Eberlein
Consulting

DITA training, information architecture, and strategy

Branch filtering

Contents

Branch filtering

The DITAVAL format provides a way to specify a set of conditions that are used to conditionally process DITA content. While these conditions are often stored outside of the content and applied globally by a processor, the `<ditavalref>` element can be used to reference a DITAVAL document in order to filter only a subset of content within or referenced by a map. This also provides the ability to process a single branch of content multiple times, while applying unique conditions to each instance of the branch.

Conditional processing (profiling) on a subset of content

The `<ditavalref>` element is designed to help manage conditional processing in two distinct situations. If a separate set of DITAVAL conditions is applied globally to the content, those take precedence over any conflicting conditions specified within a DITAVAL referenced by `<ditavalref>`.

1. A set of DITAVAL conditions is applied globally to a content set. However, one subset of information needs to be filtered using a different set of DITAVAL conditions. This is a common use case when content is aggregated from different authoring groups or companies. In this case, the `<ditavalref>` element can be used to selectively filter the subset of information, without affecting the rest of the content.
2. A set of content contains information that is common to all users, along with a subset of information that varies based on a condition such as audience or platform. The subset of information needs to be repeated once for each condition. For example, a set of software documentation contains common information, as well as installation instructions that vary by operating system. In this case, the `<ditavalref>` element can be used to publish installation instructions that are specific to each operating system, without repeating the common information that is applicable to all operating systems.

   In this case, a fully resolved view of the map contains multiple instances of a single branch of content. This might result in duplicate keys, duplicate key scopes, and URIs that specify the same resource with conflicting conditions. Metadata inside of the `<ditavalref>` element is available to provide control over these values, so that keys, key scopes, and URIs can be individually referenced within a branch. This means that processors *MUST* calculate the effects of branch filtering on keys and URIs before they can establish the complete key space for a map.

It is possible for a branch with `<ditavalref>` already in effect to specify an additional `<ditavalref>`, so that a subset of the original branch receives additional filter conditions. In this situation, suffixes and prefixes are added using an outer-to-inner approach. For example, using the file name `basefile.dita`, a branch that added two prefixes and two suffixes would result in `parentPrefix-childPrefix-basefile-childSuffix-parentSuffix.dita`. In that sample, the hyphens are part of the specified prefix or suffix (they are not added automatically).

It is an error if `<ditavalref>` driven branch cloning results in multiple distinct copies of a topic having the same resolved name. Processors *SHOULD* report an error in such cases. Processors *MAY* recover by using an alternate naming scheme for the conflicting topics.

One or more `<ditavalref>` elements can appear as a child of the `<map>`, which results in the filters being applied to the entire map. Using the `<ditavalref>` element as a child of a reference to a locally-scoped map (such as within `<mapref>`) is similar, and results in the conditions being applied to the referenced map. Using the `<ditavalref>` element within a reference to a map whose `@scope` attribute evaluates to "peer" or "external" has no effect.

Examples to illustrate each of these conditions are available in *Examples of branch filtering* on page 5.

Branch filtering: Implications of processing order

Because the branch filtering process may result in new or renamed keys, key scopes, or URIs, the full effects of the branch filtering process *MUST* be calculated by processors before they can construct the effective map and key scope structure.

The `@keyref` attribute and related attributes are explicitly disallowed on `<ditavalref>` for the same reason; otherwise, usage of those attributes on `<ditavalref>` would require that `@keyref` be resolved on all `<ditavalref>` elements before keys and key scopes could be established for the map as a whole.

Comment by Kristen Eberlein on 8 February 2014
Jarno Elvirta suggests that we simply state "branch filtering *MUST* be processed before the effective map and key space are constructed."

Jarno also suggests: "You could have a note in the spec that the spec does not mandate a given processing order or style of processing. In cases where the spec defines something using a given processing order, implementations may implement this in any way they want, but the results must match."

Comment by Kristen Eberlein on 8 February 2014
Another suggestion from Jarno Elvirta:

"Hmm... could you phrase that shortdesc so that effective map and key space construction has a dependency on branch filtering results, and that may impose processing order requirements on implementations?"

In general, the DITA specification shies away from mandating processing order, and so publication results can vary slightly depending on the order in which processes are run. In this case, any DITAVAL conditions that are specified external to the map, such as a parameter to a publishing process, take precedence over conditions referenced with the `<ditavalref>` element. For example, if the value `audience="admin"` is globally excluded by the publication process, that content will be excluded even from a branch where a `<ditavalref>` reference attempts to set `audience="admin"` to "include".

There is explicitly *no requirement* that global filters and filters specified with `<ditavalref>` be applied at the same time in a publishing process.

Processors should consider the following points when determining a processing order:

- If links are generated based on the map hierarchy, those links should be created using the renamed keys and URIs that result from evaluating `<ditavalref>`, to ensure that the links are consistent within the modified branches.
- If conrefs are resolved in topics before the `<ditavalref>` filter conditions are evaluated, content that applies to multiple audiences can be brought in; it later can be selectively filtered by branch. For example, if a set of installation steps is pulled in with conref (from outside the branch), it might contain information that is later filtered by platform during the `<ditavalref>` evaluation; this results in a copy of the steps that specific to each operating system. If conref is processed after the `<ditavalref>`, content might be pulled in that has not been appropriately filtered for the new context.
- The same scenario applies to conref values that push content into the branch; pushing content prior to resolving the `<ditavalref>` conditions allows content for multiple conditions to be pushed into the branch, and then filtered by branch based on the `<ditavalref>` conditions. If the branch using `<ditavalref>` pushes content elsewhere, resolving `<ditavalref>` first could result in multiple copies of the content to be pushed (one for each branch), resulting in multiple potentially conflicting copies pushed to the new destination.

Examples of branch filtering

This section of the specification contains examples of branch filtering. These examples illustrate the processing expectations for various scenarios involving `<ditavalref>` references. Processing examples use either before and after markup samples or expanded syntax that shows equivalent markup without the domain.

Example: Single `<ditavalref>` on a branch

A single `<ditavalref>` element may be used to supply filtering conditions for a branch.

Consider the following DITA map and the DITAVAL file that is referenced from a `<ditavalref>` element:

```
<map>
  <topicref href="intro.dita"/>
  <topicref href="install.dita">
    <ditavalref href="novice.ditaval"/>
    <topicref href="do-stuff.dita"/>
    <topicref href="advanced-stuff.dita" audience="admin"/>
    <!-- more topics -->
  </topicref>
  <!-- Several chapters worth of other material -->
</map>
```

Figure 1: `input.ditamap`:

```
<val>
  <prop att="audience" val="novice" action="include"/>
  <prop att="audience" val="admin" action="exclude"/>
</val>
```

Figure 2: Contents of `novice.ditaval`

When this content is published, the following processing occurs:

- The first topic (`intro.dita`) does not use any of the conditions that are specified in `novice.ditaval`. It is published normally, potentially using other DITAVAL conditions that are specified externally.
- The second topic (`install.dita`) is filtered using any external conditions as well as the conditions that are specified in `novice.ditaval`.
- The third topic (`do-stuff.dita`) is filtered using any external conditions as well as the conditions that are specified in `novice.ditaval`.
- The fourth topic (`advanced-stuff.dita`) is removed from the map entirely, because it is filtered out with the conditions that are specified for the branch.

Comment by robander on 21 April 2014
Review 1: David Helfinstine asks if this is true in the case where a global set of conditions sets "admin" to include. Need to validate whether the language addresses this; my interpretation is that the global condition can only win for "exclude", not "include". I feel this was discussed in the early reviews. Otherwise it is not possible to selectively filter, even while saying "Overall at a publication level, this publication includes admin info".

None of the resources that are filtered by `novice.ditaval` are renamed. This ensures that external links to those topics are stable, regardless of whether the DITAVAL is used.

Example: Multiple `<ditavalref>` on a branch

Multiple `<ditavalref>` elements may be used on a single map branch to create multiple distinct copies of the branch.

Consider the following DITA map which contains a branch with three peer `<ditavalref>` elements; two of the `<ditavalref>` elements contain sub-elements that specify how effective resource names are constructed:

```
<map>
  <topicref href="intro.dita"/>
  <!-- Begining of installing branch --!>
  <topicref href="install.dita">
    <ditavalref href="win.ditaval"/>
    <ditavalref href="mac.ditaval">
      <ditavalmeta><dvrResourceSuffix>-apple</dvrResourceSuffix></ditavalmeta>
    </ditavalref>
    <ditavalref href="linux.ditaval">
      <ditavalmeta><dvrResourceSuffix>-linux</dvrResourceSuffix></ditavalmeta>
    </ditavalref>
    <topicref href="do-stuff.dita">
      <topicref href="mac-specific-stuff.dita" platform="mac"/>
    </topicref>
    <!-- End of installing branch --!>
    <topicref href="cleanup.dita"/>
  </topicref>
</map>
```

Figure 3: `input.ditamap`

```
<val>
  <prop att="platform" val="win" action="include"/>
  <prop att="platform" action="exclude"/>
</val>
```

Figure 4: Contents of `win.ditaval`

```
<val>
  <prop att="platform" val="mac" action="include"/>
  <prop att="platform" action="exclude"/>
</val>
```

Figure 5: Contents of `mac.ditaval`

```
<val>
  <prop att="platform" val="linux" action="include"/>
  <prop att="platform" action="exclude"/>
</val>
```

Figure 6: Contents of `linux.ditaval`

When a processor evaluates this markup, it results in three copies of the installing branch. The following processing takes place:

- The first topic (`intro.dita`) is published normally, potentially using other DITAVAL conditions that are specified externally.
- The installing branch appears three times, once for each DITAVAL. The branches are created as follows:

- The first branch uses the first DITAVAL file (`win.ditaval`). Resources use their original names as specified in the map. The `mac-specific-stuff.dita` topic is removed. The resulting branch, with indenting to show the hierarchy, matches the original without the mac topic:

  ```
  install.dita
     do-stuff.dita
       ...more topics and nested branches...
     cleanup.dita
  ```
- The second branch uses the second DITAVAL file (`mac.ditaval`). Resources are renamed based on the `<dvrResourceSuffix>` element. The `mac-specific-stuff.dita` topic is included. The resulting branch, with indenting to show the hierarchy, is as follows:

  ```
  install-apple.dita
     do-stuff-apple.dita
        mac-specific-stuff-apple.dita
        ...more topics and nested branches...
     cleanup-apple.dita
  ```
- The third branch uses the last DITAVAL file (`linux.ditaval`). Resources are renamed based on the `<dvrResourceSuffix>` element. The `mac-specific-stuff.dita` topic is removed. The resulting branch, with indenting to show the hierarchy, is as follows:

  ```
  install-linux.dita
     do-stuff-linux.dita
       ...more topics and nested branches...
     cleanup-linux.dita
  ```

The example used three DITAVAL files to avoid triple maintenance of the installing branch in a map; the following map is functionally equivalent, but it requires parallel maintenance of each branch.

```
<map>
  <topicref href="intro.dita"/>
  <!-- Windows installing branch -->
  <topicref href="install.dita">
    <ditavalref href="win.ditaval"/>
    <topicref href="do-stuff.dita">
      <!-- more topics and nested branches -->
    </topicref>
    <topicref href="cleanup.dita"/>
  </topicref>
  <!-- Mac installing branch -->
  <topicref href="install.dita">
    <ditavalref href="mac.ditaval">
      <ditavalmeta><dvrResourceSuffix>-apple</dvrResourceSuffix></ditavalmeta>
    </ditavalref>
    <topicref href="do-stuff.dita">
      <topicref href="mac-specific-stuff.dita" platform="mac"/>
      <!-- more topics and nested branches -->
    </topicref>
    <topicref href="cleanup.dita"/>
  </topicref>
  <!-- Linux installing branch -->
  <topicref href="install.dita">
    <ditavalref href="linux.ditaval">
      <ditavalmeta><dvrResourceSuffix>-linux</dvrResourceSuffix></ditavalmeta>
    </ditavalref>
    <topicref href="do-stuff.dita">
      <!-- more topics and nested branches -->
    </topicref>
    <topicref href="cleanup.dita"/>
  </topicref>
  <!-- Several chapters worth of other material -->
</map>
```

Figure 7: `input.ditamap`

Example: `<ditavalref>` as a child of `<map>`

Using `<ditavalref>` as a direct child of the `<map>` element is equivalent to setting filter conditions for that map.

The following map is equivalent to processing all contents of the map with the conditions in the `novice.ditaval` file. If additional conditions are provided externally (for example, as a parameter to the publishing process), the global conditions take precedence.

```
<map>
  <title>Sample map</title>
  <ditavalref href="novice.ditaval"/>
  <!-- lots of content -->
</map>
```

Example: `<ditavalref>` in a reference to a map

Using `<ditavalref>` in a reference to a map is equivalent to setting filter conditions for the referenced map.

In the following example, `other.ditamap` is pulled into another map. The `<ditavalref>` indicates that all of the content in `other.ditamap` should be filtered using the conditions in `some.ditaval`.

```
<topicref href="parent.dita">
  <topicref href="other.ditamap" format="ditamap">
    <ditavalref href="some.ditaval"/>
  </topicref>
</topicref>
```

Figure 8: Map fragment

```
<map>
  <topicref href="nestedTopic1.dita">
    <topicref href="nestedTopic2.dita"/>
  </topicref>
  <topicref href="nestedTopic3.dita"/>
</map>
```

Figure 9: Contents of `other.ditamap`

This markup is functionally equivalent to applying the conditions in `some.ditaval` to the topics that are referenced in the nested map. For the purposes of filtering, it could be rewritten in the following way; the extra `<topicgroup>` container is used here to ensure filtering is not applied to `parent.dita`, as it would not be in the original example:

```
<topicref href="parent.dita">
  <topicgroup>
    <ditavalref href="some.ditaval"/>
    <topicref href="nestedTopic1.dita">
      <topicref href="nestedTopic2.dita"/>
    </topicref>
    <topicref href="nestedTopic3.dita"/>
  </topicgroup>
</topicref>
```

For the purposes of filtering, this map could also be rewritten as:

```
<topicref href="parent.dita">
  <topicref href="nestedTopic1.dita">
    <ditavalref href="some.ditaval"/>
    <topicref href="nestedTopic2.dita"/>
  </topicref>
  <topicref href="nestedTopic3.dita">
    <ditavalref href="some.ditaval"/>
  </topicref>
</topicref>
```

Example: `<ditavalref>` within a branch that already uses `<ditavalref>`

When a branch is filtered due to a `<ditavalref>` element, another `<ditavalref>` deeper within that branch may supply additional conditions for a subset of the branch.

In the following map fragment, a set of operating system conditions applies to installation instructions. Within that common branch, a subset of content applies to different audiences.

```
<topicref href="install.dita">
  <ditavalref href="linux.ditaval"/>
  <ditavalref href="mac.ditaval">
    <ditavalmeta><dvrResourceSuffix>-mac</dvrResourceSuffix></ditavalmeta>
  </ditavalref>
  <ditavalref href="win.ditaval">
    <ditavalmeta><dvrResourceSuffix>-win</dvrResourceSuffix></ditavalmeta>
  </ditavalref>
  <topicref href="perform-install.dita">
    <!-- other topics-->
  </topicref>
  <!-- Begin configuration sub-branch -->
  <topicref href="configure.dita">
    <ditavalref href="novice.ditaval">
      <ditavalmeta><dvrResourceSuffix>-novice</dvrResourceSuffix></ditavalmeta>
    </ditavalref>
    <ditavalref href="advanced.ditaval">
      <ditavalmeta><dvrResourceSuffix>-admin</dvrResourceSuffix></ditavalmeta>
    </ditavalref>
    <!-- Other config topics -->
  </topicref>
  <!-- End configuration sub-branch -->
</topicref>
```

In this case, the effective map contains three instances of the complete branch. The branches are filtered by operating system. Within each operating system instance, the configuration sub-branch is repeated; it is filtered once for novice users and then again for advanced users.

As a result, there are actually six instances of the configuration sub-branch:

1. The first instance is filtered using the conditions in `linux.ditaval` and `novice.ditaval`. For this instance, the resource `configure.dita` is treated as the resource `configure-novice.dita`. There is no renaming based on `linux.ditaval`, and `novice.ditaval` adds the suffix "-novice".
2. The second instance is filtered using the conditions in `linux.ditaval` and `advanced.ditaval`. For this instance, the resource `configure.dita` is treated as the resource `configure-admin.dita`. There is no renaming based on `linux.ditaval`, and `advanced.ditaval` adds the suffix "-admin".
3. The third instance is filtered using the conditions in `mac.ditaval` and `novice.ditaval`. For this instance, the resource `configure.dita` is treated as the resource `configure-mac-novice.dita`. The `mac.ditaval` resource adds the suffix "-mac", resulting in `configure-mac.dita`, and then `novice.ditaval` adds the additional suffix "-novice".
4. The fourth instance is filtered using the conditions in `mac.ditaval` and `advanced.ditaval`. For this instance, the resource `configure.dita` is treated as the resource `configure-mac-admin.dita`. The `mac.ditaval` resource adds the suffix "-mac", resulting in `configure-mac.dita`, and then `advanced.ditaval` adds the additional suffix "-admin".

5. The fifth instance is filtered using the conditions in `win.ditaval` and `novice.ditaval`. For this instance, the resource `configure.dita` is treated as the resource `configure-win-novice.dita`. The `win.ditaval` resource adds the suffix "-win", resulting in `configure-win.dita`, and then `novice.ditaval` adds the additional suffix "-novice".
6. The sixth instance is filtered using the conditions in `win.ditaval` and `advanced.ditaval`. For this instance, the resource `configure.dita` is treated as the resource `configure-win-admin.dita`. The `win.ditaval` resource adds the suffix "-win", resulting in `configure-win.dita`, and then `advanced.ditaval` adds the additional suffix "-admin".

Example: `<ditavalref>` error conditions

It is an error condition when multiple non-equivalent copies of the same file are created with the same resource name.

The following map fragment contains several error conditions that result in name clashes:

```
<topicref href="a.dita" keys="a">
  <ditavalref href="one.ditaval"/>
  <ditavalref href="two.ditaval"/>
  <topicref href="b.dita" keys="b"/>
</topicref>
<topicref href="a.dita"/>
<topicref href="c.dita" keys="c">
  <ditavalref href="one.ditaval">
    <ditavalmeta>
      <dvrResourceSuffix>-token</dvrResourceSuffix>
    </ditavalmeta>
  </ditavalref>
  <ditavalref href="two.ditaval">
    <ditavalmeta>
      <dvrResourceSuffix>-token</dvrResourceSuffix>
    </ditavalmeta>
  </ditavalref>
</topicref>
```

In this sample, the effective map that results from evaluating the branch filter conditions will have several clashes. In some cases the same file must be processed with conflicting conditions, using the same URI. In addition, because no key scope is added or modified, keys in the branch are duplicated in such a way that only one version is available for use. Evaluating the filtered branches in this sample will result in the following equivalent map:

```
<topicref href="a.dita" keys="a"> <!-- a.dita to be filtered by one.ditaval -->
  <topicref href="b.dita" keys="b"/>  <!-- b.dita to be filtered by one.ditaval -->
</topicref>
<topicref href="a.dita" keys="a"> <!-- a.dita to be filtered by two.ditaval; key "a" ignored -->
  <topicref href="b.dita" keys="b"/>  <!-- b.dita to be filtered by two.ditaval; key "b" ignored -->
</topicref>
<topicref href="a.dita"/>
<topicref href="c-token.dita" keys="c">
  <!-- c-token.ditaval to be filtered by one.ditaval -->
</topicref>
<topicref href="c-token.dita" keys="c">
  <!-- c-token.ditaval to be filtered by two.ditaval, key "c" ignored -->
</topicref>
```

As an additional example, in a situation where resource file names map directly to generated XHTML file names, this map results in the following conflicts:

1. `a.dita` generates `a.html` using three alternate set of conditions -- from `one.ditaval`, `two.ditaval`, and no ditaval.
2. `b.dita` generates `b.html` using two alternate set of conditions -- from `one.ditaval` and `two.ditaval`.

3. `c.dita` generates `c-token.html` using both extra sets of conditions.

Index