XPath Tutorial

In this tutorial, you will be given a gentle introduction to XPath, a query language that can be used to select arbitrary parts of HTML documents in calibre. XPath is a widely used standard, and googling it will yield a ton of information. This tutorial, however, focuses on using XPath for ebook related tasks like finding chapter headings in an unstructured HTML document.

Contents

Selecting by tagname

The simplest form of selection is to select tags by name. For example, suppose you want to select all the `<h2>` tags in a document. The XPath query for this is simply:

```
//h:h2        (Selects all <h2> tags)
```

The prefix *//* means *search at any level of the document*. Now suppose you want to search for `<span>` tags that are inside `<a>` tags. That can be achieved with:

```
//h:a/h:span    (Selects <span> tags inside <a> tags)
```

If you want to search for tags at a particular level in the document, change the prefix:

```
/h:body/h:div/h:p (Selects <p> tags that are children of <div> tags that are
             children of the <body> tag)
```

This will match only `<p>A very short ebook to demonstrate the use of XPath.</p>` in the *Sample ebook* but not any of the other `<p>` tags. The `h:` prefix in the above examples is needed to match XHTML tags. This is because internally, calibre represents all content as XHTML. In XHTML tags have a *namespace*, and `h:` is the namespace prefix for HTML tags.

Now suppose you want to select both `<h1>` and `<h2>` tags. To do that, we need a XPath construct called *predicate*. A *predicate* is simply a test that is used to select tags. Tests can be arbitrarily powerful and as this tutorial progresses, you will see more powerful examples. A predicate is created by enclosing the test expression in square brackets:

```
//*[name()='h1' or name()='h2']
```

There are several new features in this XPath expression. The first is the use of the wildcard `*`. It means *match any tag*. Now look at the test expression `name()='h1' or name()='h2'`. *name()* is an example of a *built-in function*. It simply evaluates to the name of the tag. So by using it, we can select tags whose names are either *h1* or *h2*. Note that the *name()* function ignores namespaces so that there is no need for the `h:` prefix. XPath has several useful built-in functions. A few more will be introduced in this tutorial.

Selecting by attributes

To select tags based on their attributes, the use of predicates is required:

```
//*[@style]              (Select all tags that have a style attribute)
//*[@class="chapter"]    (Select all tags that have class="chapter")
//h:h1[@class="bookTitle"] (Select all h1 tags that have class="bookTitle")
```

Here, the `@` operator refers to the attributes of the tag. You can use some of the XPath built-in functions to perform more sophisticated matching on attribute values.

Selecting by tag content

Using XPath, you can even select tags based on the text they contain. The best way to do this is to use the power of *regular expressions* via the built-in function *re:test()*:

```
//h:h2[re:test(., 'chapter|section', 'i')] (Selects <h2> tags that contain the words chapter or
                                           section)
```

Here the `.` operator refers to the contents of the tag, just as the `@` operator referred to its attributes.

Sample ebook

```
<html>
    <head>
        <title>A very short ebook</title>
        <meta name="charset" value="utf-8" />
    </head>
    <body>
        <h1 class="bookTitle">A very short ebook</h1>
        <p style="text-align:right">Written by Kovid Goyal</p>
        <div class="introduction">
            <p>A very short ebook to demonstrate the use of XPath.</p>
        </div>

        <h2 class="chapter">Chapter One</h2>
        <p>This is a truly fascinating chapter.</p>

        <h2 class="chapter">Chapter Two</h2>
        <p>A worthy continuation of a fine tradition.</p>
    </body>
</html>
```

XPath built-in functions

name()

The name of the current tag.

contains()

`contains(s1, s2)` returns *true* if s1 contains s2.

re:test()

`re:test(src, pattern, flags)` returns *true* if the string *src* matches the regular expression *pattern*. A particularly useful flag is `i`, it makes matching case insensitive. A good primer on the syntax for regular expressions can be found at regexp syntax