Creating Custom XPaths in Browser Tests and XML Tools

The following information uses browser test tools for examples, but the same concepts also apply to creating custom XPaths in the XML tools as well (XML Assertor, Data Bank, etc.).

SOAtest generates element locators for use in browser tests; these locators allow SOAtest to find the appropriate element during playback so that SOAtest can, for example, perform a user action on that element or validate an attribute of that element. SOAtest tries to create a locator that will still work after trivial changes to the web page. However, sometimes you will need to create a custom element locator. SOAtest may not have enough information to generate a robust locator.

When creating custom XPath expressions for complicated HTML pages, it is easier to create the XPath in parts and use the rendered view to verify that you have found the desired element. If you start out with a complicated XPath expression and nothing is highlighted, it is a challenge to determine what part of the XPath is not working.

If you need to create a parameterized XPath, first create a literal XPath and verify that it is highlighted and finds the correct element during playback. Then, create a script to parameterize the XPath. Again, this simplifies debugging.

Creating Custom Element Locators

One common use case for creating custom element locators is when dealing with tables. For example, you may want to click on a link in a table based on the content of another column in that row. You may want to always click on something in the last row of a table that may vary in size. This section outlines how to create custom XPath locators for these situations.

To create custom XPath expressions, you will need to examine the HTML for the given page. Consider a page with the following table.

```
<table border="1">
  <thead>
    <th>number</th>
    <th>comment</th>
    <th>details link</th>
    <th>other link</th>
  </thead>

  <tbody id="content">
    <tr>
      <td>000</td>
      <td>nothing special</td>
      <td><a href="#" id="details000" onclick="foo()">Details</a></td>
      <td><a href="#">Other</a></td>
    </tr>
    <tr>
      <td>123</td>
      <td>nothing special</td>
      <td><a href="#" id="details123" onclick="foo()">Details</a></td>
      <td><a href="#">Other</a></td>
    </tr>
    <tr>
      <td><em>456</em></td>
      <td>formatting: nodes within column</td>
      <td><a href="#" id="details456" onclick="foo()">Details</a></td>
      <td><a href="#">Other</a></td>
    </tr>
    <tr>
      <td> 789  + 1   </td>
      <td>whitespace</td>
      <td><a href="#" id="details789" onclick="foo()">Details</a></td>
      <td><a href="#">Other</a></td>
    </tr>
    <tr>
      <td>888</td>
      <td>find my comment</td>
      <td><a href="#" id="details888" onclick="foo()">Details</a></td>
      <td><a href="#">Other</a></td>
    </tr>
    <tr>
      <td>999</td>
      <td>last row</td>
      <td><a href="#" id="details999" onclick="foo()">Details</a></td>
      <td><a href="#">Other</a></td>
    </tr>
  </tbody>
</table>
```

Here is an ASCII rendering of the table:

```
+-----------------------------------------------------------------------+
| number | comment | details link | other link |
|---------+---------------------------------+--------------+------------|
| 000 | nothing special | Details | Other |
|---------+---------------------------------+--------------+------------|
| 123 | nothing special | Details | Other |
|---------+---------------------------------+--------------+------------|
| 456 | formatting: nodes within column | Details | Other |
|---------+---------------------------------+--------------+------------|
| 789 + 1 | whitespace | Details | Other |
|---------+---------------------------------+--------------+------------|
| 888 | find my comment | Details | Other |
|---------+---------------------------------+--------------+------------|
| 999 | last row | Details | Other |
+-----------------------------------------------------------------------+
```

Finding an Element in a Table

Assume you want to click on the "Details" link in row with "123" in the number column.

The following XPath will work with the current table:

```
/descendant::a[text()='Details'][2]
```

This will find the second link in the document with the text "Details". This will not evaluate to the desired element if any additional links with the text "Details" are added before the "123" row.

Here is a more robust XPath expression to click on the "Details" link in the "123" row:

```
//tbody[@id='content']/tr[td[1]/text() = '123']/descendant::a[text()='Details']
```

Here is a breakdown of each part of the XPath expression:

- `//tbody[@id='content']` : Find a `<tbody>` element anywhere in the document with the `id` attribute equal to "content". The "tbody" is the tag name of an element. Using `@id` references an attribute named "id". The square brackets, "`[]`", define a predicate, which you can think of as search criteria. Within the predicate, the context node is the `<tbody>` element: the `@id` refers to the attribute for the `<tbody>` element.
- `/tr[td[1]/text() = '123']` : Find a child `<tr>` row in the `<tbody>` element in which the first column has text equal to "123". The forward slash (`/`) specifies that the `<tr>` element is a child of the `<tbody>` element.
- Predicate `[td[1]/text() = '123']` : The `<td>` must be the child element of the `<tr>` element because the `<tr>` is the context node. For "td [1]", the "[1]" predicate specifies to use the first child `<td>` element -- that is, the cell in the first column. If you change the order of the columns in the table such that the "number" column is the second column, you would need to change this to "td[2]".
- `/text() = '123'` : Match a child text node of the `<td>` element with the value "123".
- `/descendant::a[text()='Details']` : Find an <a> element contained anywhere within the `<tr>` element with the given text. The "descendant::" specifies the axis to use, where an axis defines the relationship between nodes to use. The "child" axis is the default axis: "/x" is the same as "/child::x". The "descendant" access contains all children, grandchildren, great-grandchildren, etc. The W3C Recommendation defines the "descendant" axis as follows: "the descendant axis contains the descendants of the context node; a descendant is a child or a child of a child and so on; thus the descendant axis never contains attribute or namespace nodes."

The following axes are available. For more information see the W3C Recommendation. Most likely you will primarily use the "child" and "descendant" axes.

- ancestor
- ancestor-or-self
- attribute
- child
- descendant
- descendant-or-self
- following
- following-sibling
- namespace
- parent
- preceding
- preceding-sibling
- self

If the Number Column Contains Elements

The element for the "number" column for the "456" row:

```
<td><em>456</em></td>
```

You can use the previous XPath with a slight modification. If you use the predicate [td[1]/text() = '456'], this will not find any row. The reason is that td[1] /text() evaluates to a child text node of the <td> element. Here the one text node is a descendant but not a child of the <td> element. You can use the predicate [td[1]/descendant::text() = '456']. (You could have used the descendant axis here in the XPath from (1), as a child is a descendant.)

But suppose that the <td> element contained multiple text nodes, something like the following:

```
<td>order <em>456</em></td>
```

The example is contrived, but in a more complicated table, there is a chance that the visible text of the table cell will span multiple text nodes. When that is the case, the predicate [td[1]/descendant::text() = 'order 456'] does not work. The expression td[1]/descendant::text() evaluates to the text nodes "order " and "456"; neither of these is equal to "order 456". That is, text() evaluates to a collection of text nodes, not to the concatenation of those text nodes.

Instead you can use the predicate [td[1] = 'order 456']. When you compare a node with a text value, that node is converted to a string. The W3C Recommendation specifies that "The string-value of an element node is the concatenation of the string-values of all text node descendants of the element node in document order", which is what you want.

Back to our original table cell of <td>456</td>, you can use this XPath to click on the "Details" link in the row with number "456":

```
//tbody[@id='content']/tr[td[1] = '456']/descendant::a[text()='Details']
```

Note that you can use an XPath of this form for the "123" row as well. Arguably this form is more robust.

Dealing with White Space

The "789 + 1" row contains this cell for the number column:

```
<td> 789  + 1  </td>
```

The predicate [td[1] = '789 + 1'] will not find any row, because in XPath expressions white space is significant. You need to include all of the white space characters, so this would work: [td[1] = ' 789 + 1 ']. However, this will not work if the white space ever changes, and note that a newline in the HTML is also significant white space. You can ignore fluctuations in white space by using the normalize-space function, which according to the W3C Recommendation "returns the argument string with white space normalized by stripping leading and trailing white space and replacing sequences of white space characters by a single space."

Thus, you can use the following XPath:

```
//tbody[@id='content']/tr[normalize-space(td[1])='789 + 1']/descendant::a[text()='Details']
```

A number of other XPath string functions exist, such as concat, substring, contains. For more information, see the W3C Recommendation.

Dealing with Column Reordering

The previous XPath expressions will work regardless of where the "details link" column exists in the table. However, the expressions assume that the "number" column is always the first column. To handle a shifting "number" column, you need to modify the predicate td[1]. Instead of the predicate "[1]", the predicate must contain the index of the "number" header cell in the table's <thead> element.

But first: If you are content to search through all columns for a given value, then you can simply remove the index predicate on the <td> element. To find row "888":

```
//tbody[@id='content']/tr[td = '888']/descendant::a[text()='Details']
```

But note that if the string "888" ever appears as the text as another table cell in any column, this XPath may not return what you want. If you want to avoid this issue, do the following.

For this example, use the "comment" column to find the row with number "888" to demonstrate that this XPath does not look in the first column.

If you are searching for the row "888", you can use the following XPath:

```
//tbody[@id='content']/tr[td[count ( ancestor::table/thead/th[.='comment']/preceding-sibling::th ) + 1] = 'find
my comment']/descendant::a[text()='Details']
```

The XPath is the same as in the previous example, except for this new predicate for the <td> element:

```
td[count ( ancestor::table/thead/th[.='comment']/preceding-sibling::th ) + 1]
```

The predicate searches for the <thead> element within the table, finds the table cell with the value "comment", and then determines the index of that column.

- ancestor::table/thead : Use the "ancestor" axis to find the <table> element in which the <td> context node exists. Then use the table's <thead> child element.
- th[.='comment'] : Find the table cell with text equal to "comment". Here the "." is short for self::node(), which evaluates to the <th> element itself. Then the element is converted to a string value because you compare it to the string "comment". Here because the <th> element contains only one text node, you could instead use td[text()='comment'].
- /preceding-sibling::th : Find all <th> elements that are children of the <thead> element and appear before the "comment" column (the context node). Because the "comment" column is the currently second column, this would find one element. If the columns were rearranged to something like the following:

```
<thead>
  <th>details link</th>
  <th>other link</th>
  <th>number</th>
  <th>comment</th>
</thead>
```

  Then /preceding-sibling::th would find three elements.
- count (ancestor::table/thead/th[.='comment']/preceding-sibling::th) + 1

The count function returns the number of nodes returned by evaluating the input expression. In this case, it returns the number of <th> column header cells that appear before <th>comment</th>. Add 1 to the count because XPath indexes are 1-based: if this is the first column, then count returns 0.

This XPath expression is more complicated than you will usually want or need, but it demonstrates what is possible.

If you are experiencing performance problems with Internet Explorer evaluating XPath expressions, see Switching XPath Libraries for Improved Performance or Support to learn about using an alternative XPath library. You can also try a scripted element locator (see Scripting Element Locators).

Finding the Last Row

You want to click on the "Details" link in the last row, regardless of the value in the "number" column. You can use this XPath:

```
//tbody[@id='content']/tr[last()]/descendant::a[text()='Details']
```

The tr[last()] predicate returns the last <tr> child element within the <tbody>.

If you wanted to use the second-to-last row (assuming that the table contained at least two rows) you can use tr[last() - 1]. See the W3C Recommendation for more information on the last() and position() functions.

Using an XPath with a Parameterized Value

In SOAtest you can use a parameterized value, which is a value from a data source, test logic variable, or from an extraction. To create an XPath expression that references a parameterized value, you will need to create a scripted element locator.

Suppose that you want to click on the "Details" link based on a "number" value extracted in a previous test. For example, maybe the row contains a new number generated by your previous user actions. You can extract the new number from a previous browser test by using a Browser Data Bank. To create a browser test that clicks on the link specified by that generated number, use configure the browser test to use a scripted "Element Locator" value in the "User Action" tab. Then you can use a JavaScript script such as the following:

```
var Application = Packages.com.parasoft.api.Application;

// input: com.parasoft.api.BrowserContentsInput.
// context: webking.api.BrowserTestContext.
function scriptedXpath(input, context) {
  // Assumes that you configured the extraction to use column name "extractedNumber".   // Use data source name
"", null, or "Generated Data Source" because the
  // value is from an extraction, not a data source (table, CSV, etc.).
  var number = context.getValue("", "extractedNumber");
  var xpath = "//tbody[@id='content']/tr[td[1] = '" + number + "']/descendant::a[text()='Details']";
  // Output XPath for debugging purposes.
  Application.showMessage("scripted XPath: " + xpath);
  return xpath;
}
```

Scripting Element Locators

If you need to use complicated logic to find an element, a single XPath expression may become unwieldy. Note that when using a scripted element locator, you can return either an XPath string or an org.w3c.dom.Node object. As mentioned, IE can be slow to evaluate complex XPath expressions; using a scripted locator that returns a node may be faster.

A script to find the "Details" link in the last row:

```
var WebBrowserTableUtil = Packages.webking.api.browser2.WebBrowserTableUtil;

// input: com.parasoft.api.BrowserContentsInput.
// context: webking.api.BrowserTestContext.
function scriptedNode(input, context) {
  var document = input.getDocument();
  var tbody = document.getElementById("content");
  // Assumes that there are no rows within rows: this returns
  // all descendants with the given tag, not just children.
  var rows = tbody.getElementsByTagName("tr");
  var lastRow = rows.item(rows.getLength() - 1);
  var detailsColumnIndex = 2; // Zero-based.
  var detailsLink = WebBrowserTableUtil.getLinkFromCell(
    detailsColumnIndex, lastRow);
  return detailsLink;
}
```

Finding the Right Element

If you want to find elements on a web page, add attributes that allow you to find the elements that you need. For example, the <tbody> element used in these examples has a unique "id" attribute value. If the page contained multiple tables and neither the <tbody> nor <table> contained an "id" attribute (or some other combination of attributes that would enable you to uniquely identify the table), then it will be difficult to create simple XPath expressions that use that table. Furthermore, if the element for which you need a locator—for example, a link that you need to click on—already has an "id" attribute then you can either create a locator to "Use Element Properties" or create a very simple XPath, //a[@id='value'].