Native XML Scripting with E4X

John **Schneider**

Abstract

Developing software to create, navigate and manipulate XML data is a major part of almost every developer's job. However, reading, writing and maintaining XML code can be tedious, time consuming and error prone requiring a heap of reference manuals and a large technology stack. Native XML scripting provides a far more intuitive, familiar and natural way for developers to deal with XML. It extends popular scripting languages with native XML data types, XML literals and a small set of new operators useful for common XML operations, such as searching and filtering. Because it reuses familiar, intuitive programming constructs and operators, it is extremely easy for developers to learn, read and write, requiring little to no additional knowledge.

ECMAScript for XML (E4X) is a new international standard with broad industry support that adds native XML scripting to JavaScript. The Mozilla open source project includes two independent implementations of E4X that make it easy to use E4X and embed it in applications. E4X scripts are smaller than comparable XSL transforms, XML Queries or DOM routines and more intuitive to the average developer. They are easier to read, write and maintain requiring less developer time, skill and specialized knowledge. In addition, E4X is a lighter weight technology enabling a wide range of mobile applications.

Table of Contents

1. Introduction

It's no secret - XML's popularity has skyrocketed, making it a significant part of our daily jobs. Developers are inundated with a wide variety of data encoded in XML, including web pages, web services, deployment descriptors, configuration files, ANT files, XML Schemas and a myriad of XML vocabularies for vertical industries (e.g. purchase orders).

However, reading, writing and maintaining XML code can be tedious, time consuming and error prone requiring a heap of reference manuals and a large technology stack. How much time have you spent in the last year pouring over XML-related specifications; learning new ways to create, navigate, and manipulate XML data; and trying to decipher and debug that XSLT or DOM code you (or that new guy) wrote a few months ago?

Have you ever stopped to wonder whether all these new XML concepts, processing models, and languages are really necessary? After all, you've been creating, navigating, and manipulating similarly complex data structures using objects for years. Isn't there some way to flatten the XML learning curve and simplify XML processing by leveraging your existing programming skills and knowledge?

Native XML scripting provides a far more intuitive, familiar and natural way for developers to deal with XML. It extends popular programming languages with native XML data types, XML literals and a small set of new operators useful for common XML operations, such as searching and filtering. Because it reuses familiar, intuitive programming constructs and operators, it is extremely easy for developers to learn, read and write, requiring little to no additional knowledge.

ECMAScript for XML (E4X) adds native XML support to Javascript, making it the first mainstream programming language with native support for XML. E4X scripts are smaller than comparable XSL transforms, XML Queries or DOM routines and more intuitive for the average developer. They are easier to read, write and maintain requiring less developer time, skill and specialized knowledge. In addition, E4X is a lighter weight technology enabling a wide range of mobile applications.

This paper provides an overview of the challenges E4X addresses, introduces E4X and provides several examples to illustrate the benefits of EFX. In addition, it provides a brief overview of the status of E4X standards and open source initiatives.

2. Motivation

With the introduction of XML came several new programming models designed specifically for creating and manipulating XML data (e.g., XSLT, XQuery, the DOM). These programming models were innovative, but unfamiliar to mainstream software developers. They introduced a steep and continuously growing learning curve, requiring a heap of reference manuals and a lot of specialized knowledge.

Consequently, XML developers had to learn and master a complex array of new concepts and programming techniques for creating and manipulating XML data. These programming models did not integrate naturally into the mainstream programming languages or into the mainstream design, development, testing and debugging environments that software developers use for application development.

To better understand this issue, lets look at a simple XML processing problem and how it may be addressed using a few of common XML processing approaches.

2.1. An Example Problem

Given a simple XML purchase order of the following form:

```
<!-- note: no small, furry critters were harmed in this example -->
<order>
```

```
    <customer>
        <name>I. Wannabuy</name>
        <address>
            <street>53 Party Lane</street>
            <city>Bellevue</city>
            <state>Washington</state>
            <zip>98008</zip>
        </address>
    </customer>
    <item>
        <description>Small Rodent, Generic</description>
        <quantity>35</quantity>
        <price>29.99</price>
    </item>
    <item>
        <description>Catapult</description>
        <quantity>1</quantity>
        <price>149.95</price>
    </item>
    <item>
        <description>1 Liter Vodka</description>
        <quantity>10</quantity>
        <price>34.99</price>
    </item>
    <item>
        <description>300 foot measuring tape</description>
        <quantity>1</quantity>
        <price>9.95</price>
    </item>
</order>
```

we are asked to compute the total price of the order and append a new XML element named <total> that contains the result. Faced with this problem, the average Javascript developer with no XML knowledge might immediately think of something like this:

```
function appendTotal (order) {
    var total = 0;
    for each (item in order.item)
        total += item.price * item.quantity;

    order.total = total;
}
```

This is the code the Javascript programmer would normally write to accomplish this task using a Javascript object and standard Javascript operators. However, since the purchase order is XML, we must use an XML technology to perform this operation. Let's look at a few examples.

2.2. The XSLT Approach

XSLT is a language for transforming XML documents into other XML documents. Like the XML DOM (see Section 2.3 below), it represents XML data using a tree-based abstraction, but also provides an expression language called

XPath designed for navigating trees. On top of this, it adds a declarative, rule-based language for matching portions of the input document and generating the output document accordingly.

Using XSLT, the solution to the example problem above would look something like this:

```
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:variable name="total" select="0"/>
    <xsl:template match="item" priority="1">
        <xsl:set-variable name="total"
             select="$total + ./price * ./quantity"/>
    </xsl:template>
    <xsl:template match="*|/|comment()|processing-instruction()">
        <xsl:value-of "."/>
        <xsl:apply-templates/>
    </xsl:template>
    <xsl:template match="/*[position() = last()]">
        <xsl:value-of "."/>
        <xsl:apply-templates/>
        <total><xsl:value-of select="$total"/></total>
    </xsl:template>
</xsl:stylesheet>
```

Note: This example stylesheet was simplified to save space by introducing a non-standard <xsl:set-variable> element, which does not exist in XSLT. The standard XSLT solution to this problem requires some additional script and DOM code or a stylesheet about three times this size.

From the example stylesheet above, it is clear the XSLT approach for navigating and manipulating data structures is completely different from the Javascript approach for navigating and manipulating Javascript data structures. Consequently, the XSLT learning curve for Javascript programmers is quite steep. In addition to learning a new data model, Javascript programmers must learn a declarative programming model, recursive descent processing model, new expression language, new XML language syntax, and a variety of new programming concepts (templates, patterns, priority rules, etc.). These differences make XSLT code harder to read, write and maintain for the Javascript programmer. In addition, it is not possible to use familiar development environments, debuggers and testing tools with XSLT.

2.3. The DOM Approach

One of the most common approaches to processing XML is to use a software package that implements the interfaces defined by the W3C XML DOM (Document Object Model). The XML DOM represents XML data using a general purpose tree abstraction and provides a tree-based API for navigating and manipulating the data (e.g., getParentNode(), getChildNodes(), removeChild(), etc.).

Using the DOM, the solution to the example problem above would look like this:

```
function appendTotal(document) {
    total = 0;
    items = document.getElementsByTagName("item");
    for (i = 0; i < items.length; i++) {
        item = items.item(i);
        price = item.getElementsByTagName("price").item(0);
        priceValue = price.item(0).getNodeValue();
        quantity = item.getElementsByTagName("quantity").item(0);
        quantityValue = quantity.item(0).getNodeValue();
```

```
        total += priceValue * quantityValue;
    }

    totalText = document.createTextNode(total);
    totalElem = document.createElement("total");
    totalElem.appendChild(totalText);
    document.item(0).appendChild(totalElem);
}
```

This code is actually a little bit more familiar to the Javascript developer because it uses familiar Javascript syntax and familiar Javascript concepts, like properties and methods. However, this approach for navigating and manipulating data structures is still very different from the Javascript approach for accessing and manipulating native Javascript data structures. Javascript programmers must learn to write tree navigation algorithms instead of object navigation algorithms. In addition, they have to learn a relatively complex interface hierarchy for interacting with the XML DOM. The resulting XML DOM code is generally harder to read, write, and maintain than code that manipulates native Javascript data structures. It is more verbose and tends to obscure the developer's intent with lengthy tree navigation logic. Consequently, XML DOM programs require more time, knowledge and resources to develop.

2.4. Object Mapping Approach

Several have also tried to navigate and manipulate XML data by mapping it to and from native Javascript objects. The idea is to map XML data onto a set of Javascript objects, manipulate those objects directly, then map them back to XML. This allows Javascript programmers to reuse their knowledge of Javascript objects to manipulate XML data.

Initially, this seems like a good idea, but unfortunately it does not work for a wide range of XML processing tasks. Native Javascript objects do not preserve the order of the original XML data and order is significant for XML. Not only do XML developers need to preserve the order of XML data, but they also need to control and manipulate the order of XML data. In addition, XML data contains artifacts that are not easily represented by the Javascript object model, such as namespaces, attributes, comments, processing instructions and mixed element content. Therefore, object mapping only works reliably for the simplest of XML processing problems.

3. ECMAScript for XML (E4X)

ECMAScript for XML (E4X) was developed to address these issues. It was designed to provide a simple, familiar, general purpose XML programming model that flattens the XML learning curve by leveraging the existing skills and knowledge of the average Javascript developer.

Instead of introducing a completely new set of programming models, object models, operators and syntax, E4X reuses the existing Javascript object model, operators and syntax and extends them with native support for XML. Consequently, familiar Javascript operators can be used for creating, navigating and manipulating XML, such that anyone who has used Javascript is able to start using XML with little or no additional knowledge.

For example, below is the E4X code required to solve the example problem above:

```
function appendTotal (order) {
    var total = 0;
    for each (item in order.item)
        total += item.price * item.quantity;

    order.total = total;
}
```

This is exactly the same code a Javascript developer would write to perform the same operation on a Javascript object. As such, any Javascript developer can read or write this code with almost no additional knowledge about XML. It is relatively easy for even a novice programmer to understand what this code does with very little analysis and no specialized XML knowledge.

E4X adds native XML data types to the Javascript language, extends the semantics of familiar Javascript operators for manipulating XML objects and adds a small set of new operators for common XML operations, such as searching a filtering. It also adds support for XML literals, namespaces, qualified names and other mechanisms to facilitate XML processing. The remainder of this paper introduces and provides examples of these extensions.

3.1. Native XML Objects

E4X extends the Javascript type system with a native XML object type. Unlike traditional Javascript objects, the properties of XML objects are ordered, meaning developers can both specify and manipulate the order of the properties of an XML object. In addition, XML objects extend the data model of traditional Javascript objects to support attributes, comments, processing instructions, namespaces and qualified names.

There are several easy ways to create XML objects. Here are some examples:

```
// create a new XML object from a string
var order = new XML("<order/>");

// create a new XML object from a file
var order = new XML(readFile("order.xml"));

// create an XML wrapper for manipulating a DOM document object
var doc = XML(document)
```

The last example does not create a completely new object, but rather wraps an existing DOM document object so it can be manipulated using the built-in E4X operators and methods.

3.2. XML Literals

E4X also provides XML literals that make it easy to specify XML values within a Javascript program. For example, the following E4X code creates a new XML <address> element and inserts it into a <customer> element.

```
// set the customer address to a constant value
order.customer.address = <address>
    <street>53 Party Lane</street>
    <city>Big Town</city>
    <state>WA</state>
    <zip>98008</zip>
</address>;
```

Almost any valid XML element can be used as an XML literal in a Javascript program. Portions of XML literals may be dynamically computed using embedded Javascript expressions. For example, consider the following expression:

```
// set the customer data using a set of computed values
order.customer = <customer>
    <name>{custdb.fname + " " + custdb.lname}</name>
    <address>
        <street>{custdb.street}</street>
```

```
        <city>{custdb.city}</city>
        <state>{custdb.state}</state>
        <zip>{getZip(custdb.street, custdb.city, custdb.state)}</zip>
    </address>
</customer>
```

Any valid Javascript expression can be used to dynamically compute any element name, element value, attribute name or attribute value. The example above uses the properties of the custdb object and the getZip function to dynamically compute the value of the <name>, <street>, <city>, <state> and <zip> elements. The custdb object could by any standard Javascript object, including another XML object.

3.3. Reusing Familiar Operators

Once you have an XML object, you can begin navigating and manipulating it using standard Javascript operators. For example, consider the following E4X statements:

```
// get the customer's address from the order
var address = order.customer.address;

// get the second item from the order
var secondItem = order.item[1];

// calculate the total price for the second item in the order
var secondTotal = order.item[1].price * order.item[1].quantity;

// change the quantity of the first item
order.item[0].quantity = 18;

// append a grand total to the order
order.total = grandTotal;
```

The Javascript assignment operator provides a very convenient, familiar way to construct arbitrary XML documents from scratch. For example, the following E4X code:

```
// create an empty customer element, then append sub-elements
var customer = <customer/>;
customer.name = "Fred Jones";
customer.address.street = "123 Fictitious Lane";
customer.address.city = "Bellevue";
customer.address.state = "WA";
```

will create the following XML document:

```
<customer>
    <name>Fred Jones</name>
    <address>
        <street>123 Fictitious Lane</street>
        <city>Farmingham</city>
        <state>CT</state>
    </address>
</customer>
```

3.4. New Operators

E4X also adds several new operators for common XML operations, like accessing attribute values, selecting descendents, filtering XML documents and operating on element children. For example, consider the following E4X statements:

```
// attribute identifier: access XML attributes as specially named properties
var custid = order.customer.@custid;
order.item[1].@id = "123";

// descendent operator: search for descendents without specifying full path
var prices = order..price;
var paragraphs = document..p;

// filtering predicate: get descriptions of items that cost less than $50
var cheapItems = order.item.(price < 50).description;

// child list identifier: get all the child elements of order
var orderData = order.*;

// attribute list: get all XML attributes associated with the customer
var custAttributes = order.customer.@*;
```

One of the guiding design principles of E4X was to keep the number of language extensions to a minimum to avoid unnecessary complexity. E.g., it was a non-goal of E4X to provide the full functionality of XPath using new operators. We introduced new operators for the operations that were considered most common. Less common operations are available via a set of methods built-in to every XML object. The ECMAScript committee is considering making many of these new operators available for all JavaScript objects in a future version of the language.

3.5. Namespaces and Qualified Names

E4X introduces new objects and operators for using XML namespaces and qualified names. Below is a simple example demonstrating the use of E4X to manipulate a SOAP object.

```
// Create a SOAP message
var message = <soap:Envelope
    xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
    soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <soap:Body>
        <m:GetLastTradePrice xmlns:m="http://mycompany.com/stocks">
            <symbol>DIS</symbol>
        </m:GetLastTradePrice>
    </soap:Body>
</soap:Envelope>

// declare two new XML namespace objects
var soap = new Namespace("http://schemas.xmlsoap.org/soap/envelope/");
var stock = new Namespace("http://mycompany.com/stocks");

// use a qualified name to extract the soap message body
var body = message.soap::Body;
```

```
// use a qualified name to change the stock symbol
message.soap::Body.stock::GetTradePrice.symbol = "MYCO";
```

E4X also supports the ability to suggest prefixes for specific namespace URIs and the concept of default namespaces. Namespaces and qualified names can be constructed statically based on values known at compile time (like the example above) or dynamically based on computed values.

4. Standards and Open Source

E4X was approved as an international standard on 29 June 2004 by the ECMA General Assembly, the body that maintains the international Javascript standard. The specification was supported by AgileDelta, BEA, IBM, Macromedia, Microsoft, Mozilla, Netscape, Palm, RIM and others. The ECMAScript group (TC39/TG1) is now working on the second version of E4X, which includes native support for XML Schemas and will be an integral part of future versions of Javascript.

The Mozilla open source project contains two independent implementations of E4X, called SpiderMonkey and Rhino. The next official release of the Firefox browser will include native E4X support.

Javascript is the first mainstream language to include native XML support, but the idea has already spread to several other scripting languages, such as PHP, and is being considered for widely used programming languages like Java.

5. Conclusion

This paper provides an overview of some of the challenges inherent in current XML programming models, introduces E4X's approach to addressing these challenges and provides several E4X examples to demonstrate the expressive power and intuitive nature of E4X.

E4X provides a more intuitive, familiar and natural way for developers to deal with XML. E4X scripts and applications are smaller and more intuitive for software developers than comparable XSLT or DOM applications. They are easier to read, write and maintain requiring less developer time, skill and specialized knowledge. E4X also fits more naturally into familiar design, development and debugging environments. The net result is reduced code complexity, tighter revision cycles and shorter time to market for XML applications.

This paper provides only a glimpse of the power and functionality of E4X. To gain a full appreciation for E4X we recommend you download one of the freely available E4X implementations and try it for yourself!

6. Resources

E4X specification: http://www.ecma-international.org/publications/standards/Ecma-357.htm

Mozilla Rhino: http://www.mozilla.org/rhino/

Mozilla SpiderMonkey: http://www.mozilla.org/js/spidermonkey/

Mozilla Firefox: http://www.mozilla.org/projects/firefox/

Biography

John **Schneider**
AgileDelta, Inc. [http://www.agiledelta.com/]
15400 SE 30th Place, Suite 201
Bellevue
Washington
98007
United States of America

John Schneider is founder and Chief Technology Officer at AgileDelta, Inc., an independent software vendor that supplies XML and Web Services infrastructure software for use on mobile computing platforms.

Prior to founding AgileDelta, Mr. Schneider led Engineering and Program Management for Crossgain Corporation, a high profile Microsoft spin-off founded to radically simplify building and deploying web services and applications (acquired by BEA). He was also Principal Systems Engineer at MITRE where he kick-started and led a variety of international XML initiatives for the United States and NATO.

Mr. Schneider founded and led ECMA's ECMAScript for XML (E4X) group, which developed the first mainstream programming language with native support for XML. He helped develop the XML Query Language and the XML Schema Language. He was a U.S. delegate to NATO on technology initiatives related to XML, messaging, data interoperability and integrated business processes.

Mr. Schneider has twenty years experience developing technology with a focus on mobile technologies, data encoding, information management, data interoperability and messaging. He has a Masters degree and Bachelors degree in Computer Science.