Archived:Advanced filters

Archived: This article is **archived** because it is not considered relevant for third-party developers creating commercial solutions today. If you think this article is still relevant, let us know by adding the template {{ReviewForRemovalFromArchive|user=~~~~|*write your reason here*}}.

Symbian Web Runtime and Nokia Asha Web Apps have superseded the Widsets platform.

Introduction

Advanced filters are used in conjunction with some WidSets services to

- extract desired parts from input text data, and
- build an output of a desired format from the extracted parts.

Advanced filters are defined in the widget-definition XML-file that resides on server side. The filter execution is also done on server side. The client does not need to know about the filters defined for the widget. The client just calls a service that internally utilizes any filters defined for it.

Advanced filters and services

Currently advanced filters can be used in conjunction with the HTTP and webfeed services.

- The *HTTP service* normally performs an HTTP GET or POST method on a specific URL and returns the result to the client as is. If, however, the HTTP service has a filter defined, then the result from the HTTP GET/POST method is forwarded to the defined filter, and the output from the filter is returned to the calling client.
- The *webfeed service* implements a customizable Web feed handler. The service provides a customizable mechanism for extracting and returning the desired fields from the feed items of a Web feed. The service provides a set of standard return fields and also supports custom return fields. The desired standard or custom fields are listed in the widget definition XML.

Advanced filter output

In general, the output of an advanced filter can be anything. However, when targeting the output to the client, the output should be either an item or a list created with the item or list expression, respectively.

Items represent named or nameless string or integer values to be sent to the client. Lists represent named or nameless lists containing items and other lists. Lists are used to produce structured values to be sent to the client.

A nameless item or list is a plain item or list.

Example:

- “foobar” : a nameless item with the value “foobar”.
- (“foo”, “bar”, (“john”, “doe”)) : a nameless list containing the items “foo” and “bar” and the list (“john”, “doe”).

A named item or list is a name-value pair where the name holds the string valued name and the value holds the plain item or list. A pair or binding like this is usually represented as ‘name = value’.

Example:

- “title” = “foobar” : a named item with the name “title” and value “foobar”.
- “A” = (“foo”, “bar”, (“john”, “doe”)) : a named list with the name “A” and value (“foo”, “bar”, (“john”, “doe”)).

Named and nameless items and lists can appear as list members.

Example:

- “item001” = (“title” = “foo”, “content” = “bar”, “details” = (“acme”, “roadrunner”))

The values produced by item and list expressions are called targets because they are the values sent to the client. The values produced by other expressions are called non-targets, and they are used as elementary values when building targets and other

non-targets.

Filters and filter expressions

The advanced filters used by a widget are defined in the <filters> section of the widget definition XML file.

Example:

```
<filters>
  <filter id="foobar">
    ...
  </filter>
  <filter id="example">
    ...
  </filter>
  ...
</filters>
```

An advanced filter must have an id attribute so that it can be referred to. An advanced filter does not and must not have a type attribute. The type attribute is used with simpler basic filters. If type is present, the WidSets software interprets the filter as a basic filter. The filter definition within the <filter> element consists of a nested hierarchy of filter expression definitions.

Example:

```
<filters>
  <filter id="get_images">
    <list name="images">
      <foreach index="i">
        <regex>
          <![CDATA[
            <img[ ]+src=\"?([^ \"]*?)[\" ].*?>
          ]]>
        </regex>
        <item name="%%%IMAGE${i}%%%">
          <regex_match group="1"/>
        </item>
      </foreach>
    </list>
  </filter>
  ...
</filters>
```

A filter expression definition consists of

- the expression tag,
- the attributes of the expression, if any, and
- the actual expression formula or value definition of the expression, if any.

For example, the filter expression

```
<item name=”title”>
  foobar
</item>
```

that produces the named item “title” = “foobar” consists of

- the expression tag item,
- the attribute name=”title”, and
- the value ”foobar”.

Often child expressions are used to produce the value of an expression. For example, the named item “title” = “foobar” could be produced also with the following expression:

```
<item name=”title”>
  <strcat>
    <str>foo</str>
    <str>bar</str>
  </strcat>
</item>
```

The above expression forms the value “foobar” by catenating the strings “foo” and “bar”. Lists can be created by putting the element item and list expressions as child expressions of the containing list expression.

For example, the list

```
“item001” = (“title” = “foo”, “content” = “bar”, “details” = (“acme”, “roadrunner”))
```

could be created with the following filter expression

```
<list name=”item001”>
  <item name=”title”>foo</item>
  <item name=”content>bar</item>
  <list name=”details>
    <item>acme</item>
    <item>roadrunner</item>
  </list>
</list>
```

Filter evaluation

The filter expression structure reflects the evaluation order and output structure of the filter. The filter expressions are evaluated in top-down and depth-first order. The hierarchy of the target expressions, that is, item and list expressions, specifies the structure of the output value of the filter.

The input to the filter goes as input to the first filter expression of the filter and starts its flow through the expression hierarchy. A filter expression uses the input it gets to build the result. If an expression has any child expressions, then usually the expression calls its first child expression and passes the input to it "as is". If there are more than one child expressions, then usually the child expressions form an implied pipeline of expressions. A pipeline works basically so that an expression in the pipeline passes its result as an input to the next expression in the pipeline, and the result of the last expression in the pipeline will be returned to the parent expression.

The target expressions in the pipeline, that is, item and list expressions, behave differently from non-target expressions. A target expression always forms an implied tee-branch. The tee-branch works so that the input is both passed to the target expression and also forwarded as is to the next sibling expression. The result of a target expression is not forwarded to the pipeline, but is returned to the parent expression.

In general, if other than a list expression receives a target result from a descendant expression, then the expression returns the result further to its parent. A list expression will catch the target result. Thus, the result of a target expression will be caught by the closest ancestor list expression, if there are any. The catching list expression will add the caught target value (item or list) to the values of the list. The result of the outermost target expression will become the value of the filter.

For example, if we had the following RSS feed (only head shown in the sample) as input

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<rss version="2.0">
  <channel>
    <title>YLE 24 uutiset - Pääuutiset</title>
    <description>YLE 24 uutiset - Pääuutiset</description>
    <link>http://www.yle.fi/uutiset</link>
    <language>fi-FI</language>
    <copyright>YLE24</copyright>
    <ttl>1</ttl>
    <managingEditor>ari.jarvinen@yle.fi</managingEditor>
    <WebMaster>yle24.palaute@yle.fi</WebMaster>
    ...
```

and we would like to create the list

```
(“title” = “YLE 24 uutiset – Pääuutiset”, “link” = “http://www.yle.fi/uutiset”)
```

we could use the following filter

```
1   <filter id=”sample”>
2     <list>
3       <xpath>/rss</xpath>
4       <xpath>channel</xpath>
5       <item name=”title”>
6         <xpath>title/text()</xpath>
7       </item>
8       <item name=”link”>
9         <xpath>link/text()</xpath>
10      </item>
11    </list>
12  </filter>
```

The filter evaluation proceeds as follows:

1. The input XML is given as input to the filter.
2. The filter calls the list expression on line 2 passing the input XML as input.
3. The child expressions of the list expression form a pipeline. The list expression calls the first expression of the pipeline, that is, the xpath expression on line 3, passing the input XML as input.
4. The absolute xpath expression on line 3 selects all child nodes of the <rss> XML node and passes the resulting node set (A) to the next expression in the pipeline, that is, to the xpath expression on line 4.
5. The relative xpath expression on line 4 picks from the input node set (A) the <channel> node and selects all child nodes of the <channel> XML node and passes the resulting node set (B) to the next expression in the pipeline, that is, to the item expression starting on line 5.
6. The item expression on lines 5–7 sets the item name to “title” and calls the child xpath expression on line 6 passing the input node set (B) as input.
7. The relative xpath expression on line 6 picks from the input node (B) set the <title> XML node and selects the text child nodes, that is, the value of the node, and returns the value as string.
8. The calling item expression on lines 5–7 gets the return string value and sets it as a value of the item.
9. The item expression on lines 5–7 returns the resulting item up in expression hierarchy. The item is caught by the parent list expression starting on line 2. The list expression adds the item to the list.
10. The item expression on lines 5–7 forms an implicit tee-branch. Thus the expression passes the input node set (B) it got to the next expression in the pipeline, that is, to the item expression starting on line 8.
11. The item expression on lines 8–10 sets the item name to “link” and calls the child xpath expression on line 9 passing the input node set (B) as input.
12. The relative xpath expression on line 9 picks from the input node set (B) the <link> XML node and selects the text child nodes, that is, the value of the node, and returns the value as string.

13. The calling item expression on lines 8–10 gets the return string value and sets it as a value of the item.
14. The item expression on lines 8–10 returns the resulting item up in the expression hierarchy. The item is caught by the parent list expression starting on line 2. The list expression adds the item to the list.
15. The item expression on lines 8–10 forms an implicit tee-branch but is also the last expression in the pipeline. Thus the input node set (B) is returned as a result of the pipeline to the caller of the pipeline, that is, to the list expression starting on line 2. The list expression does not care about non-target values and thus the return value is ignored.
16. The list value created by the list expression starting from line 2 becomes the result of the filter.

Note. The lines 3–4 of the example filter can be combined to a single xpath expression:

```
<xpath>/rss/channel</xpath>
```

- In the example, the separate xpath expressions were used to demonstrate a pipeline with non-target expressions.

Sibling expressions, pipelines, and tees

Usually sibling expressions, that is, expressions with the same parent, form an implied pipeline. Some expressions behave differently, though.

The strcat expression is used to catenate a number of strings. The catenated strings are specified as the child expressions of the strcat expression. The child expressions do not form a pipeline, but the strcat expression calls the child expressions one by one and catenates the string results returned.

For example, consider the same example input that was used in section Filter evaluation. With that input the following expression

```
<strcat>
  <xpath>/rss/channel/title/text()</xpath>
  <str> by </str>
  <xpath>/rss/channel/managingEditor/text()</xpath>
</strcat>
```

returns the string

```
“YLE 24 uutiset – Pääuutiset by ari.jarvinen@yle.fi”
```

Another expression that does not handle child expressions as a pipeline is the choice expression. The choice expression is used to select one expression from several alternatives. The choices are specified as child expressions. The first child expression that evaluates to non-null is selected as the value of the choice expression. Xpath and regular expressions are typically used in child expressions.

Example:

```
<choice>
  <xpath>/rss</xpath>
  <xpath>/rdf:RDF</xpath>
  <xpath>/feed</xpath>
</choice>
```

Also the Archived:Advanced filters#foreach expression handles child expressions in a special way. See section Archived:Advanced filters#foreach expression for details about the foreach expression.

In addition to implied pipelines and tee-branches, there are also explicit pipeline and tee expressions. Normally, there should be seldom any reason to use them, but they might be useful in some cases.

Expression attributes

Attributes of an expression can be defined like normal XML element attributes, or as child elements using the attribute name as a

tag.

For example, a name attribute for an item expression can be defined as

```
<item name="foobar">
  ...
</item>
```

or as

```
<item>
  <name>foobar</name>
  ...
</item>
```

If a static value for an attribute is not sufficient, you can usually use an expression as the value. When the attribute expression is evaluated, it gets the same input as the parent expression.

Expression value

The expression formula or value is by default defined by the child element(s) of the expression element or by the text value of the expression element.

For example, the expression

```
<item>foobar</item>
```

produces the same result as the expression

```
<item>
  <str>foobar</str>
</item>
```

as does the expression

```
<item>
  <strcat>
    <str>foo</str>
    <str>bar</str>
  </strcat>
</item>
```

If you need to use child elements to define both the value of the expression and any attributes of the expression, you must put the value defining child elements or text value within a value tag.

For example, an item with the name "<dummy>" and value "foobar" can be defined as:

```
<item>
  <name>
    <![CDATA[
      <dummy>
    ]]>
  </name>
  <value>foobar</name>
</item>
```

Also, a simple enough value can be defined using a value attribute:

```
<item value="foobar">
  <name>
    <![CDATA[
      <dummy>
    ]]>
  </name>
</item>
```

Expression result

An expression basically takes input and usually produces a result. Some expressions can also produce "side results".

If a non-target expression is part of an expression pipeline and is not the last one in the pipeline, then the result of the expression goes as input to the next expression in the pipeline. Otherwise the result of an expression is returned to the parent expression. If the result is a target and the parent is not a list expression, then the parent expression returns the result further up in the expression hierarchy until an ancestor expression catches it and uses it to build its own result. Non-target results do not “bubble up” like target results in most cases, but the non-target results are either used or just ignored by the parent expression.

The Archived:Advanced filters#foreach expression expression handles results from descendant expressions a bit differently. See section Archived:Advanced filters#foreach expression for details on the foreach expression.

Targeting expression results

In some cases using the expression hierarchy may not be enough to define how the expression results should flow and build up into the result of a filter. Sometimes the result of an expression needs to be returned to a specific catching expression different from the default one. This can be done by using the to attribute available in some expressions. If an expression has the *to* attribute defined, then the result of the expression is returned to the referred-to expression specified by the value of the attribute. The *to* attribute value should be a reference to the desired catching expression. The reference value can be the value of the id or name attribute of the referred-to expression. When the *to* attribute is used, it overrides any default returning and catching of results.

Not everything can be referred to from anywhere. There are some visibility rules.

Visibility rules

When referring from one expression to another, for example, by using the *to* attribute, there are some visibility rules that constrain what can be referred to. The visibility rules are as follows:

We can refer from expression B to expression A if

- 1. A appears before B in the evaluation order (normal reading order).

– AND –

- 2. A is a sibling of B.

– OR –

- A is an ancestor of B

– OR –

- A is a sibling of an ancestor of B.

Example:

```
<list id="A">
  <list id="B">
    <item id="C">
      Can refer to C, B, A. Cannot refer to D, E, F, G.
    </item>
    <item id="D">
```

```
      Can refer to D, C, B, A. Cannot refer to E, F, G.
    </item>
  </list>
  <list id="E">
    <item id="F">
      Can refer to F, E, B, A. Cannot refer to C, D, G.
    </item>
    <item id="G">
      Can refer to G, F, E, B, A. Cannot refer to C, D.
    </item>
  </list>
</list>
```

Regular expressions

Regular expressions can be used to

- extract desired parts from input strings, and
- replace matching parts in input strings with other strings.

A regular expression is defined by using the *regex* expression. The value definition of a *regex* expression should be the regular expression string or an expression that returns as value the regular expression string.

Example:

```
<regex>
  <![CDATA[
    ([^0-9]*)([0-9]+)([^0-9]*)
  ]]>
</regex>
```

The used regular expression syntax is the same as in the *java.util.regex* package. The *regex* expression returns a regular expression matcher. The matcher can be given as input to the *regex_match* expression. The *regex_match* expression can be used to extract and return a desired catching group or the whole match. The desired catching group or the whole match is specified by using the group attribute. The attribute value group=”0” means a full match.

For example, the pipeline

```
<str>foo66bar9</str>
<regex>
  <![CDATA[
    ([^0-9]*)([0-9]+)([^0-9]*)
  ]]>
</regex>
<regex_match group=”0”/>
```

returns the string:

```
“foo66bar”
```

whereas the pipeline

```
<str>foo66bar9</str>
<regex>
  <![CDATA[
    ([^0-9]*)([0-9]+)([^0-9]*)
```

```
  ]]>
</regex>
<regex_match group=”1”/>
```

returns the string:

```
“foo”
```

A bit more complex example

```
<str>foo66bar9</str>
<regex>
  <![CDATA[
    ([^0-9]*)([0-9]+)([^0-9]*)
  ]]>
</regex>
<strcat>
  <regex_match group=”1”/>
  <regex_match group=”3”/>
</strcat>
```

returns the string:

```
“foobar”
```

The *regex* expression can be used also to replace the matching parts in the input string with replacement strings. The replacement string that should replace the matching part of the input string is specified with the attribute replacement. Note that the replacement is applied to the whole match. The *regex* expression returns the input string with the replacement strings applied as a side result; but the matcher is still the primary result of the *regex* expression that is forwarded to pipelines etc. The side result string value (non-target) is by default returned to the parent expression of the *regex* expression to catch. A different destination can be specified by using the *replace_to* attribute of the ,,regex,, expression. Note that the replace functionality is applied only when the *regex* expression is used in conjunction with the Archived:Advanced filters#foreach expression expression. See Archived:Advanced filters#foreach expression for more details.

foreach expression

Most of the filter expressions are normal value generating expressions, and some expressions are control structures. The most important one is the *foreach* expression. The *foreach* expression is used to loop through a set of values, to perform operations, and to produce output values for each value.

The syntax of the *foreach* expression is as follows:

```
<foreach>
  values-expression
  do-expressions
</foreach>
```

The first child expression of the *foreach* expression is the values-expression. The other expressions are *do* expressions. The sibling *do* expressions form an implied pipeline.

The values-expression specifies the value set to be looped through. The *do* expressions specify the expressions that are evaluated for each value in the value set. The *do* expressions part is optional – sometimes the side results from the values-expression are sufficient.

Usually the values-expression is either an *xpath* or *regex* expression. They both produce a value set that can be looped through. An *xpath* expression produces an XML *nodeset*, and on each round of the *foreach* loop the next node is given as input to the *do*

expressions part. A *regex* expression produces a regular expression matcher which on each round of the loop finds the next match and updates its “current match” state. The *regex_match* expression can then be used in the do-expressions part to access the current match. If the *regex* expression has a replacement attribute specified, then on each round of the loop the replacement string is applied to the current match in the input string.

By default, the *foreach* expression does not have a return value. When executing the do-expressions, any non-caught target and non-target values returned from the do-expressions are returned to the parent expression of the *foreach* expression.

The *foreach* expression can have a return value, but the value must be specifically set. This is done by specifying an id for the *foreach* expression with the id attribute and by returning a result to the *foreach* expression from one of the child expressions by using the *to* attribute, or the *replace_to* attribute of the regex expression.

Let’s see some examples of foreach/regex/xpath usage.

The following expression snippet

```
<item>
  <str>foo66bar9</str>
  <foreach>
    <regex replacement=”#”>
      <![CDATA[
        [0-9]+
      ]]>
    </regex>
  </foreach>
</item>
```

produces a nameless item with the value

```
“foo#bar#”
```

If the input is an RSS [2.0] feed, then the following filter will create a list of items where each item contains the fields named “title” and “description”:

```
<filter id=”rssitems”>
  <list>
    <foreach>
      <xpath>/rss/channel/item</xpath>
      <item name=”title”>
        <xpath>title/text()</xpath>
      </item>
      <item name=”description”>
        <xpath>description/text()</xpath>
      </item>
    </foreach>
  </list>
</filter>
```

The following filter expression snippet takes as input a string containing HTML. The expressions produce two targets: a list named “images” that contains all the

```
<img>
```

URL values in the input, and an item named “content” that has as the value the input string with all the

```
<img>
```

tags replaced with the string “IMAGE” and all other tags stripped:

```
...
<list name="images"/>
<foreach id="loop">
  <regex replace_to="loop" replacement="IMAGE">
    <![CDATA[
      <img[ ]+src=\"?([^ \"]*?)[\" ].*?>
    ]]>
  </regex>
  <item to="images">
    <regex_match group="1"/>
  </item>
</foreach>
<item name="content">
  <foreach>
    <regex replacement="">
      <![CDATA[
        (?s)<[a-zA-Z/].*?>
      ]]>
    </regex>
  </foreach>
</item>
...
```

State variables

There are two kinds of *state variables*:

- expression-specific state variables and
- free state variables.

An expression specific state variable is specific to the expression, like the index state variable of the *Archived:Advanced filters#foreach expression*. The expression updates the expression specific state variable automatically.

Currently only the *foreach* expression has one expression specific state variable. The index state variable keeps track of the index of the *Archived:Advanced filters#foreach expression* loop.

A free state variables is defined by the user with the *var* expression. The user is responsible for updating the value of a free state variable. The value can be set by targeting a result of an expression to the state variable with the *to* attribute.

You can refer to state variables from within strings used in expressions by using the following syntax:

```
${state-var-symbolic-name}
```

The state-var-symbolic-name is a symbolic name you have given to the state variable. Another way to get the value of the state variable is by using the get expression.

The state variable is specified with the id attribute of the get expression, like

```
<get id=”state-var-symbolic-name”/>
```

Note. A *get* expression without the id attribute returns the current input value. (This is actually a useful feature.)

The visibility rules are valid also for state variable references.

A symbolic name is given to an expression specific state variable by using the state variable tag as an XML attribute tag and the symbolic name as the attribute value.

For example, giving the symbolic name "i" to the state variable index of a *foreach* expression:

```
<foreach index="i">
```

Symbolic names are needed in the case there are several same kinds of expression specific state variables visible, for example, *foreach* index-variables.

A free state variable is defined with the *var* expression. The symbolic name is defined with the id attribute.

Example:

```
<var id=”temp” value=””/>
```

Note that the result of the *var* expression is the input value to the *var* expression. Thus the *var* expression acts like a *tee* expression.

Here is an example of an expression specific state variable usage.

```
<list>
  <foreach index="i">
    <regex>
      <![CDATA[
        <img[ ]+src=\"?([^ \"]*?)[\" ].*?>
      ]]>
    </regex>

    <item name="%%%IMAGE${i}%%%">
      <regex_match group="1"/>
    </item>
  </foreach>
</list>
```

This code generates a list of named items, named as **"%%%IMAGE0%%%", "%%%IMAGE1%%%”, etc.**.

Iterator

The *iterator* expression is another way to loop through a set of values. It differs a bit from the *foreach* expression. The *iterator* expression gets the *iterator* from the input value it gets. The *iterator* expression has two child expressions, next and *after_last*. The next expression, if defined, is evaluated for each iterated non-null value. The *after_last* expression, if defined, is evaluated after the last iteration if the last iterated value is non-null; the non-null last value is given as input to the *after_last* expression.

Below is an example expression sequence that utilizes the *iterator* expression. The code snippet gets a description text element of a RSS feed item and decomposes the text string to a list of text and image url segments. Commets are written as "# comment".

```
<filter id="content">
  <xpath>description/text()</xpath>

  # Store the description string for later use.
  <var id="regex_input">
    <get/>
  </var>

  <list name="content">
    # Get a regular expression matcher that can be used to split
    # the description string by the image urls – or if there are
    # no urls, a matcher that matches the whole string.
    <choice>
```

```
      <regex>
        <![CDATA[
          (?s)<img(?:[ ]+|(?:[ ]+[^>]*?[ ]+))src=['\"]?([^ '\"]*?)['\" ].*?>
        ]]>
      </regex>
      <regex>
        <![CDATA[
          (?s)$
        ]]>
      </regex>
    </choice>

    # State variable to hold the previous match end index.
    <var id="previous" value="0"/>

    # The input to the iterator expression is one of the
    # regular expression matchers!
    <iterator>
      <next>
        # Try to get the next text segment.
        <item>
          <substr>
            # Get substring from the description string
            <source>
              <get id="regex_input"/>
            </source>
            <start value="${previous}"/>
            <end>
              <regex_match_start group="0"/>
            </end>
          </substr>
          <entity_decode/>
          <untag/>
        </item>

        # Try to get an image url segment (type=”image, url=”…”)
        <list>
          <choice>
            <regex_match group="1"/>
          </choice>
          <item name="type" value="image"/>
          <item name="url">
            <get/>
          </item>
        </list>

        # Store the match end index for next iteration
        <regex_match_end to="previous" group="0"/>
      </next>

      # Handle the tail – any string after the last image url.
      <after_last>
        <item>
          <substr>
            <source>
              <get id="regex_input"/>
            </source>
```

```
            <start value="${previous}"/>
          </substr>
          <entity_decode/>
          <untag/>
        </item>
      </after_last>
    </iterator>
  </list>
</filter>
```

See also

- Archived:WidSets - Getting content with Services
- Archived:Available content fetching services
  - Archived:Syndication service in WidSets
  - Archived:Webfeed service in WidSets
  - Archived:Image service in WidSets
  - Archived:HTTP service in WidSets
- Archived:Fetcher details in WidSets
  - Archived:Feed formats in WidSets
  - Archived:HTTP authentication in WidSets
- **Archived:Advanced filters**
  - Archived:WidSets Filter expressions reference