Blog

Contents

Chapter 1. Welcome! **8**

Chapter 2. Frequently Asked Questions **11**

Chapter 3. Artificial intelligence (AI) **12**

AI XPath Extension Functions in Oxygen XML Editor 12

Oxygen AI Positron Assistant Add-on 16

AI Positron Add-on Functionality - Small Clips 19

Customizing AI Positron for Your DITA XML Project 24

Creating Custom AI Actions for the AI Positron add-on 30

Overcoming Challenges - Editing DITA XML content with AI Tools (Presentation) 42

AI-Powered XML Editing (Presentation) 43

Utilizing GenAI to Enhance Early-Stage Technical Content Creation (Presentation) 43

Using ChatGPT in a DITA XML Project 43

AI Positron Assistant - Retrospective Analysis 54

Providing content to AI engines using DITA OT plugins 58

What is New in Oxygen AI Positron Assistant 5.0 58

Chapter 4. DITA **60**

DITA XML Editing, Collaboration and Publishing Overview 60

Learning 66

Resources for learning DITA with Oxygen 66

Using DITA to Document a Software Product 118

Guided DITA Authoring Solution Overview 120

Maintaining a Healthy DITA Project 123

DITA Linking Strategies 139

DITA 1.3 Branch Filtering - Next Generation of Reuse 146

DITA 1.3 Key Scopes - Next Generation of Reuse 148

DITA Reuse Strategies 150

Cross-Book Links 163

DITA XML vs Markdown Syntax and Capabilities Comparison....169
Publishing Customizations....178
Server Side Building and Publishing....179
DITA WebHelp Publishing Solution Advantages....187
PDF....189
Embedding Diagrams in DITA topics using PlantUML....193
Embedding Diagrams in DITA Topics Using Mermaid....194
Presenting DITA Task Steps as Diagrams....195
Show Consecutive DITA Code Blocks in Separate Tabs....198
Show DITA Task Steps in Separate Tabs....199
Converting OpenAPI to DITA XML and Publishing to WebHelp (Experimental)....200
DITA to Confluence....202
Generating a list of changes between releases....205
DITA Publishing Customization Overview....207
DITA Open Toolkit Frequently Asked Questions (FAQ)....209
Enhancing DITA Publishing With Plugins....210
Adding support for embedding LaTeX equations in DITA content....213
Useful DITA OT Plugins....214
Creating a simple DITA Open Toolkit plugin to customize published HTML and PDF content....216
Using Git client-side hooks to run DITA publishing pipelines....218
Creating Link Graphs From DITA Maps....222
Publishing DITA Content Using the MKDocs Web Site Generator....222
Creating PowerPoint Slides from DITA Content....226
Adding Labels to Topics in the WebHelp Responsive Output....227
Displaying prolog details (author, creation date, read time)....229
Dynamically publishing a list with the most recent topics....231
Dynamically Generating an RSS Feed....233
Generating Google Structured Data from your DITA tasks....236
Generating Google Structured Data from your DITA frequently asked questions....239

Editing Customizations.... 242
Startup DITA Project.... 242
Customizing the DITA Visual Editing Experience.... 245
Customizing the DITA Framework Using a Framework Extension Script.... 246
Controlled Attribute Values for your DITA Project.... 250
Converting Subject Scheme Map Values to a DITAVAL.... 254
Converting .xpr Profiling Conditions to Standard DITA Files.... 255
Other.... 263
Oxygen XML As a Technical Documentation Solution FAQ.... 263
DITA For Small Technical Documentation Teams.... 266
Types Of Companies Who Would Benefit From Using the DITA XML Standard.... 273
Supporting the DITA OT Project as a Commercial Company.... 274
DITA Map Metrics Evolution.... 277
Oxygen XML Editor DITA Editing solution strong points.... 278
DITA Oriented Tips And Tricks.... 279
DITA Project Enhancements.... 285
Translating your DITA Project.... 287
DITA Map Validate and Check for Completeness Overview.... 289
Schematron Checks to help Technical Writing.... 291
Implementing your own Style Guide.... 294
Preprocessing DITA-OT Project Files.... 296
Re: 10 reasons for moving away from DITA.... 303
DITA 2.0 Specification Support Cheat Sheet.... 306
Refactoring.... 309
Creating a Knowledge Base for the Google Dialogflow Chatbot using DITA XML content.... 331
Similarities Between Technical Doc Writers and Software Engineers.... 338
Chapter 5. Migrate.... 352
Migrating to a Structured Standards-based Documentation Solution.... 352
Migrating Various Document Formats to DITA.... 354

How to Migrate from Word to DITA....357
Migrating MS Word to DITA using the Batch Documents Converter....358
Batch converting HTML to XHTML....367
Migrating Unstructured Adobe FrameMaker Content to DITA....368
Automation - Using DITA XML Topic Titles to Generate File Names....373
Chapter 6. Using Git For Technical Writing....375
Advantages of Using Git....375
Disadvantages of Using Git....376
Useful Resources....377
Applications for Working with Git....378
Typical Editing Session....379
Editing Session with Oxygen Content Fusion....382
Validation....385
Common Workflows....386
Centralized Workflow....387
Feature Branches....388
Branches For All Changes....389
Release Branches....389
GitFlow....394
Propose Changes....395
Handling Translations....396
Sharing Common Settings....397
Publishing Content From Git....397
Tame the Branch Chaos – Switch Less, Write More....398
Chapter 7. SDK....406
Customizing Oxygen XML Editor (Overview)....406
Document Type Extension Sharing....407
Sharing Schematron Validation Rules....409
Public hosted Oxygen Plugin and Framework Projects....411

Sharing New Custom File Templates for a Specific Vocabulary.... 413
Composing Author Actions.... 415
Implementing a Custom Author Action to Split a Table....417
Adding a Custom Author Action to the Content Completion Window.... 419
How Special Paste works in Oxygen....422
The Oxygen SDK (Part 1: Plugins).... 424
The Oxygen SDK (Part 2: Frameworks)....425
Your First Oxygen Add-on....427
Oxygen Add-ons Overview.... 429
Adding CALS-table related functionality to your custom Oxygen framework.... 433
Convert Code Templates to External Author Actions....435
Speed Up Oxygen XML Web Author Customization (use Web Author Test Server Add-on)....437
Chapter 8. Development.... 439
How to Use Text Files as Primary Input for XSLT in Oxygen XML Editor.... 439
Editing Complex XML Schemas with Oxygen XML Editor.... 442
Chapter 9. Miscellaneous.... 445
ISO 27001:2022 Certification Announcement....445
Oxygen XML Users Meetup @ XML Prague 2024: A Deep Dive into XML Technologies and Innovations....446
Announcing the Launch of the Oxygen XML Chinese User Group....448
Oxygen XML Ecosystem.... 449
Add-ons For Technical Documentation.... 449
Tips And Tricks....451
Checking Terminology with Oxygen XML Editor....458
10 Things I Enjoy About the New Oxygen JSON Editor.... 460
Opinions about using Oxygen.... 461
A set of rules for providing great tech support.... 461
A Short Story of Reuse.... 462
Sharing Application Settings.... 464

Collaboration for Documenting a Software Product using DITA..... 465
Collaboration (Teams working on a common XML project)..... 466
Enable massive contributions with oXygen XML Web Author and GitHub..... 467
All About Editor Variables..... 469
XSLT Training..... 472
Miscellaneous.....
Log4Shell - Oxygen XML Vulnerability Analysis FAQ..... 472
Using Oxygen XML Editor in a Web Browser (Experimental)..... 476
Oxygen XML Blog 2022 Retrospective..... 477
Chapter 10. Contributors..... 480
Index.....a

Chapter 1. Welcome!

Our blog contains various useful articles and tutorials covering various aspects of editing, developing, and publishing using Oxygen XML Editor with various XML technologies.

Sometimes (quite often actually) articles originally written for the blog are curated by our technical writers and end up being part of our official documentation: https://www.oxygenxml.com/doc/ug-editor/. However, the blog allows us to deploy various small articles faster, without the need to wait for documentation updates.

Follow us!

You can use an RSS feed reader to follow the blog for new articles: https://blog.oxygenxml.com/rss.xml.

Or you can register to receive emails when new articles are posted.

History of the Oxygen XML Editor Blog

The blog was originally hosted by Google Blogs and the first articles on the blog appeared around 2007. Since Google Blogs did not allow enough flexibility in creating the published content (for example, the inability to create a table of contents) the blog was migrated to DITA XML content stored on GitHub and is now hosted both on the Oxygen XML Web site and on the Netlify platform: https://oxygenxmlblog.netlify.app/.

How the Oxygen XML Editor Blog Works

The entire source code of the Oxygen XML Editor blog is stored in DITA content on a public GitHub repository: https://github.com/oxygenxml/blog.

There is a Gradle build file that uses a special DITA Open Toolkit to produce WebHelp Responsive, EPUB and PDF outputs from the DITA content. The following features are available in the published output:

- Color themes can be easily changed with CSS to modify how the output looks.
- Search functionality. Possibility to add index terms.
- RSS Feed is automatically generated from the entire list of topics.
- PDF output is automatically generated using Oxygen Chemistry Engine and can be customized with CSS.
- EPUB output is automatically generated using DITA For Publishers plugins.
- Each page has an **Edit online** link allowing it to be edited in the Oxygen XML Web Author.

- Each page has a comments section build using the Oxygen Feedback platform.
- Each time content is added to the project there is a configured GitHub action which runs a validation report of the DITA content (reporting for example broken links).

How You can Contribute to the Oxygen XML Editor Blog.

We want to encourage our end users and collaborators to start creating small Oxygen-related articles for our blog. To post your own articles, follow these steps:

1. Create your own GitHub account, if you do not already have one.
2. Fork the contents of the Oxygen XML Editor GitHub repository in your own GitHub Repository: https://github.com/oxygenxml/blog. More details about how to fork a repository: https://docs.github.com/en/get-started/quickstart/contributing-to-projects.
3. Make changes to your forked project or to a branch from it.
    - If you have the Oxygen XML Editor application installed, you can download our Git add-on that should allow you to clone the fork, make changes, and then commit and push them to your repository.
    - As an alternative, you can use the Oxygen XML Web Author browser editing tool that has its own Git connector: https://www.oxygenxml.com/oxygen-xml-web-author/app/oxygen.html.
    - Please remember to add the name of the author and the created date for the topic in the DITA topic prolog section like:

      ```
      <prolog>
          <author>Radu Coravu</author>
          <critdates>
              <created date="2020-01-14"/>
          </critdates>
      </prolog>
      ```

      This prolog section will be used to add in the published HTML output the author information after the topic title.
    - Each commit message must include a sign off.
4. After all changes have been made to your fork of the project, you can create a pull request to the Oxygen XML Editor original GitHub project.
5. There is a GitHub action implemented in the project which will automatically run DITA validation on the entire project branch and will report on the pull request if the validation is successful.
6. We will review your changes and if your pull requests is accepted, the blog web site will be automatically updated with the new changes.

Technologies Used to Store, Build and Publish the Oxygen XML Blog

- The blog content is stored on a public GitHub repository.
- GitHub's issues list is used to organize the work and add new improvement ideas.
- Offline editing the blog is done using Oxygen XML Editor with its Git add-on installed.
- Online editing is done using Oxygen Web Author.
- A Gradle build file is used to download a custom DITA Open Toolkit which contains the Oxygen WebHelp Responsive and PDF CSS plugins pre-installed, then run the publishing and validation stages.
- The DITA Open Toolkit is used along with custom plugins to publish the DITA content.
    - Oxygen WebHelp is used to publish the DITA content to HTML output. Publishing templates are used to customize the published output.
    - The Oxygen Feedback system is used for adding feedback forms to each page.
    - Oxygen PDF Chemistry is used to produce PDF for the publishing platform.
    - The DITA For Publishers plugins are used to produce the EPUB output.
- Oxygen DITA Validate and Check for Completeness is run as a GitHub action with the help of a special scripting license.
- Netlify is used to connect to the repository, run Gradle and publish the WebSite.

Chapter 2. Frequently Asked Questions

How do I register to receive notifications for new blog posts?

Each blog HTML page has at the end a form in which you can fill your email address if you want to be notified when new posts are made.

Can I contribute articles to the Oxygen XML Blog?

Yes, the procedure is listed here: How You can Contribute to the Oxygen XML Editor Blog. *(on page 9)*.

How do I learn DITA XML with Oxygen?

Read our Resources for learning DITA with Oxygen *(on page 66)* blog post.

How do I convert various output formats to DITA XML?

Read our Migrating Various Document Formats to DITA *(on page 354)* and How to Migrate from Word to DITA *(on page 357)* blog posts.

Are there any productivity tips and tricks?

Our Tips And Tricks *(on page 451)* and DITA Oriented Tips And Tricks *(on page 279)* posts contains lots of interesting productivity tips and tricks.

Chapter 3. Artificial intelligence (AI)

AI XPath Extension Functions in Oxygen XML Editor

Introduction

The integration of **AI-driven XPath extension functions** in Oxygen XML Editor represents a major step forward in the XML content lifecycle by introducing intelligent automation. These features improve Schematron, XSLT, and XQuery workflows by enabling natural language-driven rephrasing, validation, and custom AI-powered actions directly within your schemas and stylesheets. This post provides an overview of the key functions, practical examples, and configuration options to help you use these AI capabilities effectively, safely, and at scale.

Meet the AI XPath Extension Functions

The Oxygen AI Positron add-on introduces three primary functions (namespace `http://www.oxygenxml.com/ai/function`) that can be used directly in Schematron, XSLT, XQuery, XProc, and XPath expressions:

ai:transform-content(system, (user, assistant,)* user)

- **Purpose:** Transforms or rephrases content using AI, guided by a system prompt and user-provided input.
- **Parameters:**
  - **system:** Context or rule (e.g., “Follow company style guidelines”).
  - **user:** The input text to be transformed.
  - **assistant:** (Optional, repeatable) Previous AI responses for context in multi-stage operations.
- **Returns:** AI-generated, transformed string.

Sample Use in Schematron:

```
<sqf:replace match="text()" select="ai:transform-content('Reformulate phrase to be less than 50 words', .)"/>
```

ai:verify-content(system, (user, assistant,)* user)

- **Purpose:** Validates content using natural language prompts, returning a Boolean result.
- **Parameters:** Same as above.
- **Returns:** Boolean (true/false), ideal for driving Schematron assertions.

Sample Use in Schematron:

```
<sch:report test="ai:verify-content('Does the following content have passive voice?', .)" sqf:fix="rephrase">
    The phrase uses passive voice.
</sch:report>
```

ai:invoke-action(actionId, extraPrompt, content)

- **Purpose:** Executes a pre-configured AI action with additional context or prompts on a given content block.
- **Parameters:**
  - **actionId:** ID of the registered AI action.
  - **extraPrompt:** Additional context for the action.
  - **content:** XML node or string to process.
- **Returns:** The transformed (or analyzed) string.

Sample Use in Schematron:

```
<sqf:replace match="text()" select="ai:invoke-action('Reformulate phrase to be less than 50 words', .)"/>
```

Real-World Example: Optimizing Short Descriptions

Validation & Refactoring in One Step:

This example demonstrates a rule that validates if a short description exceeds 50 words and provides an automated fix to rephrase it to meet the word limit. The rule uses Schematron and Smart Fix (SQF) to ensure concise and effective short descriptions.

```
<sch:rule context="shortdesc">
    <sch:report test="count(tokenize(.,'\s+')) > 50" sqf:fix="rephrase">
        The phrase must contain less than 50 words.
    </sch:report>
    <sqf:fix id="rephrase">
        <sqf:description>
            <sqf:title>Rephrase phrase to be less than 50 words</sqf:title>
        </sqf:description>
        <sqf:replace match="text()" select="ai:transform-content('Reformulate phrase to be less than 50 words', .)"/>
    </sqf:fix>
</sch:rule>
```

Active Voice Checker:

This example illustrates a rule that checks if a short description contains passive voice and provides an automated fix to rephrase it into active voice. The rule leverages AI-based content verification and transformation to enhance the clarity and impact of the text.

```
<sch:rule context="shortdesc">
    <sch:report test="ai:verify-content('Does the following content have passive voice?', .)"
                sqf:fix="rephrase">The phrase uses passive voice.</sch:report>
    <sqf:fix id="rephrase">
        <sqf:description>
            <sqf:title>Rephrase text to be active voice</sqf:title>
        </sqf:description>
        <sqf:replace match="text()" select="ai:transform-content('Rephrase text to be active voice', .)"/>
    </sqf:fix>
</sch:rule>
```

Refactor Content Using XSL and AI

The combination of AI and XSLT allows for dynamic and intelligent content transformation. By utilizing AI-driven XPath extension functions, you can automate tasks such as generating alternate text for images, ensuring accessibility and compliance with minimal manual intervention.

Example: Generate Alternate Text for Images

The following XSLT template generates alternate text for images with a `keyref` attribute but without an `alt` attribute. The template employs the `ai:transform-content` function to create a concise alternate text description for the image. The generated alternate text is then added to the `alt` element within the copied image node. This method ensures that all images that lack alternate text are enhanced with meaningful descriptions, improving both accessibility and usability.

```
<xsl:template match="image[@keyref][not(alt)]">
    <xsl:copy>
        <xsl:apply-templates select="@*"/>
        <alt>
            <xsl:value-of select="
                ai:transform-content(
                'Create a short alternate text description for this image:',
                concat('{attach(', resolve-uri(@href, base-uri()), ')}'))"/>
        </alt>
        <xsl:apply-templates select="node()"/>
```

```
    </xsl:copy>
</xsl:template>
```

This example demonstrates how AI can dynamically generate meaningful alternate text for images, ensuring your content is accessible and enriched with metadata.

Example: Language Translation and Validation

The following XSLT template showcases how AI-driven XPath functions can be used for multi-language translation and validation. In this example, content is translated into French in three variants, then translated back into English. The English translations are compared with the original content to determine the most accurate French translation:

```
    <xsl:template match="text()">
        <xsl:variable name="agentFrench" select="ai:transform-content('Translate to French in 3 variants ', .)"/>
        <xsl:variable name="agentEnglish" select="ai:transform-content('Translate content from French to English ',
 $agentFrench)"/>
        <xsl:value-of select="ai:transform-content(
            'Compare the English translation with original content and use the correct French translation. ',
            'Translate to french in 3 variants ',
            $agentFrench,
            'Translate from french to english ',
            $agentEnglish,.)"/>
    </xsl:template>
```

This method ensures that the final French translation is both accurate and contextually appropriate, leveraging AI's capabilities to process and validate multilingual content effectively.

Configuration Options: Balancing Power and Performance

Oxygen provides several options to customize AI function behavior according to your project's requirements:

Enable XPath Functions

- **Purpose:** Activates AI-driven XPath functions for Schematron/XSLT validation and transformation.
- **Default:** Enabled.

Cache Responses and Reuse Them for Identical Prompts

- **Purpose:** Minimizes redundant API calls and accelerates validation/transformation by caching AI responses for identical inputs.
- **Default:** Enabled.
- **Management:** A "Clear cache" button allows you to instantly delete all stored responses.

Cache Size

- **Purpose:** Sets an upper limit on the number of responses that can be cached, preventing excessive memory/resource usage.

Notify Me When the Number of Requests Exceeds…

- **Purpose:** Provides a safeguard for large-scale operations.
- **Behavior:** If the specified threshold of AI requests is exceeded, Oxygen displays a confirmation dialog box. You can choose to stop using the XPath AI functions if the volume seems excessive.

Summary: Why Use AI XPath Functions in Oxygen?

- **Express style, validation, and transformation in natural language**, eliminating the need for numerous rigid rules.
- **Automate repetitive editorial tasks** (e.g., word count, voice, grammar) and provide AI-powered fixes directly in authoring tools.
- **Control resource usage, compliance, and performance** through detailed configuration options.

These features are revolutionizing the XML content pipeline, enabling smart, standards-compliant authoring and validation to be effortless and secure.

Ready to integrate AI into your XML workflow? Activate these options in Oxygen XML Editor and start making your content more consistent, compliant, and future-ready today.

Related information

AI Positron XPath Extension Functions

AI XPath Extension Functions Options

XPath AI Extension Functions Explained

Oxygen AI Positron Assistant Add-on

The **Oxygen AI Positron Assistant** add-on provides support for helping technical documentation writers **generate**, **re-write** content, **overview** and **research**. It also supports performing AI checks and AI-based

modifications on **multiple files** by integrating AI extensions with XML technologies like **Schematron** or **XSLT**.

This article is intended to be a general overview of the **Oxygen AI Positron Assistant** add-on installation, licensing, and functionality.

We've envisioned this add-on as a way to offer you a personal assistant that contains useful actions for checking content (readability, grammar), translating, explaining, and creating overviews.

https://www.youtube.com/embed/-qJZaD_aDMU

Installation and Licensing Overview

- Oxygen desktop allows installing two flavors of the add-on:
  - AI Positron Assistant add-on.

    This add-on works through a subscription-based model using the Oxygen Positron Service platform, which in turn uses the OpenAI API and functionality.

    Users can create new subscription accounts or authenticate using their existing **Google** or **Github** accounts. The first subscription month is free and a monthly subscription can be purchased to receive a number of credits each month.

    You can control which AI engine is used by default for performing actions and for chatting. This can be done from the Preferences page.

  - AI Positron Assistant Enterprise add-on.

    This add-on allows end users to connect to their own AI engine server. The following connection types are supported:
    - OpenAI - Connect to your own OpenAI server using your own key. Or connect to OpenAI-compatible servers like **Ollama**.
    - Microsoft Azure Open AI - Connect to an Open AI instance hosted as a virtual machine on Microsoft Azure.
    - Anthropic Claude - Connect to Anthropic Claude using your own key.

    From the licensing point of view, the **AI Positron Assistant Enterprise** add-on works free of charge with any Oxygen XML Editor installation using an **Enterprise** license type for Oxygen XML Editor. For other cases, a special license key needs to be purchased for the add-on to enable this direct access. You can use the Register button in the AI Positron Assistant side view to configure the special license key.

- Oxygen XML Web Author also allows installing an **AI Positron Assistant** plugin with similar functionality as the one in the Oxygen Desktop application. Currently, the WebAuthor and Content Fusion online demo websites come with the AI Positron add-on pre-installed. In the near future, an **AI Positron Assistant Enterprise** plugin will allow using either a license server or a special Web Author license component to configure a custom AI engine.

Integration and Features Overview

After installing the add-on, the following features are available in the Oxygen desktop:

- AI Positron Assistant side view. This side view contains the following functionality:
    - Invoke various actions to generate and re-write content using AI.
    - **Chat** with the AI.
    - A **History** list of previous interactions.
    - A **Favorites** drop-down list and a way to mark inputs as favorites.
    - The **Insert Editor Variables** drop-down list allows inserting various editor variables which are expanded when the chat message is sent to the AI engine.
    - An **Attach** button to attach images or XML text resources to the chat message.
- **AI** main menu entry with support to invoke actions or to go to the add-on's preferences page.
- **AI Positron Assistant** contextual menu for all XML documents opened in the **Author** visual editing mode.
- **AI Positron Assistant** quick assist actions available on the left vertical stripe for all XML documents opened in the **Author** visual editing mode.
- **AI Positron Refactoring** actions added to the **Project** and **DITA Maps Manager** views as a way to run actions on multiple files.
- When validation problems are displayed in the **Results** pane, you can right-click on a problem and use the **AI Positron Fix** action to ask the **AI Positron** platform for help with fixing the problem. It will propose content in the chat pane (within the **AI Positron Assistant** view) that can be used to solve the problem.
- The add-on contributes two XPath extension functions (`ai:transform-content` and `ai:verify-content`) that can be used to rephrase content or to perform validation checks on existing content.

Predefined and Custom Actions

Once you log in to the server, a set of predefined actions are available, grouped in categories: **Accessibility**, **Content Generation**, **Development**, **Rewrite**, **Overview**, **Marketing**, and **Translation**. The AI Positron Assistant Samples Playground samples project contains useful samples to test the functionality of these actions.

Custom actions can also be created and used. The same AI Positron Assistant Samples Playground contains some sample custom actions. Some usage examples can be found in the samples playground.

Visual Image Recognition (Vision)

The pre-defined **Generate Documentation Draft** and **Update Content Based on Images** actions allow using the AI to create or update content depending on existing images from your user's manual. You can also attach images in the **Chat** panel.

Batch Processing

The **AI Positron Refactoring** actions added to the **Project** and **DITA Maps Manager** views are a way to run actions on multiple files. For example, you could use the predefined **Translate to** action to translate multiple DITA topics into a certain language or apply the **Correct Grammar** or **Improve Readability** actions on multiple resources.

In addition, **XML Refactoring** actions based on XSLT or XQuery can use the XPath extension functions added by the add-on to perform batch changes in XML documents. Some examples can be found in the samples playground.

Useful Links

- AI Positron Assistant Samples Playground.
- Blog Post About AI Positron Add-on By Tom Johnson
- Webinar: AI as a Tool for Technical Content Creation
- Webinar: Leveraging the Power of AI and Schematron for Content Verification and Correction
- Meetup Recording: Oxygen AI Positron Assistant - Integrating AI Across Oxygen Tools

AI Positron Add-on Functionality - Small Clips

A list of small animated clips showcasing some interesting aspects of the functionality.

Generat a draft DITA XML topic from an image

Batch generating alternate text for DITA XML images

Generate a Mermaid diagram from a task

Add comments about grammar problems and then resolve them

Proof Read and Resolve Comments

Add Structured Content - Project based Retrieval Augmentation (RAG)

Improve Structure

Customizing AI Positron for Your DITA XML Project

The Oxygen AI Positron Assistant Add-on *(on page 16)* includes a variety of helpful AI actions designed to boost productivity. However, **DITA XML** projects have specific requirements. They often need particular **terminology** and adherence to **style guide rules**, or they may rely on company-specific **DITA** specialization DTDs/schemas that differ from the standard **DITA DTDs** that the AI models are trained on. Therefore, to generate high-quality content using the AI actions, the add-on must be configured to understand the unique aspects of the **DITA XML** project it will be used with.

AI Tools Pros/Cons

With their ability to **rewrite**, **research**, and **generate documentation drafts** AI models can be useful in a variety of cases as long as aspects specific to the current DITA XML project can be controlled and enforced.

Good	Bad/Ugly
Re-write content (correct grammar, active voice, style, readability improvements).	The company specific style guide might impose certain rules that, by default, are not taken into account by the AI model.
Generating draft content by combining multiple sources (including images, attached **PDF**, **Word** or **Markdown** documents).	Hallucinations when generating content if not enough quality context information is provided. The overall terminology or style guide is not taken into account. The DITA XML specializations are also not taken into account by default.
Research / **Explain** concepts (chat).	By default an AI model responds to questions based on its training and not on the content from the current user's manual.

Access Content From Your Project

By default, when **researching** or **asking** the AI questions, Retrieval Augmented Generation (RAG) tools give the AI access to **search for specific keywords** in your current DITA XML documentation and thus provide the AI access to the content of topics which are related and relevant to your discussion.

Bundled AI actions like **Generate Documentation Draft**, **Expand Draft**, **Create Topics** give the AI model access to the current indexed DITA XML project to retrieve the topics with content similar to the new content that needs to be created.

You can create your own custom AI actions that allow RAG access to the current project contents for the AI engine. An example can be found here.

Overall Project Context and Audience

In the Oxygen **Preferences AI Positron Assistant** preferences page you can set in the **Context** text area general information about the current project and the intended audience. This context information will be used subsequently for all AI actions and chat sessions.

Example of a context which establishes an overall project scope and audience:

```
# Project context


The current project contains technical documentation for a fun and easy to drive

dune vehicle named the Sharkfin Rover.
```

```
# Audience


Technical documentation must be precise and easy to understand.

Our clients are not technical, they are usually beginners when it comes to driving

and their understanding of English may vary.
```

The same context information should be expanded to include terminology and style guide information for the current project. An example can be found here .

Terminology Rules

A DITA XML project may define (in various formats) allowed terms and certain synonyms that are not recommended to be used instead of the allowed terminology.

The **Context** field in the **AI Positron Assistant** preferences page can also contain rules for the most important terms and disallowed synonyms.

Example of context that establishes some of the most important terms:

```
# Terminology


Specific terms in the DITA XML content must be wrapped in the <term> element. Preserve all such terms and do not

 attempt to replace them with possible synonyms.


Always prefer to use the terms below instead of possible synonyms:

```

Sharkfin Rover (instead of 'Sharkfin Buggy')

vehicle (instead of 'car')

lorry (instead of 'truck')

rear position lamp (instead of 'tailgate')

powertrain (instead of 'drivetrain')

petrol engine (instead of 'Otto engine')

motorcycle (instead of 'motor-bike')

engine (instead of 'motor')

electronic stability control (instead of 'ESP')

motor vehicle chassis (instead of 'vehicle frame')

```
```

Once the AI model receives the imposed terminology rules as part of the prompt context, it will take them into account for all interactions (e.g. for creating documentation drafts from content that was written without taking the terminology into account).

The Oxygen Terminology Checker add-on can be configured with a terminology rules file that allows it to highlight terminology problems in real time and propose replacement for them. You can find a sample terminology rules file here. There is also a sample AI action that can generate an XML rules file from the add-on based on a style guide.

Once the **Terminology Checker** add-on is installed and flags the terminology problems in the **Author** visual editor mode, you can use the **Fix Terminology Problems** AI action to fix terminology problems in the currently opened topic's contents.

As a conclusion, you can add your most important terminology rules in the **Context** setting to create and process DITA XML content while directly obeying the rules. However, installing the terminology checker add-on allows you to see terminology problems highlighted in the visual editor and to fix them using the AI engine with the bundled **Fix Terminology Problems** AI action.

Style Guide

A style guide is a set of project-specific standards and guidelines used to ensure consistency and clarity in technical documentation. A company's style guide may come in various formats (DITA XML, Markdown, Word, etc.) but some of the most important rules in the style guide can be distilled and set in the **Context** field within the **AI Positron Assistant** preferences page.

Example of context that establishes some of the most important style guide rules:

```
# Style guide


There is a set of company specific Style Guide rules which are important to follow when rewriting or creating new

 content, the most important rules are mentioned below:


- Never use a metaphor, simile, or other figure of speech.

- Never use a long word where a short one will do.

- Keep sentences short. If it is possible to cut a word out, always cut it out.

- Never use the passive where you can use the active.

- Never use a foreign phrase, a scientific word, or a jargon word if you can think of an everyday English equivalent.

- Don’t use text that points to other topics such as "In the document below" or "in the following section."

- Some words or phrases are disallowed to be used. Mapping of disallowed phrases to their valid equivalent constructs:

```
```
```

```
as well as, in addition to => and

at this time, at this/that point in time => currently, now/then

advance planning => planning

by means of => by

by reason of => because of

due to the fact => because

during the course of => during

end result => result

```
```

The Oxygen Terminology Checker add-on can also match disallowed words and phrases in the terminology rules file, allowing you to see in the visual editor the style guide problems and to correct them using the bundled **Fix Terminology Problems** AI action.

Style Guide rules may also impose structural aspects for the DITA XML content and such rules can be automated by converting them to Schematron rules *(on page 291)*. Once such Schematron rules are added to the DITA XML project and start showing errors in the edited content, the bundled **Fix Validation Problems** AI action can be used to ask the AI to fix the reported problems in the current document. If the AI engine attempts to save content to a topic and the saved content has validation problems, the AI receives the validation problems and attempts to auto fix the problems until the topic becomes valid.

An alternative to defining a style guide in the AI context prompt or to impose them after the content is generated would be to create a fine-tuned AI model trained with small examples based on the internal style guide and created also using an AI model.

DITA XML DTD/Schemas Specialization

Quite often, a company that uses the DITA XML standard may decide to create a DTD or RelaxNG based specialization that defines new XML elements that are not known by the AI engine.

Most bundled AI actions attempt to preserve the existing XML structure, as much as possible. AI actions like **Expand Draft** start from a newly created DITA XML topic and thus preserves the existing DTD/schema reference and XML elements.

Actions like **Generate Documentation Draft** can specify (in their definition) that a certain XML structure must be generated. An example, for this can be found here.

The most important XML vocabulary changes (DTD/schema references, newly added XML elements) can be distilled and set in the **Context** field within the **AI Positron Assistant** preferences page.

Example of vocabulary changes described in the prompt context:
```

```
# Vocabulary Changes


DITA XML topics generated for our project's user's manual have the following structure and DOCTYPE declaration:


```

<!DOCTYPE oxyTopic PUBLIC "OXY-PUB-TOPIC" "oxy-system">

<oxyTopic>

    <title></title>

    <body>

    </body>

</oxyTopic>

```


DITA XML concepts generated for our project's user's manual have the following structure and DOCTYPE declaration:


```

<!DOCTYPE oxyConcept PUBLIC "OXY-PUB-CONCEPT" "oxy-system">

<oxyConcept>

    <title></title>

    <body>

    </body>

</oxyConcept>

```


New DITA XML elements are defined in our specific DITA XML vocabulary as follows:


There is a new <part-term> XML element which must be used to tag various car parts.

There is a new <product> XML element which must be used to tag product names.


Example:


The <product>Hisense</product> <part-term>engine</part-term> is more efficient.
```

Conclusion

You need to invest time to adapt the AI Positron add-on to generate better content for your particular project and organization particularities.

By specifying your specific project particularities in a context prompt, content created and re-written by an AI engine can become much more precise and fit better with your existing ways of working. Some of your project's rules can be automated allowing you to also detect and fix such problems after the content is generated using the AI engine.

An example of a detailed context prompt that contains an overview of the project and audience, terminology, style rules, described vocabulary changes can be found here.

Creating Custom AI Actions for the AI Positron add-on

In the AI Positron Assistant preferences page, you can define a reference to a folder that contains custom actions.

Once the add-on is installed, the New Document wizard can be used to create either a new **AI Positron Custom Action** file that contains a custom action definition in JSON format or an **AI Positron Custom Actions List** that contains a JSON array with multiple defined actions. Validation and content completion are automatically provided for such custom action files. To edit multi-line parameter values, you can switch the edited action JSON definition file to the **Author** visual editor mode.

If the action definition files are saved in the custom actions folder defined in the AI Positron Assistant preferences page, the AI Positron Assistant view should automatically reload its Actions list to include them.

Various examples of custom AI actions can be found here and can be used as starting points for your own similar AI actions.

An AI action definition may contain the following properties:

id

A unique ID for the action. If the ID of the action starts with `agent.hidden.action.`, then the action is not visible to be invoked directly but can be called by using the `invoke_ai_action(actionID, content)` function.

Example:

```
{
        "id": "improve.grammar.custom.styles",
```

```
        "title": "...",
        ...
}
```

Note:

An Oxygen desktop plugin can contribute an extra `translation.xml` file that contains translations. Such a plugin may translate titles and descriptions for AI actions using the action ID as part of the key.

Example:

```
<key value="improve.grammar.custom.styles.title">
 <val lang="en_US">Improve Styles</val>
 <val lang="de_DE">...</val>
 ...
</key>
<key value="improve.grammar.custom.styles.description">
 <val lang="en_US">Analyze and improve styles.</val>
 <val lang="de_DE">...</val>
 ...
</key>
```

title

The action name/title that gets presented in the UI.

Example:

```
{
        "id": "improve.grammar.custom.styles",
        "title": "Improve Grammar and Styles",
        ...
}
```

description

A longer description of what the AI action does, usually used as a tooltip.

Example:

```
{
        "id": "...",
        "title": "Improve Grammar and Styles",
        "description": "Improve the grammar and styles based on the company's style guide."
}
```

short-description

A shorter description for the AI action that usually appears under the action title in the chat view. If not specified, it defaults to the description value.

Example:

```
{
    "id": "action.translate.to.english",
    "title": "English",
    "short-description": "Translate to English",
    "description": "Translate the selected content to English while preserving its
    original DITA XML markup.",
    ....
}
```

categoryId

The optional category in which the AI action is visually grouped, if not specified it defaults to `Other`.

Example:

```
{
        "id": "improve.grammar.custom.styles",
        "title": "Improve Grammar and Styles",
        "categoryId": "Rewrite"
}
```

framework

A framework pattern (like '`DITA*`') or content type name (like '`text/css`') that, when matched, presents the action when the document belonging to the framework is opened.

Example:

```
{
    "id": "generate.img.alternate.text",
```

```
    "title": "Generate Image Alternate Text",
    "framework": "*DITA*",
    ...
}
```

An example of an AI action that shows this for any edited XML document (as it matches the `text/xml` content type) can be found here.

prompt_ref

A reference to a separate file (usually encoded in Markdown) which defines the entire action prompt.

Example:

```
{
    "id": "action.update.title.based.on.content",
    ....
    "prompt_ref": "prompts/action_update_title.md"
  }
```

Important:

The prompt is the most important part of the AI action. A prompt must be as precise and detailed as possible, including examples. Some companies may adopt prompt methodologies like **COSTAR** to specify the prompt value. The context prompt defined in the Oxygen AI Positron Preferences page is used as a prefix for all AI action prompts.

Here is an example of a large prompt file.

A custom AI action can expand the context prompt in a specific part of its prompt by using the `${contextPrompt}` variable in its own prompt.

To define a custom AI action that does not use the context prompt in any way, in the AI action's JSON configuration file, you can expand the context prompt to an empty value like this:

```
{
    "id": "improve.grammar.custom",
    "title": "..",
     ....
```

```
    "expand-params": [
        {
         "name": "contextPrompt",
         "value": ""
        }
    ]
}
```

prompt

Multi-line parameter value that defines inline the prompt that is sent to the AI along with the content to be processed.

Example:

```
{
    "id": "action.review.logical.mistakes",
    "title": "Logical Inconsistencies",
    "output-action": "none",
    "prompt": "# CONTEXT #\n\nYou will act as a reviewer interested in finding inconsistency
    and logical mistakes.\n\n# OBJECTIVE #\n\nYou are tasked with reviewing the given content.
    \nFind logical and inconsistency mistakes in the content and return a report."
}
```

Important:

The prompt is the most important part of the AI action. A prompt must be as precise and detailed as possible, including examples. Some companies may adopt prompt methodologies like **COSTAR** to specify the prompt value. The context prompt defined in the Oxygen AI Positron Preferences page is used as a prefix for all AI action prompts.

Here is an example of a large prompt file.

A custom AI action can expand the context prompt in a specific part of its prompt by using the `${contextPrompt}` variable in its own prompt.

To define a custom AI action that does not use the context prompt in any way, in the AI action's JSON configuration file, you can expand the context prompt to an empty value like this:

```
{
    "id": "improve.grammar.custom",
    "title": "..",
     ....
    "expand-params": [
        {
         "name": "contextPrompt",
         "value": ""
        }
    ]
}
```

expand-params

A JSON array that can define values for parameters that should be expanded in the context content.

You can define parameters that can be configured by the end user when hovering over the AI action in the UI.

Example:

```
{
  "id": "thesaurus",
  "title": "Thesaurus/Synonyms",
  "output-action": "replace",
  "prompt": "Provide a max of ${number} synonyms for:",
  "expand-params": [
    ...
    {
      "name": "number",
      "label": "Number of synonyms",
      "value": "10",
      "choice-type": "single-choice",
      "alternate-values": [
        "10",
        "20",
        "30"
      ]
```

```
    }
  ]
}
```

The following properties are available for a defined parameter:

name

The parameter name. It can be referenced multiple times with the syntax `${param_name}` in the `context` parameter or in the external prompt file referenced using `prompt_ref`.

label

The label that appears in the UI when the parameter's value is configured by the end user.

value

The value of the parameter. It can contain various editor variables that get automatically expanded.

Example:

```
{
        "id": "action.update.prolog",
        ...
        "prompt": "Update the DITA XML topic prolog with the current date
 ${currentDate}...",
        "expand-params": [
            {
                "name": "currentDate",
                "value": "${date(yyyy-MM-dd)}"
            }
        ]
    }
```

alternate-values

An optional array of alternate values that are available to be configured by the end user when hovering over the AI action.

choice-type

An optional choice type that defines how the end user can configure the alternate values (combo box, check boxes, radio buttons).

prefix

The optional prefix value to be used for each expansion of the parameter's value.

parameters

An array of parameters that can be used to specify an engine name, temperature, or the available function definitions. Some of the most commonly set parameters:

engine

The optional name of the engine to use with the AI action, no matter what the default AI engine model is configured to be.

Example:

```
{
    "id": "action.review.logical.mistakes",
    ....,
    "parameters": {
        "engine": {"name": "o1"}
    }
}
```

temperature

Temperature is an optional parameter that governs the randomness and thus the creativity of the responses. It is always a number between 0 (more deterministic) and 1 (more random). The default value is 0 and it should, in general, remain unspecified when your AI actions are intended for technical documentation that is precise in nature.

function_refs

An optional JSON array that describes which functions can be used when executing this action.

Example of an AI action that uses **Retrieval Augmented Generation (RAG)** functions to find related content in the project:

```
{
    "id": "find.add.related.links",
    ...
    "parameters": {
    "function_refs": [
        {"ref": "get_related_resources_overview_from_project"},
        {"ref": "get_content_for_document_url"}
    ]
}
}
```

Example of an AI action that uses the `get_content_for_document_url` function to retrieve content from a web site:

```
{
    "id": "action.get.oxygen.version.web.site",
    "output-action": "insert",
    "title": "Insert Latest Oxygen Version",
    "prompt": "Obtain the latest version of Oxygen from the
    'https://www.oxygenxml.com/xml_editor/whats_new.html' web site by calling the
 function
    'get_content_for_document_url' and return it.",
    "parameters": {
        "function_refs": [
            {"ref": "get_content_for_document_url"}
        ]
    }
  }
```

Important:
There are specific actions that can be called to invoke other AI actions from an AI action, thus creating an agenting mechanism where various parts of the work is done by various actions. An example can be found here.

input

Defines which part of the current document should be sent to be processed by the AI engine.

selection, document [default]

Prefers sending the selected content. If there is no selection, it selects and sends the entire document.

selection

Prefers sending the selected content. If no content is selected, either no content is sent or an error message is received (depending on the action's `output-action`).

document

Always sends the entire document for processing, regardless if there is content selected inside it.

none

Does not send any content from the current document. See this example.

input-type

Describes how the XML content from the document that is sent to the platform should be processed, as regular XML including markup tags or as plain text without the tags and with all DITA XML content references and key references expanded. By default, its value is assumed to be `markup`, which means the content is sent exactly as it is.

Example of an AI action that requires the input to be sent without any markup tags and with the DITA XML conrefs and keyrefs expanded:

```
{
    "id": "action.answer.questions",
    "output-action": "none",
    "description": "Generate answers to questions you ask based on the selected text or on the
    entire document content.",
    "input-type": "markup",
    ....
  }
```

output-action

Defines the type of change the AI action performs on the opened document. Some of the most commonly used type values are listed below:

replace

The most common type of action. Sends the selected content to the AI platform and proposes to replace the selection with the received response. If there is no selected content automatically, the entire document is selected when the action is invoked.

Example:

```
{
  "id" : "replace.bold.with.uicontrol",
  "title" : "Replace bold with uicontrol",
  "output-action" : "replace",
  "prompt" : "Replace the bold element with the uicontrol element (<uicontrol>)....",
}
```

insert

Inserts the content received from the AI engine in the current document at the cursor position or at an offset specified by the `insert-modes`. If the inserted content is invalid at the proposed offset, the application attempts to find a proper location for it.

Example:

```
{
    "id": "action.update.title.based.on.content",
    "title": "Update Title",
    "output-action": "insert",
    "prompt": "You are tasked with creating a title for the given text, summarizing it
 and
    focusing on its main points:"
}
```

none

Sends the selected content or the entire document to the AI platform and shows the response to the end user.

Example:

```
{
    "id": "action.review.logical.mistakes",
    "title": "Logical Inconsistencies",
    "output-action": "none",
```

```
    "prompt": "Find logical and inconsistency mistakes in the content and return a
 report.",
}
```

create-new-document

Shows a dialog box that asks for a context and then creates a new document based on the received server response. The `"additional_info"/"properties"/"file_extension"` property can be set to impose an extension for the newly created document.

Example:

```
{
    "id": "action.create.schematron.rules",
    "title": "Schematron Rules from Style Guide",
    "output-action": "create-new-document",
    "additional_info": {
        "properties": {"file_extension": "sch"}
    },
    "prompt_ref": "prompts/create-schematron-rules.md"
}
```

insert-modes

A JSON array that defines places for inserting the content in an XML document. The first mode for which the XPath condition becomes true is *run*. Such modes are used when the AI action `output-action` is `insert`.

Example:

```
{
    "id": "action.update.title.based.on.content",
    ...
    "output-action": "insert",
    ...
    "insert-modes": [
      {
        "xpath-condition": "(//title)[1]",
        "target-location": "(//title)[1]",
        "action": "replace"
      },
```

```
    {
      "target-location": "(//shortdesc)[1]",
      "action": "insert-before"
    }
  ]...
}
```

embed-assist

Set this optional boolean parameter to `true` to add the AI action to the top right **AI Positron Assistant** drop-down menu that appears when editing content in the **Author** visual editing mode.

quick-assist

Set this optional boolean parameter to `true` to add the AI action to the quick assist bulb drop-down menu that appears in the left vertical stripe.

Hope this helped, as always you can use the feedback form below if you want to ask more details about the possibilities of AI actions or want to brag about the custom AI actions you have created.

Overcoming Challenges - Editing DITA XML content with AI Tools (Presentation)

The DITA XML standard is recognized (for good reason) for its versatile reuse capabilities, which allows for creating content that is consistent and can be used to single-source the entire publication needs of a company.

AI editing tools can bring lots of extra benefits for technical documentation writers, capabilities for re-writing content to improve readability, grammar checking, content generation, generating reports, and much more.

How do these benefits interact with the DITA XML features (content reuse, specialization) and how can AI tools be developed to work better with these DITA XMl specific features?

The presentation below enumerates various useful AI-related actions and checks how they interact with DITA XML content, especially with DITA XML features such as specialization, content reuse, or filtering. It all considers how the AI actions can be made more DITA XML-aware to be better integrated with a particular organization's needs.

AI-Powered XML Editing (Presentation)

We have developed AI-based tools that we believe will greatly benefit technical documentation writers. These tools are designed to enhance the editing experience and provide AI-powered assistance for published documentation.

George Bina has created a presentation titled '**AI-Powered XML Editing**', which offers a comprehensive overview of the key AI-related features we've implemented in our application over recent years. I've included this presentation below for our clients and integrators to gain a better understanding of the **AI-Powered XML Editing** tools we offer.

Utilizing GenAI to Enhance Early-Stage Technical Content Creation (Presentation)

Generative Artificial Intelligence (GenAI) is a category of AI focused on **creating content** (text, images, videos, music, etc.), as opposed to traditional AI, which aims primarily at understanding or classifying data. And in the early-stages of writing technical documentation, GenAI can be used for at least two purposes:

1. Generating a documentation draft from scratch.
2. Updating the existing text content based on recent changes in a product.

I've created a presentation titled **Utilizing GenAI to Enhance Early-Stage Technical Content Creation** about how generative AI can enhance your productivity in the early-stage technical content creation, and I've included this presentation below for our clients and integrators to gain a better understanding of the **AI-Powered XML Editing** tools we offer.

Using ChatGPT in a DITA XML Project

The ChatGPT OpenAI bot has the potential to help with various technical documentation tasks (advice about the standard, small conversions, content generation, writing style, and readability analysis).

Below, you can see a list of prompts that I gave the bot along with the answers. The prompts to the chatbot are presented as codeblocks.

Advice About DITA XML Standard Use

1. Ask about the type of DITA links to use in a topic.

```
Should I use cross references or related links in my DITA XML topic?
```

Both cross-references and related links can be useful in DITA XML topics to help users navigate and find relevant information. However, there are some differences between the two that may make one more appropriate than the other depending on the context.

Cross-references are links within a document that point to another section within the same document or to another document altogether. They are typically used to reference specific information that is related to the current topic, such as a definition or an explanation of a concept that is introduced in the current section.

Related links, on the other hand, are links to external resources that may be useful to the user in understanding the current topic. These links may point to other documents or websites that provide additional information on the same topic, or they may point to related topics that the user may find interesting or helpful.

The choice between cross-references and related links depends on the specific use case and the goals of your documentation. If you want to help users quickly access related information within your documentation, cross-references may be more appropriate. If you want to provide users with additional resources or further reading on a topic, related links may be more appropriate.

In some cases, you may even want to use both cross-references and related links in the same topic to provide users with a variety of options for finding information. Ultimately, the key is to consider the user's needs and goals when deciding which

2. Ask information about a specific DITA XML element.

```
Can I use the DITA XML uicontrol element for company names?
```

No, the `uicontrol` element in DITA XML is specifically intended to be used for user interface controls, such as buttons, checkboxes, and other interactive elements in software applications or on websites. The `uicontrol` element is meant to visually distinguish these elements from surrounding text and to provide consistent styling across the document.

Using the `uicontrol` element for company names would not be appropriate, as it does not accurately represent the meaning of the content. Instead, you should use an appropriate element for company names, such as `orgname`, `companyname`, or `personname`, depending on the context of your content.

For example, if you are referring to a company name in your documentation, you could use the `companyname` element to indicate that the text is the name of a company. If you are referring to a person or group associated with the company, you could use the `personname` or `orgname` element, respectively.

Using the correct element for the content will ensure that your documentation is accurately structured and easy for users to navigate and understand.

Generate New Content

1. Instruct ChatGPT to generate DITA content about a very common task.

```
Write a small DITA XML task about installing a software application on Windows.
```

Sure, here's a small DITA XML task about installing a software application on Windows:

xml Copy code

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE task PUBLIC "-//OASIS//DTD DITA Task//EN" "task.dtd">
<task id="task001">
  <title>Installing a Software Application on Windows</title>
  <taskbody>
    <context>The following steps assume that you have downloaded the install
    <steps>
      <step>
        <cmd>Double-click the installation file to begin the installation pr
```

2. Generate a list of index terms from existing DITA XML content.

```
Create a DITA XML prolog element containing 5-6 index term elements related to the DITA XML topic below:

....
```

Here's an example of a DITA XML prolog element containing index term elements related to the topic:

```php
<prolog>
  <metadata>
    <keywords>
      <indexterm>Sharing Application Settings</indexterm>
      <indexterm>Oxygen</indexterm>
      <indexterm>Project configuration</indexterm>
      <indexterm>Default options</indexterm>
      <indexterm>Import and export preferences</indexterm>
      <indexterm>Java API</indexterm>
    </keywords>
  </metadata>
</prolog>
```

Simple Conversions

1. Convert Markdown content to DITA XML topic.

```
Convert this Markdown content to a DITA XML topic:


# Getting Started


List of steps:


- Install application

- Login

- Perform task
```

Here's the DITA XML equivalent of the Markdown content:

xml Copy code

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE topic PUBLIC "-//OASIS//DTD DITA Topic//EN" "topic.dtd">
<topic id="getting_started">
  <title>Getting Started</title>
  <body>
    <ol>
      <li>
        <p>Install the application.</p>
      </li>
      <li>
        <p>Login.</p>
      </li>
```

2. Convert CSV content to DITA XML table.

```
Convert this CSV content containing headers to a DITA XML table element:


Common name,Scientific name,Location,Temperament,Diet,Water,Size,Region of the Aquarium,Breeding

Compressiceps,Haplochromis compressiceps,Lake Tanganyika,Territorial,Omnivore,"PH 7.0 - 8.0, Temp. 73 - 77 F",5

 inches,Bottom,Hard

Frontosa,Cyphotilapia frontosus,Lake Tanganyika,Very peaceful,Omnivore,"PH 7.8 - 8.5, Temp 75F - 82F",14

 inches,Middle-Bottom,Medium
```

Here's the DITA XML table element for the given CSV content:

xml Copy code

```xml
<table>
  <tgroup cols="9">
    <colspec colname="col1" colnum="1" />
    <colspec colname="col2" colnum="2" />
    <colspec colname="col3" colnum="3" />
```

Writing Style

1. Ask what terminology to use in certain cases.

```
Should I use for technical documentation the phrase "click a button" or "press a button"?
```

Both "click a button" and "press a button" are commonly used phrases in technical documentation and have similar meanings. However, "click a button" is more commonly used in documentation related to software applications and web interfaces, while "press a button" is more commonly used in documentation related to hardware devices such as remote controls, keyboards, or gaming controllers.

Therefore, the choice of using "click a button" or "press a button" depends on the context of your documentation. If your documentation is related to software applications or web interfaces, it would be more appropriate to use "click a button." On the other hand, if your documentation is related to hardware devices, it would be more appropriate to use "press a button."

2. Re-phrase documentation provided by engineers using the Microsoft Style Guide style and also change the point of view from first-person to third-person.

```
Re-phrase the following content to adhere to the Microsoft Style Guide, third person:


The action Show only keys with closest relative key scope from DITA Reusable Components / Keys tab was also
 added to Media tab Settings drop down menu
The states of the two actions are synchronized. If you check it in Keys tab will automatically get checked in
 Media tab
```

The following is a rephrased version of the content in third person to adhere to the Microsoft Style Guide:

The "Show only keys with closest relative key scope" action was added to the Settings dropdown menu on the Media tab, located in the DITA Reusable Components / Keys tab.

Both actions' states are synchronized. If the user checks the "Show only keys with closest relative key scope" action in the Keys tab, it will also be checked automatically in the Media tab.

3. Translate some notes from the engineer's native language (in this case, Romanian with a mix of English words) to English, re-phrase using the Microsoft Style Guide style, and also change the point of view from first-person to third-person.

```
Translate to English, rephrase to adhere to the Microsoft Style Guide, third person:


In tabul Media, in meniul Settings a aparut actiunea Show only keys with closest relative key scope

Cand o bifam trebuie sa ramana in view doar cheile relative (cheile cu forma cea mai scurta).
```

The "Show only keys with closest relative key scope" action has appeared in the Settings menu on the Media tab. When selected, only the relative keys (those with the shortest form) should remain in view.

4. Re-write an entire DITA XML topic to increase its readability.

```
Re-write the following DITA XML topic preserving all its original XML elements to increase its readability:

<topic id="dita_oriented_tips_and_tricks">

 <title>DITA Oriented Tips And Tricks</title>

 <prolog>

  <author>Radu Coravu</author>

 </prolog>

 <body>

  <p><ph keyref="oxygen"/> has about 18 years of development under its wings. During these

   years, we added lots of functionality and many of our users usually do not know more than 20-30%

   of Oxygen's features. So this presentation is intended for technical writers using Oxygen to

   write DITA content and who may want to find out more about their tool.</p>

  <section id="section_gld_tbs_gfb">

   <title>Add-ons</title>

   <p>There are lots of free add-ons provided by the Oxygen team that can be installed in an

    <xref

     href="https://github.com/search?q=topic%3Aadd-on+org%3Aoxygenxml&amp;type=Repositories"

     format="html" scope="external">Oxygen standalone version</xref>. I will enumerate some of

    the most interesting one:</p>

   <ul id="ul_xbw_ccs_gfb">

    <li><xref href="https://github.com/oxygenxml/oxygen-git-plugin" format="html"

     scope="external">Git add-on</xref>. This plugin adds a side view allowing you to do some

     common Git operations like pull, commit, push (mostly 90% of what a regular Git user would

     do).</li>
```

```
    <li><xref href="https://www.oxygenxml.com/doc/ug-editor/topics/content-fusion-addon.html"
     format="html" scope="external">Content Fusion Connector add-on</xref>. Collaborate with
     your peers by sending them links that allow them to give you feedback on your work using a
     web browser without the need to pre-install anything on their side. </li>
   </ul>
  </section>
 </body>
</topic>
```

Result:

```
<topic id="dita_oriented_tips_and_tricks">
 <title>DITA Oriented Tips And Tricks</title>
 <prolog>
  <author>Radu Coravu</author>
 </prolog>
 <body>
  <p><ph keyref="oxygen"/> has been under development for about 18 years. During this time, we
   have added a lot of functionality, but many of our users are not aware of more than
   20-30% of Oxygen's features. This presentation is intended for technical writers who use
   Oxygen to write DITA content and want to learn more about the tool.</p>
  <section id="section_gld_tbs_gfb">
   <title>Add-ons</title>
   <p>Oxygen provides a range of free add-ons that can be installed in the <xref
     href="https://github.com/search?q=topic%3Aadd-on+org%3Aoxygenxml&amp;type=Repositories"
     format="html" scope="external">Oxygen standalone version</xref>. Here are some
    of the most interesting ones:</p>

   <ul id="ul_xbw_ccs_gfb">
    <li><xref href="https://github.com/oxygenxml/oxygen-git-plugin" format="html"
      scope="external">Git add-on</xref>: This plugin adds a side view that allows
     you to perform common Git operations like pull, commit, and push (similar to
     what a regular Git user would do).</li>
    <li><xref
      href="https://www.oxygenxml.com/doc/ug-editor/topics/content-fusion-addon.html"
      format="html" scope="external">Content Fusion Connector add-on</xref>:
     Collaborate with your peers by sending them links that allow them to give you
     feedback on your work using a web browser, without the need for them to
```

```
    pre-install anything on their side.</li>
   </ul>
  </section>
 </body>
</topic>
```

Remarks: If the prompt insists that all XML elements should be preserved, the re-write does few things, but if the prompt does not insist that all XML elements should be preserved, the inline elements and links are removed.

Statistics

Create readability analysis for a DITA XML topic.

```
Create a readability analysis for the following DITA XML topic:


<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE topic PUBLIC "-//OASIS//DTD DITA Topic//EN" "topic.dtd">
<topic id="topic_whr_nly_c5">
  <title>Sharing Application Settings</title><prolog><author>Radu Coravu</author></prolog>
  <body>
    <p>There are various ways in the <b>Oxygen</b> standalone version through which a team of writers
.........
```

The DITA XML topic "Sharing Application Settings" has an average readability level that may be difficult for some readers to understand. Here's a breakdown of the analysis:

- Flesch-Kincaid Grade Level: 10.1
- Flesch Reading Ease: 60.7
- Gunning Fog Index: 12.2
- Coleman-Liau Index: 11.7
- Automated Readability Index: 10.3

The average grade level for this topic is around 10.1, which means that a high school student or a person with some college education may understand the content. However, readers with lower education levels or those who are not familiar with technical terms may have difficulty comprehending the text.

The Flesch Reading Ease score of 60.7 indicates that the topic is moderately difficult to read, and readers may need to spend extra time understanding the content. The Gunning Fog Index of 12.2 suggests that the text is difficult to read and may require a higher level of education to comprehend. The Coleman-Liau Index of 11.7 indicates that the text is slightly difficult to read, while the Automated Readability Index of 10.3 suggests that the text is easy to read for individuals with some college education.

Overall, the topic "Sharing Application Settings" has an average readability level that may be difficult for some readers to understand, especially those who are not familiar with technical terms.

Custom Validation (Schematron-based)

1. Create an ISO Schematron validation check that counts words.

```
Create an ISO Schematron schema which reports if a DITA short description element has more than 20 words.
```

Here's an example ISO Schematron schema that checks if a DITA short description element has more than 20 words:

xml Copy code

```xml
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://purl.oclc.org/dsdl/schematron" queryBinding="xslt2">
  <ns prefix="dita" uri="http://www.dita.org/xsd/dita"/>

  <pattern>
    <title>Check if short description has more than 20 words</title>

    <rule context="dita:shortdesc">
      <assert test="count(tokenize(normalize-space(.), '\s+')) &lt;= 20">
        Short description should have no more than 20 words.
      </assert>
    </rule>
  </pattern>
</schema>
```

Regenerate response

Remark: The namespace declaration and usage is wrong (DITA elements are in no namespace). The assert test is correct.

2. Create an ISO Schematron validation check from a description.

```
Create an ISO Schematron validation rule which reports an error if a DITA XML image element does not contain a
 scale attribute
```

Here's an example ISO Schematron rule that checks if a DITA XML image element contains a `scale` attribute:

xml Copy code

```xml
<schema xmlns="http://purl.oclc.org/dsdl/schematron" queryBinding="xslt2">
  <ns prefix="dita" uri="http://www.dita.org/xsd/dita"/>

  <pattern>
    <title>Check for presence of scale attribute on image elements</title>

    <rule context="dita:image">
      <assert test="@scale">Image element must have a scale attribute.</asse
    </rule>
```

Remark: The namespace declaration and usage is wrong (DITA elements are in no namespace). The assert test is correct.

AI Positron Assistant - Retrospective Analysis

It's been more than one year since we started working on the Oxygen AI Positron Assistant Add-on *(on page 16)*, back when the **OpenAI GPT 3.5** AI engine was released, and we are currently on version 4.0.0 of the add-on. Our main direction (which we still keep) was to provide powerful tools based on AI engines for technical documentation writers (targeted mostly towards DITA XML users) and for our XSLT/Schematron developers. I've always envisioned the AI set of tools as an exoskeleton, as a way to add more power to a technical documentation writer's punch throughout the day. The table below contains a list of concepts (mostly add-on specific features or feature categories), each with a short description based on personal experience.

Eating our own dog food?

Are we using AI tools ourselves? Yes, I have started using AI tools more and more to improve readability and fix grammar or logical inconsistencies in articles like this one. I don't ask the AI to create new article ideas, but I do use AI tools to make the content I write easier to read and understand.

Usefulness

Are AI engines useful for a technical writer's work? Definitely yes. People might think they can just send the AI a bunch of content with a simple instruction like `Spell check this content` and the AI will always return the correct content. The AI engine gives meaningful answers when the prompt is detailed and precise. This is why we include many predefined actions with our add-on.

Platform availability

From the start, we wanted the add-on to work for both the desktop Oxygen application and the WebAuthor in-browser editor. We included core features like predefined actions, chat support, and the ability to create custom actions. This way, we can offer most of these features on both the desktop and the web application.

AI engine choice flexibility

Our initial offer was a **subscription**-based license using the Oxygen AI Positron Platform as a bridge to OpenAI. We noticed that some companies wanted a choice of which AI engine to use, so we created the **Oxygen AI Positron Enterprise** add-on. This add-on allows companies to use their own AI engine directly, with support for connectors to **OpenAI**, **Microsoft Azure OpenAI**, and **Anthropic Claude**. There are also proxy servers that can wrap many AI engines like **LLama** and **Mistral**, allowing them to be accessed using the OpenAI interface. Is the current flexibility we offer enough? Not in all cases. We plan to allow developers to create their own connectors to use their own APIs to access the engines.

So are all AI engines equal? Our predefined actions have and are actively tested with **OpenAI** engines and with engines provided by **Anthropic Claude**. We also tried to use engines like **Llama 3.1** or **Mistral** with our current predefined actions but the results are not fantastic, as the possibility to obtain broken **DITA XML** content with these open source engines is much higher.

Predefined actions

Over time, we improved how we write prompts for the many predefined actions that come with the add-on. For each action, we try to give a clear prompt with lots of details so the AI engine can produce the best results. We test our predefined actions using both manual and automatic tests. We got the best results with Open AI GPT-4 engines and the engines provided by **Anthropic Claude**, like **Sonnet**.

Custom actions

We wanted to give companies the flexibility to create their own custom actions using specific prompts to solve their unique problems. With the add-on, we provide a **JSON** schema that defines all aspects of implementing custom AI actions. The key to a successful action is to make it as precise and detailed as possible.

Content generation

There are two main ways AI-based content generation can be useful:

- Summarizing a large amount of text. For example, predefined actions like **Short Description** try to summarize a big piece of content. These actions usually work very well.
- Expanding a small amount of text. Content generation actions like **Add Structured Content** or **Generate Documentation Draft** use the AI engine to expand existing content and create valid DITA XML content. Since AI engines can make mistakes if they don't have enough details, we added support for **RAG** (retrieval augmented generation) functions. These functions use the existing project content to give the AI engine enough useful information to create new DITA XML content. A common use for a writer is to gather data from different sources and create a first draft of a DITA XML topic, which they can refine later. This is what we aim to provide with our predefined actions.

Vision support

Certain actions like **Generate Image Alternate Text**, **Generate Documentation Draft**, or **Update Content Based on Images** use the capabilities of engines like **GPT-4o**. These engines read content from images provided by the user and then use that information to write new DITA XML content or update existing content. In our experience, these kinds of actions are useful and work quite well to update DITA XML content based on application screenshots.

Rewrite actions

We created useful actions to **Correct Grammar**, **Improve Readability**, **Use Active Voice**, and **Improve Structure**. These actions are applied to existing content, they have precise prompts that instruct the AI to preserve the existing content as much as possible, and they are solid examples of how the AI can help a writer during their daily work.

Review actions

At some point, we had the idea that the AI, instead of fixing problems, could leave the original content as it is and just add comments around the content that needs to be changed. So, the **Proofread** action appeared as a way to allow the AI engine to give advice on existing content without making any changes. In a way, the AI engine becomes a reviewer of a first document draft. The **Resolve Comments** action appeared as a way to allow the AI engine to fix existing comments by changing the content.

Overview actions

We created some useful actions to **Summarize** or report on the **Readability** of a DITA XML topic. We also created actions to use AI to better understand what questions a certain topic answers. In the future, using an AI engine might also include asking the AI to find **logical inconsistencies** in already written content that may be too large for a person to fully digest easily.

Translate actions

There are studies that say the translation abilities in AI engines are better than **Google Translate** or **DeepL**. The available translation actions have clear prompts that tell the AI engine to skip certain DITA

XML tags like `<codeblock>` or `<codeph>`. In general, they seem to do a good job of keeping the existing DITA XML structure, including the parts of reusable content and key references.

Intelligent agents

In the most recent add-on release, we added new actions like **Split Topic** and **Create Topics**. These actions give the AI more freedom to process existing content and create new topics on disk. The AI uses a feedback loop where it first proposes a plan of action, evaluates how the content could be changed, and then makes the necessary changes. These actions try to use the AI engine's potential to the max by letting it plan and execute more complex tasks.

Reuse actions

DITA XML is about content reuse. So, how can the AI engine help with this? For now, we have created two predefined actions. These actions use functions to let the AI engine access already reusable components. One action replaces product names with existing key references. The other action retrieves the closest reusable component to the currently selected paragraph. But I think more can be done in this area in the future.

Chat

Using AI engines means accepting that an action might not always give the right content. We give you control to preview the changes suggested by the AI and then keep chatting with the AI to fine-tune the answer until it is correct.

Chatting with the AI can also be a useful way to learn how different concepts work. We added **RAG** (retrieval augmented generation) support in the chat window. This lets people search for content and get precise answers based on their own documentation project.

Specializations support

So, how does the AI work with DITA XML specializations that might have custom `DOCTYPE` declarations or custom DITA XML elements?

Most of the built-in actions should work well with DITA XML specializations. For example, actions that rewrite the content or translate it should keep the existing DITA XML elements in the selection unchanged. Actions like **New DITA Topic** create basic DITA XML topics. However, there are actions like **Add Structured Content** that expand the content of an already created DITA XML topic. This allows the user to first create the DITA XML topic based on their specialization before using the AI engine. Actions like **Improve Structure** add base DITA XML elements to the selected content, so they might not use all the specialized DITA XML elements.

Third Party CMS support

Both Oxygen desktop and in-browser editing tools can be integrated with various content management systems. Most AI predefined actions should work when Oxygen is integrated with a third-party CMS. However, actions that write content on disk (like actions from the **Intelligent agents** category) might not be able to create new topics on the CMS. Actions that are improved by using **RAG** (retrieval augmented generation) might also not work well since the project content is stored on the CMS. Oxygen desktop has an **Enable indexing for remote resources** checkbox in the **Open/Find Resource** preferences page. This allows our indexing engine to index content from the CMS. But the in-browser Web Author editing tool does not yet support indexing CMS-specific content.

I hope this article, based on my experience working on AI editing tools and with AI editing tools, will be useful to you. As always, you can leave feedback using the Oxygen Feedback form, which appears below this published article.

Providing content to AI engines using DITA OT plugins

As part of the DITA Open Toolkit Day 2025 conference in Copenhagen, I presented a couple of DITA Open Toolkit plugins I developed that can use AI or provide content to be used by the AI:

- DITA OT plugin to preprocess using AI to apply AI in the DITA OT pre-processing stage (for example, to translate content at publishing time).
- DITA OT plugin to apply AI validation rules on pre-processed content (with conrefs expanded) and create a report.
- DITA OT plugin to create content using AI from a Style Guide, fine-tuning JSONL content for the AI.
- DITA OT plugin to upload Markdown content to an AI Assistant vector store.
- DITA OT Summarize plugin to create a summary of an entire DITA map in markdown.

If you find any of these subjects interesting, the entire set of slides is attached below. You can also use the feedback form if you want to have discussions about any of these plugins.

What is New in Oxygen AI Positron Assistant 5.0

The new **Oxygen AI Positron Assistant 5.0** brings meaningful improvements to help content creators use AI more effectively in structured authoring.

One of the most useful updates is the ability to **generate or update files** like **XSLT**, **XSD**, **Schematron**, **JSON Schema**, and **DTD** just by providing a brief description. You can then review changes using the built-in **visual file comparison** tool.

For **DITA users**, the assistant now supports advanced **content transformation**, making it easy to convert between document types or restructure topics using AI suggestions.

Other key updates include:

- **Eclipse plugin support**
- The option to **exclude specific files** from AI processing
- **Callback functions** for validation or tool integration
- Support for attaching **Word, PowerPoint, and PDF** files as input
- Access to new reasoning models like **o3**, **o1**, **o4-mini**, and more
- Integration with **xAI Grok** and other custom AI services through specialized connectors

Additional refinements improve comment resolution, validation fixes, and allow referencing external resources or prompts.

You can see the new features in action in this overview video:

https://www.youtube.com/embed/DXYYYvCtjfI

Chapter 4. DITA

DITA XML Editing, Collaboration and Publishing Overview

Oxygen XML provides a wide range of features for editing, reusing, profiling, reviewing, and publishing DITA XML content. We strive to offer a complete editing, collaboration package which is still flexible enough to be customized for a variety of use cases.

Overview

You can find an overview of the application's DITA XML editing support here: Oxygen XML Editor DITA Editing solution strong points. *(on page 278)*

Here are some of the key features:

- Permanent user-based licensing.
- Available on multiple platforms.
- Comprehensive support for editing, reviewing, and publishing DITA XML content.
- Excellent technical support.

For answers to common questions about the overall solution, which companies can benefit from using it, and how it is typically purchased, please see: Oxygen XML As a Technical Documentation Solution FAQ *(on page 263)*.

General Advantages of Structured DITA XML Editing With Oxygen

- You own your content, the storage is on your own servers, privacy is insured.
- You are using a standard (DITA XML) for which lots of tools and publishing choices exist.
- Reuse content and easily update the reused content in multiple places.
- Reduce translation costs by reusing content.
- Lost of publishing format choices (WebHelp, PDF, Windows Help, Word, Markdown, etc).
- Enforce consistency through structured validation (custom Schematron rules, Quick Fixes).

DITA Maps Manager

The DITA Maps Manager view allows editing, validating, making project-wide changes and publishing.

- Create new topics and references to them either individually or in bulk by using the **Fast Create Topics** action.
- Project- wide Operations:

    - Validate and Check for Completeness. *(on page 289)*
    - Search references.
    - Rename, move, or delete existing topic references.
    - Map-wide XML refactoring actions.
    - Find/Replace in Files.
    - Spell Check in Files.
    - Open/Find Resources.

If you want to learn more about editing DITA XML with Oxygen, you can watch recordings of past webinars here: Resources for learning DITA with Oxygen *(on page 66)*.

Topic Editing

Oxygen XML Editor provides comprehensive support for editing DITA XML Topics. Here are some key features:

- Ability to change the Tags Visualization Mode.
- Access to a CSS Styles drop-down menu.
- Availability of a Content Completion Window.
- Helpful side views, including the **Outline**, **Attributes**, **Elements** views, and **DITA Reusable Components**.
- DITA-specific actions accessible through the Menu, Contextual Menu, and Toolbar, such as:
    - Inserting links.
    - Reusing content.
    - Editing Profiling Attributes.
    - Working with tables.
    - Inserting references to image and media resources.
    - Inserting lists, paragraphs, and applying bold, italic, or underline styling.
- Various methods to insert links and content references, including copy/paste and drag and drop.

For more information on editing DITA XML with Oxygen, you can watch past webinar recordings: Resources for learning DITA with Oxygen *(on page 66)*.

Content Reuse

The DITA XML standard offers a wide range of content reuse *(on page 150)* features. Oxygen allows you to harness the full potential of the DITA 1.3 standard:

- Reuse product names using key definitions with keywords.
- Reuse XML elements in multiple places.

- Reuse ranges of XML elements.
- Push reusable content into target topics.
- Reuse entire DITA XML topics in various parts of the DITA map.
- Use DITA 1.3 key scopes to reuse topics with variable content in different contexts.

Oxygen provides several user interface features to insert reusable content:

- **DITA Maps Manager** -> **Insert Key Definition with Keyword**
- DITA XML topic editing toolbar with options to **Reuse Content**.
- Using special paste to insert content references in topics.
- The **DITA Reusable Components** side view.
- Content Completion actions to insert reusable content (keyrefs and conrefs).

For more information on DITA XML content reuse with Oxygen, check out the past webinar recordings: Resources for learning DITA with Oxygen *(on page 66)*.

Profiling/Filters

Profiling capabilities make it possible to create many similar user manuals from the same DITA XML content.

With Oxygen, you can:

- Set values for the DITA profiling attributes. These can be easily managed to filter content in the published output.
- Change between profile sets. This lets you see how the edited content will look before publishing.

You can watch past webinar recordings that cover DITA XML profiling with Oxygen: Resources for learning DITA with Oxygen *(on page 66)*.

Project-wide Refactoring

- The **Main Files** feature in the Oxygen **Project** view enables powerful refactoring operations:
    - You can move or rename one or multiple files.
    - You can change directory names or move entire directories.
- You can apply XML refactoring actions to multiple files or entire folders.

Validation, Structure, and Terminology Checks

- The Validate and Check For Completeness *(on page 289)* tool available in the **DITA Maps Manager** view performs dozens of useful validation checks starting from a DITA map or from a DITA Open Toolkit project:
    - Validate each map and topic according to the standard schemas.
    - Validate each map and topic with a custom Schematron schema.
    - Check for broken links.
    - Check for profiling/filter problems.
    - and so much more....
- Schematron checks *(on page 291)* with **quick fixes** can be added to report structural problems.
- The built-in spell checker and the Terminology Checker *(on page 458)* add-on can be used to check for content-related problems.

Migrating from Other Formats

- You can use the smart paste feature to convert Microsoft Word or web content into DITA XML format when you paste it.
- Alternatively, you can use the Oxygen Batch Documents Converter add-on to convert various document formats *(on page 354)* including **Word**, **DocBook**, **Markdown**, **HTML** to **DITA XML**.
- You can also perform on-the-fly conversions for Markdown, OpenAPI, or Microsoft Word documents that are directly referenced in a DITA map.

Integrating Other File Types in Your DITA Map

You can integrate Markdown files directly in your DITA Map, allowing subject matter experts to contribute content.

You can also integrate other file formats like Word, Excel, OpenAPI, HTML directly in the DITA Map.

Framework Customizations

You can customize the overall DITA XML editing *(on page 407)* functionality in several ways:

- Add new actions to the DITA-specific menus and toolbars.
- Remove existing actions that you don't need.
- Customize the content completion window *(on page 419)* by adding or removing entries.
- Customize the visual editing *(on page 245)* experience by using your own custom CSS.
- Use your own new file templates *(on page 413)* to create new documents.

- Add custom validation stages by using custom schemas or Schematron checks *(on page 409)*.
- And there are many other customization options available!

Useful Add-ons

The Oxygen default add-ons repository provides a range of helpful add-ons. Some of the most notable ones include:

- **Git Client** add-on - Allows you to connect to Git repositories.
- **Content Fusion Connector** - Enables subject matter experts (SMEs) to review technical documentation content directly in a web browser.
- **Batch Documents Converter** - Converts documents from various formats such as **MS Word**, **DocBook**, **Markdown**, and **XHTML** to **DITA XML**.
- **Terminology Checker** - Checks for terminology-related issues in your content.
- **Fluenta DITA Translation Add-on** - Uses the Fluenta third-party library to create XLIFF exports from DITA maps and import DITA maps from translated XLIFF files.
- **Oxygen AI Positron Assistant** - Provides AI-powered assistance for content creation and editing.
- **Oxygen Feedback Connector** - Provides ways to see and change (delete, accept, reject) comments offered in the Oxygen Feedback system to writers working in the desktop application.

Storage

Possibilities to store the DITA XML content:

- Using a Git project. *(on page 375)*.
- Subversion project.
- Sharepoint.
- WebDav or eXist DB.
- Using a commercial available CMS.
- Building your own CMS integration plugin.

Review

There are several ways for people to provide feedback on technical documentation:

- Internally review the content directly in the Oxygen desktop application:
    - Use change tracking to modify existing content or to accept/reject changes.
    - Use the comments feature to add comments, reply to comments, or resolve them.
- Review content internally using the Content Fusion platform.

- Review content internally or externally by publishing it to **WebHelp Responsive** output and using the Feedback platform.
- Review content internally or externally by publishing it to **WebHelp Responsive** output or **PDF** and using edit links.

Content Fusion Collaboration Platform (Web Browser)

- Installation as a server component in the client's own network.
- Live demo web site to test and explore: https://fusion.oxygenxml.com.
- Create and upload review tasks for subject matter experts.
- Connect to Git projects:
    - Allow creating review tasks.
    - Edit DITA XML content in your personal workspace.
    - Edit other types of documents.
    - Create publications and publish to WebHelp and PDF outputs.
    - Use connectors to push publications to live web sites.

Translate

Oxygen provides support for:

- Exporting a DITA map as a zip file from the **DITA Maps Manager** view, which can be sent to a translation agency.
- Using the **Fluenta DITA Translation Add-on**, a third-party library, to generate XLiff exports from DITA maps and import DITA maps from translated XLiff files.

For a more detailed explanation of the translation process, please refer to the following article: Translating your DITA Project *(on page 287)*

Publishing

Oxygen comes with a built-in **Oxygen Publishing Engine** that is based on the **DITA Open Toolkit** open-source project. This engine allows you to publish DITA XML content to various outputs such as **WebHelp Responsive**, **PDF (CSS customization based)**, **PDF (XSL-FO customization based)**, MS Word, EPUB, Windows Help, Markdown, and more.

You can customize the **WebHelp Responsive** and **PDF (CSS customization based)** outputs by creating publishing templates in two ways:

- By using the Oxygen Styles Basket Web Site.
- By following the user guide articles and samples to customize the CSS files: Resources for customizing the DITA output formats *(on page 67)*.

Publishing can also be configured to run server side *(on page 179)* by using the Oxygen Publishing Engine.

For more information about publishing customizations, see the following articles: Publishing Customizations *(on page 178)*.

Related information

DITA XML Editing, Collaboration and Publishing Overview - webinar recording an XML Prague.

DITA XML Editing, Collaboration and Publishing Overview - webinar recording an Tekom.

Learning

There are lots of resources available for learning to work with the DITA XML standard both online and in on Oxygen XML Editor user's manual and blog.

Resources for learning DITA with Oxygen

From time to time we get requests from beginners or from users migrating from other tools who want to start using Oxygen with DITA and they need to know a set of useful resources.

Resources for editing DITA with Oxygen:

We have a getting started section in our user's manual: https://www.oxygenxml.com/doc/ug-editor/topics/eppo-first-dita-topic.html and a larger section on DITA authoring: https://www.oxygenxml.com/doc/ug-editor/topics/author-dita.html.

We have a list of videos, some of them DITA-related here: https://www.oxygenxml.com/videos.html and a series of past webinar recordings about editing DITA with Oxygen.

Getting Started with Oxygen and DITA

https://www.oxygenxml.com/events/2020/webinar_getting_started_with_dita_using_oxygen_xml_editor.html

Basic DITA Profiling and Reuse

https://www.oxygenxml.com/events/2020/webinar_working_with_dita_in_oxygen.html

Advanced DITA Profiling and Reuse

https://www.oxygenxml.com/events/2021/webinar_working_with_dita_in_oxygen_advanced_profiling_and_reuse_strategies.html

Working with DITA in Oxygen - Customizing the Editing Experience

https://www.oxygenxml.com/events/2021/webinar_working_with_dita_in_oxygen_customizing_the_editing_experience.html

Working with DITA in Oxygen - Quick start with the DITA Startup Project

https://www.oxygenxml.com/events/2021/webinar_working_with_dita_in_oxygen_startup_project.html

Working with DITA in Oxygen - Migrating to DITA and Refactoring

https://www.oxygenxml.com/events/2021/webinar_working_with_dita_in_oxygen_migrating_to_dita_and_refactoring.html

DITA XML Editing, Collaboration and Publishing Overview

https://www.oxygenxml.com/events/2024/oxygen_user_meetup_Prague.html#video_7

All these webinars above are also listed here: https://www.oxygenxml.com/working_with_dita_in_oxygen.html.

Resources for learning DITA:

If you want to start learning about DITA in general there is a web site called Learning DITA.

DITA Introduction articles by Stefan Jung.

The free Oxygen Live Tutorials add-on contains various small tutorials for learning DITA.

The DITA 1.3 standard specification can be found here: https://www.oxygenxml.com/dita/1.3/specs/#introduction/dita-release-overview.html.

There are also a number of good books like DITA For Practitioners and the DITA Style Guide.

Resources for understanding and working with DITA XML by ACM.

List of resources for learning and experimenting with DITA content by Yedda Wang

Resources for customizing the DITA output formats

Usually customizing the XHTML based outputs means creating your custom CSS selectors. If you generate WebHelp output using Oxygen, we have a section explaining basic WebHelp customizations.

For PDF-based outputs we recommend, support and maintain our CSS-based PDF publishing which is free to use from Oxygen XML Editor. There are also a number of different choices to obtain PDF from DITA: Possibilities to obtain PDF from DITA *(on page 191)*.

We have a series of past webinar recordings about customizing the published output:

- Transforming DITA documents to PDF using CSS
  - Part 1 – Page Definitions, Cover Page and PDF Metadata
  - Part 2 – Book design, Pagination, Page Layout and Bookmarks
  - Part 3 – Advanced Fonts Usage
  - Part 4 – Advanced CSS Rules
- Techniques for Customizing the WebHelp Responsive Output
- Using Oxygen Styles Basket to Create CSS Customization from Scratch

And if you are interested in server side building and publishing this topic might be useful:Server Side Building and Publishing *(on page 179)*.

Courses for learning DITA with Oxygen XML Editor

- **Comtech Services**: https://comtechservices.com/training/. Comtech has three training classes that use Oxygen exclusively: DITA Basics, Advanced Reuse Strategies, Publishing for DITA.
- **Ryffine**: https://www.ryffine.com/.
- **Udemy**: https://www.udemy.com/course/technical-writing-how-to-write-using-dita-xml/.
- Basic principles and theory of DITA, 10 hour recorded training course by **Tony Self**: https://firehead-training.net/course/dita-concepts.
- Mekon: http://mekon.com/news/dita-authoring-oxygen-author/
- French course by **Tireme**: http://dita.tireme.fr/#definition/m2_session_adapter_dita.html
- "Writing DITA content" - course with a two day duration by **FlowTime**: https://www.flowtime.be/en/writing-dita-xml-content-schrijven.
- "Empowering Structured Authoring with oXygen XML" - **Metapercept** provides advanced DITA-XML and oXygen XML training programs tailored for professionals seeking to deepen their expertise in structured content authoring, semantic tagging, and multichannel publishing. Their offerings include self-paced modules and instructor-led sessions with real-world, enterprise-aligned use cases. Through the DITA Authoring with oXygen XML course on Metapercept Academy, learners progress beyond the basics to master advanced concepts such as specialization, content reuse strategies, key-based linking, conditional profiling, and integration with custom DITA-OT publishing pipelines.

DITA Trivia

There are a number of blogs on which you can read various DITA-related articles:

- Oxygen XML Blog: Welcome! *(on page 8)*.
- Magda Caloian's Think DITA blog: http://think-dita.com/.
- Scriptorium Blog: http://www.scriptorium.com/blog/.
- Eliot Kimber's Blog: http://drmacros-xml-rants.blogspot.ro/.
- A larger selection of technical writing and DITA articles:http://idratherbewriting.com
- List of learning DITA blog posts from Mekon: http://mekon.com/bitesize-dita/
- DITA XML Benefits: https://technicallywewrite.com/2023/09/11/aboutdita
- Working with DITA projects in Oxygen: https://technicallywewrite.com/2023/09/14/aboutoxygen

Community

The DITA Users List is probably the first place where you can register and ask for help with DITA-related issues. A bunch of other community support forums for the publishing engine are also listed here: https://www.dita-ot.org/support.

There is also a DITA Awareness Group on Linked In.

Learn DITA Editing with Oxygen with Small Clips

Short Clips - Insert Image References

Learn all ways to insert image references in DITA topics using small clips.

- Use Toolbar Action to Insert a Figure with an Image Reference *(on page 69)*
- Drag and Drop to Insert an Image Reference *(on page 70)*
- Copy/Paste to Insert an Image Reference *(on page 70)*
- Configure DITA Reusable Components/Media Tab Working Sets and Insert an Image Reference *(on page 71)*

Use Toolbar Action to Insert a Figure with an Image Reference

Drag and Drop to Insert an Image Reference

Copy/Paste to Insert an Image Reference

Configure DITA Reusable Components/Media Tab Working Sets and Insert an Image Reference

Short Clips - Insert Links

Learn all ways to insert links in DITA topics using small clips.

- Use Toolbar Action to Insert a Cross Reference to Another Topic *(on page 72)*
- Use Toolbar Action to Insert a Cross Reference to a Table *(on page 72)*
- Drag and Drop to Insert a Cross Reference *(on page 73)*
- Copy/Paste to Insert a Cross Reference *(on page 73)*
- Copy Paste to Insert a Cross Reference To a Table *(on page 74)*
- Use Toolbar Action to Insert a Related Link to Another Topic *(on page 74)*
- Use Toolbar Action to Insert a Link to a Web Site *(on page 75)*
- Use Toolbar Action to Insert a Link to a PDF *(on page 75)*

Use Toolbar Action to Insert a Cross Reference to Another Topic

Use Toolbar Action to Insert a Cross Reference to a Table

Drag and Drop to Insert a Cross Reference

Copy/Paste to Insert a Cross Reference

Copy Paste to Insert a Cross Reference To a Table

Use Toolbar Action to Insert a Related Link to Another Topic

Use Toolbar Action to Insert a Link to a Web Site

Use Toolbar Action to Insert a Link to a PDF

Short Clips - Reuse Content

Define a Key for a Product Name and Use It

Use an Already Defined Key for a Product Name Using The Toolbar Action

Add a Content Reference to a Reusable Note Using the Toolbar Actions

Add a Content Reference Using Copy/Special Paste Operations

Add a Content Reference Using the DITA Reusable Components View

DITA Perspectives

DITA is not defined as a flat list of elements, but each element is either a base element or it derives as a specialized version of another element. This hierarchy should actually decrease the cognitive complexity of a vocabulary because it allows you to find an element faster than working with a flat list.

In this article, I want to show the hierarchy of elements in DITA and then explore how you can take advantage of this in understanding the DITA architecture, learning DITA, document authoring, etc.

All the scripts I used to generate these diagrams can be found in the DITA Perspectives Github Project.

Overview of DITA

DITA Shells

DITA 1.3 defines multiple types of documents. Along with the generic topic and map, there are also many specialized topics types and maps types. Each type of document is defined by a schema that is marked as a `shell` schema, a schema that should be an entry point, the one that should be referred from an XML document. All the defined shells are presented in the following diagram, grouped by the folder they are defined in:

DITA Modules

The DITA specification mentions that each element has a `class` attribute that encodes information about the type of that element, for example:

```
<step class="- topic/li task/step">
```

and this information includes:

> "A sequence of one or more tokens of the form `"modulename/typename"`, with each token separated by one or more spaces, where `modulename` is the short name of the vocabulary module and `typename` is the element type name. Tokens are ordered left to right from most general to most specialized."

By analyzing all the class values from all the elements defined in the schemas, I can identify the base modules (top level ones). In this case, `topic` and a hierarchy implied by how the elements are defined

by this relation from more general to more specialized modules, thus identifying how modules are specialized from others (in this example, `task` is specialized from `topic`).

The base DITA modules:

The modules specialization hierarchy:

DITA Elements

The total number of DITA elements defined in the schemas and how they are split into base elements, specialized element, and a highlight of the duplicate elements, while having the same name but being defined on a different module can be observed here:

Elements Split by Map and Topic Base

Topic Elements

The element information is projected on the `topic` base module:

Map Elements

The element information is projected on the `map` base module:

bookmap

Defined in ../data/rng/bookmap/rng/bookmapMod.rng

learningSummary

Defined in ../data/rng/learning/rng/learningSummaryMod.rng

learningPlan

Defined in ../data/rng/learning/rng/learningPlanMod.rng

learningOverview

Defined in ../data/rng/learning/rng/learningOverviewMod.rng

learningContent

Defined in ../data/rng/learning/rng/learningContentMod.rng

learningObjectMap

Defined in ../data/rng/learning/rng/learningObjectMapMod.rng

learningBase

Defined in ../data/rng/learning/rng/learningBaseMod.rng

learningGroupMap

Defined in ../data/rng/learning/rng/learningGroupMapMod.rng

learningAssessment

Defined in ../data/rng/learning/rng/learningAssessmentMod.rng

troubleshooting

Defined in ../data/rng/technicalContent/rng/troubleshootingMod.rng

reference

Defined in ../data/rng/technicalContent/rng/referenceMod.rng

task

Defined in ../data/rng/technicalContent/rng/taskMod.rng

glossentry

Defined in ../data/rng/technicalContent/rng/glossentryMod.rng

glossgroup

Defined in ../data/rng/technicalContent/rng/glossgroupMod.rng

concept

Defined in ../data/rng/technicalContent/rng/conceptMod.rng

subjectScheme

Defined in ../data/rng/subjectScheme/rng/subjectSchemeMod.rng

DITA Domains

Some DITA elements are defined to be part of a domain that can be added on any DITA document type as a pluggable component. A domain defines a number of semantic elements that are derived from other elements, providing a specialization of the base elements but reflecting the semantics of a domain, such as programming, software, etc.

DITA Learning Interaction Base 2 Domain

DITA Learning Map Domain

DITA Learning Interaction Base Domain

DITA Learning Metadata Domain

DITA Learning Domain

DITA Learning 2 Domain

DITA Abbreviated Form Domain

DITA Markup Name Mention Next v Domain

DITA MathML Domain

Added locally

DITA SVG

DITA User ADD Domain

DITA Equation Domain

DITA Task Requirements Domain

DITA Programming Domain

DITA XML Construct Domain

DITA Glossary Reference Domain

DITA Software Domain

DITA Release Management Domain

DITA XNAL Domain

DITAVAL Reference Domain

DITA Delay Resolution Domain

DITA Indexing Domain

DITA Hazard Statement Domain

DITA Highlight Domain

DITA Map Group Domain

DITA Utilities Domain

DITA Subject Classification Domain

Taking Advantage of DITA Element Hierarchy

Elements in DITA are defined not as a flat list of elements, but instead as a hierarchy, similar to a type hierarchy, and elements are either base elements or they are derived from another element. If we look at the analogy with a type system, the base elements are similar to the primitive types and the other elements are equivalent to derived types.

This is realized using values specified in the **class** attribute. These values specify the category and the name of the current element and (if we talk about a derived element) also the name and category of its parent, as well as all the ancestors up to the base element.

For example, the `@class` value for the `<lcPlanTitle>` element is:

```
class="- topic/fig learningBase/fig learningPlan/lcPlanTitle"
```

This means that the current element is `<lcPlanTitle>` from the `learningPlan` category and this is derived from the `<fig>` element from the `learningBase` category, which in turn is defined from the `<fig>` element from the `topic` category.

We can take this information into account in some situations.

Element Selection During Editing

To take advantage of the fact that elements in DITA form a hierarchy, one possibility is to follow this hierarchy when we present the user with the choice of an element to insert, so instead of presenting a flat list of choices, we can organize the elements according to the hierarchy defined by the `@class` attribute values and thus, if the user selects an unordered list (`<ul>` element) to insert (for example), we can further present all the elements specialized from the `<ul>` element that are valid in that position in the document. Of course, this makes sense if the user is browsing for an element to insert, and the user already knows the element and types its name, then we can just filter that name and eventually, if there are elements specializing the one identified by the entered string, then we can show them further.

Another possibility is to present a drop-down for an element that has specialized elements, and those are valid at that location, and allow the user to move to a more specialized element by selecting it from that drop-down.

Automatic Markup Detection

If we first detect the base element and then determine if it is in fact a more specialized one, it may help because it reduces the number of choices.

Soft Generalization

To visualize how a document will look like when it is generalized but without really generalizing it (that is without replacing the elements with more general ones), we implemented a CSS rendering to show the XML tags as if the document is generalized.

For example, an element on which we make the tags visible using CSS static content placed before and after the element looks like this:

If we generalize on level, showing the first derived element type, then this sample `<xmlelement>` will turn into `<markupname>`:

If we generalize to show the base elements, then many of the tag names will change, as highlighted in the following screen shot:

Exchanging DITA Documents

One advantage of using DITA is that we can tag at semantic level. For this to work, we need to create a specialization that exactly reflects the concepts the users of that specialization are familiar with. If one tries to layer its actual needs over an existing specialization (or one of the standard schemas), these may not exactly match the concepts of the audience and thus it will look like DITA is not the best fit.

However, the problem with a specialization is how do we exchange it with other parties? We need to also provide them with our specialization, and if they do not have our specialization, then what happens?

One approach is to export the DITA content as generalized content. Then import such a generalized package into an existing DITA installation by specializing as much as possible, taking into account what is available on the target DITA installation.

How can this be achieved?

From the class of the root element, we can identify the possible specialized root elements and check if we have them defined in a schema in the target DITA installation. Once we choose a root element, then we need to analyze the domain attributes of the root element and see what domains are defined in the target topic type and then specialize elements from those domains back. Otherwise, if a domain is not present in the target schema, we need to leave the base element.

We have support for generalization, which is relatively easy to implement, and we can probably also obtain the export package in a generalized form. However, we are missing a tool to take a generalized package, analyze a current DITA installation, and specialize everything so that it matches what is available in that specific DITA installation.

Using DITA to Document a Software Product

Besides working on an XML editing tool with lots of **DITA** editing functionality we also use DITA internally for editing the **Oxygen User's Guide**.

In this article I will try to give you an overview of our entire workflow as evidence that **DITA** does work and that it can be used and implemented without expensive solutions.

First, here's an overview of our needs:

- **Online Help** which is available as **WebHelp Responsive** on our website. Our WebHelp Feedback integration allows users to add comments and ratings to each topic. Those comments can later be used by us to rephrase and improve our documentation.
- **PDF** containing the entire contents of the user's manual. The PDF output is styled using CSS and obtained from DITA content using our Oxygen Publishing Engine. Nowadays, most of our users use the online **WebHelp** because it can be used much easier for finding certain topics. So, in our case, at least the **PDF** output is not popular anymore among users.
- **Offline Help** which is available inside the installed application. Oxygen is a multi-platform application, so we need to generate both **HTML Help** (CHM) for **Windows** and **JavaHelp** for the **Mac OSX** and **Linux** installations. Also, for the **Oxygen Eclipse Plugin**, we need to generate **Eclipse Help**.

We have two main distributions (**Standalone** and **Eclipse plugin**) and three main products (**Editor**, **Developer** and **Author**). We also have a web editing tool, WebHelp generation plugin and publishing engines. So we need to produce more than 9 different publications from the same **DITA** content depending on the shipped product.

And here's an overview of the tools we use:

Git as a Version Control System

We store our DITA content in a private **GitHub** repository and we also made a public **GitHub** repository containing a copy of our user manual's DITA content:https://github.com/oxygenxml/userguide. We use the free Oxygen Git client add-on for collaborating on the project. Other external Git clients like **SourceTree** or **Fork** can also be used to check out, edit and push changes to the Git repository. Our editing workflow is similar to what is described in this past webinar.

Oxygen XML Author

This may not come as a surprise but we use our own product to edit **DITA** content, partly because it's ours and partly because it is a very good tool. During the last couple of years this has been a good opportunity to improve our product based on our own feedback (feedback coming from our technical writers).

Oxygen is used by the technical writers to write **DITA** content but we also have colleagues who review content and propose changes directly from the web browser.

Oxygen Content Fusion

The Content Fusion add-on for XML Author allows uploading the edited content to a cloud server and providing a web editing tool for people to provide feedback and review the DITA content directly in a web browser: https://www.oxygenxml.com/content_fusion.html.

Oxygen Publishing Engine

We use the Oxygen Publishing Engine (based on the DITA Open Toolkit) to publish DITA content to the outputs we are interested in, mostly WebHelp Responsive, PDF and Windows Help.

Jenkins integration server

We have an automated script which builds all the user manual outputs every night.

Automated DITA Content Validation

There is a Validate and Check for Completeness script which runs on a test server and does various types of checks on the **DITA** content (including checks for validation, filtering problems, broken links, etc).

Atlassian Jira for Ticketing and Workflow

We use **Atlassian Jira** to provide a workflow both for the issues which are related directly to our software product and for the issues which are related exclusively with our user's manual. The JIRA is integrated with both our **SVN** and **GIT** repositories so it shows for a certain issue all resources which have been modified to fix it.

Guided DITA Authoring Solution Overview

We have some past blog posts about how Oxygen can be used to impose various editing behaviors for your team. In this one, will try to bring all of these solutions together in a comprehensive overview.

Learning to Work with DITA and Oxygen

You can find useful links for learning how to edit DITA using Oxygen in this previous blog post: Resources for learning DITA with Oxygen *(on page 66)*.

Migrating to DITA

There are multiple reasons why you would want to migrate from unstructured content to structured: Migrating to a Structured Standards-based Documentation Solution *(on page 352)*.

This older blog post details some possibilities of migrating Word documents to DITA: How to Migrate from Word to DITA *(on page 357)*.

You also have ways to migrate from XML-based standards (like DocBook or XHTML to DITA) using a set of predefined transformation scenarios.

Restricting the Visual Editing Experience

The entire visual editing experience in Oxygen's **Author** editing mode is driven by **CSS**. Oxygen has support for defining various CSS layers that can be applied when editing DITA content. For example, if you choose to create a Lightweight DITA topic in Oxygen, it has a special editing layer that contributes a combination of buttons, hints, and form controls designed to assist and guide the author. The following blog post details how a custom CSS that will be used to enhance the visual editing experience can be created and shared with others: Customizing the DITA Visual Editing Experience *(on page 245)*.

Implementing Your own Style Guide

Suppose you are a team of technical writers collaborating on a DITA-based project and you have your own various best practices in regards to which elements to use and when to use them. So, at some point you gather a set of HTML resources that explain how various DITA elements should be used, you store them on an internal server, and you want all your team members to have access to that set of HTML resources directly from Oxygen. This blog post provides more details and useful links to help you get started: Implementing your own Style Guide *(on page 294)*.

Imposing Controlled Attribute Values

If you want to impose DITA attribute values that need to be set for profiling or general use, this blog post should cover all you need to know about this: Controlled Attribute Values for your DITA Project *(on page 250)*.

Imposing Business Rules and Structure Restrictions to the DITA Content

In most cases, instead of relying on people to memorize numerous internal documentation style rules, you can convert many of these rules to Schematron and allow the application to automatically signal the content author when a rule is violated. You can also add quick fixes to provide authors various ways to rectify the problem. This blog post contains more details about this: Schematron Checks to help Technical Writing *(on page 291)*.

The DITA framework can be extended to add new Schematron rules: Sharing Schematron Validation Rules *(on page 409)*.

Running Batch Validation Checks on all of Your DITA Content

The **Validate and Check For Completeness** tool available in the **DITA Maps Manager** view performs a lot of different consistency checks on all your DITA topics. It can also be used to apply **Schematron** business rules on all of your topics: DITA Map Validate and Check for Completeness Overview *(on page 289)*.

Sharing DITA Editing Customizations with Your Team

Most of the custom editing behaviors, toolbar, and menu buttons that are available when editing DITA content are defined in the **DITA** framework configuration. A framework configuration's general anatomy is described here: The Oxygen SDK (Part 2: Frameworks) *(on page 425)*.

The framework configuration can be shared with all of your team members. For example, here is a way to restrict team members from using certain DITA elements: Document Type Extension Sharing *(on page 407)*.

Furthermore, here is a way to distribute new DITA file templates to your team: Sharing New Custom File Templates for a Specific Vocabulary *(on page 413)*.

Sharing Global Application Settings with Your Team

Suppose you want all of your team members to enable the automatic spell checker when writing documentation, or you want all of them to use a custom term dictionary or a custom set of learned words. This older blog post offers some hints about how global Oxygen settings can be distributed to your team members: Sharing Application Settings *(on page 464)*.

Collaboration, Content Management, and Version Tracking

All major Content Management Systems (CMSs) have plugins that can be installed in Oxygen to provide access to the CMS: https://www.oxygenxml.com/partners.html#cmssolutionpartners.

Even if you lack the funds to buy a commercial CMS, there are still plenty of open source version tracking solutions that provide the possibility of collaboration on a single DITA project: Collaboration (Teams working on a common XML project) *(on page 466)*. For example, the Oxygen User's Manual is written in **DITA** and we use a **GitHub** private repository to collaborate on it: Collaboration for Documenting a Software Product using DITA *(on page 465)*.

Allowing Subject Matter Experts to Review Content

Many technical writers are interested in having their content reviewed by the subject matter experts who are directly involved in building the tools. Oxygen has support for change tracking and adding comments directly in the edited content. Subject matter experts do not necessarily need to have the standalone version of Oxygen installed. The Oxygen Web Author is an online editing and reviewing solution that allows them to add comments and propose changes directly in the DITA content by using any device with a web browser (laptop, tablet, phone): https://www.oxygenxml.com/xml_web_author.html.

I hope this overview will help you to implement a complete guided authoring experience using Oxygen. As usual, if you have any questions or suggestions, they are welcome.

Maintaining a Healthy DITA Project

How should a Happy and Healthy Documentation Project Look Like?

Do you have a Healthy Project?

- Focus on writing and not on workflow.
- Involve peer-reviews, SMEs and end users.
- Easy start for first-time contributors.
- Easier produce deliverables and correct errors in older deliverables.
- Allow for future evolution: more writers, more outputs, more content, more products.

Why do big companies use the DITA standard?

- Standard means owning your content and no vendor lock-in (editing or publishing).
- DITA works very well with topic-based authoring.

- Lots of content reuse potential.
- Reuse lowers translation costs.

DITA Doc Project Aspects

- Storage
- Workflow
- Collaboration
    - Sharing Common Constraints (editing, validation, spell check dictionaries, ...)
- Structure
    - Managing links and reusable content
- Translation
- Publishing (Producing Deliverables)

Storage

- Commercial content management systems (CMS).
- Open Source version control systems: Git, Subversion, CVS

Version Control

- Ability to Tag Releases and Create Branches.
- See history for resources.

Working with the storage system

- Commercial CMSs – Remote editing, locking.
- Open Source version control systems – Local working copies, no editing restrictions → conflicts.

  Hint: Maybe you can use the same storage system as software developers in your company.

Collaboration and Workflow

- We invest a lot of time each day collaborating with our team or external collaborators.
- Collaboration should be as comfortable as possible.

Workflow

Issue tracking

- Using workflow features in the CMS
- Using issue management systems like Bugzilla, Atlassian JIRA or Trello.
    - **Tip:** Linking the product development with the documentation development.

Issue Tracking Examples

Custom workflows

- Documentation task specific workflow
- Integrate QA and documentation in software development process

Issue tracking – Simple Documentation Workflow

Issue tracking – Development and Documentation Workflow

Issue Tracking and Storage Integration

Issue Tracker can provide a single place where you can monitor a ticket from start to end, including:

Issue description and details

- Who worked on that issue
- What was changed in the application
- What was changed in the documentation
- Who should be notified when issue is resolved.

Involving Subject Matter Experts

- SMEs provide original content (DITA or Markdown or ...)
- Let SMEs review the published output.
    - HTML with feedback forms
    - PDF with comments.
    - Formal review with web editing tool integrated with storage system.
    - Informal review DITA content using change tracking and comment capabilities

How can end users collaborate with us?

- Send feedback via email/forum/phone.
- Send feedback in the published HTML output.
- Give feedback using an online DITA editing tool with comment-only capabilities.

Contribution Consistency

Sharing common settings between writers

- Custom style guide.
- Specific editing enhancements.
- Specific validation settings.
- Controlled attribute values.
- Custom spell and auto-correct dictionaries.
- Various other common preferences.

Custom Style Guide

The style guide is internal documentation about how to write documentation.

How can we remember what's written in the style guide?

- Searchable help output from internal style guide.
- Find an automatic way to impose style guide rules when editing.

Automating Style Guide Rules

Schematron Checks to help Technical Writing *(on page 291)*

- Schematron to add custom validation rules.
- Schematron Quick Fixes to propose quick fixes for each custom error message.

Using the same terminology rules

Checking Terminology with Oxygen XML Editor *(on page 458)*

- Custom Spell dictionaries.
- Custom auto-correct mappings.
- Advanced terminology checkers like Acrolinx, HyperSTE or LanguageTools.
- Building your own terminology checker using Schematron.

DITA Project Structure

- Organizing various resources in various folders
- Some CMSs may not consider this relevant.

File and folder naming/organization conventions

- By type:
  - Tasks/t_installation.dita
  - Concepts/c_profiler.dita
- By semantics:
  - xslt_debugger/backmapping.dita

DITA Style Guide

Managing Content Reuse

DITA Reuse Strategies *(on page 150)*

- Separate folders containing reusable content.
- Keep dictionaries of reusable components
- Prefer indirect references (conkeyrefs)

Managing Links

DITA Linking Strategies *(on page 139)*

- Prefer indirect links (key references)
- Reuse link targets
- Re-direct links depending on publication
- Use relationship tables

Project-wide refactor operations

- Convert between various topic types.
- Rename or move one or more topics.
- Change XML structure in topics from the entire project.
    - Example: Change the value of a specific attribute.

Translation

Translating your DITA Project *(on page 287)*

- You create your content in the primary language using a DITA authoring tool .
- Send a copy of the relevant DITA files to the localization service provider (LSP).
- Receive translated DITA content back from (LSP).

Optimizing for translation

- Use a controlled vocabulary (simplified English).
- Avoid reusing inline elements other than product names.

  https://lists.oasis-open.org/archives/dita/201301/msg00029.html

- Avoid profiling/filtering content at inline level.

Publishing

- Map-wide Validation and Consistency Checks
- Validate each topic according to DITA standard.
- Check for broken links, key references and content references, missing images or referenced resources.
- Check for broken links to remote web sites.
- Check for broken links in the context of profiling filters.

Producing the deliverables

- Checking the project before publishing.
- Sharing publishing customizations
- Automatic production of deliverables either via CMS or via an automated open source server (Jenkins).

Useful links

- DITA Style Guide (by dr. Tony Self):

  https://www.oxygenxml.com/dita/styleguide/webhelp-feedback/

- Intelligent Style Guide (by George Bina):

  https://github.com/oxygenxml/dim

- Oxygen XML Blog (Reuse, Linking, custom validation, sharing settings):

  Welcome! *(on page 8)*

Conclusions

A healthy DITA project needs to:

- Be Manageable.
- Allow for scalability.
- Allow for easy collaboration.

- Allow for detection and correction of mistakes before the deliverables are published.
- Allow for correction of mistakes after the deliverables are published.

But don't panic if you do not have all the aspects of a project covered, your project does not need to be perfect, it needs to be perfectible.

DITA Linking Strategies

This small tutorial is based on the "DITA Linking Strategies" presentations I made for the DITA Europe 2016 and DITA North America 2017 conferences. It's a general overview about DITA linking possibilities and best practices. Also, it's meant as a continuation of the DITA Reuse Strategies *(on page 150)* blog post.

According to Wikipedia:

> "A link, is a reference to data that the reader can directly follow either by clicking, tapping, or hovering."

Basically, we should regard linking as yet another form of content reuse, except that instead of presenting the content in place, it re-directs the end user to some other resource.

I'll start with describing linking at DITA Map level.

Map-Level Linking

A DITA Map uses topic references to assemble the content of a publication.

```
<topicref href="installation.dita">
 <topicref href="server_installation.dita"/>
 <topicref href="client_side_installation.dita"/>
</topicref>
```

Depending on the output format, the topic reference may be a link in the table of contents for the XHTML-based outputs or it may be interpreted as a content reference for the PDF-based output that generates a single monolith document. So the role of the `topicref` is dual, it may sometimes be regarded as a link to a topic and sometimes as a content reference.

Chunking

DITA topic modules should be kept as small as possible, but sometimes the end user may need to read more than one topic to achieve a single task. So, when publishing to HTML-based outputs, you will end up asking yourself this question:

> “Should I prefer larger HTML files or more links in the TOC?”

And you should always consider these two ideas:

- Links are disruptive. Ideally, users would not need to jump around in content to read the entire story they are searching for.
- Small topics that are usually read consecutively by the end user can probably get merged together.

For example, if the installation of your product requires installing both a server-side and a client-side component, by using DITA chunking you can choose to have separate DITA topic modules for each of the installation procedures but merge the topics together in the web-based outputs:

```
<map>
 <title>User Guide</title>
 <topicref href="installation.dita" chunk="to-content">
  <topicref href="server_installation.dita" toc="no"/>
  <topicref href="client_side_installation.dita" toc="no"/>
 </topicref>
</map>
```

You can read more about chunking in the DITA 1.3 specification. The DITA Style Guide also has a good overview about why it is preferable to write small topics and then merge them together using the chunking mechanism.

Topic-Level Linking

Links that appear inside topics can be divided into various categories and I'll discuss each of these categories separately.

In-Content Links

In-content links are links added manually in the topic content:

```
<li>See: <xref href="http://www.../" format="html" scope="external"/></li>
```

You should keep in mind that this kind of link is disruptive to the reading experience because when end users encounter them, they need to decide weather to read further on or to follow the link. On the other hand, this may sometimes be a good thing. For example, one of the installation steps may require the end user to download a certain library from an external website before continuing.

You can read more about links in general in the DITA 1.3 specification. The DITA Style Guide, written by Tony Self, also discourages the use of in-content links.

Related Links

Related links are placed at the end of the DITA topic and they allow the end user to explore additional resources after the current topic has been read.

```
<related-links>
    <link href="http://tomcat.apache.org/" format="html" scope="external"/>
</related-links>
```

To minimize disruption when reading the content in general, the preferred place where to place links is at the end of the generated HTML page.

You can read more about related links in the DITA 1.3 specification.

Defining Related Links using Relationship Tables

Related links do not need to be manually added at the end of each topic. You can define relationship tables in the DITA Map:

```
<reltable>
  <relrow>
```

```
  <relcell>
   <topicref href="client_side_installation.dita"/>
  </relcell>
  <relcell>
   <topicref href="server_installation.dita"/>
  </relcell>
 </relrow>
 ……..
</reltable>
```

These tables can define associations between two or more topics, associations that automatically contribute to the related links creation in the generated HTML output.

Here are some benefits of using relationship tables:

- A topic should have as few links as possible defined directly within. This makes it easier to reuse the topic in various contexts and keeps it as separate as possible for other parts of the DITA project, decreasing the possibility of broken links.
- By default, links defined in relationship tables are bi-directional, allowing users to land on any of the topics when searching for solutions and find their way to the related ones.
- Using a relationship table separates the task of writing topics from the task of finding relationships between topics.

You can read more about relationship tables in the DITA 1.3 specification. The DITA Style Guide also recommends using relationship tables.

Indirect Links (Key References)

All the link samples we've look at so far have been direct links, links that point to the target using the `@href` attribute. Indirect links require two steps:

1. Define a key in the DITA Map for the target.

   ```
   <keydef keys="client_installation" href="client_side_installation.dita"/>
   ```

2. Use the defined key to reference the target resources.

   ```
   <xref keyref="client_installation"/>
   ```

Here are some of the benefits of indirect linking:

- Offers the ability to reuse link target text and meta data. If you want to have custom text for a certain link, you can define it directly in the DITA Map:

```
<keydef keys="dita_ot_website" href="http://www.dita-ot.org/" format="html"
  scope="external">
  <topicmeta>
   <linktext>DITA Open Toolkit Web Site</linktext>
  </topicmeta>
 </keydef>
```

  and then add key references in all other places:

```
<xref keyref="dita_ot_website"/>
```

- Easier conditional linking (including links to topics that sometimes may be missing). If you want your topic to link either to one target or to another depending on the filtering/profiling conditions, instead of adding profiling directly on the link, you can add the profiling conditions directly in the DITA Map:

```
 <topicref keys="slicing" href="slicing_vegetables_for_experts.dita" audience="expert"/>
 <topicref keys="slicing" href="slicing_vegetables_for_novices.dita" audience="novice"/>
 <keydef keys="slicing" audience="noLink"><topicmeta><keywords>
    <keyword>Slicing</keyword></keywords></topicmeta>
 </keydef>
```

  and then link to the key from each topic:

```
<xref keyref="slicing"/>
```

- Easier link management. A good overview about all the outbound links in your project helps you maintain and control lists of allowed external web sites. With indirect references, you can define all references to external resources in a separate DITA Map. An example of a DITA project using indirect links to achieve separation of links by purpose can be found here: https://github.com/oxygenxml/dita-project-best-practices.
- Makes it easier to move/rename topics. When you move or rename a topic referenced via indirect links, only the link defined in the DITA Map will break, making it easier to fix broken links.

The DITA 1.3 specification has a chapter about indirect links.

Auto-Generated Links

Until now, I've talked about manually added links, either in the topic or in relationship tables. Using the DITA `@collection-type` attribute, you can define relationships between parent and child topic references in the DITA Map, relationships that result in automatic links added between them:

```
<topicref href="installation.dita" collection-type="sequence">
 <topicref href="server_installation.dita"/>
 <topicref href="client_side_installation.dita"/>
</topicref>
```

There are 3 useful types of `@collection-type` values:

- **Unordered** - Links are generated from parent to children, and from children to parent.
- **Family** - Links are generated from parent to children, from children to parent, and from sibling to sibling.
- **Sequence** - Links are generated from parent to children, from children to parent, and from child to previous sibling (if applicable) and next sibling (if applicable).

You can read more about auto-generated links in the DITA Style Guide.

Conditional Links in Distinct Publications

You may publish documentation for multiple products from the same **DITA** content. Also, you may want to have links point to various targets depending on the product for which you want to publish the documentation. Or, you may want to suppress links completely in certain publications.

When using direct linking, you will need to profile each link depending on the publication:

```
Find our more about slicing vegetables: <xref href="slicing_vegetables_for_experts.dita" audience="expert"/>
<xref href="slicing_vegetables_for_novices.dita" audience="novice"/>.
```

With indirect links, you can define the profiling attributes as DITA Map level:

```
<topicref keys="slicing" href="slicing_vegetables_for_experts.dita" audience="expert"/>
<topicref keys="slicing" href="slicing_vegetables_for_novices.dita" audience="novice"/>
```

and thus, simplify the reference made in the topic content:

```
Find our more about slicing vegetables: <xref keyref="slicing/>.
```

Conditional Links in the Same Publication

Using DITA 1.3 key scopes, you can reuse a topic multiple times in a DITA Map and have each referenced topic contain links to various target topics. For example, if my `preparing_vegetables.dita` topic has a link:

```
<link keyref="slicing"/>
```

you can define various key scopes in the DITA Map that bind the "slicing" key to various targets:

```
<topichead navtitle="Cooking for Experts" keyscope="expert">
 <topicref href="preparing_vegetables.dita" keys="preparing"/>
 <topicref href="slicing_vegetables_for_experts.dita" keys="slicing"/>
</topichead>
<topichead navtitle="Cooking for Novices" keyscope="novice">
 <topicref href="preparing_vegetables.dita" keys="preparing"/>
 <topicref href="slicing_vegetables_for_novices.dita" keys="slicing"/>
</topichead>
```

This previous blog post *(on page 148)* contains more details about key scopes.

Link Text

When linking to an external resource or to a DITA topic or element, the publishing engine will attempt to deduce the link text from the target context. For example, the link to a DITA topic or element that contains a `<title>` will use that title as the link text. The link to an external resource (for example to **http://www.oxygenxml.com**) will, by default, use the **HTTP** location as the link text. You can also customize each link text individually. So, ask yourself this question:

> "Should I leave the link text to be automatically computed or should I set a more friendly text?"

For internal links to elements that have a title, in general it is more flexible to not set a custom text and let the publishing engine decide one for you. For external links, you should usually specify your custom link text.

Should I Link or Should I Reuse?

Suppose you want to bring a certain paragraph, note, or section to the end user's attention. If that particular target element is not very large, you should always reuse it (using a content reference) instead of linking to it.

Conclusions

As with all large projects, managing links in a growing **DITA** project can be problematic, so you need to become organized. As an overview of what we've discussed so far, I suggest the following best practices:

- Linking is a form of reuse so:
  - Reuse small pieces of content instead of linking to them
  - Avoid too much linking (linking is disruptive)
- Use indirect links. It will allow you to reuse link text and make profiling/filtering easier while giving you a better overview of the outbound links for your project.

If you want to experiment with the various linking strategies I discussed above, you can find some samples here: https://www.oxygenxml.com/forum/files/linking-strategies-samples.zip.

DITA 1.3 Branch Filtering - Next Generation of Reuse

In this blog post, I'm going to give you a small example of how branch filtering can benefit two cases of reuse, which could not be done previously. You can read more about branch filtering in the DITA 1.3 specs.

Case 1 - Combine Two Profiles in the Same Publication

Suppose that you have a **DITA Project** about preparing and cooking vegetables and your DITA map looks like this:

```
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map>
 <title>Cooking vegetables</title>
 <topicref href="cleaningTableArea.dita" audience="novice"/>
 <topicref href="preparingVegetables.dita"/>
 <topicref href="addingExtraFlavor.dita" audience="expert"/>
 </map>
```

You have some content common both for **expert** and **novice** users but you also have content that is specific for a target audience. You do not need to teach **expert** chefs how to clean the table and you do not want to teach **novice** cooks about enhanced flavoring techniques.

All of this is fine until at some point you decide to produce a publication that contains content merged inside both the novice and the expert map content. Here's where branch filtering can help. You can create a main DITA map that reuses your current DITA map with two profiling contexts:

```
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map>
 <title>Cooking Vegetables.</title>
 <topichead navtitle="Cooking for Beginners" keyscope="novice">
  <topicref href="vegetables.ditamap" format="ditamap">
   <ditavalref href="novice.ditaval"/>
  </topicref>
 </topichead>

 <topichead navtitle="Cooking for Experts" keyscope="expert">
  <topicref href="vegetables.ditamap" format="ditamap">
   <ditavalref href="expert.ditaval"/>
  </topicref>
 </topichead>
</map>
```

Case 2 - Reusing Common Topics with Different Product Names

Suppose that you have a simple **DITA** task in which you have described how a certain task can be performed for a certain product. In our case, the task describes peeling **potatoes**:

The task works and at some point in your **Vegetables Soup** publication, you realize that you need to write a similar task about peeling **cucumbers**. The task is exactly the same, except for the product name. So naturally you want to reuse the existing written task. For this, we re-write the task so that instead of the product **potatoes**, it contains two consecutive profiled product names:

```
Peeling <ph product="potatoes">potatoes</ph><ph product="cucumbers">cucumbers</ph>
```

and include the task in the main DITA map in two places with different ditaval filters applied:

```
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map>
 <title>Peeling Vegetables.</title>
 <topichead navtitle="Potatoes">
  <topicref href="peeling.dita">
   <ditavalref href="potatoes.ditaval"/>
  </topicref>
 </topichead>

 <topichead navtitle="Cucumbers">
  <topicref href="peeling.dita">
   <ditavalref href="cucumbers.ditaval"/>
  </topicref>
 </topichead>
</map>
```

This kind of usage will produce two topic HTML files from the single `peeling.dita` (in the HTML output folder), one for each filter context.

The DITA samples for this post can be downloaded from https://www.oxygenxml.com/forum/files/branchFilteringBlogSamples.zip.

As usual, any feedback is welcomed.

DITA 1.3 Key Scopes - Next Generation of Reuse

In this blog post I'm going to give you a small example of how key scopes can benefit simple cases of reuse which could not be done previously.

Let's say you have a simple **DITA** task in which you have described how a certain task can be performed for a certain product. In our case, the task describes peeling a **potato**:

Peeling potatoes

1. Wash the potatoes in luke-warm water.
2. Use the peeler gently to remove the outer layer.

The task works and at some point in your **Vegetables Soup** publication you realise you need to write a similar task about peeling **cucumbers**. The task is exactly the same, except the product name. So naturally you want to reuse the existing written task. For this we re-write the task so that instead of the product **potatoes** it contains a key reference:

```
<ph keyref="vegetable"/>
```

Next we need to define in our DITA Map the **vegetable** key and bind it to a specific value in the potatoes context:

```
<topicref href="potatoes_overview.dita" keyscope="potatoes">
 <!-- Define the vegetable key value in this key scope -->
 <keydef keys="vegetable">
  <topicmeta>
   <keywords>
    <keyword>potatoes</keyword>
   </keywords>
  </topicmeta>
 </keydef>
 <!-- Reference to the common task -->
 <topicref href="peeling.dita"/>
</topicref>
```

and add in our DITA Map another key scope with the overview and the task which deal with **cucumbers** peeling:

```
<topicref href="cucumbers_overview.dita" keyscope="cucumbers">
 <!-- Define the vegetable key value in this key scope -->
 <keydef keys="vegetable">
  <topicmeta>
   <keywords>
```

```
    <keyword>cucumbers</keyword>
   </keywords>
  </topicmeta>
 </keydef>
 <!-- Reference to the common task -->
 <topicref href="peeling.dita"/>
</topicref>
```

As you may have noticed, we have not used the key scope names for anything. Just by defining the key scopes, we made the product name to be expanded differently in both contexts. But our **Vegetables Soup** publication may also contain a topic which lists all possible vegetables. This topic is defined in a context outside any key scope:

```
<topicref href="vegetables_over.dita"/>
```

and this overview topic can refer to each product name using the full keyscope key reference value:

```
<!DOCTYPE topic PUBLIC "-//OASIS//DTD DITA Topic//EN" "topic.dtd">
<topic id="vegetables_over">
  <title>Vegetables Overview</title>
  <body>
    <p>This is an overview of all vegetables necessary to make soup. You will learn how to use
      vegetables like <ph keyref="potatoes.vegetable"/> and <ph keyref="cucumbers.vegetable"/> to
      make a great starter soup.</p>
  </body>
</topic>
```

As stated before, this kind of reuse was not possible using the standard DITA 1.2 standard constructs. As it turns out, with DITA 1.3 we can also implement this kind of reuse using branch filtering. The DITA samples for this post can be downloaded from https://www.oxygenxml.com/forum/files/keyscopesBlogSamples.zip.

As usual any feedback is welcomed.

DITA Reuse Strategies

Introduction

This small tutorial is based on a presentation called **DITA Reuse Strategies** I made at DITA Europe 2015. It's main purpose is to explore the numerous possibilities of reusing content within the **DITA** standard.

First of all I think the main reasons we want to reuse content in technical documentation are these ones:

- Consistent explanations for the same situations.
- Less content to translate.
- Decreased time spent writing content.
- Obtain different publications from common content.

I would like to start by saying that technical documentation writers have two very important roles:

- Record knowledge about using tools and processes.
- Spread knowledge to reach large audiences.

As a software engineer, having a product user's manual which is rich in examples and answers to frequently asked questions saves me time. Instead of individually explaining to end users various application behaviors I can give links to the manual or better yet our end users find that content by themselves. Because there are just not enough human resources in any company in order to individually help each end user.

We'll start with a top down approach to reuse. Complete small examples for most of the reuse situations described below can be found here: https://www.oxygenxml.com/forum/files/dita_reuse_strategies_samples.zip.

Version Control and Reuse

Version Control allows you to reuse content tagged at a certain point in time in order to produce older versions of your publications. So no matter what open source version control system like SVN or GIT you are using or commercial CMS, you should always have the possibility to produce older bug-fix versions for your documentation. So you can think of Version Control as content reuse on the time line axis.

Converting XML content to various output formats

XML in itself is perfect for reuse because:

- **XML** is an intermediary format. We don't do XML for the pleasure of it. We do it because we want to obtain multiple outputs from it and it has enough content and structure inside to allow for it. Some call this **single source publishing** but it can be just as easily be called **content reuse**.
- **XML** contains the necessary content.
- **XML** contains the necessary structure.
- **XML** is a standard. So you have a choice between open source and commercial tools.
- **XML** is a standard for defining standards. Among which **DITA**, the most versatile standard **XML** vocabulary when it comes to reuse.

Whatever output you will obtain from the **XML**, there is a constant, this **XML** format which contains all your data will contain more semantic meaning than any of the published outputs.

You can read more about the selling points of using XML in this older blog post: A Short Story of Reuse *(on page 462)*.

Create larger publications from existing ones

You can merge multiple existing DITA Maps in various new publications.

The only existing danger for this would be if you define keys with the same name but different values in both publications. Fortunately DITA 1.3 comes to the rescue with the new keyscopes support which allows keys with the same name to be resolved to various values on each scope:

```
<!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
<map>
    <title>Vegetables Soup</title>
    <topicref href="carrots/carrots.ditamap" format="ditamap" keyscope="ks1"/>
```

```
    <topicref href="potatoes/potatoes.ditamap" format="ditamap" keyscope="ks2"/>

</map>
```

Even if you have a single root map you can keep related sections or chapters in different DITA Maps. Besides adding more logical structure to your content you never know when you'll reuse those sub-maps in different publications.

Reuse content for similar products

This is the most common case for successful reuse, you have multiple similar products which share common functionality. So similarly the technical documentation for each of those products will also share common content. This is usually done in two ways. In the following sections I will use the term **root map** for referring to the DITA Map which will actually get published.

1. Use multiple Root Maps.

Each root map is published to obtain the output for a certain product type. As major benefits you can:

- Reuse entire topics.
- Define variable product names.
- Remap links and reused content using keys.

Publication maps for phone models X1000 and X2000 using almost similar content except Blue-tooth chapter which appears in only one of them.

2. Use a single Root Map.

You have a single publication root map which gets published for various products using profiling filters applied on it. These filters can be applied either at topic or element levels. The product name is variable and depends on the applied filters.

Related information

DITA Profiling / Conditional Text

Reuse fragments of content

Until now we have regarded the topic as an indivisible unit in our project. But there are many times when it becomes useful to reuse smaller elements in various places throughout the publication.

Content References

Content references are the initial and probably the mostly used reuse mechanism in the DITA specification. They allow reusing elements from a topic in various other topics throughout the publication.

Small example of content referencing

Reusable Component from topic `reusables.dita`:

```
<dd id="CPU">
  <ul id="ul_lym_bqd_x4">
    <li>Minimum - <tm tmtype="tm">Intel Pentium III</tm>/<tm tmtype="tm">AMD Athlon</tm>
      class processor, 1 <term>GHz</term>.</li>
    <li>Recommended - Dual Core class processor.</li>
  </ul>
</dd>
```

Content reference:

```
<dd conref="path/to/reusables.dita#topicID/CPU"/>
```

You can read more about how content references can be inserted in Oxygen here: https://www.oxygenxml.com/doc/ug-editor/#topics/eppo-create-conref.html.

Content Key References

When compared to direct content references, content key references are done with indirect addressing. You first need to define a key for the topic which contains the reused content and make the content key reference using that key.

Small example of content key referencing

Reusable Component from topic `reusables.dita`:

```
<dd id="CPU">
  <ul id="ul_lym_bqd_x4">
    <li>Minimum - <tm tmtype="tm">Intel Pentium III</tm>/<tm tmtype="tm">AMD Athlon</tm>
      class processor, 1 <term>GHz</term>.</li>
    <li>Recommended - Dual Core class processor.</li>
  </ul>
</dd>
```

Key definition in DITA Map:

```
<keydef keys="reusable.install" href="reusables/reusables.dita"/>
```

Content key reference:

```
<dd conkeyref="reusable.install/CPU"/>
```

You can read more about how content key references can be inserted in Oxygen here: https://www.oxygenxml.com/doc/ug-editor/#topics/eppo-create-conkeyref.html

Content Reference Ranges

Instead of reusing a series of consecutive elements (for example steps, list items) one by one you can reuse an entire range of sibling elements. For this to work, both the initial and the final elements need to have IDs defined on them.

Small example of content key reference with ranges

Reusable steps from task `reusable_steps.dita`:

```
<steps>
    <step id="washing">
      <cmd>Wash the vegetables thoroughly.</cmd>
    </step>
    …..
    <step id="peeling">
      <cmd>Pass the peeler gently over the vegetable.</cmd>
    </step>
  </steps>
```

Key definition in DITA Map:

```
<keydef keys="reusable_steps" href="reusable_steps.dita"/>
```

Content key reference range:

```
  <steps>
    <step conkeyref="reusable_steps/washing" conrefend="default.dita#default/peeling">
      <cmd/>
    </step>
  </steps>
```

The usual dialog from Oxygen used to insert reusable content can also be used to select the range of elements to insert: https://www.oxygenxml.com/doc/ug-editor/#topics/insert-dita-content-reference.html.

Content Reuse Tips and Tricks

I tried to compile below a set of best practices to follow when reusing content:

- Keep all your reused content in special topics located in special folders. Technical writers need to know that they are editing content which potentially is used in multiple contexts.
- Keep a description for each reused element. You can have topics which act like dictionaries of reused content. A table of reused content can have two columns. On the first column each cell

contains the reused element and on the second one you can have a small description for each reused element. The description acts as metadata, it may give the technical writer more details about how that content should be reused.

- Use **conkeyrefs** instead of **conrefs**. Because relative referenced paths used in conrefs always break when you move topics around. But more about conkeyrefs in the next section.
- When using conkeyrefs you should create a special map with key definitions. This keeps the reused content and the keys for it separate from the live content.

Pushing Content

Besides the techniques we've seen so far for pulling reused content in multiple places you can also push content to a certain specified place inside an existing topic.

So why push content?

Imagine you have an existing publication "Cooking Book" containing a task with a couple of steps for peeling vegetables. At some point you create the DITA Map for a larger publication called "Cooking Book for Pros" which reuses the entire original publication by referencing to the original publication DITA Map. But you somehow need to add extra steps in the original task when the larger publication gets printed.

Pushing Content to an existing sequence of steps

Sequence of steps from the original task:

```
<steps>
 ...........
 <step id="peeler_handling">
   <cmd>Pass the peeler gently over the vegetable.</cmd>
 </step>
</steps>
```

Key definition in DITA Map for the task which will push the content:

```
<keydef href="stepsPusher.dita" keys="peeling"/>
```

Content key reference push done from the `"stepsPusher.dita"` task:

```
<steps>
    <step conaction="mark" conkeyref="peeling/peeler_handling">
        <cmd/>
    </step>
    <step conaction="pushafter">
        <cmd>Read the instructions.</cmd>
    </step>
</steps>
```

So the only purpose of the `"stepsPusher.dita"` task which is referenced with a **resource-only** processing role and thus does not appear at all in the output is to modify the content of the original task which gets published.

How do we push content in Oxygen? First you would need to define an ID on an element which will be the target for our push. The conref push mechanism allows us either to replace, insert an element before or after this target element. After this you can create the topic which pushes the content, create the step which will be pushed. You can right click inside this steps and choose **Reuse->Push Current Element...**.

Key References (Variables)

You can reuse simple variables like product name, executable, and so on by defining keywords in the **Dita Map** and then using **keyref**'s in topics to reuse those text fragments.

Reusing keywords

Defining the reused keyword in the DITA Map:

```
<!-- product name -->
  <keydef keys="product" product="editor">
    <topicmeta>
      <keywords>
        <keyword>Oxygen XML Editor</keyword>
      </keywords>
    </topicmeta>
  </keydef>
```

Reusing the keyword in a topic:

```
<title>Installation Options for <ph keyref="product"/></title>
```

In Oxygen you can create key definitions in the **DITA Map** by right clicking in the **DITA Maps Manager** and choosing **Append Child->Key definition with keyword...**. After this, in the topic you can use Oxygen's regular **Reuse Content** action to insert the **keyref**.

DITA 1.3 Contributions to Reuse

DITA 1.3 takes content reuse to an entire new level allowing you to:

- Reuse topic with variable content depending on context (keyscopes).
- Reuse the same content profiled in various ways in the same publication (branch filtering).

Reuse with Key Scopes

Using DITA 1.3 **key scopes** you can reuse a topic in multiple places in the DITA Map with slightly different content.

Reuse using key scopes

Let's say you write a topic about **Windows** installation for your software product:

```
<!DOCTYPE topic PUBLIC "-//OASIS//DTD DITA Topic//EN" "topic.dtd">
<topic id="installation">
  <title><ph keyref="osName"/> Installation</title>
  <body>
    <p>
```

```
      <ol id="ol_g5h_st4_zt">
        <li>Download the executable.</li>
        <li>Run the executable by double clicking it.</li>
        <li>Follow steps described in the installation wizard.</li>
      </ol>
    </p>
  </body>
</topic>
```

and at some point your realise that exactly the same steps need to be followed for the Linux installation. The only difference is the name of the operating system. You use a keyref to refer to the operating system name but just with DITA 1.2 support the key will resolve to a single value.

Using keyscopes in the DITA Map you can define multiple values for your key depending on the context:

```
<topicgroup keyscope="windows">
 <keydef keys="osName">
  <topicmeta>
   <keywords>
    <keyword>Windows</keyword>
   </keywords>
  </topicmeta>
 </keydef>
 <topicref href="installation.dita"/>
</topicgroup>
<topicgroup keyscope="linux">
 <keydef keys="osName">
  <topicmeta>
   <keywords>
    <keyword>Linux</keyword>
   </keywords>
  </topicmeta>
 </keydef>
 <topicref href="installation.dita"/>
</topicgroup>
```

You can find a more detailed example and download samples for reuse based on key scopes in this blog post: DITA 1.3 Key Scopes - Next Generation of Reuse *(on page 148)*.

Reuse with Branch Filtering

With branch filtering you can combine two profiles of the same DITA Map in a larger publication.

Creating a Phones Catalogues publication

If you already have a DITA Map from which you can obtain publications for various mobile phone versions based on the profiling filters applied to it, you can use branch filtering to create a larger publication which incorporates the publications for all mobile phone versions:

```
…................
  <topicref href="phoneDetails.ditamap" format="ditamap">
   <ditavalref href="ditaval/X1000Branch.ditaval">
    <ditavalmeta><dvrResourceSuffix>1</dvrResourceSuffix></ditavalmeta>
   </ditavalref>
  </topicref>
…........................
  <topicref href="phoneDetails.ditamap" format="ditamap">
   <ditavalref href="ditaval/X2000Branch.ditaval">
    <ditavalmeta><dvrResourceSuffix>2</dvrResourceSuffix></ditavalmeta>
   </ditavalref>
  </topicref>
…....................
```

You can find a more detailed example and download samples for reuse based on branch filtering in this blog post: DITA 1.3 Branch Filtering - Next Generation of Reuse *(on page 146)*

Reuse non-DITA resources

Besides DITA topics you can reuse other resources in your DITA project:

- Reuse images either referenced directly or via a key reference.
- Reuse other linked resources (like videos, PDFs and so on).

As binary resources are not embedded in the DITA topics, they are naturally reused by being kept in separate files and linked when necessary.

You can reuse images and link to other resources either via direct references or via indirect key references. What to choose may depend on how many times you refer to a certain image or binary resource. If you refer to it only once or twice you can use direct referencing.

If you have problems getting images to appear the same size when published to PDF and XHTML-based outputs you should make sure they do not have the dots-per-inch information saved inside them: https://www.oxygenxml.com/doc/ug-editor/topics/stretched-images-pdf-output.html.

Conclusions

The **DITA** standard can provide for you quite a large toolbox for reuse scenarios.

Besides the tips which are spread during this tutorial here is some additional advice for you:

- Know a little bit about all these possibilities (at least know that they exist), you never know when one of them might come in handy.
- For any given potential reuse situation you may find out that you can use multiple reuse strategies. So at a given time you could reuse a piece of simple text either via direct conrefs, indirect conkeyrefs or keyword keyrefs. Choosing one of the strategies will depend on the situation. For example if you plan in the future to also have inline elements in the reused text, you should go with either conref or conkeyref. If you reuse that content only in one or two places you can go with conref. But if you reuse it extensively you can define a key and use conkeyref.
- Try to keep the reused content separately, in special folders. Writers will know that when they are editing resources from these special folders they might modify content which is potentially used in multiple places.
- If you plan to translate your content to other languages try not to reuse inline elements (other than product name and constants which do not change when translated). Usually the translators need to translate entire block level-elements in order to have a good flow of translated content. The DITA 1.3 specs contains quite an useful recommendation for this: https://www.oxygenxml.com/dita/1.3/specs/index.html#non-normative/elementsMerged.html.

Cross-Book Links

We use cross-book links in our DITA books, and I wanted to share some of our experiences in using them.

A *cross-book link* is a link that crosses book boundaries. More specifically, it is a link from a topic in one book to another topic in another book. Cross-book links are made possible by the introduction of *keyscopes* in DITA 1.3. For more about keyscopes, see the related blog post, DITA 1.3 Key Scopes - Next Generation of Reuse *(on page 148)*.

Note:

References to "book" in this article apply equally to both `<map>` and `<bookmap>` maps.

To define a cross-book link, two components are needed:

- In the source book map, a *peer map* reference defines a keyscope for each target map that might be referenced by a cross-book link:

```
<map>
    <title>Book 1</title>
    <mapref href="book2.ditamap" keyscope="book2" processing-role="resource-only" scope="peer"/>
        ...
    </map>
```

  In DITA, a peer map is a map that exists alongside your current map. It is a declaration of another map's *existence*, but not an inclusion of that map for processing or publication.

- To create a cross-book link in a source book topic, use a `keyref` (key-based) reference to the target topic that includes the keyscope prefix for the target book:

```
<xref keyref="book2.topic2">topic 2</xref>
```

The following example shows a simple case with two books, each containing a topic that references a topic in the other book:

Note:
The figures in this article omit `processing-role="resource-only"` from `<mapref scope="peer">` elements for brevity. Although the DITA-OT implicitly treats peer map references as resource-only by default, we do explicitly define their processing role in our books.

Cross-book `<xref>` elements must explicitly contain their target text inside them. Although Oxygen automatically resolves and displays cross-book link target text during authoring, the DITA-OT does not resolve or include the target text during publishing (because peer maps are not processed). As a result, you must manually include the target text inside cross-book link elements.

Oxygen provides good support for creating cross-book links in its authoring environment. You can create cross-book links in two ways:

- Choosing a scoped key from the **Cross reference** insertion dialog box
- Doing a drag-and-drop operation from the source book in the **DITA Maps Manager** view into the topic editing window
- Doing a **Copy** of the target topic's `<topic>` element, then a **Paste special > Paste as link** where you want the link

After you create the link, check that the `@keyref` value uses the expected keyscope value. If not, check that the map context in the **DITA Maps Manager** is set to the source map so that the link is created within the correct source context. This can be done either by setting the context drop-down to **<Current map>** and ensuring that the source map tab is selected, or by explicitly setting the context drop-down to the source map (but remember to change it back if you do this).

An interesting cross-book link problem occurred when we started reusing topics across books. We encountered a case where (1) multiple books reused the same topic, and (2) that topic had cross-book links to topics in those books:

In this example, the `shared.dita` topic is included in both book1 and book2, and it contains references to other topics in those books. The link to `book1.topic1` was broken for the topic instance in book1 (because book1 had no knowledge of a "`book1`" keyscope), but the same link worked for the topic instance in book 2 (because book2 had a "`book1`" peer map keyscope defined). A similar issue happened for the link to `book2.topic2` in book2.

The solution was to define local map-level keyscopes for book1 and book2:

This allowed keyscoped links in reused topics to always resolve properly, whether they pointed to a topic in the same book or a peer map book.

Note:

If you use local map-level keyscopes, be sure to use Oxygen 24.0 or later to author your content. It contains some fixes to create cross-book links properly in locally keyscoped map contexts.

If you use cross-book links, it is important to understand how the DITA-OT handles them. When a book containing cross-book links is published by itself, the cross-book link elements are accepted syntactically, but during processing they become unresolved links because the target peer maps are not processed. To fix the links, you can run some kind of post-processing utility on the final set of published content. For HTML-based output (such as WebHelp), we use the following utility:

https://github.com/chrispy-snps/DITA-fix-xbook-html-links

(We do not yet have a solution for cross-book links in our PDF output, but we plan to look into it in the future.)

But, something very interesting happens when these books are published together as part of a higher-level "collection" map:

olh.ditamap

```
<map>
  <title>Online Help</title>
  <mapref href="book1.ditamap" keyscope="book1"/>
  <mapref href="book2.ditamap" keyscope="book2"/>
  <mapref href="book3.ditamap" keyscope="book3"/>
</map>
```

book1.ditamap

```
<map keyscope="book1">
  <title>Book 1</title>
  <mapref href="book2.ditamap" keyscope="book2" scope="peer"/>
  <mapref href="book3.ditamap" keyscope="book3" scope="peer"/>
  <!-- book 1 topics -->
</map>
```

book2.ditamap

```
<map keyscope="book2">
  <title>Book 2</title>
  <mapref href="book1.ditamap" keyscope="book1" scope="peer"/>
  <mapref href="book3.ditamap" keyscope="book3" scope="peer"/>
  <!-- book 2 topics -->
</map>
```

book3.ditamap

```
<map keyscope="book3">
  <title>Book 3</title>
  <mapref href="book1.ditamap" keyscope="book1" scope="peer"/>
  <mapref href="book2.ditamap" keyscope="book2" scope="peer"/>
  <!-- book 3 topics -->
</map>
```

Note that the top-level "collection" map includes each book map within its relevant keyscope. As a result, links that were "cross-book" in the individual maps automatically resolve and become local links in the higher-level collection map! And because these links resolve during publishing, no post-processing is needed.

Note:

When a keyscoped `<mapref>` points to a map that has its own local map-level keyscope, the keyscopes are not cascaded. Instead, a single keyscope with the union of keyscope values is used, so that any keyscope in the union can be used to reference into the map. For details, see 2.3.4.2 Key scopes.

This demonstrates a very useful aspect of keyscoped links between books—links automatically become cross-book or local as needed, depending on what is included in publishing. This gives you the freedom to publish books individually, in a single collection, or in multiple collections. And you can even rearrange books across collections, all without having to modify any links in the DITA source.

DITA XML vs Markdown Syntax and Capabilities Comparison

The following article is a comparison between the DITA XML standard and Markdown. The comparison attempts to cover syntax specification and features. I attempted to write this comparison without any implicit bias towards one or the other. If there are DITA XML or Markdown features that I missed, that was done out of ignorance and not out of malice. Feedback is always welcomed, as usual.

Table 1. DITA XML vs Markdown

	DITA XML	Markdown
Short description	DITA XML is a standard for designing, writing, managing, and publishing information. There are multiple versions of the DITA standard, the most popular one being version 1.3.	Markdown is a lightweight markup language that you can use to add formatting elements to plain text documents. There was an effort to standardize Markdown to a specification named CommonMark. There are lots of Markdown flavors and extensions, most of them sharing a common set of features. The most popular are probably CommonMark and Github-flavored Markdown.

Table 1. DITA XML vs Markdown (continued)

	DITA XML	Markdown
Why should I use this format?	• You work (or want to work) for a company which manufactures similar products (large possibility of content reuse). • You want to obtain multiple output formats (HTML based, quality PDF). • You want to work with a standard and be able to change tools if necessary. • You want to impose structural validation constraints on the edited content.	• You work for a company which has mostly one product and publishes the documentation mostly on the web. • You work with ticketing systems which use Markdown for styling content. • You produce internal documentation or small articles.
Useful resources for learning	Resources for learning DITA with Oxygen *(on page 66)*	• Basic Markdown Syntax • Markdown: Syntax
Pros	DITA XML pros and cons • OASIS Open standard. • Advanced support for content reuse *(on page 150)* either at topic, block, or inline level. • Advanced support for filtering *(on page 153)* (generating multiple similar user guides from the same content). • Open-source publishing engine with lots of supported output formats (some free, some commercial) like HTML5, Windows Help, PDF, Word, EPUB, and so on. Why use DITA	Markdown pros and cons. • Large user base. Familiar to software engineers who use it to write issues. • Basic syntax, easy to learn. • Easier to read without specialized tools. • Offline and online free editing tools. • For the base syntax, quite easy to edit the content in a plain text editor tool. • Lots of static web site generator open-source tools like MKDocs or Jekyll.

Table 1. DITA XML vs Markdown (continued)

	DITA XML	Markdown
	Pros and Cons	
Cons	• Smaller user base. • Harder to learn. • XML is more verbose than plain text. • Visual editing requires the use of a commercial tool like Oxygen. • Smaller number of open source tools to generate professional looking outputs. Pros and Cons	• Not all language features are available in the base Markdown "specification". There are various flavors with various syntax differences between them and you probably need to pick a flavor to use and stick to it. • Advanced features like content reuse, for example, are not in the base standard but may be implemented with different syntaxes for various flavors. • Static web site generators are not compatible with each other (they have various specific configuration files) or to link between files. • Not many possibilities to assemble multiple Markdown files and publish outputs like PDF or Word, for example. • Cannot render complex cell content (multiple paragraphs, for example) in table cells or in list items.
Cross-Compatibility	A DITA Map can refer to a Github-flavored Markdown file and the publishing engine can perform a dynamic conversion from Markdown to DITA while editing.	-
Table of contents	Gathering multiple DITA topics in a larger publication and defining the table of contents is done by using DITA Maps. Working with DITA Maps	**CommonMark** does not define the possibility to create a table of contents or to aggregate multiple Markdown files in larger publications.

Table 1. DITA XML vs Markdown (continued)

	DITA XML	Markdown
		Various static web site generators have various ways to define table of contents, usually based on **Yaml**, like MKDocs.
Validation	• Validation according to the DITA specification DTDs/schemas done when publishing or when editing. • Additional validation can be done with Schematron rules *(on page 409)*.	• Usually with Markdown, you can look at a live preview while typing to see that everything looks OK. • There are various processors that may be used to validate Markdown, for example using a set of JSON rules.
Publishing	• The DITA Open toolkit publishing engine comes with default support to publish DITA Maps and customize to plain HTML5, PDF. • There are additional open-source plugins to publish to MS Word or EPUB. • Other curated open-source plugins are available in the DITA OT plugins registry. • Commercial plugins are available to publish to WebHelp output like Oxygen WebHelp or Fluid Topics.	Most publishing libraries rely on the conversion from Markdown to HTML. • Lots of open-source static web site generators. • Lots of libraries (Javascript, Java, Python, etc) to convert Markdown to HTML. • Other conversion types available using Pandoc.
Translation	There are translation agencies directly accepting DITA XML content or you can convert DITA XML to XLiff and use a translation system. Each DITA XML topic or map can have an `@xml:lang` attribute to specify the current language in which it is written. Translating your DITA Project *(on page 287)*	There are various tools like Simpleen that seem to specifically handle Markdown translation.

Table 1. DITA XML vs Markdown (continued)

	DITA XML	Markdown
Extensibility	• Possibility to define a new specialization of the DITA vocabulary with new element names. • Use the `@outputclass` attribute value on elements to set custom values used when styling the output. • Use the DITA `<data>` element with custom names and values and take them into account with publishing time customizations. • Use the DITA `<foreign>` element (for example, embed HTML inside it using a custom publishing plugin).	• Use HTML elements inside Markdown, for example, when defining complex tables or you do not have a Markdown equivalent. • Yaml headers. • Ability on certain Markdown flavors/extensions to define attributes for each element.

Metadata	• The DITA `<prolog>` element can contain lots of metadata information, but not visible in the published output. Example:
	`<topic id="topic_wcj_tgy_5wb">`
	`<title>The Title</title>`
	`<prolog>`
	`<author>The Author</author>`
	`<metadata>`
	`<keywords>`
	`<keyword>one</keyword>`
	`<keyword>two</keyword>`
	`</keywords>`
		• Sometimes, Markdown files may contain Yaml headers before the actual content that define simple keys and values. Example:
		`---`
		`title: The Title`
		`author: The Author`
		`keywords: [one, two, three, four]`
		`---`
		`# A Heading`
		`Text body.`

Table 1. DITA XML vs Markdown (continued)

	DITA XML	Markdown

	`</metadata>`
	`</prolog>`
	• The `<indexterm>` elements are also considered metadata, as they are used to generate an index table.	

Content reuse:	• Key references to re-used product names. • Content references to re-used elements in multiple places. • Content reference push to push content in multiple places. • Code references to re-use pieces of code in multiple places. • Key scopes and branch filtering to re-use the same topics in multiple contexts with different content in each context. • Re-use a topic in multiple places in the publication.	No content re-use support is in the standard base. Various extensions do exist, for example: • Redocly uses HTML `<embed>` tags with references to Markdown files to re-use entire chunks of Markdown content placed inside a file. • Hugo uses special notations named shortcuts.
Filters	You can use profiling attributes in DITA XML topics or on topic references in a DITA Map map. By using a single DITA Map and filtering *(on page 153)* it differently, you can obtain multiple publications from it. For example, for the Oxygen user's manual, we obtain lots of distinct publications for "Oxygen XML Editor", "Oxygen XML Author", "Oxygen XML Web Author" from the same DITA Map.	There may be, but I am not aware of such a feature in Markdown.

Table 1. DITA XML vs Markdown (continued)

	DITA XML	Markdown

Headings	• DITA topics have a `<title>` element that appears as a heading 1 when published and is also used for the `<title >` element in the published HTML document.
	• You can nest topics one inside the other and the generated HTML output will have `<h2>`, `<h3>`, etc for each nested topic, depending on the nested depth.
	• You can have `<section>` elements with `<title>`elements inside a topic (they cannot be nested one inside the other).
	
	`<topic id="topic_wcj_tgy_5wb">`
	`<title>Title1</title>`
	`<body>`
	`<section>`
	`<title>Section 1</title>`
	`<p>paragraph</p>`
	`</section>`
	`</body>`
	`<topic id="inner">`
	`<title>Inner topic title</title>`
	`</topic>`
	`</topic>`
		You can use a number of `#` characters followed by space and text to define a new heading. Headings do not necessarily need to be incremental, you can start with heading level 2 and then have a heading level 1.
		
		`# Heading level 1`
		`### Heading level 3`
		`## Heading level 2`
		`....`

Block elements	There are multiple topic types like `<concept>`, `<task>`, `<reference>`, and extra topic types can be added using a specialization. The basic block elements are `<topic>`, `<title>`, paragraph `<p>` elements, `<codeblock>`, lists `<ul>` `<ol>`, `<table>`, `<section>`, `<fig>`, `<image>`, `<note>`.	Block elements: Paragraphs, tables, lists, images, block quotes, etc.

Table 1. DITA XML vs Markdown (continued)

	DITA XML	Markdown
	There are also other block-level elements, depending on the topic type.	
Inline elements	`<b>`, `<i>`, `<u>`,`<sup>`,`<sub>` and other inline elements with more semantic meaning (like `<codeph>`, `<uicontrol>`, `<filepath>`).	Bold, italic, underline. Depending on the Markdown flavor, other inlines like subscript, superscript, strike-through.
Equations / Formulas	• MathML equations added directly inside the DITA topics or referenced as separate files. • Latex equations *(on page 213)* embedded using third party plugins in DITA `<foreign>` elements.	GitHub flavored Markdown and other flavors as well support embedding Latex equations in Markdown content using `$`.
Audio/Video	The DITA `<object>` element can be used to reference audio, video or iframe content.	No official support, maybe use embedded HTML content or add a link to the audio/video instead.
Tables	The DITA `<table>` element is based on the CALS table specification. Cells can span multiple rows or columns and contain inside block elements content like lists, paragraphs. The table can have header and body rows.	Markdown tables are usually written in an ASCII graphic, like representation allowing for cells content to be aligned left or right. By default, cells can contain only plain text inside. If more complex table structures are needed, HTML tables can be inserted directly in Markdown if the used Markdown flavor supports HTML elements inside it.
Lists	Ordered `<ol>`, unordered `<ul>`, or definition lists `<dl>`. Other topic types like `<task>` contain, for example, the `<steps>` element that is an ordered list of steps. Each list item can contain block elements like paragraphs, other lists, tables, etc.	Ordered and unordered lists. Each list element contains simple content. It cannot contain block- level elements like additional lists or multiple paragraphs. The task list is an interesting extension to show checkboxes next to each list item. Other types of lists (definition list for example) or list items that contain multiple block-lev-

Table 1. DITA XML vs Markdown (continued)

	DITA XML	Markdown
		el elements can be inserted directly in Markdown if the used Markdown flavor supports HTML elements inside it.
Links	• Internal links (cross references): ◦ Link to another topic. ◦ Link to a particular element in another topic. ◦ Links to web resources. • Related links (at the end of each topic) ◦ Link to another topic. ◦ Link to a particular element in another topic. ◦ Links to web resources.	Links to other web resources.
Conclusion	• Harder to type in a plain text area, requires DITA editing tools (most of which, are not free). • Advanced support for structured validation. • Advanced support for content re-use and profiling conditional text. • Publishing engine allows publishing to multiple output formats like HTML, PDF, and others based on plugins that can be installed.	• Easy to manually type in a plain text area but a preview definitely helps. • More complex elements need to be inserted as HTML elements. • Various Markdown extensions have extra support for example for content reuse. • Mostly targeted towards obtaining web-based HTML content. • Looks like a language that is not intended to do the heavy lifting of producing multiple deliverable formats and deliverables from the same content.

Publishing Customizations

An overview of various useful publishing customizations that appear as individual articles on the Oxygen XML Blog.

DITA Project Overview

- DITA Map Metrics Evolution *(on page 277)*
- Creating Link Graphs From DITA Maps *(on page 222)*

Embed Latex Equations in DITA Topics

Adding support for embedding LaTeX equations in DITA content *(on page 213)*

Embed PlantUML/Mermaid Diagrams in DITA Topics

- Embedding Diagrams in DITA topics using PlantUML *(on page 193)*
- Embedding Diagrams in DITA Topics Using Mermaid *(on page 194)*

DITA Task Steps Presentation Alternatives

- Presenting DITA Task Steps as Diagrams *(on page 195)*
- Show DITA Task Steps in Separate Tabs *(on page 199)*

Generate Google Structured Data Content from DITA Content

- Generating Google Structured Data from your DITA tasks *(on page 236)*
- Generating Google Structured Data from your DITA frequently asked questions *(on page 239)*

Present Consecutive Elements as Tabs in HTML Output

- Show Consecutive DITA Code Blocks in Separate Tabs *(on page 198)*

Convert OpenAPI Content to DITA

Converting OpenAPI to DITA XML and Publishing to WebHelp (Experimental) *(on page 200)*

Various Applications for DITA to Markdown Publishing

- Creating PowerPoint Slides from DITA Content *(on page 226)*
- Publishing DITA Content Using the MKDocs Web Site Generator *(on page 222)*

DITA Prolog Customizations

- Displaying prolog details (author, creation date, read time) *(on page 229)*
- Dynamically publishing a list with the most recent topics *(on page 231)*
- Dynamically Generating an RSS Feed *(on page 233)*

Marking DITA Topics with Labels for the HTML-based Output

Adding Labels to Topics in the WebHelp Responsive Output *(on page 227)*

Generating Content for Google Chatbot

Creating a Knowledge Base for the Google Dialogflow Chatbot using DITA XML content *(on page 331)*

Externally Contributed Articles

- Cross-Book Links *(on page 163)*
- Preprocessing DITA-OT Project Files *(on page 296)*

Server Side Building and Publishing

The Oxygen Publishing Engine is a commercial licensed command-line tool based on the **DITA Open Toolkit**. This means that instead of publishing directly from Oxygen XML Editor, you can use an integration server to run the transformations and publish the output or simply use it from a command-line interface.

Possible ways to run the Oxygen Publishing Engine on the server side:

- Building and Publishing Using a Script *(on page 180)*.
- Building, Validating, and Publishing Using GitHub Actions *(on page 181)*.
- Creating a Docker Image for the Oxygen Publishing Engine *(on page 184)* and then Running the Publishing Engine Docker Image on AWS *(on page 185)*.
- Creating a Maven project and running it from a Jenkins Job.

Features of the Oxygen Publishing Engine:

Publishing WebHelp Responsive Output

The WebHelp Responsive *(on page 187)* system features a very flexible layout and is designed to adapt to any device and screen size to provide an optimal viewing and interaction experience. It is based upon the *Bootstrap* responsive front-end framework and is available for DITA document types.

Publishing PDF based on CSS

You can use the same Publishing Templates to build both the WebHelp and PDF outputs. The Oxygen Styles Basket allows you to build a base publishing template visual customization based on CSS, and this customization can later be used to produce the outputs.

Publishing to Zendesk

The Enterprise edition of the license allows you to publish DITA XML content directly as **Zendesk** articles.

Change Tracking and Comments in the Published Output

You can enable a parameter to show Oxygen tracked changes and comments in the generated WebHelp and PDF outputs.

Syntax Highlight inside Code Blocks

Syntax highlights of code blocks in the WebHelp and PDF outputs.

Embedding Media Video/Audio Resources

Embed video and audio resources in the published WebHelp and PDF outputs.

DITA Metrics Report

Build statistics and metrics for a DITA XML project.

Dynamic Word/Excel/OpenAPI/Makdown to DITA Conversions

Dynamically convert Word/Excel/OpenAPI and Markdown files referenced in the DITA Map to DITA XML content while publishing.

Add Edit Links to Published Output

Add edit links for each published topic, connecting the published output to an online editor for providing feedback.

Related information

Webinar: Introducing the Oxygen Publishing Engine for DITA

Building and Publishing Using a Script

As an example of scripting the publishing on the server side, the **Oxygen XML Blog** has a Gradle build file that can be invoked to automatically build and publish WebHelp and PDF outputs of the blog content. Features of the sample script:

- Downloads and unzips the Oxygen Publishing Engine.
- Creates a license key in the publishing engine's folder based on an environmental variable named `LICENSE_KEY`.
- Runs the publishing on the DITA OT Project file bundled with the Blog project that defines all deliverables (WebHelp, EPUB).
- Produces all outputs in a `site` folder in the project's base folder.

You can start from such an example to build your own script that downloads the Oxygen Publishing Engine, sets up the license details, and then publishes:

```
cd ./build; sh ./gradlewCustom --info --stacktrace
```

Building, Validating, and Publishing Using GitHub Actions

If you have a GitHub-hosted DITA XML project (either on a private or public repository), you can define GitHub actions to validate and publish your DITA XML project.

As an example, the Oxygen XML Blog contains GitHub actions to validate and publish a DITA XML project:

- Publish the Oxygen XML Blog DITA XML project to **Netlify** using a **GitHub** action:
    1. Create an account on Netlify and create a site for the account. Create an authentication token to be able to access Netlify from a command line and deploy to the site from a remote location. In the Netlify UI, obtain the ID of your site.
    2. In the GitHub project's **Settings->Secrets and variables->Actions** category, define two variables to keep the Netlify authentication token and the side ID values. Also, create a secret for the **Oxygen Publishing Engine** license key.
    3. Create a custom Gradle build file that downloads the **Oxygen Publishing Engine**, licenses it, produces the output, and stores it in a specific folder.
    4. Create a custom GitHub action that runs the build file and then uses a plugin to upload the entire folder to the Netlify web site. The end result looks like this: https://oxygenxmlblog.netlify.app/.
- Publish the DITA XML project to an **Amazon S3** server using a **GitHub** action:
    1. Create an **Amazon S3** bucket and configure it for static website hosting. Make note of the bucket name and region.
    2. In the **IAM** console, create a new user with programmatic access and attach a policy that grants write access to the S3 bucket. Obtain the access key ID and secret access key for this user.

3. In the GitHub project's **Settings->Secrets and variables->Actions** category, define variables to store the S3 bucket name, region, access key ID, and secret access key. Also, create a secret for the **Oxygen Publishing Engine** license key.
4. Create a custom Gradle build file that downloads the **Oxygen Publishing Engine**, licenses it, produces the output, and stores it in a specific folder.
5. Create a **GitHub** action that runs the build file and then uses the aws-actions/configure-aws-credentials action to configure the AWS credentials, followed by the aws-actions/s3-sync action to sync the output folder with the S3 bucket.

- Validate the entire DITA project using the Oxygen **Validate and Check for Completeness** command-line script:
  1. In the GitHub project's **Settings->Secrets and variables->Actions** category, define a variable named `SCRIPTING_LICENSE_KEY` that has the Oxygen scripting license key as its value.
  2. Set up a Gradle build file that downloads an Oxygen all platforms kit, licenses it, and runs it over the DITA map. The build file also applies an XSLT stylesheet over the XML report to produce a Markdown report.
  3. Create a GitHub action that is triggered by pull requests or modifications done to pull requests, sets up Java, calls the Gradle build file, and then if there are errors, outputs the report in Markdown format using the `$GITHUB_STEP_SUMMARY` variable. The report of running

such a GitHub action contains the severity, description, and a link to the topic where the error was found. The link can be followed to open the topic in Oxygen XML **Web Author**:

validate-check-completeness summary

Validate and check for completenss Results

Incident

Severity: error

Description: Attribute value "b_wqb_ fbc_d1c" of type NMTOKEN must be a name token.

Location: topics/10_things_i_enjoy_about_the_new_oxygen_json_editor.dita

Incident

Severity: error

Description: File not found: "topics/notexisting.dita".

Location: topics/10_things_i_enjoy_about_the_new_oxygen_json_editor.dita

Job summary generated at run-time

- Validate only the changed topics in a pull request using **OpenAI** and a custom prompt to check for grammar problems:
    1. In the GitHub project's **Settings->Secrets and variables->Actions** category, define a variable named `AI_LICENSE_KEY` that has the **OpenAI** key as its value.
    2. Create a custom Gradle build file that finds all changed topics in the pull request and then uses the API key to send the content to **OpenAI** asking for a report of changes in **Markdown** format.
    3. Create a custom GitHub action that saves the list of changed files as an environmental variable and passes this information to the build file, afterwards displaying the report using

the `$GITHUB_STEP_SUMMARY` variable. The report of running such a GitHub action looks like this:

```
@@ -21,7 +21,7 @@
                    <li>Value for money. The 12 months subscrption license is <xref
                            href="https://www.oxygenxml.com/json_editor/buy.html" format="html"
                            scope="external">less than 100 USD</xref> per year and gives you always
-                       access to the latest JSON Editor released version.</li>
+                       access to the latest JSON Editor release version.</li>
                    <li>Support to create JSON Schemas and to use JSON Schemas to validate JSON
                        instances. There is a great <b id="b_zgq_pbc_d1c">JSON Schema</b> diagram whi
                        can be used to visually compose JSON schema files. I used this support quite
@@ -33,7 +33,7 @@
                        <li>Find Replace capabilities.</li>
                        <li>and much more....</li>
                    </ul></li>
-               <li>As another neat feature of being an Oxygen application, you can install in the
+               <li>As another neat feature of being an Oxygen application, you can install into the
                    Oxygen JSON Editor free useful add-ons like the <xref
                        href="https://www.oxygenxml.com/doc/ug-editor/topics/git-addon.html"
```

- A sample project which uses OpenAI to check topics for grammar problems and to create automatic pull requests for this.

Creating a Docker Image for the Oxygen Publishing Engine

Docker is a platform used to build, share, and run portable application containers.

While the Oxygen Publishing Engine does not provide a docker image by default, such an image can be constructed quite easily based on the instructions in the DITA Open Toolkit documentation.

A sample Docker configuration file for the Oxygen Publishing Engine is included in the Oxygen XML Blog project. Characteristics of the Docker configuration file:

- Installs Java 17.
- Downloads the Oxygen Publishing Engine.
- Sets up a script that will provide the license key to the publishing engine using the `LICENSE_KEY` environmental variable.

Steps to use the sample Docker configuration file:

Build a Docker Image

Assuming you have installed Docker, open a console or terminal on a Mac or Linux system, "cd" to the folder where the `Dockerfile` configuration file is located, and run the command:

```
docker image build -t oxygen-publishing-engine:1.0 .
```

Running a Container Using the Docker Image

Assuming you have obtained a license key to use the Oxygen Publishing Engine commercially, you can paste the license key in a `licensekey.txt` file, then provide the license key content as an environmental variable to the docker image. Other provided parameters are the mapped folder from where the Docker container should obtain the DITA XML content, the image name and version, and parameters that get sent to the `dita` command for publishing. The container is automatically removed from Docker once finished as the produced output is already saved on the shared volume.

```
$: docker run -e LICENSE_KEY="$(<licensekey.txt)" \

--rm --name dita-ot-publish\

-v /Users/.../samples/dita/flowers:/src  \

oxygen-publishing-engine:1.0 \

-i /src/flowers.ditamap \

-o /src/out \

-f webhelp-responsive -v
```

Running the Publishing Engine Docker Image on AWS

You can create an EC2 instance on the Amazon Cloud Computing AWS platform, run the publishing there if changes are detected in a Git project, and publish the content to the same platform.

Create an Account on AWS

Sign in to create an account on AWS.

Create an EC2 Instance

Create an EC2 Instance on AWS.

Configure the Security Group and Install a Web Server

Configure the Security Group associated to the EC2 instance to allow for HTTP or HTTPS connections and install a web server.

Connect to the EC2 Instance

Use the AWS Instance Connect to open a console in the web browser to the instance.

Install Git in the EC2 instance

Run this command in the console to install git:

```
sudo yum install git
```

Install Docker in the EC2 Instance

Run this command in the console to install Docker and start the Docker daemon:

```
sudo yum install docker

sudo systemctl start docker
```

Clone the Git Project

Create a local clone of the Git project by running this command in the console:

```
git clone https://github.com/oxygenxml/blog.git
```

Create the Docker Image

Change directory to the folder containing the `Dockerfile` and run Docker to create an image.

```
cd blog/build

sudo docker image build -t oxygen-publishing-engine:1.0 .
```

Save the Oxygen Publishing Engine License to a file

Save the Oxygen Publishing Engine license key to a file (for example, in `/home/ec2-user/licensekey.txt`).

Run a Container Based on the Docker Image to Publish

```
sudo docker run -e LICENSE_KEY="$(</home/ec2-user/licensekey.txt)" \

--rm --name dita-ot-publish\

-v /home/ec2-user/blog/:/src  \

oxygen-publishing-engine:1.0 \

-p /src/blog-project.xml \

-o /src/out \

-f webhelp-responsive -v
```

Copy the Published Content to the HTTP Web Server's Folder

```
cp -rf /home/ec2-user/blog/out/site/* /var/www/html/
```

Script to Update the Repository And Publish If Necessary

Create a script (for example, named `pullPublish.sh`) with the contents:

```
#! /bin/sh

cd /home/ec2-user/blog

git -C . remote update &> /dev/null

checkgit=`git -C . status`
```

```
if [[ ! "$checkgit" =~ "up to date" ]]; then

    # Pull

    git -C . pull

    # And publish

    sudo docker run -it -e LICENSE_KEY="$(</home/ec2-user/licensekey.txt)" -v /home/ec2-user/blog/:/src

  oxygen-publishing-engine:1.0 -p /src/blog-project.xml -o /src/out -f webhelp-responsive -v

    # Move web site to web server's folder.

    rm -rf /var/www/html/*

    mv -f /home/ec2-user/blog/out/site/* /var/www/html/

fi
```

Schedule Script to Run Every 10 Minutes

Edit the `crontab` file:

```
cd /etc

sudo vi crontab
```

and add to the file:

```
*/10 * * * * /home/ec2-user/pullPublish.sh
```

DITA WebHelp Publishing Solution Advantages

Oxygen XML Editor's own user guide is generated from DITA XML content and published as WebHelp Responsive output: https://www.oxygenxml.com/doc/ug-editor/.

Below, I tried to compile a list of advantages that the published WebHelp output obtained from DITA XML content has:

- It's a pure HTML5+Javascript publishing solution without the need of any server-side component.
- The WebHelp output can be used both offline and online.
- It's compatible with all browser vendors: https://www.oxygenxml.com/doc/ug-webhelp-responsive/topics/wh_getting_started.html.
- It has a responsive layout, adjusts to the screen size, and can be used on a wide variety of devices (mobile devices, tablets, or desktops).
- The WebHelp layout can be fully customized: https://www.oxygenxml.com/doc/ug-webhelp-responsive/topics/webhelp-responsive-description.html.
- The WebHelp publishing comes with 7-8 pre-configured color themes, but you can also create your own publishing templates: https://www.oxygenxml.com/doc/ug-webhelp-responsive/topics/whr_publishing_templates.html.

- You can also use the free Oxygen Styles Basket website to customize the WebHelp output: https://styles.oxygenxml.com/.
- It contains its own search engine implemented in Javascript (with support for AND, OR, exact match, auto complete, and highlights). The search engine can assign various weights and priorities to various elements.https://www.oxygenxml.com/doc/ug-webhelp-responsive/topics/whr_searching.html.
- It automatically generates an Index page if the DITA content contains `<index>` elements.
- For each section contained inside a topic, a mini table of contents is generated: https://www.oxygenxml.com/doc/ug-webhelp-responsive/topics/whr-html5-pages.html#whr-topic-page-layout__topic-topic-toc.
- Inner topics, sections, and topics can be folded.
- Code block content can be automatically syntax highlighted when publishing: https://www.oxygenxml.com/doc/ug-webhelp-responsive/topics/add-syntax-highlights-codeblocks.html.
- Tracked changes and comments made in the DITA XML content can be published as side notes in the WebHelp output: https://www.oxygenxml.com/doc/ug-webhelp-responsive/topics/webhelp-responsive-plugin-additional-parameters.html#webhelp-responsive-plugin-additional-parameters__webhelp_show_changes_and_comments.
- The WebHelp output comes with support for the following built-in languages: English, French, German, Japanese, and Chinese, while support for additional languages can be added. It can also be used with right-to-left text orientation: https://www.oxygenxml.com/doc/ug-webhelp-responsive/topics/whr-localization.html.
- It can be used for context-sensitive help, called with various help IDs from an application:https://www.oxygenxml.com/doc/ug-webhelp-responsive/topics/whr-context-sensitive.html.
- The WebHelp Feedback system can be integrated in every WebHelp page and provides support for users to add comments and offer feedback (create account, post message, moderate, manage multiple versions of documentation): https://www.oxygenxml.com/doc/ug-webhelp-responsive/topics/whr-feedback-system.html.
- You can embed Facebook, Twitter, or Google Analytics: https://www.oxygenxml.com/doc/ug-webhelp-responsive/topics/webhelp-social-media-integration.html.
- WebHelp documentation can also be produced on the server side: https://www.oxygenxml.com/xml_webhelp/download_oxygenxml_webhelp.html.
- The WebHelp output was tested for Accessibility problems, it works well with a screen narrator, and one of the WebHelp color themes is fully 508 compliant for people with reduced vision: https://www.oxygenxml.com/doc/ug-webhelp-responsive/topics/whr-accessibility.html.

The following past webinars may further help people who want to customize the WebHelp output:

- Techniques for Customizing the WebHelp Responsive Output
- Documentation Review Workflow Using Oxygen Feedback
- DITA Publishing and Feedback with Oxygen Tools

PDF

DITA OT PDF Customization - Table with Alternate Row Background Colors

Based on this forum thread I will try to give some steps in which you can create your own DITA Open Toolkit PDF customization folder for customizing a table to have alternate row background colors.

1. First of all you need to know the XSLT template that you need to override.
   - You can open a DITA topic which has a table inside it and select in the **Outline** view the table row. The **Attributes** view will show you its **@class** attribute value which is **- topic/row** .
   - Use the **Find/Replace in Files** tool to search in the PDF plugin folder (for example `DITA-OT-DIR/plugins/org.dita.pdf2/`) for the string **topic/row**.
   - In the XSLT stylesheet `DITA-OT-DIR/plugins/org.dita.pdf2/xsl/fo/tables.xsl` you will find a template which matches all rows from a table body:

     ```
     <xsl:template match="*[contains(@class, ' topic/tbody ')]/*[contains(@class, ' topic/row ')]">
         <fo:table-row xsl:use-attribute-sets="tbody.row">
             <xsl:call-template name="commonattributes"/>
             <xsl:apply-templates/>
         </fo:table-row>
     </xsl:template>
     ```

     That is the template which you will need to overwrite in your customization.
2. Copy the entire folder `DITA-OT-DIR/plugins/org.dita.pdf2/Customization` to an external location. For example in my case I copied it to my Desktop.
3. Renamed in that copied folder the `catalog.xml.orig` file to `catalog.xml`, edit it and uncomment the line:

   ```
   <uri name="cfg:fo/xsl/custom.xsl" uri="fo/xsl/custom.xsl"/>
   ```

   This custom catalog file will be automatically used to contribute in the PDF publishing process with high priority the XSLT stylesheet located in `Customization/fo/xsl/custom.xsl`.
4. Rename in the `Customization/fo/xsl` folder the `custom.xsl.orig` file to `custom.xsl`. This stylesheet will contain all your template overrides.
5. Overwrite in the `custom.xsl` the original template like:

   ```
   <xsl:template match="*[contains(@class, ' topic/tbody ')]/*[contains(@class, ' topic/row ')]">
       <fo:table-row xsl:use-attribute-sets="tbody.row">
   ```

```
        <xsl:choose>
            <xsl:when test="(count(preceding-sibling::*[contains(@class, ' topic/row ')]) mod 2) = 0">
                <!-- Even row, light blue -->
                <xsl:attribute name="background-color">rgb(210, 222, 253)</xsl:attribute>
            </xsl:when>
            <xsl:otherwise>
                <!-- Odd row, white -->
                <xsl:attribute name="background-color">white</xsl:attribute>
            </xsl:otherwise>
        </xsl:choose>
        <xsl:call-template name="commonattributes"/>
        <xsl:apply-templates/>
    </fo:table-row>
</xsl:template>
```

6. If you want the table frame border colors to have a custom color you can override some attribute sets defined in the `DITA-OT-DIR/plugins/org.dita.pdf2/cfg/fo/attrs/tables-attr.xsl`:

```
<xsl:attribute-set name="table__tableframe__top" use-attribute-sets="common.border__top">
    <xsl:attribute name="border-top-color">blue</xsl:attribute>
</xsl:attribute-set>
<xsl:attribute-set name="table__tableframe__bottom" use-attribute-sets="common.border__bottom">
    <xsl:attribute name="border-bottom-color">blue</xsl:attribute>
</xsl:attribute-set>
<xsl:attribute-set name="table__tableframe__right" use-attribute-sets="common.border__right">
    <xsl:attribute name="border-right-color">blue</xsl:attribute>
</xsl:attribute-set>
<xsl:attribute-set name="table__tableframe__left" use-attribute-sets="common.border__left">
    <xsl:attribute name="border-left-color">blue</xsl:attribute>
</xsl:attribute-set>ou
```

7. Edit your PDF transformation scenario and set the parameter **customization.dir** to point to your customization folder.
8. Publish and enjoy :)

If you want to create a **DITA Open Toolkit** plugin to achieve the same result you can use the **dita.xsl.xslfo** plugin extension to contribute your own XSLT stylesheet to the publishing process: Creating a simple DITA Open Toolkit plugin to customize published HTML and PDF content *(on page 216)*

Possibilities to obtain PDF from DITA

Lately while attending Tekom 2015 and DITA OT Day 2015 I happened to come across two more DITA to PDF solutions that I did not know about so I tried to compile below a list of all DITA to PDF possibilities that I know of, along with some details:

- **DITA Map PDF - based on HTML5 & CSS**. This commercial plugin is developed and maintainced by Oxygen as an alternative to the default PDF publishing. It allows styling the DITA content using CSS and publishing to PDF using our own **Chemistry** engine or other commercial solutions like **Prince XML** or **Antenna House**.
- Default DITA to PDF solution embedded in the DITA Open Toolkit distribution. Besides Apache FOP you can also use RenderX XEP or Antenna House for publishing. It's customization can be done either via a plugin or via a PDF customization folder: https://www.oxygenxml.com/doc/ug-editor/index.html#topics/dita_pdf_output_customization.html.
- Antenna House DITA OT plugin for producing PDF from DITA using Antenna House. There is also a DITA specialization plugin which allows you to set specific formatting attributes directly to DITA elements.
- The TopLeaf XML Publisher commercial product allows you to produce various outputs like PDF and EPUB from XML content. The application also has a visual PDF layout formatter, it comes also with a DITA OT plugin and has installation instructions for Oxygen.
- Miramo, again a commercial solution which recently started developing support for DITA using a DITA OT plugin. Possibly user based DITA to PDF publishing will be free and server based production will require buying a license. Like TopLeaf, it also has a visual PDF layout designer.
- Adobe FrameMaker can also publish to PDF using its own solution.

Now the question would be, why there are so many solutions available? In my opinion all these solutions are available because the default DITA to PDF output still requires quite a lot of XSLT skills in order to customize.

What do you think?

DITA PDF publishing - Force page breaks between two block elements

Let's say that at some point in your DITA content you have two block level elements, like for example two paragraphs:

```
<p>First para</p>

<p>Second para</p>
```

and you want to force in the PDF output a page break between them.

Here's how a DITA Open Toolkit plugin which would achieve this could be implemented:

1. You define your custom processing instruction which marks the place where a page break should be inserted in the PDF, for example:

```
<p>First para</p>
<?pagebreak?>
<p>Second para</p>
```

2. In the **DITA Open Toolkit** distribution in the `plugins` directory you create a new plugin folder named for example `pdf-page-break`.
3. In this new folder create a new `plugin.xml` file with the content:

```
<plugin id="com.yourpackage.pagebreak">
  <feature extension="package.support.name" value="Force Page Break Plugin"/>
  <feature extension="package.support.email" value="support@youremail.com"/>
  <feature extension="package.version" value="1.0.0"/>
  <feature extension="dita.xsl.xslfo" value="pageBreak.xsl" type="file"/>
</plugin>
```

   The most important feature in the plugin is that it will add a new XSLT stylesheet to the XSL processing which produces the PDF content.
4. Create in the same folder an **XSLT** stylesheet named `pageBreak.xsl` with the content:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:fo="http://www.w3.org/1999/XSL/Format"
    version="1.0">
  <xsl:template match="processing-instruction('pagebreak')">
    <fo:block break-after="page"/>
  </xsl:template>
</xsl:stylesheet>
```

5. Install your plugin in the **DITA Open Toolkit** by running the DITA OT ANT integrator task.

   If you are running the publishing from **Oxygen XML Editor** you can use the predefined transformation scenario: https://www.oxygenxml.com/doc/ug-oxygen/#topics/dita-ot-install-plugin.html.

   If you run DITA OT from the command line please follow these guidelines: http://www.dita-ot.org/2.0/dev_ref/plugins-installing.html.

Embedding Diagrams in DITA topics using PlantUML

PlantUML is a neat library that takes textual descriptions of diagrams and produces visual diagrams from them.

Suppose you have a DITA topic that embeds such a textual description of a diagram: https://github.com/oxygenxml/dita-ot-diagrams-plugin/blob/master/com.oxygenxml.diagrams.svg/samples/testPlantUMLDiagram.dita.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE topic PUBLIC "-//OASIS//DTD DITA Topic//EN" "topic.dtd">
<topic id="testPlantUML">
    <title>Test diagram</title>
    <body>
        <p>
            <foreign outputclass="embed-plant-uml">
                @startuml
                Alice -> Bob: Authentication Request
                Bob --> Alice: Authentication Response

                Alice -> Bob: Another authentication Request
                Alice &lt;-- Bob: Another authentication Response
                @enduml</foreign>
        </p>
    </body>
</topic>
```

To provide support in Oxygen for using such diagrams in DITA topics, you need to install the following components:

- An Oxygen plugin that presents PlantUML textual descriptions as images when editing the content in the **Author** visual editing mode: https://github.com/oxygenxml/wsaccess-javascript-sample-plugins/tree/master/plantuml-images-support.
- A DITA Open Toolkit publishing plugin that, once installed in the DITA Open Toolkit, will convert the textual descriptions to diagrams for both the HTML-based and PDF outputs: https://github.com/oxygenxml/dita-ot-diagrams-plugin.

Once you have this setup, you can both visualize the diagrams when editing and publish them while having the diagram representations dynamically converted to SVG in the published output.

Related information

Embedding Diagrams in DITA Topics Using Mermaid *(on page 194)*
Presenting DITA Task Steps as Diagrams *(on page 195)*

Embedding Diagrams in DITA Topics Using Mermaid

Mermaid is a neat library that takes textual descriptions of diagrams and produces visual diagrams from them.

Suppose you have a DITA topic that embeds such a textual description of a diagram: https://github.com/oxygenxml/dita-ot-diagrams-plugin/blob/master/com.oxygenxml.diagrams.svg/samples/testMermaidDiagram.dita.

```
<topic id="testMermaidDiagram">
    <title>Test Mermaid diagram</title>
    <body>
        <p>
            <foreign outputclass="embed-mermaid-diagram">graph TD;
                A-->B;
                A-->C;
                B-->D;
                C-->D;</foreign>
        </p>
    </body>
</topic>
```

Test Mermaid diagram

To provide support in Oxygen for publishing to HTML using such diagrams in DITA topics, you need to install this free publishing plugin that, once installed in the DITA Open Toolkit, will convert the textual descriptions to SVG diagrams for both the HTML-based and PDF outputs: https://github.com/oxygenxml/dita-ot-diagrams-plugin.

> **Tip:**
> You can preview the created diagrams using the Mermaid live editor: https://mermaid.live/edit.

Related information

Embedding Diagrams in DITA topics using PlantUML *(on page 193)*
Presenting DITA Task Steps as Diagrams *(on page 195)*

Presenting DITA Task Steps as Diagrams

By using our free DITA Open Toolkit plugin, you can embed diagram text descriptions (either in PlantUML *(on page 193)* or Mermaid *(on page 194)* format) in DITA topics and have them rendered as diagram images in the published output.

I experimented with dynamically constructing diagrams (using the Mermaid *(on page 195)* framework) for DITA tasks at the time of publishing.

Suppose you have a DITA task with a set of steps.

```
<steps>
  <step>
    <cmd>Begin by cutting out all the dead branches.</cmd>
    <substeps>
      <substep>
        <cmd>Find shears</cmd>
      </substep>
      ....
    </substeps>
  </step>
  <step>
    <cmd>Remove all tangled or crossed over branches. This allows air to circulate and
      reduces bug and fungi infestation. </cmd>
  </step>
  .......
</steps>
```

A DITA Open Toolkit plugin can use an XSLT customization to dynamically create a Mermaid text diagram representation from the DITA task steps:

```
  <xsl:template match="steps[step]">
    <xsl:variable name="var">
      <foreign outputclass="embed-mermaid-diagram" class="- topic/foreign ">
stateDiagram-v2
Title: <xsl:value-of select="normalize-space(ancestor::task/title)"/>
Title --> State1
<xsl:for-each select="step">
<xsl:variable name="statePos" select="position()"/>
  State<xsl:value-of select="$statePos"/>: <xsl:value-of select="normalize-space(cmd)"/>
<xsl:if test="position() > 1">
  State<xsl:value-of select="$statePos - 1"/> --> State<xsl:value-of select="$statePos"/>
</xsl:if>
<xsl:if test="substeps/substep">
<xsl:for-each select="substeps/substep">
State<xsl:value-of select="$statePos"/><xsl:value-of select="position()"/>: <xsl:value-of select="normalize-space(cmd)
"/>
</xsl:for-each>
state State<xsl:value-of select="$statePos"/>{
```

```
<xsl:for-each select="substeps/substep">
<xsl:if test="position() > 1">
State<xsl:value-of select="$statePos"/><xsl:value-of select="position() - 1"/> -->
 State<xsl:value-of select="$statePos"/><xsl:value-of select="position()"/>
</xsl:if>
</xsl:for-each>
}
</xsl:if>
</xsl:for-each>
      </foreign>
    </xsl:variable>
    <xsl:apply-templates select="$var"/>
    <xsl:next-match/>
  </xsl:template>
</xsl:stylesheet>
```

A complete DITA Open Toolkit that contains the customization can be found here: resources/com.oxygenxml.task.diagram.zip.

At the time of publishing, the task's HTML page will contain this diagram inside it:

Related information

Embedding Diagrams in DITA topics using PlantUML *(on page 193)*
Embedding Diagrams in DITA Topics Using Mermaid *(on page 194)*

Show Consecutive DITA Code Blocks in Separate Tabs

When inserting `<codeblock>` elements in DITA content with code from various languages, you can use the `@outputclass` attribute to provide syntax highlighting for each codeblock in the generated WebHelp and PDF outputs: https://www.oxygenxml.com/doc/ug-editor/topics/add-syntax-highlights-codeblocks-x.html.

Suppose you have consecutive `<codeblock>` elements, each one of them displaying how something can be accomplished in a particular programming language, like this:

```
<codeblock xml:space="preserve" outputclass="ruby">require 'rest-client'

require 'json'
```

```
headers = {
  'Content-Type' =&gt; 'application/octet-stream',
  'Accept' =&gt; 'application/json',
  'Authorization' =&gt; 'Bearer {access-token}'
}
</codeblock>
    <codeblock xml:space="preserve" outputclass="python">import requests
headers = {
  'Content-Type': 'application/octet-stream',
  'Accept': 'application/json',
  'Authorization': 'Bearer {access-token}'
}</codeblock>
```

You can install this open-source DITA Open Toolkit plugin to display codeblocks as separate tabs in the published WebHelp output: https://github.com/oxygenxml/dita-wh-codeblock-tabs.

The WebHelp output should look like this:

Code samples

Ruby | Python

```
require 'rest-client'
require 'json'

headers = {
  'Content-Type' => 'application/octet-stream',
  'Accept' => 'application/json',
  'Authorization' => 'Bearer {access-token}'
}
```

Show DITA Task Steps in Separate Tabs

You can present individual DITA task steps in separate tabs in the generated WebHelp output by using this free plugin: https://github.com/oxygenxml/dita-wh-tabs.

Suppose you have a DITA task that contains steps like this:

```
<steps>
    <step>
        <cmd>Begin by cutting out all the dead branches.</cmd>
    </step>
    ....
    <step>
        <cmd>Clean up the area.</cmd>
    </step>
</steps>
```

You can install this open-source DITA Open Toolkit plugin to display the steps as separate tabs in the published WebHelp output: https://github.com/oxygenxml/dita-wh-tabs. To enable generating separate tabs for each DITA task step, you also need to set the `tabs.for.steps` publishing parameter to the value of `true`.

The WebHelp output should look like this:

Converting OpenAPI to DITA XML and Publishing to WebHelp (Experimental)

OpenAPI documentation is usually stored in JSON or YAML formats. In this topic, we will attempt to convert OpenAPI documentation to Markdown using widdershins, convert Markdown to DITA content using the Oxygen Batch Converter add-on, and then publish to WebHelp Responsive output.

1. Download the Yaml or JSON representation of an OpenAPI specification to the local disk. For this example, I used the OpenAPI documentation from the Oxygen Content Fusion server: https://fusion.oxygenxml.com/api/swagger.json.
2. Download and install widdershins.
3. From a command prompt or terminal, produce a Markdown representation of the original API file:

```
widdershins --search false --language_tabs 'ruby:Ruby' 'python:Python' --summary /../swagger.json -o swagger.md
```

4. In the Oxygen application, use the **Tools > Batch Documents Converter > > Markdown to DITA** conversion to convert the Markdown file to a DITA Map.

5. Once you have the DITA Map, you can publish it to WebHelp Responsive or integrate it as part of a larger DITA Map. The map can be further modified by adding chunking attributes to avoid breaking the published output in too many HTML files.
6. When publishing the `swagger.ditamap` to WebHelp, we set the `validate` parameter to `no` because the produced DITA content had some invalid content inside.
7. The resulting WebHelp output is available here: https://blog.oxygenxml.com/resources/open-api-webhelp/returns_the_current_user%27s_details.html. Codeblocks that contain sample source code are divided into multiple tabs using this plugin: Show Consecutive DITA Code Blocks in Separate Tabs *(on page 198)*.

DITA to Confluence

Atlassian Confluence is a content collaboration tool used to help teams collaborate and share knowledge efficiently. Based on my limited knowledge of Confluence, I've identified several ways that you can write technical documentation in DITA and then make it available on the Confluence platform:

Publish DITA to Microsoft Word and import in Confluence

Oxygen XML Editor comes bundled with support to publish DITA content to Microsoft Word using the DITA to Word plugin: https://www.oxygenxml.com/doc/ug-editor/topics/ditamap-ms-word.html.

Once you publish the DITA project to Word, you can use the **Import Word Document** action in Confluence to import the Word document.

Share
Attachments (0)
Page History
Restrictions
Analytics
Import documents as Confluence pages
Page Information
Resolved comments
Link to this Page...
View in Hierarchy
View Source
View Storage Format
Export to PDF
Export to Word
Import Word Document
Slack Notifications
Copy
Move
Delete
Earth - Spring, Su
ound throughout

During the import process, you will be able to choose to split the document into multiple pages depending on the headings.

Publish DITA to HTML5 and import in Confluence

Although Confluence supports out-of-the-box importing only from Microsoft Word, there is a commercial file importer plugin for Confluence: https://marketplace.atlassian.com/apps/1221333/all-in-one-file-importer-for-confluence. This plugin can import various file formats to Confluence, including Markdown and HTML.

The plugin contributes an import action that can be used to either import individual HTML or Markdown files or entire folders. The plugin might try to import various unwanted file formats (such as CSS and image files) so these types of files need to be manually deleted before the import process. Also, the plugin does not seem to properly resolve links between the imported HTML files.

Publish DITA to HTML5 and paste from Web Browser to Confluence

If you just want to update a few pages, you can publish DITA to HTML, open the published HTML content in a web browser, select and copy the contents, then paste in a Confluence page. Confluence seems capable of converting the pasted HTML content to its own HTML-like format. Some manual cleanup may be required after pasting the content.

Creating a special DITA to Confluence-like HTML publishing flow

The internal storage format Confluence uses is a kind of HTML mixed with XML content: https://confluence.atlassian.com/doc/confluence-storage-format-790796544.html.

If a DITA OT plugin were to implement special DITA to Confluence-like HTML output, it would probably need special handling for:

- Internal links.
- Links to binary resources.
- Images.

Generating a list of changes between releases

Sometimes you may had PDFs or HTML content with the documentation to reviewers and instead of them having to read again the entire user's manual you may want to give them also a list of changed topics.

Such a list can be manually maintained in a separate DITA topic but if your project uses version control and you tag your releases you may also want to try and manually build this list of changes.

For DITA projects which use Git for source control you can use a modified version of the DITA Open Toolkit custom ANT parser plugin to automatically generate a list of changes between two versions in the published output. I will enumerate the steps to do this below:

- Check out your Git project. If all your releases are tagged as branches, running a command line like this:

  ```
  git diff --name-only tagName
  ```

  will show in the console a set of changed topics between the current changes and the `tagName` branch. For example for the Oxygen XML Editor user's manual I can run `git diff --name-only oxygen22.0` to see all changes between the current contents in the repository and the Oxygen 22.0 branch in the repository.
- In the main DITA Map add a reference to a topic which will be the recipient of the generated content at publishing time:

  ```
  <topicref href="dynamicChangeHistory.gen" format="ant-parser"/>
  ```

  Create this `dynamicChangeHistory.gen` topic next to the DITA Map but leave it with empty contents, the publishing plugin will be responsible of updating its contents.
- Download the `com.oxygenxml.ant.parser.dita` plugin from here: https://github.com/oxygenxml/dita-custom-ant-parser/tree/generate-changes-list.
- Integrate/install the plugin in your DITA Open Toolkit `plugins` folder.
- Open the build file `com.oxygenxml.ant.parser.dita/resources/build.xml` in Oxygen XML Editor. The build file has three stages which may need to be slightly changed to fit your project's structure:
  - Run the `git` command in your project's folder and find all differences related to a certain branch. In my case the branch is called `oxygen22.0` so it needs to be changed to your particular branch name.
  - Apply the XSLT stylesheet `createChangesTopic.xsl` on the changes list and generate a DITA topic. You will need to make changes to the stylesheet so that the generated paths to the changed topics are relative to the place where the generated DITA topic is located.
  - Run an XSLT stylesheet on the DITA topic to add the `@class` attributes.
- Once the plugin is properly configured, an extra topic will be produced in the output (either PDF or HTML) containing links to the changes list.

DITA Publishing Customization Overview

Oxygen XML Editor publishes **DITA** content to various output sources using a bundled version of the DITA Open Toolkit (short name: **DITA OT**) publishing engine. Oxygen versions prior to version 20 come with two **DITA OT** versions:

- **DITA OT 1.8** located in the *OXYGEN_INSTALL_DIR\frameworks\dita\DITA-OT* folder.
- **DITA OT 2.x** located in the *OXYGEN_INSTALL_DIR\frameworks\dita\DITA-OT2.x* folder.

while Oxygen version 20 comes bundled only with **DITA OT 2.x** located in the folder specified above. You can find the exact version of DITA OT bundled with Oxygen by looking in the main menu **Help->About** at the **Frameworks** tab.

DITA Open Toolkit Overview

The DITA Open Toolkit is an open source publishing engine that can publish **DITA** content to various output sources such as **XHTML**, **PDF**, or **Windows Help (CHM)**. Since it has a plugin-based architecture, it can be extended with extra plugins that either define new formats for conversion or customize an existing conversion format. You can run the **DITA OT** from Oxygen using its transformation scenarios or you can run it directly from a command line:http://www.dita-ot.org/dev/topics/building-output.html.

The **DITA OT** bundled with Oxygen contains more plugins than the standard **DITA OT** that can be downloaded from the DITA OT official web site. For example, it contains pre-installed plugins for converting DITA content to Word, EPUB, Oxygen WebHelp, or to publish to PDF using CSS to customize the output: Useful DITA OT Plugins *(on page 214)*.

Publishing Customizations (Before you begin)

Some customizations, usually for HTML-based output, can be made simply by creating a custom CSS and they do not involve modifying the **DITA OT** engine in any way. But most customizations might involve adding a new plugin to the **DITA OT**. So here are some best practices before you begin your customization:

1. Copy the bundled **DITA OT** folder (usually *OXYGEN_INSTALL_DIR\frameworks\dita\DITA-OT2.x*) to a location outside of Oxygen. This will allow you to have full write access to the folder in order to install new plugins:https://www.oxygenxml.com/doc/ug-editor/topics/dita-ot-install-plugin.html.
2. In the Oxygen **Preferences->DITA** page, set the default DITA OT distribution to be the external copied one. This will mean that Oxygen will use the external **DITA OT** for DITA validation and publishing. After doing this, you will be able to upgrade the Oxygen version and benefit from all editing-related improvements without affecting the publishing system.

3. Share that external **DITA OT** copy with the rest of the team. If you are using a repository like Subversion or Git for collaboration *(on page 465)*, you can commit the entire modified DITA OT publishing engine as part of your project. This will allow everybody else in your team to use the official changes that you made. This will also allow you to set up some kind of automatic publishing system using an open-source integration server like Travis.

Customizing the XHTML-based outputs

Usually **XHTML**-based outputs can be modified by using a custom **CSS** stylesheet to override various styles. If you edit an **XHTML**-based transformation scenario in Oxygen, there is a parameter called **args.css** that can be set to point to your custom CSS and a parameter called **args.copy.css** that can be set to **yes** to copy the CSS to the output folder. To know what CSS styles to override, you can use your web browser's CSS Inspector tools to look at the produced HTML output. The same parameters can be set when publishing from a command line: http://www.dita-ot.org/dev/topics/html-customization-css.html.

You can also create plugins to customize the XHTML-based outputs by adding an extra XSLT stylesheet: for example, Creating a simple DITA Open Toolkit plugin to customize published HTML and PDF content *(on page 216)*. A list with all DITA OT XSLT extension points can be found here: http://www.dita-ot.org/dev/extension-points/plugin-extension-points-xslt-import.html.

Customizing the Oxygen WebHelp-based output

Oxygen's **DITA OT** comes bundled with specific plugins developed by Oxygen that allows publishing **DITA** content to **WebHelp Responsive** output. Oxygen's User's Guide has detailed topics about how to customize these outputs: https://www.oxygenxml.com/doc/ug-editor/topics/wh-responsive-customization-tips.html.

Customizing the PDF Classic output

The **DITA to PDF** output can be customized either by creating a PDF customization folder (in this case the DITA OT folder will not be modified at all) or by creating a PDF customization plugin: https://www.oxygenxml.com/doc/ug-editor/topics/dita-pdf-output.html. There is also a book called DITA For Print that contains details about how to customize various aspects.

Customizing the DITA + CSS to PDF output

In recent versions, Oxygen added a new transformation scenario called **DITA Map PDF - WYSIWYG** that allows you to use **CSS** to style the PDF output, lowering the required knowledge for implementing a PDF customization. There is an entire chapter in the Oxygen users manual covering various PDF customization details: https://www.oxygenxml.com/doc/ug-editor/topics/dcpp_the_customization_css.html#dcpp_the_customization_css.

DITA Open Toolkit Frequently Asked Questions (FAQ)

I tried to put together below a set of frequently asked questions and useful links about the **DITA Open Toolkit**.

What is the DITA Open Toolkit?

The **DITA Open Toolkit** is a publishing tool used to convert **DITA** content into various output formats. It's official web site, containing download links and documentation can be found here: http://www.dita-ot.org/.

How do I use the DITA Open Toolkit?

You can download, install and build output from **DITA** content using the command line.

Besides this, there are applications which come with the **DITA Open Toolkit** bundled. For example Oxygen XML Editor comes bundled usually with the latest bundled **DITA Open Toolkit**. Oxygen provides visual means to run the bundled DITA Open Toolkit using a concept called transformation scenarios.

What version of DITA Open Toolkit should I use?

You should try to use the latest DITA Open Toolkit release available on the official download page.

What outputs can I obtain using the DITA Open Toolkit?

The entire set of default available output formats is available here: https://www.dita-ot.org/dev/topics/output-formats.html. But the **DITA Open Toolkit** can be enhanced by installing plugins to provide additional output formats.

What is the general architecture of the DITA Open Toolkit?

The **DITA Open Toolkit** is a quite large mixture of ANT build scripts, Java libraries and XSLT scripts. It has a pipeline-based architecture which uses plugins to publish DITA content to various output formats. Most of the DITA Open Toolkit customizations that you want to make in order to add new publishing capabilities or to customize existing publishing choices can be made without modifying its internal core.

What is a DITA Open Toolkit plugin?

A **DITA Open Toolkit** plugin can either provide a new publishing format, customize an existing publishing stage or provide a **DITA** specialization vocabulary. The plugin can use one of the numerous extension points available in the **DITA Open Toolkit**: https://www.dita-ot.org/dev/extension-points/plugin-extension-points.html.

Once you have created a plugin you can install it in the **DITA Open Toolkit** either by manual installation or using the new automated installation procedure.

How do I customize the HTML-based outputs?

There are a number of parameters which can be set to customize the HTML-based outputs:http://www.dita-ot.org/dev/parameters/parameters-base-html.html. For example you can specify your own **CSS** stylesheet to be used with the generated HTML output.

You can also create a plugin to customize the HTML outputs by adding a custom XSLT stylesheet: Creating a simple DITA Open Toolkit plugin to customize published HTML and PDF content *(on page 216)*.

How do I customize the PDF output?

Oxygen comes bundled with a DITA Open Toolkit plugin which uses CSS to style the DITA content and produce PDF: https://www.oxygenxml.com/doc/ug-editor/topics/pdf-css-customization.html. This plugin is free to use from inside Oxygen but is part of a commercial product if you want to automate the publishing on the server side. This is our recommended plugin for publishing DITA to PDF.

In addition the DITA Open Toolkit comes bundled with a free PDF generation plugin which uses XSL-FO. The PDF output is obtained by passing the original **DITA** content to **XSL-FO** and then generating **PDF** using an **XSL-FO** processor. The default bundled and used XSL-FO processor is the Apache FOP but you can also install separately and use commercial PDF processors like Antenna House or RenderX XEP.

You can customize the **PDF** output either using a PDF customization folder or by creating a PDF customization plugin *(on page 216)*.

There are a number of other solutions for obtaining **PDF** from **DITA**: Possibilities to obtain PDF from DITA *(on page 191)*.

Enhancing DITA Publishing With Plugins

The DITA Open Toolkit publishing engine comes with support for predefined output formats such as HTM5, PDF, and Eclipse Help. As the architecture of the publishing engine is plugin-based, over time we have developed lots of useful plugins in the **Oxygen XML GitHub** account that enhance the publishing and in this blog post, I will enumerate some of them. For the plugins that are already installed within **Oxygen XML Editor's** DITA Open Toolkit engine, I added the **[Bundled]** marker.

Plugin that Converts DITA Maps to PDF Using CSS 3 [Bundled]

Maybe our most important work so far, this plugin can publish DITA to PDF using CSS. As a publishing engine, it can use either our **Oxygen XML Chemistry** processor (freely bundled with **Oxygen XML Editor**) or the Antenna House and Prince XML engines.

DITA Metrics Report [Bundled]

This is a very useful open source plugin that generates an HTML report from an existing DITA project containing lots of useful information:

- Total number of maps and topics that are part of the project.
- Total number of elements used in topics and maps along with a table presenting all element names and their usage counter.
- The used elements used from each DITA domain.
- Total number of attributes used in topics and maps along with a table presenting all attribute names and their usage counter.
- Statistics about the conditional attributes used in the project.
- Information about content reuse.
- Text and content statistics, including both total words (word count) and unique words (vocabulary).
- List of largest and smallest topics and the number of words each one used.
- Listing of all links to resources outside of the project.

Export DITA Map Plugin [Bundled]

You can use this free plugin to create a ZIP file from your entire DITA project. The plugin also takes filters/profiling into account when including topics.

Publish DITA Content with References to Video and Audio Resources. [Bundled]

A DITA Open Toolkit plugin that converts the DITA `<object>` element to various HTML 5 structures such as <`<video>`, `<audio>`, or `<iframe>`.

Plugin That Adds Edit Links in HTML or PDF-based Outputs [Bundled]

This plugin adds edit links in the HTML or PDF-based outputs allowing subject matter experts to give feedback on the published content directly using a DITA web editing tool (such as **Oxygen XML Web Author**).

Create Single Merged XML Document From Entire DITA Project [Bundled]

This plugin produces a merged output from the entire DITA map structure without further processing. It is useful if you want to further process the merged XML document for producing various reports.

Dynamically Publish Excel Content as DITA

A DITA Open Toolkit plugin that dynamically converts to DITA Excel files referenced with `format="excel"` in DITA maps.

Dynamically Use JSON Content in DITA Topics

A DITA Open Toolkit plugin that dynamically converts to DITA JSON files referenced with `format="json"` in DITA maps.

Embed HTML Content in DITA Topics [Bundled]

A plugin that allows embedding well-formed HTML content in a DITA topic inside a special element.

Embed LateX Equations in DITA Content

A DITA Open Toolkit plugin that allows publishing embedded Latex mathematical equations to HTML and PDF.

Embed UML Diagrams in DITA Content

A DITA Open Toolkit plugin that allows publishing embedded UML diagrams equations to HTML and PDF.

Float Images in HTML and PDF Outputs

A plugin that allows floating an image referenced in a DITA topic left or right depending on the `@outputclass` attribute value specified on it.

Embed Referenced MathML and SVGZ Images in HTML Output

A DITA Open Toolkit plugin that allows you to embed referenced MathML and SVG images in the HTML5 and XHTML output.

Dynamically Convert DITA Tables to Graphs

A DITA Open Toolkit plugin that converts DITA tables having a certain structure to SVG graphs.

Show Oxygen Change Tracking Information in the PDF Output [Bundled]

This plugin allows you to see **Oxygen XML Editor** track changes (insertions, deletions, or comments) in the PDF output.

Sample Customization Plugin for Classic PDF Output

This sample DITA Open Toolkit PDF customization plugin is a good starting point if you want to:

- Customize fonts
- Customize a cover page to provide custom logos and coloring
- Customize page headers and footers

PDF - Generate Numbers Before Topic's Title

A DITA-OT PDF2 customization plugin that can be installed to generate numbers before each topic's title.

PDF Plugin That Presents Chapters With Landscape Orientation

A PDF customization folder that allows you to define landscape orientation for a certain chapter.

Adding support for embedding LaTeX equations in DITA content

LaTeX is a high-quality typesetting system that includes features designed for the production of technical and scientific documentation. LaTeX can also be used to express mathematical formulas in a textual format. By default, web browsers and PDF readers do not have support to show mathematical equations written in LaTeX but there are open source projects that can read LaTeX and convert it to other image types.

Adding support for writing LaTeX equations in DITA topics would imply three stages:

- Find a way to write the equation in the DITA XML content. You can either create a DITA DTD specialization and add a new element (for example, called `<latex>` and it extends the DITA `<foreign>` element). Alternatively, you can use the DITA `<foreign>` element with a specific `@outputclass` attribute value directly:

```
<!DOCTYPE topic PUBLIC "-//OASIS//DTD DITA Topic//EN" "topic.dtd">
<topic id="testEquation">
    <title>Test equation</title>
    <body>
        <p><foreign outputclass="embed-latex">L' = {L}{\sqrt{1-\frac{v^2}{c^2}}}</foreign></p>
```

```
    </body>
</topic>
```

- If you want Oxygen to properly present the LaTeX equation when editing in the **Author** visual mode, you need an Oxygen plugin that converts the equation content to an image. There is a sample plugin that does that here: https://github.com/oxygenxml/wsaccess-javascript-sample-plugins/tree/master/latex-images-support. You can download and copy the plugin folder **latex-images-support** to the Oxygen **plugins** folder, then restart Oxygen.
- Publishing to HTML-based and PDF outputs. This DITA Open Toolkit plugin automatically converts LaTeX images to SVG when publishing: https://github.com/oxygenxml/dita-latex.

Useful DITA OT Plugins

The **DITA Open Toolkit** is the most commonly used engine for publishing **DITA** content to various output formats. It has a plugin-based architecture and it comes bundled by default with plugins for publishing to **XHTML**-based and **PDF** formats. There are other useful open-source plugins that can be installed in the **DITA Open Toolkit** and provide various publishing or reporting capabilities. Below is a list of my favorite open source plugins, many of which are bundled by default with **Oxygen XML Editor**:

DITA to MS Word

This very useful plugin developed by Jarno Elovirta (the main DITA OT developer) can be used to publish **DITA** to **MS Word (OOXML)**: https://github.com/jelovirt/com.elovirta.ooxml. The plugin is robust and can handle large DITA projects.

Convert MS Word to DITA

The **DITA For Publishers** project developed by **Eliot Kimber** contains a plugin that can convert **MS Word (OOXML)** documents to DITA topics: http://www.dita4publishers.org/d4p-users-guide/user_docs/d4p-users-guide/word2dita/word-to-dita-plugin.html. Useful advice and other ways to migrate from **MS Word** to **DITA** can be found here: How to Migrate from Word to DITA *(on page 357)*.

DITA to EPUB

The **DITA For Publishers** project developed by **Eliot Kimber** contains a plugin that can convert **DITA** content to **EPUB3**: http://www.dita4publishers.org/d4p-users-guide/user_docs/d4p-users-guide/epub-plugin/generating-epubs.html. There are also plugins that can take the **EPUB3** output and generate **Kindle** or **Mobi** formats.

Use Markdown content directly in DITA Maps

This plugin developed by Jarno Elovirta can be used to publish hybrid DITA projects containing both DITA and Markdown topics to any output format: https://github.com/jelovirt/org.lwdita.

Embed video and audio resources in DITA topics

This handy little plugin developed by **Oxygen** allows you to embed audio, video, or YouTube resources in **DITA** topics and properly display the embedded content in the XHTML-based outputs: https://github.com/oxygenxml/dita-media-support.

Publish RelaxNG-based DITA topics

RelaxNG is the default schema type in which the **DITA** vocabulary is described by the specification developers. But the **DITA Open Toolkit** does not allow publishing RNG-based **DITA** content by default. Installing this plugin developed by **Oxygen** will make this possible: https://github.com/oxygenxml/dita-relaxng-defaults.

Present Oxygen track changes in the PDF output

This useful plugin developed by **Oxygen** allows you to preserve **Oxygen** change tracking highlights in the PDF output: https://github.com/oxygenxml/dita-classic-pdf-review.

Publish DITA to PDF using CSS

The difficult thing about the default PDF output is the customization that involves modifying XSLT scripts and requiring knowledge of both **XSLT** and **XSL-FO**. This plugin developed by **Oxygen** allows you to use **CSS** to customize the **PDF** obtained from **DITA** content: https://github.com/oxygenxml/dita-css. The plugin needs a commercial license of **RenderX XEP** or **Antenna House** in order to produce the PDF: https://www.oxygenxml.com/doc/ug-editor/topics/ditamap-pdf-css-html5-2.html. But for **Oxygen XML Editor 19**, we are actively working on a **PDF CSS** engine (called **Chemistry**) that will allow the plugin to work from **Oxygen** without any additional licenses: http://archive.xmlprague.cz/2017/files/presentations/css2pdf-xslfo.pdf.

A list with all open-source DITA OT plugins developed and shared by **Oxygen XML Editor** can be found on **GitHub**: https://github.com/search?q=topic%3Adita-ot-plugin+org%3Aoxygenxml&type=Repositories. My **DITA OT Day 2016** presentation covers most of these: https://www.oxygenxml.com/events/2016/dita-ot_day.html#Seven_Open-Source_DITA-OT_plugins.

Creating a simple DITA Open Toolkit plugin to customize published HTML and PDF content

I recently worked on a **DITA Open Toolkit** plugin which can be used to provide syntax highlight when publishing DITA **codeblock** elements to **HTML**-based or **PDF** outputs.

Based on this experience I've put together some steps to help anyone wanting to create an **XSLT** customization plugin for the **DITA Open Toolkit** for HTML and PDF based outputs.

1. Create a folder for your plugin in the DITA OT **plugins** folder. The **DITA OT** bundled with Oxygen can be found here: *OXYGEN_INSTALL_DIR/frameworks/dita/DITA-OT*. In my case I created the following folder: *OXYGEN_INSTALL_DIR/frameworks/dita/DITA-OT/plugins/com.oxygenxml.highlight*.
2. Create a **plugin.xml** file in that folder containing the plugin's extension points. In my case, the plugin descriptor file contains:

```
<plugin id="com.oxygenxml.highlight">
  <feature extension="package.support.name" value="Oxygen XML Editor Support"/>
  <feature extension="package.support.email" value="support@oxygenxml.com"/>
  <feature extension="package.version" value="1.0.0"/>
  <feature extension="dita.xsl.xhtml" value="xhtmlHighlight.xsl" type="file"/>
  <feature extension="dita.xsl.xslfo" value="pdfHighlight.xsl" type="file"/>
</plugin>
```

   The important extensions in it are the references to the XSLT stylesheets which will be used to style the HTML and the PDF outputs. You can find a bunch of other DITA OT plugin extension points here:https://www.dita-ot.org/dev/extension-points/plugin-extension-points.html.
3. Create an XSLT stylesheet called **xhtmlHighlight.xsl** located in the same plugin folder. As I want to overwrite the creation of the HTML content from a DITA **codeblock** element I will first need to find the XSLT template that I need to overwrite. A DITA **codeblock** element has the **class** attribute value "**+ topic/pre pr-d/codeblock** ". Usually in such cases I take part of the class attribute value and search using the "**Find/Replace in Files**" Oxygen action in all of the DITA OT XSLT resources. In this case I searched for **topic/pre**and found this XSLT stylesheet:*OXYGEN_INSTALL_DIR/frameworks/dita/DITA-OT/plugins/org.dita.xhtml/xsl/xslhtml/dita2htmlImpl.xsl* containing this XSLT template:

```
<xsl:template match="*[contains(@class, ' topic/pre ')]" name="topic.pre">
  <xsl:if test="contains(@frame, 'top')"><hr /></xsl:if>
  <xsl:apply-templates select="*[contains(@class, ' ditaot-d/ditaval-startprop ')]" mode="out-of-line"/>
  <xsl:call-template name="spec-title-nospace"/>
  <pre>
```

```
    <xsl:attribute name="class" select="name()"/>
    <xsl:call-template name="commonattributes"/>
    <xsl:call-template name="setscale"/>
    <xsl:call-template name="setidaname"/>
    <xsl:apply-templates/>
  </pre>
  <xsl:apply-templates select="*[contains(@class, ' ditaot-d/ditaval-endprop ')]" mode="out-of-line"/>
  <xsl:if test="contains(@frame, 'bot')"><hr /></xsl:if><xsl:value-of select="$newline"/>
</xsl:template>
```

thus my **xhtmlHighlight.xsl** will overwrite the content of the template like:

```
<xsl:template match="*[contains(@class, ' topic/pre ')]" name="topic.pre">
  <xsl:if test="contains(@frame, 'top')"><hr /></xsl:if>
  <xsl:apply-templates select="*[contains(@class, ' ditaot-d/ditaval-startprop ')]" mode="out-of-line"/>
  <xsl:call-template name="spec-title-nospace"/>
  <pre>
    <xsl:attribute name="class" select="name()"/>
    <xsl:call-template name="commonattributes"/>
    <xsl:call-template name="setscale"/>
    <xsl:call-template name="setidaname"/>
     <!--Here I'm calling the styler of the content inside the codeblock.-->
     <xsl:call-template name="outputStyling"/>
  </pre>
  <xsl:apply-templates select="*[contains(@class, ' ditaot-d/ditaval-endprop ')]" mode="out-of-line"/>
  <xsl:if test="contains(@frame, 'bot')"><hr /></xsl:if><xsl:value-of select="$newline"/>
</xsl:template>
```

and call another XSLT template which applies as a Java extension the XSLTHL library to style the content.

4. Create an XSLT stylesheet called **pdfHighlight.xsl** located in the same plugin folder which will contain the PDF XSLT customization. In this case I will overwrite the XSLT template from:*OXYGEN_INSTALL_DIR/frameworks/dita/DITA-OT/plugins/org.dita.pdf2/xsl/fo/topic.xsl* which has the content:

```
    <xsl:template match="*[contains(@class,' topic/pre ')]">
        <xsl:call-template name="setSpecTitle"/>
        <fo:block xsl:use-attribute-sets="pre">
            <xsl:call-template name="commonattributes"/>
            <xsl:call-template name="setFrame"/>
```

```
            <xsl:call-template name="setScale"/>
            <xsl:call-template name="setExpanse"/>
            <xsl:apply-templates/>
        </fo:block>
    </xsl:template>
```

5. In order to install your plugin in the DITA OT you need to run the integrator. In the Oxygen **Transformation Scenarios** view there is a **Show all scenarios** action available in the drop down settings button. Just check that and execute the transformation scenario called **Run DITA OT Integrator**. And that's it, your XSLT content will be applied with priority when publishing both to XHTML-based and to PDF-based outputs.

   Let's take now a look at what that misterious step (5) - running the integrator to install the plugin - really did:

   - In the XSLT stylesheet:*OXYGEN_INSTALL_DIR/frameworks/dita/DITA-OT/plugins/org.dita.xhtml/xsl/dita2html-base.xsl* a new import automatically appeared:

     ```
     <xsl:import href="plugin:com.oxygenxml.highlight:xhtmlHighlight.xsl"/>
     ```

     This import is placed after all base imports and because of this it has a higher priority. More about imported template precedence can be found in the XSLT specs:http://www.w3.org/TR/xslt#import.
   - Likewise, in the top-level stylesheets related to PDF publishing like:*OXYGEN_INSTALL_DIR/frameworks/dita/DITA-OT/plugins/org.dita.pdf2/xsl/fo/topic2fo_shell.xsl* a new import statement has appeared:

     ```
     <xsl:import href="plugin:com.oxygenxml.highlight:pdfHighlight.xsl"/>
     ```

   Now you can take your plugin's folder and distribute it to anyone having a DITA OT installation along with some simple installation notes. Your customization will work as long as the templates you are overwriting have not changed from one DITA OT distribution to the other.

   This video from the DITA OT Day 2018 in Rotterdam has also steps for implementing a simple DITA OT plugin: https://www.oxygenxml.com/events/2018/dita-ot_day.html#step_by_step_implementation_of_a_DITA.

Using Git client-side hooks to run DITA publishing pipelines

This topic describes how to use Git client-side hooks to execute DITA commands and publishing workflows, with an emphasis on using the Oxygen's Git Client.

Prerequisites

1. Oxygen's Git client is based on the JGit library. When running on Windows, this library relies on CYGWIN to run the Git hooks. CYGWIN must be installed and added to the path.
2. If you are running on Windows, you either need an Oxygen Git client version 2.0.0 or one newer than 2.1.1 (at the time this post is written, 2.1.1 was the latest version, but a new one will soon be released). We are basically avoiding versions that come bundled with JGit 5.6.0, which has this known issue.

   If you need to install version 2.0.0 of the plugin, since this version is no longer available in the add-on, you have to install it manually:
   a. Uninstall the current Oxygen Git client version, if one is present.
   b. Download Oxygen Git client version 2.0.0.
   c. Unzip it inside `{oxygenInstallDir}/plugins`. Make sure you don't create any intermediate folders. The path should be like this: `{oxygenInstallDir}/git.support-2.0.0/plugin.xml`.
3. The DITA-OT command utility needs to be added to the path so that the presented script will work.

What is a Git hook?

Git has a way to fire off custom scripts when certain important actions occur. There are two groups of these hooks: *client-side* and *server-side*. *Client-side* hooks are triggered by operations such as committing and merging, while *server-side* hooks run on network operations such as receiving pushed commits. You can read more about what types of hooks are available at Customizing Git - Git Hooks.

Using a client side hook with a DITA project

The main scenario we are focusing on in this topic is about using client-side hooks while working with DITA and storing it in a Git repository, and how to enable Oxygen's built-in Git Client to execute these hooks.

The hooks are all stored in the `hooks` subdirectory of the Git directory, which is `.git/hooks` by default. A useful client-side hook is the *pre-commit* hook. It's used to inspect the snapshot that's about to be committed, to see if you've forgotten something, to make sure tests like Oxygen's Validate and Check for completeness run properly, or to examine whatever you need to inspect in the code.

To make it easier to follow, though, we will use a use case reported by one of Oxygen's users. That is to generate a Markdown version of the documentation whenever you commit something. When I tried to do the same thing myself, I encountered some challenges and I think it will help others if they read how I managed to solve it.

Installing a *pre-commit* hook

Git automatically populates the `.git/hooks` directory with example scripts when you initialize a repository. All we need to do is rename *pre-commit.sample* to simply *pre-commit* and put in it the script we are interested in.

> **Note:**
> The DITA-OT command utility needs to be added to the path in order for the following script to work.

The content of the *pre-commit* file is something like this: we assume that the map to publish is called `README.ditamap` and it is located inside the project root directory.

> **Note:**
> Oxygen's Git client is based on the JGit library. When running on Windows, this library relies on CYGWIN to run these Git scripts. CYGWIN must be installed and added to the path.

```
#!/bin/sh
set -x

echo "Start hook"

export GIT_HOOKS_DIR=`cd "\`dirname "\\\`readlink "$0" || echo $0\\\`"\`" && pwd`

# We assume the hooks directory is the default one: wc/.git/hooks
export ROOT_DIR="$GIT_HOOKS_DIR/../.."

 # OS specific support.  $var _must_ be set to either true or false.
cygwin=false;
case "`uname`" in
  CYGWIN*) cygwin=true ;;
esac

# For some reasons, if we let a cygwin path pass, the dita pipeline fails with:
# Error: Failed to run pipeline: [DOTA069F][FATAL] Input file
 'file:/cygdrive/c/Users/.../git-hooks-sample/.git/hooks/../../README.ditamap' cannot be
```

```
 located or read. Ensure that file was specified properly and that you have permission to
 access it.
# It works if we pass it instead as: C:/Users/.../git-hooks-sample/README.ditamap
# Considering the fact that the dita script just passes the --input further on to ANT, it might
 have something to do with cygwin processing performed in ANT.
if $cygwin; then
  ROOT_DIR=`cygpath --mixed "$ROOT_DIR"`
fi



dita --input=$ROOT_DIR/README.ditamap --format=markdown --output=$ROOT_DIR/docs/


# Exit with status of last command
exit
```

The first challenge in the script above is that if I don't pass the ditamap path through `ROOT_DIR=`cygpath --mixed "$ROOT_DIR"``, the DITA publishing pipeline fails with:

```
Error: Failed to run pipeline: [DOTA069F][FATAL] Input file
 'file:/cygdrive/c/Users/.../git-hooks-sample/.git/hooks/../../README.ditamap' cannot be located or read. Ensure that
 file was specified properly and that you have permission to access it.
```

Having done that, running this hook on Windows will still fail, inside the DITA pipeline, with:

```
Buildfile: \cygdrive\d\tools\dita-ot-3.4\build.xml does not exist!
```

To fix this, I had to go inside `{dita.ot.dir}/bin/dita` and search for this line:

```
# Add build script to arguments
ant_exec_args="$ant_exec_args \"-buildfile\" \"$DITA_HOME/build.xml\" \"-main\"
 \"org.dita.dost.invoker.Main\""
```

and replace it with:

```
##################################################
BUILD_FILE="$DITA_HOME/build.xml"
if $cygwin ; then
BUILD_FILE=`cygpath --mixed "$BUILD_FILE"`
fi
##################################################
```

```
# Add build script to arguments
ant_exec_args="$ant_exec_args \"-buildfile\" \"$BUILD_FILE\" \"-main\"
 \"org.dita.dost.invoker.Main\""
```

That's it. Each time you commit something in this repository, the Markdown version of your documentation gets generated.

Creating Link Graphs From DITA Maps

It may be sometimes useful as an overview to obtain a graphical visualization of how your DITA topics refer to each other. The VisJS Javascript library can be used to construct node network diagrams. Here's how you can use it for your DITA project:

1. Open the main DITA Map in the Oxygen XML Editor **DITA Maps Manager** view, use the **Configure Transformation Scenarios** button, then create a new transformation scenario of type **DITA OT Transformation** choosing the **Merge DITA Content to Single XML file** transformation type. If you are not using Oxygen XML Editor you can download and install the **dita-merge** plugin separately from Github.
2. Apply the new transformation scenario on the DITA Map, once the merged XML document is generated, create another transformation scenario of type **XML transformation with XSLT** and transform to HTML using the XSLT stylesheet: https://github.com/oxygenxml/dita-merged/blob/master/possible-applications/createHTMLVisGraph.xsl
3. Open the result HTML document in a web browser. Depending on the complexity of the DITA project it may take some time for the diagram to show up. As an example the related links section below contains a reference to a diagram representation of the Oxygen XML Blog. If we regard the diagram as a circle, on the outer bounds of the circle are topics which have few incoming and outgoing links and in the center topics which have many outgoing and incoming links. Links from the DITA Map to the topics are not taken into account. You can interact with the diagram by dragging nodes and by using the mouse wheel to zoom in or out.

Related information

Graph Visualization of the Oxygen XML Blog DITA Map

Publishing DITA Content Using the MKDocs Web Site Generator

MKDocs is a static site generator that is geared towards building project documentation. Documentation source files are written in Markdown and configured with a single YAML configuration file.

Here is how to use DITA XML content to build a static site with MKDocs:

1. Open your DITA map in the Oxygen XML Editor **DITA Maps Manager** view. For my example, I will use the `OXYGEN_INSTALL_DIR/samples/flowers/flowers.ditamap` sample.
2. Use the **Configure Transformation Scenarios** toolbar button, create a new transformation scenario of type **DITA-OT transformation** and choose **GitHub-flavored Markdown** as the transformation type.
3. In the transformation scenario's **Output** tab, you can configure the folder where the Markdown content should be generated.
4. Use the new transformation scenario to transform your DITA content into Markdown.
5. Install the *mkdocs* package on your system: https://www.mkdocs.org/getting-started/.
6. Create a new project using the `mkdocs` command line:

```
Developer$ mkdocs new flowers

INFO     -  Creating project directory: flowers

INFO     -  Writing config file: flowers/mkdocs.yml

INFO     -  Writing initial docs: flowers/docs/index.md

Developer$ cd flowers
```

7. In the created project's `flowers/docs` folder, copy the entire set of Markdown files produced by publishing the DITA content.
8. Use the *mkdocs* serve command to start a local web server:

```
Developer:flowers $ mkdocs serve

INFO     -  Building documentation...

INFO     -  Cleaning site directory

INFO     -  Documentation built in 0.24 seconds

INFO     -  [14:03:38] Serving on http://127.0.0.1:8000/
```

9. Open the `flowers/mkdocs.yml` file and configure it further, change the name of the project to something more appropriate (e.g. **Flowers**). Change the theme to some other theme supported by `mkdocs` (e.g. **readthedocs**).

```
site_name: Flowers

theme: readthedocs
```

Result: Back in the browser, the new site name and color theme should be applied, the search should work, and by default, the table of contents presents the files in the order that they appear on disk:

The `flowers/docs/index.md` file contains the table of contents derived from the original DITA map structure. Its contents can be used to create a navigation map which can be included in the `flowers/mkdocs.yml` configuration file.

10. Open the `index.md` file generated from the DITA map in Oxygen XML Editor and in the **HTML** preview right click and choose to export as HTML.
11. Create an XSLT stylesheet that when applied over the HTML content, it generates the entire structure of the table of contents:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

    xmlns:xs="http://www.w3.org/2001/XMLSchema"

    exclude-result-prefixes="xs"

    version="2.0">

    <xsl:output method="text"/>

    <xsl:template match="/">

        <xsl:apply-templates select="//*:a"/>

    </xsl:template>

    <xsl:template match="*:a">

        <xsl:variable name="noDescendents" select="count(parent::*:li/descendant::*:li)"/>

        <xsl:for-each select="ancestor-or-self::*:li"><xsl:text>

  </xsl:text></xsl:for-each>- <xsl:value-of select="text()"/>:<xsl:choose><xsl:when test="$noDescendents =

 0"><xsl:value-of select="@href"/></xsl:when><xsl:otherwise><xsl:text>

  </xsl:text><xsl:for-each select="ancestor-or-self::*:li"><xsl:text>

  </xsl:text></xsl:for-each>- <xsl:value-of select="text()"/>:<xsl:value-of select="@href"/></xsl:otherwise></x

sl:choose>
```

```
        <xsl:text>
</xsl:text>
    </xsl:template>
</xsl:stylesheet>
```

12. Apply the stylesheet over the HTML content to produce the content that gets included in the `flowers/mkdocs.yml` configuration file:

```
site_name: Flowers
nav:
    - Home: index.md
    - Introduction: topics/introduction.md
    - Care and Preparation: topics/care.md
    - Pruning: tasks/pruning.md
    - Garden Preparation: tasks/gardenPreparation.md
    - Flowers by Season: topics/index.md
    - Spring Flowers: concepts/springFlowers.md
    - Iris: topics/flowers/iris.md
    - Snowdrop: topics/flowers/snowdrop.md
    - Summer Flowers: concepts/summerFlowers.md
    - Gardenia: topics/flowers/gardenia.md
    - Lilac: topics/flowers/lilac.md
    - Autumn Flowers: concepts/autumnFlowers.md
    - Chrysanthemum: topics/flowers/chrysanthemum.md
    - Salvia: topics/flowers/salvia.md
    - Winter Flowers: concepts/winterFlowers.md
    - Gerbera: topics/flowers/gerbera.md
    - Genus: concepts/glossaryGenus.md
    - Pollination: concepts/glossaryPollination.md
    - Sepal: concepts/glossarySepal.md
    - Rhizome: concepts/glossaryRhizome.md
    - Bulb: concepts/glossaryBulb.md
    - Cultivar: concepts/glossaryCultivar.md
    - Perennial: concepts/glossaryPerennial.md
    - Panicle: concepts/glossaryPanicle.md
    - Copyright: topics/copyright.md
theme: readthedocs
```

Result: Now the output should have the table of contents generated from the original DITA map:

> **Tip:**
> There is a *mkdocs-literate-nav* mkdocs plugin which can be installed after **step 5** above and you can add the plugin information to the `mkdocs.yml` (instead of **step 9** above). This plugin utilises the `index.md` file built by the transform instead of the `nav` section in the `mkdocs.yml` to order the side toc correctly.

Creating PowerPoint Slides from DITA Content

Pandoc is a command-line interface that can convert lots of input formats (e.g. Markdown) to many types of output formats (e.g. Powerpoint).

Here is how to publish DITA XML content to Powerpoint using Pandoc:

1. Open your DITA map in the Oxygen XML Editor **DITA Maps Manager** view. For my example, I will use the `OXYGEN_INSTALL_DIR/samples/flowers/flowers.ditamap` sample.
2. Use the **Configure Transformation Scenarios** toolbar button, create a new transformation scenario of the type **DITA-OT transformation**, and choose **GitHub-flavored Markdown** as the transformation type.
3. In the transformation scenario's **Output** tab, specify the folder where the Markdown content should be generated.
4. In the same transformation scenario's **Parameters** tab, set the **root-chunk-override** parameter to the value of `to-content` to produce a single large Markdown file that contains all the DITA map's content.

5. Run the transformation scenario to transform your DITA content into Markdown.
6. Download the *Pandoc* command-line utility specific for your platform from: https://pandoc.org/installing.html.
7. Convert the Markdown file produced from DITA content to PPTX using the `pandoc` command line:

```
pandoc -f markdown -t pptx -o /path/to/out/flowers.pptx /path/tp/flowers/out/flowers.md
```

Result: You can open the resulting `flowers.pptx` in Powerpointl. For each DITA topic, you should have a separate slide:

> **Tip:**
> You can use the same approach of producing Markdown from DITA and then using Pandoc to produce other outputs such as Wiki, EPUB, Word, RTF, or PDF.

> **Tip:**
> To produce web slides from DITA content, you can also use the free DITA to RevealJS plugin created by Stefan Jung: https://github.com/doctales/org.doctales.reveal.

Adding Labels to Topics in the WebHelp Responsive Output

DITA topics can contain `<keyword>` elements that are usually placed in a `<prolog>` element, but they can also be used inside the topic body contents.

```
<topic id="topicID">
    <title>DITA 1.3 Features</title>
    <prolog>
        <metadata>
            <keywords>
```

```
            <keyword>DITA 1.3</keyword>
        </keywords>
    </metadata>
</prolog>...
```

The Oxygen WebHelp Responsive output uses these keywords as a way of giving more priority to certain sequences of words when searching for content. By default, keywords do not appear in the published output.

This article is an attempt to add a new concept of a **label**:

- A **label** is a **keyword** element with a special `@outputclass="label"` attribute. The label is always visible in the published output. It can be placed either in a prolog section or anywhere inside the topic where a keyword is allowed.
- When a label is selected, the WebHelp output displays all other topics that have the same **label** value set to them.

Inserting Labels

We can decide to use the DITA `<keyword>` element with a certain `@outputclass` attribute value set to `label` to insert such keywords that should be displayed in the published output.

```
<topic id="topicID">
    <title>DITA 1.3 Features</title>
    <prolog>
        <metadata>
            <keywords>
                <keyword outputclass="label">DITA 1.3</keyword>
            </keywords>
        </metadata>
    </prolog>...
```

To make the insertion of such labels easier, you can, for example, create a custom Author action that inserts the labels and add the custom action to the content completion window: Adding a Custom Author Action to the Content Completion Window *(on page 419)*.

To have special styling for the inserted labels when editing, you can customize the CSS stylesheets used for DITA editing: Customizing the DITA Visual Editing Experience *(on page 245)*.

Factoring Labels When Publishing

The WebHelp publishing by default creates visual labels in the published output.

Displaying prolog details (author, creation date, read time)

DITA topics can contain a `<prolog>` element with various details such as the **author name**, **creation date**, and other details that are not displayed in the published output by default.

```
<topic id="...">
    <title>...</title>
    <prolog>
        <author>...</author>
        <critdates>
            <created date="2022-01-13"/>
        </critdates>
```

At the time of publishing, we can show the author name, creation date, and the number of minutes it takes to read the article before the actual topic content by customizing the WebHelp Responsive output.

DITA WebHelp output can be customized using a publishing template mechanism. Inside a publishing template folder, there is an `opt` file that can contain links to various XSLT stylesheets that are useful for customizations. For example, we'll add a link to a stylesheet for processing the DITA `<prolog>`:

```
<publishing-template>
    <name>.....</name>
    ......
        <xslt>
            ....
            <extension file="xslt/prolog.xsl" id="com.oxygenxml.webhelp.xsl.dita2webhelp"/>
            .....
        </xslt>
    </webhelp>
</publishing-template>
```

The `prolog.xsl` stylesheet will display the author name, creation date, and an approximate number of minutes it takes to read the article:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" exclude-result-prefixes="xs" version="2.0">
    <xsl:template match="*[contains(@class, ' topic/prolog ')]">
```

```
        <!-- Display the author name -->
        <xsl:variable name="avatar-author" select="replace(*[contains(@class, ' topic/author ')],' ','_')"/>
        <div class="author">
           <xsl:value-of select="*[contains(@class, ' topic/author ')]"/>
        </div>
        <!-- Display the creation date -->
        <xsl:if test="exists(.//*[contains(@class, ' topic/created ')]/@date)">
            <div class="date">
                <xsl:variable name="cd" select=".//*[contains(@class, ' topic/created ')]/@date"/>
                <xsl:value-of select="format-date(xs:date($cd),
                    '[D] [MNn,3-3] [Y0001]')"/>
            </div>
        </xsl:if>
        <!-- Display the number of minutes it takes to read the article -->
        <div>
            <xsl:variable name="fileContent" select="/"/>
            <xsl:variable name="text" select="normalize-space($fileContent)"/>
            <xsl:variable name="textWithoutSpaces" select="translate($fileContent, ' ', '')" />
            <xsl:variable name="fileCountWords" select="string-length($text) - string-length($textWithoutSpaces) +1"/>
            <xsl:variable name="readMin" select="format-number($fileCountWords div 50, '0')"/>
            Read time: <xsl:value-of select="$readMin"/> minute(s)
        </div>
        <xsl:next-match/>
    </xsl:template>
</xsl:stylesheet>
```

The WebHelp publishing template used for publishing this blog already has a customization to display information from the prolog: https://github.com/oxygenxml/blog/tree/master/publishing/webhelpBlogTemplate.

Each published topic contains information about the author name, created date, and an estimate number of minutes it takes to read the article. The author name display can be enriched with an avatar photo:

Frequently Asked Questions

Radu Coravu
29 Mar 2022

Read time: 2 minute(s)

How do I register to receive notifications for new blog posts?

Dynamically publishing a list with the most recent topics

DITA topics can contain an optional `<created>` element that specifies the date when they were created by the technical writer.

```
<topic id="...">
    <title>...</title>
    <prolog>
        <author>...</author>
        <critdates>
            <created date="2022-01-13"/>
        </critdates>
```

At the time of publishing, we can dynamically generate a topic that contains a list of the most recent topics, while taking the creation date into account for each topic by customizing the WebHelp Responsive output.

DITA WebHelp output can be customized using a publishing template mechanism. Inside a publishing template folder, there is an `opt` file that can contain links to various XSLT stylesheets that are useful for customizations. For example, we'll add a link to a stylesheet for processing such special keyword label elements:

```
<publishing-template>
    <name>.....</name>
    ......
        <xslt>
            ....
            <extension file="xslt/updateWhatsNew.xsl" id="com.oxygenxml.webhelp.xsl.createTocXML"/>
            .....
        </xslt>
```

```
    </webhelp>
</publishing-template>
```

The `updateWhatsNew.xsl` stylesheet will sort the first five most recent topics and create a new DITA topic that contains it:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" exclude-result-prefixes="xs" version="2.0"
    xmlns:toc="http://www.oxygenxml.com/ns/webhelp/toc">

    <!-- Artifically impose content to the what's new topic. -->
    <xsl:template match="/">
        <xsl:next-match/>
        <xsl:result-document href="{resolve-uri('topics/what_s_new.dita', base-uri())}">

 <xsl:processing-instruction name="workdir-uri"><xsl:value-of select="replace(resolve-uri('topics/what_s_new.dita',
 base-uri()), 'what_s_new.dita', '')"/></xsl:processing-instruction>
            <xsl:processing-instruction name="path2project-uri">../</xsl:processing-instruction>
            <xsl:processing-instruction name="path2rootmap-uri">../</xsl:processing-instruction>
            <topic xmlns:ditaarch="http://dita.oasis-open.org/architecture/2005/"
                id="topic_v5l_lts_fsb" ditaarch:DITAArchVersion="1.3"
                domains="(topic abbrev-d) a(props deliveryTarget) (topic equation-d) (topic hazard-d) (topic hi-d)
 (topic indexing-d) (topic markup-d) (topic mathml-d) (topic pr-d) (topic relmgmt-d) (topic sw-d) (topic svg-d) (topic
 ui-d) (topic ut-d) (topic markup-d xml-d) "
                class="- topic/topic ">
                <title class="- topic/title ">What's New!</title>
                <body class="- topic/body ">
                    <p class="- topic/p ">Most recent blog posts:</p>
                    <section class="- topic/section ">
                        <!-- Find the most recent topics by looking at the creation date and sorting them descending
 -->
                        <xsl:for-each
                            select="//topicref[@href][not(@format) or @format =
 'dita'][doc-available(resolve-uri(@href, base-uri()))][document(resolve-uri(@href,
 base-uri()))/*/prolog/critdates/created/@date]">
                            <xsl:sort select="document(resolve-uri(@href,
 base-uri()))/*/prolog/critdates/created/@date" order="descending"/>
                            <!-- Present only the first 5 topics -->
                            <xsl:if test="position() &lt; 6">
```

```
                                <xsl:variable name="doc" select="document(resolve-uri(@href, base-uri()))"/>
                                <p class="- topic/p ">
                                    <xref href="{replace(@href, 'topics/', '')}" class="- topic/xref
 "><xsl:value-of select="$doc/*/title"/></xref>
                                </p>
                            </xsl:if>
                        </xsl:for-each>
                    </section>
                </body>
            </topic>
        </xsl:result-document>
    </xsl:template>
</xsl:stylesheet>
```

In the DITA Map, we need to add a reference to a stub DITA topic that will be overwritten by the publishing process:

```
    <topicref href="topics/what_s_new.dita" print="no"/>
```

The published output will produce a what's new topic as the one here: what_s_new.html.

The WebHelp publishing template used for publishing this blog already has a customization to display a recent list of topics: https://github.com/oxygenxml/blog/tree/master/publishing/webhelpBlogTemplate.

Dynamically Generating an RSS Feed

When publishing a website from DITA XML content, we can dynamically generate an RSS XML feed by customizing the WebHelp Responsive output.

DITA WebHelp output can be customized using a publishing template mechanism. Inside a publishing template folder, there is an `opt` file that can contain links to various XSLT stylesheets that are useful for customizations. For example, we'll add a link to a stylesheet for processing such special keyword label elements:

```
<publishing-template>
    <name>.....</name>
    ......
        <xslt>
            ....
            <extension file="xslt/createRSS.xsl" id="com.oxygenxml.webhelp.xsl.createNavLinks"/>
            .....
```

```
        </xslt>
    </webhelp>
</publishing-template>
```

The `createRSS.xsl` stylesheet generates an `rss.xml` file that contains references to all topics referenced in the DITA Map:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    exclude-result-prefixes="xs"
    xmlns:toc="http://www.oxygenxml.com/ns/webhelp/toc"
    version="2.0">

    <xsl:param name="JSON_OUTPUT_DIR_URI"/>
    <xsl:template match="/toc:toc">
        <xsl:next-match/>
        <xsl:apply-templates mode="rss" select="."/>
    </xsl:template>

    <xsl:template match="/toc:toc" mode="rss">
        <xsl:result-document href="{concat($JSON_OUTPUT_DIR_URI, '/../../../../rss.xml')}" indent="yes">
            <rss version="2.0" xmlns:atom="http://www.w3.org/2005/Atom">
                <channel>
                    <title>TITLE OF RSS FEED HERE</title>
                    <link>/rss.xml</link>
                    <description>DESCRIPTION OF FEED HERE</description>
                    <language>en-us</language>

  <atom:link href="https://my.website/rss.xml" rel="self" type="application/rss
+xml" xmlns:atom="http://www.w3.org/2005/Atom"/>
                    <lastBuildDate><xsl:value-of select="format-dateTime(current-dateTime(),'[F], [D] [M01] [Y] [h] [P]
 [z]')"/></lastBuildDate>
                    <docs>https://oxygenxmlblog.netlify.com/</docs>
                    <generator>FEED GENERATOR NAME</generator>
                    <managingEditor>EDITOR EMAIL INFORMATION</managingEditor>
                    <webMaster>WEBMASTER EMAIL INFORMATION</webMaster>
                    <copyright>Copyright Info Here. All rights reserved.</copyright>
                    <category>News</category>
                    <ttl>30</ttl>
```

```
                    <image>
                        <url>http://my.website/img/rss-image.gif</url>
                        <title>BLOG TITLE HERE
                        <link>/rss.xml</link>
                    </image>
                    <xsl:for-each select="//toc:topic">
                        <xsl:if test="not(@href = 'javascript:void(0)')">
                            <item>
                                <title><xsl:value-of select="toc:title"/></title>
                                <link><xsl:value-of select="concat('/', @href)"/></link>
                                <guid isPermaLink="false"><xsl:value-of select="@href"/></guid>
                                <xsl:variable name="ref" select="replace(resolve-uri(@href, base-uri()), '\.html',
'.dita')"/>
                                <xsl:variable name="date" select="document($ref)/*/prolog/critdates/created/@date"/>
                                <xsl:choose>
                                    <xsl:when test="$date">
                                        <pubDate><xsl:value-of select="
                                            format-date(xs:date($date),
                                            '[F], [D01] [MNn,*-3] [Y] 00:00:00 GMT')"/></pubDate>
                                        <!-- Format like: Thu, 20 Dec 2022 02:46:11 UTC -->
                                    </xsl:when>
                                    <xsl:otherwise>
                                        <!-- Set some fixed date so that the topic does not appear as new -->
                                        <pubDate> Wed, 1 Jan 2020 02:46:11 GMT</pubDate>
                                    </xsl:otherwise>
                                </xsl:choose>
                            </item>
                        </xsl:if>
                    </xsl:for-each>
                </channel>
            </rss>
        </xsl:result-document>
    </xsl:template>
</xsl:stylesheet>
```

The published output will produce an RSS feed similar to the one here: https://blog.oxygenxml.com/rss.xml. For each topic referenced in the feed, a publication date is computed based on its creation date specified in the prolog:

```
<topic id="rss_feed_generation">
    <title>...</title>
    <prolog>
        <author>...</author>
        <critdates>
            <created date="YYYY-MM-DD"/>
        </critdates>
```

The WebHelp publishing template used for publishing this blog already has a customization to create the RSS feed: https://github.com/oxygenxml/blog/tree/master/publishing/webhelpBlogTemplate.

Using the RSS Feed With Follow.it to Receive New Posts by Email

Websites such as **Follow.it** allow you to register a link to your RSS feed and produce a small HTML form that can be embedded in your blog. Users who register to receive notifications when new posts are available in the RSS feed will receive automatic emails.

Once the RSS feed is available on the website, the following steps can be followed to integrate with **Follow.it**:

1. Go to the Follow.it website and add a reference to your RSS feed.
2. On the website, customize the form that will be embedded in your blog HTML contents.
3. In the the `opt` file (inside the Oxygen WebHelp publishing template folder), add references to an XML file that contains the structure for it to appear after the content of each article:

   ```
   <fragment file="html-fragments/subscribe.xml" placeholder="webhelp.fragment.after.toc_or_tiles"/>
   <fragment file="html-fragments/subscribe.xml" placeholder="webhelp.fragment.after.feedback"/>
   ```

4. Create the `subscribe.xml` file and paste the form HTML content created by **Follow.it** inside it. Make the HTML content well-formed.

Generating Google Structured Data from your DITA tasks

HTML pages published on the web can contain metadata specified using the Google Structured Data specification. Once such metadata exists in an HTML page, the Google search engine can present, for example, steps to complete a certain task directly in the search page without the need to open the target HTML page. Below is a set of steps for automatically generating **Google Structured Data** metadata for DITA tasks when publishing DITA content to Oxygen WebHelp Responsive output, which can be customized using a publishing template mechanism.

1. In your DITA project, create a task type of topic with a specific `@outputclass` attribute value to signal that you want the Google structured data to be automatically generated for it.

```
<task id="task_id" outputclass="google-structured-data-steps">
  <title>My task</title>
</task>
<steps>
      <step>
        <cmd>Step 1 content.</cmd>
      </step>
      <step>
        <cmd>Step 2 content.</cmd>
      </step>
</steps>
```

2. Inside a WebHelp publishing template folder, there is an `opt` file that can contain links to various XSLT stylesheets that are useful for customizations. For example, we'll add a link to a stylesheet for processing such special tasks and producing a special script that contains details for each step.

```
<publishing-template>
    <name>.....</name>
    ......
        <xslt>
            ....
            <extension file="xslt/addGoogleStructuredData.xsl" id="com.oxygenxml.webhelp.xsl.dita2webhelp"/>
            .....
        </xslt>
    </webhelp>
</publishing-template>
```

3. Create the `addGoogleStructuredData.xsl` XSLT stylesheet that processes the task contents and adds in the HTML head a script that contains the steps in Google Structured Data format.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  exclude-result-prefixes="xs"
  version="2.0">
  <xsl:template match="*[contains(@class, ' topic/prolog ')]">
    <xsl:if test="/*[@outputclass='google-structured-data-steps']">
      <xsl:apply-templates select="/*" mode="google-structured-data"/>
    </xsl:if>
```

```
    <xsl:next-match/>
  </xsl:template>
  <xsl:template match="*" mode="google-structured-data">
    <script type="application/ld+json">
      {
      "@context": "https://schema.org",
      "@type": "HowTo",
      "name": "<xsl:value-of select="title"/>",
      "step": [
      <xsl:for-each select="taskbody/steps/step">
        {
        "@type": "HowToSection",
        "name": "Step",
        "position": "<xsl:value-of select="position()"/>",
        "itemListElement": [
        {
        "@type": "HowToStep",
        "position": "1",
        "itemListElement": [
        {
        "@type": "HowToDirection",
        "position": "1",
        "text": "<xsl:value-of select="normalize-space(cmd)"/>"
        }]}]}
        <xsl:if test="position() &lt; last()">,</xsl:if>
      </xsl:for-each>
      ]}
    </script>
  </xsl:template>
</xsl:stylesheet>
```

4. Publish the DITA XML Content to a web site.
5. Test your HTML page using the Google Rich Results Tester: https://search.google.com/test/rich-results.

6. Once Google indexes your page, google search for it.

Related information

Generating Google Structured Data from your DITA frequently asked questions *(on page 239)*

Generating Google Structured Data from your DITA frequently asked questions

HTML pages published on the web can contain metadata specified using the Google Structured Data specification. Once such metadata exists in an HTML page, the Google search engine can present, for example, lists of frequently asked questions directly in the search page without the need to open the target HTML page. Below is a set of steps for automatically generating **Google Structured Data** metadata for DITA frequently asked questions when publishing DITA content to Oxygen WebHelp Responsive output, which can be customized using a publishing template mechanism.

1. In your DITA project, create a topic with a specific `@outputclass` attribute value to signal that you want the Google structured data to be automatically generated for it.

```
<topic id="frequently_asked_questions" outputclass="google-structured-data-faq">
    <title>Frequently Asked Questions</title>
    <body>
        <section>
            <title>How do I register to receive notifications for new blog posts?</title>
            <p>Each blog HTML page has at the end a form in which you can fill your email address if
                you want to be notified when new posts are made.</p>
        </section>
..............
```

2. Inside a WebHelp publishing template folder, there is an `opt` file that can contain links to various XSLT stylesheets that are useful for customizations. For example, we'll add a link to a stylesheet for processing such special DITA topics and producing a special script that contains details for each question/answer pair.

```
<publishing-template>
    <name>.....</name>
    ......
        <xslt>
            ....
            <extension file="xslt/addGoogleStructuredData.xsl" id="com.oxygenxml.webhelp.xsl.dita2webhelp"/>
            .....
        </xslt>
    </webhelp>
</publishing-template>
```

3. Create the `addGoogleStructuredData.xsl` XSLT stylesheet that processes the task contents and adds a script in the HTML head that contains the frequently asked questions in Google Structured Data format.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  exclude-result-prefixes="xs"
  version="2.0">
  <xsl:template match="*[contains(@class, ' topic/prolog ')]">
    <xsl:choose>
      <xsl:when test="/*[@outputclass='google-structured-data-faq']">
        <xsl:apply-templates select="/*" mode="google-structured-data-faq"/>
```

```
      </xsl:when>
    </xsl:choose>
    <xsl:next-match/>
  </xsl:template>


  <xsl:template match="*" mode="google-structured-data-faq">
    <script type="application/ld+json">
     {
      "@context": "https://schema.org",
      "@type": "FAQPage",
      "mainEntity": [
        <xsl:for-each select="body/section">
          {
            "@type": "Question",
            "name": "<xsl:value-of select="normalize-space(title)"/>",
            "acceptedAnswer": {
            "@type": "Answer",
            "text": "<xsl:value-of select="normalize-space(string-join(*[not(self::title)], ''))"/>"
            }
          }
          <xsl:if test="position() &lt; last()">,</xsl:if>
        </xsl:for-each>
      ]
     }
    </script>
  </xsl:template>


</xsl:stylesheet>
```

4. Publish the DITA XML Content to a web site using the **WebHelp Responsive** transformation.
5. Test your HTML page using the Google Rich Results Tester: https://search.google.com/test/rich-results.

6. Once Google indexes your page, google search for it.

Related information

Generating Google Structured Data from your DITA tasks *(on page 236)*

Editing Customizations

The DITA XML editing features (visual editing, validation, custom actions) can be customized by extending and changing the base DITA framework configuration.

Startup DITA Project

The Startup DITA Project contents are useful for starting a new DITA project with Oxygen XML Editor version 23.0 or newer, using collaboration version control systems such as **Git** or **Subversion**.

When the `project.xpr` is open in the **Project** view, it imposes a custom set of options, as well as an imposed custom DITA framework extension.

Project Structure

The project contains the following folder structure:

- `custom-settings/frameworks` - Contains custom DITA and DITA Map framework extensions.
- `custom-settings/learned-words` - Contains learned words for the English dictionary.
- `filters` - This initially empty folder can be updated to contain DITAVAL Filter files used to profile the DITA content when publishing.
- `images` - Contains images used in the DITA project.
- `oxygen-term-checker` - Contains a terminology file with an example rule.
- `publishing` - Contains a DITA-OT project file and a publishing template.
- `reusables` - Contains various reusable components. For example, in the "reusable_components.dita" topic, you can add various DITA elements (such as notes, paragraphs, lists, and list items) that are reused in various places in the project. See more about reusing DITA content.
- `topics` - Contains all the DITA topics in the project.

Settings Saved at Project Level

Most of the preference pages in **Options > Preferences** can be saved at project level. When the project is open in the **Project** view, the following options are imposed:

- The **Editor** / **Spell Check** preferences page is saved at project level with automatic spell checking enabled.
- The **Editor** / **Spell Check** / **Dictionaries preferences** page is saved at project level and all learned words are saved in the `custom-settings/learned-words/en.tdi` file.
- The **Editor** / **Edit Modes** / **Author** preferences page is saved at project level.
- The **DITA** / **Maps** preferences page is saved at project level.
- The **Document Type Association** preferences page disables certain framework configurations that are not useful (e.g. **DocBook 5**).
- The **Document Type Association** / **Locations** preferences page references two custom framework configurations that are described further.

Framework Customizations

The editing environment for **DITA Maps** and **Topics** can be customized by extending their current frameworks.

The framework extension for editing **DITA Topics** is defined in the `custom-settings/frameworks/dita-extension/dita-extension.exf` framework script file. You can add your own custom actions to the toolbars, menus, or contextual menus, and add, change, or remove transformation scenarios or validation scenarios.

- You can customize the existing DITA Topic file templates by making changes to the `custom-settings/frameworks/dita-extension/templates/` folder.
- The content completion configuration extension file `custom-settings/frameworks/dita-extension/resources/cc_config_ext.xml` allows you to customize various aspects of the content completion feature.
- You can add custom Schematron validation rules *(on page 291)* in the `custom-settings/frameworks/dita-extension/resources/customRules.sch` Schematron file.
- You can also customize the visual editing *(on page 245)* by modifying the CSS `custom-settings/frameworks/dita-extension/css/custom.css`.

The framework extension for editing **DITA Maps** is defined in the `custom-settings/frameworks/dita-map-extension/dita-map-extension.exf` framework script file and can me customized in a similar way. You can customize the existing DITA Map file templates by making changes to the `custom-settings/frameworks/dita-map-extension/templates/` folder.

Renaming/Moving Topics and Other Resources

The project has a logical folder named `Main Files` with a reference to the main DITA map. This enables the *Main Files* support in the project, allowing you to rename or move files while the references to those files are automatically updated.

Terminology Checking

For checking terminology, you can use the Oxygen Terminology Checker add-on. You can add terminology rules to the `oxygen-term-checker` folder. For example, the Oxygen user manual uses a set of Vale rules to check for common terminology problems.

Publishing

You can publish the content of the `project.ditamap` in two ways:

- You can open the DITA-OT project file named `publish-project.xml` from the `publishing` subfolder and two predefined publishing scenarios become available. You can read more about DITA-OT project files in the user manual.
- You can also open the `project.ditamap` file in the DITA Maps Manager view, invoke the "Configure Transformation Scenario(s)" dialog box, and in the Project section, there are two

predefined transformation scenarios (one for WebHelp, the other for PDF) that use the publishing template archived in the zip file. This past webinar shows more details about creating and customizing a publishing template.

Starting with Oxygen XML Editor version 24, the **Startup DITA Project** will be available as a new project template in the **Project > New Project** dialog box. If you want to try it with Oxygen XML Editor version 23 you should download the project template (`.zxpr` file) from the project releases page and set it as a new document template.

Customizing the DITA Visual Editing Experience

The Author visual editing experience in Oxygen is **CSS** driven. Let's say I have a team of writers using Oxygen and they want to visually edit DITA **dl** elements in a table-like layout.

All the validation, editing and publishing support Oxygen has for a specific XML vocabulary is defined in a framework configuration *(on page 425)*.

Instead of copying an entire framework configuration folder (such as **DITA** or **Docbook**), modify and distribute it, you can choose to extend that framework and distribute the extension. This way, you will benefit from new functionality added to the base framework by newer Oxygen versions and still use your customizations.

The steps below describe how an extension of the **DITA** framework that removes certain elements from the content completion list can be constructed and shared:

1. Somewhere on your disk, in a place where you have full write access, create a folder structure like: **custom_frameworks/dita-extension**.
2. In that folder, create a new CSS stylesheet (for example **custom.css**) that will contain your custom CSS styles:

```
dl{
    display:table !important;
}
dlentry{
    display:table-row !important;
}
dt, dd {
    display:table-cell !important;
    border: 1px solid black;
```

```
    padding: 2px;
}
```

3. In the **Document Type Association / Locations** preferences page, add the path to your **custom_frameworks** folder in the **Additional frameworks directories** list. Then click **Apply** in the **Preferences** dialog box.
4. In the **Document Type Association** preferences page, select the **DITA** document type configuration and use the **Extend** button to create an extension for it.
5. Give a custom name to the extension (for example **DITA - Custom**), then change its **Storage** to **external**, and save it to a path like: **path/to/.../custom_frameworks/dita-extension/dita-extension.framework**.
6. Make whatever changes are necessary to the framework configuration, and in the **Author** tab click the **CSS** tab and add a reference to your custom CSS. Do not set a title for the CSS and also do not check the **Alternate** checkbox as you want your CSS to be applied by default.
7. Click **OK** to close the dialog box and then either **OK** or **Apply** to save the changes to the preferences.

After you perform these steps, you will have a fully functioning framework in the **dita-extension** folder that can be shared with others: Document Type Extension Sharing *(on page 407)*.

To check that your framework extension works, you can create a new **DITA** topic and insert a **dl** element inside it. It should now be presented in a table-like layout.

In order to know which **CSS** styles to override for a specific **DITA** element, you can right click inside that element in the **Author** visual editing mode and use the **Inspect Styles** action to see all CSS styles defined for it by default. You can also define alternate **CSS** styles that are applied manually by the writer by using the **Author** toolbar **Styles** drop-down menu.

Customizing the DITA Framework Using a Framework Extension Script

How to customize an existing framework (e.g. DITA) using a framework extension script.

All the validation, editing, and publishing support Oxygen has for a specific XML vocabulary is defined in a framework configuration *(on page 425)*. Oxygen comes bundled with such frameworks for popular XML vocabularies, such as DITA. A common use case is to make changes to these built-in frameworks, to tailor them according to specific requirements.

A framework extension script is an XML file that defines the changes to perform on a base framework. Suppose that you want to make the following changes to the DITA framework:

1. Customize the new document templates.
2. Change the Author mode rendering with a new CSS file.
3. Remove the **Bold**, **Italic**, **Underline** actions from the Author mode.
4. Add the **Insert Note** action to the toolbar.

Creating the Framework Extension Script

The easiest way to create such a script is to use the **New** document wizard and choose the **Extend Framework Script** or **Create Framework Script** template. Define it as an extension of the DITA framework through the `@base` attribute. Also, set a high priority, through the `<priority>` element, to make sure the framework will be picked in favor of the DITA one.

```
<script xmlns="http://www.oxygenxml.com/ns/framework/extend"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.oxygenxml.com/ns/framework/extend
 http://www.oxygenxml.com/ns/framework/extend/frameworkExtensionScript.xsd"
  base="DITA">
  <name>Custom DITA</name>
  <description>A custom DITA framework.</description>
  <priority>High</priority>
</script>
```

You need to save the script inside a framework directory (the same place as the *.framework file). For example, if you save it inside `.../custom-frameworks/custom-dita/dita-script.exf`, then you need to add `.../custom-frameworks` to the Additional frameworks directories list.

> **Note:**
> Oxygen 23.1 or later will automatically detect the script and load the framework. If you want to use the framework in an older Oxygen XML Editor version that does not have support for these scripts, you can compile the script to obtain the *.framework file by using the **Compile Framework Extension** script action from the contextual menu or by running the `scripts/compileFrameworkScript.bat` external tool (available in the All Platforms distribution only).

Customizing the New Document Templates

The document templates appear when the user invokes the **New...** action. To add a new template, you need to do the following:

- In the directory where the script is saved, create a new file (e.g. `templates/My custom topic.dita`). The content of the file represents the template's content.
- In the script, specify the new template locations by adding this fragment inside the `<script>` element.

```
<documentTemplates inherit="none">
  <addEntry path="${framework}/templates"/>
</documentTemplates>
```

Note:
In the example snippet above, the `@inherit` attribute is set to not inherit any of the document templates defined in the base framework.

Changing the Author Mode Rendering With a New CSS File

The author mode is driven by CSS rules. To add new rules, you need to:

- Create a new CSS file in the directory where the script is saved. Give it a name (e.g. `css/custom.css`) and, for example, a rule to make titles red:

```
title {
  color:red;
}
```

- In the script, specify the path to the new CSS by adding this fragment inside the `<script>` element.

```
<author>
  <css>
    <addCss path="${framework}/css/custom.css"/>
  </css>
</author>
```

Removing the Bold, Italic, Underline Actions From the Author Mode

An author action is just a configuration that describes which operation to use depending on the context. Each action has a unique ID. Suppose that you do not want **Bold**, **Italic**, and **Underline** actions from the built-in DITA framework because their markup is not semantic. After you inspect the actions preferences and we find out their IDs, you can filter them from all toolbars and menus by adding this fragment inside the `<script>` element.

```
<author>
    <authorActions>
      <removeAction id="bold"/>
      <removeAction id="italic"/>
      <removeAction id="underline"/>
    </authorActions>
  </author>
```

Adding the Insert Note Action to the Toolbar

The **Insert Node** action is already defined in the DITA framework, but it is not present on the toolbar. To add it to the toolbar, you need to:

- Go to **Options->Preferences**, edit the DITA framework, and search in the Actions tab for the **Insert Node** action. Make note of its ID.
- Edit the framework extension script and put the action in the toolbar by adding this fragment inside the `<script>` element.

```
<author>
  <toolbars>
    <toolbar>
      <addAction id="insert.note" anchor="paragraph"/>
    </toolbar>
  </toolbars>
</author>
```

Note:
In this example, an action is used that is already present on the toolbar as an anchor. The new action is added to the toolbar after the action that inserts a paragraph.

Related information

https://www.oxygenxml.com/doc/ug-editor/topics/framework-customization-script-usecases.html
https://www.oxygenxml.com/events/2021/webinar_creating_frameworks_using_an_extension_script.html

Controlled Attribute Values for your DITA Project

Frequently when editing DITA content you will feel the need to enforce a controlled set of values when editing certain attributes. For example you may want to impose that the values for the **@outputclass** attribute on the element **codeblock** are either **language-xml** or **language-css**. This is useful in order to remind writers that any other value will not be interpreted by the build process in a significant manner.

Oxygen has a couple of easy ways in which controlled values can be imposed for certain attributes:

1. You can edit the XML configuration file **OXYGEN_INSTALL_DIR/frameworks/dita/resources/cc_value_config.xml** and provide additional entries. In the case of our small example for providing controlled values for the **@attribute** the configuration file should contain an additional entry:

   ```
   <match elementName="codeblock" attributeName="outputclass">
    <items action="addIfEmpty">
     <item value="language-xml" annotation="XML Syntax Highlight"/>
     <item value="language-css" annotation="CSS Syntax Highlight"/>
    </items>
   </match>
   ```

   Besides providing a hard-coded list of values the content completion configuration file is flexible enough to allow calling an XSLT stylesheet which could retrieve those values from other sources (for example via HTTP from an Exist database).
2. Provide those controlled values via a Subject Scheme Map (my favorite). Coming back to our example, you can create a small Subject Scheme map with the file name `controlledValues.ditamap` and the content:

   ```
   <!DOCTYPE subjectScheme PUBLIC "-//OASIS//DTD DITA Subject Scheme Map//EN""map.dtd">
   <subjectScheme>
       <subjectHead>
           <subjectHeadMeta>
               <navtitle>Provide controlled attributes</navtitle>
           </subjectHeadMeta>
       </subjectHead>
       <hasInstance>
           <subjectdef keys="languageTypeKey">
               <subjectdef keys="language-xml">
                   <topicmeta>
                       <navtitle>XML Syntax Highlight</navtitle>
                   </topicmeta>
   ```

```
            </subjectdef>
            <subjectdef keys="language-css">
                <topicmeta>
                    <navtitle>CSS Syntax Highlight</navtitle>
                </topicmeta>
            </subjectdef>
        </subjectdef>
    </hasInstance>
    <enumerationdef>
        <elementdef name="codeblock"/>
        <attributedef name="outputclass"/>
        <subjectdef keyref="languageTypeKey"/>
    </enumerationdef>
</subjectScheme>
```

then you can refer to it from your main DITA Map like:

```
<topicref href="controlledValues.ditamap" format="ditamap" type="subjectScheme"/>
```

3. If the attributes on which you want to impose certain values are DITA profiling attributes, you can go to the Oxygen **Preferences**->**Editor / Edit modes / Author / Profiling/Conditional Text** page and define the set of allowed values for them.

The only problem with the first approach is the fact that validation will not impose those values and writers will not receive validation error messages if they set another value for the specific attribute. So you will probably need to add a Schematron check in order to signal errors when a certain attribute's value does not match the list of controlled attribute values. For both the second and third approaches, validation will warn the writers if certain attribute values do not match values in the controller values list.

Controlled Attribute Values (Part 2 - Advanced)

As already presented in Controlled Attribute Values for your DITA Project *(on page 250)*, Oxygen allows you to add or replace possible values for attributes or elements based on a simple configuration file. A more complex scenario is one in which in order to decide which values to provide, you need more context information. Let's take this DITA fragment:

```
<metadata>
  <othermeta name="name" content="value"/>
</metadata>
```

What we want is to offer proposals for `@@content` but the possible values for `@@content` depend on the value of `@@name`. We will see how we can solve this dependency.

Note: Starting with Oxygen 17.1 there is a simpler way to achieve the use case presented in this post. The contextElementXPathExpression parameter will be bound to an XPath expression that identifies the element in the context of which the content completion was invoked.

The configuration file

The configuration file (**cc_value_config.xml**) allows calling an XSLT stylesheet and that's just what we will do:

```
<match elementName="othermeta" attributeName="content">
 <xslt href="meta.xsl" useCache="false"/>
</match>
```

As you can see, we can't express the dependency between `@@content` and `@@name` inside the configuration file . I also want to mention that because the values for `@@content` are dynamic, we want the XSLT script to execute every time the values are requested (we shouldn't cache the results). We enforce this by setting `@@useCache` to **false**.

The XSLT script

The XSLT script has access to the XML document (through the **documentSystemID** parameter) but it lacks any context information, we can't really tell for which `<othermeta>` element was the script invoked. To counter this limitation, we will use Java extension functions and we will call Oxygen's Java-based API from the XSLT. Here how it looks:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xd="http://www.oxygenxml.com/ns/doc/xsl"
    xmlns:tei="http://www.oxygenxml.com/ns/doc/xsl"
    xmlns:prov="java:ro.sync.exml.workspace.api.PluginWorkspaceProvider"
    xmlns:work="java:ro.sync.exml.workspace.api.PluginWorkspace"
    xmlns:editorAccess="java:ro.sync.exml.workspace.api.editor.WSEditor"
    xmlns:saxon="http://saxon.sf.net/"
    xmlns:textpage="java:ro.sync.exml.workspace.api.editor.page.text.xml.WSXMLTextEditorPage"
    xmlns:authorPage="java:ro.sync.exml.workspace.api.editor.page.author.WSAuthorEditorPage"
    xmlns:ctrl="java:ro.sync.ecss.extensions.api.AuthorDocumentController"
    exclude-result-prefixes="xs xd"
```

```
   version="2.0">
<xsl:param name="documentSystemID" as="xs:string"/>
<xsl:template name="start">
  <xsl:variable name="workspace" select="prov:getPluginWorkspace()"/>
  <xsl:variable name="editorAccess" select="work:getEditorAccess($workspace, xs:anyURI($documentSystemID), 0)"/>
  <xsl:variable name="pageID" as="xs:string" select="editorAccess:getCurrentPageID($editorAccess)"/>
  <xsl:variable name="name" as="xs:string">
  <xsl:choose>
      <xsl:when test="$pageID='Text'">
         <xsl:variable name="textpage" select="editorAccess:getCurrentPage($editorAccess)"/>
         <!-- In the text page, the context is the @content attribute -->
         <xsl:value-of select="textpage:evaluateXPath($textpage, 'xs:string(./parent::node()/@name)')"/>
     </xsl:when>
     <xsl:when test="$pageID='Author'">
        <xsl:variable name="authorPage" select="editorAccess:getCurrentPage($editorAccess)"/>
        <xsl:variable name="caretOffset" select="authorPage:getCaretOffset($authorPage)"/>
        <xsl:variable name="ctrl" select="authorPage:getDocumentController($authorPage)"/>
        <xsl:variable name="contextNode" select="ctrl:getNodeAtOffset($ctrl, $caretOffset)"/>
        <!-- In the author page, the context is the "othermeta" element -->
        <xsl:value-of select="ctrl:evaluateXPath($ctrl, 'xs:string(@name)', $contextNode, false(), false(), false(),
false())[1]"/>
    </xsl:when>
  </xsl:choose>
 </xsl:variable>
 <items>
  <xsl:choose>
     <xsl:when test="$name = 'temperatureScale'">
       <item value="Celsius" annotation="(symbol C)"/>
       <item value="Fahrenheit" annotation="(symbol F)"/>
    </xsl:when>
    <xsl:when test="$name = 'measurement'">
       <item value="Metric" annotation="Metric system"/>
       <item value="Imperial" annotation="Also known as British Imperial"/>
    </xsl:when>
  </xsl:choose>
 </items>
```

```
 </xsl:template>
</xsl:stylesheet>
```

Converting Subject Scheme Map Values to a DITAVAL

Suppose you already have a **Subject Scheme Map** in your project and you use it to control attribute values: Controlled Attribute Values for your DITA Project *(on page 250)*.

In the Oxygen **Colors and Styles** preferences page, you can also assign various colors and styles to each profiling attribute (name, value) combination. One option for this is to manually re-add attributes and values in that list. Another option would be to create an XSLT stylesheet to gather all profiling attribute names and values from the **Subject Scheme Map** and create a **DITAVAL** file. The stylesheet would look like this:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    exclude-result-prefixes="xs"
    version="2.0">
    <xsl:output indent="yes"/>
    <xsl:template match="/">
        <val>
            <xsl:for-each select="subjectScheme/enumerationdef">
                <!-- For each attribute name -->
                <xsl:if test="subjectdef/@keyref and attributedef/@name">
                    <xsl:variable name="attrName" select="attributedef/@name"/>
                    <xsl:variable name="keyref" select="subjectdef/@keyref"/>
                    <!-- For each key value -->
                    <xsl:for-each select="//*[@keys=$keyref]/*//@keys">
                        <xsl:variable name="attributeValue" select="."/>
                        <prop action="flag" att="{$attrName}" val="{$attributeValue}"/>
                    </xsl:for-each>
                </xsl:if>
            </xsl:for-each>
        </val>
    </xsl:template>
</xsl:stylesheet>
```

After you obtain the **DITAVAL** file, you can import it directly in the **Colors and Styles** preferences page. If the **DITAVAL** file has flagging information, that information will be used directly to style each attribute value.

A possibility to enhance this workaround is to specify profiling styles for each attribute value directly in the Subject Scheme map using the `<data>` element like:

```
<subjectdef keys="linux">
    <data name="color" value="yellow"/>
</subjectdef>
```

in this case the **XSLT** stylesheet would create the **DITAVAL** file by picking colors directly from the **Subject Scheme Map**:

```
….….….....
<prop action="flag" att="{$attrName}" val="{$attributeValue}">
    <xsl:choose>
        <!-- Here you can also set flagging colors depending on the profiling attribute value -->
        <xsl:when test="data[@name='color']">
            <xsl:attribute name="color" select="data/@value"/>
        </xsl:when>
    </xsl:choose>
</prop>
….….….….
```

In this way, your **Subject Scheme Map** will keep both the controlled attribute values and various colors and styles, which can later be used to create a **DITAVAL** file and either publish with those styles or import the **DITAVAL** file in Oxygen to highlight certain elements with various colors: https://www.oxygenxml.com/demo/Colors_and_Styles_for_Profiled_Content.html.

Converting .xpr Profiling Conditions to Standard DITA Files

Profiling Conditions in the Oxygen .xpr Project File

When we first started using Oxygen and DITA, we defined our profiling conditions in the Oxygen `.xpr` project file:

- Attribute values were defined in **Profiling/Conditional Text > Attributes and Condition Sets > Profiling Attributes**:

- Colors were defined in **Profiling/Conditional Text > Attributes and Condition Sets > Colors and Styles**:

We organized our DITA content into three different Git repositories. The DITA content in each Git repository used different profiling conditions. But we wanted to share a single `.xpr` file across all three repositories, so we defined all profiling conditions in that file.

Storing profiling conditions in a shared `.xpr` file caused some issues:

- Writers were shown profiling conditions from other repositories that were not relevant to them.

  This was okay in the beginning because we did not have many profiling conditions. But over time, the number of profiling conditions increased as we increased content reuse and converted more content to DITA.

- It was time-consuming to add or update profiling conditions.

Writers are not permitted to modify our `.xpr` file, so I needed to understand and implement each new condition. Then I had to roll out the updated `.xpr` file to all three Git repositories (and sometimes to multiple branches in a repository).

- Color definitions could become desynchronized.

  Colors were defined in two places—in the `.xpr` file for the Oxygen UI and in DITAVAL files for transformations—and I needed to remember to update the colors in both places.

Oxygen supports DITA subject scheme files for defining profiling condition attribute *values* in a DITA-standard way, so we started using them. This allowed writers to add or update attribute values, which was a step in the right direction. But I still needed to update and distribute the corresponding *colors* in the `.xpr` file.

When we decided to add a fourth Git repository that required many new profiling conditions, we decided we needed a better approach. And fortunately the Oxygen v26.0 release saved the day with a solution.

Profiling Conditions in Standard DITA Files

The Oxygen v26.0 release provides a new feature that allows profiling condition colors to be loaded dynamically from DITAVAL files:

In previous versions, Oxygen could statically import colors from DITAVAL files into the `.xpr` file, but that did not resolve the fundamental challenges with a shared `.xpr` files described above.

With this new feature, Oxygen loads the color definitions directly from the DITAVAL files in the **Main Files** list when you open a DITA map or topic.

Now we can define profiling conditions entirely with standard DITA files, with no product-specific information needed in the `.xpr` file:

- Attribute values are defined by DITA subject scheme files
- Attribute colors are defined by DITAVAL files, for both editing and publishing

Finally, profiling condition updates are completely self-service for writers! They can freely add or update profiling condition values or colors as they want.

What Are DITA Subject Scheme Files?

A DITA *subject scheme* file is a specialized type of DITA map file that can constrain attributes and elements to particular defined values. It is included as a submap in a DITA content map. In this case, we can use it to constrain profiling attributes.

Subject scheme maps accomplish this task in two steps:

- A `<subjectdef>` element defines a set of values for a subject.
    - For example, a subject could be a product family, and its values are variants within that product family.
- An `<enumerationdef>` element binds the values for one or more subjects (products) to the attribute specified by `<attributedef>`.

In our environment, we keep a centralized set of value definition files in a "warehouse" folder:

my-map.ditamap

```
<map>
 ...
 <mapref href="my-map/profiling.ditamap"/>
 ...
</map>
```

Reference subject scheme submap

my-map/profiling.ditamap

```
<subjectScheme>
 <!-- this is a map-specific <enumerationdef> file that constrains profiling
      conditions to the values defined in one or more <subjectdef> files -->
 <schemeref href="../warehouse/subjectdefs-audience.ditamap"/>
 <schemeref href="../warehouse/subjectdefs-product.ditamap"/>
 <enumerationdef>
  <attributedef name="audience"/>
  <subjectdef keyref="subjectdefs-audience"/>
 </enumerationdef>
 <enumerationdef>
  <attributedef name="product"/>
  <subjectdef keyref="subjectdefs-product"/>
 </enumerationdef>
</subjectScheme>
```

Bind values to attributes using **<enumerationdef>** and **<attributedef>**

warehouse/subjectdefs-audience.ditamap

```
<subjectScheme>
 <subjectdef keys="subjectdefs-audience">
  <topicmeta>
   <navtitle>@audience values</navtitle>
  </topicmeta>
  <subjectdef keys="novice">
   <topicmeta>
    <navtitle>Novice users</navtitle>
   </topicmeta>
  </subjectdef>
  <subjectdef keys="expert">
   <topicmeta>
    <navtitle>Expert users</navtitle>
   </topicmeta>
  </subjectdef>
 </subjectdef>
</subjectScheme>
```

Define values using **<subjectdef>**

warehouse/subjectdefs-product.ditamap

```
<subjectScheme>
 <subjectdef keys="subjectdefs-product">
  <topicmeta>
   <navtitle>@product values</navtitle>
  </topicmeta>
  <subjectdef keys="A">
   <topicmeta>
    <navtitle>Product A</navtitle>
   </topicmeta>
  </subjectdef>
  <subjectdef keys="B">
   <topicmeta>
    <navtitle>Product B</navtitle>
   </topicmeta>
  </subjectdef>
 </subjectdef>
</subjectScheme>
```

Define values using **<subjectdef>**

DITA subject scheme structures might seem intimidating at first. However, they can be easier to understand when populated with attributes and values that are familiar to you.

Converting .xpr Profiling Conditions With Refactoring

To make it easier to move from `.xpr`-defined profiling conditions to standard DITA profiling conditions, I created a refactoring operation that processes an input `.xpr` file as follows:

- Converts profiling condition value definitions to DITA subject scheme files

  A `profiling.ditamap` file is created that can be referenced by DITA content maps.

- Converts profiling condition color definitions to a single `all_colors.ditaval` file

  This file can be added to the **Main Files** list defined in the Oxygen `.xpr` project file. Although the color definitions might change over time, the file location does not, and so no changes are needed to the `.xpr` file.

- Removes the converted value and color definitions from the input `.xpr` file

You can use the following test case to experiment with this refactoring operation:

converting_xpr_profiling.zip

The `oxygen.xpr` file in the test case contains profiling conditions that you can convert to DITAVAL and DITA subject scheme files:

To try the refactoring operation,

1. (Optional) Create a backup copy of the `.xpr` file.

   The refactoring operation removes converted profiling conditions from the input `.xpr` file, so you might want to save a copy of the original file.

2. Open the `.xpr` file in an Oxygen editing window.

   Locate the `.xpr` file in the Project view, then right-click it and choose **Open with > Internal editor**. When prompted for the format, accept the default of **XML Document**.

3. Run the refactoring operation on the file.

   In the editing window, right-click and choose **Refactoring > Profiling conditions > Convert .xpr profiling conditions**.

4. In the configuration dialog, specify how to perform the conversion:

   - For **Attributes to convert**, specify the space-separated list of profiling attributes to convert.
   - For **When writing <subjectScheme> files**, specify how to distribute the value definitions (`<subjectdef>`) and attribute bindings (`<enumerationdef>`) across one or more subject scheme files.

     For example, if you are using profiling attribute groups (the testcase uses groups for the `@product` attribute), having each group defined in its own `<subjectdef>` file can help different writers track their own changes over time in Git.

5. Click the **Preview** button to preview the operation, then complete the refactoring operation.

6. Add the `oxygen.xpr-profiling/` folder to the **Main Files** list.

   Right-click the folder in the **Project** view, then choose **Add to Main Files**.

7. Configure Oxygen to dynamically load colors from the newly created DITAVAL file.

   In **Profiling/Conditional Text > Attributes and Condition Sets > Colors and Styles**, enable the **Automatically apply colors and styles...** checkbox.

The newly created `profiling.ditamap` file defines all profiling condition values. If a particular DITA map needs only some conditions, you can create a map-specific copy of the `profiling.ditamap` file that binds only the values of interest. In our environment, we use the convention of *`mapdir/`*`profiling.ditamap`, keeping the same "`profiling.ditamap`" file name but placing it inside the relevant map content folder.

To create a map-specific copy of `profiling.ditamap` in the test case,

1. Perform the conversion as described above.
2. Copy the `profiling.ditamap` file from the `oxygen.xpr-profiling/` folder to the `my-map/` folder.
3. Open the newly copied `my-map/profiling.ditamap` file and fix any `<schemeref>` references to `*-subjectdefs.ditamap` files.

   This step is only necessary if you configured the refactoring operation to create "two" or "many" `<subjectScheme>` files.
4. Add a map reference to the map-specific profiling file.

   Open the `my-map.ditamap` file in the **DITA Maps Manager**, then add a map reference to the `my-map/profiling.ditamap` file.

Related information

Controlled Attribute Values for your DITA Project *(on page 250)*

Other

Oxygen XML As a Technical Documentation Solution FAQ

This topic covers some of the most frequent questions we get from technical documentation writer teams willing to investigate the **Oxygen XML** set of tools as a possible solution for writing and publishing content.

What does Oxygen XML provide as a tool to a technical documentation team?

Oxygen is a **DITA XML** desktop editing tool that can be used by technical documentation writers to **write**, **reuse**, and **publish** DITA XML content to a variety of output formats, such as **WebHelp Responsive** and **PDF**.

Some of our strong capabilities are listed here *(on page 278)*.

To which companies would you recommend a DITA XML based solution for writing technical documentation?

I would recommend the use of the DITA XML standard for writing technical documentation with Oxygen XML as an editing tool to companies that have one of the following conditions fulfilled:

- They have multiple similar products and want to take advantage of DITA XML high reuse capabilities and condition profiling/filters to produce multiple similar, but not identical, manuals from the same project.
- They want to add structure-based validation rules that must be followed by all members of a team.
- They want to create their own custom vocabulary of semantic elements.
- Besides web-based outputs, they also want to produce PDFs and want to have a relatively high degree of control over the PDF customization.

More details here: Types Of Companies Who Would Benefit From Using the DITA XML Standard *(on page 273)*.

Is Oxygen a Content Management System?

Oxygen is not a **content management system**, we do not store the content that you are editing on our side.

Our clients use **Oxygen XML** for collaboration on a technical documentation project in one of these ways:

- We have clients who use Oxygen's **Git client add-on** to collaborate on the same project using a Git repository *(on page 375)*.
- **Oxygen** also has built-in support for working with **SharePoint** repositories if you want to store the edited content on **SharePoint**.
- There are commercial **CMSs** (content management systems) that have integrations with **Oxygen**. Some of them are listed here. These commercial content management systems such as **Astoria**, **Bluestream**, **Componize** usually have plugins that allow **Oxygen** to connect to the **CMS** remote storage, open documents, edit and save them back. But the price of the commercial content management systems is separate from the price of an **Oxygen** license.

What Oxygen products should I buy for our technical documentation team?

All of our prices are transparently available here.

The usual purchase suggestion for a technical writer's team is something like this:

- One permanent user-based license of Oxygen XML Editor Professional: https://www.oxygenxml.com/buy_new_licenses_professional_smp.html.

  This one license would be for the person who is in charge of customizing the published output as well as writing content.

- A permanent license of Oxygen XML Author professional for each full time tech writer: https://www.oxygenxml.com/buy_new_licenses_author_professional_smp.html.

  The **Oxygen XML Author** allows writers to visually edit **DITA XML** content and to publish it to **WebHelp**, **PDF**, and other formats.

- Any other occasional contributor or reviewer (subject matter experts or engineers, for example) can use our Content Fusion platform to give feedback on the content directly from a web browser.

  A video of how **Content Fusion** Works: https://www.oxygenxml.com/content_fusion/take_a_tour.html.

  And the pricing, depending on the number of people who would simultaneously review the content: https://www.oxygenxml.com/content_fusion/buy_content_fusion.html.

What type of customer support and training will I get from the Oxygen team?

We help customers with advice and any problems they encounter when using our application.

For example, if you want to create your own publishing customizations, these are usually done using CSS. You would have one colleague on your side creating the customization and asking us questions along the way. We do not implement full customizations for you, but we guide you along the way.

As for training, we have lots of videos and online resources for learning to use Oxygen to produce technical documentation: Resources for learning DITA with Oxygen *(on page 66)*.

We can offer you a one-time web meeting to show you how Oxygen works in general, but if you want a more rigorous training, we have partners like Ryffine or Mekon who offer such courses: Courses for learning DITA with Oxygen XML Editor *(on page 68)*.

How can I convert my current content to DITA XML?

You can find advice about how to convert various document formats to DITA XML here. For any other document format, the advice is to possibly try to export it to HTML content and then use the free Oxygen **Batch Documents Converter** add-on.

Do you have case studies or references from companies who are successfully using Oxygen?

Some of our customers who have agreed to be officially listed are mentioned here: https://www.oxygenxml.com/customers.html.

Also some testimonials: https://www.oxygenxml.com/case_studies.html.

DITA For Small Technical Documentation Teams

There are many cases in which a handful of technical writers in a company need to collaborate on a DITA project and they may not have the resources to invest in an open source content management system. So this blog post is about setting up a viable, cost efficient DITA documentation editing and publishing solution which can scale up (both in number of writers and content) using Oxygen XML Author and Github. As a showcase for the proposed solution we will have the Oxygen XML Blog *(on page 8)*.

Overview of Tools Proposed for the Solution

The following list of tools and application will be used in our solution:

Content Management and Workflow

Github for storing the DITA content.

Github for creating and managing issues.

Editing

Oxygen XML Author for editing the DITA content and for offline publishing.

[Optional] Oxygen Web Author for online editing/reviewing of DITA content.

Review

[Optional] Oxygen Content Fusion for creating review tasks for subject matter experts.

Publishing

Oxygen XML Author for publishing from inside the application.

[Optional] Oxygen Publishing Engine for publishing to PDF and WebHelp DITA content on an integration server.

[Optional] Oxygen WebHelp Feedback engine for embedding feedback forms in the HTML pages.

[Optional] Gradle to script the publishing in preparation for an integration server.

[Optional] Netlify to build and host a web site for the DITA content.

[Optional] Oxygen Validate and Check for Completeness used with a scripting license to check for validation problems on an integration server.

The same set of tools has been used for editing and publishing the Oxygen XML Blog: Welcome! *(on page 8)*.

Learning DITA

There are various online resources for learning DITA, there are also learning DITA courses held by consulting companies. This older blog post has some useful links: Resources for learning DITA with Oxygen *(on page 66)*.

Project Storage - Using GitHub

You need a system to:

- Store your DITA content.
- Collaborate with your colleagues on writing content in the same project.
- Allow for reviewers to propose changes without having the right to commit changes.
- Have a history of changes on each DITA resource, being able to revert to previous versions of topics.
- See who changed what content in each DITA topic and map.
- Be able to tag releases so you can build manuals for older releases with some changes applied to them.
- Maintain a list of issues which are closed as your work progresses.

Some small teams instead of using a version control system resort to shared network drives. Shared network drives do not address all the features above and they are dangerous in the fact that you can overwrite other people's work and there is no way to revert changes.

A free Github account can cover all these use cases listed above. GitHub exposes through a web interface access to a Git repository. This blog post is not about learning Git, it's about having minimum Git skills to get you started. There is a good book about learning Git Here: https://www.manning.com/books/learn-git-in-a-month-of-lunches.

We'll outline below some steps to get you started with a GitHub project:

1. Create a GitHub account: https://github.com/join.
2. Create a new public or private repository in the organization for your project.
3. Create an organization: https://github.com/settings/organizations. The organization will be useful to group work-related projects. For adding private repositories in an organization you will need to switch to a paid plan.
4. Add your team members to the organization. Using the team member GitHub user names you can send them invitations to be contributors to the project.

Creating and Modifying Content in the Repository

You can create and edit DITA topics in a number of ways:

- Edit topics offline and use GitHub's user interface to upload them, or modify existing topics using GitHub's web interface.
- Use the Oxygen Web Author to connect to the Git Repository and create a new topic or modify an existing one.
- Clone the repository to the local file system using Oxygen XML Author with the Git Add-on installed.

Using Oxygen XML Author to Create/Modify Content in the Repository

- Install the Git Add-on in Oxygen XML Author.
- Open the **Git Staging** view and use the **+** toolbar button to paste the reference to the reference to your project, for example `https://github.com/oxygenxml/blog`.
- Choose a location on the local drive where the project will be cloned.
- The **Git Staging** view does not show all existing resources in the project, it will just show the modified resources, so go to the **Project** menu and create a new Oxygen project in the folder where the project was cloned.
- In the **Git Staging** view the newly created `project.xpr` should now appear. Select it, use the **Stage Selected** button to stage it, then write a commit message and use the **Commit** message to commit it. Now the file is committed in your local repository. You can use the toolbar **Push** button to push it to the GitHub account.
- Create a DITA Map and a couple of topics using Oxygen's **Project** view.
- Using the same **Git Staging** view you can see the newly added files, stage, commit and push them to the remote GitHub repository. You can check on the remote **GitHub** repository that the files were updated.
- If changes are made by someone else using the same technique or if changes to maps and topics are made directly from the **GitHub** web interface or from the Oxygen WebAuthor, the **Git Staging** view has a **Pull** toolbar button allowing you to obtain those changes locally.
- On resources which have been modified a number of times you can right click in the staging area and choose **Show in history** to see a history of changes. You can also choose **Show blame** to see who modified each of the lines in the document.

Here's a simplified overview of how you could work with Oxygen and Git:

- Every time you start Oxygen or every time before you commit use the **Pull** toolbar button from the **Git Staging** view to get changes locally.
- Stage and commit your changes, then push them to the remote repository.

- If you attempt to pull changes including a file which is already modified locally by you, the pull operation will not succeed. You will need to copy that file separately, pull the content from the server and then compare your changes to the ones coming from the server and merge your changes.
- If you have already committed to your local repository a resource modified also on the remote repository, you have a conflict, the **Git Staging** view allows you to edit the conflict, merge the changes locally and then commit to the local repository and push changes to the server.
- You do not have to pull and push changes from and to the remote repository very often, you can continue working with your local repository for days and push all changes at the end but not pulling changes from the server often might result in various conflicts if the same resources are modified both on the server and in your local repository.

Organizing Your Project Structure using the Master Files Support

You can now start to organize your content. The Oxygen user's manual contains a Getting Started guide for DITA editing.

You can right click the **Project** root and enable main files, then in the **Main Files** folder add a reference to the main DITA map (or maps). All refactoring operations (move resources to subfolders, rename topics or resources) will properly update references in the entire DITA project hierarchy. A minimal DITA project would probably contain:

- One or more main DITA Maps.
- One or more DITA submaps, defining keys for reusable content.
- One or more folders containing reusable DITA elements.
- A folder containing one or more DITAVAL filter files.
- A folder containing images or binary resources.

Overall management of the DITA project

- The **Search References** contextual menu actions from the DITA Maps Manager and from the main editor allow you to check where a topic, map or image is being used.
- The **Remove from disk** action in the DITA Maps Manager view will warn if the removed topic is used in multiple places.
- Validation errors are issued for references to missing topics or images.
- **Move/Rename** actions located both in the Oxygen Project and DITA Maps Manager view allow you to move topics/maps or images and have references to them automatically updated.
- Search for references to IDs or rename IDs in the entire project.

- The **Find unreferenced resources** contextual menu action in the DITA Maps Manager allows you to find orphan topics or other resources (images, etc).
- You can define profiling condition sets and use them to see in the editing area what content is filtered, then use the same profiling condition sets for publishing.
- The **Open/Find Resource** side view allows you to locate files by content.
- The change tracking support in Oxygen allows you to make changes with change tracking or to add or reply to existing comments.

You can find out more DITA editing tips in the DITA Editing Tips And Tricks blog *(on page 279)*.

Sharing Common Settings using the Project

The Oxygen project configuration can save various sets of global options at project level. As an example:

- Go to the Oxygen **Preferences->Editor / Spell Check** page and change the radio button at the bottom to **Project Options**. Then check the **Automatic spell check** checkbox.
- The **Git Staging** view should have a modified version of the XPR showing as modified, stage, commit and push it.
- Now everybody using the same `project.xpr` project file loaded in Oxygen will have spell check enabled.

Other useful settings you could switch to **Project Options**:

- The **Project Level Settings** preferences dialog allows you to control if various settings are saved at project or global level. For example you can save the **Validate and check for completeness** settings at project level and share them with others.
- The **Editor / Spell Check / Dictionaries** preferences page can contain your own set of dictionary terms.
- The **Document Templates** preferences page allows you to use your own custom new file templates.
- In the **Editor / Save** preferences you can save at project level various save options.
- Switching the **Menu Shortcut Keys** preferences page to project level allows you to share with others custom menu shortcut keys.

You can use the **${pd}** editor variable to refer to resources located in the current project folder. You can read more about editor variables here *(on page 469)*.

Manual Publishing

You can open DITA Maps in the application **DITA Maps Manager** view and publish them to various outputs using the **Configure Transformation Scenarios** toolbar button. A transformation scenario can also specify a DITAVAL filter, parameters and can be saved at project level. The published content can be later uploaded to a web server.

For **WebHelp** output you can create a publishing template containing custom CSSs, headers, footers and logos. The publishing template can be saved in the project and referenced from the transformation scenario.

For the **PDF** output the same publishing template can be used. To build your custom CSS you can use the Oxygen Styles Basket online CSS-builder.

If you are publishing multiple DITA Maps or you have multiple filters or multiple deliverable formats you may want to create a DITA Open Toolkit project file: https://www.oxygenxml.com/doc/ug-editor/topics/dita_open_toolkit_project.html. A sample DITA OT project file can be found in the Oxygen XML Blog project on GitHub.

Working with Branches

You may need to mark releases of your product and also to be able to contribute small fixes to those releases. So here's how you work with branches:

- The default branch in the project is called **master** and this is the branch we've been working until now.
- Using **GitHub**'s web interface you can create your own branch.
- Check out again in the **Git Staging** view the project, this time with the separate branch in a separate folder.
- Open the `project.xpr` from that branch's checkout, make changes to the content there, stage, commit and push as usual.

Establishing a Workflow

You can use the **GitHub** issues list for your project to create new issues. You can attach files to the issue, assign the issue, discuss with other participants, register to watch particular issues. Once a commit is made with a particular **#issue_number**, the issue will get an automatic link pointing to the commit.

Inside an organization or inside a repository you can create project boards. A project board can define states for your issues. When an issue is opened it can be assigned to a certain project board. It can also be assigned various labels. Issues can also be assigned to various people in your team.

Oxygen's change tracking support allows you to insert or delete content with change tracking or to make comments on content. The Web Author online editing tool allows you to also make changes with change tracking.

The Oxygen Content Fusion add-on allows you to create review tasks and to allow subject matter experts to give you feedback on DITA content directly from a web browser.

Adding Custom Validation Checks to the Project

There are a couple of ways in which you can add custom validation checks to your project:

- You can create a custom Schematron schema containing custom validation rules: Sharing Schematron Validation Rules *(on page 409)*.
- The **Validate and Check For Completion** action from the **DITA Maps Manager** view contains various settings which can be saved at project level. In the **Preferences->Project Level Settings** page there is a **Save DITA Map validate and check for completeness settings at project level** checkbox.
- You can also add terminology checking in the application: Checking Terminology with Oxygen XML Editor *(on page 458)*.
- In GitHub projects you can configure custom actions and run validate and check for completeness with a special scripting license on pull requests. An example for such a validation action is in the Oxygen XML Blog: https://github.com/oxygenxml/blog/blob/master/.github/workflows/. The validation runs on any opened pull request: https://github.com/oxygenxml/blog/pull/25.

Adding Custom Actions

You can use a framework customization saved at project level to add custom actions which can be used in the **Author** visual editing mode and can be mounted in the toolbar, menus and the content completion window: Implementing a Custom Author Action to Split a Table *(on page 417)*.

Dynamic Publishing using an Integration Server

Using a bit of scripting you can publish the modified content on the fly with either using free online services like Netlify or Travis or using an internal Jenkins server. The Oxygen XML Blog *(on page 8)* is an example for this. It has a Gradle script which is run as part of a Netlify account whenever changes are made to the topics.

Translating your content

The Oxygen Translation Package Builder add-on allows you to find the changed resources in a DITA project and then to pack those changed resources and send them for translation. It also helps with

integrating the received translation in the project. This blog post *(on page 287)* has more details and useful links about translating DITA projects.

Related information

Webinar: Using DITA for Small Technical Documentation Teams

Types Of Companies Who Would Benefit From Using the DITA XML Standard

In general, I would divide the companies that benefit from using DITA XML for their technical documentation needs in three large sections:

Companies Creating Physical Products

These companies create real-world objects, such as cars, refrigerators, or computer chips. Each version of each product has an unique product ID and needs to be documented. Different products are sometimes similar in functionality but they also have some differences.

Companies Creating Version-ed Software Products

These companies provide software components that are version-ed and usually get installed on the client's side. As each client may have an older version of the product, user manuals for older versions need to be made available.

Companies Creating Version-less Software Products

These companies usually provide cloud-based services, the end-user always receives access to the latest stable release, and there is no user manual made available for past releases.

The benefits for companies would be as follows:

Table 2. Benefits

Benefits	Company Types		
	Physical Products	**Version-ed Software Products**	**Version-less Software Products**
Take advantage of **DITA XML** high reuse capabilities and condition profiling/filters to produce	Produce manuals for similar components.	Produce user manuals for similar types of software products.	Produce user manuals for similar types of software products.

Table 2. Benefits (continued)

Benefits	Company Types		
	Physical Products	**Version-ed Software Products**	**Version-less Software Products**
multiple similar, but not identical, manuals from the same project.			
Add structure-based validation rules that must be followed by all members of a team.	Use common validation and terminology rules for the entire team.	Use common validation and terminology rules for the entire team.	Use common validation and terminology rules for the entire team.
Create custom vocabulary of semantic elements.	Useful to add specific semantic meaning to content specific for the company.	Useful to add specific semantic meaning to content specific for the company.	Useful to add specific semantic meaning to content specific for the company.
Besides web-based outputs, produce PDFs with a relatively high degree of control over the PDF customization.	Useful to produce PDFs for print or online delivery.	Useful to produce PDFs for print or online delivery.	Useful to produce PDFs for print or online delivery.

Supporting the DITA OT Project as a Commercial Company

Overview

This small article is about the relationship between our company, **Syncro Soft** (which produces the Oxygen XML tools), and the open source DITA Open Toolkit project.

As a commercial company using the **DITA Open Toolkit** in multiple projects, we try to base this relationship on:

- **Give and take**
- **Respect**
- **Growth**

but as in any relationship, there is probably always room for improvement.

History and Context

We started integrating the **DITA Open Toolkit** into Oxygen about **14** years ago, some time around **2008** in Oxygen XML Editor version **10**. The DITA Open Toolkit remains a very important part of the DITA XML editing and publishing solution provided by us.

Currently, we are using the **DITA Open Toolkit** as a publishing component in our **Oxygen Publishing Engine** in multiple projects:

- **Oxygen Publishing Engine (based on the DITA OT)**
- **Oxygen XML Editor**
- **Oxygen XML Author**
- **Oxygen XML Developer**

and there will probably be more to come in the future.

Implementing Plugins for the DITA Open Toolkit

During these years, we've implemented a number of **commercial** plugins for the DITA Open Toolkit, plugins bundled in the Oxygen Publishing Engine:

- **DITA to WebHelp Responsive**
- **DITA to PDF (using CSS for styling)**
- **DITA to Zendesk**
- **CodeBlock Syntax Highlight**

We also created a considerable number of open-source DITA Open Toolkit plugins (about **23**) available on the Oxygen GitHub organization, with some of the most popular ones being:

- **DITA Metrics Report**
- **Export DITA Map**
- **DITA Media Support**
- **DITA PDF Track Changes**
- **Embed PlantUML and Mermaid diagrams**
- **Embed Latex Equations**

These open-source plugins are useful both for their original purpose but also as sample plugins for people wanting to implement something similar. We try to maintain the open-source plugins (as time allows) and answer any questions related to them.

Helping with Publishing-related Questions

There are various channels where people ask questions about DITA OT publishing:

- DITA Users List (about **1000** answers and replies so far)
- DITA OT Users Google Group
- DITA OT Discussions List
- DITA OT Slack Channel
- Stack Overflow (about **80** answers so far)

There are also Oxygen-specific channels:

- Oxygen XML Forum (**2328** DITA editing/publishing related topics so far)
- Oxygen Users List

As time allows, we try to help people with **DITA Open Toolkit**-related questions on these channels.

DITA Open Toolkit Publishing-related Articles

The Oxygen XML Blog has more than 20 DITA Open Toolkit publishing related articles.

Registering and Analyzing Issues, Opening Pull Requests

Our general procedure when encountering **DITA Open Toolkit** bugs is to:

- Open issues for them on the DITA Open Toolkit list.
- If possible find time to analyse the issues.
- Propose workarounds or fixes.

We are possibly the biggest contributor of bug and improvement requests to the **DITA Open Toolkit**. Most of these bugs and improvements are discovered by our end users and added by us on the DITA OT issues list. During these years we've opened more than 400 issues on the DITA Open Toolkit project.

In the last few years, we became a major contributor of pull requests that contain bug fixes (more than 70 contributed pull requests so far).

Contributing with Improvement Ideas to the DITA Open Toolkit Project

During these years, we contributed improvement ideas to the **DITA Open Toolkit** and were involved in the process where they were implemented. Here are some improvement ideas in which we were involved:

- The **DITA Open Toolkit Project** file.
- **Metadata** to specify parameter names, descriptions, and default values in the plugin descriptor files.
- Processing **RelaxNG-based** DITA Maps and topics.

Contributions to the DITA Open Toolkit Documentation

The DITA Open Toolkit documentation has relevant content for people who want to get started using and customizing the DITA Open Toolkit. We frequently help our users by giving them links to the DITA OT documentation.

When necessary, we contribute issues on the DITA Open Toolkit documentation issues list: https://github.com/dita-ot/docs. I contributed to the DITA OT Docs project about 20 issues so far.

The DITA Open Toolkit Day

We started hosting and sponsoring the **DITA Open Toolkit Day** about 8 years ago, back in **2014**, as a way to bring the community of plugin/customization developers together with the main contributors to the **DITA OT** project. The **DITA OT Day** is also a means to celebrate the work of the main **DITA OT** contributors. Recordings for all videos from past DITA OT Day events can be found here: https://www.dita-ot.org/dev/topics/dita-ot-day-videos.html.

DITA Map Metrics Evolution

The **DITA Map Metrics Report** transformation scenario available in Oxygen can be applied on a DITA Map and produce all kinds of interesting metrics for your DITA project:

- The number of processed maps and topics.
- Content reuse percentage.
- Number of elements, attributes, words, and characters used in the entire DITAmap structure.
- DITA conditional processing attributes used in the DITA maps.
- Processing instructions.
- External links.
- All `@outputclass` attribute values gathered from the DITA project.

There is also an open source DITA Open Toolkit plugin on the Oxygen XML GitHub organization which can be used to generate a metrics report either in HTML or XML format.

If you generate metrics for various stages in the history of your DITA project (for example for various releases) you can combine those metrics and generate graphs which can show various indicators like:

- Fluctuations in number of topics and maps.
- Fluctuations in content reuse.

Metrics reports and metrics evolution reports are useful to get an overview of a DITA-based documentation project and to discuss its future directions. They are also useful indicators to discuss with management.

Here are some steps to generate a metrics evolution report for a DITA project hosted on a Git server:

1. Install this DITA Open Toolkit plugin which can publish DITA tables as SVG graphs: https://github.com/oxygenxml/dita-table-svg.
2. Install the DITA Metrics report plugin in your DITA Open Toolkit installation (if not yet installed): https://github.com/oxygenxml/dita-ot-metrics.
3. Switch your Git project to various tagged versions, after each switch apply the **metrics-report-xml** transformation type on the DITA Map and generate an XML file representing the metrics for that particular version. As an example I created 4 metrics reports for 4 previous Oxygen XML User's Guide versions: https://github.com/oxygenxml/dita-ot-metrics/tree/master/evolution/samples.
4. Apply the provided XSLT stylesheet to generate a DITA topic containing tables which show how various indicators change between versions.
5. Apply the **DITA to HTML5** transformation on the topic to create an HTML document showing the evolution of different indicators as a graph.

Oxygen XML Editor DITA Editing solution strong points.

We are sometimes asked which are the differentiating features between Oxygen XML Editor and its competitors when it comes to DITA Editing. So below I tried to list some differentiating features (strong points) that Oxygen has:

- User based license which allows somebody to install Oxygen on multiple computers (work computer, home computer) as long as they are the only ones using it: https://www.oxygenxml.com/eula.html.
- Transparent pricing, prices are available on the web site: https://www.oxygenxml.com/buy.html.
- Great technical support. You can ask us questions via email or via the forum.
- Cross platform availability. Being able to run Oxygen on Windows, Mac or Linux.
- Complete DITA Editing support: *(on page 120)*
    - Lots of pre-defined actions to insert reusable content, images, links, videos.
    - DITA-specific helper views (DITA Maps Manager, DITA Reusable Components)
    - Support to customize the editing environment: add custom actions, remove current actions, customize content completion items, create custom validation or transformation scenarios.

    - Powerful publishing solution for producing responsive WebHelp and PDF using CSS to style the output.
    - Publishing to output formats like Windows Help (CHM), Ms Word, EPUB, Eclipse Help included.
    - Lots of commercial DITA CMSs have integrations with Oxygen.
- Support for change tracking and for adding comments in the editor along with a special **Review** view to review changes: https://www.oxygenxml.com/doc/ug-editor/topics/author-managing-changes.html
- Support to add third party plugins and enhance Oxygen's functionality. And lots of free add-ons already available to add support for Git integration, DITA translations and Batch conversions from various formats to DITA.
- Support to add custom validation rules based on your internal style guide. The rules can be implemented using the Schematron standard and you can also add quick fixes for them: Schematron Checks to help Technical Writing *(on page 291)*.
- Support to send DITA content to reviewers using the Content Fusion add-on.

DITA Oriented Tips And Tricks

Oxygen XML Editor has about 18 years of development under its wings. During these years, we added lots of functionality and many of our users usually do not know more than 20-30% of Oxygen's features. So this presentation is intended for technical writers using Oxygen to write DITA content and who may want to find out more about their tool.

Add-ons

There are lots of free add-ons provided by the Oxygen team that can be installed in an Oxygen standalone version. I will enumerate some of the most interesting one:

- Git add-on. This plugin adds a side view allowing you to do some common Git operations like pull, commit, push (mostly 90% of what a regular Git user would do).
- Content Fusion Connector add-on. Collaborate with your peers by sending them links that allow them to give you feedback on your work using a web browser without the need to pre-install anything on their side.
- Translation package builder. Small plugin that can be used to prepare a zip file containing only the topics that have changed from one version to another.
- Batch convertor add-on. Converts multiple HTML, Markdown, Excel files to DITA.

Project-related Functionality:

The Oxygen Project view is the place where you can organize and apply batch changes to all your DITA resources:

- Main files support. Enable main files support in the DITA project, add your top level DITA Maps as main files and any structure changes, renaming or moving files will automatically update all links in the DITA Project. Also works for renaming/moving image resources.
- Store options at project level. Almost all of Oxygen's preferences pages can be saved at project level. Once you do that, you can share the **project.xpr** file with somebody else and when they open it, they will also get the settings set inside the project.
- Compare files/folders.
- Refer to multiple folders.
- Batch operations: Select a folder in the project and use the "Transform", "Validate", "Format and Indent", "Find/Replace in Files", or "Spell Check in Files". Or use the "XML Refactoring" action to apply a refactoring stylesheet over all the selected files.
- Filter files. You can instruct the Project view to hide various file types.

Navigation:

Actions to find opened files, find files containing a certain content or navigate between files.

- In the main **Window** menu, you can find the **Next/Previous Editor** actions and their shortcuts. Together with the **Switch editor tab** action, these help you navigate between opened XML documents.
- The **Open/Find Resource** view (main menu **Window->Show View**) allows you to search for file names, search in file contents, or search for files containing comments or change tracking.
- Right-click the tab of an opened DITA document and use the **Copy location** action. Or use the **Show in Explorer/Finder** action to locate the file.
- Use the main menu **File->Reopen last closed editor** action to re-open the last closed file.
- Open a DITA topic in the **Text** editing mode, right click, choose **Go to definition** to open the DTD, XML Schema or RelaxNG Schema at the precise location where that DITA element is defined, something useful for debugging DITA specializations.
- Use the **Navigation** toolbar to **Go back/forward/last modification**.

Find/Replace Functionality

The Find/Replace functionality is the bread and butter of any editing application and Oxygen has lots of functionality in this regard:

- Find/Replace in Files.
    - Restrict to XPath - If you want to make replacements only in certain parts of the XML document, the XPath restriction can be used to focus only on certain attributes or element content.
    - Ignore Whitespaces - You should usually check this checkbox as there is always there is a possibility what the words you are searching for may be split on multiple lines in the XML content.
- The **Find** menu → **Find Next/Previous** actions allow you to find the next/previous occurrence of the current selected word(s). Or you can use the **Find All** action to find and highlight all occurrences of a word or a sequence of words.
- The **Find** menu → **Find all Elements** action shows a dialog box allowing you to find elements or attributes containing a sequence of words.
- The **Find** menu → **Quick find toolbar** is a compact toolbar giving you access to search functionality.
- The **XPath** allows you to use **XPath** expressions (i.e. `//comment()[contains(., 'TODO')]`) to search the current topic or a set of topics for elements, attributes, or comments matching a set of conditions.

DITA Maps Manager

The DITA Maps Manager view's contextual menu gives you quite an impressive array of useful actions:

- Fast create new topics. Create a set of DITA topics by specifying only their titles.
- Add references to multiple topics.
- Create key definition with keyword.
- Edit Properties - You can even set profiling attributes on multiple selected topic references.
- Export DITA Map - Export your DITA Map to a zip archive ready for translation.
- Find Unreferenced Resources - Find all resources which are no longer used in a DITA project.
- Refactoring menu
    - Rename/Move topic - Renames a topic and all references to it.
    - Rename key - Rename a DITA key and all its references in the entire project.
    - Conversions between topics - Convert between topics, tasks, concepts, and references for multiple selected DITA files.
    - Convert nested sections to new topics - Convert all sections located inside a topic to new topics.
    - Convert nested topics to new topic - Convert all nested topics to new topic files.
    - Apply custom refactoring scripts - Create and apply your own XSLT or XQuery refactoring scripts.
- Find/Replace in Files.

- Spell Check in Files.
- Search References - Search all references to a particular referenced topic.
- Remove from Disk - Check in how many places a topic is referenced and then remove it from disk.
- Open with resolved topics - Open the DITA Map with all topic references expanded, useful if you want to have an overview of the entire publication.
- Synchronized selection between DITA Maps manager and main editor area - Anytime a DITA topic is opened both in the DITA Maps Manager and the main editing area, moving the selection in one instance will also move the selection to the corresponding place in the other.

Editing in the Author Visual Editing Mode

- Maximize editor area - Double-click the opened editor's tab to hide all side views and maximize editing space, double click again to show the side views.
- Increase/decrease editor font - Use **Ctrl/CMD +/-** to increase or decrease the font in the current edited document.
- Inserting elements - Pressing ENTER in the Author visual editing mode will show you a list with all possible elements that can be inserted. If you choose an invalid element, Oxygen will find a place for it. You can also add custom actions or code templates to the content completion list.
- Select content then press **ENTER** to surround the content in a new element.
- Select multiple intervals - Press the **Ctrl/CMD** button when selecting text to select multiple text intervals. Once you do that, you can use toolbar actions like **Bold**, **Italic**, or **Underline** or you can use the **Edit Profiling Attributes** action to set profiling attributes on all the selected items.
- In-place attributes editor - Instead of using the **Attributes** view, you can press **Alt-Enter** to edit attributes for the current element in a small pop-up dialog box.
- Select an entire element - Triple-click inside an element's contents to select the entire element. After this, you can move the element to a new place.
- Double-click to select by word, then drag the mouse to select the content word by word.
- Move elements (especially paragraphs, list items) up or down (**ALT-UP/DOWN**), indent or unindent list items (**Tab**, **Shift-Tab**).
- Select content and then use the **Toggle comment** action (contextual menu **Refactoring** submenu) to quickly add an XML comment around it.
- In the contextual menu the **Text** submenu contains useful actions to count the number of words in the entire document (or in the selected text), convert the selection to upper, lowercase, or sentence case
- Bookmarks - Click the vertical left side bar in the editing area to add a bookmark. Navigate to that bookmark even after the file has been closed using the **Ctrl/CMD + number** shortcut.
- Collapse other folds - Click a fold triangle in the Author visual editing mode and you can choose to close all other folds, useful when working with multiple sections in the same file.

- Code templates - Define small fragments of XML content that can be inserted either by defining a shortcut key or by pressing **ENTER** in the editing area.
- Editor variables - Certain Oxygen-specific macros can be automatically expanded. For example, a code template that inserts the current author name can use this editor variable:

  ```
  <author>${author.name}</author>
  ```

  or a code template that surrounds the selection inside a `<keyword>` element:

  ```
  <keyword>${selection}</keyword>
  ```

  or a code template that first asks the end user for their name and then inserts it in the document:

  ```
  <author>${ask('Author Name', generic, 'John Doe')}</author>
  ```

  Code templates can also be used when creating new file templates *(on page 413)*.
- Outline Quick find filter - The Outline view contains a filter that can be used to reduce the number of elements displayed in it (for example, display only the contained topics).

DITA-specific Editing in the Author Visual Editing Mode

- DITA Reusable Components view. You can use this side view to quickly search for and insert references to keys or to reusable components.
- Inserting links to resources. Besides using the toolbar actions you can also:
  - Drop a file from an outside location. Drag a DITA topic or other resource from the Explorer/Finder or from the Oxygen Project and drop it inside an opened DITA topic to insert a reference to it.
  - Paste a URL in the editing area to insert a reference to it.
- Links to images:
  - Drop images from the Project view in the main editing area to insert references to them.
  - Copy an image and then paste it in the editing area to insert a reference to it.
  - Double-click broken image reference to modify attributes.
- Reuse content:
  - Copy an element that has an ID set on it and then right-click elsewhere and use the "**Paste Special** → **Paste as content reference**" to insert a content reference to the element.
- Select multiple sibling paragraphs and use the **Insert list item** toolbar item to convert them all to list items. Select multiple list items and use the **Insert table** toolbar action to convert them to a table.
- Working with tables:

    - You can right-click inside a table and use the **Table Properties** action to change lots of table-related settings.
    - You can sort tables based on certain criteria.
    - In the contextual menu **Refactoring** submenu you can find actions to convert between CALS and simple tables.
    - You can select an entire table by clicking its left corner and select an entire row by clicking to the left of it.
- Add hotspots to images - You can right-click any image reference and use the **Image Map Editor** to configure target links for various parts of the image.
- You can right-click inside an element and use the **About Element** contextual menu item to find out more about it.
- Pasting content from web browsers, Excel spreadsheets, or Word documents inside a DITA topic produces the equivalent DITA content.
- The **Styles** drop-down toolbar menu allows you to choose between various CSS styles to apply while editing. The **Hints** and **Inline actions** layers should be interesting for you to experiment with. Or you can add your own *(on page 245)*.

Preferences:

Oxygen has a lot of global settings that can be configured, I will just list what I consider to be the most important ones:

Menu shortcut keys - You can use this preferences page to assign or to see the shortcut for any action available in Oxygen.

Fonts - This page allows you to change the default fonts used in the application.

Appearance - Change the default color theme in Oxygen (the **Graphyte** theme is quite popular).

Spell check - Customize the spell checker settings, use custom spell check dictionaries.

Save - Settings to automatically save or to check for errors before saving the file.

External tools - Define various command line tools that you can launch from Oxygen.

Annotations - Disable the tooltips which appear when hovering over various elements (useful if you have a small screen).

Tools

The **Tools** main menu is worth exploring sometimes. You can invoke XML refactoring actions from it or open other small applications like the SVG viewer.

Import

The **File** menu offers the possibility to import HTML, CSV, Excel files, or database content as XML.

Other Views and Toolbars:

You can right-click in the toolbar area and use the **Configure Toolbars** action to see what toolbars are available or what toolbars can be removed.

The **Symbols** toolbar is quite interesting if you often want to insert symbols that are not on the current keyboard.

Other Small Tips:

The main **Window** menu has actions to tile all opened XML documents and also to enable synchronous scrolling in them (useful if you want to look at similar XML documents and maybe to copy/paste between them).

Keeping in Touch

Oxygen's **Help** menu allows you to use the **Report problem** action to contact us directly. The **Support Tools->Randomize XML Content** action allows you to randomize the content of an XML project before sending it to us for tests.

And there are a lot of other ways to get in touch with us or to find various videos or tutorials to read: https://www.oxygenxml.com/technical_support.html.

DITA Project Enhancements

A DITA project file defines a set of deliverables. For each deliverable you have a context containing the published DITA Map and filters. The deliverable also specifies one or more transtypes, for each transtype specifying a set of parameter names and values. The file format of the DITA Project can be either XML or JSON. So what could we do with the DITA Project in Oxygen?

Publishing

In the future the DITA Open Toolkit (probably version 3.4 or 4.0) will allow publishing multiple deliverables by providing such a DITA project from the command line.

Editing the DITA Project

We need a special framework containing a schema, custom CSS for editing the DITA project in the Oxygen Author visual editing mode. We also need an ANT-based transformation scenario for publishing the DITA project.

Adding the DITA Project in the "Master Files" folder

Once such a DITA project is added to the "Master Files" folder Oxygen could parse it and obtain a list of (root map, filter) pairs. We can use this list of root maps when renaming or moving content (topics, maps, resources) in the Project view. We can use the (root map, filter) pairs to allow in the DITA Maps Manager selecting such a pair if somebody wants to edit a topic in such a particular context.

Using the DITA Project information for editing

We have a specific drop-down "Filters" button both in the DITA Maps manager view and in the main editing area. The button takes effect over all the files opened in the application. We also have a "Root map" combo box in the DITA Maps Manager. Again, it takes effect over all the maps and topics opened in the application. The DITA Project would contribute (root map, filter) pairs which could be contributed in the UI. Where could we contribute them? Maybe the "Root map" combo box could become a "Contexts" combo box listing either DITAVAL files or (map, filter) pairs. If we want to allow for certain files people to use certain editing contexts (thus avoiding to apply the editing context on a per-application wide basis) we have an architectural problem, the DITA Keys resolver is more or less a singleton object, but you would need sometimes multiple such keys resolvers.

Using the DITA Project for the "Validate and check for completeness" action

Right now the "Validate and check for completeness" starts validation from a particular DITA Map and has a set of DITAVAL filters which can be set. Maybe if a DITA Project is opened in the main editing area, we could have a special "Validation" button for it on the toolbar, validating all the DITA Maps with all the filters specified inside it.

DITA Project extensibility

From what I discussed with Jarno, ideally in the DITA project file we could include our own XML tags from a specific namespace. In this way, for a certain deliverable context we could for example also define the "reusable" content folders, the "images" folders, information which could be used for information displayed in our "DITA Reusable Components" view.

Translating your DITA Project

Usually when working with a **DITA**-based project you can either store the project contents using a **Content Management System (CMS)** or some open-source version control system like **Git** or **SVN**. **CMSs** usually come with their own translation support so this blog post is mostly for end users who use **Git** or **SVN** to store and collaborate on their **DITA** project.

Choosing a Translation Agency

Ideally your translation agency should be able to handle **DITA** content directly, without you needing to convert the **DITA** to some intermediary format. This means that you will have the full benefit of **DITA** reuse features to minimize translation costs.

As a very important rule, if you plan to translate your project you should get in touch with a **DITA**-aware translation agency very early in your project's timeline. Reliable translation agencies that translate **DITA** content directly (for example WHP) usually need to have a preliminary discussion with you about how the project is structured, what terms need to be skipped when translating, how various measuring units are translated, content reuse, taxonomy, and the handling of screenshots that appear in your **DITA** content. So the way that you write your **DITA** content will be influenced by your discussion with the translation agency.

If your translation agency does not directly handle **DITA** content, there are commercial tools that can be used to convert **DITA** to **XLIFF**: https://www.maxprograms.com/products/fluenta.html.

Optimizing Content for Translation

In general, there are three main principles to take into account when writing **DITA** content that will be translated at some point:

1. Use a controlled vocabulary (usually the Simplified Technical English vocabulary).
2. Avoid reusing inline elements other than product names. The following *DITA Users List* discussion describes the reasons for this: https://lists.oasis-open.org/archives/dita/201301/msg00029.html.
3. Avoid profiling/filtering content at inline level. For the same reasons as (2).

General DITA Project Structure

Usually you need to keep a folder that contains all your **DITA** maps/topics in English and have separate folders for other languages with equivalent **DITA** topics translated in that specific language. This article could be useful: https://www.maxprograms.com/articles/organize_files.html.

General Translation Workflow when the Translation agency accept DITA documents

When translating **DITA** content, the most common process involves these steps:

1. You create your content in the primary language using a **DITA** authoring tool (**Oxygen XML Editor**).
2. Before each release, you gather all the **DITA** topics that have been changed and need to be translated. The Oxygen Translation Package Builder plugin might be handy for this.
3. Send a copy of the relevant **DITA** files to the translation agency (known also as "localisation service provider").
4. Receive translated **DITA** content back from the translation agency and integrate it in each language-specific project folder.

Translation Workflow when the Translation agency accepts XLIFF files

XLIFF (**XML Localization Interchange File Format**) is an XML-based format created to standardize the way data are passed between and among tools during a localization process. If your translation agency accepts this format, the translation workflow usually has these steps:

1. At various milestones (for example, when a new version is released), you generate XLIFF files for each language you translate to.
2. You send the XLIFF file to the translation service provider.
3. Once the XLIFF returns from translation, you generate a translated version of your map and topics from the XLIFF file.

> **Important:**
> The Fluenta DITA Translation add-on can help with all of these steps.

Publishing your Translated Content

All your translated DITA maps and topics should have the **xml:lang** attribute set with the appropriate value on the root element. Besides the actual translated content, the published output may contain various static text (such as the word **Table** followed by the table number, **Figure** following by the number, or **Note** appearing before each DITA `<note>` content). The **DITA Open Toolkit** includes support for various languages for HTML-based output and PDF-based output. You can also add support for other languages: http://www.dita-ot.org/dev/topics/plugin-addgeneratedtext.html#ariaid-title1. There is also a specific topic that describes how to add a new language to the Oxygen-specific **WebHelp Responsive** output: https://www.oxygenxml.com/doc/ug-editor/topics/localize-webhelp-responsive.html.

Liability

So who is responsible for a bad translation that may produce damage to a client following a set of mis-translated steps? From my discussions with translation service providers, the translation agencies do not assume any liability for incorrectly translated content. Usually a company that needs to translate their

DITA content in multiple languages has regional headquarters in various countries and somebody from the company's regional headquarters would be responsible to review and accept the translated content as appropriate.

This concludes my **DITA** translation overview. As we do not translate the Oxygen User's Manual in various languages, our internal knowledge of translating **DITA** content is quite limited so any feedback on this small article is welcomed.

DITA Map Validate and Check for Completeness Overview

The **Validate and Check For Completeness** is an action available on the **DITA Maps Manager** view toolbar and it can be used to perform thorough checks on the entire **DITA Map** structure and set of referenced topics. We've made this action available to you a couple of years ago and during these years, based on your direct feedback we kept adding additional checks and functionality to it. We always took care to optimize the processing speed in order to allow for validating projects containing thousands of resources in 10-15 seconds.

In this blog post I will try to make a list of all the checks that the action does in order to ensure you that your DITA content is valid and proper:

- Validate each **DITA** resource directly or indirectly referenced from your **DITA Map** with its associated **DTD** or **XML Schema** and report any errors which may arise.
- Validate each **DITA** resource with an additional **Schematron** resource which you can provide. **Schematron** is quite handy when it comes to enforcing internal rules on the DITA content and we use it quite a lot for checking our user's manual *(on page 291)*.
- Batch validate referenced DITA resources. This setting validates each **DITA** resource according to the validation scenario associated with it in **Oxygen**. This will decrease the validation speed quite a bit but if you have **DITA** 1.3 resources which are **Relax NG** based you should check it in order to validate each resource according to the **Relax NG** Schema.
- Use specific **DITAVAL** or profiling condition filters when performing the validation. From a single published **DITA Map** you may get multiple publications based on the profiling filters applied. Because these filters are used to remove entire topics or parts of topics, you may have **links** and **conrefs** which become invalid when certain filters are applied on the map. So it makes sense to validate your DITA project by applying all profiling filters you would apply when publishing it in order to be aware of these potential broken references.
- Report profiling attributes or values which are not valid according to the **Subject Scheme Map** associated with your project. You can read more about controlling profiling attributes and values here:Controlled Attribute Values for your DITA Project *(on page 250)*.

- Identify possible conflicts in profile attribute values. When the profiling attributes of a topic contain values that are not found in parent topic profiling attributes, the content of the topic is overshadowed when generating profiled output.
- Check the existence of **non-DITA** referenced resources. You will get reports if links to local images or other resources are broken. You can also decide to verify the existence of remote links. For example if you have links to various external web sites, you might be interested in seeing if those remote servers are still there.
- Report links to topics not referenced in DITA maps. Checks that all referenced topics are linked in the DITA map. Otherwise you may get working links to topics which are not included in the table of contents.
- Check for duplicate topic IDs within the DITA map context. By default the topic ID can be used in the WebHelp output for context sensitive help. Also certain CMSs require that a certain topic ID would be unique in the entire DITA Map.
- Report elements with the same ID placed in the same DITA Topic according to the specification.
- Report missing **domains** attribute which may indicate an improper DITA specialization.
- Report invalid **class** attribute values according to the specification.
- Report invalid **key** names according to the specification.
- Report references to **missing keys** or **links** which refer to **keys** which have no target resource defined on them.
- Report problems when elements referenced using DITA content reference range are not siblings or are not properly sequenced.
- Report links which have no target set on them either via **href** or **keyref**.
- Report non-portable absolute references to **DITA** resources.
- Report when links contain invalid encoded characters or **Windows-like** path separators.
- Report when resources are referenced with incorrect path capitalization.
- Report a mismatch between the referenced resource type and its **format** attribute.
- Report a mismatch between the referenced resource type and its **type** attribute.
- Report topic references in a **DITA Map** pointing to non-topic elements in the target topics.
- Report invalid content references and content key references, references to non-existing resources, to non-existing IDs, report when the source element is not a specialization of the target element.

I think I covered most of the checks that this validation does.

Are there other checks you would like to see in a future version? Would you like to see this validation available as a separate process which could be run on a server?

Schematron Checks to help Technical Writing

The Oxygen XML Editor User's Manual is written in DITA. In an older post I described in more detail *(on page 465)* how we collaborate internally on our User's Guide Project. And we also made available a copy of our User's Manual as a project on GitHub.

During these years on working on it, we progressively developed a set of simple rules which were originally kept in a plain text document. The problem is that nobody can really remember all these rules when actually writing. So recently we started to migrate these rules to Schematron and have them reported automatically has validation warnings and errors while editing the topics. And we can also add quick fixes for each of these problems.

So below I will try to tell you what each rule imposes and what it's Schematron implementation looks like. If you want to quickly test these rules on your side, you can add them to the Schematron file which is used by default to validate DITA topics located in: `OXYGEN_INSTALL_DIR/frameworks/dita/resources/dita-1.2-for-xslt2-mandatory.sch`.

1. Try as much as possible to add at least an indexterm element in each topic. This is useful when the **Index** page is created for the PDF output or the **Index** tab is created for the WebHelp output. As this is not a requirement, we wanted to report this issue as an error. The end result looks like this:

And the Schematron pattern looks like this:

```
<pattern xmlns:sqf="http://www.schematron-quickfix.com/validator/process">
 <rule context="/*">
  <assert test="prolog/metadata/keywords/indexterm" role="warn" sqf:fix="addFragment">
      It is recommended to add an 'indexterm' in the current '<name/>' element.
  </assert>
  <!-- Quick fix to add the indexterm element element and its parents -->
  <sqf:fix id="addFragment">
      <sqf:description>
          <sqf:title>Add the 'indexterm' element</sqf:title>
      </sqf:description>
```

```
      <sqf:add match="(title | titlealts | abstract |
 shortdesc)[last()]" position="after" use-when="not(prolog)">
          <xsl:text>
    </xsl:text><prolog xmlns=""><xsl:text>
      </xsl:text><metadata><xsl:text>
        </xsl:text><keywords><xsl:text>
           </xsl:text><indexterm><xsl:text> </xsl:text> </indexterm><xsl:text>
        </xsl:text></keywords><xsl:text>
      </xsl:text></metadata><xsl:text>
    </xsl:text></prolog>
      </sqf:add>
  </sqf:fix>
 </rule>
</pattern>
```

2. The ID of each topic must be equal to the file name (minus the extension). One of the outputs we produce (I think CHM) had a limitation when building the context mapping between help IDs and actual HTML content so this was an important rule for us, thus an error is reported on this. Also a quick fix is added to auto-correct the topic ID based on the file name. The end result looks like this:

and the Schematron pattern is:

```
<!-- Topic ID must be equal to file name -->
<sch:pattern>
 <sch:rule context="/*[1][contains(@class, ' topic/topic ')]">
  <sch:let name="reqId" value="replace(tokenize(document-uri(/), '/')[last()], '.dita', '')"/>
  <sch:assert test="@id = $reqId" sqf:fix="setId">
   Topic ID must be equal to file name.
  </sch:assert>
  <sqf:fix id="setId">
   <sqf:description>
    <sqf:title>Set "<sch:value-of select="$reqId"/>" as a topic ID</sqf:title>
    <sqf:p>The topic ID must be equal to the file name.</sqf:p>
```

```
    </sqf:description>
    <sqf:replace match="@id" node-type="attribute" target="id" select="$reqId"/>
   </sqf:fix>
  </sch:rule>
 </sch:pattern>
```

3. Report when plain links or related links to web resources have the same text inside them as the value of the **@href** attribute. We had cases in which writers would input web links like this:

```
<xref href="http://www.google.com" format="html" scope="external">http://www.google.com</xref>
```

which is redundant because when you set no text to the link, the publishing uses the **@href** attribute value as the link text. So we wanted to report such cases as warnings and to have a quick fix which removes the link text:

The Schematron pattern looks like this:

```
<sch:pattern>
 <sch:rule context="*[contains(@class, ' topic/xref ') or contains(@class, ' topic/link ')]">
  <sch:report test="@scope='external' and @href=text()" sqf:fix="removeText">
   Link text is same as @href attribute value. Please remove.
  </sch:report>
  <sqf:fix id="removeText">
   <sqf:description>
    <sqf:title>Remove redundant link text, text is same as @href value.</sqf:title>
   </sqf:description>
   <sqf:delete match="text()"/>
  </sqf:fix>
 </sch:rule>
</sch:pattern>
```

4. Avoid using the **@scale** attribute on images. We wanted to smooth scale images in an external image editor so it was prohibited to use the **@scale** attribute on images. The Schematron pattern for this:

```
<pattern>
 <rule context="*[contains(@class, ' topic/image ')]">
```

```
  <assert test="not(@scale)">
   Dynamically scaled images are not properly displayed, you
   should scale the image with an image tool and keep it within
   the recommended with and height limits.
  </assert>
 </rule>
</pattern>
```

We have an upcoming event dedicated to Schematron Quick Fixes if you want to become more familiar with the technology. And we also have a GitHub project which tries to combine the notion of a style guide for writing documentation inside a company with a very simple manner of defining checks which can be applied to impose the styleguide rules.

I would be interested in your feedback, especially if you have checks that you perform right now on your content and you consider that they might benefit others.

Implementing your own Style Guide

Let's say you are a team of tech writers collaborating on a DITA-based project and doing things your way, maybe you have various best practices about what elements to use and when to use them, maybe you want to impose a set of controlled values for certain attributes. So at some point you gather on an internal server a set of HTML resources which explain how various DITA elements should be used. This blog post will attempt to show you how these best practices can be shared with your team so that they are readily available when editing DITA content in Oxygen.

Custom "Style Guide" toolbar button

As you have your style guide HTML resources on a server, you can add a custom toolbar button which will appear on the **DITA** toolbar when editing DITA topics in the Author editing mode. When you press that toolbar button, a web browser opens up and shows you the style guide main page. Here are some steps about how to do this:

1. In the Oxygen **Preferences->Document Type Association** page edit the **DITA** framework. Instead of editing the DITA framework directly you may choose to extend it in order to share the extension more easily:Document Type Extension Sharing *(on page 407)*.
2. In the **Author** tab go to the **Actions** tab and there is an action with the ID **styleguide**. If you edit the action, it invokes an operation with a parameter called **resourcePath**. You can edit that parameter to point to your internal (or public) server where the WebHelp output is stored. You should also set

an icon to it, you can use **/images/BrowseReferenceManual16.png** (it's a default icon which comes with Oxygen). Save your changes in that dialog.

3. In the **Author** tab there is a **Toolbar** sub-tab in which you can add the **styleguide** action to the toolbar in the place where you want it. Press **OK** a couple of times in the dialogs and then action should become available on the toolbar for each topic.

Link to Style Guide for each element in the content completion window

When you press the **ENTER** key in the Author editing mode, you get a list of available elements. For each element there is documentation available, that documentation can be customized, for example you could add links for each element to a specific section in your style guide. This topic should tell you more about how this can be achieved:https://www.oxygenxml.com/doc/ug-editor/#tasks/author-styleguide-annotations.html

Impose controlled attribute values

For certain attributes (for example profiling attributes, **@outputclass** attributes) you may want to impose a set of controlled attribute values. This blog post will tell you how: Controlled Attribute Values for your DITA Project *(on page 250)*.

Show validation errors or warning when guidelines are breached

If possible, some of your rules can be converted to Schematron, allowing the application to signal to the writer when a rule is not obeyed. You can also add quick fixes to show writers various ways to rectify the problem. This blog post should give you more details about this:Schematron Checks to help Technical Writing *(on page 291)*.

Bringing all of this together

There is an Oxygen XML GitHub project called **DIM** which attempts to approach most of these aspects in an unified manner: https://github.com/oxygenxml/dim.

Preprocessing DITA-OT Project Files

Project files were introduced in the DITA-OT 3.4 release. They provide a standardized XML way to define how input DITA files should be published to output content files, including details such as filtering, transformation parameters, and output directory locations.

Our basic publishing requirements are as follows:

- We publish some books multiple times in multiple DITAVAL filtering conditions.
- We publish to both PDF (using PDF Chemistry) and online help (using Oxygen WebHelp).
    - PDFs are published individually per-book.
    - WebHelp is published as a collection of books, with in-help links to the PDF files.
- We have "review" and "final" versions of our output.
    - These versions use different DITA-OT parameters and different DITAVAL flagging files.

As I attempted to create a DITA-OT project file to produce our deliverables, I encountered some limitations. This blog post describes how I created an XSLT-based preprocessing approach to work around these limitations.

Quick Overview of Project Files

A project file uses three primary building block elements:

- `<content>` - an input DITA map to process
  - Can include one or more associated DITAVAL files
- `<publication>` - a transformation to apply
  - Can include transformation parameters
- `<deliverable>` - output content to create, by transforming a `<context>` with a `<publication>`
  - Can include an output subdirectory path (relative to the overall output directory)

In its simplest form, a `<deliverable>` can provide its own `<context>` and `<publication>` information within itself:

```
<deliverable id="del-pdf-A">

  <context>
    <input href="dita/common_UG.ditamap"/>
    <profile>
      <ditaval href="filter_A.ditaval"/>
    </profile>
  </context>

  <output href="olh-A/pdf">

  <publication transtype="pdf2">
    <param name="..."/>
    <param name="..."/>
  </publication>

</deliverable>
```

For more complex output content situations, a `<deliverable>` can reference shared `<context>` and `<publication>` elements by `@idref` references to `@id` values:

```
<context id="context-A">
  <input href="dita/common_UG.ditamap"/>
  <profile>
    <ditaval href="filter_A.ditaval"/>
  </profile>
</context>
```

```
<publication transtype="pdf2" id="pub-pdf">
  <param name="..."/>
  <param name="..."/>
</publication>
```

```
<deliverable id="del-pdf-A">
  <context idref="context-A"/>
  <output href="olh-A/pdf">
  <publication idref="pub-pdf"/>
</deliverable>
```

This `@idref` mechanism allows many deliverables to share common context and publication definitions. If there is a change to a `<context>` (perhaps a different map or new DITAVAL condition) or a `<publication>` (perhaps an updated DITA-OT parameter), then all relevant deliverables inherit the change automatically.

In addition, DITA-OT project files can use `<include>` statements to structure their information across multiple files. This allows contexts to be organized by product writer teams, publications to be placed in files maintained by a DITA environment maintainer, and so on.

Limitation – Specify Per-Deliverable PDF File Names (#3682)

When I publish multiple PDFs from the same map using DITAVAL conditions, I needed to define the `outputFile.base` parameter on a per-`<deliverable>` basis to control the output PDF file name:

DITA-OT versions before 4.0 do not allow `<param>` elements to be controlled from a `<publication>` reference in a `<deliverable>`. I filed the following DITA-OT enhancement request for this:

#3682: In DITA-OT project files, allow a PDF <deliverable> to specify its output file name

It was implemented for DITA-OT 4.0 in the following pull request:

#3907: Support param in publication reference

Limitation – Consider DITAVAL in Both <context> and <publication> (#3690)

I needed to apply DITAVAL from both `<context>` (for `@product` filtering) and `<publication>` (for `@audience`/`@deliveryTarget`/`@rev` filtering/flagging of "review" and "final" deliverables):

DITA-OT versions before 4.0 do not properly combine `<context>` and `<publication>` DITAVAL filtering. I filed the following DITA-OT issue for this:

#3690: In DITA-OT project files, apply both <context> and <publication> DITAVAL filtering

It was implemented for DITA-OT 4.0 in the following pull request:

#3907: Add profiles to publication project file

Using Preprocessing to Work Around the Limitations

To work around these limitations in earlier DITA-OT versions before 4.0 is released, I created an XSLT file to do the following:

- Read the input DITA-OT project file
    - Resolve `<include>` statements to pull all content into a single file
- Convert all DITAVAL file references to absolute paths (to work around #3873)
- In `<deliverable>` elements, replace all `@idref`'ed `<context>` and `<publication>` elements with copies of the referenced elements (so we can modify them per-`<deliverable>`)
- Find `<param>` elements in `<deliverable>`, move them to `<publication>` instead (to work around #3682)
- Find `<ditaval>` elements in `<publication>`, move them to `<context>` instead (to work around #3690)

I then applied this XSLT file as a preprocessing step to translate the unsupported project file constructs into supported constructs in a temporary preprocessed project file, then ran DITA-OT publishing using that temporary file. For example,

```
#!/bin/bash

rm -rf ./out


export DITAOT=$(dirname $(dirname $(which dita)))

export SAXON_JAR=~/saxon/saxon-he-10.6.jar


echo "Creating preprocessed DITA-OT project file..."

java \

  -jar ${SAXON_JAR} \

  -xsl:frameworks/dita/preprocess_project_file.xsl \

  -s:project.xml \

  -o:project.xml-preprocessed.xml


echo "Publishing preprocessed DITA-OT project file..."

${DITAOT}/bin/dita --project project.xml-preprocessed.xml -t temp --verbose


rm project.xml-preprocessed.xml
```

This worked well from a linux command line, but we also needed our writers to be able to run it from Oxygen. To do this, I created a copy of Oxygen's project file build script at

```
<OXYGEN_INSTALL>/frameworks/dita/dita_project/build_dita_project.xml
```

and added similar XSLT preprocessing to it using Ant commands, then placed the modified version at

```
frameworks/dita/build_dita_project_preprocessed.xml
```

in our Oxygen project directory. Then I extended the DITA-OT project file framework and created an extended DITA-OT project file transformation pointing to the modified build script:

This new transformation allowed Oxygen to publish project files that used the preprocessing workaround. The unsupported constructs still result in schema violations when the original (non-preprocessed) project files are opened for editing in Oxygen, but at least the publishing aspect works.

Note:

The preprocessing XSLT stylesheet requires Saxon to run. To support, this, the preprocessing-based DITA-OT project transformations specify a list of additional `.jar` libraries to use. To see these libraries, click the **Libraries** button in the dialog box shown above.

The following Oxygen project demonstrates this preprocessing approach:

preprocessed_ditaot_project_files.zip

To run it,

1. Extract the archive and open `OPENME.xpr` in Oxygen.
2. In the **Project** view, expand the **Main Files** list, right click on deliverables-all.xml and choose **Transform > Transform With**, and choose the **Publish Preprocessed DITA-OT Project (all deliverables)** transformation.

   This will build "review" and "final" versions of both the "Product A" and "Product B" online help collections in the `out/` directory, complete with correctly-named PDF files integrated into each online help collection.

   There are other deliverable files for specific subsets of deliverables, organized into logical folders in the **Main Files** list.

To view the XSLT stylesheet without downloading the archive, click on the following link:

preprocess_project_file.xsl

There are comments in the code to explain how it works.

Exploring How the XSLT Transformation Works

To help you explore how the XSLT transformation works, the Oxygen project also makes it available as a refactoring operation that you can manually preview on project files.

To do this,

1. In the **Project** view in the **Main Files** list in the `project_files/` directory, right-click one of the `deliverable*.xml` files, then choose **Refactoring > XML refactoring**.
2. In the refactoring operation list, choose **Synopsys > Preprocess DITA-OT project file** refactoring operation.
3. Click the **Preview** button to see what the XSLT transformation would do.

For example,

Be sure not to actually apply the refactoring operation to the file. Otherwise, you will need to re-extract the archive to restore the original project file.

Re: 10 reasons for moving away from DITA

I'm following Tom Johnson `@tomjohnson` on Twitter and so should you for the very interesting articles related to technical documentation. One of his recent posts reminded us about the 10 reasons for moving away from DITA article he wrote about 7 years ago so I re-visited the article and I wanted to reply to each of the points he makes in the **10 reasons to move away from DITA** section, 7 years after his original post:

1. **DITA doesn't easily integrate into existing web frameworks.**

   I think the **DITA to Markdown** transformation developed by Jarno Elovirta (which is freely available in the DITA Open Toolkit) has opened the possibility of using static web site

generators. Here is an article I wrote where I investigated using the **MKDocs** static site generator:

Publishing DITA Content Using the MKDocs Web Site Generator *(on page 222)*.

2. DITA doesn't easily integrate with JavaScript libraries.

Again, once you can publish DITA to Markdown, you can use the existing static web site generator infrastructure.

3. I never bought into DITA's information typing model.

I partially agreed, as most writers who are not externally constrained by the company or by some publishing customization to use explicit topic types (me included) would use the base DITA `topic` type for all of their work.

4. Writing in XML is a chore.

For me, when writing in Markdown, adding paragraphs is easy, but when adding links, image references, or tables, it is a chore. It depends on what you are comfortable with and what you use every day. If I were to write Markdown every day, I would start to remember how I should encode a link and consider it a natural way of doing things. But indeed, using a plain text editor to write in XML is more difficult, in general. With a visual editing tool, it is much easier.

5. Open-source DITA solutions develop too slowly.

If we are discussing storage, DITA content (the content of this blog for example) can be stored in Git repositories and used with regular Git client tools: DITA For Small Technical Documentation Teams *(on page 266)*.

If we are discussing the publishing ecosystem, once you can publish DITA to plain HTML or Markdown using the DITA Open Toolkit, you can benefit of all web based tools available for Markdown and plain HTML content.

6. You can't easily customize the output.

Again, it depends on what you are comfortable with. You can use Markdown content produced from DITA XML content with static web site generators like **MKDocs** or **Jekyll**. Indeed, if you use (for example) DITA to produce Oxygen WebHelp, although most of the customizations are made with CSS, you might still need to use XSLT for more advanced customizations. But as someone who spent hours trying to understand why **Hugo** does not properly follow links between articles, there is a learning curve for anything, and every static web site generator out there has its own configuration capabilities that need to be

learned and explored. Also, once you get comfortable with a certain publishing process, you naturally consider that one easy and the others more difficult.

About publishing DITA to PDF, the default customization capabilities for the classic DITA to PDF publishing require knowledge of both XSLT and XSL-FO. This is why we invested in the Oxygen Chemistry CSS-based PDF processor, to be able to style the content using CSS and then use the CSS both for WebHelp and PDF publishing: https://styles.oxygenxml.com/.

7. DITA doesn't integrate well with other non-DITA content.

DITA content integrates well with Markdown. We contacted Tom less than a year ago to again explore Oxygen's capabilities with a docs-as-code approach and his well thought out and complete article about combining DITA XML with Markdown is available here: https://idratherbewriting.com/learnapidoc/pubapis_oxygenxml.html.

In recent Oxygen versions, we also explored integrating other file types like Word or HTML directly with DITA projects: https://www.oxygenxml.com/doc/ug-editor/topics/dynamically_convert_word_excel_html_markdown_to_dita.html.

8. Everything I wanted to do with DITA, I can do with Markdown and Liquid on Jekyll.

I am not familiar with Liquid, but I agree there are indeed ways that you can also reuse content with Markdown, not part of a standard but part of how a particular framework decided to support such extensions, making the solution 100% not portable when switching between web-based frameworks. I do think there are more powerful ways to check if the structure is correct with XML-based standards using Schematron. The Oxygen Validate and check for completeness *(on page 289)* action also does a lot of consistency checks.

An article about thoughts on Markdown: https://www.smashingmagazine.com/2022/02/thoughts-on-markdown/.

9. Innovation with DITA is too reliant on vendors.

Once you can obtain plain HTML and plain Markdown from DITA XML content, you can benefit from all the tools created to process and display HTML and Markdown. The web tools publishing space is much more dynamic than the DITA XML publishing space. For me, looking into the web tools space from the outside, it's in a constant turmoil of innovation, which brings stress to any web developer who might want to choose a framework since there are so many frameworks developed over night while re-inventing solutions for the same concepts in different ways that you end up not knowing what to choose and with the Markdown content containing various framework-specific extensions not being able to easily switch between them. It's like tearing down the entire house and re-inventing it each day and sometimes forgetting about various problems that have been fixed in the

previous iteration. While Markdown might now be the future winning format when it comes to web tools, some innovators want to switch to JSON even for writing web content: https://www.smashingmagazine.com/2022/02/thoughts-on-markdown/.

10. **DITA is the wrong language for the API doc space.**

    I would like to further explore using DITA XML in the API docs space, that's for sure. Now days, I think by "API Docs", most people would be referring to Swagger, OpenAPI documentation, mostly API documentation for web-based end points. I experimented a bit using `widdershins` to generate documentation from API docs to Markdown and then converting this Markdown content to DITA XML, converting the DITA content to WebHelp Responsive. Also, I'm not sure if the API docs world is larger than web-based end points using REST, but there are other programming languages as well but probably most API documentation is about accessing server end points now days.

So these are my brief remarks on Tom's **10 reasons to move away from DITA** article written some 7 years ago. Any feedback is welcomed as usual.

DITA 2.0 Specification Support Cheat Sheet

You can use the **Convert DITA 1.3 topics and maps to the DITA 2.0 standard** XML refactoring action bundled with Oxygen 24 (or later) to convert DITA 1.3 content to the DITA 2.0 standard. The same XML refactoring action is also available as a separate GitHub project: https://github.com/oxygenxml/dita_1_3_to_2_x_converter.

The DITA Open Toolkit publishing engine will support publishing a mix of DITA 1.3 and 2.0 topics and maps. The list of supported DITA 2.0 features in the publishing engine is available here: https://www.dita-ot.org/dev/reference/dita-v2-0-support.html.

Table 3. DITA 2.0 Support Cheat Sheet

Change	Details	Handled By Conversion	Supported in DITA OT Publishing Engine
The DITA 2.0 topics/maps have specific schema references.	References to DTD/XML Schema/RNG schemas need to be changed in all topics and maps.	Yes	Yes
Removed elements: `titlealts, itemgroup, topicset, topicsetref,`	Removed some seldom used elements.	Yes, automatically removes the elements and	-

Table 3. DITA 2.0 Support Cheat Sheet (continued)

Change	Details	Handled By Conversion	Supported in DITA OT Publishing Engine
`longquoteref, anchor, anchorref, hasInstance, hasKind, hasNarrower, hasPart, hasRelated, relatedSubjects, subjectRelTable, subjectRelHeader, subjectRel, subjectRole`		shows messages for each removed element.	
Removed attributes: `@mapkeyref, link/@query, hazardsymbol/@longdescref, @xtrf, @xtrc, @spectitle,@specentry`	Removed some seldom used attributes.	Yes, automatically removes the attributes and shows messages for each removed attribute.	-
Removal of `@navtitle`/`@locktitle`/`@lockmeta` attributes	Removed deprecated attributes, they can be replaced with equivalents.	Yes, automatically converts them to `<topicmeta>` equivalents.	Yes
Removal of `@domains` default attribute.	This is a default attribute that does not appear in the topic and map instances. It was used mostly for defining new attribute specializations, and all attribute specialization schemas need to be re-written to use the new `@specializations` default attribute.	-	Yes

Table 3. DITA 2.0 Support Cheat Sheet (continued)

Change	Details	Handled By Conversion	Supported in DITA OT Publishing Engine
Removal of **@copy-to** attribute	Needs to be replaced with a `<resourceid>` element.	Yes	No
Removal of the `@print` attribute	Needs to be replaced with the `@deliveryTarget` attribute.	Yes	-
Removed `<substep>` and `<substeps>` task elements	Needs to be replaced with the `<steps>` and `<step>` elements.	Yes	No
Removed `<sectiondiv>` element.	Needs to be replaced with the `<div>` element.	Yes	-
Added new `<audio>` and `<video>` elements.	New elements were added to refer to multimedia audio and video content and to define parameters.	-	Partially
Added new `<em>` and `<strong>` elements	New elements similar in style to `<b>` and `<i>` but with more semantics.	-	No
Added new `<partno>`, `<hw-control>`.	New elements for documenting hardware.	-	No
Added new `<diagnostics>`, `<diagnostics-general>`, `<diagnostics-steps>` elements.	New elements added to the troubleshooting topic type.	-	-
Added new `<include>` element.	Similar to `<coderef>`.	-	Yes
Titles, cell and row span attributes for `<simpletable>` elements.	Simple tables can have titles and their cells can span rows and columns.	-	Yes

Table 3. DITA 2.0 Support Cheat Sheet (continued)

Change	Details	Handled By Conversion	Supported in DITA OT Publishing Engine
Style output using ditaval `@outputclass` attribute.	Profiling attributes defined on DITA elements can be converted to custom `@class` attribute values in the generated HTML output by adding specific `@outputclass` attributes on the filters in the `ditaval` filter file.	-	Yes
Added new `<keytext>` element.	Used to simplify syntax for defining a key pointing to a small text fragment.	-	No
New Bookmap `<mapresources>` element.	A wrapper for various elements like `<keydef>`, `<ditavalref>` which could not be used on the first level of the DITA Book Map.		

Refactoring

Sorting Glossary Lists in a DITA Bookmap

The idea behind this DITA refactoring action started from this forum post: https://www.oxygenxml.com/forum/viewtopic.php?f=2&t=15284&p=45137#p45137.

Suppose you have a DITA Bookmap that at some point references all of its glossary entries:

```
<glossarylist>
    <topicref href="glossary/glossItem1.dita"/>
    <topicref href="glossary/glossItem2.dita"/>
    <topicref href="glossary/glossItem3.dita"/>
```

```
            <topicref href="glossary/glossItem4.dita"/>
        </glossarylist>
```

and you want to have all of these glossary topics sorted alphabetically by title as they appear in the PDF output.

One option for this is to create a PDF customization that automatically sorts the glossary entries no matter what order they were originally specified in the DITA Map.

The other option is to create a custom Oxygen XML Refactoring operation that, when applied on the DITA Bookmap containing the "glossarylist", will sort the glossentries in alphabetical order according to the title of each glossentry.

The following steps will help you achieve this:

1. Somewhere on disk, create a folder (for example, **customRefactor**) and add a reference to it in the Oxygen **Preferences->"XML / XML Refactoring"** page.

2. In that custom folder, create an XML file (for example, **sortGlossentries.xml**) that has the following content:

```
<refactoringOperationDescriptor
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.oxygenxml.com/ns/xmlRefactoring" id="op_tck_rp1_hcb" name="Sort glossentries">
    <description>Sort glossentries.</description>
    <script type="XSLT" href="sortGlossentries.xsl"/>
</refactoringOperationDescriptor>
```

   This particular descriptor file contains the name of the operation, its description, and points to an XSLT stylesheet that will be applied to sort the entries in the DITA Bookmap.

3. In the same folder, create a file called **sortGlossentries.xsl** with the following content:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    exclude-result-prefixes="xs"
    version="2.0">
    <xsl:template match="node() | @*">
        <xsl:copy>
            <xsl:apply-templates select="node() | @*"/>
        </xsl:copy>
    </xsl:template>
```

```
    <xsl:template match="glossarylist | topicgroup[@outputclass='glossarylist']">
        <xsl:copy>
            <xsl:apply-templates select="@*"/>

  <xsl:variable name="closestXMLLang" select="ancestor-or-self::*[attribute::xml:lang][1]/@xml:lang"/>
            <xsl:choose>
                <xsl:when test="exists($closestXMLLang)">
                    <xsl:for-each select="*" >

  <xsl:sort select="document(@href, .)/*/glossterm/text()" lang="{$closestXMLLang}" case-order="lower-first"/>
                        <xsl:apply-templates select="."/>
                    </xsl:for-each>
                </xsl:when>
                <xsl:otherwise>
                    <xsl:for-each select="*" >
                        <xsl:sort select="document(@href, .)/*/glossterm/text()" case-order="lower-first"/>
                        <xsl:apply-templates select="."/>
                    </xsl:for-each>
                </xsl:otherwise>
            </xsl:choose>
        </xsl:copy>
    </xsl:template>
</xsl:stylesheet>
```

This particular XSLT processing copies almost all the Bookmap content unchanged. However, the topicrefs in the **glossarylist** are sorted according to the **glossterm** titles specified inside each of them.

4. Restart Oxygen.

5. Open the bookmap that contains the **glossarylist** in the Oxygen main editing area, right-click, choose **Refactoring** and in the **Other operations** submenu you will find the new **Sort glossentries** refactoring action. Invoke the action and use the **Preview** button to check if the sorting is properly done.

The XML Refactoring XML descriptor and XSLT stylesheet can also be found here: https://github.com/oxygenxml/dita-refactoring-examples/tree/master/18.%20Sort%20Glossentries.

Alternative approach - AI Positron Assistant add-on

An alternative approach is to implement a custom AI action which uses a prompt to instruct the AI to read the contents of the referenced glossentries, extract their titles and then sort them: https://github.com/oxygenxml-incubator/ai-positron-assistant-samples/blob/main/Custom%20Actions/sort-glossentries-dita-map.json

Replacing Direct Image References with Key References in a DITA Project.

Suppose that you have a large **DITA** project and all the image references in your topics are direct references, using the **@href** attribute like this:

```
<image href="../../images/Iris_sanguinea.jpg" scale="50"/>
```

For better scalability and reuse possibilities, suppose you want to convert these direct references to DITA 1.2 key references:

```
<image keyref="Iris_sanguinea.jpg" scale="50"/>
```

Doing something like this manually means making replacements in hundreds of places and also manually building a DITA map that maps the image file name to the image location.

This blog post will try to describe some steps that you will help you to automate this change in your project:

1. The first big step is to generate the DITA Map that maps each image file name (which will be used as a key) to the image location. So, the generated DITA map will look like this:

   ```
   <!DOCTYPE map PUBLIC "-//OASIS//DTD DITA Map//EN" "map.dtd">
   <map>
   …….....
   <keydef href="Iris_sanguinea.jpg" keys="Iris_sanguinea.jpg"/>
   ….....
   </map>
   ```

   We will assume that all images are placed in an **images** folder and we can create an **ANT** build file that lists all the images in a parameter and then calls an **XSLT** script to process the list of images further:

   ```
   <project basedir="." name="Create Image Keys Definition Map">
       <fileset id="dist.contents" dir="images/" includes="*"/>
       <property name="prop.dist.contents" refid="dist.contents"/>
       <xslt in="createKeyrefsMap.xsl" style="createKeyrefsMap.xsl" out="images/imageKeydefs.ditamap" destdir=".">
   ```

```
        <param name="filesList" expression="${prop.dist.contents}"/>
    </xslt>
</project>
```

The **XSLT** stylesheet **createKeyrefsMap.xsl** is responsible for creating the mapping DITA map:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    exclude-result-prefixes="xs"
    version="1.0">
    <xsl:param name="filesList"/>
    <xsl:output doctype-public="-//OASIS//DTD DITA Map//EN" doctype-system="map.dtd" indent="yes"/>
    <xsl:template match="/">
        <map>
            <xsl:call-template name="tokenizeString">
                <xsl:with-param name="list" select="$filesList"/>
            </xsl:call-template>
        </map>
    </xsl:template>
    <xsl:template name="tokenizeString">
        <xsl:param name="list"/>
        <xsl:param name="delimiter" select="';'"/>
        <xsl:choose>
            <xsl:when test="contains($list, $delimiter)">

  <keydef href="{substring-before($list,$delimiter)}" keys="{substring-before($list,$delimiter)}"/>
                <xsl:call-template name="tokenizeString">
                    <xsl:with-param name="list" select="substring-after($list,$delimiter)"/>
                </xsl:call-template>
            </xsl:when>
            <xsl:otherwise>
                <keydef href="{$list}" keys="{$list}"/>
            </xsl:otherwise>
        </xsl:choose>
    </xsl:template>
</xsl:stylesheet>
```

After this step you will have a new DITA map with all image mappings and afterwards you can link it in your main project's DITA map.

2. We still need to make changes to all DITA topics and replace all image hrefs with keyrefs. Oxygen has support for XML Refactoring actions and you can define custom XSLT scripts that can be applied to modify an entire set of topics. In the **OXYGEN_INSTALL_DIR/refactoring** folder, you can add an XSLT script along with an XML description of the refactoring action. An **XSLT** script that would replace **@href** attributes on images with **@keyref** would look like this:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="2.0"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:f="http://www.oxygenxml.com/ns/functions">
    <xsl:function name="f:getKeyref" as="xs:string">
        <xsl:param name="element" as="element()"/>
        <xsl:variable name="imageFile" select="tokenize(translate($element/@href, '\', '/'), '/')[last()]"/>
        <xsl:variable name="key" select="substring-before($imageFile, '.')"/>
        <xsl:value-of select="$key"/>
    </xsl:function>
    <xsl:template match="node() | @*">
        <xsl:copy>
            <xsl:apply-templates select="node() | @*"/>
        </xsl:copy>
    </xsl:template>
    <xsl:template match="image[@href and not(@keyref)]">
        <xsl:copy>
            <xsl:apply-templates select="@* except @href"/>
            <xsl:attribute name="keyref" select="f:getKeyref(.)"></xsl:attribute>
            <xsl:apply-templates select="node()"/>
        </xsl:copy>
    </xsl:template>
</xsl:stylesheet>
```

You can right-click anywhere in the **DITA Maps Manager** view and choose **Refactoring->XML Refactoring**, then use your custom refactoring action to modify all resources.

A set of samples, including the build file, XSLT stylesheets, and refactoring action XML descriptor can be found here:https://www.oxygenxml.com/forum/files/batchImageHrefToKeyref.zip.

Deleting Elements in a Refactoring Operation

Recently, a writer wanted to remove the index from their DITA book. This required the following:

- Removing the `<indexlists>` element from the map:

```
<backmatter>
  <booklists>
    <indexlist/>
  </booklists>
</backmatter>
```

- Removing topic-level `<indexterm>` elements from topic prologs:

```
<topic id="feature_A">
  <title>About Feature A</title>
  <prolog>
    <metadata>
      <keywords>
        <indexterm>feature A</indexterm>
      </keywords>
    </metadata>
  </prolog>
```

- Removing inline `<indexterm>` elements from topic content:

```
<p>This is about <indexterm>feature B</indexterm>feature B.</p>
```

Oxygen provides a "Delete element" refactoring operation. However, it does precisely what it says—deletes the specified elements, leaving everything else in place:

```
<topic id="feature_A">
  <title>About Feature A</title>
  <prolog>
    <metadata>
      <keywords>


      </keywords>
    </metadata>
  </prolog>
```

I decided to create an XSLT refactoring operation that does the following:

- Deletes the specified elements
- Deletes any containing (ancestor) elements that became empty as a result

- Updates whitespace/newline formatting around deleted elements as needed
- Serves as an easily customizable template for other element deletion uses

Fortunately, as described in Custom Refactoring Operations, Oxygen allows us to package up customized XSLT refactoring operations in an easy-to-use way. For the XML descriptor file, put this content into a `remove-index.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<refactoringOperationDescriptor
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.oxygenxml.com/ns/xmlRefactoring" id="remove-index"
    name="Remove index from a DITA book">
    <description>Remove index terms and backmatter index from a DITA book.</description>
    <script type="XSLT" href="remove-index.xsl"/>
    <category>DITA</category>
</refactoringOperationDescriptor>
```

For the XSLT file itself, put this content into a `remove-index.xsl` file:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  exclude-result-prefixes="xs"
  version="2.0">

  <!-- elements to delete -->
  <xsl:variable name="elements-to-delete" select="('indexterm', 'indexlist')"/>

  <!-- delete up to (and including) these elements, if they become empty -->
  <xsl:variable name="delete-up-to" select="('prolog', 'backmatter')"/>

  <!-- baseline identity transform -->
  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
```

```
<!-- remove elements-to-delete -->
<xsl:template match="*[name() = $elements-to-delete]"/>


<!-- remove whitespace/newlines before elements-to-delete -->
<xsl:template match="text()
                     [following-sibling::*[1]
                       [name() = $elements-to-delete]]
                     [matches(., '^\s*\n\s*$')]"/>



<!-- remove elements that contain our to-be-deleted elements,
     but only if they become empty -->
<xsl:template match="*[ancestor-or-self::*[name() = $delete-up-to]]
                      [descendant::*[name() = $elements-to-delete]]">


  <!-- apply templates to this element's contents and see what we get -->
  <xsl:variable name="contents" as="node()*">
    <xsl:apply-templates select="node()"/>
  </xsl:variable>


  <!-- if children elements remain, copy this element (and its preceding whitespace/newlines)
       and put its contents inside -->
  <xsl:if test="$contents[self::*]">
    <xsl:copy select="preceding-sibling::node()[1][self::text()][matches(., '^\s*\n\s*$')]"/>
    <xsl:copy select=".">
      <xsl:sequence select="$contents"/>
    </xsl:copy>
  </xsl:if>
</xsl:template>


<!-- remove whitespace/newlines before elements-to-delete
     (we re-add whitespace/newlines above, if needed -->
<xsl:template match="text()
                     [following-sibling::*[1]
                       [ancestor-or-self::*[name() = $delete-up-to]]
                       [descendant::*[name() = $elements-to-delete]]]
                     [matches(., '^\s*\n\s*$')]"/>
```

```
</xsl:stylesheet>
```

At the beginning of the refactoring operation, two XSLT variables are defined:

- `elements-to-delete` - the element names to delete, regardless of their contents
- `delete-up-to` - the highest-level containing element names to delete, if they become empty

The refactoring operation works as follows:

- The `elements-to-delete` elements are always deleted.
  - Any whitespace/newline `text()` nodes directly preceding them are also deleted.
- Any elements that (1) contain an `elements-to-delete` element as a descendant, (2) are contained by or are themselves a `delete-up-to` element, and (3) become empty due to the element deletion, are deleted.
  - To determine if a "containing" element becomes empty due to the deletion, `<xsl:apply-templates>` is called, then the results are checked to see if any elements remain. This is what allows the deletion to continue dynamically up through the containing elements.
- To conditionally keep the whitespace/newline `text()` node directly preceding a "containing" element,
  - A standalone unconditional template always deletes the whitespace/newline `text()` node preceding a containing element, whether it will be kept or not.
  - Inside the template that conditionally keeps containing elements, that same preceding `text()` node is re-included if the containing element is kept.

The following example shows a `<prolog>` element that disappears completely because it does not contain anything other than an `<indexterm>` element:

Before refactoring

```
<topic id="feature_A">
  <title>About Feature A</title>
  <prolog>
    <metadata>
      <keywords>
```

After refactoring

```
<topic id="feature_A">
  <title>About Feature A</title>
```

Before refactoring

```
        <indexterm>feature A</indexterm>
      </keywords>
    </metadata>
  </prolog>
```

After refactoring

The following example shows a `<prolog>` element that is partially kept because it also contains a `<resourceid>` element:

Before refactoring

```
<topic id="feature_A">
  <title>About Feature A</title>
  <prolog>
    <metadata>
      <keywords>
        <indexterm>feature A</indexterm>
      </keywords>
    </metadata>
    <resourceid id="feature_A"/>
  </prolog>
```

After refactoring

```
<topic id="feature_A">
  <title>About Feature A</title>
  <prolog>
    <resourceid id="feature_A"/>
  </prolog>
```

This same refactoring code can be adapted to other use cases by editing the `elements-to-delete` and `delete-up-to` variables as needed.

Resolving DITA Cross-References in Refactoring Operations

We needed a way to resolve DITA `<xref>` and `<link>` elements to their target element in a refactoring operation. We also needed to determine whether the link was a local or peer-map reference. Fortunately, Oxygen v25.1 provided us with the solution!

The Missing Piece - Resolving a @keyref

For `<xref>` and `<link>` elements, we use a mix of `@href` and `@keyref` references in our content. In XSLT refactoring operations, we could resolve `@href` references to the target file (and optionally, an element in that file) by using the XPath `document()` function. However, we had no way of resolving `@keyref` references because a refactoring operation only processes the current file (it doesn't consider Oxygen's map context or the keys defined within it). In addition, writing such a resolver in XSLT that properly considers keyscopes would be enormously difficult.

In the Oxygen v25.1 release, new API functions were introduced to provide information about `@keyref` references:

- The `getKeyRefInfo()` function returns information about a reference's type (local, peer, unresolved, and so on).
- The `getKeyRefAbsoluteReference()` function resolves a `@keyref` to its equivalent `@href` reference URL.

> **Note:**
>
> There were subsequent improvements to the API functions in Oxygen v25.1 build 2023070306, so you should use that release or later for best results.

Thanks to this new API, we could use a simple `@href` resolver written in XSLT for `@keyref` references too!

Resolving Cross-References in Refactoring Operations

The attached test case provides the following XSLT file that you can include in your own refactoring operations:

`frameworks/dita/refactoring/util-get-referenced-element.xsl`

This file defines a `mode="get-referenced-element"` template that, when applied to *any* element with an `@href` or `@keyref` attribute, returns the referenced element. If the reference cannot be resolved, the template returns an empty sequence.

The template works as follows:

1. References with `@scope="external"` always return an empty sequence.
2. References with `@format` set to a value other than `"dita"` always return an empty sequence.
3. If the reference has a `@keyref`, it is converted to an `@href` value using `getKeyRefAbsoluteReference()`.
4. The `@href` value is parsed into its components as follows:

   `[file]#topic_id[/element_id]`

5. The target document is obtained as follows:

- If no *file* is specified, the in-memory document that contains the cross-reference element is used.
- If a *file* is specified and that file contains the cross-reference element, the in-memory document that contains the cross-reference element is used.
- Otherwise, the specified file document is loaded from disk using the XPath `document()` function.

This heuristic approach ensures that in multiple-pass refactoring operations, the in-memory version of the content is preferred over the on-disk version.

6. The topic that matches the *topic_id* value is obtained from the target document.
7. If an *element_id* is specified, the element in the topic that matches the *element_id* value is obtained.

   Because non-topic `@id` values do not need to be unique, the code ensures that no subtopics within the matching topic are searched to avoid incorrect matches.

To view the XSLT stylesheet without downloading the archive, click on the following link:

util-get-referenced-element.xsl

There are comments in the code to explain how it works.

The `@keyref` API functions require that a map context be active in Oxygen. If no context is available (for example, no map is open in the **DITA Maps Manager**), there will be no key information available to resolve the reference.

Example Test Case

The following Oxygen project provides examples of how cross-reference resolutions can be used in refactoring operations and Schematron checks:

resolving_refs_refactoring.zip

Specifically, it provides the following:

- An "Update Cross References" refactoring operation is provided that:
  - Sets (or updates) the `@type` attribute for `<xref>` and `<link>` elements.

  - Populates the target text for `<xref>` and `<link>` elements that reference topics in peer maps (i.e. cross-book links).

- Schematron checks are provided that:
  - Warn about `<xref>` and `<link>` elements that reference topics in peer maps (i.e. cross-book links) but do not contain any target text.

    This check also offers a "quick fix" that populates the target text for you.

  - Show the value of the `getKeyRefInfo()` and `getKeyRefAbsoluteReference()` API calls for any element with a `@keyref` attribute.

These informational checks are commented out by default. You can uncomment them in the following file:

`frameworks/dita/sch/checks.sch`

When target text is added to a peer map (cross-book) reference, an `<?oxy-peertext?>` processing instruction is added to indicate that the text was updated automatically. If you remove this processing instruction and customize the target text, your customized text will not be disturbed by future automatic updates.

Inserting and Reformatting Content in Refactoring Operations

In Oxygen, refactoring operations are a powerful way to provide content modification functionality to users. This blog post describes some challenges that can occur when adding, changing, or deleting content. It also provides solutions that you can use in your own refactoring operations.

Re-indenting XML After Content Updates

In XSLT, all document content is represented by a node (elements, attributes, text, comments, and so on). In fact, even the whitespace text surrounding indented elements are explicit text nodes in the document.

Note:

In the figures below:

- Notable space characters are represented by the Unicode "U+2420: Symbol For Space" (␠) character.
- Notable line-end characters are represented by the Unicode "U+240A: Symbol For Line Feed" (␊) character.

When you delete an element node in a refactoring operation, any preceding or subsequent text nodes are left in place:

```
    <p>line1</p>#
    <p>line2</p>#
    <p>line3</p>#
```

```
    <p>line1</p>#
    #
    <p>line3</p>#
```

When you insert a new element node before or after an existing element node, no new indenting whitespace is added (unless the refactoring operation explicitly inserts it):

```
    <p>line1</p>#
    <p>line2</p>#
    <p>line3</p>#
```

```
    <p>line1</p><p>after</p>#
    <p>line2</p>#
    <p>before</p><p>line3</p>#
```

The test case below provides a `mode="indent-stuff"` template mode that looks for `@indent` attribute directives (they are removed as part of the reformatting process).

In the following example, the indent-reformatting template is applied around a `<p>` deletion:

```
<!-- TOP-LEVEL PASS - apply multiple processing passes to input document -->
<xsl:template match="/*" priority="100">
  <xsl:variable name="result" as="element()">
    <xsl:next-match/>  <!-- apply #default mode templates first -->
  </xsl:variable>
  <xsl:apply-templates select="$result" mode="indent-stuff"/>  <!-- process/remove @indent -->
</xsl:template>
```

```
   <p>line 1</p>
   <p indent="self">line 2</p>#
   <p indent="self">line 4</p>
   <p>line 5</p>
```

```
   <p>line 1</p>FE,(LF)
   <p>line 2</p>FE,(LF)
   <p>line 4</p>
   <p>line 5</p>
```

In the following example, the indent-reformatting template is applied to `<p>` insertions:

```
<!-- TOP-LEVEL PASS - apply multiple processing passes to input document -->
<xsl:template match="/*" priority="100">
  <xsl:variable name="result" as="element()">
    <xsl:next-match/>  <!-- apply #default mode templates first -->
  </xsl:variable>
  <xsl:apply-templates select="$result" mode="indent-stuff"/>  <!-- process/remove @indent -->
</xsl:template>
```

```
   <p>line 1</p>
   <p>line 2</p><p indent="self adjacent-siblings">after</p>
   <p>line 3</p>
   <p indent="self adjacent-siblings">before</p><p>line 4</p>
   <p>line 5</p>
```

```
   <p>line 1</p>
   <p>line 2</p>
   <p>after</p>
   <p>line 3</p>
   <p>before</p>
   <p>line 4</p>
   <p>line 5</p>
```

The `@indent` attribute contains a space-separated list that supports the following values:

- `self` - Reformat indenting whitespace before the current element (and after, for the last child of a parent).
- `adjacent-siblings` - Reformat indenting whitespace before the preceding and following sibling elements (and after, for the last child of a parent).
- `siblings` - Reformat whitespace around *all* sibling elements in the same parent element.
- `children` - Reformat whitespace around all child elements.
- `deep` - Reformat whitespace around all child and descendant elements.

The `@indent` attribute allows you to control the scope of content change so that the revision control change footprint can be controlled.

The `mode="indent-stuff"` template works as follows:

- Different files can have different indenting width conventions.
    - The template uses the most common indenting value (the *statistical mode*) in each file.
- Elements inserted in a block context (i.e. `<p>`) should be indented, but elements in an inline context (i.e. `<codeph>`) should not.
    - Some elements (i.e. `<indexterm>`) should be treated as block *or* inline elements based on their context (such as when they are in `<prolog>` or `<p>`).
- There might or might not be existing indenting whitespace around an element, and it might or might not be correct.

    - Indenting is added or repaired as needed.

- The last child element of a parent element requires one less indent level before its parent's closing tag.

To determine where indenting is appropriate, the stylesheet defines `inner-tag-type` and `outer-tag-type` XSLT3 maps that indicate the expected type of content at the inside and outside of each element's tags. Possible values are `block` (block elements), `text` (text content or inline elements), or an empty string (unknown or ambiguous content type). For example:

Element	`outer-tag-type`	`inner-tag-type`
`body`	`block`	`block`
`p`	`block`	`text`
`ph`	`text`	`text`
`menucascade`	`text`	`block`
`indexterm`	`''` (empty string)	`text`
`entry`	`block`	`''` (empty string)

For elements with unknown or ambiguous content types at the tag boundaries, the stylesheet applies heuristics to guess the indentation needs.

To view the XSLT stylesheet without downloading the test case, click on the following link:

util-indent.xsl

There are comments in the code to explain how it works.

Considering Content Models During Content Insertion

When inserting new content, the changes should not introduce any DITA content model (schema) violations. Some DITA content models require a specific element ordering. For example, the content model for `<prolog>` is:

`<prolog>` =

`<author>`*, `<source>`?, `<publisher>`?, `<copyright>`*, `<critdates>`?, `<permissions>`?, `<metadata>`*, `<resourceid>`*, (`<data>` | `<data-about>` | `<foreign>` | `<sort-as>` | `<unknown>`)*

Commas in the content model indicate an ordering requirement. For example, a `<resourceid>` element must be inserted after its preceding elements and before its following elements.

The test case below provides a `mode="insert-stuff"` template mode that inserts content at a specified element path inside the selected element.

In the following example, the content-insertion template is applied to a `<topic>` element to insert a `<resourceid>` element in a `<prolog>` element:

```
<!-- insert a <resourceid> into /topic/prolog -->
<xsl:template match="/topic">
  <xsl:apply-templates select="." mode="insert-stuff">
    <xsl:with-param name="path" select="'prolog'" as="xs:string*"/>
    <xsl:with-param name="content" as="element()*" tunnel="yes">
      <resourceid appid="topic-id" indent="self"/>
    </xsl:with-param>
  </xsl:apply-templates>
</xsl:template>
```

topic.dita (before)

```
<topic id="topic">
   <title>My Topic</title>
   <prolog>
      <author>chrispy</author>
      <data name="my-data"/>
   </prolog>
   <body/>
</topic>
```

topic.dita (after)

```
<topic id="topic">
   <title>My Topic</title>
   <prolog>
      <author>chrispy</author>FE,(LF)
%,%,%,%,%,%,<resourceid appid="topic-id"/>
      <data name="my-data"/>
   </prolog>
   <body/>
</topic>
```

In the following example, the content-insertion template is applied to a `<topic>` template to insert an `<indexterm>` element using a multiple-level insertion path:

```
<!-- insert an <indexterm> into /topic/prolog/metadata/keywords -->
<xsl:template match="/topic">
  <xsl:apply-templates select="." mode="insert-stuff">
    <xsl:with-param name="path" select="'prolog/metadata/keywords'" as="xs:string*"/>
    <xsl:with-param name="content" as="element()*" tunnel="yes">
      <indexterm indent="self">my-term</indexterm>
    </xsl:with-param>
  </xsl:apply-templates>
</xsl:template>
```

topic1.dita (before)

```
<topic id="topic1">
   <title>My Topic</title>
   <body/>
</topic>
```

topic1.dita (after)

```
<topic id="topic1">
   <title>My Topic</title>FE₃(LF)
⁘⁘⁘<prolog>FE₃(LF)
⁘⁘⁘⁘⁘⁘<metadata>FE₃(LF)
⁘⁘⁘⁘⁘⁘⁘⁘⁘<keywords>FE₃(LF)
⁘⁘⁘⁘⁘⁘⁘⁘⁘⁘⁘⁘<indexterm>my-term</indexterm>FE₃(LF)
⁘⁘⁘⁘⁘⁘⁘⁘</keywords>FE₃(LF)
⁘⁘⁘⁘⁘⁘</metadata>FE₃(LF)
⁘⁘⁘</prolog>
   <body/>
</topic>
```

topic2.dita (before)

```
<topic id="topic2">
   <title>My Topic</title>
   <prolog>
      <author>chrispy</author>
      <metadata>
         <audience type="user"/>
      </metadata>
      <data name="my-data"/>
   </prolog>
   <body/>
</topic>
```

topic2.dita (after)

```
<topic id="topic2">
   <title>My Topic</title>
   <prolog>
      <author>chrispy</author>
      <metadata>
         <audience type="user"/>FE₃(LF)
⁘⁘⁘⁘⁘⁘⁘⁘⁘<keywords>FE₃(LF)
⁘⁘⁘⁘⁘⁘⁘⁘⁘⁘⁘⁘<indexterm>my-term</indexterm>FE₃(LF)
⁘⁘⁘⁘⁘⁘⁘⁘⁘</keywords>
      </metadata>
      <data name="my-data"/>
   </prolog>
   <body/>
</topic>
```

The `mode="insert-stuff"` template works as follows:

- The `path` parameter can be a sequence of element name strings, a path string value using slash ("`/`") separators, or a mix of both.
    - An empty string or sequence value inserts the content directly into the applied element.
- New elements are inserted into existing elements using content model information.
    - The template creates or reuses intermediate element levels as needed, considering content models along the way.
- New elements created by a `@path` specification have a default `@indent` value of `self`.

You can override the default `@indent` value by specifying the `indent` parameter with the `mode="insert-stuff"` template. For example:

  - Setting `indent` to `"self adjacent-siblings"` reformats content more aggressively around newly created intermediate elements.
  - Setting `indent` to an empty string value disables reformatting for newly created elements.

- The insertion content itself is not indented unless it has explicitly defined `@indent` attributes.
- Only simple ordered-list content model modeling is supported.

To view the XSLT stylesheet without downloading the test case, click on the following link:

util-insert.xsl

There are comments in the code to explain how it works.

Example Test Case

The following Oxygen project provides the moded templates described earlier, along with an example refactoring operation that demonstrates their use:

refactoring_inserting_reformatting.zip

Specifically, it provides the following:

- `frameworks/dita/refactoring/util-indent.xsl`

  This stylesheet file defines the `mode="indent-stuff"` template.

  The `outer-tag-type` and `inner-tag-type XSLT3` maps are defined at the top of the file. You can update or add to these as needed.

- `frameworks/dita/refactoring/util-insert.xsl`

  This stylesheet file defines the `mode="insert-stuff"` template.

  The `content-models` map is defined at the top of the file. Only `<prolog>`-related content models are defined in this test case. You can add to these as needed.

- `frameworks/dita/refactoring/insert-by-content-model.xml`

`frameworks/dita/refactoring/insert-by-content-model.xsl`

This Oxygen refactoring operation allows you to interactively experiment with content insertion and reformatting. It includes and uses both of the stylesheets above.

To try out the example refactoring operation:

1. Open the `OPENME2.ditamap` file in the **DITA Maps Manager**.
2. Open "My Topic" (`topic.dita`) in the Oxygen editor.
3. Right-click in the topic editing window, then choose **Refactoring > Content insertion > Insert element by content model**.
4. For the configuration parameters, enter the following:

5. When you preview the refactoring operation, the results should be as follows:

Creating a Knowledge Base for the Google Dialogflow Chatbot using DITA XML content

A *Chatbot* is a computer program that simulates human conversation through voice commands or text chats or both. Chatbot (short for chatterbot), is an artificial intelligence (AI) feature that can be embedded and used through any major messaging application. *Chatbots* helps add convenience for customers, they are automated programs that interact with customers like a human would, and cost little to nothing to engage with.

Suppose you have a simple DITA topic that contains pairs of questions and answers placed as rows in a table:

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE topic PUBLIC "-//OASIS//DTD DITA Topic//EN" "topic.dtd">

<topic id="frequently_asked_questions">

    <title>Frequently Asked Questions</title>

    <body>

        <p>
```

```
            <table frame="all" rowsep="1" colsep="1" id="table_bkf_4dl_vsb">
                <tgroup cols="2">
                    <colspec colname="c1" colnum="1" colwidth="1*"/>
                    <colspec colname="c2" colnum="2" colwidth="1*"/>
                    <tbody>
                        <row>
                            <entry>
                                <p>What is Oxygen XML Editor?</p>
                                <p>What is Oxygen XML?</p>
                            </entry>
                            <entry>
                                <p>Oxygen XML Editor is a great tool.</p>
                            </entry>
                        </row>
                        <row>
                            <entry>How do I learn DITA?</entry>
                            <entry>You can read the "Oxygen" user's guide, and the Oxygen XML Blog.</entry>
                        </row>
                    </tbody>
                </tgroup>
            </table>
        </p>
    </body>
</topic>
```

We can use the following XSLT stylesheet to extract the training data as a CSV file from the XML content:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" exclude-result-prefixes="xs" version="2.0" xmlns:oxy="abc">
    <xsl:output method="text"/>
    <xsl:template match="/">
        <xsl:text>Question,Answer
</xsl:text>
        <xsl:for-each select="//tbody/row">
            <xsl:for-each select="entry[1]">
```

```
                <xsl:choose>
                    <xsl:when test="p">
                        <xsl:for-each select="p">
                            <xsl:value-of select="oxy:escapeForCSV(normalize-space(.))"/>,<xsl:value-of
                                select="oxy:escapeForCSV(normalize-space(string-join(../../entry[2]//text(),
 '')))"/><xsl:text>
</xsl:text>
                        </xsl:for-each>
                    </xsl:when>
                    <xsl:otherwise>
                        <xsl:value-of select="oxy:escapeForCSV(normalize-space(.))"/>,<xsl:value-of
                            select="oxy:escapeForCSV(normalize-space(string-join(../entry[2]//text(),
 '')))"/><xsl:text>
</xsl:text>
                    </xsl:otherwise>
                </xsl:choose>
            </xsl:for-each>
        </xsl:for-each>
    </xsl:template>
    <xsl:function name="oxy:escapeForCSV">
        <xsl:param name="value"/>
        <xsl:value-of select="replace(replace($value, '&quot;', '&quot;&quot;'), ',', '&quot;,&quot;')"/>
    </xsl:function>
</xsl:stylesheet>
```

Create a transformation scenario using these 2 files. Save the output to a CSV file, as in the picture below:

After applying the transformation, the following CSV file should appear:

```
What is Oxygen XML Editor?,Oxygen XML Editor is a great tool.

What is Oxygen XML?,Oxygen XML Editor is a great tool.

How do I learn DITA?,You can read the ""Oxygen"" user's guide"," and the Oxygen XML Blog.
```

Notice:

The structure of the CSV file contains a table with 2 columns, where the first column represents the question and the second column the answer, separated by a comma. In general, a line is in the form: "Question, Answer". One limitation is that each question can have only one answer. For the same answer to be suitable for several questions, you can copy it in the columns corresponding to the questions.

Next, follow these steps to use the data from the CSV file in the **Google Dialogflow** chatbot:

1. Create an account on https://dialogflow.cloud.google.com/.
2. Create a new agent and complete the required fields:

3. In Settings, enable **BETA FEATURES**.

4. Go to **Knowledge > CREATE KNOWLEDGE BASE**, enter a name for this knowledge base, and click the SAVE button.

5. Use the CSV document for training. For this: Press **Create the first one** and complete the fields. Select the **Update file from your computer** option and select the desired file. Click the **CREATE** button.

The document will be registered in the knowledge base, as in the image below:

6. Convert questions and answers into intents.

   Click the **View** link from the image above. Check all questions that need to be converted to intents. Click the **CONVERT TO INTENTS** button.

Now, when you click on **Intents**, the converted questions should appear:

7. Do a simulation. Click **Integration > Web demo**, and if the demo feature is disabled, enable it.

8. Integrate the **Chatbot** in your HTML Content. Go to the **Dialogflow Integrations** page, click the **Web Demo** integration and you will get an `<iframe>` HTML element which can be pasted in your HTML page. Using a **WebHelp** customization you can also integrate the Chatbot in the footer for each generated HTML page.

This was a basic way to show how a chatbot can be trained with intents and answers by using a DITA topic to produce a comma-separated value file.

Similarities Between Technical Doc Writers and Software Engineers

I'm a software engineer, working for Oxygen XML Editor along side my colleagues in developing tools used by technical documentation writers. I also do lots of technical support and sometimes I write small blog posts like this one or update our technical documentation.

Over time, I've come to view the aspects of a technical writer's job (especially a tech writer who is using standards that have lots of potential for reuse, like DITA XML) quite similar to many of the aspects of a software developer's job. So I started putting together a list of concepts and the way in which both of these jobs (which may seem incompatible with each other at first) would utilize such concepts.

Concept	Software Engineers	Tech Docs Writer
Accessibility	We build our software products by following accessibility best practices, we test our products using screen readers and collaborate with our vision-impaired users to fix accessibility problems. We also try to make sure there is enough color contrast in the application components and icons, that all font sizes are configurable and ship various color themes for the application.	Published technical documentation also needs to be accessible, for example by adding alternate text for images and using relevant text for links. It's also important to make sure the color theme of the published output has enough contrast. WebHelp and PDF outputs obtained using Oxygen from DITA XML content are section 508 compliant.
Accurate and Concise Messages	We are required to present accurate and concise error messages, dialogs, and prompts to our end users. Over time, we've learned to pass most of these error messages by our tech docs writers, asking their opinion to help improve them.	The technical documentation must also be accurate and concise. The technical writer's advice can also greatly benefit the software developers when they work in developing the messages in the application.
Balance between Time and Impact	Time is a precious commodity, as an engineer I need to make the best of the time I have allocated in order to implement a certain feature or functionality or to refactor content. If time does not allow me to perform a full redesign of a certain functionality I can still gradually improve it.	The tech docs writer may also not have enough time to refactor for example the entire project in a certain way but can gradually work towards a certain direction.
Building Complex Systems from Basic Building Blocks	We build small components (classes) and then we use them to build projects. We then use the many individual projects to build entire large products. We keep the small components as simple as possible to maximize their reuse potential. With these simple components, we build large complex projects.	Publications are built by using small topics of information or small reusable elements like notes or tables. Even larger publications can be built by combining the DITA maps used to describe subcomponents of the product. Individual topics/tasks are small, simple, and easy to follow. With these sim-

Concept	Software Engineers	Tech Docs Writer
		ple topics, you can build large technical documentation projects.
Code Review	We have a stage where the code that was written by someone is reviewed by a usually more experienced peer. The code reviewer may propose various changes, either small message changes or changes from the point of view of someone who knows the architecture of the project better. There is also the possibility for a novice to code review, as this exposes them to new techniques and they also can give a fresh perspective on things.	Having experienced technical writers review the work of novice writers is a great way to acclimate the novice writers to a specific way that the content must be written for a certain company. Experienced tech writers can also, for example, better determine if the content should be placed somewhere else in the project or if the current topic should be split into multiple ones as they have a better overall overview of the project.
Content Reuse	We reuse entire small projects in multiple products. We reuse libraries, classes, and functions. Content reuse gives us the ability to build products faster and to build different products from the same content.	Similarly, starting from the top down, entire DITA maps can be joined together to create documentation for larger publications, topics can be used in multiple places in a DITA map, and elements like notes, tables, lists can be reused in multiple places. Useful links: • DITA Reuse Strategies *(on page 150)* • DITA Style Guide
Copy/Paste vs Reuse	Sometimes I may decide that it's in the best interest of the project I'm working on to copy/paste instead of reusing. I may prefer copy/pasting for example in order not to introduce dependencies between modules or when using incompat-	It's best to reuse as much as possible but sometimes the writer may decide that copy/pasting and then maybe making small adjustments is a better decision.

Concept	Software Engineers	Tech Docs Writer
	ible data formats. Or if I plan to alter the content in certain ways. I may create auto tests to notify you when the source of truth has changed.	
Custom Validation Rules	We have various automatic tools (like **Sonar**) that report warnings when certain best practices for writing the code are not followed. This gives us more consistency as we are many devs working on the same code base and the code we produce needs to be easily read, understood, and maintained by others.	Using Schematron Rules *(on page 291)*, technical writers also can impose custom validation rules, specific for their project. These custom validation rules ensure consistency throughout the tech docs project.
Deleting Unused/Deprecated Content	All mature software products contain components that in time may become unmaintained or unused. Over time, such components may become security problems or may just contain inaccurate information. So, software developers also spend time deprecating and then removing these components.	Technical documentation also may contain content that is no longer accurate or topics that refer to features that no longer exist. So, deleting unused content is also a part of a tech writer's job.
Document my Code/Documentation	Most programming languages have support to add comments containing documentation about what the code does and why various choices were made. This documentation gives more information that could not be derived just by reading the code. We also have internal wikis with diagrams and descriptions of various architecture choices and steps to set up various projects for people who just started working on them.	It's a best practice to try Implementing your own Style Guide *(on page 294)* where various best practices and architectural decisions for the documentation project are described. Such a Style Guide makes on-boarding easier and over time, the rules in the style guide may start to be automatically applied using Schematron or a terminology checker. Publishing customizations should also be documented in a style guide and a quick getting started tutorial for people who start working on the spe-

Concept	Software Engineers	Tech Docs Writer
		cific technical documentation project may also be useful.
Enjoy	For the long term, it's good to find particular aspects about your job that you particularly enjoy doing. For example, in my case I've always been interested in investigating why certain bugs occur and following through, spending hours to investigate and solve them. I'm also very eager to delete unused code and functionality.	Whether it's rewriting/correcting problems in existing content, writing new content, adding custom validation rules, helping others, discussing with engineers, or improving the architecture of the existing technical documentation project, technical writers may also have preferred activities. In general, we get to know what each person does best within the team and who to ask for advice in certain cases.
Evolve	We need to constantly deepen our understanding of the current techniques we use, learn new techniques, and new frameworks or programming languages. Learning new things also helps us to obtain a different view of the current techniques we are using.	By reading various resources like the DITA Style Guide, the DITA 1.3 standard, or the Oxygen XML Blog *(on page 8)*, a technical writers can deepen their understanding of the used standard, what features it offers, and what best practices to follow. There are lots of useful resources about approaching technical documentation online like: Diátaxis. And various community places *(on page 69)* where one can ask for help or follow up on conversations. Sometimes, technical writers also become involved in styling the published HTML or PDF outputs, in uploading the deliverables to the proper places on the web site, producing training material, or receiving user feedback and integrating it back in the documentation. So, they

Concept	Software Engineers	Tech Docs Writer
		move towards scripting and being power-users.
Helping Yourself	When I do not know how to do something, I first try to figure things out on my one before asking for help. I use various internal resources (our internal wiki) or external resources like Google, ChatGPT, StackOverflow. I always tend to remember the solution better if I put some effort into finding it myself.	Overcoming obstacles by oneself is possible if the company has internal wikis or internal style guides that the technical writers can use. By helping ourselves, experienced writers develop a certain grit that allows them to overcome daily obstacles.
Inversion of Control	Most of the time, we use libraries developed by us or by others to build larger projects. So, our code uses other code. But there are frameworks like **Java Spring** that have this concept of **inversion of control**. With this concept, our code does not use libraries, but those libraries in turn orchestrate and use our code.	The DITA XML standard has the concept of key scopes *(on page 148)*, which allows key references and content key references in topics to have different values depending on where the topic is referenced in the DITA map. So in a way, your topics contain variable place holders that may expand to different content depending on where they are used in the DITA Map. The DITA XML standard also has this concept of content reference push *(on page 158)*. With conref push content can be placed in a topic from the outside. A DITA topic that contains key references is not truly instantiated until it's referenced in a certain place in the DITA map.
Issues/Tickets List	We work internally mostly with **Atlassian JIRA** to create issues/tickets. We have various workflows for our issues that allow us to have quality assurance,	An issues/tickets list is also very important for tech doc writers to keep track of what needs to be done.

Concept	Software Engineers	Tech Docs Writer
	and documentation connected to the same issue ID. The issue connects all the commits and modifications made both in the development phase and in the documentation phase. So, looking at an issue, I can also see links to all changes made to fix and document the issue. For some projects, we also work with GitHub issues or GitLab issues/tickets and although more simple, they also provide us with some basic way of keeping track of work progress.	
Learn by Teaching	Teaching is a great way to learn more about the concepts yourself. You do not need to be highly experienced in a subject matter in order to make small presentations in the team about it. I sometimes make small presentations to start discussions about the subject matter and gather best practices from other team members who now more about the subject matter than I do.	You can read about an aspect of a standard like **DITA XML** that you are not using in the company (for example, about branch filtering *(on page 146)* or **relationship tables**), make a small presentation in the team, and then try to start a discussion about how the feature could be used in the tech docs project and what problems it could solve.
Managing Dependencies/Linking between objects	The more dependencies a class, object, or project has, the more difficult is to use it in another context. In our work, we try to uncouple objects, to decrease the dependencies between them to be able to reuse the objects in multiple contexts.	There are various linking strategies *(on page 139)*, the main idea being to avoid direct references between topics and instead use indirect links and relationship tables. This gives you more flexibility to re-use topics in multiple contexts.
Metadata/Extensibility	From the point of view of a Java developer like myself, support for annotations was added to the Java programming language about 20 years ago and it al-	The DITA XML standard has various ways that users can enhance the publishing or add metadata that is useful for specific needs:

Concept	Software Engineers	Tech Docs Writer
	lowed the creation of various important libraries that in a way extend the language to offer all kinds of useful functionality.	• Using the `<metadata>` element and sub-elements to define key words, index terms, categories that can be used to add various publishing features. The Oxygen WebHelp output uses keywords to enhance the search functionality and index terms to create and index page. • The `@outputclass` attribute can be set on almost all DITA elements and allows you to easily customize elements using CSS in the published output. For example, you can use it to specify the language of codeblock elements. • The `<foreign>` element allows adding non DITA XML content and can be used for various enhancements like embedding HTML content directly in DITA topics. • The `<data>` element can be used almost anywhere to specify key value pairs that are later used in the published output. • The DITA XML standard itself can be specialized or restricted using a specific procedure. This possibility to extend the standard makes it much more valuable and allows targeting specific company needs.
No Appreciation for Fluff	The written code should be as compact and easy to read as possible. An appli-	Small topics describing key concepts, small tasks describing how various

Concept	Software Engineers	Tech Docs Writer
	cation's features should again be intuitive to use with a user's interface that does not present more than you need to accomplish a certain task.	things can be accomplished. Precise messages, simplified English, no fancy words, no thesaurus.
Obtaining multiple products from the same code base	Using configuration files, we can reuse an entire product code base to create multiple products, each with a specific subset of features.	Using profiling/filters or multiple main DITA maps, a DITA XML based project can be used to create user guides for multiple products *(on page 153)*.
One trick pony / Jack of all trades	Even if over time we specialize and become very productive with one set of tools and when working on a certain aspect of the application, having a larger overview and being able to use other tools/languages/frameworks when necessary gives us lots of flexibility to help members of other teams or to make changes to sub-projects that we are not directly involved in.	Being able to write documentation in multiple formats (Markdown, DITA XML, MS Word), being able to use various version control systems like Git or content management systems gives you the ability to apply for and be able to perform any technical documentation job, besides the benefit of using much of the workflows and tools other people in the company might use.
Overview of Complex Systems	Either when making a small improvement or implementing a large feature, having an overview of the entire project's architecture helps a lot.	Having an overview of the documentation project helps you know where to make a change, what pleasant or unpleasant side effects it generates and how to possibly re-structure the content to facilitate making that change. Content reuse adds a lot of efficiency to the process of building a user's guide and reusing large chunks of content between user manuals. Content reuse unfortunately also sometimes makes it easier to propagate errors in multiple places.
Pair Programming	We have this concept that we apply from time to time in which two people work on the same issue from the same computer. These people can be peers with	I'm not sure if in the domain of tech docs writing people use such pair docs writing but I think it could also benefit a novice to see exactly how people with

Concept	Software Engineers	Tech Docs Writer
	the same knowledge, in which case one of them thinks about the larger context in which the issue is resolved and asks questions while the other implements it. Or the engineers can be an expert and a novice, in which case the novice directly sees how the expert considers and implements the issue.	more experience organize themselves, understand the requirements, and work on issues.
Project Structure and Naming Resources	Experienced software developers are interested in properly naming things. We try to give good names to functions, variables, and packages. We try to be consistent when naming things using compound names, either using hyphens or upper-lower-case sequences.	It makes sense for technical documentation writers to also have best practices when it comes to a project's structure, how files should be named, where images should be stored, how key names should be given, and so on. Sample Project.
Publishing Orchestration	We have various automated flows that generate nightly build installation kits for our product that can be further tested by the QA team.	Although the published output can be obtained from an Oxygen installation by running a transformation scenario, over time it's a good idea to automate the publishing by using an integration server like Jenkins, for example, to constantly poll the user guide project for modifications and update an internal web site with the latest HTML or PDF outputs to show how the documentation is evolving and give material to engineers who want to review the published content. For the Oxygen XML blog, for example, a Gradle build file *(on page 10)* is used to automate the publishing.
Read the Docs	We learn a lot by doing, or by copy pasting or reusing content from other similar projects that we have. But sometimes taking the time to read the documen-	By reading the DITA 1.3 standard documentation, for example, a technical documentation writer might find various features that can be used in their project.

Concept	Software Engineers	Tech Docs Writer
	tation for the frameworks and libraries that we use is really worth it.	They might also understand certain limitations in the way things work. There are lots of resources *(on page 66)* for learning DITA or for learning how to better use Oxygen to edit DITA XML content. So, it is sometimes worth taking the time to explore the tools you are using and the capabilities of the standard.
Requesting Help from Peers	Besides an internal wiki of content and procedures, we rely on a network of colleagues, some from our teams, some from others or from the system admins team. Over time, we come to know each other's strengths and know who to call when we need advice. Helping someone else usually takes priority over the current task.	Besides asking their peers for help, technical documentation writers also have various engineers and subject matter experts who could help when writing content either by giving feedback or by providing some initial content (some notes for example) to be used for the official documentation.
Security	We do our best to protect our applications from security vulnerabilities by updating old libraries and by considering various scenarios in which attackers would attempt to gain control over an application.	Besides keeping the user guide contents in a secure location to avoid exposing information to the competition or features that have not yet been released, the HTML output obtained from the user guide contents also needs to be updated to contain the latest Javascript libraries and not be susceptible to vulnerabilities when loaded in a web browser. Content in the user's guide in general also needs to be considered from the point of view of not giving an attacker details about how the product could be configured for use in other scenarios

Concept	Software Engineers	Tech Docs Writer
		other than the ones intended by the developers.
Server Side Validation	Validation and automated tests run on the server side (we use a Jenkins integration server) whenever changes are made to source code for a certain project. If automatic tests fail, emails are sent to the people who have committed the latest changes that broke the builds.	Whenever you have multiple topic files, filters, or content reuse, you may end up with potential validation problems in the edited content, problems ranging from broken links to invalid ID attribute values or invalid topics. Oxygen has a very useful **Validate and check for completeness** action that checks lots of potential problems in the context of a DITA map, but maybe not all people run such an action before they commit. Such an action can also be scripted to run on the server side in an integration server like **Jenkins**, for example.
Terminology	There are lots of ways that a message can be conveyed. All messages presented to the end user need to have a certain style, tone, and use plain language.	Controlled terminology *(on page 458)* is also an important aspect of a technical writer's job.
Translation	In general, all user interface strings in an application will be translated to multiple languages so we need to take this into account when designing the application.	A DITA XML project that needs to be translated into multiple languages also needs to follow a sequence of best practices *(on page 287)*.
User's Point of View	All features need to be discussed and implemented from the point of view of the people who will actually benefit from using them. For technical people, this is sometimes difficult to do but we try to constantly remind ourselves to implement features that help certain real-world workflows to be performed	The documentation also needs to target certain user profiles and be written to help people perform tasks and understand concepts. The product knowledge level of the end user is almost never the same as that of the tech writer.

Concept	Software Engineers	Tech Docs Writer
	by the end user and not for the sake of closing tickets.	
User Feedback	It's important to get feedback from end users. This feedback gives us ideas about what to implement in the future and also allows us to modify existing features to better accommodate certain workflows. So, we try to make it as easy as possible to give feedback, we have a **Report problem** action directly in the application, we have a support email address, a forum, user's list, and on each of our user's manual pages, there is also a feedback comments section.	Receiving user feedback for the written documentation is also very important to improve the documentation to cover the actual user needs. For example, the Oxygen Feedback platform can be easily embedded in the generated web pages to allow users to provide feedback related to the documentation. We use it for the Oxygen user's manual, blog, and it's a very good mechanism to get in touch with users and to improve the docs.
User Interface	An application may work very well from the functionality point of view but if it does not look very nice (ugly color theme, ugly icons, not intuitive user interface layouts) it will not end up getting used very much.	A published user's guide also needs to have appropriate colors and fonts to improve readability and find ability. So, besides the actual content care, decisions must also be made in terms of what font styles and sizes to use, or what color theme to implement.
Version Control	We use a mix of Git repositories and Subversion repositories for our software projects. Version control allows us to collaborate, to have a history of changes, and to have release branches for our products.	Using a version control system like Git similarly gives the possibility to collaborate on documentation projects with various workflows *(on page 375)*. As an alternative, you can use a commercial content management system or in the cloud editing solution. No matter what storage system you use to collaborate, having a history of changes and release branches is a must.

Concept	Software Engineers	Tech Docs Writer
Working with AI tools	Developers can work with ChatGPT to ask advice about how to write various pieces of code, especially in programming languages that they are not familiar with. They can also ask for help re-writing messages or translating small pieces of content.	Working directly with AI tools or using an add-on like the Oxygen AI Positron Assistant view *(on page 16)*, tech doc writers can also use the AI tools to improve readability, remove grammar problems, re-write content in various styles, or translate content. So, AI tools are valuable for writers as co-pilots.

Writers and software engineers have lots of things in common because we are all working with language and we communicate in a specific language. Computer programming languages contain less words and are more simple than the English vocabulary, but in the end we are all in the market of describing, expressing, and building large systems using simple terms.

If you have feedback about these concepts and how they influence your writing, it is as usual, welcomed.

Chapter 5. Migrate

Oxygen XML Editor provides various facilities to migrate content from one format to another.

Migrating to a Structured Standards-based Documentation Solution

Potential clients come to this world of structured content authoring from two main sources:

1. They are starting fresh and after a little bit of comparing between structured and unstructured editing, between opened and closed solutions and some soul searching they come to regard structured authoring with a specific XML standard in general (and usually **DITA** in particular) as the possible solution for them.
2. They are migrating from a previous unstructured or structured solution.

I think people in this second category start thinking about structured writing when they start encountering certain limitations with their current approach. These limitations they experience with their current system could be:

- The need to reuse more content.

  With structured **XML** authoring in general and with **DITA** in particular you have so many ways of reusing content. In a previous blog post I tried to come up with an overview about all the ways in which you can reuse content using **DITA**: DITA Reuse Strategies *(on page 150)*

- Produce multiple outputs from the same content using some complex profiling conditions which are not supported in the current work flow.

- Stop thinking about how the content is styled.

  You may want to focus more on the actual content and on semantically tagging it than on the way in which it will be presented in a certain output format.

- Publish to more output formats than the current editing solution allows.

  Using a widely adopted open source standard like **DITA** for documentation also means having access to a variety of commercial and open source tools to generate various output formats from it. For example for obtaining the **PDF** you have about 5-6 distinct possible solutions:Possibilities to obtain PDF from DITA *(on page 191)*.

- Enforce certain internal rules on the documents.

It's hard to impose best practices in unstructured documents. But with structured XML content, you can use Schematron to easily cover this aspect and even to provide quick fixes for your authors: Schematron Checks to help Technical Writing *(on page 291)*.

- Benefit of advice and help from a larger community of writers and developers.

  When you are using a closed source solution, you may have only one forum and a couple of people willing to help. When you have a larger community you will be able to reach out with a single email to lots of people, and somebody may want to help you.

- Share documentation between different companies.

  If a larger company which uses structured writing takes over a smaller one, the smaller company will need to adopt structured writing as well.

- Own your content.

  Some editing solutions are closed source, you are forced to use a single tool because there are no other tools being to read that format. Then you need to ask yourself the question: *"Is this content actually mine?"*

- Problems with your current tool vendor.

  If the format is closed source and the tool vendor is not responsive to your needs, you need to somehow move your content over to a market with multiple tool vendors available because competition also means smaller prices and better customer support.

Switching to structured content writing also has its problems. And I think the main ones are these:

- The people. The fact that we all are reluctant to change. The learning curve. Writers might need to re-learn how to structure and write their documentation. Besides the technical aspects they will need to learn to divide content in small modules and to reuse parts in multiple places. Writers may not be willing to do this. We usually are very reluctant to change tools if we do not see instant benefits deriving from it.
- Effort to convert the current available content to structured content. You can either choose manual conversion or automated conversion or in most cases a mixture of the two. Conversion will never be perfect, you will still need to go through the entire content and re-structure it taking into account module-based editing.
- Customize the obtained output format. You may get out of the box various outputs from your content but you will always need to customize it to adhere to company standards. If you are using

the **DITA Open Toolkit** for publishing you will need basic XSLT development skills to customize the **PDF** and **CSS** skills to customize the **XHTML** based output.
- Money. You need to spend more money to get new tools, possibly a new **CMS**. Although I consider that starters, for a pilot project **DITA** does not need to be expensive. Here's how we're using **DITA** internally for our user's manual: Collaboration for Documenting a Software Product using DITA *(on page 465)*.
- Sometimes you might need to control the styling of your obtained output so much and it would be impossible to separate the styling information from the content.

So can we draw a conclusion from all this?

Well, maybe not everybody interested in structured authoring will succeed to convert to it. But I think that one thing will hold true in most cases:

Once you convert to structured content, you will never go back.

Migrating Various Document Formats to DITA

Most companies do not start new DITA-based projects from scratch. They already have content written in various other formats and somehow they need that content converted to DITA. In this blog post, I will offer some conversion advice depending on the format of your current project.

Migrating DocBook Content to DITA.

You can migrate one or multiple DocBook documents to DITA using the **Oxygen Batch Documents Converter** add-on: https://www.oxygenxml.com/doc/ug-addons/topics/batch-converter-addon.html.

The **DocBook to DITA** conversion contains an option named **Create DITA maps from DocBook documents containing multiple sections**. When this option is selected, all sections from your DocBook document will be separated into individual DITA topics and referenced in a DITA map.

Migrating Microsoft Word Content to DITA

The Oxygen XML User Manual has a detailed topic enumerating the possibilities to convert Microsoft Word content to DITA: https://www.oxygenxml.com/doc/ug-editor/topics/ooxml-to-dita.html.

Migrating Excel Content to DITA

You can use Oxygen's *Smart Paste* functionality to copy content from an Excel spreadsheet and paste it inside an opened DITA topic. Also, as an alternate possibility, the **Oxygen Batch Documents Converter**

add-on was updated to be able to batch convert Excel to DITA: https://www.oxygenxml.com/doc/ug-addons/topics/batch-converter-addon.html.

Migrating LibreOffice Content to DITA

LibreOffice documents can be saved in Word format, and once you do that, you can convert the Word content to DITA as described above. Alternatively, you can save the LibreOffice documents to DocBook and then apply the DocBook to DITA conversion technique described above.

Migrating Google Docs to DITA

You have three possibilities to convert Google Docs to DITA using Oxygen:

- Copy/Pasting from Google Docs to a DITA Topic opened in Oxygen in the Author visual editing mode should work and convert the pasted content to DITA.
- Save the Google document as OpenDocumentFormat (**ODF**) then save the ODF document as **DocBook** with **Libre Office**, then apply the **DocBook to DITA** transformation scenario shipped in Oxygen to convert DocBook to DITA.
- Save the Google document as HTML then use the Oxygen batch converter add-on to convert it to DITA: https://www.oxygenxml.com/doc/ug-addons/topics/batch-converter-addon.html.

Migrating Markdown Content to DITA

The DITA Open Toolkit publishing engine bundled with Oxygen allows you to reference Markdown files directly in a DITA map and either publish them directly or export the Markdown files to DITA one by one: https://www.oxygenxml.com/doc/ug-editor/topics/markdown-dita-2.html. If you want to convert multiple Markdown documents at once, you can use the **Oxygen Batch Documents Converter** add-on: https://www.oxygenxml.com/doc/ug-addons/topics/batch-converter-addon.html.

Migrating HTML Content to DITA

Using Oxygen's *Smart Paste* functionality, you can open the HTML documents in a web browser, then copy the contents and paste it in a DITA topic opened in Oxygen's **Author** visual editing mode. If you want to convert multiple HTML files, you can use the **Oxygen Batch Documents Converter** add-on: https://www.oxygenxml.com/doc/ug-addons/topics/batch-converter-addon.html.

Migrating Unstructured FrameMaker to DITA

There is a detailed blog post enumerating the possibilities to convert Unstructured FrameMaker content to DITA: Migrating Unstructured Adobe FrameMaker Content to DITA *(on page 368)*.

Migrating MadCap Content to DITA

This open source project contains such a stylesheet which attempts to convert a Flare project to DITA XML and instructions about how to use it. As an alternative some recent MadCap versions seem to have facilities to export content directly to DITA.

Migrating Confluence Content to DITA

To convert Confluence content to DITA, you can use the **Oxygen Batch Documents Converter** add-on: https://www.oxygenxml.com/doc/ug-addons/topics/batch-converter-addon.html.

You first need to export the content to HTML. For this, log in to your Confluence account and navigate to the specific space that you want to export. Then go to **Space Settings→Export space** and choose to export it as HTML. Then, back on **Oxygen**, you can then use the **Confluence to DITA** action (available once the add-on is installed) to convert the exported `index.html` file into a DITA map with topics.

Migrating AsciiDoc to DITA

The Asciidoctor third-party application can be used to convert AsciiDoc content to DocBook. Then, you can convert the DocBook content to DITA using the method described here *(on page 354)*.

Migrating reStructuredText to DITA

The Pandoc third-party application can be used to convert reStructuredText content to DocBook or HTML. Then, you can convert the DocBook or HTML content to DITA using the Oxygen Batch Documents Converter add-on.

Migrating LaTex to DITA

You may use a third-party application (like Pandoc) to convert LaTex content to Word or HTML. Afterwards use the **Oxygen Batch Documents Converter**: https://www.oxygenxml.com/doc/ug-addons/topics/batch-converter-addon.html

Migrating Other Formats to DITA

You may find third-party applications (like Pandoc) that can convert your content to HTML or to some kind of XML format like DocBook. Once you have HTML or DocBook content, you can convert them to DITA using the advice above.

How to Migrate from Word to DITA

The need for migrating Microsoft Office® Word documents to XML formats, and particularly to DITA, is quite a frequently encountered situation. As usual, migration from proprietary formats to XML is never perfect and manual changes need to be made to the converted content. However, the methods below should help you find the best approach for your particular case:

Oxygen Batch Documents Converter add-on

The Oxygen Batch Documents Converter add-on can be installed in Oxygen and it allows you to convert one or more documents to various formats.

More details about the main stages of the Word to DITA migration using the Batch Documents Converter add-on: Migrating MS Word to DITA using the Batch Documents Converter *(on page 358)*

> **Note:**
> The Batch Documents Converter add-on is the recommended way to convert one or multiple Word documents to DITA content.

Smart Paste

1. Open the Word document in MS Office, select all the content, and copy it.
2. Open **Oxygen** and create a new **DITA** topic in the Author visual editing mode.
3. Paste the selected content. Oxygen's smart paste functionality *(on page 422)* will attempt to convert the content to DITA.

Word to HTML to DITA

1. Save your MS Office Word document as HTML.
2. Once you obtain that HTML, you have two possibilities:
    - In **Oxygen**, Select **File**->**Import**->**HTML File** to import the HTML as XHTML. Then open the XHTML in Oxygen and in the "**Transformation Scenarios**" view there should be four pre-configured transformation scenarios to convert **XHTML** to **DITA** topics, tasks, references, or concepts.
    - Open the HTML file in any Web browser, select all of its content, and copy it. Then open **Oxygen**, create a new **DITA** topic in the Author visual editing mode, and paste the selected content. Oxygen's smart paste functionality *(on page 422)* will attempt to convert the HTML to DITA.

Word to DocBook to DITA

1. Open the Word document in the free **Libre Office** application and save it as **DocBook**.
2. Open the **DocBook** document in **Oxygen**.
3. Run the predefined transformation scenario called **DocBook to DITA**.

Word to DITA using DITA For Publishers

1. If the Word document is in the new DOCX format you can open it in **Oxygen**'s **Archive Browser** view and then open the **document.xml** file contained in the archive.
2. Run the predefined transformation scenario called **DOCX DITA**. This **ANT** scenario runs the following build file: OXYGEN_INSTALL_DIR/frameworks/dita/DITA-OT/plugins/net.sourceforge.dita4publishers.word2dita/build-word2dita.xml over the **DOCX** archive and should produce a **DITA** project that contains a DITA map and multiple topics.
3. You may need to do some reconfiguring to map DOCX styles to DITA content.

Note: This method may also be helpful if you want to run it automatically with scripts, since it is based on the **DITA OT** and **Dita For Publishers** plugins.

Migrating MS Word to DITA using the Batch Documents Converter

This blog post presents how you can migrate from MS Word to DITA using the Oxygen Batch Documents Converter add-on.

The migration process has three main stages:

1. Preparing Word Document for Migration *(on page 359)*
2. Converting the Word Document *(on page 360)*
3. Post-processing the DITA Content Converted from Word *(on page 362)*

A webinar presenting this migration is available here: Working with DITA in Oxygen - Migrating to DITA and Refactoring

A list of frequently asked questions about the migration process can be found here: Word to DITA Conversion FAQ *(on page 363)*

Preparing Word Document for Migration

The Word to DITA conversion uses the styles found in the Word document for creating structure and formatting the text into the output document. The converter ignores the font, size or color set on text content. To achieve the best results, the Word document has to be *well styled*.

This preparation step assumes correcting various cases when formatting of the content was not made using styles. These are the things that you should take into account when checking the Word document:

1. It's important to make sure that titles and headings were formatted using styles because the topics and sections structure of the output document results from these.

   This is an example of bad formatting: Instead of using the "**Heading 1**" style, when formatting a title paragraph, the writer changed its font, size and color, keeping the default "**Normal**" style:

2. For the rest of the content, you can try to use the default Word styles for creating semantics where it's possible.

   This is an example of bad formatting: Instead of using the "**Quote**" style, when formatting a paragraph, the writer added it between quotes characters and changed its font and alignment keeping the default "**Normal**" style:

   Converting the paragraphs containing the "Normal" style will result in DITA p elements. So if you omit this step, you also have the option of adding semantics by post-processing the resulting DITA content.

> **Note:**
> The text content formatted using the bold, italic, underline, strike-through, subscript or superscript actions are handled by the converter. So you don't need to replace them with correspondent styles like "Strong" and "Emphasis".

3. Save the shapes as SVG images because the converter doesn't support them. See the following link for more information: How can I preserve the Word shapes? *(on page 366)*

Converting the Word Document

For converting the Word document you have to install the Batch Documents Converter add-on in Oxygen. For this, you can follow the installation procedure from the documentation.

Configuring Word styles mapping

The Word to DITA conversion has two main steps: Word to HTML to DITA. The converter allows customising the first step by setting a mapping between Word styles and HTML elements. It already contains a default mapping configuration and because of this, it knows to handle the default styles and the custom styles that are based on the default styles.

Here you have more information about the **Word Styles Mapping** option: Conversions from Word

This is an example of styles configuration: How can I fix unrecognized style warnings? *(on page 363)*

Configuring the Maximum Heading Level for Creating Topics

The converter has an option named "**Maximum Heading Level for Creating Topics**" in the Oxygen **Preferences > Batch Documents Converter** preferences page that allows setting a maximum heading level to process as a DITA topic. The headings with a higher nesting level will be converted to `<section>` elements.

This option can correspond with the value of the **Show levels** option that you configure in MS Word for generating a table of contents.

For example, consider a document with the following headings hierarchy in the Word document:

Contents

Introduction (Heading 1)
  Pruning (Heading 2)
  Garden Preparation (Heading 2)
Flowers by Season (Heading 1)
  Autumn Flowers (Heading 2)
    Salvia (Heading 3)
      Species (Heading 4)
      Characteristics (Heading 4)
Glossary (Heading 1)
Copyright (Heading 1)

if you keep the default **5** value for the **Maximum Heading Level for Creating Topics** option, all Word sections will be converted to DITA topics. If you set the option to **3**, the "**Species**" and "**Characteristics**" Word sections will be converted to DITA sections inside the "**Salvia**" DITA topic.

Converting the document

You can invoke the **Word to DITA** action from the following places for converting the Word documents:

- **Batch Documents Converter** from the **Tools** menu.
- **Batch Documents Converter** in the contextual menu from the **Project** view.
- **Additional conversions from** the **FileImport** menu.
- **Import** from the **Append child**, **Insert Before**, or **Insert After** sub-menus in the contextual menu from the **DITA Maps Manager** view. Note that these actions automatically insert references to the resulting documents into the DITA map.

The Word to DITA conversion dialog contains an option named **Create DITA Maps from Word documents containing multiple headings** that allows us to choose if the output will be a single **DITA Topic** or a **DITA Map**. When the option is checked, the resulting DITA topics will be extracted into separate files and referenced into an output DITA Map, otherwise, the topics will be grouped into a DITA Composite file. The resulting topics can be controlled using the Maximum Heading Level for Creating Topics *(on page 360)* option.

Post-processing the DITA Content Converted from Word

Oxygen provides refactoring actions that can be used for **organizing** and **customizing** the resulting DITA documents, to take into account various DITA best practices.

These are some of the most used refactoring operations that can be applied after the conversion process:

- In the **Project** view, the **Move Resource** and **Rename Resource** actions can be applied to DITA or non-DITA resources: https://www.oxygenxml.com/doc/ug-editor/topics/dita-main-files.html#dita-main-files__moving_or_renaming_nondita_resources
- In the **DITA Maps Manager** view you can arrange the topics and nodes using the **drag and drop** support: https://www.oxygenxml.com/doc/ug-editor/topics/dita-maps-manager.html#dita-maps-manager__drag_and_drop_in_the_dita_maps_manager
- There are refactoring actions for converting between DITA topic types. So, if the resulting topics look more like **DITA Tasks** or **DITA Concepts**, you can simply convert them using refactoring actions like **Convert to Task**, **Convert to Concept** and others.
- There are actions for generating unique IDs for elements (**Generate IDs**) or changing the ids of the topics using the file names (**Change topic ID to file name**).
- When resulting documents contain *nested topics* or *sections*, you can use **Convert Nested Topics to New Topics** or **Convert Sections to New Topics** to extract new files.
- Topics can be extracted from the map hierarchy into a new DITA map, using the **Extract to new DITA Map** action.
- The **Convert DITA Map to Bookmap** and **Convert Bookmap to DITA Map** actions allow converting between map and bookmap types.
- For converting links inside your documents from *direct addressing* to *indirect key-based addressing*, you can use the **Define keys for all topic references** action.
- There are also operations for modifying elements inside documents: https://www.oxygenxml.com/doc/ug-editor/topics/predefined-refactoring-operations-x-tools.html#predefined-refactoring-operations__refactoring_operations_for_elements

In the main menu **Tools > XML Refactoring** dialog you can find the list of all built-in refactoring operations: https://www.oxygenxml.com/doc/ug-editor/topics/predefined-refactoring-operations-x-tools.html

If you want to make custom batch changes to the converted documents you can create a custom refactoring operation: https://www.oxygenxml.com/doc/ug-editor/topics/custom-refactoring-operations-x-tools.html.

Example of a custom refactoring operation *(on page 364)*.

Word to DITA Conversion FAQ

How can I fix unrecognized style warnings?

When converting a Word document, the styles that don't have a mapping in the **Word styles mapping** table from the preferences page are converted to simple paragraph elements and a warning is reported for each of them in the **Results** view.

As an example, suppose I have converted the Word document and I see the following warnings in the Results view:

Results

	Info	Description - 3 items	Resource	System ID
	–	W Unrecognized "Document Title" style for "p" Word element. You can configure the mapping between styles and elements in the Batch Documents Converter preferences page.	sample.docx	C:\Users\cosmin_duna\Desktop\sample.docx
	–	W Unrecognized "Keyboard Key" style for "r" Word element. You can configure the mapping between styles and elements in the Batch Documents Converter preferences page.	sample.docx	C:\Users\cosmin_duna\Desktop\sample.docx
	–	W Unrecognized "Keyboard Key" style for "r" Word element. You can configure the mapping between styles and elements in the Batch Documents Converter preferences page.	sample.docx	C:\Users\cosmin_duna\Desktop\sample.docx

Here are the steps that you should follow for this configuration:

1. Open the **Plugins / Batch Documents Converter** preferences page in Oxygen.
2. To fix the `Unrecognized "Document Title" style for "p" Word element` warning, add a new row in the **Word styles mapping** table with the following cells:
    a. Type "p" in the **Word element** cell (because the unrecognized style was found on a Word paragraph).
    b. Type "Document Title" in the **Word style** cell.
    c. In the "HTML elements" cell, you need to add a corresponding HTML element. For this example, a corresponding element is "`<h1>`" for the default mapping of the "Title" style. So, type "*h1:fresh*" into this cell. The ":fresh" suffix instructs the converter to create a new element every time it finds this kind of paragraph. When it's not set, the converter will try to reuse the elements and combine sequences of the same style of paragraphs into a single element.
3. To fix the `Unrecognized "Keyboard Key" style for "r" Word element` warning, add a new row in the **Word styles mapping** table with the following cells:
    a. Type "r" in the **Word element** cell (because the unrecognized style was found on a Word character).
    b. Type "Keyboard Key" in the **Word style** cell.
    c. In the **HTML elements** cell, you need to add a corresponding HTML element. For this example, a corresponding element is "`<kbd>`" for the default mapping of the "HTML Keyboard" style. So, type "*kbd*" in this cell.

After completing these steps, you should have these two rows in the configuration table:

p	Document Title	h1:fresh
r	Keyboard Key	kbd

For more information about the **Word styles mapping** configuration, see the following section from the documentation: Conversions from Word.

How can I configure the styles mapping when the wanted element doesn't exist in HTML?

A frequent case when setting the mapping configuration for a custom style is to not find a correspondent element in HTML, although one exists in DITA.

As an example, suppose I have a Word document that contains a character custom style named "filepath". We know that a correspondent element exists in DITA, but we cannot find one in HTML.

These are the steps that can be applied to handle this case:

1. Go to the **Plugins / Batch Documents Converter** preferences page and add the following mapping in the **Word styles mapping** table:

r	Filepath	i.filepath

2. Convert the Word document to DITA. The characters styled with the *Filepath* style in Word are converted to the `<i>` element with the "*filepath*" `@outputclass` attribute.
3. Move the output files to your project, select them, and apply the "**Rename element**" refactoring operation using the "*//i[@outputclass = 'filepath']*" XPath for matching the target `<i>` element and changing them to the `<filepath>` DITA element.
4. Apply the "Remove attribute" refactoring operation using the "*//filepath[@outputclass = 'filepath']*" XPath for matching the target elements and deleting the `@outputclass` attributes.

Instead of steps 3 and 4, you can also create a custom refactoring operation that makes these two changes, like this:

1. Create an XSLT file (for example, named `batch-converter-post-processing.xsl`) that iterates over all elements from the document, finds the `<i>` elements with the "filepath" `@outputclass` attribute, and replaces them with the `<filepath>` elements without copying the `@outputclass` attribute:

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
    xmlns:xs="http://www.w3.org/2001/XMLSchema" exclude-result-prefixes="xs" version="2.0">


    <xsl:template match="@* | node()">
        <xsl:copy>
            <xsl:apply-templates select="@* | node()"/>
        </xsl:copy>
    </xsl:template>


    <xsl:template match="i[@outputclass = 'filepath']">
        <xsl:element name="filepath">
            <xsl:apply-templates select="node()"/>
        </xsl:element>
    </xsl:template>
</xsl:stylesheet>
```

2. Create an XML Refactoring operation descriptor (for example, named `batch-converter-post-processing.xml`) that references the stylesheet and provides descriptions:

```
<?xml version="1.0" encoding="UTF-8"?>
<refactoringOperationDescriptor xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.oxygenxml.com/ns/xmlRefactoring" id="op_qzq_y2x_nsb"
    name="Post-processing Batch Documents Converter">
    <description>Post-process the resulting DITA documents from the Word conversion using the Batch Documents
 Converter add-on.</description>
    <script type="XSLT" href="batch-converter-post-processing.xsl"/>
</refactoringOperationDescriptor>
```

3. Copy these two files to a folder scanned by Oxygen XML Editor when it loads the custom operation.
4. Apply the new custom operation named "*Post-processing Batch Documents Converter*" that can now be found in the list of refactoring operations from the "XML Refactoring" dialog box.

How can I configure the styles mapping for paragraphs styled as code blocks?

In Word, a code block is represented as a sequence of paragraphs styled with a custom style that adds a custom font and border, like this:

```
/**¶
*·Get·access·to·API·used·to·control·the·Oxygen·editors.¶
*·This·static·access·can·be·used·when·running·the·standalone·or·the·Eclipse·version·of·Oxygen·
from·any·part·of·the·developer's·code.¶
*·¶
*·@return·Returns·the·pluginWorkspace.¶
*/¶
public·static·PluginWorkspace·getPluginWorkspace()·{¶
··return·(PluginWorkspace)StaticComponentsRegistry.get(StaticComponentKey.PLUGIN_WORKSPACE);¶
}¶
```

To add a mapping for this custom style, follow these steps:

1. Open the **Options** → **Preferences** → **Plugins** → **Batch Documents Converter** preferences page in Oxygen.
2. Add a new row to the **Word styles mapping** table.
3. Enter "*p*" in the **Word element** cell, and enter "*Code Snippet*" in the **Word style** cell to match the paragraphs styled with "Code Snippet".
4. Enter "*pre:separator('\n')*" in the **HTML elements** cell. The `<pre>` element is the corresponding HTML element for these types of paragraphs. Since we want to merge these sequences of "Code Snippet" style paragraphs, the ":fresh" marker was not used. The "*:separator('')*" syntax configures a separator when the same type of consecutive paragraphs are merged. If it isn't specified, we would obtain a `<pre>` element with a single line of text.

Using this configuration, a `<pre>` element is the result in the converted **DITA** output for every code block sequence. For obtaining `<codeblock>` elements in DITA, see How can I configure the styles mapping when the wanted element doesn't exist in HTML? *(on page 364)* for setting a class attribute on the resulting `<pre>` HTML element (by setting "*pre.codeblock:separator('\n')*" in the **HTML elements** cell) and creating a custom refactoring operation.

How can I preserve the Word shapes?

The converter doesn't support the Word shapes and these will be ignored when the document is converted to HTML or DITA. A workaround is to save them manually as SVG images in MS Word before the conversion. For this, you should follow these steps for every shape inside the document:

1. Select the shape.
2. Invoke the **Cut** action from the contextual menu.

3. Invoke the **Paste Special...** action from the **Home > Paste** menu.
4. In the resulting dialog box, select **Picture (SVG)** and **OK**.

After completing these steps, the document will contain "SVG" images that will be preserved in the HTML or DITA results.

Batch converting HTML to XHTML

Suppose you have a bunch of possibly "not-wellformed" **HTML** documents already created and you want to process them using **XSLT**. For example, you may want to migrate the **HTML** documents to **DITA** using the predefined **XHTML to DITA Topic** transformation scenario available in Oxygen. So you need to create valid XML wellformed **XHTML** documents from the existing **HTML** documents and you need to do this in a batch processing automated fashion.

There are lots of open source projects that deliver processors that can convert **HTML** to its wellformed **XHTML** equivalent. For this blog post, we'll use HTML Tidy. Here are some steps to automate this process:

1. Create a new folder on your hard drive (for example, I created one on my **Desktop**: `C:\Users\radu_coravu\Desktop\tidy`).
2. Download the HTML Tidy executable specific for your platform (http://binaries.html-tidy.org/) and place it in the folder you created in step 1.
3. In that same folder, create an **ANT** build file called `build.xml` with the following content:

```
<project basedir="." name="TidyUpHTMLtoXHTML" default="main">
    <basename property="filename" file="${filePath}"/>
  <target name="main">
      <exec command="tidy.exe -o ${output.dir}/${filename} ${filePath}"/>
  </target>
</project>
```

4. In the Oxygen **Project** view, link the entire folder where the original **HTML** documents are located.
5. Right-click the folder, choose **Transform->Configure Transformation Scenarios...** and create a new transformation scenario of the type: **ANT Scenario**. Modify the following properties in the transformation scenario:
    a. Change the scenario name to something relevant, like **HTML to XHTML**.
    b. Change the **Working Directory** to point to the folder where the ANT build file is located (in my case: `C:\Users\radu_coravu\Desktop\tidy`).

   c. Change the **Build file** to point to your custom **build.xml** (in my case: `C:\Users\radu_coravu\Desktop\tidy\build.xml`).
   d. In the **Parameters** tab, add a parameter called **filePath** with the value **${cf}** and a parameter called **output.dir** with the value of the path to the output folder where the equivalent XHTML files will be stored (in my case, I set it to: `C:\Users\radu_coravu\Desktop\testOutputXHTML`).
6. Apply the new transformation scenario on the entire folder that contains the HTML documents. When it finishes, in the output folder you will find the XHTML equivalents of the original HTML files (XHTML documents that can later be processed using XML technologies such as **XSLT** or **XQuery**).

Migrating Unstructured Adobe FrameMaker Content to DITA

This article explores ways to convert unstructured FrameMaker documents to a DITA XML project.

Using a FrameMaker Plugin

There is a FrameMaker plugin that can be used for this type of conversion: http://leximation.com/tools/info/fm2dita.php.

Using FrameMaker Conversion Table

FrameMaker has a "Conversion Table" feature that can be used for this type of conversion: https://help.adobe.com/en_US/framemaker/using/using-framemaker/user-guide/WSB3FD6CD7-6930-474f-BB1E-64762D105DB6_ver12.0.html.

Unstructured FrameMaker to HTML to DITA

To perform this migration, your documents must be in the ".mif" format.

If your documents are in the '.fm' format, you can use the **MIF Wash Utility** to convert your book into a *.mif* document: https://helpx.adobe.com/framemaker/kb/how-to-use-mifwash-utility-in-framemaker.html.

The '*mif*' document must go through the following conversion and correction steps to obtain DITA content:

Publish Your Unstructured FrameMaker to HTML

In this step, the *mif* document is converted to HTML using the **Basic HTML** scenario from FrameMaker. Here are the substeps:

1. Create a settings file for publishing that contains mappings to HTML elements for all paragraph and character styles:

It's very important to have mappings to HTML heading elements for paragraph styles that mark headings in your document. In the example above, the "*Chapter*", "*Heading*", and "*Title*" styles are mapped to the "*h1*" element, the "Subtitle" and "Section" styles to the 'h2' element, and the "Subsection" style to the 'h3' element.

The styles set on the list should be mapped to the 'li' elements. In the example, it's the "Bullet", "Step", and "Step1" styles.

The paragraph styles without mapping are converted to HTML paragraphs, so you can skip the ones where the mapping is not necessary. The "Body", "CellBody", and "CellHeading" styles don't have a mapping in the example above.

2. Publish the '*mif*' document to HTML using **Basic HTML**:

This should result in an '.htm' document located in the output directory:

Correct the Resulting HTML Using a Custom Refactoring Operation

The HTML document that results from publishing requires some extra processing before applying the DITA conversion.

For this, do the following:

1. Download the content from the following GitHub repository: https://github.com/oxygenxml/dita-refactoring-examples.
2. Copy the `correctHtmlFromMif.xml` and `correctHtmlFromMif.xsl` files from the "43. FM conversion fixes" directory in one of the locations that Oxygen scans when loading the custom refactoring operations: https://www.oxygenxml.com/doc/ug-editor/topics/sharing-custom-refactoring-operations.html.
3. Restart Oxygen.
4. Add the output directory that contains the HTML file to your project.
5. In the Project view, right-click the HTML file, and invoke the **Refactoring**> **XML Refactoring** action.
6. In the resulting dialog box, search for the operation named *Correct HTML resulting from 'mif'* and apply it to the HTML document.

The applied refactoring operation corrects two problems in the HTML document:

- The list items without a parent element ('ol' or 'ul').
- Targets of the cross references that have the "name" attributes from anchor elements instead of the "id" attributes.

Convert HTML to DITA Using the Oxygen Batch Documents Converter Add-on

The Batch Documents Converter add-on has to be installed in Oxygen for this step. Follow the installation procedure here.

Before converting the document, you should configure which headings are handled as topics or sections. For this, use the **Maximum Heading Level for Creating Topics** options from the add-on's preferences page.

For converting the document, do the following:

1. Right-click the HTML document in the Project view and invoke the "HTML to DITA" option from the Batch Documents Converter submenu.
2. In the resulting dialog box, make sure that the **Ignore HTML 'div' elements** option is selected (because this element is not useful in the output) and that the **Create DITA maps from HTML documents containing multiple headings** option is also selected:

3. Click the **Convert** button.

The resulting DITA map preserves the structure of the document, images, tables, and internal and external links:

Note that migration from proprietary formats to XML is never perfect and manual changes need to be made. The names of the original styles are preserved on the `@ouputclass` attributes and these can be used for adding extra processing using refactoring operations.

Adobe and FrameMaker are either registered trademarks or trademarks of Adobe in the United States and/or other countries.

Automation - Using DITA XML Topic Titles to Generate File Names

When you create a new DITA topic in Oxygen XML Editor, you can usually specify the title of the topic and that title is used by the application to propose a file name. For example, for a topic title like **WebHelp Output**, the file name could be `webhelp_output.dita`.

In general, I think having file names that match the topic titles seems like a best practice technique to identify each topic based solely on the file name. However, there are cases where the file names do not match the topic or map titles and for such cases, I created a free-to-use Python script to update the file names of all topics and maps based on the titles specified inside them. The script should also update any link or conref references to reflect the file name changes it makes.

I think this script that updates file names based on titles for an entire DITA XML project could be useful in these cases:

- There are content management systems (CMSs) that use unique IDs as file names (for example, `abc434243243.xml`). When such projects are exported from the CMS, the file names look quite ugly on disk and also when publishing the projects to HTML-based outputs, you might get links to HTML files like `abc434243243.html`, which are ugly and hard to remember.
- In time, from one version of the user's guide to another, the titles of certain topics may change and having a way to update the file names based on titles in the entire project may be useful.
- Sometimes after migrating another file format like Microsoft Word to a DITA Map using the Oxygen **Batch Documents Converter** add-on, you may want to make sure the topic file names reflect their titles.

Below, you can leave feedback or comments about such a practice or about using the script for your project.

Chapter 6. Using Git For Technical Writing

The **Git** version control system is a feasible solution for working and collaborating on technical documentation projects regardless of their scale and number of resources.

In this blog post, we'll look into why technical writers should use Git and how various technical writing workflows can be accomplished using the Git distributed version control system.

Definition of **Git** taken from Wikipedia:

"**Git** is a distributed version control system that tracks changes in any set of computer files, usually used for coordinating work among programmers collaboratively developing source code during software development. Its goals include speed, data integrity, and support for distributed, non-linear workflows (thousands of parallel branches running on different systems). (wiki)."

So, since Git was originally intended for developers, why did technical writers start using it and why is it a good idea for technical writers to use it?

The following small articles cover various aspects of a technical documentation solution using Git:

- Advantages of Using Git *(on page 375)*
- Disadvantages of Using Git *(on page 376)*
- Useful Resources *(on page 377)*
- Applications for Working with Git *(on page 378)*
- Typical Editing Session *(on page 379)*
- Validation *(on page 385)*
- Common Workflows *(on page 386)*
- Propose Changes *(on page 395)*
- Handling Translations *(on page 396)*
- Sharing Common Settings *(on page 397)*
- Publishing Content From Git *(on page 397)*

Advantages of Using Git

- Git as a version control system is present in most software development companies. In my opinion, knowing Git is a nice asset for any technical documentation writer to have in their resume.
- Free and stable system, used by many, with relatively few bugs, has available tutorials, documentation, easy access to people who might help with Git-related questions.
- Version control:

    - History of changes made to a project.
    - Ability to revert to previous changes and to see who made certain changes.
- Collaborate on the same project with support to:
    - Check out.
    - Update.
    - Commit/Revert changes.
    - Resolve conflicts.
- Support for branching (branches used for versioning the documentation or for working on larger features) and for merging between branches.
- Support for working on your own repository even without online access and then to merge changes in the remote repository when the online connection is available.
- Collaborate with other teams in the same company on projects, either by having the documentation as part of the larger Git repository of the product, or by allowing developers who know Git to contribute reviews.
- Support for creating pull requests for people who might not have write access or want to have their work reviewed.

Related information

https://www.atlassian.com/git/tutorials/why-git

Disadvantages of Using Git

- No built-in support for a ticketing system.

> **Tip:**
> This support is added on the side either by using a commercial ticketing system (such as JIRA) or by using Git-based servers (such as GitHub or GitLab), which have their own basic ways of adding and closing issues.

- No built-in support for publishing.

> **Tip:**
> Publishing pipelines need to be built separately.

- No way to lock resources when editing to avoid conflicts.

> **Tip:**
> However, there are ways to easily resolve most conflicts.

- Mistakes can be made when using Git.

> **Tip:**
> Usually mistakes made that break the Git history or project structure can be reverted, or you can check out the entire project again.

- Some training is required to use and understand the branching and history models in Git. Ideally, you should have a **Git person** who understands how to resolve conflicts, revert and cherry-pick commits, and in general, recover from any Git issues or at least who is willing to invest some time in understanding the Git model.

> **Tip:**
> Such training tutorials and courses are readily available.

- Native Git is inefficient when asked to store large binary files (videos) or large numbers of binary files (images, PDFs) because every revision of every file is stored locally.
- No granular support for user roles. Users usually either have full read-write access to a repository or read-only access.

> **Tip:**
> When using GitHub or GitLab, users with read-only access to a repository can still create pull requests with the changes they want to incorporate into the main repository.

Related information

https://www.atlassian.com/git/tutorials/why-git

Useful Resources

Learning Git

- Git Basics tutorial by Atlassian.
- The Pro Git Book.

- Learn Git Braching Interactive Tutorial.
- Blog post: Collaborating on Docs: Best Practices and Strategies From JetBrains Writers.
- Lots of courses about learning Git on Udemy.
- Book written by a technical documentation writer: Learning Git: A Hands-On and Visual Guide to the Basics of Git - Anna Skoulikari on Amazon.

Workflows and Collaboration

- DITA For Small Technical Documentation Teams *(on page 266)*
- Docs as Code:
  - GitLab Docs As Code blog post
  - Docs as Code blog post
  - Write the Docs - Docs as code blog post
- Git Workflows
- Tactical Git - Micro Commits

Applications for Working with Git

Oxygen Applications

- **Oxygen Content Fusion** is a **web-based Git-enabled DITA CMS**. It is designed to provide the missing pieces that Git lacks when working with a DITA project, such as review workflows and publishing.
- **Git Client** add-on in Oxygen XML Editor allows you to use Git from the comfort of your authoring tool.
- **Oxygen XML Web Author** has support for all Git hosting servers. It is customisable and can be integrated with your workflows and tools, such as JIRA issue tracking.

Repository hosting solutions

- GitHub (either online or on-premise).
- GitLab (either online or on-premise).

Both of these solutions come with:

- Support for creating and managing private/public repositories.
- Basic support for creating and handling tickets/issues.
- Support for creating pull/merge requests.

Other client-side applications

There are lots of free client-side applications that can be used to work with Git repositories:

- **Git Fork**
- **Git Tower**
- **SourceTree**
- **GitHub Desktop**
    - Allows you to cherry-pick commits with drag & drop, but otherwise has limited functionality
- The `git` command-line client
    - Useful for scripting or advanced commands

Typical Editing Session

Usually, an editing session has the following stages that are repeated throughout the day:

- **Clone/Checkout Branch From Remote Repository** - Checkout a branch from the remote repository on which you want to work. Usually done only once.

- **Edit Content** - The project is edited and modified in the current working copy.
- **Commit Content** - Commit the changes to the local working copy.
- **Pull Content From Remote Repository** - Obtain the latest content for the edited project branch.
- **Push Content To Remote Repository** - From time to time, push the commits to the remote repository so that others can use them as well. You must always pull content from the repository before pushing to it, otherwise the pull will fail.

Commit Message

Each commit has a commit message. The commit message is in general a brief description of the changes. When using a ticketing system, as a best practice the commit message contains the ID of the ticket.

Micro Commits

Commit often, whenever a small change in the documentation is beneficial and makes sense by itself. Useful links: https://stackoverflow.blog/2022/12/19/use-git-tactically/

Handling Simple Conflicts

Conflicts are usually caused by trying to push changes made to resources that, in the meantime, have been modified by others on the remote repository. Conflicts normally occur when working with Git, they can be resolved and they are not a sign that anyone did anything wrong.

- If changes are made to text files on different lines, the conflicts are automatically merged.
- If the conflicting changes are on the same lines, you can resolve the conflict.

The conflict can be resolved by keeping your version, or the version of the remote collaborator, or by manual merging:

Editing Session with Oxygen Content Fusion

You can use the Oxygen Content Fusion authoring features to create content in your Git repository.

Here is your usual editing workflow to edit content on a Git branch:

- Open the workspace for that branch
- Synchronize the workspace
- Edit
- Pull incoming commits
- (Optional) Validate
- Commit

I will detail each step below.

Open the workspace

Content Fusion allows you to create a workspace for each Git branch. The workspace is lightweight meaning that it is created instantly and with minimal resource overhead. To open it, select the branch and click the **Edit in workspace** button.

Synchronize the workspace

In most cases, your workspace is updated automatically when you open it. However, if your workspace has some stale changes (maybe from an old abandoned edit), you will have to manually synchronize it. You have two tools:

1. Use the **Discard all changes** action from the **Changes** side view if the existing changes are not relevant anymore:

2. **Pull** the incoming commits. This lets you bring in your workspace commits made by your collaborators.

Edit

Use the workspace tools to work on your content.

Pull incoming commits

While you were editing, your collaborators pushed some commits. Before you can commit your changes, you have to integrate theirs first. In rare cases, you might run into merge conflicts. To help you solve them, Content Fusion has a dedicated merge tool:

Validate

Use the **Problems** view to check if your changes introduce any validation problem or a broken link

Commit

To make your changes available in the Git repository, you have to **Commit** them. Make sure to provide a descriptive commit message.

> **Note:**
> In Content Fusion, when you commit changes from your workspace they are immediately available on the underlying branch.

Validation

DITA XML content needs to be validated against the DITA schemas and possibly with additional business rules (usually implemented using Schematron checks). There are various ways to ensure the proposed content changes are valid:

- Validate the entire DITA project on the server side *(on page 272)*. Examples:
  - Run validation on an integration server like Jenkins.
  - Run validation as a GitHub action *(on page 181)*.
- Validate on the client side before committing or pushing changes. For example, the Oxygen Git Client add-on allows two types of client side validation:
  - Validate modified resources before committing.
  - Validate the entire DITA project before pushing.

Common Workflows

Workflows are recipes for working and collaborating successfully on a Git project.

Common workflows for working with Git:

- Centralized Workflow *(on page 387)*
- Feature Branches *(on page 388)*
- Release Branches *(on page 389)*
- GitFlow *(on page 394)*

Re-base or merge branches?

In most these workflows changes are made on one branch and then integrated on other branches. The changes can be integrated either with Git merge or with Git rebase. This post from Atlassian explains the differences. Oxygen's Git client add-on supports in its **Git Branch Manager** view only the possibility to merge a branch in another one. Our general advice is to use **Git merge**.

Some highlights for each of these choices:

- **Git Merge**: Combines two branches by creating a new commit. It keeps the history of both branches as-is.

- **Git Rebase**: Moves the entire history of the branch you're on to the tip of the branch you're rebasing onto. It gives a linear commit history.

> **Important:**
> The golden rule of `git rebase` is to never use it on *public* branches.

Centralized Workflow

We all work with the same project(s) on the main repository branch. We have separate branches which mark past releases. Both for small bug fixes and for features which may take a longer time to implement, we work on the main branch. https://www.atlassian.com/git/tutorials/comparing-workflows

Common workflow:

- Check in the project (one time operation).
- Pull to get changes from the main repository.
- Commit and push changes.
- Deal with possible conflicts which may appear.
- Request review either before commit or after the change was incorporated in the main branch.

Workflow for updating documentation for a past release:

- Switch to branch specific for that release.
- Pull to get changes from the main repository.
- Commit and push changes.
- Deal with possible conflicts which may appear.

Cherry Pick

Pick a commit from another branch and incorporate it. Useful to take small fixes for a version branch and add them to the main branch.

https://www.atlassian.com/git/tutorials/cherry-pick

Feature Branches

We work on the smaller fixes on the main branch, we create separate branches for larger features/fixes. We also create separate branches for past released versions of the documentation in order to add small fixes and be able to re-publish documentation for them.

- Create a branch on which you work on a larger feature either by yourself or with some other colleagues.
- You can also request review on the branch.
- When finished, merge the branch in the main repository branch or create a pull request to merge the branch.

Related information

https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow

https://betterprogramming.pub/solution-architecture-docs-as-code-366a7b40f9e5

Branches For All Changes

We create branches for any change, either for large features or small fixes. We also create branches for past released versions of the documentation in order to be able to add small fixes and re-publish documentation for a specific past version.

- Create a branch on which you work to fix small problems or to work on a larger feature either by yourself or with some other colleagues.
- You can also request review on the branch.
- When finished, merge the branch in the main repository branch or create a pull request to merge the branch.

Release Branches

My name is Chris Papademetrious and I work for Synopsys Inc., a semicoductor design automation software company. I am an occasional guest contributor to the Oxygen XML blog, and I wanted to share how we use Git.

Our team writes documentation for many different products. And for the most part, all products follow a common software and documentation release schedule (every three months). Our technical writers are knowledgeable on the products they write for, but they tend to be novices with Git.

Our Attempt at the Centralized Workflow

When we first adopted DITA and Git, we used the centralized workflow. However, we ran into issues when a particular product group needed to delay their release schedule. When the rest of the group began working on the next release in the main branch, the delayed group could not stay in the main branch because then "last-release" and "next-release" content would get mixed and there was no single point where the main branch represented the final release state. And so in these cases, we had to fork a release branch off the main branch and move the delayed product writers there. (And sometimes writers would keep working in the main branch out of inertia, and so the main branch content would get mixed up anyway...)

We considered using the feature branch workflow. However, this would require writers to be responsible for branch actions (forking feature branches and submitting pull requests), which they had never done before. Many products have multiple writers, which would require those feature branches to be synchronized on the server for collaboration. Some products have 40+ features in a release cycle, and our past experiences told us that writers could sometimes forget to switch branches. And so we decided not to use this workflow.

We needed a workflow in which a writer could simply continue to write for a release and not worry about switching branches until the release was done—*regardless of whether their product finished its release early or late*. So, we moved to a *release branch* workflow.

The Release Branch Workflow

In our release branch workflow, we create a branch for each release, named by the scheduled release date. A writer can work in a release branch as long as they like, and they can move to the next release branch whenever they are ready. If a feature requires collaboration by multiple writers, they can collaborate in that release branch.

The key feature of our release branch workflow is the *cascaded release merge*. Each week, we merge new commit activity forward from earlier release branches to later release branches. If a branch has no new commit activity since the last merge, no merge is performed from it.

In the preceding diagram, the release1 branch is completed and quiet. Product writers are finishing work in the release2 branch and moving to the release3 branch. Their work in the release2 branch is automatically propagated to release3. At some point, a product team finds that a retroactive documentation fix is needed for a feature in the release1 branch. This fix is automatically propagated forward into the release2 and release3 branches.

In this workflow, the writer must remember only to check out the next release branch after they complete their current release. And this is a notable enough accomplishment that they tend to remember to do this.

However, writers can work in multiple release branches if they want to. Most products have major releases on every third release interval. In this case, writers can make minor typo fixes and rewordings in the next minor release branch, but get an early start on feature documentation in the next major release branch. (Although in this case, the writers must be diligent again about remembering to switch branches!)

For the most part, writers don't need to worry about the logistics of the release merges. However, there are some cases in which they do need to be aware of them. For example,

- If a writer writes content for one release and wants to incrementally modify that content for the next release, they must wait for the merge to propagate the work forward to the next branch to be able to continue work on it.
- If a writer wants to add content that should not propagate forward to subsequent releases (such as a temporary bug or limitation that is fixed in the next release), then they add content to the earlier release, wait for a merge, then remove it in the next release.

In these cases, the writers can ask me to run a merge manually so they do not have to wait.

The Branch-Merging Perl Script

The release branch merge is performed by a perl script called `merge_repo.pl`. The usage is as follows:

```
$ merge_repo.pl --help
Usage:
      --dry-run
           Do not actually commit any changes
      --starting-at branch_name
           Start merging at this branching (merging from previous branch)
```

The script uses the command-line `git` client to perform its work. One notable requirement is that the script needs a local repository to perform its merging work in. The script automatically fetches and pulls all remote updates, creating new local branches and deleting local branches that no longer have matching remote branches as needed.

The script obtains the chronological list of release branches by querying and sorting all branches that match `release/*` - there is no need to update any branch list text files or databases as new release branches are created.

Here is an example of its output:

```
$ merge_repo.pl
Checking for commits to merge from 'release/2022.09' to 'release/2022.12'...
Merging 1 commits from 'release/2022.09' to 'release/2022.12'...
Pushing merge to 'release/2022.12' upstream...
Merge from 'release/2022.09' to 'release/2022.12' succeeded.

Checking for commits to merge from 'release/2022.12' to 'release/2023.03'...
Merging 4 commits from 'release/2022.12' to 'release/2023.03'...
Pushing merge to 'release/2023.03' upstream...
Merge from 'release/2022.12' to 'release/2023.03' succeeded.
```

```
Checking for commits to merge from 'release/2023.03' to 'release/2023.06'...
Merging 8 commits from 'release/2023.03' to 'release/2023.06'...
Pushing merge to 'release/2023.06' upstream...
Merge from 'release/2023.03' to 'release/2023.06' succeeded.
```

Here is the resulting commit graph as shown on the Gitlab server web page, from the previous merge to the just-completed merge:

As mentioned earlier, no matter how many "earlier" branches exist, no forward merge is performed from a branch unless it contains new activity since the last merge.

If a release merge result cannot be pushed (perhaps because a writer just pushed something new to the server), the script abort its current merge, updates its local repository, and retries the merge.

If the script encounters a merge conflict, it prints information to STDERR about how to manually reproduce and resolve the conflict, then continue the merge:

```
$ merge_repo.pl
Checking for commits to merge from 'release/2022.09' to 'release/2022.12'...
Merging 1 commits from 'release/2022.09' to 'release/2022.12'...
```

```
Pushing merge to 'release/2022.12' upstream...
Merge from 'release/2022.09' to 'release/2022.12' succeeded.

Checking for commits to merge from 'release/2022.12' to 'release/2023.03'...
Merging 4 commits from 'release/2022.12' to 'release/2023.03'...
Merge from 'release/2022.12' to 'release/2023.03' failed:
Auto-merging some_topic.dita
CONFLICT (content): Merge conflict in some_topic.dita
Automatic merge failed; fix conflicts and then commit the result.
some_topic.dita:21: leftover conflict marker
some_topic.dita:24: leftover conflict marker
some_topic.dita:25: leftover conflict marker
some_topic.dita:27: leftover conflict marker

Perform the following commands manually:

git checkout release/2023.03
git merge release/2022.12
...resolve conflict...
git push
merge_repo.pl --starting-at release/2023.06
```

The script runs every Wednesday as a `cron` job. For conflicts or other errors (like server downtime), the resulting output to STDERR causes `cron` to email the command output to the script owner (in this case, me). Thanks to this behavior of `cron`, the script operates quietly unless attention is needed.

GitFlow

Gitflow is a legacy Git workflow that was originally a disruptive and novel strategy for managing Git branches.

The main branch holds the official content which will be released, we mostly commit small fixes on a development branch and use feature branches based on it.

Related information

https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

Propose Changes

Besides the team of technical writers who have full write access to the documentation repositories, there may be other collaborators (e.g. subject matter experts) who have read access to the repository and sometimes want to propose changes.

The workflow for proposing changes to a repository usually works like this:

1. **Identify the Need for a Change**

   The SME (subject matter expert) starts by reading the published output and thinking about what to change

2. **Click “Edit Online”**

   They click on an "Edit online" link on the page they want to change

3. **Make the Edits**

   Using the Oxygen XML Web Author the SME can make changes directly to the DITA topic—no need to download or clone the repository.

4. **Submit the Changes**

   When they try to commit, Web Author automatically creates a pull request with their changes

5. **Review and Merge**

   The technical writer reviews the change, improve the content if needed, and merges the pull request.

You can see this workflow presented in detail in the video below. Short on time? Jump straight to the demo section and see it in action.

https://www.youtube.com/embed/YzQ0XhmtPUI

Related information

Pull Requests

Handling Translations

Some useful advice for translating a DITA XML-based project can be found here: Translating your DITA Project *(on page 287)*.

The translated DITA content can also be stored on Git:

- Keep the translation for each language in a separate folder in the same Git project. As a benefit, binary resources can be shared between projects.

- Keep a separate parallel project with all the translations.

Sharing Common Settings

It makes sense to share various common editing settings when working on a common Git project:

- Spell check dictionaries and learned words.
- Checking Terminology *(on page 458)*.
- Actions and behaviors for editing content *(on page 419)*.
- Custom validation and quick fixes *(on page 291)*.
- Publishing templates.

There are various ways to share these custom editing settings between the people who collaborate on the same Git projects:

- Use a web editing tool to edit the Git content directly in a web browser with all settings being common in the tool. Example: Oxygen XML Web Author Git connector.
- Manually share a common set of settings.
- Share the common settings directly in the Git project. Example: DITA Startup Project. When an Oxygen project that contains settings is distributed in a Git project, writers automatically get the latest updates to the Oxygen environment without any work on their part.

Publishing Content From Git

There are several ways to publish:

- Publish on the client side using the local Git repository and then manually upload the published content to a web server.
- Server Side Building and Publishing *(on page 179)* by connecting for example integration server to the Git project. For each version branch, a separate server-side job could be created to run when the content is modified on that branch.
- Use **Oxygen Content Fusion** to publish your content. It offers the automation of a continuous integration and continuous delivery (CI/CD) server without all the configuration work.

Tame the Branch Chaos – Switch Less, Write More

You're mid-sentence, deep in the flow of writing. Then a Slack ping: "Can you quickly check something on the release notes branch?" You sigh. Time to stash your changes, switch branches, rebuild context—and just like that, your momentum is gone.

For technical writers, this isn't a rare disruption. It's daily reality. Between juggling multiple documentation tasks, waiting for SME feedback, and jumping between feature branches, staying focused becomes a luxury.

The Multi-Tasking Writer's Dilemma

In an ideal world, a technical writer picks up a task, finishes it in a focused work session, and moves on to the next. In reality? Writing often happens in fragmented windows between SME replies, engineering changes, and shifting priorities.

This leads to a juggling act between:

- **Multiple feature branches** for in-progress tasks at different stages of completion.
- **Quick hotfixes or urgent edits** that need to go out *now*.
- **Reviews and feedback** that come in late, requiring you to revisit a branch you last touched two weeks ago.

Context switching happens not just between topics, but between Git branches, editor states, and mental models.

Each time you need to switch branches, you face a decision tree:

- Do you commit half-finished work?
- Do you stash and risk conflicts later?
- Do you clone the whole repository again into another folder?

This constant overhead takes a toll: it interrupts focus, slows down delivery, and makes a high-cognitive-load job even harder. Writers end up spending more time managing branches than actually writing.

Four Ways to Juggle Branches

When you're deep in documentation work and need to temporarily switch to another branch, you have a few options. Some are more elegant than others—but each comes with its own trade-offs.

Let’s walk through the most common approaches, what they involve, and where things can go wrong.

The Git Stash Dance

Stash your current changes, switch to the other branch, then come back and re-apply the stash.

Workflow in Oxygen:

1. Stash your local changes
    a. If you haven't already, install the Oxygen Git Client Add-on
    b. In the **Git Staging** view, open the triple dots menu and choose **Stash changes**

c.

2. Switch the branch
3. Make quick edits, commit or push as needed
4. Switch back to the original branch
5. Apply the stash
    a. In the **Git Staging** view, open the triple dots menu and choose **List changes**

b.

CLI workflow

If you are a script passionate, you can use these commands to achieve the same result as the workflow above:

```
git stash push -u -m "WIP on 'master' - taming the branch chaous [27 Mar 2025]"

git checkout release-notes-branch
```

```
# Make quick edits, commit or push as needed

git checkout master

git stash pop
```

Tip:

- Make sure to leave the "Include untracked files" checkbox on, as it is by default. Otherwise, some newly added files may not be included in the stash.
- Always preview stashes before applying.
- Include the branch name in the stash name. This way, you avoid applying the stash on the wrong branch.
- Include the date in the stash name. This way you can cleanup old stashes with confidence.

Temporary WIP Commit

Quickly commit your in-progress work to get it out of the way.

Oxygen Workflow

1. Stage and commit all the local changes
2. Switch the branch
3. Make quick edits, commit or push as needed
4. Switch back to the original branch
5. Open the history view & filter commits by "Current local branch"
   a.

6. Reset branch to the parent commit with "mixed mode"
   a.

CLI workflow

```
git add .

git commit -m "WIP: halfway done with config docs"

git checkout release-notes-branch

# Do your other work

git checkout master

git reset HEAD~1 --mixed
```

Tip:

- If working with others on the same branch, do not push! Your local commits may affect rebases or create confusion.
- When undoing the commit, make sure you have the "mixed" option checked.

Checkout in a Separate Folder

Create another working directory with a separate clone. This approach is simple to understand but is heavy - you duplicate the entire repository and is not uncommon for companies to have multi-gigabyte repositories.

As a mitigation, Git introduced the `worktree` command which allows you to reuse the `.git` folder between multiple working copies. For example, the working copy of the Oxygen's user guide has ~250MB and the `.git` folder is ~200MB of them.

To use this feature you need to go to the terminal and work with these commands:

```
# From your repo root
git worktree add ../release-notes release-notes-branch
cd ../release-notes
# Make your changes and commit them
cd ../my-main-repo
```

> **Tip:**
> `worktrees` are fantastic for long-lived parallel work. Just don't forget where each copy lives and to remove them once you are done: `git worktree remove ../release-notes`.

Oxygen Content Fusion Workspaces

Oxygen Content Fusion is a **web-based Git-enabled DITA CMS** providing structured content authoring, AI-assisted refinement via Oxygen AI Positron, and publishing.

One of its benefits for an author is that you can create multiple lightweight authoring workspaces in your browser, for *any* branch, and switch between them by just switching browser tabs. Unlike multiple checkouts, workspaces don't duplicate files. They're created instantly without copying any content, and with minimal overhead.

Workflow in CF:

1. Stay on your current task in your authoring tool.
2. In Content Fusion, select a different branch to create a new workspace.
   a.

3. Edit or review content independently from your current Git state.
4. Commit changes when you are ready

a.

5. ... or just leave the changes there, and switch to some more urgent work

> **Tip:**
> Need more feedback from a SME? Just create a review task from your workspace and set a due date. Content Fusion will automatically remind the SME—so you don't have to chase them for a response.

How Do *You* Handle Branch Chaos?

Every technical writer has their own way of dealing with the Git shuffle. Maybe you're a `stash` master. Maybe you keep six cloned repositories open at all times (no judgment). Or maybe you've found a completely different workflow that works for your team.

We'd love to hear from you:

- Which approach saves you the most time?
- Have you tried Git `worktree`s or Content Fusion workspaces?
- What's the weirdest branch-juggling ritual you've developed?

Important:

Share your tips, tricks, or horror stories in the comments—or reach out to us directly. We're always looking to learn from real-world workflows and improve the tools writers rely on every day.

Related information

Webinar: Solving Common Git and DITA Challenges

XML Based Docs As Code – Combining DITA XML With GitHub and Continuous Delivery

Webinar- Docs as Code: Documentation Management Inspired by Software Development

Chapter 7. SDK

Oxygen XML Editor provides various ways to customize the application either by adding frameworks to customize editing for a certain XML vocabulary or by creating plugins.

Customizing Oxygen XML Editor (Overview)

Let's assume you just bought Oxygen and want to customize it in order to better suit your team's workflow or to add support for your custom XML vocabulary. I will attempt to provide a set of use cases along with indications and useful links about what customization should be done in order to cover each use case:

Add support for editing a specific XML vocabulary

This kind of support usually should include:

- New file templates.
- Validate opened XML documents according to a custom schema.
- Provide custom CSS to be used in the **Author** visual editing mode.
- Provide custom toolbar, menu, and contextual menu actions to be used in the **Author** visual editing mode.

Such support is done using a document type/framework configuration: The Oxygen SDK (Part 2: Frameworks) *(on page 425)*.

Customize the existing support for a specific XML vocabulary

Oxygen comes with pre-configured framework configurations for the most commonly used XML standards such as **DITA**, **Docbook**, **TEI**, or **XHTML**. You may want to customize the existing configuration for one of these vocabularies and share the customization with others. Such customizations may include:

- Provide custom file templates: Sharing New Custom File Templates for a Specific Vocabulary *(on page 413)*.
- Provide a custom CSS layer: Customizing the DITA Visual Editing Experience *(on page 245)*.
- Provide custom Schematron validation rules: Sharing Schematron Validation Rules *(on page 409)*.
- Provide custom actions or modify the existing actions.

All of these customizations can be done by extending the existing framework configuration and then sharing the extension: Document Type Extension Sharing *(on page 407)*.

Provide custom settings

You may want to provide all members of the team with a fixed set of options: Sharing Application Settings *(on page 464)*.

Modify the application appearance or behavior using plugins

Oxygen plugins can be used to:

- Contribute custom views and toolbars for the entire application.
- Filter out existing views or toolbar actions.
- Contribute a fixed set of options/settings to the application.
- Register various listeners and be notified when a document is opened, closed, saved, and so on.
- Use the existing API to make changes to the opened documents.

A more detailed overview of Oxygen plugins can be found here: The Oxygen SDK (Part 1: Plugins) *(on page 424)*.

The most popular plugin extension is the Workspace Access plugin extension: https://www.oxygenxml.com/doc/ug-editor/topics/workspace-access-plugin.html. This extension is notified when the application starts and can contribute custom views, custom main menu items, custom main toolbar actions, or add listeners for various document-related changes.

There is also a JavaScript-based workspace access plugin extension that can use JavaScript to call our Java API: https://www.oxygenxml.com/doc/ug-editor/topics/workspace-access-plugin-js.html.

A set of sample plugins built with the JavaScript-based extension can be found here: https://github.com/oxygenxml/wsaccess-javascript-sample-plugins.

Our Maven-based SDK can be used to develop both plugins and framework Java customizations: https://www.oxygenxml.com/oxygen_sdk_maven.html.

Document Type Extension Sharing

Instead of copying an entire framework configuration folder (like **DITA** or **Docbook**), then modifying and distributing it, you can choose to extend that framework and distribute the extension. This way, you benefit from new functionality added to the base framework by newer Oxygen versions and you can still use your customizations.

The steps below describe how to construct and share an extension of the **DITA** framework that removes certain elements from the content completion list:

1. Somewhere on your local disk (in a place where you have full write access), create a folder structure like: `custom_frameworks/dita-extension`.
2. In the **Document Type Association / Locations** preferences page, add the path to your `custom_frameworks` folder in the **Additional frameworks directories** list. Click **OK** or **Apply** to save your changes.
3. In the **Document Type Association** preferences page, select the **DITA** document type configuration and use the **Extend** button to create an extension for it.
4. Give the extension a custom name (for example, **DITA - Custom**), then change its **Storage** to external, and save it to a path like: `path/to/.../custom_frameworks/dita-extension/dita-extension.framework`.
5. Make changes to the extension (for example, go to the **Author->Content completion** tab and add elements that should not be presented to the end users in the **Filter - Remove content completion items** list).
6. Click **OK** to close the dialog box and then either **OK** or **Apply** to save the preferences changes.

After you perform the steps above, in the `dita-extension` folder you will have a fully functioning framework that can be shared with others.

The framework can be shared with others in several ways:

- Copy it to their `[OXYGEN_DIR]/frameworks` directory.
- Somewhere on the local disk, create a `custom_frameworks` folder, copy the framework there, and then in the **Document Type Association / Locations** preferences page, add the path to your `custom_frameworks` folder in the **Additional frameworks directories** list.
- Distribute the framework along with a project by following these steps:
    1. On your local drive, create a directory with full write access that contains the project files and a `custom_frameworks` folder that contains your `dita-extension`.
    2. Start the application, go to the **Project** view, and create a project. Save it in the newly created directory.
    3. In the **Document Type Association / Locations** preferences page, select **Project Options** at the bottom of the page.
    4. In the **Additional frameworks directories** list, add an entry like: `${pd}/custom_frameworks`.
    5. Add other resources to your project (for example, you can have all of your DITA content located inside the project folder).

6. You can then share the new project directory with other users. For example, you can commit it to your version control system and have them update their working copy. When they open the customized project file in the **Project** view, the new document type becomes available in the list of Document Types.

- Deploy the framework/document type configuration as an add-on.

After your team members install the framework, they can check in the **Document Type Association** preferences page (in the list of Document Types) to see if the framework is present and if it appears before the bundled **DITA** framework (meaning that it has higher priority).

Sharing Schematron Validation Rules

Custom **Schematron** rules are a great way to ensure consistency for the edited XML structure, especially when there is a large team working on the same set of documents. You can use **Schematron** for numerous use cases. For example, to make sure that certain elements are not used at all, to impose restrictions on the amount of text for an element, or to impose restrictions on certain elements based on various attribute values or text content set in other elements. Furthermore, you can define quick fixes for each **Schematron** rule to give the technical writer proposed solutions for the reported problem. In a previous blog post, I described how we use **Schematron** rules for our **Oxygen User's Manual** to impose restrictions when editing DITA content: Schematron Checks to help Technical Writing *(on page 291)*.

Once you define the **Schematron** rules, they can be shared with the other members of your team via a document type **framework configuration**: The Oxygen SDK (Part 2: Frameworks) *(on page 425)*.

In this post, I'm going to provide some precise steps to share a set of **Schematron** rules with a **DITA** framework extension.

Instead of making changes directly to the **DITA** framework configuration folder (located in **OXYGEN_INSTALL_DIR\frameworks\dita**) and distributing it, you can choose to extend the **DITA** framework and distribute the extension. This way, you will benefit from new functionality added to the base framework in newer versions of **Oxygen** and you can still use your customizations.

The following steps describe how you can extend a **DITA** framework that contains an additional Schematron rules file:

1. Create a folder structure (for example, **custom_frameworks/dita-extension**) somewhere on your local disk, in a place where you have full write access.
2. In that new folder structure, create another folder (**custom_frameworks/dita-extension/rules**) that will contain all of your custom Schematron files.

3. Create a new **ISO Schematron** file (for example, **additionalRules.sch**) that contains the additional rules you want to impose. For example, the following Schematron rule reports an error if the **@scale** attribute is used on an image:

```
<schema xmlns="http://purl.oclc.org/dsdl/schematron" queryBinding="xslt2"
    xmlns:sqf="http://www.schematron-quickfix.com/validator/process">
    <pattern>
        <rule context="*[contains(@class, ' topic/image ')]">
            <assert test="not(@scale)">
                Dynamically scaled images are not properly displayed, you
                should scale the image with an image tool and keep it within
                the recommended width and height limits.
            </assert>
        </rule>
    </pattern>
</schema>
```

4. In the *Document Type Association / Locations* preferences page, add the path to your **custom_frameworks** folder in the **Additional frameworks directories** list. Click **OK** or **Apply** in the **Preferences** dialog to save your changes.
5. In the *Document Type Association* preferences page, select the **DITA** document type configuration and use the **Extend** button to create an extension for it.
6. Give the extension a custom name (for example, **DITA - Custom**), then change its **Storage** option to **External**, and then save it to a path such as: **path/to/.../custom_frameworks/dita-extension/dita-extension.framework**.
7. Make whatever changes you desire to the extension, go to the **Validation** tab, edit the default **DITA** validation scenario, and add an extra validation stage to it (a stage that uses your custom **Schematron** schema). More details about validation scenarios can be found here: https://www.oxygenxml.com/doc/ug-editor/topics/dg-validation-scenarios.html.
8. Click **OK** to close the dialog box and then either **OK** or **Apply** to save the preferences changes.

After you perform the steps above, in the **dita-extension** folder you will have a fully functioning framework extension that can be shared with others.

There are several methods for sharing the framework and you can read about them here: https://www.oxygenxml.com/doc/ug-editor/topics/author-document-type-extension-sharing.html.

After your team members install the framework, they can check the *Document Type Association* preferences page to make sure the framework is present in the list of **Document Types** and that it appears before the bundled **DITA** framework (meaning that it has higher priority).

You can use the framework extension mechanism to customize numerous aspects of the **DITA** functionality in Oxygen. For example, you can remove various elements from the content completion list: Document Type Extension Sharing *(on page 407)*. Or you can distribute custom new file templates: Sharing New Custom File Templates for a Specific Vocabulary *(on page 413)*.

Public hosted Oxygen Plugin and Framework Projects

All resources, Frameworks and Plugins which we make publicly available to contributors can be found on the **oxygenxml** GitHub group:https://github.com/oxygenxml/:

- Plugins: https://github.com/search?q=topic:oxygen-standalone-plugin+org:oxygenxml&type=Repositories

  Examples:
  - Workspace containing sample Oxygen plugins implemented in Javascript: https://github.com/oxygenxml/wsaccess-javascript-sample-plugins.
- Frameworks: https://github.com/search?q=topic:oxygen-framework+org:oxygenxml&type=Repositories.

  Examples:
  - Framework which adds **Daisy** support in **Oxygen**: https://github.com/oxygenxml/Daisy
  - Framework which adds **STRATML** support to **Oxygen**: https://github.com/oxygenxml/stratml
  - **S1000D** Framework which adds some limited support to edit **S1000D** documents in the Author visual editing mode:https://github.com/oxygenxml/S1000D

Besides, there are interesting **Oxygen** plugins and frameworks which are developed as separate public projects and maintained by third parties (either users or companies). I will try to compile a list below:

- Project **Argon** by German company Axxepta provides a plugin which can connect Oxygen to a BaseX server: https://github.com/axxepta/project-argon. The integration also provides CMS-like capabilities for searching and version control. The plugin is available as an add-on: http://argon-author.com/.
- **LanguageTool** plugin for **Oxygen**: https://github.com/danielnaber/oxygen-languagetool-plugin
- The latest version of **XSpec** framework is available as an external framework or an add-on. See XSpec Wiki for more details.
- You can define terminology terms and then create Schematron rules for them using the Doctales terminology checker Oxygen framework: https://github.com/doctales/org.doctales.terminology.

- Framework which adds JATS/NLM support for **Oxygen** developed by Wendell Piez: https://github.com/wendellpiez/oXygenJATSframework.
- Framework for validating **XSL-FO** with **Antenna House** extensions developed by Antenna House: https://github.com/AntennaHouse/focheck.
- Fork of the **JATS** framework which adds Schematron checks and Literatum package building: https://github.com/le-tex/oXygenJATSframework_Literatum
- Three open-source frameworks for editing ATA-2300, S1000D and RIF XML documents in Oxygen developed by Amplexor: https://github.com/Amplexor.
- Besides being bundled with **Oxygen** the **TEI** framework is also available as a project partly maintained by the **TEI** community: https://code.google.com/p/oxygen-tei/

- HisTEI: An **Oxygen** framework for historical documents encoded in TEI.

  More details: https://github.com/odaata/HisTEI, https://www.oxygenxml.com/pipermail/oxygen-sdk/2014-November/000182.html

- oXbytei is an oXygen framework for editing TEI, that is configured by TEI's header. It offers high-level functions that facilitate everyday work on TEI documents. More details: https://github.com/SCDH/oxbytei#an-oxygen-framework-configured-by-tei

- Ediarium is an extension package for TEI editing within Oxygen.

  More details:https://www.bbaw.de/en/bbaw-digital/telota/research-and-software/ediarum, https://github.com/telota/ediarum

- TEI Facsimile Plugin offers a side view in which users can load an image and see the marked areas (all the **zone** elements from a TEI document), draw new areas over the image and copy them into the editor:https://github.com/oxygenxml/TEI-Facsimile-Plugin.
- A Workspace Access Plugin for Oxygen XML Editor that creates TEI-conform UUIDs starting with a letter: https://github.com/digicademy/CustomUUID
- TEI Authorizer is a plugin for Oxygen which lets you query remote authority files via HTTP and use retrieved data to autocomplete attribute values in your TEI documents and define and implement forms to fill out new data and post it to your server via HTTP:https://github.com/BCDH/TEI-Authorizer
- Framework for editing UBL documents (UBL 2.1, 2.2 csd01 and 2.2 Pre-award csd02wd01pa01) developed by G. Ken Holman: https://cranesoftwrights.github.io/resources/ubl/#oxygenubl
- Framework for validating OASIS genericode 1.0 documents developed by G. Ken Holman: https://cranesoftwrights.github.io/resources/ubl/#oxygengc.
- Automatic builder for **Oxygen** frameworks which allows user to describe framework's behaviour by using only **XQuery**, **HTML**, and **CSS**, and automatically generate the framework archive ready to

be deployed (developed by Claudius Teodorescu):https://github.com/claudius108/oxygen-addon-builder-plugin
- Plugin developed by Clausius Teodorescu which allows opening a JavaFX-based web browser in Oxygen: https://github.com/claudius108/oxygen-webview-plugin.
- **OpenDocs** Plugin allows you to focus on specific file types opened in the editor in order to perform various actions on them. GitHub repository: https://github.com/nkutsche/opendocs/.
- **XsltGui** Project by Patrik Stellmann is an example of how you can show dialogs to the end users from an XSLT script: https://github.com/dita-semia/XsltGui.
- Plugin developed by **pagina GMBH** which expands a custom editor variable called **${clipboard}** to the clipboard contents: https://github.com/paginagmbh/oxygen-plugin_clipboard-editor-variable.
- The **NameDropper** Oxygen plugin can be used to simplify the process of tagging names in XML and associating those names with authoritative identifiers:https://github.com/emory-libraries-ecds/namedropper-oxygen.
- Sample Oxygen plugin demo by Tobias Fisher: https://github.com/tofi86/oxygen_PluginDemo.

If anyone else wants to add something else to the list, just drop us an email.

Sharing New Custom File Templates for a Specific Vocabulary

The support **Oxygen** provides for editing **DITA** topics comes with quite an extensive set of new file templates used to create new DITA topic types. If you have a team of writers, you may want to filter out certain new file templates or add your custom new file templates, then share these custom templates with your team members.

This blog post will attempt to give you some clear steps for sharing a custom set of new file templates with your team.

All the original **DITA** new topic templates are located in the folder: `OXYGEN_INSTALL_DIR\frameworks\dita\templates`.

Instead of making changes directly to that folder, copying the entire **DITA** framework configuration folder (like `OXYGEN_INSTALL_DIR\frameworks\dita`), modifying and distributing it you can choose to extend the **DITA** framework and distribute the extension. In this way, you will benefit of new functionality added to the base framework by newer Oxygen versions and still use your customizations.

The steps below describe how an extension of the **DITA** framework which adds a custom set of new file templates can be constructed and shared:

1. Create somewhere on your disk, in a place where you have full write access a folder structure like: `custom_frameworks/dita-extension`.
2. In that new folder structure create another folder `custom_frameworks/dita-extension/templates` which will contain all your custom new topic templates.
3. In the **Document Type Association / Locations** preferences page add in your **Additional frameworks directories** list the path to your `custom_frameworks` folder. Click **OK** or **Apply** in the **Preferences** dialog to save your changes.
4. In the **Document Type Association** preferences page select the **DITA** document type configuration and use the **Extend** button to create an extension for it.
5. Give a custom name to the extension, for example **DITA - Custom** and then change its **Storage** to external, then save it to a path like: `path/to/.../custom_frameworks/dita-extension/dita-extension.framework`.
6. Make changes to the extension, go to the **Templates** tab, remove all previous entries from it and add a new entry pointing to your custom templates folder: `${frameworkDir}/templates`.
7. Click **OK** to close the dialog and then either **OK** or **Apply** to save the preferences changes.

After you perform the steps above you will have in the `dita-extension` folder a fully functioning framework extension which can be shared with others.

The framework can then be shared with others in several ways:

- Copy it to their `[OXYGEN_DIR]/frameworks` directory.
- Create somewhere on disk a `custom_frameworks` folder, copy the framework there and then from the **Document Type Association / Locations** preferences page add in your **Additional frameworks directories** list the path to the `custom_frameworks` folder.
- Distribute the framework along with a project.

  Follow these steps:

  1. On your local drive, create a directory with full write access, containing the project files and a `custom_frameworks` folder containing your `dita-extension` framework.
  2. Start the application, go to the **Project** view and create a project. Save it in the newly created directory.
  3. In the **Document Type Association / Locations** preferences page, select **Project Options** at the bottom of the page.
  4. Add in the additional framework directories list an entry like `${pd}/custom_frameworks`.
  5. Add other resources to your project, for example you can have all your DITA content located inside the project folder.

6. You can then share the new project directory with other users. For example you can commit it to your version control system and have they update their working copy. When they open the customized project file in the **Project** view, the new document type becomes available in the list of Document Types.

- Deploy the framework/document type configuration as an add-on.

After your team members install the framework they can check in **Document Type Association** preferences page in the list of Document Types to see if the framework is present and if it appears before the bundled **DITA** framework (meaning that it has higher priority).

You can use the framework extension mechanism to customize lots of aspects of the **DITA** functionality in Oxygen. For example you can remove various elements from the content completion list: Document Type Extension Sharing *(on page 407)*.

Composing Author Actions

Suppose that each time you insert a **DITA** table in the **Author** visual editing mode, you want to always have the attributes `colsep="1" rowsep="1" frame="all"` set on it. The purpose of this post is to create a new **DITA**-specific action for inserting a table that invokes the current table insertion action and then forces those three attributes to be set on the **table** element. Here are some steps to accomplish this:

1. Follow the steps listed in this older blog post to create an extension of the **DITA** framework: Customizing the DITA Visual Editing Experience *(on page 245)*.

2. In the **Document Type Association** preferences page, edit the **DITA** framework extension you just created. Go to the **Author->Actions** tab and create a new action with the ID **change.table.colsep**. Use the predefined **ChangeAttributeOperation** to set the (colsep="1" attribute) on the closest table element. The custom action would look like this:

3. Based on the same idea, create two more new actions called "**change.table.rowsep**" and "**change.table.frame**" that will set the **rowsep="1"** and **frame="all"** attributes on the closest table, respectively.

4. Create a new action with the ID **insert.table.fixed.attributes** and use the predefined **ExecuteMultipleActionsOperation** to call 4 actions in a row, the original **insert.table** action ID that inserts the table, followed by the three action IDs that set various attribute values to the inserted table.

5. Go to the **Author->Toolbar** tab and in the "**Current actions**" panel, remove the current "**insert.table**" action and replace it with the **insert.table.fixed.attributes** action ID.

6. When editing a DITA topic, pressing the toolbar action for inserting a table should now call your custom action that sets those three attributes on the inserted table.

Implementing a Custom Author Action to Split a Table

Let's say you are editing XML documents belonging to a certain vocabulary (like **DITA**) for which there is a framework configuration available. The purpose of this post is to create a new custom Author action for splitting the current edited table in two based on the table row in which the cursor is located. The custom action will use Javascript to call Oxygen's API and accomplish this. Here are some steps to follow:

1. Follow the steps 1,3, 4 and 5 listed in this older blog post to create an extension of the **DITA** framework: Customizing the DITA Visual Editing Experience *(on page 245)*.

2. In the **Document Type Association** preferences page, edit the **DITA** framework extension you just created. Go to the **Author->Actions** tab and create a new action with the ID **split.table**. Use the

predefined **JSOperation** to invoke a custom Javascript code. The custom action definition would look like this:

3. Set as value to the **script** parameter of the operation the following **Javascript** code:

```
function doOperation(){

    current =

 authorAccess.getDocumentController().getNodeAtOffset(authorAccess.getEditorAccess().getCaretOffset());

    tableNode = null;

    rowNode = null;

    while(current != null) {

      if(tableNode == null && ("table".equals(current.getName()) || "informaltable".equals(current.getName())))

 {

        tableNode = current;

      }
```

```
      if(rowNode == null && ("row".equals(current.getName()) || "strow".equals(current.getName()))) {
        rowNode = current;
      }
      current = current.getParent();
    }
    if(tableNode != null && rowNode != null) {
      //Create a fragment starting from the row to the end of the table
      secondTable = authorAccess.getDocumentController().createDocumentFragment(rowNode.getStartOffset(),
tableNode.getEndOffset());
      //Delete the content from the first table.
      authorAccess.getDocumentController().delete(rowNode.getStartOffset(), tableNode.getEndOffset() - 1);
      //Insert the second table.
      authorAccess.getDocumentController().insertFragment(tableNode.getEndOffset() + 1, secondTable);
    }
}
```

4. Go to the **Author->Toolbar** tab and use the **Current actions** panel to add the action with ID **split.table** to the toolbar.
5. When editing a DITA topic, pressing the toolbar action for splitting the table should now call your custom action and split the current table.
6. You can add keyboard shortcuts for all custom actions either when defining them or from the Oxygen main menu **Preferences**->**Menu Shortcut Keys** page.

Adding a Custom Author Action to the Content Completion Window

Let's say you are editing XML documents belonging to a certain vocabulary (like **DITA**) for which there is a framework configuration available. The content completion window appears in the **Author** visual editing mode when you press the **ENTER** key and usually presents the set of XML elements allowed at the caret position. The same list of elements is present in the **Elements** view. These elements are usually proposed based on the DTD, XML or Relax NG vocabulary used to validate the XML document.

But there may be cases in which you want to customize the content completion items. There is a content completion configuration file which can be changed to allow for various use cases:

- Disallow a certain set of elements to appear in the content completion window.
- Declare a set of required child elements which should be inserted automatically when the parent element is inserted.
- Declare a set of required attributes which should be inserted when a certain element is inserted.

There are also a number of use cases not covered by the content completion filter file:

- You want to add a new item in the content completion filter which triggers a complex action. For example when the item is selected you show a dialog and ask the end user from some data, then insert a custom XML fragment based on that data the end user filled out.
- You want to replace an existing item in the content completion window with an item with the same name but which again triggers a more complex action.

Further in the blog post we'll go through the steps necessary to replace an item from the content completion window with a complex action. As an example for the **DITA** vocabulary when the **created** item is selected in the content completion window we want the inserted XML content to contain the current date like:

```
<created date="2020-03-26"/>
```

Here are some steps to follow:

1. Follow the steps 1,3, 4 and 5 listed in this older blog post to create an extension of the **DITA** framework: Customizing the DITA Visual Editing Experience *(on page 245)*.

2. In the **Document Type Association** preferences page, edit the **DITA** framework extension you just created. Go to the **Author->Actions** tab and create a new action with the ID `created.current.date`. Use the predefined **InsertFragmentOperation** to insert a small XML fragment. The XML fragment uses an editor variable which will be expanded to the current date. There are plenty of other editor variables *(on page 469)* to choose from.

   The custom action definition would look like this:

Action

ID: created.current.date

Name: created

Menu access key:

Description: Insert the DITA <created> element with the current date.

How to translate frameworks

Large icon (24x24): Browse

Small icon (16x16): Browse

Shortcut key: Clear

Enable platform-independent shortcut keys

Operations

1

Activation XPath: *oxy:allows-child-element*("**created**")

This XPath expression applies only to elements and attributes. More details

Operation: ro.sync.ecss.extensions.commons.operations.InsertFragmentOperation Choose

Arguments:

Name	Description	Type	Value
fragment	The fragment to be inserted	Fragment	<created date="${date(yyyy-MM-dd)}"/>
insertLocation	An XPath expression indicating the insert locat...	XPathExpress...	
insertPosition	The insert position relative to the node deter...	ConstantList	Inside as first child
schemaAware	Controlling if the insertion is schema aware or...	ConstantList	true
goToNextEditablePo...	After inserting the fragment, the first editable...	ConstantList	true

Operation priority: Increase Decrease

Evaluate activation XPath expressions even in read-only contexts

OK Cancel

There are plenty of other pre-defined operations which can be used from an Author action. For example a Javascript-based operation *(on page 417)*, there are some other examples of what such operations can do in a separate GitHub project.

3. Go to the **Author->Content Completion** tab, find the new action in the **Available Actions** list and add it to the **Current actions** list. You will be able to add it both in the content completion list and the **Elements** view. In the **Filter - Remove content completion items** list add an entry for `created` because we want to remove the original item and replace it with the action.
4. Save all your changes by using the **Ok** button both in the document type configuration dialog and in the **Preferences** dialog.
5. When editing a DITA topic in the **Author** visual editing mode you can press **ENTER** to show the content completion window, choose `created` and your custom Author action will be called instead of the original item.

How Special Paste works in Oxygen

If you've worked with one of the **XML** vocabularies for which Oxygen has out of the box support like **DITA**, **Docbook**, **TEI**, **XHTML** you've probably already used the support Oxygen has for converting content pasted in the application from external applications like **Microsoft Word**, **Excel** or from any **web browser**. This is a very useful feature for converting various types of content to XML because it preserves and converts styling, links, lists, tables and image references.

The feature relies on the fact that when copying content in the applications mentioned above, they set in the clipboard the HTML equivalent of the copied content. So all Oxygen has to do is clean up that HTML, make it wellformed XHTML and apply conversion XSLT stylesheets over it.

This support is not hardcoded and anybody who is developing an Oxygen framework customization *(on page 425)* for a certain XML vocabulary can provide conversion stylesheets for external pasted HTML content.

I will describe how this works for the **DITA** framework and you can do the same for yours. You can also use this information to modify the way in which smart paste works for the bundled framework configurations.

1. In the **Preferences->Document Type Association** page you can choose to edit (or extend) the **DITA** document type association.
2. In the **Extensions** tab the **Extensions bundle** implementation is set to **DITAExtensionsBundle** which resides in the **DITA** Java extensions archive **dita.jar**.

3. The **DITAExtensionsBundle** is an extension of the **ExtensionsBundle** API and it provides its own external object extension handler:

```
/**
 * @see ro.sync.ecss.extensions.api.ExtensionsBundle#createExternalObjectInsertionHandler()
 */
@Override
public AuthorExternalObjectInsertionHandler createExternalObjectInsertionHandler() {
  return new DITAExternalObjectInsertionHandler();
}
```

4. The **DITAExternalObjectInsertionHandler** extends the base class **AuthorExternalObjectInsertionHandler** and provides a reference to its specific conversion stylesheet:

```
/**
 * @see
ro.sync.ecss.extensions.api.AuthorExternalObjectInsertionHandler#getImporterStylesheetFileName(ro.sync.ecss.ex
tensions.api.AuthorAccess)
 */
@Override
protected String getImporterStylesheetFileName(AuthorAccess authorAccess) {
  return "xhtml2ditaDriver.xsl";
}
```

> **Note:**
> The **Extensions** tab also allows you to specify the external object insertion handler as a separate extension.

5. In the same **Document Type** edit dialog in the **Classpath** tab you will see that there is a reference to a framework-specific resources folder like:`${framework}/resources/`
6. If you look on disk in the DITA framework **resources** folder: `"OXYGEN_INSTALL_DIR \frameworks\dita\resources"` you will find the **xhtml2ditaDriver.xsl** stylesheet there. The stylesheet imports various other stylesheets which you could probably fully reuse and which apply various cleanups on HTML produced with MS Word. It also handles the conversion between the pasted HTML content and DITA so it is a good starting point, you can copy the entire set of XSLT stylesheets to your framework and use those as a starting point.

The Oxygen SDK (Part 1: Plugins)

During the last years we added a lot of **API** and extension points to Oxygen in order to allow for different customizations to the application. But our documentation is sometimes lacking. We mostly rely on **Javadoc** documentation and on **Java samples**. Here's some feedback we got at the last **Oxygen Users Meetup** in **Munich** this year:

> "*Too less information about frameworks, plugins, everything is spread over certain documents, webinars, etc . Please centralize these information in one form.*"

I will try to centralize these resources and add some useful links for people who want to start customizing Oxygen. First the difference between a **framework** and a **plugin**:

- A **plugin** can be used to customize the behavior of the entire application no matter what XML document is currently being edited.
- A **framework** configuration provides validation, content completion and editing support for a specific XML vocabulary.

Plugins:

A plugin is a folder containing a descriptor **plugin.xml** file and various other **JAR** libraries and resources.http://www.oxygenxml.com/doc/ug-oxygen/index.html#topics/preferences-plugins.html

Only the **standalone** version of Oxygen supports plugins. The **Eclipse Plugin** version of Oxygen is itself a plugin and can be customized by adding a plugin in the Eclipse workbench which depends on the **Oxygen Eclipse plugin**. Despite of this, most of the API is common.

The plugin can be deployed either by copying it to the **plugins** folder of an Oxygen installation:https://www.oxygenxml.com/doc/ug-editor/topics/howto-install-plugins.html or by deploying it as an **add-on**:http://www.oxygenxml.com/doc/ug-oxygen/index.html#tasks/deploying-addons.html

The **Oxygen Plugins SDK**:https://www.oxygenxml.com/oxygen_sdk.html#Developer_Plugins contains Java sources and Javadoc for all the API accessible from a plugin. The Plugins SDK also contains sample plugins and their Java code. This Java code should be very helpful to get you started and to show how various API can be used.

Although there are many types of plugins:http://www.oxygenxml.com/doc/ug-oxygen/index.html#topics/pluginTypes.html the most useful plugin extension type is the "**Workspace Access**" extension type:http://www.oxygenxml.com/doc/ug-oxygen/index.html#concepts/workspace-access-plugin.html This kind of plugin allows you to use the API and add or remove toolbar and main menu buttons, add custom views

and toolbars. It also allows you to access and control/make changes to the XML documents opened in the workbench.

As an example, all full-featured integrations which have been created to connect Oxygen with a specific **CMS** or remote repository use a combination of "Workspace Access" and "Custom Protocol" plugin:http://www.oxygenxml.com/doc/ug-oxygen/index.html#topics/howto-cms-plugin.html

You can create **automated tests** for your plugins:http://www.oxygenxml.com/doc/ug-oxygen/index.html#topics/automated-tests.html and even **debug their functionality**:http://www.oxygenxml.com/doc/ug-oxygen/index.html?q=/doc/ug-oxygen/topics/debug-plugin.html

The Oxygen SDK (Part 2: Frameworks)

This is the second part of a blog post I started some time ago:

The Oxygen SDK (Part 1: Plugins) *(on page 424)*.

There are two ways of customizing the application, by implementing a plugin or by implementing a framework:

https://www.oxygenxml.com/doc/ug-editor/topics/api_faq_plugin_framework_difference.html

A **framework** configuration provides validation, content completion and editing support for a certain XML vocabulary.

If you are already using Oxygen for editing **DITA**, **Docbook**, **XHTML** or **TEI** documents you may notice that **Oxygen** knows how to validate these vocabularies and that it can propose content completion entries while you are editing. Also when you are editing in the Author visual editing mode you have lots of custom vocabulary-specific toolbar buttons which can be used to insert links, images, to manipulate tables and so on. This happens because each **Oxygen** installation comes with pre-bundled framework configurations for certain XML vocabularies that we consider to be more important for our users.

Knowing how to create and modify a framework/document type association configuration will benefit you in two ways:

1. Create your own framework which adds editing support to Oxygen for certain specific XML vocabularies and then distribute it to your team.
2. Customize an existing framework bundled with the installation (DITA, Docbook, etc) and change certain behaviors in it.

Our user manual contains a special step by step tutorial which explains how a new framework configuration (document type association) can be created and configured:

https://www.oxygenxml.com/doc/ug-editor/topics/authoring_customization.html

The Oxygen **Preferences->Document Type Association** page lists all detected frameworks (document type associations). Usually looking inside one of the pre-configured document type associations (eg: **DITA**) is a good place to start exploring what such a customization contains:

1. **Association rules** - when one of these rules matches the opened XML document, Oxygen will associate it with the current document type association. The rules are pretty simple to compose, they refer to a certain root name, namespace, certain attributes set on the root and so on.
2. **Schema** - specifies a grammar to be used to providing validation and content completion if the opened XML document does not refer directly to any particular gramar.
3. **Classpath** - a list of JAR libraries which contain Java extensions for this specific framework.
4. **Author** - contains all necessary support for editing the XML in the Author visual editing mode:
    - **CSS** - one or more CSS files to be used when rendering the XML. If you define alternate CSSs, you will be able to switch between them when editing. The user manual contains a list of supported CSS features and additional available extensions.
    - **Actions** - a list of actions specific for modifying the edited content. An action has a name, description, icons and shortcut key. It also has one or more activation contexts which depending on an XPath expression enable a certain operation be executed. A fair amount of basic operations are already available but you can create your custom operations.
    - **Menu**, **Contextual menu** and **Toolbar** - you can easily mount defined actions to the main document type menu, to the contextual menu or to the special Author toolbar(s).
    - **Content Completion** - add defined actions to the content completion window (shown when ENTER is pressed in the Author editor mode) or remove existing entries from the content completion window. You can for example replace some of the insert suggestions given by the association grammar with your own custom actions.
5. **Templates** - points to folders which contain new file templates for this particular framework. These new file templates will be shown in the **New** wizard dialog.
6. **Catalogs** contains a list of XML catalogs which will be used to indirectly solve various references (like references to schemas or other XML documents).
7. **Transformation** may contains a predefined list of transformation scenarios which are available when you want to publish your opened XML document to various output formats.
8. **Validation** may contain a predefined list of validation scenarios which are used to add complex multi-stage validation (with multiple engines) for the XML documents matching the document type association.

9. **Extensions** - contains implementations of the available Java extensions which are used to provide further functionality for editing in the Author visual editing mode. Here's what some of the extensions do:
    - **AuthorExtensionStateListener** - provides a way to be notified when the XML was opened in the Author editing mode. You can then add all kinds of listeners and react to edit events done by the user. For example add a modification listener, send the edited content to an external spell checker engine and then add highlights in the content on invalid constructs.
    - **AuthorExternalObjectInsertionHandler** - reacts to drag and drop and copy/paste events containing with HTML content or resources. In the case of DITA for example this handler is responsible of the automatic conversion of HTML pasted from the browser to DITA content.
    - **SchemaManagerFilter** - filter and modify the insertion items detected from the associated grammar when editing XML content. For example even if the schema proposes certain elements as valid insertions at the caret offset, you can filter out and restrict the suggestions given by the associated schema (grammar).
    - **StylesFilter** - take control over the rendering styles for each node by adding this layer of Java customization over the styles provided by the associated CSSs.
    - **AuthorSchemaAwareEditingHandler** - handle special editing cases and provide fallbacks which keep the document in a valid state. For example if the user starts typing text between two paragraphs, the handler can automatically create a new paragraph.

You can create **automated tests** for your frameworks:

http://www.oxygenxml.com/doc/ug-oxygen/index.html#topics/automated-tests.html

and even **debug their functionality**:

https://www.oxygenxml.com/doc/ug-editor/topics/debug-sdk.html

Your First Oxygen Add-on

A framework configuration *(on page 407)* provides validation, content completion, and visual editing support for a certain XML vocabulary.

Existing framework configurations that come bundled with **Oxygen** can be extended and modified. For example, the **DITA** framework configuration that adds support for editing **DITA** documents can be modified to remove existing actions, add custom actions, or modify many other behaviors: Document Type Extension Sharing *(on page 407)*.

One way of distributing framework configurations is by using Oxygen's **add-on** support: https://www.oxygenxml.com/addons.html. An add-on configuration is usually composed of only two files:

1. A ZIP file containing the zipped framework folder.
2. An add-on XML configuration file that references the zipped framework folder and defines various properties of the add-on. A small example can be seen below:

```
<xt:extensions xmlns:xt="http://www.oxygenxml.com/ns/extension"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.oxygenxml.com/ns/extension
 http://www.oxygenxml.com/ns/extension/extensions.xsd">
    <xt:extension id="dita-extension">
        <xt:location href="dita_extension.zip"/>
        <xt:version>1.0.0</xt:version>
        <xt:oxy_version>18.0+</xt:oxy_version>
        <xt:type>framework</xt:type>
        <xt:author>Radu Coravu</xt:author>
        <xt:name>DITA Extension which removes the bold action</xt:name>
        <xt:description xmlns="http://www.w3.org/1999/xhtml">
            This extension removes the bold action
        </xt:description>
        <xt:license>
<![CDATA[
Everybody's welcomed to use this
]]></xt:license>
    </xt:extension>
</xt:extensions>
```

   In the add-on descriptor file you usually need to specify the following :
   - The ID of the extension.
   - The version of the add-on.
   - The minimum supported Oxygen version.
   - Author name.
   - Name and description of add-on.
   - Licensing details.

Changing the add-on version in the XML file will trigger notifications to the end user to install the new add-on. Therefore, you can modify the framework customization in a common location and change the add-on version to signal all end-users who have installed the add-on that a new version is available. More details about packing an add-on can be found here:https://www.oxygenxml.com/doc/ug-editor/topics/packing-and-deploying-addons.html#packing-and-deploying-addons.

If you want to test the small add-on I exemplified, you can find the sample add-on resources here:https://www.oxygenxml.com/forum/files/sample-add-on.zip.

Installing an add-on is straight-forward using the **Install new add-ons** dialog box from the Oxygen **Help** menu. You can add a reference to the add-on **XML descriptor** file that can reside either on the local hard drive or on a web server, and Oxygen will present all available add-ons: https://www.oxygenxml.com/doc/ug-editor/topics/installing-and-updating-add-ons.html

The add-ons support can also be used to install Oxygen plugins *(on page 424)*. Some of the public hosted Oxygen plugins and frameworks are also available as add-ons: Public hosted Oxygen Plugin and Framework Projects *(on page 411)*.

In conclusion, if you want to share Oxygen customizations with others, you should try the add-on approach, it is a better approach to manually distributing frameworks and plugins.

Oxygen Add-ons Overview

Oxygen XML Editor can be extended to support new functionalities and vocabularies. These extensions can be either plugins or frameworks: https://www.oxygenxml.com/doc/ug-editor/dev_guide/introduction-dev-guide.html.

This blog post is intended as a small overview of Oxygen XML Editor add-ons, what they are, sample add-ons implemented by third parties, sample add-ons implemented by us and where we want to go from here.

Short History of Frameworks Support in Oxygen XML Editor

- The concept of framework appeared around Oxygen XML Editor version 9.

  A framework (or document type) is a means to group together resources for editing, rendering and validating a specific XML vocabulary: https://www.oxygenxml.com/doc/ug-editor/glossary/framework.html.

- Originally frameworks could be manually installed by being copied to the `OXYGEN_INSTALL_DIR\frameworks` folder or by setting them up in the Preferences page.

  Around version 14 we added support to pack plugins and frameworks as add-ons *(on page 427)* and thus automate the installation process.

- In version 16 we added support to extend a base framework. Such an extension can be afterwards be bundled as an add-on.
- A sample framework extension add-on which adds custom visual display support can be found here: DITA Topics as Slides.

Short History of Plugin Support in Oxygen XML Editor

- The initial support for plugins was primitive, it allowed adding a contextual menu action in the Text editing mode for manipulating the selected content: https://www.oxygenxml.com/doc/ug-editor/topics/text-editing-mode-plugins.html.
- In version 11.2 the Workspace Access plugin type was added: https://www.oxygenxml.com/doc/ug-editor/topics/workspace-access-plugin.html.

  Workspace Access plugins allow you to:
  - Add custom toolbar and menu actions.
  - Add custom side views.
  - Add various behaviors depending on various events which occur in the application:
    - Open, Save, Close documents.
    - Switch between document.
    - Open or close the application.

  Such plugins can be implemented either in Java or Javascript (since version 17.1).
- Originally plugins could be manually installed by being copied to the `OXYGEN_INSTALL_DIR\plugins`. Around version 14 we added support to pack plugins and frameworks as add-ons *(on page 427)* and thus automate the installation process.

Public Hosted Plugins and Frameworks

There is an older blog post with an up to date list of Public hosted Oxygen Plugin and Framework Projects *(on page 411)*.

Private Hosted Plugins and Frameworks

Content Management Systems like Astoria, Bluestream, Ixiasoft, SDL have implemented plugins which allow Oxygen XML Editor to connect to their remote storage servers.

There are also lots of privately used frameworks used for editing various varieties of XML documents inside companies.

Why are we (the Oxygen XML Editor team) working on add-ons?

- Open source add-ons are useful as samples for third parties.
- We can work with external collaborators on add-ons.
- Add-ons can be released at a much faster pace.
- Interns who work for us for a couple of months can implement useful functionality without making changes to the application main code base.
- We can work on add-ons before Oxygen XML Editor releases when we enter a code freeze stage.
- We can use our own API and abandon the mindset of regarding the main application as a monolith.
- Sincerely, working on add-ons is more FUN.

Plugins and Frameworks Developed by Oxygen XML Editor

Samples:

- Sample Javascript-based plugins: https://github.com/oxygenxml/wsaccess-javascript-sample-plugins.
    - Apply XSLT filter on open/save.
    - Add custom editor variable resolver.
    - Impose options.
    - Add custom image renderer for Author visual editing mode.
    - Add custom validation problems filter.
- Sample Workspace-access plugin Maven based: https://github.com/oxygenxml/sample-plugin-workspace-access
- User interface filter plugin: https://github.com/oxygenxml/oxygen-components-filter-plugin

Useful add-ons with public sources:

- Git support add-on: https://github.com/oxygenxml/oxygen-git-plugin
- Batch Resources Convertor add-on: https://github.com/oxygenxml/oxygen-resources-converter
- DocBook Validation add-on: https://github.com/oxygenxml/oxygen-docbook-validate-check-completeness
- Add-on for connecting to any CMIS-enabled storage server (like Alfresco): https://github.com/oxygenxml/oxygen-cmis-plugin.
- DITA-specific add-ons:
    - Translation helper: https://github.com/oxygenxml/oxygen-dita-translation-package-builder
    - Prolog updater: https://github.com/oxygenxml/oxygen-dita-prolog-updater
    - Outgoing References View: https://github.com/oxygenxml/oxygen-dita-outgoing-references-view

- Support for rendering various image formats in the Author visual editing mode:
  - EMF: https://github.com/oxygenxml/oxygenxml.emf.support
  - CGM: https://github.com/oxygenxml/oxygenxml.cgm.support
  - PDF: https://github.com/oxygenxml/oxygen-pdf-image
  - LaTex: https://github.com/oxygenxml/wsaccess-javascript-sample-plugins/tree/master/latex-images-support
  - PlantUML: https://github.com/oxygenxml/wsaccess-javascript-sample-plugins/tree/master/plantuml-images-support

Useful add-ons with private repositories (can be installed from application default add-on update site):

- Content Fusion Oxygen add-on - bundled with Oxygen.
- WebHelp Feedback Oxygen add-on.
- Emmet plugin for Oxygen.

Using Frameworks and Plugins in the WebAuthor in-browser editor

The WebAuthor in-browser editing tool reuses on the server most of the internal architecture of an Oxygen XML Editor standalone version:

- The WebAuthor can use for editing XML vocabularies the same framework configurations used in Oxygen XML Editor: https://www.oxygenxml.com/doc/ug-waCustom/topics/customizing_frameworks.html
- Plugins can be implemented to connect the WebAuthor to various servers.

Where can we go from here?

Possible future plugins we can work on:

- Support for Google Drive (already available in WebAuthor)
- Support for Perforce (already available in WebAuthor)
- Use Grammarly for spell check (no API available on Grammarly side)
- Presentation mode plugin (increase fonts, disable certain settings).
- Context side view plugin using XPath with support for favorite expressions.
- Plugin which highlights various characters or text fragments in the content.

Possible new API enhancements:

- Add support for a plugin to contribute a custom editing perspective (toolbars/views arrangement).

Besides the possible enhancements listed below, we want your feedback. So are you using Oxygen add-ons? Are you developing such add-ons? What do you need from us to better support your use cases?

Adding CALS-table related functionality to your custom Oxygen framework

Oxygen comes with full support for **CALS** tables in **DITA** and **Docbook** documents, meaning that you can easily make selections, resize columns, and invoke special actions like insert or delete rows and columns, join, or split cells. You can also easily customize tables properties such as alignments, separators, and table borders. But what if you are editing documents from other vocabularies, containing tables with **CALS** structure? What you can do to obtain the same table editing features?

One option is to use a plugin that adds CALS table editing support along with specific table related actions: https://github.com/oxygenxml/web-author-sample-plugins/tree/master/web-author-CALS-table-plugin.

Another option is to customize your framework configuration:

Let's suppose that you already created an Oxygen framework for your documents vocabulary (if you need further information about frameworks, see The Oxygen SDK (Part 2: Frameworks) *(on page 425)*). What we want to obtain next is to extract all the CALS tables related support from **Docbook** framework and add it to your custom framework. Why **Docbook** and not **DITA** as reference? Because the **DITA** customization is based on the "class" attribute checking while the **Docbook** one is more general (element-name oriented).

1. **Table rendering**
   a. copy `[oXygen_install_dir]\frameworks\docbook\css\cals_table.css` and `[oXygen_install_dir]\frameworks\docbook\css\html_cals_table.css` in your framework css directory
   b. in the `Document Type` edit dialog, `Author` tab, `CSS` sub-tab, add the `${framework}/css/cals_table.css` entry to the list of the CSSs

   In this way the table will be rendered properly in Author mode and the following features will be available:

   - select cells (`CTRL+click`), rows (click before the row), columns (click on top of the row), tables (click in the left-up corner of the table)
   - resize table column
2. **Create table actions**

Here are the table-related actions implemented in Docbook that you can implement also in you framework:

- Expand colspecs
- Insert/Delete Rows
- Insert/Delete Columns
- Table Properties
- Join cells
- Split cell
- Colspecs (both table.collapse.colspec and table.expand.colspec)

a. Copy `[oXygen_installation_dir]\frameworks\docbook\docbook.jar` in your framework directory (this jar contains all the table operations classes needed to create table actions). In `Document type` edit dialog go to `Classpath` tab and add the docbook.jar from your framework directory to the list of paths. In this way you have access to all table operations from your framework.

b. For every table action you have to define a corresponding action in the `Document type` configuration dialog. Here are the details: https://www.oxygenxml.com/doc/ug-editor/#topics/the-action-dialog.html. You can look at the corresponding Docbook action to see the properties (id, name, icons, the conditions that enables it, the specific operation for CALS tables).

Once you created these actions you can add them to the UI.

3. **Add table actions to menu, toolbar and contextual menu**

a. To add an action to the menu go to `Document Type` configuration dialog, `Author` tab, `Menu` sub-tab, select the action from `Available actions` section and add it to the `Current action` section.

b. To add an action to the contextual menu go to `Document Type` configuration dialog, `Author` tab, `Contextual menu` sub-tab, select the action from `Available actions` section and add it to the `Current action` section.

c. To add an action to the toolbar go to `Document Type` configuration dialog, `Author` tab, `Toolbar` sub-tab, select the action from `Available actions` section and add it to the `Current action` section.

If all goes well, your custom framework which uses the standard CALS table naming mechanism will properly handle tables, both for display and for table-related operations.

Convert Code Templates to External Author Actions

Describes how to bring code templates into a framework as author actions.

Code templates are code fragments that can be inserted quickly at the current editing position. Author actions that are available in the Author mode offer a lot more flexibility. For example, you can decide what fragment to insert based on the current context. If you want to create a new framework or customize an existing framework *(on page 246)*, then it makes sense to convert code templates to author actions. You can then contribute them to the content completion window. You can also define the framework using a Framework extension script.

1. Create a framework or extend an existing framework using a Framework extension script.
2. Go to **Options > Preferences > Code Templates** and Export the code templates to a file named `code-templates.xml`.
3. Create an XSLT file with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"

    xmlns:xs="http://www.w3.org/2001/XMLSchema"

    exclude-result-prefixes="xs"

    version="3.0">

    <xsl:output method="xml"/>

    <xsl:template match="//codeTemplateItem[field[@name='contentType']/String/text() = 'text/xml']">

        <xsl:variable name="actionName" select="field[@name='renderString']/String/text()"/>

        <xsl:result-document href="externalAuthorActions/{$actionName}.xml">

            <a:authorAction xmlns:a="http://www.oxygenxml.com/ns/author/external-action"

                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

                xsi:schemaLocation="http://www.oxygenxml.com/ns/author/external-action

 http://www.oxygenxml.com/ns/author/external-action/authorAction.xsd"

                id="{$actionName}">

                <a:name><xsl:value-of select="$actionName"/></a:name>


  <a:description><xsl:value-of select="field[@name='descriptionString']/String/text()"/></a:description>

                <a:operations>

                    <a:operation id="InsertFragmentOperation">

                        <a:xpathCondition>true()</a:xpathCondition>

                        <a:arguments>

                            <a:argument name="fragment">
```

```
                                <!-- Try parsing the fragment. if it succeeds, we basically unwrap it from the
 CDATA. -->
                                <xsl:try

 select="parse-xml-fragment(field[@name='unparsedInsertString']/String/text())">
                                    <xsl:catch>
                                        <!-- Parsing failed. It might have unresolved entities. Keep it in a
 CDATA. -->
                                        <xsl:text disable-output-escaping="yes">&lt;![CDATA[</xsl:text>

 <xsl:value-of disable-output-escaping="true" select="field[@name='unparsedInsertString']/String/text()"
 ></xsl:value-of>
                                        <xsl:text disable-output-escaping="yes">]]&gt;</xsl:text>
                                    </xsl:catch>
                                </xsl:try>
                            </a:argument>
                        </a:arguments>
                    </a:operation>
                </a:operations>
                <a:accessKey/>
            </a:authorAction>
        </xsl:result-document>
    </xsl:template>

    <xsl:template match="text()"/>
</xsl:stylesheet>
```

4. Create an XSLT Transformation scenario and apply the XSLT over the `code-templates.xml` file.
5. Go to the directory of `code-templates.xml` and copy the generated `externalAuthorActions` to the framework directory (the directory with the framework extension script).
6. Create another XSLT with the following content, then create an XSLT Transformation scenario and apply it over the `code-templates.xml` file.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    exclude-result-prefixes="xs"
    version="3.0">
```

```
    <xsl:output method="xml" omit-xml-declaration="yes" indent="true"></xsl:output>

    <xsl:output method="xml"/>

    <xsl:template match="/">

        <contentCompletion>

            <authorActions>

                <xsl:apply-templates/>

            </authorActions>

        </contentCompletion>

    </xsl:template>

    <xsl:template match="//codeTemplateItem[field[@name='contentType']/String/text() = 'text/xml']">

        <xsl:variable name="actionName" select="field[@name='renderString']/String/text()"/>


        <addAction id="{$actionName}" inCCWindow="true"/>

    </xsl:template>


    <xsl:template match="text()"/>

</xsl:stylesheet>
```

7. Copy the resulting `<contentCompletion>` into the Framework extension script root element.

There is a new external author action for each initial code template. All actions are contributed to the content completion window.

Speed Up Oxygen XML Web Author Customization (use Web Author Test Server Add-on)

Are you working on any of these tasks:

- Starting a new Oxygen XML Web Author customization project?
- Testing if an existing Oxygen XML Editor framework works with Web Author?
- Making quick updates to a framework already used in a production Web Author server?

The **Web Author Test Server Add-on** is designed to make your work easier and faster.

This **Oxygen XML Editor add-on** lets you quickly preview and edit XML documents in a local **Web Author Test Server**. It uses the same framework you are working on. You do not need to package, deploy, or do any extra setup.

Once installed, you'll see a new button in the toolbar, called **Start Web Author Server** that:

- Launches a local Web Author Test Server automatically.
- Opens your current document in it.
- Loads the associated framework instantly.

Watch this video for more details about how to install it and to see how efficient your customization workflow can become when you will use it:

https://www.youtube.com/embed/u3WhXzxdc28

Note:
The update site shown in the video, that you can use to install the add-on, is: https://www.oxygenxml.com/InstData/Addons/optional/updateSite.xml

Tip:
You can drag the following **Install** button and drop it into the main editor in Oxygen XML Editor to quickly initiate the installation process: Install

Chapter 8. Development

How to Use Text Files as Primary Input for XSLT in Oxygen XML Editor

When working with Oxygen XML Editor, many users are accustomed to transforming XML or JSON documents using XSLT or XQuery. But what if you want to process a plain text file—perhaps a log, configuration, or other non-XML data—using XSLT? With Oxygen's powerful transformation scenarios and XSLT 3.0's new features, you can!

In this article, I'll show you how you can use a text file as the primary input for an XSLT transformation inside Oxygen XML Editor, even if the file isn't XML or JSON.

Background: Transformations in Oxygen XML Editor

Oxygen provides a robust set of features for transforming XML and JSON documents (XML Transformation with XSLT/XQuery, JSON Transformation with XSLT/XQuery). Users typically create transformation scenarios to automate these actions.

Traditionally, XSLT requires an XML or JSON input, but as of **XSLT 3.0**, you can use the **initial template** feature. This allows you to define a `xsl:initial-template`. entry point for your transformation, with no input document required. This feature opens up new possibilities—including processing text files!

Step-by-Step: Transforming Text Files with XSLT in Oxygen

1. Create an XSLT using an Initial Template

Write an XSLT 3.0 stylesheet that declares an initial template with `xsl:initial-template`. This template will serve as the main entry point.

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="3.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">


    <!-- PARAMETERS • $input-uri – location of person.txt • $delim – field delimiter -->

    <xsl:param name="input-uri" select="'person.txt'"/>   <!-- default -->

    <xsl:param name="delim"      select="'\|'" />         <!-- "|" escaped for regex -->


    <!-- result -->

    <xsl:output method="xml" indent="yes"/>


    <!-- entry point (initial template) -->
```

```
    <xsl:template name="xsl:initial-template">
        <persons>
            <!-- read whole file, split it into separate lines -->
            <xsl:variable name="lines"
                select="tokenize(unparsed-text($input-uri), '\r?\n')" />

            <!-- iterate over the lines, skipping empties -->
            <xsl:for-each select="$lines[normalize-space()]">
                <!-- tokenize the current line into its fields -->
                <xsl:variable name="f" select="tokenize(., $delim)" />

                <person>
                    <id>        <xsl:value-of select="$f[1]"/></id>
                    <firstName> <xsl:value-of select="$f[2]"/></firstName>
                    <lastName>  <xsl:value-of select="$f[3]"/></lastName>
                    <age>       <xsl:value-of select="$f[4]"/></age>
                    <city>      <xsl:value-of select="$f[5]"/></city>
                </person>
            </xsl:for-each>
        </persons>
    </xsl:template>
</xsl:stylesheet>
```

2. Define the Text File Location via Parameter

Instead of hardcoding the file path, make the text file's location a parameter (as shown above, `$input-uri`).

```
id | firstName | lastName | age | city
001|John      |Doe       |29  |New York
002|Jane      |Smith     |34  |London
003|Paul      |Adams     |42  |Sydney
```

3. Create a Transformation Scenario

- Go to **Transformation Scenarios** in Oxygen.
- Create a **new XSLT transformation scenario**.
- Specify your XSLT as the stylesheet.
- In the parameters tab, add:

  - **Name:** `input-uri`
  - **Value:** `${currentFileURL}`

The `${currentFileURL}` variable automatically passes the path of your currently edited or selected file to the stylesheet.

4. Associate and Apply the Scenario

You can:

- **Open the text file** in the editor and press the Transform button in the Transformation view.
- **Select multiple text files** in the Project View, right-click, and run the scenario on all of them.
- **Automate batch processing** The scenario will automatically process each selected text file, regardless of format, as long as your stylesheet can handle it.

Now, your XSLT can process non-XML files using the full power of the XSLT 3.0 language!

Use Cases

- Convert plain text lists into XML
- Parse configuration logs
- Normalize or restructure CSV or tab-delimited files (with some tweaking)
- Batch transform multiple text documents from the Project view

Learn More

- XSLT 3.0 Initial Template in Oxygen
- XML Transformation with XSLT/XQuery
- JSON Transformation with XSLT/XQuery

Conclusion

With Oxygen XML Editor and XSLT 3.0's initial template mechanism, you're no longer limited to XML or JSON data. Whether you're transforming text logs, flattening CSV, or assembling custom data workflows, you can harness the flexibility of scenarios, parameters, and XSLT for all your file types.

Give it a try and unlock new transformation possibilities in Oxygen!

Editing Complex XML Schemas with Oxygen XML Editor

Using Main Files and Advanced Tools

As your XML-based projects grow in complexity, managing and editing large sets of interdependent XML Schema (XSD) files can become a challenge. Luckily, **Oxygen XML Editor** is designed with robust, productivity-boosting features to help you handle large schema structures with ease and accuracy. In this blog post, we'll dive into key techniques—including main files support and built-in schema exploration tools—to streamline your schema development workflow.

Why Main Files Are Important in Complex Schema Projects

In big projects, XML Schemas are rarely monolithic. Instead, they're modular, often spanning dozens or hundreds of XSD files that `import` or `include` each other. Editing such a network of modules without clear context can lead to validation errors, missed dependencies, and incomplete code assist.

Main File Support: Your Project's Entry Point

Oxygen's **main files support** feature is a game-changer for complex schema structures. By defining a *main* XSD file for your project, you:

- **Ensure accurate validation.** Each module gets validated in the context of the full schema set—not just in isolation.
- **Unlock intelligent content completion.** Autocomplete and validation suggestions take the entire schema hierarchy into account, not just the current file.
- **Simplify component-wide refactoring and navigation.** Changes like renaming elements, searching for references, or finding definitions are made in the context of the whole schema, enabling robust, error-free edits.

Read more about main file support in Oxygen's official documentation.

How to Set a Main XSD File in Oxygen

1. Use Oxygen to automatically detect the main files in your project. Right-click the project root and select **Detect Main Files**.
2. Add main files manually. Right-click your main XSD files in the project and select **Add to Main Files**.
3. Oxygen will now use this as the root context for validation, content assist, and other schema-aware tasks.

To see how the main file workflow works, watch this video demo.

Boost Schema Development with Helper Views

Oxygen XML Editor comes with powerful side views that provide insight and easy navigation within your schema ecosystem:

1. Component Dependencies View

This tool quickly shows relationships between schema components (elements, types, groups, etc.). For any selected component, you can visualize where it's used or which files it references. This is invaluable for impact analysis before making changes.

→ Learn more: Component Dependencies

2. Referenced/Dependent Resources View

This view helps you understand how schemas are included or imported. Whether you need a big-picture view or want to find circular dependencies, this tool shows the full chain of files involved.

→ Learn more: Referenced/Dependent Resources

3. Schema Refactoring and Search

Oxygen also provides powerful search and refactoring tools. For example, you can:

- Quickly find all references to a type or element in the project.
- Rename elements and types safely, updating all their uses.
- Extract schema components into new files or incorporate them from other modules.

Best Practices for Large XSD Structures in Oxygen

- **Break schemas into modules:** Organize schemas into logical parts based on functionality or domain.
- **Set one main entry point:** Use main file support to let Oxygen understand your entire schema.
- **Leverage helper views:** Use Component Dependencies and Resource Hierarchy to understand and maintain your schema landscape.

- **Refactor with confidence:** Use Oxygen's search and refactor features to safely evolve your schemas.
- **Validate often:** Take advantage of context-aware validation to catch integration errors early.

Conclusion

For those working with complex, multi-file XML Schema projects, Oxygen XML Editor's main file support and advanced schema navigation tools are indispensable. They transform schema management from a source of dread into a streamlined, error-resistant, and even enjoyable process.

Ready to bring order to your XML schemas?

Explore these features in your next Oxygen-powered project, and let structure work for you rather than against you!

References:

- Editing XML Schema Main Files in Context - Oxygen User Guide
- Component Dependencies View
- Referenced/Dependent Resources View

Video Demo: Oxygen Main File Support Explained

What tools or techniques do you use when working with large XML Schemas? Share your thoughts in the comments!

Chapter 9. Miscellaneous

ISO 27001:2022 Certification Announcement

Syncro Soft Achieves ISO 27001:2022 Certification, Reinforcing Our Commitment to Information Security.

At Syncro Soft, we are happy to announce that our company has successfully achieved ISO 27001:2022 certification, the most widely recognized international standard for information security management systems (ISMS).

You can view our ISO 27001:2022 certificate here.

What is ISO 27001:2022?

The ISO 27001:2022 standard provides a comprehensive framework for implementing, maintaining, and continually improving an ISMS within an organization. It establishes best practices for identifying, assessing, and mitigating potential risks to information security, ensuring the confidentiality, integrity, and availability of critical data.

By adhering to the rigorous requirements set forth by ISO 27001:2022, Syncro Soft has demonstrated its ability to effectively manage and protect sensitive information, safeguarding the interests of our clients and stakeholders.

The Benefits of ISO 27001:2022 Certification

Achieving ISO 27001:2022 certification is a significant milestone for Syncro Soft, as it underscores our commitment to excellence in information security and brings numerous benefits to our customers and partners:

1. **Enhanced Data Protection**: With robust security controls and processes in place, our clients can have confidence that their sensitive information is protected against potential threats, such as unauthorized access, cyber-attacks, or data breaches.
2. **Improved Risk Management**: The ISO 27001:2022 framework enables us to identify, assess, and mitigate risks more effectively, ensuring that our security measures are proactive and responsive to evolving threats.
3. **Compliance with Regulations**: By adhering to the stringent requirements of ISO 27001:2022, Syncro Soft demonstrates compliance with various data protection regulations, providing our clients with peace of mind and a competitive advantage in their respective industries.

4. **Continuous Improvement**: The certification process mandates regular audits and reviews, fostering a culture of continuous improvement within our organization. This ensures that our security practices remain up-to-date and aligned with industry best practices.
5. **Competitive Edge**: In today's increasingly digital landscape, where data security is of paramount importance, our ISO 27001:2022 certification sets us apart as a trusted and reliable partner, capable of delivering secure and compliant software solutions.

At Syncro Soft, we remain committed to upholding the highest standards of information security and protecting the interests of our clients. The achievement of ISO 27001:2022 certification is a significant milestone in our journey, and we will continue to invest in security measures that ensure the confidentiality, integrity, and availability of sensitive data entrusted to us.

For more information about our information security practices and ISO 27001:2022 certification, please contact us.

Oxygen XML Users Meetup @ XML Prague 2024: A Deep Dive into XML Technologies and Innovations

At the start of the summer, the beautiful city of Prague will welcome, once more, the XML Prague conference (June 6-8, 2024) - a unique opportunity for developers, markup enthusiasts, information managers, and students to come together in person.

Continuing its tradition, the Oxygen XML team will be present at the conference to host the highly anticipated **Oxygen XML Users Meetup**. This half-day event, taking place on **June 6th**, promises to be packed with insightful sessions that cater to both seasoned XML users and newcomers alike, straight from the dedicated individuals behind the Oxygen XML suite of products.

In similar fashion to its past editions, our presenters prepared an agenda full of eye-opening sessions regarding the cutting-edge solutions and tools that we've been tirelessly developing. Here is a preview of what attendees can expect:

Oxygen Short History and Tools Overview - *presented by the Oxygen XML team*

Kicking off the meetup at 15:00 (EEST), this session will take attendees through a brief history of Oxygen XML Editor's development and an overview of the tools it offers. It's a great opportunity to see how Oxygen has evolved over the years and what it now offers to enhance your XML editing capabilities. This will benefit attendees by providing a solid foundation in the tools available for any XML editing needs.

Oxygen AI Positron Assistant - *presented by Radu Coravu*

Explore the cutting-edge integration of AI within the Oxygen suite in this session. Learn how AI is being utilized to streamline XML editing and content management processes. Attendees will benefit from firsthand demonstrations of new functionalities, which are designed to significantly enhance productivity and user experience.

XPath AI Extension Functions - *presented by Octavian Nadolu*

This technical session will delve into the innovative use of AI in XPath functions, illustrating how these can simplify complex XML operations. It's a perfect session for those looking to enhance their technical skills and understand the practical applications of AI in XML workflows.

Oxygen Content Fusion Projects - *presented by Bogdan Dumitru*

Discover new features in Oxygen Content Fusion that facilitate project management and publishing directly from Git repositories. This session is particularly beneficial for teams looking for robust collaboration tools in document editing and publishing.

The New JSON Editor - *presented by Mircea Enachescu*

With the introduction of the new Oxygen JSON editor, this session will showcase its diverse capabilities, including various editing modes and comprehensive validation tools. Attendees will gain insights into how these features can be leveraged to improve data handling and integration in their projects.

YAML Support in Oxygen - *presented by Mircea Enachescu*

Learn about the latest improvements and features for YAML editing within the Oxygen framework. This session is essential for those who work with YAML and XML, providing tools that ensure greater consistency and error reduction in document management.

DITA XML Editing, Collaboration and Publishing Overview - *presented by Radu Coravu*

This session focuses on the comprehensive capabilities of Oxygen XML Editor for DITA projects, from editing to publishing. It's an excellent opportunity to learn how to maximize the efficiency of DITA content management and collaboration.

Oxygen Roadmap and Feedback - *presented by Octavian Nadolu*

Conclude the day with a forward-looking session where the future roadmap of Oxygen products will be discussed. Participants will have the chance to provide feedback, making it an interactive and engaging session that shapes the future of Oxygen tools.

It's important to know that the Oxygen XML Users Meetup and XML Prague are designed not only to impart technical knowledge but also to strengthen the social fabric of the XML user community through meaningful interactions and shared experiences. You get the chance to **meet like-minded individuals**, **form professional relationships**, and **create lasting connections** within the XML community.

Make sure to mark your calendars and join us on the 6th of June in Prague for this enriching experience! For more details regarding the event and its agenda, please visit Oxygen XML Users Meetup in Prague.

Announcing the Launch of the Oxygen XML Chinese User Group

We are pleased to announce the establishment of the **Oxygen XML Chinese User Group**, a dedicated community for Chinese-speaking users of Oxygen XML suite of products. This initiative is a partnership between Syncro Soft and Ray Content Tech (Beijing) that aims to foster collaboration, knowledge sharing, and mutual support among professionals who utilize Oxygen XML tools in Chinese-language environments.

What are the objectives of the Oxygen XML Chinese User Group?

- *Community Engagement:* Connect with fellow users to discuss experiences, share insights, and explore solutions tailored to the nuances of working with Oxygen XML in Chinese contexts.
- *Resource Sharing:* Access and contribute to a repository of resources, including tutorials, templates, and best practices specifically designed for Chinese-language documentation and workflows.
- *Collaborative Learning:* Participate in discussions and collaborative projects that enhance your proficiency and keep you informed about the latest developments and features of the Oxygen XML suite of products.

At Syncro Soft, we are committed to supporting our diverse user base. The Oxygen XML products offer comprehensive support for creating Chinese content, ensuring a seamless experience for users working with these languages. The editor supports Chinese input methods, displaying composed text in context and allowing for efficient content creation in both Author and Text editing modes.

To learn more about the Oxygen XML Chinese User Group and become part of this community, you can visit raytech-oxygen-xml-community.replit.app/#user-group.

We look forward to building a strong and supportive network of Oxygen XML users in the Chinese-speaking world!

Oxygen XML Ecosystem

The set of Oxygen XML related products has expanded over the years. It is helpful to have a general understanding of the functionality we offer and what each of these products can do. I enjoyed the presentation by **George Bina** titled **Use-cases using the Oxygen XML tools** at Convex Baltimore 2023. It provided an overview of all our products. I am embedding it below for our clients and integrators to also have a general understanding of our products' ecosystem.

Add-ons For Technical Documentation

Oxygen XML Author has various free and useful add-ons for technical documentation writers, including a language model for editing, batch document converter, Git client, content fusion connector, feedback comments manager, terminology checker, DITA translation, and live tutorials.

Oxygen XML Author can be enhanced by installing add-ons. There are a large number of free and useful add-ons for technical documentation writers and we will explore some of them. Documentation and installation instructions for all Oxygen add-ons can be found here.

Oxygen AI Positron Assistant Add-on ***(on page 16)***

Use the **Oxygen AI Positron** service based on **ChatGPT** to enhance your editing experience.

Fluenta DITA Translation Add-on

Create and manage XLiff files from your DITA XML content using the Fluenta libraries.

- Create translation projects
- Convert DITA XML to XLiff
- Convert XLiff to DITA

Batch Documents Converter

Convert various formats like Word, HTML, Markdown, DocBook, Open API, or Confluence into other formats like DITA XML.

Git Client add-on

Collaborate with your colleagues on a Git project. Comes with support for:

- Git Staging
  - Settings to validate before commit or before push.
- Git History
- Git Branch Manager

Content Fusion Connector Add-on

Create review tasks for collaborators and subject matter experts who can later review the tasks in a web browser.

Feedback Comments Manager Add-on

View and manage comments made in the Oxygen WebHelp Feedback forms directly in the Oxygen application.

Terminology Checker Add-on

- Define terminology rules specific for your application to suggest replacements for certain combinations of words.
- Use rules already implemented in Vale syntax.

Live Tutorials Add-on

Create or explore already available tutorials to learn various Oxygen features.

DITA Prolog Updater Add-on

Automatically update the author name and revision dates in the prolog section when saving DITA topics or maps in Oxygen.

DITA References View Add-on

See the set of outgoing and incoming references for each opened DITA topic, including references defined in a relationship table.

Tips And Tricks

Oxygen XML Editor has about 18 years of development under its wings. During these years, we added lots of functionality and many of our users usually do not know more than 20-30% of Oxygen's features. So this presentation is intended for people using Oxygen and who may want to find out more about their tool.

General Layout - Views and Toolbars:

- Side views and toolbars can be arranged by drag and drop. Views can be switched to the auto-hide mode to gain screen size.
- To hide all side views you can use the main menu **Window->Maximize Editing Area** action. Or you can maximize the editing area by double clicking on the tab of an opened XML document.
- To hide all toolbars you can use the main menu **Window->Hide all toolbars** action.
- You can also right-click in the toolbar area and use the **Configure Toolbars** action to see what toolbars are available or what toolbars can be removed.
- The **Symbols** toolbar is quite interesting if you often want to insert symbols that are not on the current keyboard.
- The entire layout of the views and toolbars can be saved (**Window->Export Layout**) and then shared with others (**Window->Load Layout**).

Side Views You Probably Never Used

- The **Information** view.
- The **Properties** view.
- The **Scratch Buffer** view.

Project-related Functionality:

The Oxygen Project view is the place where you can organize and apply batch changes to all your resources:

- Master files support. Enable master files support in the project, add your top level XML documents as master files and any structure changes, renaming or moving files will automatically update all links in the Project. Also works for renaming/moving image resources.
- Store options at project level. Almost all of Oxygen's preferences pages can be saved at project level. Once you do that, you can share the **project.xpr** file with somebody else and when they open it, they will also get the settings set inside the project. Transformation and validation scenarios can also be saved at project level.
- Compare files/folders by selecting them in the **Project** view, right click and using the **Compare...** popup menu action.
- Add references to multiple folders from various parts of your current file system and organize your work in one place.
- Batch operations: Select a folder in the project, right click and use the **Transform**, **Validate**, **Format and Indent**, **Find/Replace in Files**, or **Spell Check in Files**. Or use the **XML Refactoring** action to apply a refactoring stylesheet over all the selected files.
- Filter files. You can instruct the Project view to hide various file types.
- The **Image Preview** view will show the image current selected in the **Project** view.
- Open command prompt in a certain folder (right click on a folder, **Show in->Terminal**.
- Open the current selected file or folder in the system file browser (right click and choose **Show in->Finder**).

Navigation:

Actions to find opened files, find files containing a certain content or navigate between files.

- In the main **Window** menu, you can find the **Next/Previous Editor** actions and their shortcuts. Together with the **Switch editor tab** action, these help you navigate between opened XML documents.
- The **Open/Find Resource** view (main menu **Window->Show View**) allows you to search for file names, search in file contents, or search for files containing comments or change tracking.
- Right-click the tab of an opened XML document and use the **Copy location** action. Or use the **Show in Explorer/Finder** action to locate the file.
- Use the main menu **File->Reopen last closed editor** action to re-open the last closed file.
- Use the **Navigation** toolbar to **Go back/forward/last modification**.

- The main **Window** menu has actions to tile all opened XML documents and also to enable synchronous scrolling in them (useful if you want to look at similar XML documents and maybe to copy/paste between them).

Find/Replace Functionality

The Find/Replace functionality is the bread and butter of any editing application and Oxygen has lots of functionality in this regard:

- Find/Replace in Files.
  - Restrict to XPath - If you want to make replacements only in certain parts of the XML document, the XPath restriction can be used to focus only on certain attributes or element content.
  - Ignore Whitespaces - You should usually check this checkbox as there is always there is a possibility what the words you are searching for may be split on multiple lines in the XML content.
- The **Find** menu → **Find Next/Previous** actions allow you to find the next/previous occurrence of the current selected word(s). Or you can use the **Find All** action to find and highlight all occurrences of a word or a sequence of words.
- The **Find** menu → **Find all Elements** action shows a dialog box allowing you to find elements or attributes containing a sequence of words.
- The **Find** menu → **Quick find toolbar** is a compact toolbar giving you access to search functionality.
- The **XPath** toolbar allows you to use **XPath** expressions (i.e. `//comment()[contains(., 'TODO')]`) to search the current topic or a set of topics for elements, attributes, or comments matching a set of conditions.

Text Editing:

- You can use the mouse wheel or the **Ctrl + and Ctrl -** shortcuts to increase or decrease the font in the current opened XML document.
- The **Preferences->Editor / Edit Modes / Text** page allows you to wrap the visual line and show a print margin.

- Code templates can be used in order to define small code snippets which are often inserted. Editor variables can be used inside those code templates and they will get expanded on insertion. Various useful editor variables:
  - `${author.name}`
  - `${selection}`
  - `${ask('Author Name', generic, 'John Doe')}`
  - `${xpath_eval(upper-case(substring('${cfn}', 1, 4)))}`

Code templates can also be saved at project level or exported in order to share them with others. You can read more about the contexts in which editor variables can appear in this blog post: All About Editor Variables *(on page 469)*.

- Triple click an element tag to select the entire element.
- Various navigation shortcuts: Go after next tag. Select parent. Use the top level breadcrump.
- Move lines up and down (**Alt-Up/Down**).
- Click line number to select the entire line.
- Go to line/column (main menu **Find->Go to...**).
- Open file at cursor on references to navigate to referenced file or create new file (**Ctrl-ENTER**).
- Bookmarks (main menu **Edit->Bookmarks**).
- Quick find filter for side views (Outline, Entities, Transformation Scenarios).
- Drag/drop resources from a file browser or copy/paste to create links to the resources.
- Remove results of XPath execution.
- Use contextual menu to:
    - Apply base 64/32 encoding or decoding to various parts of the XML document. (for example insert base 64 encoded images using the **** protocol.
    - Escape/unescape selection.
    - Toggle comment.
    - Indent selection.
- Split editor (main menu **Window->Split** actions) to copy paste between different parts of the XML document.
- Open an XML document in the **Text** editing mode, right click, choose **Go to definition** to open the DTD, XML Schema or RelaxNG Schema at the precise location where that XML element is defined.

Huge File Editing

Huge XML and Text documents (Gigabytes) can be opened and edited in the application.

Grid editing mode:

- The **Grid** editing mode is useful for looking at XML documents exported from data bases.
- You can use this mode to sort tables or to copy table data to Excel spreadsheets.
- The JSON editor also has a Grid editing mode.

Editing in the Author Visual Editing Mode

- Increase/decrease editor font - Use **Ctrl/CMD +/-** to increase or decrease the font in the current edited document.
- Inserting elements - Pressing ENTER in the Author visual editing mode will show you a list with all possible elements that can be inserted. If you choose an invalid element, Oxygen will find a place for it. You can also add custom actions or code templates to the content completion list.
- Select content then press **ENTER** to surround the content in a new element.
- Select start tag of element, press **ENTER** to rename the element.
- Select multiple intervals - Press the **Ctrl/CMD** button when selecting text to select multiple text intervals. Once you do that, you can use toolbar actions like **Bold**, **Italic**, or **Underline** or you can use the **Edit Profiling Attributes** action to set profiling attributes on all the selected items.
- In-place attributes editor - Instead of using the **Attributes** view, you can press **Alt-Enter** to edit attributes for the current element in a small pop-up dialog box.
- Select an entire element - Triple-click inside an element's contents to select the entire element. After this, you can move the element to a new place.
- Double-click to select by word, then drag the mouse to select the content word by word.
- Move elements (especially paragraphs, list items) up or down (**ALT-UP/DOWN**), indent or unindent list items (**Tab**, **Shift-Tab**).
- Easily insert lists using framework specific auto correct support. Insert for example **1.** at the beginning of a paragraph, press the **Space** key and the inserted content becomes an ordered list. Or in a DITA topic at the beginning of a paragraph type **```java** and press space to insert a `<codeblock>`.
- Select content and then use the **Toggle comment** action (contextual menu **Refactoring** submenu) to quickly add an XML comment around it.
- In the contextual menu the **Text** submenu contains useful actions to count the number of words in the entire document (or in the selected text), convert the selection to upper, lowercase, or sentence case).
- Bookmarks - Click the vertical left side bar in the editing area to add a bookmark. Navigate to that bookmark even after the file has been closed using the **Ctrl/CMD + number** shortcut.
- Collapse other folds - Click a fold triangle in the Author visual editing mode and you can choose to close all other folds, useful when working with multiple sections in the same file.

- Code templates - Define small fragments of XML content that can be inserted either by defining a shortcut key or by pressing **ENTER** in the editing area.

- Editor variables - Certain Oxygen-specific macros can be automatically expanded. For example, a code template that inserts the current author name can use this editor variable:

```
<author>${author.name}</author>
```

or a code template that surrounds the selection inside a `<keyword>` element:

```
<keyword>${selection}</keyword>
```

or a code template that first asks the end user for their name and then inserts it in the document:

```
<author>${ask('Author Name', generic, 'John Doe')}</author>
```

Editor variables can also be used when creating new file templates *(on page 413)*. You can read more about the contexts in which editor variables can appear in this blog post: All About Editor Variables *(on page 469)*.

- Outline Quick find filter - The Outline view contains a filter that can be used to reduce the number of elements displayed in it (for example, display only the contained topics).
- The **Author** editing mode is best for editing bidirectional text with Arabic or Hebrew content.
- Editing MS Word and LibreOffice documents in the Author mode.

Preferences:

Oxygen has a lot of global settings that can be configured, I will just list what I consider to be the most important ones:

- Global - Change application language or show memory status.
- Appearance - Change the default color theme in Oxygen (the **Graphite** theme is quite popular).
- Application Layout - impose custom layout or change editor tabs placement.
- Menu shortcut keys - You can use this preferences page to assign or to see the shortcut for any action available in Oxygen.
- Fonts - This page allows you to change the default fonts used in the application. You can also see what font can be used for a certain set of characters.
- Spell check - Customize the spell checker settings, use custom spell check dictionaries.
- External tools - Define various command line tools that you can launch from Oxygen.
- Editor / Content Completion / Annotations - Disable the tooltips which appear when hovering over various elements (useful if you have a small screen).
- Editor / Save - Settings to automatically save, recover content data on computer crash or to check for errors before saving the file.

CSS-based PDF Publishing

The embedded **Chemistry** engine can be used to publish any XML document to PDF using CSS to style the published output. There is a new transformation type **XML to PDF transformation using CSS** which can be used for this. The **CSS Inspector** view can be used to see what CSS styles get applied.

Tools

The **Tools** main menu is worth exploring. You can:

- Invoke XML refactoring actions.
- Open the SVG Viewer.
- Find various JSON-related tools.

Import - Convert

- The **File** menu offers the possibility to import HTML, CSV, Excel files, or database content as XML.
- For some XML frameworks like DITA, TEI, DocBook and XHTML you can use the smart paste functionality.
- The Batch convertor add-on allows you to convert multiple HTML, Markdown, Excel files to DITA.

Connectivity

Oxygen comes bundled with an SVN client.

You can also work with resources directly from WebDav repositories, Exist and various other databases.

There are also a couple of add-ons which add extra connectivity:

- Git add-on for Oxygen.
- CMIS add-on for Oxygen.
- BaseX add-on for Oxygen.

Add-ons

In the application main menu you can go to the main menu **Help->Install new Add-ons** and our official add-ons update site contains a couple of available add-ons. There are also other free add-ons provided by the Oxygen team that can be installed in an Oxygen standalone version.

I will enumerate some of the most popular interesting add-ons below:

- Git add-on. This plugin adds a side view allowing you to do some common Git operations like pull, commit, push (mostly 90% of what a regular Git user would do).
- Content Fusion Connector add-on. Collaborate with your peers by sending them links that allow them to give you feedback on your work using a web browser without the need to pre-install anything on their side.
- Batch convertor add-on. Converts multiple HTML, Markdown, Excel files to DITA.

Keeping in Touch

- Oxygen's **Help** menu allows you to use the **Report problem** action to contact us directly.
- The **Support Tools->Randomize XML Content** action allows you to randomize the content of an XML project before sending it to us for tests.
- And there are a lot of other ways to get in touch with us or to find various videos or tutorials to read: https://www.oxygenxml.com/technical_support.html.

Checking Terminology with Oxygen XML Editor

In this blog post, I will offer a general overview about the current possibilities you have available to impose your own language checking rules when working with Oxygen XML Editor.

Built-in spell checker

Oxygen comes bundled with the popular Hunspell spell checker and, along with the regular bundled dictionaries for English, German, Spanish, and French, it allows you to install new dictionaries, either for other languages or custom dictionaries (for example, dictionaries for medical terms) that you can build separately: https://www.oxygenxml.com/doc/ug-editor/topics/spell-dictionary-Hunspell.html.

Aside from the spell checker, Oxygen also has support for Auto-correct and you can add your own Auto-correct pairs to Oxygen, but the spell checker is limited to checking individual words.

Oxygen Terminology Checker

In the main application, you can select **Install new add-ons** from the **Help** menu, browse the **Default update site**, and install the add-on named **Terminology Checker**. The terminology checker add-on is available for Oxygen version 20.0 or newer.

Once the add-on is installed, it will allow you to highlight matched terms (words or sequences of words) in the **Author** visual editing mode.

First, you need to use the **File->New** dialog wizard to create a new **Terminology File**, edit that file, and add mappings between matched terms and their suggested replacements, as in the following example:

```
<incorrect-terms>
    ....
    <incorrect-term ignorecase="true">
        <match>virtual assistant</match>
        <suggestion>personal digital assistant</suggestion>
        <message>Prefer 'pda' over 'va'</message>
```

```
    </incorrect-term>
   ......
</incorrect-terms>
```

Then go to the **Preferences->Plugins / Oxygen Terminology Checker** page and in the **Terminology Folder** text field, reference that folder where your terminology files are located. The incorrect terms that have been previously configured should now be highlighted in current editor in **Author** mode.

You can right-click the highlights to access various actions that allow you to replace each individual highlight with suggestions or to apply the same suggestion in multiple places in the currently open document.

Commercial alternatives

Acrolinx is a very popular commercial tool for checking content for consistency and terminology. The plugins that Acrolinx developed for Oxygen standalone, Oxygen Eclipse plugin, and Oxygen Web Author allow you to run the Acrolinx checker directly from inside the application.

Etteplan HyperSTE is another popular commercial tool for checking content and terminology. They also have a plugin for Oxygen standalone.

The Congree Language Checker also has an integration for Oxygen standalone.

AI Positron Add-on

The Oxygen AI Positron Assistant Add-on *(on page 16)* has builtin actions to correct grammar or improve readability of selected content. Other custom actions based on company specific rules can be added.

Open-source alternatives

LanguageTool is an open-source proofreading program for English, French, German, Polish, and more than 20 other languages . There is an open-source plugin for Oxygen available on GitHub.

The DITA Open Toolkit terminology checker plugin from Doctales contains Schematron rules to check that various words adhere to the terminology dictionaries that are custom built using DITA.

Building your own terminology checker

The fastest and simplest way to build a simple terminology checker is by using Schematron rules. The Doctales plugin is a good example for this.

At some point, as the terminology dictionary keeps growing, you may encounter delays and slow-downs when editing the document and validating it using the custom Schematron rules. So, an alternative to this is by using our Author SDK to build your own Oxygen plugin, which can use our API to check the content and then add highlights. The LanguageTool open-source plugin may be a good starting example for this.

10 Things I Enjoy About the New Oxygen JSON Editor

We released the Oxygen JSON Editor a couple of months ago as a practical tool for web developers and people interested in general in editing, validating, or converting JSON documents.

In this article, I will list some of the features of Oxygen JSON Editor that I enjoy the most:

- Value for money. The 12 months subscription license is less than 100 USD per year and always gives you access to the latest JSON Editor released version.
- Support to create JSON Schemas and to use JSON Schemas to validate JSON instances. There is a great **JSON Schema** diagram editor that can be used to visually compose JSON schema files. I used this support quite a lot when creating JSON configuration files for the Oxygen **AI Positron Assistant** add-on.
- Built on top of the Oxygen application platform, the JSON Editor offers the regular Oxygen facilities:
    - Multi platform application (Windows, Mac, Linux).
    - Project view.
    - Find Replace capabilities.
    - and much more....
- As another neat feature of being an Oxygen application, in the Oxygen JSON Editor, you can install useful free add-ons like the Git client-add on, **Batch Documents Converter**, or the JSON Schema Documentation generator add-on.
- All the nifty conversion tools (**JSON to YAML**, **YAML to JSON**, **XML to JSON**) allow you to change between representations in various formats of the same data. For my work, I used, for example, the **XML to JSON** converter to convert a piece of HTML content representing an unordered list to a JSON object that I further proceeded to load from JavaScript.
- When editing a JSON file, the **Author** visual editing mode allows me to add new lines in values by pressing ENTER, new lines that are escaped automatically to `\n` when switching to the Text editing mode.
- You can run **XPath** expressions over opened JSON documents using the **XPath** toolbar.

- You can edit much **more** than JSON documents. This is a web developer editor in disguise, allowing you to edit **Markdown**, **HTML**, **CSS**, **Javascript**, **YAML**, and of course **JSON**.
- Support to create, edit, and convert **OpenAPI** specification documents either in **JSON** or **YAML** format.
- The **Tools** menu's **Compare** utilities allow you to compare a large variety of file formats either file per file or by comparing entire folders.

I consider this tool to be an excellent purchase for developers to edit configuration files in **JSON** or **Yaml**, validate configuration files, perform conversions, even for writing small Markdown documentation topics.

Opinions about using Oxygen

1. https://jaymanalotoibm.wordpress.com/2013/11/27/breathing-oxygen-xml-in-windows-7/
2. Aaron Thayer https://aaronthayer.wordpress.com/2014/11/02/review-oxygen-xml-editor-powerful-xml-authoring-ditto-for-dita/
3. Ray Gallon: http://humanistnerd.culturecom.net/2013/04/11/the-humanist-nerd-reviews-oxygen-14-2/http://humanistnerd.culturecom.net/2015/05/15/the-evolution-of-oxygen-17/
4. Kurt Cagle: https://www.linkedin.com/pulse/praise-oxygen-kurt-cagle?trk=hb_ntf_MEGAPHONE_ARTICLE_POST
5. Doug Morrison: http://www.jks.co.uk/recommendations/choosingxmleditor.html

A set of rules for providing great tech support

I've been doing technical support for more than 10 years and I feel that I've gathered a few ideas about what **great** tech support would be. And doing tech support the right way can be great both for product users and for the product developer as it provides lots of opportunities to further enhance and steer the application.

1. Never say **never**. Never assume for certain that a feature request will not get implemented. Just register it and wait for feedback from others. I've had so many cases in which requests which initially seemed not worthy of implementation became important features in later versions.
2. When asked for a solution to a specific problem, give the solution but also provide a sequence of deductive steps you took in order to find the solution. So give them the fish but also discuss about how they can fish for themselves. Help people evolve and you will have less tech support to do.
3. Try to steer conversations as much as possible from private emails to forums and public user lists. These become repositories of knowledge and you'll avoid explaining the same thing multiple times if there is already a place on the web explaining it.

4. In order to avoid answering the same question multiple times you have the following constructive options:
    - Add a topic in the product's User's Manual explaining the problem
    - Improve the product so that it becomes easier to perform those particular tasks.
5. Some of our users know more and work more with certain aspects of the application than we do. So when certain work-flows are not appropriate for them, it's important that you listen and possibly change the application accordingly.
6. Whenever an older behavior is changed in the application, even if you consider that the change is for the better, you will get complaints. And you will need to decide if users just need a time to adjust to the changes or if you've taken the application in the wrong direction.
7. You may get asked questions which are not particularly related to what the application does. But you might still be able to give your personal opinion and a few useful links to get your client moving in the right direction.
8. You will gain access to various user samples and work with the application to reproduce certain problems. Various times while working with the application to reproduce a problem you will also notice other behaviors which can be improved as well. Contribute all those side-effect suggestions to your issues list as well.
9. Make it as easy as possible for people to report problems or to ask questions about the product. For example in Oxygen in the Help menu we have a "Report Problem" action which can be used to quickly report to us any issue which may arise while using the application.
10. Always try to provide a feasible workaround for a bug or for a lack of functionality.

If you have more ideas about this, please share them with me.

A Short Story of Reuse

Give the smartest human in the world a piece of **wood** and ask him/her to make **paper**. Give them **no prior tools** and it will take **years** to come up with a decent process which would result in some brown almost usable thick piece of paper.

This blog post is about **reuse**, not necessarily **reuse of tools and materials**, but **reuse of knowledge**. Humanity has evolved not because each generation is smarter than the last one but because we learned to **reuse and pass knowledge** to larger and larger audiences and from one generation to another.

Almost all **tools** that we use today are made up of quite a complex set of components which interact with each other. There is no one person in a car assembly factory who still knows all the pieces and how they come together.

Although using the tool is easier than interacting with all components which make it up, you still need knowledge to operate it and in this day and age having enough people to teach how a certain tool can be used is no longer an option. You need to pass knowledge **in other forms**, on **paper** or in some kind of **digital form**. So I would define **technical communication** as a means of passing knowledge about using tools to a **larger audience**.

Reuse in technical communication can be structured on **many levels**:

- Reuse written content by **publishing it to more than one format** (PDF, HTML, EPUB, MS Word).

  It turns out that **XML** is perfect for publishing content in more than one format. **XML** is not designed to be consumed directly by end users and its benefit lies directly in this. Your **XML** content should contain all the data necessary for obtaining all the output formats. So if you are using **XML** in your **technical content**, no matter what **standard** or **custom** vocabulary, you can safely check the first and most important level of reuse.

- Create **larger publications** from **existing ones**.

  Either using an **XML** standard like **XInclude** or using standards with their own diverse and powerful methods of reuse like **DITA**, or by performing custom techniques you can merge **XML** content in larger publications.

- Reuse content written for a **certain tool** to document the functionality and behavior of a very **similar tool**.

  In most mature **XML** standards like **DITA** and **Docbook** there is this implemented concept of profiling which allows you to dynamically filter at publishing time content marked with certain attributes from your original **XML** project. In this way from the same **XML** content you can publish documentation for multiple similar tools.

- Reuse **smaller pieces** of common content in **more than one** publication.

  Again, using **XML** standards like **XInclude** or **DITA** specific standards like **content references** you can create and maintain small libraries of **reusable XML content**, then reuse these components across various publications.

- Reuse **images** and other **binary resources** in multiple publications.

  Because **XML** content does not embed **binary resources**, these resources are stored separately and thus they can be reused in multiple places.

So these are what I consider to be the main selling points for using **XML** in technical documentation. As usual any feedback is welcomed.

Sharing Application Settings

There are various ways in the **Oxygen** standalone version through which a team of writers can share and use a set of common settings. Below I will try to list each of these possibilities along with useful links:

- Save the settings at project level and have all users use the same project **"projectName.xpr"** file in the Project view when editing. Most of the Oxygen Preference pages have a switch radio button which allows you to pass the settings to Project level and save them in the current project configuration file opened in the Project view. If you commit that Project configuration file to a version control system like **GIT** or **SVN**, all writers can open and use that project when working with the repository content.

  For example, when we work on the **Oxygen User's Manual** using our common **GitHub** repository, we all open in the **Project** view the **"userguide.xpr"** project which comes with various fixed options (eg: enabling automatic spell checking):https://github.com/oxygenxml/userguide

  You can find out more details about sharing projects here: https://www.oxygenxml.com/doc/ug-editor/#topics/sharing-projects.html.

- Create and then provide a **default options XML document** to others. When placed in a specific place in the Oxygen installation or referenced via a specific property in the application startup script, this default options XML document will be used by the application to provide default values for certain settings. You can find out more about how default options work here:https://www.oxygenxml.com/doc/ug-editor/#topics/default-options.html.
- Have others manually **import** an options XML document which you **exported** from Oxygen:https://www.oxygenxml.com/doc/ug-editor/#topics/import-export-preferences.html.

- An Oxygen plugin could use our **Java API** to impose fixed options when the application starts using the API method: **PluginWorkspaceProvider.getPluginWorkspace().setGlobalObjectProperty(key, value)**. In our Maven-based SDK we have a sample plugin called **ImposeOptions** which imposes a fixed set of options when the application starts.

If you want to share **document type configurations** (for editing certain XML vocabulary) you can find more details in this older blog post: Document Type Extension Sharing *(on page 407)*.

Collaboration for Documenting a Software Product using DITA

Besides working on an **XML Editor** with lots of **DITA** editing functionality we also use DITA internally for editing the **Oxygen User's Guide**.

In this article I will try to give you an overview of our entire workflow as evidence that **DITA** does work and that it can be used and implemented without expensive solutions.

First here's an overview of our needs:

- **Offline Help** which is available inside the installed application. Oxygen is a multi-platform application so we need to generate both **HTML Help** (CHM) for **Windows** and **JavaHelp** for the **Mac OSX** and **Linux** installations. Also for the **Oxygen Eclipse Plugin** we need to generate **Eclipse Help**.
- **Online Help** which is available as **WebHelp with Feedback** on our web site and allows users to add comments to each topic. Those comments can then be used by us to rephrase and improve our documentation.
- **PDF** containing the entire contents of the user's manual. Nowadays most our users use the online **WebHelp** because it can be used much easier for finding certain topics so in our case at least the **PDF** output is not popular anymore along users.

We have two main distributions (**Standalone** and **Eclipse plugin**) and three main products (**Editor**, **Developer** and **Author**). So we need to produce about six (6) different publications from the same **DITA** content depending on the shipped product.

And here's an overview of the tools we use:

Oxygen XML Editor

This may not come as a surprise but we use our own product to edit **DITA** content, partly because it's ours and partly because it is a very good tool. During the last couple of years this has been a good opportunity to improve our product based on our own feedback (feedback coming from our technical writers).

Oxygen is used in two ways:

1. By the technical writers to write DITA content.
2. By the reviewers to review documented issues by adding comments, making changes with change tracking enabled.

DITA Open Toolkit + WebHelp plugin

We use the DITA Open Toolkit to publish DITA content to the outputs we are interested in. The **WebHelp** and **WebHelp with Feedback** outputs are our own additions to the **DITA Open Toolkit**. But we do not use any special customizations for the other outputs.

Jenkins integration server

We have an automated script which builds all the user manual outputs every night.

Automated DITA Content Validation There is a script which runs on a test server and does three types of checks on the **DITA** content:

1. Validate and check for completeness, check for broken links, images, broken web links and so on.
2. Check and report topics, resources and images which are no longer referenced anywhere.
3. Spell check the entire DITA content.

Git as a version control system

For a long time we used **Subversion** for version control. Recently we moved our DITA content to a private **GitHub** repository and we also made a public **GitHub** repository containing a copy of our user manual's DITA content:https://github.com/oxygenxml/userguide. We use the **SourceTree** application to work with **Git** and we are quite happy with it.

Atlassian Jira for workflow

We use Atlassian Jira to provide a workflow both for the issues which are related directly to our software product and for the issues which are related exclusively with our user's manual. The JIRA is integrated with both our **SVN** and **GIT** repositories so it shows for a certain issue all resources which have been modified to fix it.

More details about how with work with **DITA** can be found in these slides I presented at **DITA Europe 2014**:https://www.oxygenxml.com/forum/files/usingDitaForOxygenUsersManual.odp.

Video demonstration showing how collaboration on a technical publication with Subversion can be achieved: https://www.oxygenxml.com/demo/Collaborative_Authoring_Using_Subversion.html.

Collaboration (Teams working on a common XML project)

Sometimes I get the feeling that there are still many users who collaborate on XML projects using **shared network drives**. We got this question yesterday from one of our users:

> “ I am just curious if housing and working off a network drive is possible, and if so, what are the risks involved?”

In my opinion having multiple writers edit documents from a common **shared network location** is dangerous because you may end up overwriting the content that someone else has been writing on. You also have **no history** of who made what modification and **no redundancy** of content. If somehow the network drive fails because of a hardware problem or files get corrupted you have **no copy of your repository** to start over. My advice is to start using a free and **open source version system** like **Subversion (SVN)**, **CVS** or **GIT**. For example **Oxygen** comes with an embedded **SVN Client** and for writing our user manual which is DITA-based we collaborate using a **subversion server** installed on a Linux machine. We made a small video demonstration showing how this collaboration is done:https://www.oxygenxml.com/demo/Collaborative_Authoring_Using_Subversion.html The advantage to using version systems is immense:

1. We know the history of each resource.
2. We know who made what change.
3. We can create tags and branches for marking certain versions of the documentation.
4. Each user has his own local copy of the repository and if the server fails we have the same content duplicated in many other places.

Other approaches: Use a **WebDav** repository. When Oxygen opens an XML document from a **WebDav** repository, it locks the **XML** document and thus prevents other users from saving changes to it while still allowing them to open the document. Use a **Commercial CMS** which in addition to an open source version system will bring more workflow related tools and lots of ways to keep your content valid. And there are quite a few CMSs which have an integration with Oxygen:https://www.oxygenxml.com/partners.html#solutionpartners%28cms%29

Here's a very nice post by Eliot Kimber about implementing DITA without a CSM:http://drmacros-xml-rants.blogspot.com/#4627052924135934849

Enable massive contributions with oXygen XML Web Author and GitHub

Early in 2016 a new product was added to the oXygen XML set of tools - the oXygen XML Web Author. This leverages the power of oXygen XML Author - which basically runs on the server side - and provides access to XML authoring from any modern device that supports a browser capable of rendering HTML5+JavaScript, including desktops and mobile devices, like your smart phone or tablet!

The real power of web-based XML authoring can be seen when it is integrated as part of a workflow, simplifying it by reducing a large number of steps to a few - and this is what the GitHub connector provides!

If you have XML content on GitHub then you can provide a link that will open a file for editing in the oXygen XML Web Author and anyone will be able to review or update that just by accessing the link and saving - a GitHub account is of course required.

When you save a file, assuming you do not have commit access on that repository, the GitHub connector will automatically

- fork the project into your account, if you do not have a fork already
- create a new branch from the edited branch
- commit your changes on this newly created branch
- create a pull request from your newly created branch to the originally edited branch
- switch the editor to your branch, so further save operations will just add new commits to your branch, thus updating the pull request with new changes

This is a great simplification of the contribution process, a contributor just follows a link and saves the file, and all the magic happens automatically to create the pull request.

If the XML source is published then it is possible to include an *"Edit this page"* link on the published format that will allow immediate access to the editor. An example of such access is provided for the DITA-OT documentation project. The development branch is published at http://www.dita-ot.org/dev/ and every page contains at the bottom an *"Edit this page"* link that gives immediate access to the DITA topic that page is generated from. For example, the https://www.dita-ot.org/dev/ home page has an **Edit this page** button will get you to edit the topic in the WebAuthor.

Edit, then save, and a pull request with your changes will be automatically generated - contribution cannot be easier than this!

Next we plan to have the **Edit this page** option for the oXygen documentation, which is also hosted on GitHub at https://github.com/oxygenxml/userguide.

Hope you find this useful!

George

All About Editor Variables

In various places in the Oxygen XML Editor application there is support for expanding variables. These variables/macros usually take the form **${variableName}** and are expanded by the application dynamically when necessary. A list with all supported editor variables can be found in the user's manual: https://www.oxygenxml.com/doc/ug-editor/topics/editor-variables.html. I will enumerate below all major contexts in which using such editor variables may prove useful:

Transformation Scenarios

Most transformation scenario types have lists of parameters and fields where you can configure the place where the output should be saved. You can use editor variables in these places to make the transformation scenario portable and thus to be able to share it with your colleagues. Here are some examples:

XML with XSLT based transformation scenario types

If you edit such a scenario and go to the **Output** tab you can specify the **Save as** field to use editor variables like this `${cfd}/${cfn}.html` which will get expanded to use the current XML document folder and file name but with a different extension when the output of the transformation gets saved on disk. You can also use editor variables like **${date()}** to save the output file name using the current date: `${cfd}/${cfn}-${date(yyyy-MM-dd)}.html`.

The **XSLT** tab already uses the **${currentFileURL}** editor variable in order to be applied on any XML document opened in the editor area. The **Parameters** list allows you to specify XSLT parameters with values which may contain editor variables which will be expanded by the application before the transformation is run. For example as value for a parameter you can use an **${ask()}** editor variable which will end up requesting the value from the end user when the transformation is started: `${ask('New Parameter Value', generic, 'default')}`.

ANT based transformation scenario types

In the **Parameters** tab you can add new parameters which will be passed to the ANT build file. Values for such parameters can also contain editor variables. You can add for example a parameter called **currentXMLDocument** with value **${pd}/specificFileName.xml** if you want a path to a specific XML file in the current XML project to be passed to the ANT build file as a variable. Again you can use **${ask()}** editor variable which will end up requesting the value from the end user when the transformation is started.

DITA Open Toolkit transformation scenario types

DITA Open Toolkit transformation scenarios are based on ANT so the previous tips apply. In addition you can use the **${rootMapFile}** related editor variables which gets expanded to the

current root map. For example in the **Filters** tab I can specify the reference to the DITAVAL file like this: `${rootMapDir}/filter.ditaval` to refer to the filter relative to the folder where the current root map is published.

You can also extract the root DITA Map file name using the **${xpath_eval()}** editor variable: `${xpath_eval(tokenize('${rootMapURL}', '/')[last()])}`.

New File Templates

You can create your own new file templates and have them use when the Oxygen XML Editor **File->New** dialog wizard is used to create new documents: Sharing New Custom File Templates for a Specific Vocabulary *(on page 413)*.

The content of these new file templates can have inside editor variables which are automatically expanded when a new XML document is created. For example a new file template like this:

```
<topic id="topic_${id}">
    <title>${caret}</title>
    <prolog>
        <author>${ask('Author Name?', generic, 'default')}</author>
    </prolog>
    <body>
        <p></p>
    </body>
</topic>
```

makes use of multiple editor variables:

- The **${id}** editor variable expands to an unique short ID value containing alphanumerical characters. You can also use the **${uuid}** editor variable to generate a truly unique but longer ID value.
- The **${caret}** editor variable marks the position where the caret will be placed after the XML document created from the new file template is initially opened.
- The **${ask}** editor variable will ask the end user to provide the name of the author and will get expanded in the XML content. You can also use **${answer}** editor variables to use the same answer to the **${ask}** editor variable in multiple places.

Code Templates

Oxygen XML Editor's code templates support allows you to define small pieces of XML content which can later be inserted by pressing the **Ctrl-Space** keyboard shortcut (or **ENTER** in the Author visual editing

mode). Code templates can also contain editor variables which get expanded when the code template is used. Examples:

- You can use the **${selection}** editor variable to surround the current selected content in the main editing area inside the code template:

```
<em>${selection}</em>
```

- The **${caret}** editor variable allows you to precisely choose a place for the caret after the code template is inserted:

```
<ph keyref="oxygen"/>${caret}
```

  In the example above the caret will be placed after the inserted element when the code template is chosen.
- The **${ask}** editor variable will trigger the application to ask for a value to insert in a particular place of the code template:

```
<problem reason="${ask('Reason?', radio, ('lost':'lost';'illegible':'illegible';'omitted':'omitted';),
 'lost')}"/>
```

  In the example above when the code template is chosen, the end user will need to choose the reason in a dialog showing a combo box of possible choices and the chosen reason will be expanded in the code template before being inserted in the XML content.

Custom Author Actions

Oxygen XML Editor framework configurations allow defining custom actions for the Author visual editing mode and then contributing these actions to framework-specific toolbars and menus. An example of implementing such a custom action can be found here: Implementing a Custom Author Action to Split a Table *(on page 417)*.

In some of the default operations that you can use in a custom Author action (for example in the **InsertFragmentOperation**) you can use editor variables.

External Tools

You can run external command line scripts from Oxygen XML Editor by configuring external tools in the **Preferences** page. The defined external tools allow using editor variables like **${pd}** to access the current project folder and the editor variables will be automatically expanded before the external tool is invoked.

Custom Editor Variables

You can define your custom editor variables in the application Custom Editor Variables preferences page. A custom editor variable can contain as values other editor variables and can be used in all places where a predefined editor variable is.

You can use Oxygen XML Editor's API to provide custom editor variables and their expanded values using custom Java or Javascript code. For example this plugin adds support for a new **${clipboard}** editor variable.

XSLT Training

Putting together a list of resources (courses, books, training) valuable for learning XSLT:

- Michael Kay's "XSLT 2.0 and XPath 2.0", especially Part I: Foundations.
- XSLT Frequently Asked Questions maintained by Dave Pauson.
- XSLT training courses by Liam Quinn.
- XSLT training by Tomos Hillman.
- An Udemy course by Ken Holman containing also some free lectures.
- XSLT Training courses by Dimitre Novatchev on PluralSight.
- XSLT consulting by Mulberry Technologies.
- Our Oxygen XML training partners.

Log4Shell - Oxygen XML Vulnerability Analysis FAQ

NIST has announced a vulnerability (CVE-2021-44228 code-named Log4Shell) in the Apache Log4j library. Syncro Soft has released a new critical security advisory CVE-2021-44228 and prepared this page containing frequently asked questions and answers related to this vulnerability and its impact on our software products.

In the following days, more vulnerabilities (with far lower impact on Oxygen products) were discovered on the Log4j library: CVE-2021-45046, CVE-2021-45105, CVE-2021-44832.

Please subscribe to our Security Advisories mailing list (found on the top of the Security Advisories page), and revisit this page periodically to get the latest information.

1. How to protect against this vulnerability?
    - Upgrade to the latest version of our products. We have already released new minor bug fix versions for our products. For example, for Oxygen XML Editor/Author/Developer versions 24.0, 23.1, and 22.1.
    - If you cannot upgrade to the latest version, patch or update the Log4j library.
2. How can I patch or update the Log4j library?
    - If you are using one of these products:
        - Oxygen XML Editor/Author/Developer standalone installation
        - Oxygen XML Editor/Author/Developer plugin for Eclipse installation
        - Oxygen XML Web Author

      you can use our free tool (https://github.com/oxygenxml/oxygen-log4j-patcher) to upgrade the Log4j 2.x libraries or to remove the problematic `JndiLookup` class from inside them.

      The recommended Oxygen versions to apply this tool on range from 16.1 to 22.1 inclusively.
    - If you are using Oxygen Content Fusion, you can use the Content Fusion Log4j Patcher: https://github.com/oxygenxml/content-fusion-log4j-patcher.
    - For other scenarios:
        - Scan your system for occurrences of the `log4j-core` JAR file.
        - Stop your running Java application (e.g. Oxygen XML Editor).
        - Delete the `JndiLookup` class from those JAR files. For example, use the following command on a Linux system:

          ```
          zip -q -d log4j-core-*.jar org/apache/logging/log4j/core/lookup/JndiLookup.class
          ```
3. Am I still vulnerable if I set the `-Dlog4j2.formatMsgNoLookups=true` system property?
    - Although our products do not meet the preconditions for CVE-2021-45046, setting this system property is no longer considered to offer bullet-proof protection.
4. Am I still vulnerable if I set the environment variable: `LOG4J_FORMAT_MSG_NO_LOOKUPS="true"`?
    - Although our products do not meet the preconditions for CVE-2021-45046, setting this environment variable is no longer considered to offer bullet-proof protection.
5. Are there components that may have this problem used by the application outside of the Oxygen installation folder?
    - Oxygen add-ons may come bundled with Log4j 2.x libraries. Update all your add-ons to their latest version by using the **Help** > **Check for add-ons updates** menu action.
    - The Oxygen application may have external tools configured in the **Preferences** > **External Tools** page. Check that your external tools do not use Java processes that may have this problem.

- You may have configured extra JAR libraries used for validation or transformation scenarios. Check that all of these separate libraries do not contain the Log4j 2.x libraries.
- There may be data sources (for example, for connections to eXist-db servers) created in the **Preferences** > **Data Sources** page that have references to Log4j 2.x libraries. If this is the case, update your connection libraries to the latest ones provided for the latest server installation.
- You may run DITA publishing using an external DITA Open Toolkit publishing engine that has older versions for the bundled Log4j 2.x libraries provided by the Oxygen-specific publishing plugins. Update the Oxygen plugins used by the custom DITA Open Toolkit to the latest versions available on our website.

6. I am using the Oxygen SDK and cannot upgrade. How can I upgrade just the Log4j library?
   - Make sure that you do not have the version of Log4j hard-coded in your `pom.xml` file.
   - You can control the version of Log4j to be included by adding the following configuration in the `dependencyManagement` section of the `pom.xml` file of your project:

```
<dependency>
  <groupId>org.apache.logging.log4j</groupId>
  <artifactId>log4j-bom</artifactId>
  <version>2.17.1</version>
  <type>pom</type>
  <scope>import</scope>
</dependency>
```

   - To test which version of Log4j your Maven project uses, run: `mvn dependency:list -DincludeGroupIds=org.apache.logging.log4j -U`.
   - If you are using the `web-author-component` artifact, you can use a Maven WAR Overlay to replace the packaged Log4j library distributed with a newer version.
7. Am I still vulnerable if I block all outbound traffic?
   - If you block ALL the outbound traffic, you may be safe, but:
     - Be sure to block also DNS traffic, as it can exfiltrate data such as environment variables.
     - Be sure to re-assess which servers are trusted - maybe some trusted servers had the same vulnerability and were exploited.
   - Regardless, it is a very good idea to restrict outbound traffic from Oxygen tools. This will protect you against various types of vulnerabilities.
8. Am I still vulnerable if I use a new version of Java?
   - The vulnerability is known to exist with the latest versions of Java (e.g. > 1.8u191) if you are also using Apache Tomcat or Websphere. If you are not using those servers, you may still be vulnerable, but we are not aware of any exploits.

9. Should I remove the `JndiLookup` class?
    - Yes. Removing it will remove the vulnerable code. We have high confidence that other problems will not be generated.
10. Should I set firewall rules based on HTTP headers or URLs to block attacks?
    - These kind of rules do not offer full protection. The URLs that exploit the vulnerability can be written in various ways and are not recognized by common WAF rules that you might find.
11. How to test if I am vulnerable?
    - If you did not apply any mitigation, you can assume you are.
12. Should I do anything if the affected server is deployed inside the intranet?
    - Yes. The attack can be triggered from another server that was exploited or by one of the employees when they follow a link in a phishing email.
13. Should I upgrade to the latest version of your products?
    - Yes. New maintenance builds that avoid this vulnerability are available for versions that are in the maintenance period.
14. Is this vulnerability related to Java being insecure?
    - No. It is caused by an open-source component called Log4j that had a vulnerability. This component is widely used in the Java ecosystem.
15. How is Syncro Soft addressing the problem?
    - We patched all our public servers against this vulnerability.
    - We provided a security advisory with current solutions (https://www.oxygenxml.com/security/advisory/CVE-2021-44228.html) and we produced maintenance builds that remove the vulnerable version of Log4j for all product versions that are under maintenance. We are working on solutions for older versions.
    - We analyzed and updated our internal servers to make sure they are not vulnerable.
    - We scanned the logs of the services we use or provide to look for the pattern that triggers the vulnerability.

Related information

https://www.lunasec.io/docs/blog/log4j-zero-day/

https://msrc-blog.microsoft.com/2021/12/11/microsofts-response-to-cve-2021-44228-apache-log4j2/

https://www.oxygenxml.com/security/advisory/CVE-2021-44228.html

Using Oxygen XML Editor in a Web Browser (Experimental)

The JetBrains Projector is a technology for rendering Swing applications over the network and is an interesting way to run and access these apps remotely. Some of its use cases are listed here: https://jetbrains.github.io/projector-client/mkdocs/latest/ij_user_guide/jetbrains/#use-cases.

Since Oxygen XML Editor is a Java Swing-based application, this blog post explains how Oxygen XML Editor could be accessed remotely and used from a web browser.

Figure 1. Oxygen XML Editor Running in the Safari Web Browser Using JetBrains Projector

To prepare an Oxygen XML Editor installation on the server side for running in a web browser:

1. Download an All Platforms version of the Oxygen XML Editor installation kit (`oxygen.tar.gz`): https://www.oxygenxml.com/xml_editor/download_oxygenxml_editor.html?os=Other.
2. Unpack it to the `oxygen` folder.
3. Download the Jetbrains projector server and unzip it in the Oxygen installation folder (the folder path should be something like: `oxygen/projector-server-1.5.0`).
4. Download the Jetbrains Java 11 runtime enviroment and unpack it to the `oxygen/jbr` folder.

5. Create a `start-projector-oxygen.bat` script in the `oxygen` installation folder so that it adds the JetBrains libraries to the classpath and runs the JetBrains server giving it the name of the main class as a parameter to run it with the contents:

```
@echo off

SET

 CP="%~dp0;%~dp0/classes;%~dp0/lib/oxygen-basic-utilities.jar;%~dp0/lib/oxygen.jar;%~dp0/projector-server-1.5

.0/lib/*"


%~dp0\jbr\bin\java.exe %OXYGEN_JAVA_OPTIONS% -Dcom.oxygenxml.app.descriptor=ro.sync.exml.EditorFrameDescriptor

 -cp %CP% -Dorg.jetbrains.projector.server.classToLaunch=ro.sync.exml.Oxygen

 org.jetbrains.projector.server.ProjectorLauncher %*
```

6. Run the `start-projector-oxygen.bat` script, which will start an HTTP server on port `8887` by default.
7. Connect to the `localhost:8887` port from the same computer or from another computer using a web browser.

Result: You should be able to use Oxygen XML Editor in a web browser and have full access to the server-side resources.

Oxygen XML Blog 2022 Retrospective

So here we are at the end of **2022** and I figured it's a good time to go through some of the improvements we made and posts we added in the **Oxygen XML Blog**. A short summary of our activity on the **Oxygen XML Blog** can be found below:

Statistics

As an overview of the most read articles on the blog, we found out that articles about conversions from various formats (especially MS Word) to **DITA XML**, articles about learning **DITA XML**, and general **tips and tricks** articles were the most read. Also, I think we provided some good quality content covering these aspects in 2022. In total, we provided about 30 new articles to the blog in 2022.

Collaborators

This year, we had a lot more internal collaborators with contributors to the **Oxygen XML Blog**. Our web designers **Mihai** and **Luci** created a new color theme for the blog. My colleagues **Cosmin Duna**, **George Bina**, and **Alex Smarandache** contributed various interesting and useful articles to the blog. Our external collaborator **Chris Papademetrious** also submitted an article about enhancing DITA Open Toolkit project files.

Moving the product forward

The blog also served as a place to experiment with ideas that were later incorporated into **Oxygen**. For example, based on articles written in the blog, we added to **Oxygen** the capability to convert **OpenAPI** documents to **DITA XML** or to generate **Google Structured Data** content when publishing **DITA XML** content to **WebHelp Responsive** Output.

We hope you found the articles we added to the **Oxygen XML Blog** this year useful. If you have ideas about possible future articles that we could add to the blog, we are always open to suggestions, or you can also contribute articles yourselves: How You can Contribute to the Oxygen XML Editor Blog. *(on page 9)*

A more detailed list of contributions to the blog can be found below:

Colors and Layout Redesign

My colleagues **Mihai** and **Luci** (who maintain and design our web site) have had time this year to do a full redesign of the Oxygen **WebHelp** publishing template we are using to publish our blog's DITA XML content to WebHelp.

So thanks to their work the entire layout of the blog looks better now!

We also changed the **WebHelp** feedback authentication available on each page of the blog to the **Oxygen XML** organization so that with the same authentication, you can leave us feedback either on the **Oxygen Users Guide** Web pages or on the **Oxygen XML Blog**.

Migrating Word and Unstructured FrameMaker to DITA XML

Our colleague **Cosmin Duna** submitted two interesting migration related articles:

- Migrating MS Word to DITA using the Batch Documents Converter *(on page 358)*
- Migrating Unstructured Adobe FrameMaker Content to DITA *(on page 368)*

DITA Publishing

In 2022, I wrote quite a few small articles based on my experiences with various DITA publishing customizations and they are all linked in this overview post: Publishing Customizations *(on page 178)*.

I also wrote an article about the various ways that we support the **DITA Open Toolkit** open source engine: Supporting the DITA OT Project as a Commercial Company *(on page 274)*.

My colleague **Alexandru Smarandache** submitted his first article about creating learning content for the Google Chatbot from DITA XML content *(on page 331)*.

Our client and friend **Chris Papademetrious** also submitted an article about enhancing DITA Open Toolkit project files: Preprocessing DITA-OT Project Files *(on page 296)*.

These articles were also presented at the **DITA Open Toolkit Day** 2022 along other interesting talks: https://www.oxygenxml.com/events/2022/dita-ot_day.html.

DITA Perspectives

Our boss and colleague **George Bina** contributed an article and a GitHub project with his experiments creating automatic graphs showing the hierarchy of DITA elements in the DITA specification: DITA Perspectives *(on page 79)*.

Short Clips - DITA Editing

We started a series of small articles presenting various features of **DITA XML** editing with **Oxygen** using small videos:

- Short Clips - Insert Image References *(on page 69)*
- Short Clips - Insert Links *(on page 72)*
- Short Clips - Reuse Content *(on page 76)*

Chapter 10. Contributors

Everybody can contribute interesting XML and publishing-related articles to the Oxygen XML Blog: How You can Contribute to the Oxygen XML Editor Blog. *(on page 9)* All contributors to the Oxygen XML Blog are listed below.

Table 4. List of contributors to the Oxygen XML Blog

Author Name	Details
Alex Jitianu	Project Manager, DITA/Authoring team. Oxygen XML Editor
Alex Smarandache	Software Developer. Oxygen XML Editor
Alin Belu	Marketing Specialist. Oxygen XML Editor
Chris Papademetrious	Senior Technical Writer, Synopsys Inc.
Cosmin Duna	Senior Software Developer. Oxygen XML Editor
Cristi Talau	Project Manager, Content Fusion. Oxygen XML Editor
George Bina	Owner, DITA XML Expert. Oxygen XML Editor
Sorin Carbunaru	Senior Software Developer. Oxygen XML Editor
Octavian Nadolu	Project Manager, Development team. Oxygen XML Editor
Mihaela Calotescu	Project Manager, Web Authoring team. Oxygen XML Editor
Radu Coravu	Helping people with XML-related technologies since the start of this century. Oxygen XML Editor
Steven Higgs	Senior Technical Writer. Oxygen XML Editor
Stefan Vasile	CEO. Oxygen XML Editor

Index

A

Add-ons

artificial intelligence

16

AI Positron Assistant

16