XML Parsing in JavaScript

Alberto Simões

Centro Algoritmi, University of Minho, Braga, Portugal; and
Instituto Politécnico do Cávado e do Ave, Barcelos, Portugal
asimoes@ipca.pt

Abstract

With Web 2.0 the dynamic web got to a reality. With it, some new concepts arived, like the use of asynchronous calls to receive missing data to render a website, instead of requesting a full new page to the server. For this task, and in the recent years, developers use mostly the JSON format for the interchange of data, than XML. Nevertheless, XML is more suitable for some kind of data interchange but, and even if the web is based in SGML/XML standards, processing XML using directly JavaScript is tricky.

In this document, a set of different approaches to parse XML with JavaScript will be described, and a new module, based on a set of translation functions, will be presented. At the end, a set of experiments will be discussed, trying to evaluate how versatile the proposed approach is.

1998 ACM Subject Classification I.7.2 Document Preparation / Markup languages, D.3.4 Processors / Parsing

Keywords and phrases XML, JSON, Parsing, JavaScript

Digital Object Identifier 10.4230/OASIcs.SLATE.2017.9

1 Introduction

The Internet is built in a set of standards. First, the SGML (Standard Generalized Markup Language) from which HTML (HyperText Markup Language) was derived. Despite the fact that a lot of problems arrived from the use of such a permissive standard, the XHTML version of HTML, built on top of XML (eXtensible Markup Language) did not stick. Example of it is the HTML5 standard that, although suggesting XML well formed documents, is still based in SGML.

On account of that, browsers needed to implement two different parsers, one for each of the standards[1](SGML and XML). Although not discussed in this document, the Cascading Style Sheets (CSS) standards started to appear. Since JavaScript entered the web world enabling dynamic web sites, the Internet is no longer the same.

To support JavaScript, browsers needed to add a new parser, for the language, together with its interpreter. As for other programming languages, JavaScript support constructs to define data structures, and soon JSON, the JavaScript Object Notation was defined to allow the serialization of data.

The simplicity of JSON compared to the verbosity of XML lead to the use of JSON for most of the Web 2.0 asynchronous calls. And this happened because most tools developed for the web deal with very structured data that is easily serializable in JSON. But XML and JSON has very different capacities, and when in the need of mixed content, XML stands out [6].

[1] In fact, each one of these parsers has a lot of tweaks in order to cope with some behaviors that were the default during the first browsers implementations, and for which compatibility was desired [12].

6th Symposium on Languages, Applications and Technologies (SLATE 2017).
Editors: R. Queirós, M. Pinto, A. Simões, J. P. Leal, and M. J. Varanda; Article No. 9; pp. 9:1–9:10
Open Access Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

But, even if XML parsers were added a long time ago on browsers, the truth is that it is still quite hard to manipulate data obtained from an XML file. There are a couple of possibilities:

- taking the advantage of including XML directly in a web page, and just format it using Cascading StyleSheets [3], with the disadvantage that the structure manipulation is limited to the addition or generated content, or the hiding of non relevant data;
- another option is to use XSLT (Extensible Stylesheet Language Transformations) [5, 7, 8], but although browser support for its first version is available on all major browsers, newer versions not available natively on any browser. It also has the major drawback of requiring programmers to learn how to use it. If the syntax itself is basic, given it is a XML based language, the way transformations are declared (in a declarative way, not imperative as most programmers are used to use) can be quite challenging.
- using a JavaScript library, either the built-in DOMParser [10] or any other available, like jQuery. The usage of DOMParser would be the better approach, given its availability on all browsers, and even as a module for node.js. Nevertheless, the Document Object Model (DOM) structure is not trivial, and the methods available directly in JavaScript to manipulate it are not versatile.

This document will only focus on the usage of JavaScript to manipulate the XML file, given the limitations of CSS, and the lack of support of XSLT.

In the next section I will start by explaining what mixed content is, and why and when it matters. Section 3 will discuss the two main approaches used for processing XML in any language. Section 4 will restrict it to JavaScript, and present some of the available tools to manipulate the XML DOM available in Browsers. Section 5 describes the approach implemented and why it is useful. Before concluding, section 6 will show some uses of the tool.

2 XML, JSON and Mixed Content

As stated in the introduction, XML and JSON, both, can be used for serialization and, in that arena, JSON is more compact and, with its binary counterparts, like BSON, can be quite fast. Nevertheless, when the goal is to encode mixed content, XML is more adequate, as discussed in depth in [6].

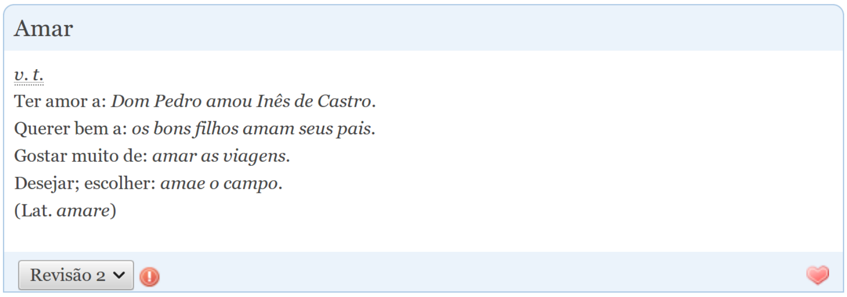
In order to understand the problem, we will consider an example extracted from the project that motivated this problem [13], an on-line dictionary with a RESTful API: Dicionário-Aberto.[2]

One of the methods available in the API is the retrieval of entries from the dictionary (either randomly or by a specific word). Figure 1 shows a random entry from the dictionary as presented to the common user in the web interface.

As can be seen, there are multiple lines, with different definitions. Some of them can be of pure text, but other include quotes and examples, that need to be differentiated. Although this could be codified in JSON, it is not so natural.

The current API for Dicionário-Aberto can return records in both formats. Listing 1 shows how the entry in Figure 1 is encoded in JSON, while Listing 2 shows that same entry encoded in XML.

[2] http://dicionario-aberto.net



Amar

v. t.

Ter amor a: *Dom Pedro amou Inês de Castro*.

Querer bem a: *os bons filhos amam seus pais*.

Gostar muito de: *amar as viagens*.

Desejar; escolher: *amae o campo*.

(Lat. *amare*)

Figure 1 Entry for the word 'amar' (*to love*).

Listing 1 Entry for 'amar' encoded in JSON.

```
{ "entry" : {
    "@id" : "amar",
    "form" : { "orth" : "Amar" },
    "sense" : [ {
      "gramGrp" : "v. t.",
      "def" : [
        ["Ter amor a: ", { "quote": "Dom Pedro amou Inês de Castro" }],
        ["Querer bem a: ", { "example": "os bons filhos amam seus pais" }],
        ["Gostar muito de: ", { "example": "amar as viagens" }],
        ["Desejar; escolher: ", { "example": "amai o campo" }]
      ]},
    "etym" : { "@orig" : "Lat", "#text" : "amare" }
} }
```

Listing 2 Entry for 'amar' encoded in XML.

```
<entry id="amar">
 <form><orth>Amar</orth></form>
 <sense>
  <gramGrp>v. t.</gramGrp>
  <def>Ter amor a: <quote>Dom Pedro amou Inês de Castro</quote>.</def>
  <def>Querer bem a: <example>os bons filhos amam seus pais</example>.</def>
  <def>Gostar muito de: <example>amar as viagens</example>.</def>
  <def>Desejar; escolher: <example>amai o campo</example>.</def>
 </sense>
 <etym orig="Lat">(Lat. <mentioned>amare</mentioned>)</etym>
</entry>
```

While the presented JSON is not too complicated as in some other entries, this example shows how to encode a simple entry a set of recursive lists and dictionaries are needed.

The best thing about the JSON approach, is how JavaScript makes it easy to manipulate. For the XML, there is a big advantage. New browsers allow the inclusion of this snippet directly inside an HTML page, able to be formatted through CSS. The problem is when there are minor details that need to be manipulated somehow.

3 XML Processors

There are two main approaches when parsing XML documents:

- SAX (Simple API for XML), that works mostly by defining callbacks to every element tag found in the document, or any entity [4]. Instead of processing the document in a structural fashion, SAX parsers will transform the document progressively, as it is parsed. While this approach is quite simple to implement, it is not versatile for complex XML

  transformations. For that, a couple of external data structures would be needed, mostly as if the user create her own document tree structure.
- DOM (Document Object Model) parsing, in the other hand, works by creating an abstract syntax tree for the whole document, as created by mostly compilers of conventional programming languages [9]. The main disadvantage of this approach is the amount of memory needed when processing large documents, as it needs to be all loaded into memory. But, in the other hand, it is quite easy to traverse the tree and do changes, rearranging the branches, pruning them, or adding new ones.

  For the pointed disadvantage, there are two main approaches that have been used for large documents:
  - XML Pull Parsing allows the construction of DOM trees for specific parts of a document, that are retrieved as needed.
  - As most of the large XML documents have a repetitive structure, just as logs or collections of resources, there are parser implementations that chunk the large document on the repetitive element, and parses their contents using a standard DOM Parser.

  Nevertheless, usually the size of XML documents sent through the web during AJAX calls are small, and this is not a relevant problem.

Most programming languages have libraries or modules that use some of these approaches. The well known Expat[3] parser and LibXML[4] support both approaches, and have binding for most of the common programming languages. Unfortunately that is not true for JavaScript.

4 XML and JavaScript

Despite the fact that XML is parsed by browser for a long time, the amount of tools to process XML with JavaScript is quite limited. This might be a result of the arrival of JSON and the small number of users actually needing real mixed content.

Browsing the Internet for JavaScript libraries to manipulate XML there are two obvious answers:

- use the built-in DOMParser [10], and its DOM structure, navigating through each element top-down (from the root node to the leafs), looking for the relevant data;
- use jQuery [2, 11] and its selectors[5] (based on CSS selectors).

Both approaches are easy to use, but not very versatile. To explain this, consider the example shown in Listing 2, and a pair of simple tasks:

- **Task I.** Find the orthographic form of the entry (`orth` tag):

  DOM using the DOM tree is not too hard, specially when looking for a specific leaf of the DOM tree. Considering the variable `entry` to contain the XML fragment above, the following code would retrieve the orthographic form:

  ```
  var parser = new DOMParser();
  var doc = parser.parseFromString(entry , "text/xml");
  var term = doc.activeElement.children[0]
                               .children[0].childNodes[0].data);
  ```

[3] `https://libexpat.github.io/`

[4] `http://xmlsoft.org/`

[5] Note that jQuery can be used to manipulate the DOM as well, but it just adds a couple of extra methods to make the tree traversal easier. Also, although jQuery uses CSS selectors, there is the possibility to add support for XPath as well.

Accordingly with the standard, the method `getElementsByName` should be available in an XMLDocument instance (`doc` in the code above). Nevertheless, it does not work correctly on all browsers. A recent Firefox would complain about a non existing function (even if the Mozilla Developers Network documents that an XMLDocument instance inherits methods from Document).

jQuery this is the simplest task for jQuery: as there is only one tag with that name, a simple selector can be used. Considering that the variable `entry` is a string containing the XML fragment, the following code would suffice:

```
var term = $(entry).find('orth')[0];
```

- **Task II.** Remove the `example` elements, and remove the colon before them.

DOM giving the quite unstable API to manipulate an XMLDocument directly in the browser, no solution will be presented using directly the DOM. From the example before it could be seen that a traversal approach would take too long to write, and would be error prone (how many children levels?), and the lack of support for XMLDocument methods would make the resulting code work (or completely not work) accordingly with the used browser.

jQuery using jQuery for this task is a little more tiresome. The first task is to remove the `example` elements, while the second is to cycle all `def` elements to remove the colon. While the whole jQuery syntax is based in the functional paradigm, for this task it is needed to remind that JavaScript is an object oriented programming language, and therefore, changes need to be done directly on the XML object. Also, notice that at the end the user gets a jQuery XML document, and not a string with the parsed XML[6].

```
$xml = $(entry);
$xml.find('example').remove();
$xml.find('def').map(function(i,val){
    val.innerText = val.innerText.replace(/:\s*\.\s*$/,".");
});
```

5 Traversing the DOM Tree

The implemented approach is based in a Perl module, named XML::DT [1], that uses a bottom-up approach to process the DOM tree. Following its brother name, the JavaScript implementation is named XML-DT-JS.

It works like a dispatch table where, for each element, a function is defined. The traversal algorithm will start with the leafs, and feed the function with the element name, its contents (the CDATA) and the associated properties. The function can do whatever is needed to this data, and must return a string (that can contain XML).

The non-leaf nodes' functions receive the element name, and the associated properties, as the leaf nodes' functions, but the content itself, is supplied as returned by the child nodes processors. In the case the element has more than one child, then their results are concatenated into a single string.

Listing 3 shows the code to convert from the following input XML document to the respective output string:

[6] This fact can be seen as an advantage or disadvantage, depending on the user goals.

Input:

`<root><foo>zbr</foo><bar>Something</bar></root>`

Output:

`<root><zbr>foo</zbr>Hello</root>`

Listing 3 Simple XML-DT-JS code example.

```
xml$dt.process(input,
{ root: function(q,c,v) { return xml$dt.tag(q,c,v); }
  foo:  function(q,c,v) { return xml$dt.tag(c,v); },
  bar:  function(q,c,v) { return "Hello"; } });
```

The `xml$dt.process` function is the main method to call for the structural processing. First argument is the XML string to process. Second argument is a mapping from tag names to functions. Each function receive the tag name (`q` variable[7], the tag contents (`c` variable) and a map of attribute names to their values (`v` variable). The functions should return the processed node as a string.

The utility `xml$dt.tag` function allows the quick creation of a XML string, given the tag name, tag contents, and attributes (in the same order as they are received by the `process` function.

There are three special element names that can be defined:

- **#document** allows to define a function associated to the root node, without the need to know what is its name. It also allows to define a function to deal with the final tree, after the root node processing. By default it is the identity function.
- **#text** allows to define a function to process all the text nodes, before them being processed by the respective enclosing element. By default it is the identity function.

  Listing 4 shows the code to convert from the following input XML document to the respective output string:

  Input:

  `<list><item qt="2">banana</item><item qt="5">pineapple</item></list>`

  Output:

  `<list>two bananas</item><item>five pineapples</item></list>`

  Listing 4 XML-DT-JS code using `#text` rule.

  ```
  var nrs = ['zero','one','two','three','four','five'];
  xml$dt.process(input, {
   '#text': function(q,c)   { return c + "s"; },
   item:    function(q,c,v) { return xml$dt.tag(q, nrs[v.qt] +" "+ c); },
   list:    function(q,c,v) { return xml$dt.tag(q,c,v); } });
  ```

- **#default** defines a function to process any element whose processing function is not defined. Therefore, in cases where the processing algorithm can be derived from the element name or its attributes, a simple default processing function can be enough.

  Listing 5 shows the code to prefix every tag with a namespace, as in the following example:

  Input:

  `<list><item qt="2">banana</item><item qt="5">pineapple</item></list>`

[7] These variable names can be changed, but are kept in our examples to keep the same variable names used by the Perl version that were, also, kept from Omnimark.

Output:

```
<ex:list><ex:item qt="2">banana</ex:item>
         <ex:item qt="5">pineapple</ex:item></ex:list>
```

Listing 5 XML-DT-JS code using `#default` rule.

```
xml$dt.process(input, {
 '#default' : function(q,c,v) { return xml$dt.tag("ex:"+q, c, v); });
```

On top of this basic traversal algorithm, a few features were added, to allow more control of the processing functions, and to allow easier definition of markup converters:

- For markup conversion, where the goal is just to change the element name from one to another, a shortcut mapping can be defined.
  Listing 6 shows the code to convert some tags directly to HTML tags:

  Input:

  ```
  <list><item>bananas</item><item>pineapples</item></list>
  ```

  Output:

  ```
  <ul><li>bananas</li><li>pineapples</li></ul>
  ```

  Listing 6 XML-DT-JS code using `#map` shortcut.

  ```
  xml$dt.process(input, { '#map' : { list: 'ul', item: 'li' }});
  ```

- In some situations, it is relevant to store some data from one node on its father, rather than just returning it as a string. For example, it can be handy when returning two distinct values, or when it is easier for the elements fathers to process a list rather than a concatenated string, as shown in the next example:
  Listing 7 shows the code to convert some tags directly to HTML tags:

  Input:

  ```
  <list><item qt="4">bananas</item><item qt="2">pineapples</item></list>
  ```

  Output:

  ```
  <list total="6’’><item qt="4">bananas</item>
                  <item qt="2">pineapples</item></list>
  ```

  Listing 7 XML-DT-JS code using `father` variable.

  ```
  xml$dt.process(input, {
    item: function(q,c,v){
      if ('total' in xml$dt.father) xml$dt.father.total += v.qt;
      else                          xml$dt.father.total  = v.qt;
      return xml$dt.tag(q,c,v);
    },
    list: function(q,c,v) { return xml$dt.tag(q,c,v); }});
  ```

  Notice that the `father` shortcut accesses the attributes of the father element. Therefore, when calling the processing function for that element, the attributes defined using that shotcut will be available in the `v` variable.

This section concludes with the implementation of the two tasks described in section 4:

- Obtaining the orthographic form for an entry:

  ```
  var term;
  xml$dt.process(entry, { orth: function(q,c,v) { term = c; } });
  ```

This is not a clean solution, as it is not a functional approach, doing the job using lateral effects. Nevertheless, it is not that easy to implement this same behavior using the functional paradigm.

- Removing examples from the definitions:

```
var result = xml$dt.process(entry,
{ example: function() { return ""; },
  def: function(q,c,v) {
     return xml$dt.tag(q, c.replace(/:\s*\.\s*$/, "."), v); }
});
```

The main advantage from this solution when compared with the jQuery solution presented before, is that it does not require the user to know how to use the `map` function, or to deal with the rather obscure `innerText` property.

6 Using XML-DT-JS

This section presents some real examples where XML-DT-JS is being used, in the context of Dicionário-Aberto. Fortunately, modern browsers allow the embedding of XML snippets inside of HTML documents, and their formatting with CSS rules. Therefore, everything that can be accomplished just by the definition of CSS rules has priority over the processing of the entries. Simple tasks, like changing the font weight or the block-formatting of tags are done directly in CSS.

- Extracting the orthographic form from the entry identifier and, if present, the sense number, formatting it properly in HTML (see Listing 8.

Listing 8 Extract orthographic form and sense number from an entry.

```
function getEntryTerm(data) {
    return xml$dt.process(data, {
      entry: function(q,c,v) {
        var word = v.id;
        if (word.match(/:\d+$/) {
          word = word.replace(/:(\d+)/, "<sup>$1</sup>");
        }
        return word;
      }
    });
}
```

This task is quite similar to the extraction of the orthographic form presented before. In this case, only the `id` attribute from the `entry` tag is processed and extracted. Given this is the root node, it suffices to return it, obtaining a better functional approach.

- Other task currently being solved with XML-DT-JS is the formatting of an entry. Given some entries include definitions with old wiki-like markup (underscores instead of italic), some rules treat the element textual contents. In the other hand, some entries include common new-lines to mark different senses, and therefore, they need to be correctly formatted as line breaks. Finally, the `form` tag needs to be renamed, given that HTML already uses it (see Listing 9).

There are some other places where XML-DT-JS is handy, but those situations does not add any more to this document, and therefore, will not be presented.

Listing 9 Formatting a dictionary entry, with some pre-processing.

```
function formatEntry(data) {
    return = xml$dt.process(data, {
        '#map' : { 'form': 'div' },
        '#default' : function(q,c,v) { return xml$dt.tag(q,c,v); },
        'def' : function(q,c,v) {
          var s = c.replace(/(^\n(\s*\n)*|\n(\s*\n)*$)/g,"")
                   .replace(/_([^_]+)_/g, "<i>$1</i>")
                   .replace(/\n(\s*\n)*/g,"<br/>");
          return xml$dt.tag(q,s,v);
        },
        'etym' : function(q,c,v) {
            return c.replace(/_([^_]+)_/g, "<i>$1</i>");
        },
      });
}
```

7 Conclusions

In this document a small library to process XML documents using JavaScript in the browser was presented[8]. The library uses a bottom-up approach to process the structure of an XML document. Given that the traversal algorithm is predefined, the user just needs to implement the rules of how each XML tag will be processed.

The tool was developed in the context of a project where the mixed content support of XML is relevant. While current usage in the context of the project is quite limited, the experience of using this kind of processors with the Perl programming language shows that this approach is very versatile.

In the future, it is intended to add to this tool the following functionality:

- support types: allow the definition of structural types for some tags (for example, specifying that a list is a collection of items) instead of always using the concatenation of strings;
- allow its usage in the server side, with node.js, where DOMParser is not available by default;
- support direct access to the root element of the tree (using a similar approach as the *father* attribute presented before);
- better support for entities and entities escaping;
- an analysis on the impact of the traversal time, versus the usage of CSS or XPath expressions.

References

1 José João Almeida and José Carlos Ramalho. XML::DT a Perl Down-Translation module. In *XML-Europe'99*, Granada, Spain, May 1999.

2 Bear Bibeault and Yehuda Katz. *jQuery in Action, Second Edition*. Manning Publications, 2 edition, 2010.

3 Bert Bos. Descriptions of all CSS specifications. Technical report, World Wide Web Consortium (w3c), 2017. URL: `https://www.w3.org/Style/CSS/specs.en.html`.

4 David Brownell. *Processing XML Efficiently with Java*. O'Reilly Media, 2002.

[8] The current version of the library is available at the following GIT repository: `https://gitlab.com/ambs/xml-dt-js`

5 James Clark. XSL Transformations (XSLT) – version 1.0. Technical report, World Wide Web Consortium (w3c), 1999. URL: `https://www.w3.org/TR/xslt`.

6 Rúben Fonseca and Alberto Simões. Alternativas ao XML: YAML e JSON. In José Carlos Ramalho, João Correia Lopes, and Luís Carríço, editors, *XATA 2007 – 5th Conferência Nacional em XML, Aplicações e Tecnologias Associadas*, pages 33–46, February 2007.

7 Michael Kay. XSL Transformations (XSLT) – version 2.0. Technical report, World Wide Web Consortium (w3c), 2007. URL: `https://www.w3.org/TR/xslt20/`.

8 Michael Kay. XSL Transformations (XSLT) – version 3.0. Technical report, World Wide Web Consortium (w3c), 2017. URL: `https://www.w3.org/TR/xslt30/`.

9 Peter-Paul Koch. The document object model: an introduction. *Digital Web Magazine*, May 2001.

10 Travis Leithead. DOM parsing and serialization. Technical report, World Wide Web Consortium (w3c), 2016. URL: `https://www.w3.org/TR/DOM-Parsing/`.

11 Code Lindley. *jQuery Succinctly*. Syncfusion, 2012.

12 Mark Pilgrim. *HTML5: Up and Running*. O'Reilly Media, Inc., 1st edition, 2010.

13 Alberto Simões, Álvaro Iriarte, and José João Almeida. Dicionário-aberto – a source of resources for the portuguese language processing. *Computational Processing of the Portuguese Language, Lecture Notes for Artificial Intelligence*, 7243:121–127, April 2012.