Choose between XPath and jQuery with an XPath-jQuery phrase book

Have it both ways with powerful access to structured data

Chris Herborth

Introduction

XML is a well-supported Internet standard for encoding structured data in a way that can be easily decoded by practically any programming language and even read or written by humans using standard text editors. Many applications, especially modern standards-compliant Web browsers, can deal directly with XML data.

XPath (the XML Path Language) is a powerful query language for selecting nodes in an XML document. Version 1.0 of the XPath standard is widely implemented in a wide range of languages such as Java™, C#, and JavaScript.

jQuery is a de-facto standard cross-browser JavaScript library for selecting and manipulating nodes in an XHTML document (and in XML documents loaded through Ajax). It has been adopted by a large number of prominent companies including Google, IBM®, Microsoft®, and Twitter. It's current version 1.4 was released as I was writing this article; so I upgraded immediately to take advantage of the promised more speed. Note that the jQuery examples in the article should work unmodified with jQuery 1.3.2, the previous version.
Why use jQuery when XPath exists in JavaScript?

If XPath is a W3C standard, and implementations exist in JavaScript, why bother using jQuery instead?

XPath is a generalized XML standard, while jQuery is a lightweight library designed to deal with the intricacies of cross-browser compatibility so you don't have to worry about which browser your users are running. It's flexible enough to work within the browser's DOM using standard JavaScript idioms, and it provides additional features that make Web application development much less painful, such as powerful Ajax and animation support.

You should, however, always use the right tool for the job at hand; knowing more about these two tools will definitely help you pick the right technology for your next project.

The example

Throughout this article, you'll refer back to a handy sample XML document, which you can find here in Listing 1. This list of books includes various bits of information such as author, a couple of entirely fictional prices and the title.

Listing 1. A sample XML document (book.xml)

```xml
<?xml version="1.0" encoding="utf-8"?>
<catalog>
  <book format="trade">
    <name>Jennifer Government</name>
    <author>Max Barry</author>
    <price curr="CAD">15.00</price>
    <price curr="USD">12.00</price>
  </book>
  <book format="textbook">
    <name>Unity Game Development Essentials</name>
    <author>Will Goldstone</author>
    <price curr="CAD">52.00</price>
    <price curr="USD">45.00</price>
  </book>
  <book format="textbook">
    <name>UNIX Visual QuickPro</name>
    <author>Chris Herborth</author>
    <price curr="CAD">15.00</price>
    <price curr="USD">10.00</price>
  </book>
</catalog>
```

Note that I have no affiliation with the authors and/or publishers, except for the obvious one there. The prices are entirely made up and you should check your favorite book store for actual pricing.

XPath assumptions

For the XPath code in this article, you're going to make these assumptions:

- You've loaded the book.xml file (from Listing 1) into a format that your XPath implementation can use.
• You're starting your searches with an object representing the root of the document. That is, the object that has the <catalog> element as its child. You'll call this root because it's the root of the XML document hierarchy.

Because there are so many XPath implementations on so many different platforms, you'll focus on the XPath statements themselves and use a pseudocode similar to JavaScript to show them in context; check the class library of your favorite development platform for information about loading XML documents and the specific XML node objects you have available.

jQuery assumptions

The jQuery code in this article makes these assumptions:

• You're using the latest (version 1.4.0) jQuery code (see Related topics for a link).
• You've loaded the book.xml file through the jQuery.get() or jQuery.post() method and have stored the resulting XML document in a variable named root (to be the same as your XPath examples).

Some sample code for doing this is in Listing 2.

Listing 2. Loading the XML sample with jQuery

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html;charset=utf-8"/>
    <title>Book Catalog</title>
    <script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/jquery/1.4.0/jquery.min.js"></script>
    <script type="text/javascript">// <![CDATA[
var root = null;
$(document).ready( function(){
    $.get( "http://localhost/~chrish/books.xml",
        function( data ) {
        root = data;
        "#status" ).text( "Loaded." );
        } );
// ]]> </script>
</head>
<body>
<p id="status">
Loading book.xml...
</p>
</body>
</html>
```

In the $(document).ready() function, you use the jQuery get() method to load books.xml from the local Web server, store the resulting document object in the root variable, and set the text of the paragraph with the status ID to indicate that the XML is done loading. For more information about jQuery, check the list of related links in Related topics at the end of the article.
Selecting nodes

The fundamental purpose of both XPath and jQuery is to select nodes from a document. Once you select a node (or a collection of nodes), you can find the data you’re looking for and manipulate the document when you need to.

XPath is designed to return exactly the nodes you’ve asked for; it’s generally very specific. jQuery, on the other hand, makes it very easy to operate on large collections of nodes, so sometimes you’ll have to be careful to narrow down the matches before you start to work through the nodes.

Selecting a node by name

When you search for a specific node, you often know its name, or the name of its parent element.

To find a specific element, you use its name as in Listing 3.

Listing 3. Selecting nodes by name

```javascript
/* Find all <book> elements through XPath: */
var result = root.find( "//book" );

/* Find all <book> elements through jQuery: */
var result = $(root).find( "book" );
```

The XPath statement to select all of the <book> elements (//book) uses two forward slashes (//) to specify that all matching nodes, starting from the current node (root in the example), are to be found. This is the default behavior of jQuery, so you don’t need to include anything else. In both cases, the result will be all three <book> elements from Listing 1.

You can often narrow the search results by specifying a path of elements; the results will be matching nodes from the end of the path (see Listing 4).

Listing 4. Selecting nodes by path—these don’t behave the same

```javascript
/* Be more specific (XPath): */
var result = root.find( "//catalog//book" );

/* Be more specific (jQuery): */
var result = $(root).find( "catalog book" );
```

Starting from the root element (/), this XPath statement will look for the first <catalog> element, and then return all of the <book> elements from that first <catalog>. The jQuery statement behaves a little differently; it will return all <book> elements from all <catalog> elements (see Listing 5). With the example book.xml file, the result is the same set of nodes, but what if you wanted to get all of the <author> elements from the <book> elements? You’d start the XPath expression with two forward slashes (//) like you did in Listing 3.
Listing 5. Pulling out embedded nodes by path—these examples behave the same

/* Get all authors from all books (XPath): */
var result = root.find( "//book//author" );

/* Get all authors from all books (jQuery): */
var result = $(root).find( "book author" );

To make jQuery return the <book> elements from the first <catalog>, like the XPath sample in Listing 4, you have to instruct it to use the first <catalog> it finds (see Listing 6).

Listing 6. Matching the books in the first catalog—these examples behave the same

/* All books from the first catalog (XPath): */
var result = root.find( "/catalog//book" );

/* All books from the first catalog (jQuery): */
var result = $(root).find( "catalog:first book" );

Finding the last occurrence of an element, such as the last list item in a bulleted list, or the last option in a selection list, is also a common operation. To properly append something to the end of the list, you'll need to know the location of that end (see Listing 7).

Listing 7. Finding the last book in the catalog

/* The last book from the first catalog (XPath): */
var result = root.find( "/catalog/book[last()]" );

/* The last book from the first catalog (jQuery): */
var result = $(root).find( "catalog:first book:last" );

In both cases, you get the last <book> element from the first <catalog> element, which is what you were looking for. In the XPath example, the last() function returns the index of the last matched element, which you use in square brackets.

Selecting any node

Sometimes you don't know the name of the element you're looking for, or you need to find an element that might be inside of several different elements. In both XPath and jQuery, you can use an asterisk (*) to match any element (see Listing 8).

Listing 8. The any element

/* Find all authors in all elements inside of <catalog> (XPath): */
var result = root.find( "/catalog/*/author" );

/* Find all authors in all elements inside of <catalog> (jQuery): */
var result = $(root).find( "catalog:first * author" );

Note that I've used :first in the jQuery sample to make it work exactly like the XPath version.
Selecting a node by attribute

Similar elements often have unique attributes, such as the `id` attribute used in XHTML elements to give them a unique reference ID (see Listing 9). Sometimes you don't care as much about the specific element as you do about it having an attribute with a specific value.

Listing 9. Find those pesky textbooks

```javascript
/* Find all books that are textbooks (XPath): */
var result = root.find( "//book[@format='textbook']" );

/* Find all books that are textbooks (jQuery): */
var result = $(root).find( "book[format='textbook']" );
```

Both examples will return all `<book>` elements that have a `format` attribute set to `textbook` (there are two in the book.xml file from Listing 1). XPath's syntax uses an at sign (`@`) to match attributes (jQuery just encloses them in square brackets) and you need to include two forward slashes (`//`) to match all `<book>` elements, but the two queries are very similar and straightforward.

jQuery includes a couple of shortcuts for the two most commonly matched-against attributes (`id` and `class`) in XHTML. In XPath, you'll have to write them out explicitly (see Listing 10).

Listing 10. Matching XHTML based on the `id` and `class` attributes

```javascript
/* Find the "status" <p>, then the highlighted elements (XPath) */
var result1 = xhtml_root.find( "//p[@id='status']" );
var result2 = xhtml_root.find( "//@class='highlight'" );

/* Find the "status" <p>, then the highlighted elements (jQuery) */
var result1 = $( "p#status" );
var result2 = $( ".highlight" );
```

Assuming that your XHTML document is valid (and it is, right?), the ID matching queries will only return one element, because IDs must be unique in a valid XML document.

If you're a fan of Cascading Style Sheets (CSS), you might notice that the jQuery selectors are pretty much the same as CSS selectors. This is handy, because you only need to remember one standard for finding the elements you want through jQuery and for styling them with CSS.

Multiple selectors

Both XPath and jQuery let you combine more than one selector to retrieve every node that matches any of the queries (that is, you'll get the union of the results). In XPath, you'll combine statements with the vertical bar (`|`) character, while in jQuery you'll use a comma (`,`) (see Listing 11).

Listing 11. Finding the results of multiple selectors

```javascript
/* Find all book names and all authors (XPath) */
var result = root.find( "//name|//author" );

/* Find all book names and all authors (jQuery) */
var result = $(root).find( "name,author" );
```
In both cases, the result will be a list of all `<name>` and `<author>` elements from anywhere in the document. In **Figure 1**, see the XPath result using AquaPath (for more about AquaPath, a tool for Mac OS X Tiger, see Related topics).

**Figure 1. XPath result with highlighted name and author tags for all books in the book.xml file**

Traversing nodes
In addition to selecting nodes, you often need to traverse the structure of a document, either to find related data or to perform complex manipulations. XPath and jQuery have you covered when you need to get around in your documents.

Given what you've learned previously, you can use these traversal methods to help find ancestors (that is, elements that contain the current element) or descendants (elements contained by the current element).

For example, **Listing 12** allows you to find the `<catalog>` that contains the last `<book>` you've already found.
**Listing 12. What catalog lists the last book?**

/* Find the catalog for the last book you know about (XPath) */
var result = root.find( "//book[last()]/ancestor::catalog" );

/* Find the catalog for the last book you know about (jQuery) */
var result = $(root).find( "book:last" ).closest( "catalog" );

Figure 2 shows the result.

**Figure 2. The catalog ancestor of the last book**

![Figure 2](image-url)

One thing to note is that the jQuery closest() method works more like XPath’s ancestor-or-self; it will include the current node if it matches. In this case, it won't, but it’s something to keep in mind if you can nest elements with the same name, or if you’re matching on attributes.

If you need to go the other way and find elements that might be deeply nested from the one you have, you can do that too (see Listing 13).
Listing 13. Find the prices listed in the catalog

/* Find the prices of everything in the catalog. (XPath) */
var result = root.find( "//catalog/descendant::price" );

/* Find the prices of everything in the catalog. (jQuery) */
var result = $(root).find( "catalog price" );

Like ancestor in XPath, descendant has a descendant-or-self for those special cases where the selected node might match what you're looking for (see Figure 3).

Figure 3. All the prices, selected

Simulating advanced XPath features

XPath specifies a number of useful features that aren't really necessary in jQuery; after all, jQuery is running in the browser where it can take full advantage of JavaScript, while XPath is often used in more restricted environments, such as XSLT processing.

Of course, that won't stop you from implementing these features in JavaScript if you want to use them.
You can easily count the number of results from your query (see **Listing 14**).

**Listing 14. How many nodes match the selector?**

```javascript
/* How many price entries do you have? (XPath) */
var result = root.find( "count(//price)" );

/* How many price entries do you have? (jQuery) */
var result = $(root).find("price").length;
```

Sometimes you only need to know if a node contains a string or not (see **Listing 15**).

**Listing 15. Does the third <author> have Chris in it?**

```javascript
/* Does the third <author> have "Chris" in its contents? (XPath) */
var result = root.find( "contains(//book[3]/author,'Chris')" );

/* Does the third <author> have "Chris" in its contents? (jQuery) */
var result = $(root).find( "book:eq(2) author:contains('Chris')" ).length > 0
```

A very important difference to note in **Listing 15** is that XPath's indexes start at 1, instead of starting with 0. In jQuery, you have to use `:eq(2)` to get the third result.

XPath also has a `sum()` function, which will take the contents of the matching nodes, convert them to numeric values, and return the sum of those values. You have to simulate this with a short function when using jQuery (see **Listing 16**).

**Listing 16. Summing the contents of some nodes**

```javascript
/* Sum the Canadian prices (XPath) */
var result = root.find( "sum(//price[@curr='CAD'])" );

/* Sum the Canadian prices (jQuery) */
function sum( root, selector ) {
    var x = 0;
    $(root).find( selector ).map( function() {
        if( this.text ) {
            // Internet Explorer-only
            return x += ( this.text * 1 );
        }
        // Firefox and W3C-compliant browsers
        return x += ( this.textContent * 1 );
    });
    return x;
}
var result = sum( root, "price[curr='CAD']" );
```

The `map()` method in jQuery runs the specified function for each of the result nodes. Note that you have to do a little trickery to get at the contents of the result nodes, too. Be sure to test this sort of JavaScript on all of your favorite browsers.

You should now be well on your way to understanding when and how to use XPath 1.0 and jQuery 1.4 for similar tasks.
Summary

XPath and jQuery have powerful querying semantics for selecting nodes from well-formed XML documents, including XHTML pages. Although their syntax is different, using one or the other to select important or interesting nodes based on element names or attribute values from a document is relatively easy.

Both XPath and jQuery support straightforward traversal semantics for matching element nodes in relation to the currently matched element. In addition, because jQuery is running in a full JavaScript interpreter, you can simulate some advanced features from XPath with a little bit of coding.
Related topics

• **Process XML in the browser using jQuery** (Uche Ogbuji, developerWorks, December 2009): Find out how to process XML directly in the browser with jQuery.

• **The Java XPath API: Querying XML from Java programs** (Elliotte Rusty Harold, developerWorks, August 2008): Learn about the Java XPath API.

• **Get started with XPath** (Bertrand Portier, developerWorks, May 2004): Learn what XPath is, the syntax and semantics of the XPath language, how to use XPath location paths, how to use XPath expressions, how to use XPath functions, and how XPath relates to XSLT.

• **Simplify your Ajax development with jQuery** (Jesse Skinner, developerWorks, April 2007): Learn about the jQuery philosophy, discover its features and functions, perform some common Ajax tasks, and find out how to extend jQuery with plug-ins.

• **XML Path Language (XPath): Create expressions relating to portions of an XML document** (developerWorks, February 2007): Find out more about XPath.

• **jQuery Tutorials page**: Cover the fundamentals of the jQuery library and more in-depth topics such as learning how to use jQuery in your XHTML pages in these tutorials.

• **The XML FAQ**: Explore another excellent source of XML information, the XML FAQ edited by Peter Flynn.

• **More articles by this author** (Chris Herborth, developerWorks, March 2006-current): Read articles about XML and other technologies.

• **jQuery version 1.4.0**: Download jQuery and speed up your Web development with a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions.

• **IBM certification**: Find out how you can become an IBM-Certified Developer.

• **XML technical library**: See the developerWorks XML Zone for a wide range of technical articles and tips, tutorials, standards, and IBM Redbooks.

• **IBM product evaluation versions**: Get your hands on application development tools and middleware products.

© Copyright IBM Corporation 2010
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)