Localization and the DITA Open Toolkit

Simon Bate
Sr. Technical Consultant, Scriptorium

October 30, 2017

SCRIPTORIUM

Overview

Out of the box, the DITA Open Toolkit (OT) looks as though it's localization-ready. The HTML and PDF plugins contain strings for over 50 languages. So it would seem that all you have to do is specify the language in your DITA files and maps and you're good to go…or are you?

This white paper addresses some of the issues Scriptorium has encountered while generating localized output from the DITA OT—and how we solved them. The paper begins by describing the various aspects of localization that are supported in the DITA Open Toolkit, including string handling, fonts, and indexes. To wrap up, it describes the steps you need to take when working with a new localization.

> **Note:** This white paper covers technical implementation details in the DITA Open Toolkit (up to and including DITA OT 2.5). You should be familiar with Apache Ant and should have a reading knowledge of XSLT.

The xml:lang attribute

Localization in the DITA Open Toolkit begins with the DITA XML attribute xml:lang. This attribute specifies the language and country of the contents of a particular element. In order for DITA OT localization to work, you must use the xml:lang attribute on the root element of your map (or bookmap) and on the root elements of any topics referenced by the map. If the xml:lang attribute is not specified, the DITA OT defaults to US English.

For instance, this xml:lang attribute indicates that a Concept topic contains Canadian French content:

```
<concept id="c_202934802" xml:lang="fr-ca">
```

When DITA topic and maps are localized, translators should add or modify the xml:lang attribute, specifying the target language and locale.

> **Tip:** Even if you're creating content for US English, it's a good idea to add the xml:lang attribute when creating topics and maps. That way, the attribute is present in the files when they are received by the translator; this helps remind the translator that the xml:lang attribute must be updated.

It's (mostly) about strings

When DITA Open Toolkit plugins transform DITA topics into output, they must occasionally insert additional text. For example, chapter titles (in English) often contain the word "Chapter", admonitions usually need the labels "Note" or "Warning", and page footers might contain a statement forbidding copying. These text strings are usually different in each target language.

These inserted text strings are not stored in the DITA topics; some come from the DITA Open Toolkit itself, other strings come from DITA OT plugins. Most plugins that create output define text strings. Usually they define multiple sets of strings, one for each language that the plugin supports.

```
<!-- Nav bar buttons -->
<str name="contents_button">Contents</str>
<str name="index_button">Index</str>
<str name="search_button">Search</str>
<str name="print_button">Print</str>
```

It’s good programming practice to store strings in a separate file from code; the code then references each string when it is needed. This eliminates duplication and makes maintenance and translation much easier. Additionally, when a different language is required, a file (or set of files) with strings for the target language can be programmatically selected.

The DITA Open Toolkit uses this principle. Plugins can organize strings into parallel files, one file per language. The DITA OT selects the string file based on the xml:lang setting.

String storage and retrieval has been present in the DITA OT since the first versions. This original file organization is still used by most HTML plugins; this paper refers to these as “HTML strings files.” Strings are retrieved from these files with the getString template.

Idiom Technologies, the creators of the default PDF plugin in the DITA Open Toolkit, was aware of many issues associated with internationalization and localization. In creating the PDF plugin, Idiom realized that returning a single string was not flexible enough for some situations. Instead, Idiom ignored the HTML strings files (and the getString template) and implemented a much more sophisticated series of XSL templates. This paper refers to these as “PDF strings files.” Strings are retrieved from these fies with the insertVariable template.

The DITA Open Toolkit 2.1 introduced a more generalized way of accessing strings (the getVariable template), which works equally well with both HTML string files and PDF string files.

All three of these mechanisms are present in the modern DITA Open Toolkit (currently v2.5). Newer XSL stylesheets should use the getVariable template and the PDF string files. But it’s quite common to find legacy plugins that use the older templates and the HTML strings files.

HTML strings files

There are two parts to the HTML strings file organization. Each plugin contributes:

- One or more strings files, one for each target localization.
- A strings.xml file that associates its strings files with xml:lang identifiers.

The DITA OT itself defines default strings files and uses its own strings.xml file to organize them. You add the strings.xml file to the DITA OT through the plugin integrator.

A strings.xml file contains one or more <lang> elements. Each <lang> element associates an xml:lang attribute value with a file containing strings for that language. The following two

<lang> elements associate both Arabic and Egyptian Arabic with the Open Toolkit file strings-ar-eg.xml:

```
<lang xml:lang="ar" filename="strings-ar-eg.xml"/>
<lang xml:lang="ar-eg" filename="strings-ar-eg.xml"/>
```

The filename attribute identifies the strings file for that language and country. Each strings file contains one or more <str> elements. The <str> element associates a common identifier (the name attribute) with a language-specific text string. For instance, the German strings-de-de.xml associates the task_example name with the German string "Beispiel" using this <str> element:

```
<str name="task_example">Beispiel</str>
```

Translators should never translate the name attribute (task_example in this example); it is the common thread that allows the OT to find a specific string in any of the defined languages. Each language string file defines a parallel set of <str> elements (each with the same name attributes, but with different, language-specific text). So the Spanish version, strings-es-es.xml, contains:

```
<str name="task_example">Ejemplo</str>
```

and the U.S. English version, strings-en-us.xml, contains:

```
<str name="task_example">Example</str>
```

PDF strings files

The PDF strings files are located through a plugin's customization.dir property (rather than through integrator-maintained files).

The default PDF strings files are found in $DITA_HOME/plugins/org.dita.pdf2/cfg/common/vars.[1] Most of the filenames use only the two-letter language code (en.xml), but some regional languages include locale and other details. As with the HTML strings files, the DITA OT defines PDF strings files for 50 languages and locales.

Each PDF strings file contains one or more <variable> elements. The <variable> element associates a common identifier (the id attribute) with a language-specific text string. For instance this <variable> element (from the German de.xml) associates the "Table of Contents" id with the German string "Inhalt":

```
<variable id="Table of Contents">Inhalt</variable>
```

As with the HTML strings files, each plugin maintains a parallel set of strings files, each using the same id values and containing language-specific text.

[1] $DITA_HOME refers to the directory where the DITA Open Toolkit is installed.

Character sets and fonts

Computers use numbers (as codes) to represent everything they have to process, from ID codes to machine instructions to characters (or “glyphs”) in human languages. Thus, in a computer, a lower-case “a” can be represented by the value (or code) 97. Each character in a given character set has a different code.

For many years, character sets were limited by the number of values (or characters) that could be represented by a single byte (usually 255 characters). Character sets were also constrained by the fact that each byte represented exactly one character. Conflicts could arise, because one character set might represent characters differently from another; additionally, the same codes might represent different characters in different character sets. You can still see this when using older versions of fonts, such as Zapf Dingbats, where the same code you use for a lowercase “s” instead represents a filled triangle (▲). If you have dealt with these kind of character sets and fonts, you know you need to use the right font for each character, otherwise your output will be garbled.

With the advent of Unicode (a variable, multi-byte encoding scheme), almost every character known in the world has a specific code or “code point.” Unicode can contain hundreds of thousands of code points. In Unicode if you need to write a lower case “s”, you use one code point (U+0073); if you need to write a filled triangle, you use a different code point (U+25B2). This frees you from worrying about using the right character set and the right font, unless you have a need to use one of these older fonts.

Although Unicode provides internal representation for almost every known character, your computer still needs be able to display that character on screen or print it on a piece of paper. This is the role of computer fonts[2].

Crafting a font that can represent a full set of Unicode code points is a massive undertaking. Instead, most fonts implement a more focused subset of code points (such as those required by a regional set of languages). A font usually contains characters and punctuation for a given language or region, plus a number of additional symbol characters. A typical font will implement up to 40,000 code points.

When localizing DITA content, it is crucial that you find and use fonts that are appropriate for your content. The fonts you choose should contain glyphs for all the code points you use.

If you’re looking for full Unicode code point coverage, consider:

- The Google Noto font family probably contains the most complete set of code points that is currently available. It is so large, it is divided into subsets so that the size of font files remains reasonable.
- For Western languages, the Arial Unicode MS font is one of the more densely populated fonts, containing more than 50,000 code points. It is distributed in Microsoft Office, but also bundled in Mac OS X v10.5 and later.

[2] Although the word “typeface” is more accurate, the word “font” has come to mean both a specific typeface and all the specific variations of that typeface.

Fonts in HTML

In HTML-related plugins, most font selection is determined in CSS files, where the font-family property identifies one or more possible fonts to use for a given selector. The font-family property can contain a comma-separated list of fonts. A browser uses the first font in the list that is installed on the computer (including web fonts downloaded with the `@font-face` declaration).

It is important to note that once the browser has found a font in the font-family property, it uses that font *whether or not the characters using that font exist in the font*; it will not look to the next font in the font-family list.

If a string of characters is styled using font-family and the selected font does not contain characters used in that string, the browser will either fall back on an internal font that can display most characters, or—worst case—you may end up with "Tofu" (empty boxes, like this "□", in place of the characters).

It's also important to consider which fonts are actually available to the browser. Typically there are two ways of approaching this:

- Use the standard fonts that are available on most computers. Using CSS generic font families is a good fall-back choice for the font-family property (`Serif`, `Sans-serif`, and `Monospace`)
- Use downloadable web fonts. This requires you to add additional code to your HTML or CSS (or both). It might also require you to work with your web services team to make sure that the fonts are consistently available for download, particularly if you use custom fonts.

Fonts in PDF

Font handling in the PDF plugin addresses two issues with fonts and localized content:

- A single font will rarely contain glyphs for all code points.
- Fonts are usually not designed to look pleasing in all languages.

When creating output for different languages, you usually need to use a different font for each language.

To address issues of fonts and localized content, the transforms in the PDF plugin use placeholders (called "logical fonts") for particular types of text. For instance, there can be different logical fonts for body text, titles, running headers and footers, and monospaced content. Transforms can define as many logical fonts as the content requires.

A font mapping file in the PDF plugin defines the relationship between logical fonts and the language-specific physical fonts used in the final PDF. Thus, a logical font for body text could be mapped to the physical Garamond font when the source text is in English, but when the source text is in Simplified Chinese, the body text logical font is mapped to the physical SimHei font.

The physical fonts are often determined by the style guidelines for your company or organization; they ensure that your information products project a consistent look and feel.

To localize a PDF transform, you need to ensure that the logical fonts address the different font usage in your content and that the mapping from logical fonts to physical fonts generates the desired output.

Special characters in PDF

Special characters (such as trademark or registered trademark) can sometimes be an issue with fonts in PDFs. A font that supports characters for a particular language might not contain the glyphs for some of the code points you rely on. To solve this, the PDF plugin allows you to associate special characters with fonts that implement them.

The folder $DITA-OT/org.dita.pdf2/cfg/fo/i18n contains a series of language-specific files that identify Unicode character ranges for languages, as well as special characters that occur in those languages. Each character is assigned to a particular character set.

Sorting indexes

Index generation is a crucial aspect of localization because indexes must be sorted differently in each target language. Out-of-the-box, only four DITA Open Toolkit transformation types provide index generation (JavaHelp, Eclipse Help, HTML Help, and PDF); the others do not.

For the HTML-based help plugins (JavaHelp, Eclipse Help, HTML Help), the process of extracting, sorting, and generating indexes is not implemented in XSL. Instead, these steps are performed by Java classes in lib/dost.jar. These classes use the language-specific sorting sequences that are a part of Java, thus the results are appropriate for your target language.

PDF index processing is performed by both XSL templates and by additional Java classes.

The next two sections describe the HTML and PDF index processing in greater detail.

HTML indexes

If you are creating a plugin and need to generate an index as part of your output, you have three alternatives:

- Download and modify the sources for dost.jar. Add support for your output format, recompile, and include the new dost.jar in your plugin.
- Use one of the existing index types available from dost.jar, then modify the output (perhaps using XSL) to suit your needs. This is usually the easiest approach.
- Create your own index-building programs. However, if you do this, you need to ensure that the program's sort capabilities can support all your target languages and locales.

PDF indexes

When the PDF plugin processes a map, it merges all the individual DITA topic files into a single XML file named stage1.xml. As part of merging the files, the plugin gathers all the <indexterm> elements, sorts and groups them, and writes them to the merge file, using the namespace opentopic-index.

The plugin performs this action for bookmaps and maps; it doesn't matter whether the element was specified in the bookmap. Nor does it matter which FO processor you use (FOP, Antenna House, or RenderX XEP).

The entries are sorted by the com.idiominc.ws.opentopic.fo.index2.IndexPreprocessorTask Java class, which gets its sort order from the sort-order files in cfg/common/index. The files in this folder are named like the files in cfg/common/vars, with the form: *lang_LOC*.xml. Each file defines the character sorting order for one language and locale.

To add a new language or locale to the PDF plugin, create a new sort-order file. Start with an existing sort-order file that is close to your target language and country and copy that file to Customization/common/index.

Conclusion

When you localize your DITA content, remember that DITA OT plugins also contain language-specific information.

The strings, images, icons, and fonts used by your plugins must be translated or localized with the same care and cultural sensitivity as your DITA content.

Many parts of the DITA Open Toolkit were created with localization in mind. It's much easier (and more consistent) to use those mechanisms than to try to make language-specific changes in your transforms.

About the author

With more than 40 years experience in technical publications, *Simon Bate* has acquired extensive knowledge in writing, managing, production, book design, template design, and document conversions. Simon takes great delight in programming and scripting. His motto is: "let the computer do the work." He also enjoys teaching and sharing his knowledge with others.

Simon divides his time at Scriptorium between tools development and training. He has worked on many DITA Open Toolkit projects, including customizing existing DITA Open Toolkit transforms, building Scriptorium's webhelp transform, and creating DITA to LMS transforms for Scriptorium's LearningDITA web site (where he has authored a number of lessons).