xpointer)) !important;
}

xi|fallback {
    display:morph !important;
    margin: 2px !important;
    border: 1px solid #CB0039 !important;
}

xi|fallback:before {
    display:morph !important;
    content:"XInclude fallback: " !important;
    color:#CB0039 !important;
}

oxy|doctype {
    display:block !important;
    background-color: transparent !important;
    color:blue !important;
    border-width:0px !important;
    margin:0px !important;
    padding: 2px !important;
}

oxy|error {
    display:morph !important;
```

```
    editable:false !important;
    white-space:pre !important;
    color: rgb(178, 0, 0) !important;
    font-weight:bold !important;
}

*[xlink|href]:before {
    content:url(../images/link.gif);
    link: attr(xlink|href) !important;
}


/*No direct display of the MathML and SVG images.*/
svg|svg{
 display:inline !important;
    white-space: trim-when-ws-only;
}
svg|svg svg|*{
    display:none !important;
    white-space:normal;
}

mml|math{
 display:inline !important;
    white-space: trim-when-ws-only;
}
mml|math mml|*{
    display:none !important;
    white-space: normal;
}
```

In the CSS used for rendering the XML in **Author** mode do the following:

- import the special Author namespace;
- use a special selector to customize the `entity` node.

Example:

```
@namespace oxy url('http://www.oxygenxml.com/extensions/author');
oxy|entity {
   background-color: inherit !important;
   margin:0px !important;
   padding: 0px !important;
    -oxy-display-tags:none;
}
```

You can overwrite styles in the predefined CSS in order to custom style comments, processing instructions and *CData* sections. You can also customize the way in which `xi:include` elements are rendered.

Insert an Element with all the Required Content

Question

I'm inserting a DITA *image* XML element, using the Author API, which points to a certain resource and has required content. Can the required content be automatically inserted by the application?

Answer

The API `ro.sync.ecss.extensions.api.AuthorSchemaManager` can propose valid elements which can be inserted at the specific offset. Using the method `AuthorSchemaManager.createAuthorDocumentFragment(CIElement)` you can convert the proposed elements to document fragments (which have all the required content filled in) which can then be inserted in the document.

```
AuthorSchemaManager schemaManager = this.authorAccess.getDocumentController().getAuthorSchemaManager();
WhatElementsCanGoHereContext context =
schemaManager.createWhatElementsCanGoHereContext(this.authorAccess.getEditorAccess().getCaretOffset());
List<CIElement> possibleElementsAtCaretPosition = schemaManager.whatElementsCanGoHere(context);
loop: for (int i = 0; i < possibleElementsAtCaretPosition.size(); i++) {
  CIElement possibleElement = possibleElementsAtCaretPosition.get(i);
  List<CIAttribute> attrs = possibleElement.getAttributes();
  if(attrs != null) {
    for (int j = 0; j < attrs.size(); j++) {
      CIAttribute ciAttribute = attrs.get(j);
```

```
      if (ciAttribute.getName().equals("class")) {
        if (ciAttribute.getDefaultValue() != null
            && ciAttribute.getDefaultValue().contains("  topic/image ")) {
          //Found a CIElement for image
          //Create a fragment for it. The fragment contains all required child elements already built.
          AuthorDocumentFragment frag = schemaManager.createAuthorDocumentFragment(possibleElement);
          //Now set the @href to it.
          //Ask the user and obtain a value for the @href
          //Then:

          String href = "test.png";
          List<AuthorNode> nodes = frag.getContentNodes();
          if(!nodes.isEmpty()) {
            AuthorElement imageEl = (AuthorElement) nodes.get(0);
            imageEl.setAttribute("href", new AttrValue(href));
          }
          //And insert the fragment.

this.authorAccess.getDocumentController().insertFragment(this.authorAccess.getEditorAccess().getCaretOffset(),
 frag);
          break loop;
        }
      }
    }
  }
}
```

Obtain the Current Selected Element Using the Author API

Question

If in the Author mode, an element is fully selected, I would like to perform an action on it. If not, I would like to perform an action on the node which is located at the caret position. Is this possible via the API?

Answer

When an element is fully selected by the user the selection start and end offsets are actually outside of the node's offset bounds. So using `AuthorDocumentController.getNodeAtOffset` will actually return the parent of the selected node. We have some special API which makes it easier for you to determine this situation: `WSAuthorEditorPageBase.getFullySelectedNode()`.

```
AuthorDocumentController controller = authorPageAccess.getDocumentController();
AuthorAccess authorAccess = authorPageAccess.getAuthorAccess();
int caretOffset = authorAccess.getEditorAccess().getCaretOffset();

AuthorElement nodeAtCaret = (AuthorElement) authorAccess.getEditorAccess().getFullySelectedNode();
if (nodeAtCaret == null) {
     //We have no fully selected node. We can look at the caret offset.
     nodeAtCaret = (AuthorElement) authorAccess.getDocumentController().getNodeAtOffset(caretOffset);
    //Or we could look at the selection start and end, see which node is the parent of each offset and get the
 closest common ancestor.
}
```

Index

A

C

O