How to configure an editor

An overview of how we built Fonto

Martin Middel
FontoXML
`<martin.middel@fontoxml.com>`

Abstract

In 2012 a web agency took on the challenge of building an XML editor. This paper gives an overview of a number of concepts that proved to be useful, and some concepts that did not.

1. Introduction

FontoXML is an editor for XML document or structured content in more general terms. It's primary intended to be used by Subject Matter Expert who do not necessarily have any knowledge of XML. FontoXML editor is much a platform then a shrink-wrapped piece of software. It can be tailored to support any kind of XML document format, including DITA 1.3, JATS , TEI, Office Open XML and other proprietary format. The platform itself is highly confirurable so it can be tailored to specific use cases.

All of these different editors have three different parts of configuration:

- the schema defines which elements may occur where;
- elements are assigned to 'families' causing them to receive a visualization and basic cursor behaviour;
- and operations, which define the effect of the toolbar buttons and hotkeys.

Especially the operation part is where FontoXML has seen some major API redesigns. This paper will discuss a number of key decisions we've made and where the XML manipulation layer of FontoXML will move to. We hope that this paper will give an insight in how a small team of JavaScript developers with medium knowledge of XML technologies made the platform on which a large number of XML editors have been and are being built.

2. Iteration 0

FontoXML started in late 2012 when a publisher of commentary on legislative content moved to use an XML schema to encode their content in. Their authors were happily working in MS Word and threatened the company with leaving for their competitor if the new tooling would hinder them in any way. The publishing company required a solution where the user experience is the number one priority and where anything technical would just be taken care of. At that moment, there were no existing XML editors which met their requirements. This is the birth of the precursor of FontoXML, which started life as a specific, bespoke solution.

Writing a user interface containing a toolbar, designing a way to render XML as an HTML view, updating the view as efficiently as possible and many other parts of this version of FontoXML were fun and challenging to do, but this paper will focus on the XML manipulation side of the editor: what happens when I press a button?

2.1. The beginning: canInsert

The first iteration of the XML manipulation was as simple as it gets. There were **commands** which could be queried for their state: whether they were enabled or disabled or whether they were 'active'. Getting state was implemented separately from actually executing the command. Schema validation was hard-coded and exposed functions that could check whether adding an element could exist at a given location. The command then had to implement an execute method to perform the intended mutation if it is allowed. At this point, every command had to query the system by itself. There were no atomic transactions: when a command inevitably had to try multiple things, the command had to undo all of them when it found out that it could not run to completion.

2.2. Schemata as regular expressions

One of the major hurdles of writing an XML editor in JavaScript is the absence of most XML tools for the browser environment. When we started to work on the editor in 2012, there was no usable XPath engine, no usable DOM implementation (besides the browser implementations, which all had a number of inconsistencies, browser-specific issues and performance issues). An XML-schema implementation was also not available. However, XML Schema is a **regular tree grammar**???, meaning the content models of an element are regular languages. We used actual JavaScript regular expressions as a basic schema validator:

```
// The note element may contain p, ol or ul elements, repeated
indefinitely
const noteSchema = /((<p>)|(<ol>)|(<ul>))*/;
// To allow us to work with regular expressions,
// we need to 'stringify' the contents of an element.
function createElementContentString (element) {
  return element.children.map(child => '<' + child.nodeName +
'>').join('');
}
```

```
const noteElement = document.createElement('note');
noteElement.appendChild(document.createElement('p'));
const isValid = noteSchema.test(createElementContentString(nodeElement));
// isValid is true

noteElement.appendChild(document.createElement('note'));
const isValid = noteSchema.test(createElementContentString(nodeElement));
// isValid is false, a note may not contain a note
```

This string-based validation allows us to compose a *'canInsert'* function quite elegantly:

```
function canInsert (element, offset, nodeName, schemaRegex) {
  const elementContents = element.children
    .map(child => '<' + child.nodeName + '>');
  // Use the Array#splice method to add a new element at the given offset
  elementContents.splice(offset, 0, '<' + nodeName + '>');
  return schemaRegex.test(elementContents.join(''));
}
```

We used the browser's built-in RegEx engine to evaluate these expressions.

Even though this approach is a very pragmatic and easy way to implement a schema validator, it is not the best way. The regular expressions were hand-written and hard to maintain. Besides the maintainability issue, string regular expressions are not very extensible. 'Repairing' a split element by generating missing elements turned out to be one of the most code-intensive parts of the editor and the regular-expression approach to validation could not give us any information that we could use as input for this problem.

Perhaps surprisingly, performance was not a bottleneck when we used regexes for schema validation. One would expect the string allocations would cause performance problems when validating elements with a large number of children. We did not run into these issues because the documents we loaded were relatively small. Nodes with relatively many children still only had dozens of child elements, not hundreds.

3. Iteration 1

3.1. Validation

As the first iteration of the XML schema validator, we opted to still use a regular language-based implementation, but to implement our own engine for running them[7]. By compiling the schema to a non-deterministic finite automaton (NFA), we can check whether our element is valid[2].

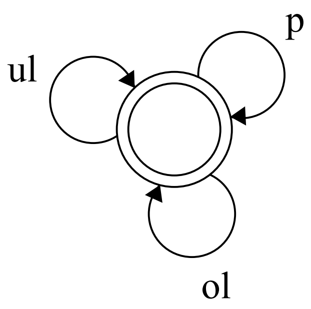

Figure 1. Example validation state machine

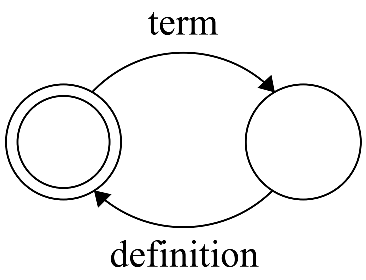

Figure 2. Example of a state machine for a repetition schema

3.2. Synthesis

The schema for which we initially developed the editor was relatively simple and was specialized for different departments of our client. A figure could contain a title followed by an image and a list had to contain list items. During the development of the editor, the schema grew into different sub-schemas. Some of them forced or disallowed titles in figures, while some added new elements which better expressed their specific content. This required us to make the commands we had better interpret the schema, to see which choices could be made where.

We needed to not only know whether an element is valid or not, but we also needed to know WHY it is deemed invalid and whether we can 'fix' it by creating required elements at the correct positions. A major advantage of the new NFA-based schema validator is extensibility. By tracking the path(s) taken though the NFA that lead to its acceptance of a given input sequence, we can see which decisions led to the acceptance or rejection. This can, in turn, tell us which elements are missing under a node.

This process essentially adds an additional state transition type: *record*, which does not 'eat' an input but leaves data on the trace:

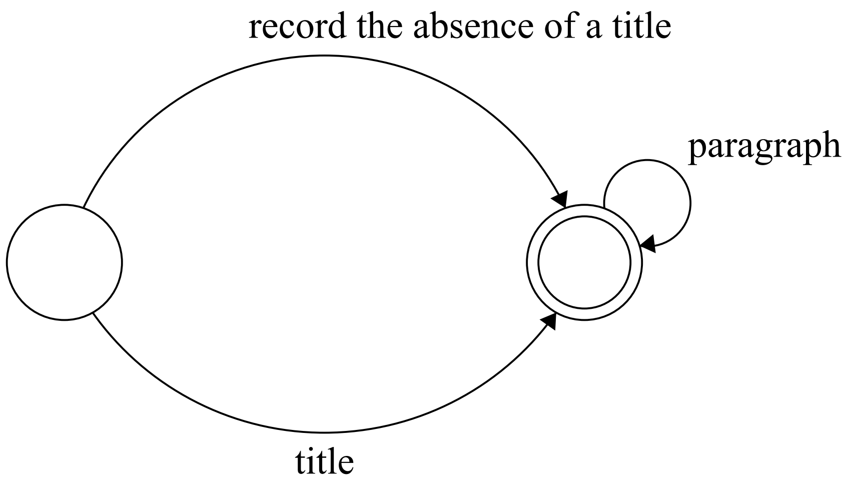

Figure 3. Diagram of the state machine describing the content model (title paragraph*)

When we validate an element containing a single `<paragraph>` with the content model `(title paragraph*)`, it will not be valid. If we however use synthesis, the *record* branch will be taken. This will leave us with the information that we are missing a `<title>` element, and the exact position at which to insert it in order to make the document valid again.

We call this process synthesis and even though randomly creating elements may result in an unexpected situation, this happens relatively rarely in practice, so we still depend on it.

3.3. Flow behaviour

The second hurdle when writing more abstract XML mutations is defining an abstract flow behaviour. Authors expect that having a cursor inside an image should be prevented and that pressing the right arrow with the cursor before a footnote shoulder should not cause the cursor to appear inside the footnote. However, the way images and footnotes are represented as XML varies from schema to schema.

We call the set of properties of elements that define such global navigation and editing behaviour flow properties. Among other things, these directly determine the effect of pressing enter or backspace, as well as the cursor behaviour in and around these elements. The most important flow properties of an element are:

- **Splittability**: Whether the element should be cut in two then we press enter in it, or when we insert a new element in it which can not be contained. A paragraph is usually splittable, while a link or a semantic element like a TEI `<persName/>` is not. This property also defines whether we can merge an element with similar ones, to keep the merging and splitting behaviour consistent.

- **Closed**: Whether we can move in and out of this element using the cursor keys. Footnotes are closed and do not influence the cursor behaviour around them.
- **Detached**: Similar to **closed**, but with additional concern that it is fully detached from its parent and should act like it isn't there. Elements containing metadata are detached.
- **Removable if empty**: Whether the element should be deleted if it is empty and backspace is pressed in front of it. `<note>` elements in DITA can be *removable-if-empty*: if one presses backspace when the cursor is in an empty note, it should be removed.
- **Auto-mergeable / auto-removable if empty**: Empty HTML `<b>` elements are useless, and two adjacent `<b>` elements are equivalent to a single element spanning their combined content. FontoXML normalizes them when it finds them being empty or adjacent.
- **Default text container**: If any, what element should be created when one starts to type in it? For example, typing inside an empty <note> may need to first create a <paragraph> to contain the new text

3.4. Blueprints

Asking for validity or trying to come up with the definitive structure before changing anything in the dom did not allow for very manageable code, so we came up with a new component: blueprints.

A blueprint is basically an overlay over the DOM, it intercepts all accesses to the DOM (like retrieving the parent of a node, or which attributes it has). It also intercepts the mutations to the underlying DOM, so that they don't affect the underlying "real" DOM, but the updated relations in the blueprint can still be queried.

```
const blueprint = new Blueprint();
const newElement = document.createElement('newElement');
blueprint.appendChild(document.documentElement, newElement);
const newParent = blueprint.getParentNode(newElement);
// newParent is now set the documentElement node.

blueprint.setAttribute(newElement, 'attr', 'value');
const attrValue = blueprint.getAttribute(newElement, 'attr');
// attrValue is now set to the string 'value'
```

```
const isValid = validator.isValidNode(blueprint, newParent);
// The validator is passed the blueprint so it can see
// whether the new structure is valid

if (isValid) {
  // We know it's valid, so we can safely apply these changes to the DOM
  blueprint.realize();
}
```

This approach allows us to make atomic changes to the DOM; we can revert everything by just not applying the new relations. The programming model for these commands is more maintainable and easier. Instead of trying to predict what a command will result in, the command can just apply changes and query them.

Because these changes are atomic, we can implement a very simple ‘getState’ function for all of these commands. Instead of seeing whether a command can work, we can just run the normal command, validate the outcome and simply not apply its changes to the actual DOM.

Bueprints let us make a proposed change to the dom. We extend this concept to ‘overlays’, which are essentially blueprints over blueprints. By using these overlays, we can compose different ‘atomic’ changes to essentially ‘puzzle’ our way through a mutation; making edits as we go along, but being able to revert them as we go along.

3.5. Schema-independent primitives: vertical insert nodes

Blueprints and overlays let us make changes to the document and revert them. This caused the birth of schema-independent primitives like ‘insertNode’, which attempts to insert a node by splitting the ancestry:

```
Vertical insert nodes:

Let $position be the position at which we will insert,
  consisting of a container node and an offset
Let $node be the new node we'd like to insert

A:
Start an overlay on the blueprint to contain all the changes in this
mutation.

B:
If $position resides in a text node:
Split the text node at the given offset
Set $position to after the first half of the result
```

```
C:
Start an overlay on the blueprint
Insert the $node at the given position
Validate the parent of $node, using the new relations present in the
blueprint

If the result is valid:
  Apply all pending overlays on the blueprint
  Return true to indicate success

D:
If the result is invalid:
  Discard the overlay on the blueprint to revert the insertion of $node
  If the container of the $position is the document element:
    Discard the blueprint overlay and return false to indicate failure.
  Otherwise:
    Split the container of the $position at the given offset
    Insert the new second half of the container to after the first half.
    Set the $position to after the first half and before the second half
      of the old container.
    Continue at section C.
```

3.6. Operations

At this point in time, these mutations were composed using JavaScript. This works well for most mutations, but soon we found ourselves re-inventing the same commands over and over and that flowing data from things like modals to command were cumbersome to implement. We currently still use a component that we invented during that timeframe: operations.

Operations are pipelines that allow composing XML manipulating "commands", data-modifying "transforms", modals and other operations. Arguments for the command, which is usually the last step, flow through the pipeline and are modified and appended to by each of the steps.

As with commands, operations support a "getState" mode, which performs a dry-run in order to determine the validity of an operation without affecting the current state of the application.

3.6.1. JSONML

These operations are written in JSON files, which pose another challenge. XML does not let itself be inlined all too well in a format like JSON. Line breaks in JSON strings are disallowed and it's easy to forget to escape a double quote somewhere. We often use JSONML[4]when an XML fragment should be made serializable or could be included in JSON at some point, like the operations pipeline.

JSONML also blends in nicely with our existing data interpolation functionality in operation.json files:

```
{
  "insert-a-note": {
    "label": "Insert note",
    "summary": "insert a note, with the type attribute set from a modal",
    "steps": [
The ask-for-type modal will write to the 'type' of the
 data object we pass along in the pipeline.
      { "type": "modal/ask-for-type" },
      {
        "type": "command/insert-node",
        "data": {
          "childNodeStructure": [
            "note",
            {
The following line lets us set the 'type' attribute to
 the value of the modal
              "type": "{{type}}"
            }
          ]
        }
      }
    ]
  }
}
```

The main limitation of the operation pipeline lies in predictability; an operation does not explicitly define which keys of the 'data' object it reads from, nor does it define to which keys it will write. We attempted to resolve this using documentation, but in practice, this does not adequately address the problem.

4. Iteration 2

4.1. Families, Content Visualization Kit

Editors using the FontoXML platform were growing to cover more and more different schemata, but all elements were configured all of the different 'flow' properties separately. Different FontoXML implementations 'felt' differently and were getting inconsistent in their behaviour of similar elements. However, most elements represented the same 'concept' within a document. Some act like blocks of text, some act like unsplittable, semantic inlines. We call these roles 'families', which each define a set of flow properties and a visualization, intended to make its properties predictable for an author.

These families are made up of three different groups:

- **Containers**,which contain blocks
- **Blocks**, which contain lines
- **Inlines**, which can contain other inlines and text

These different groups consist of different families, which may have their own sub-families:

- **Frame**: An unsplittable entity, which contains blocks. They are ignored for navigation next to the default text container so that the arrow-keys directly jump between the blocks it contains. Frames are visualized with a border to indicate they can't be split using the enter key.
- **Sheetframe**: A sub-family of **frames** are sheet-frames, which represent whole documents or units of work. They are the root element in FontoXML, the white 'sheets' in which one types.
- **Block**: A 'block' of text, like an HTML `<p/>` element. They are splittable, not detached, removable if they're empty, but not automatically removable, nor automatically mergeable. They are not ignored for navigation, but their parents' elements often are.
- **Inline formatting**: Like the HTML `<b/>`, `<i/>`, `<sup/>` elements. They may be split, do not influence the behaviour of the enter keys but are automatically mergeable.
- **Inline frames**: The inline variant of a **frame**. Also unsplittable, but they can directly contain text.

4.2. Selectors

Nodes are assigned a family using a system inspired by CSS selectors, using a JavaScript fluent API:

```
// Configure elements matching the
// 'self::span[@type="bold"] and parent::span' XPath selector
configureAsInlineFormatting(
  matchNodeName('span')
    .withAttribute('type', 'bold')
    .requireChild(matchNodeName('span')),
  'Parent of nested span')
```

If an element matched multiple configurations, we chose one based on the 'specificity[5]' of the selector. The specificity algorithm is based on CSS selector specificity.

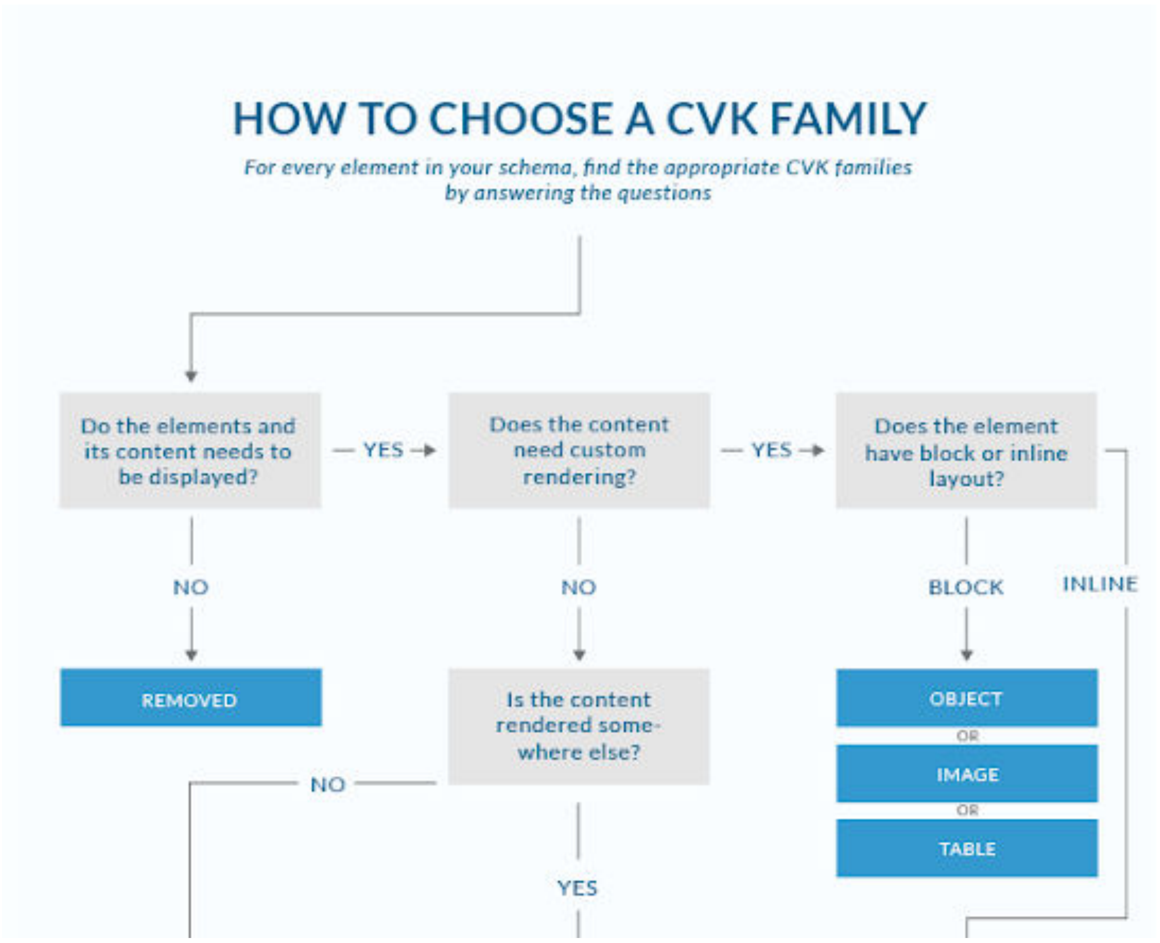
HOW TO CHOOSE A CVK FAMILY
For every element in your schema, find the appropriate CVK families by answering the questions
Do the elements and its content needs to be displayed?
YES
Does the content need custom rendering?
YES
Does the element have block or inline layout?
NO
NO
BLOCK
INLINE
REMOVED
Is the content rendered some-where else?
OBJECT
OR
IMAGE
OR
TABLE
NO
YES

Figure 4. Excerpt of the "CVK family chart"

4.3. Stencils

The new generic commands were powerful when combined with the operations pipeline, but they were hard to target. Provisioning modals with data also started to be very code-heavy. We needed a new component that could read from and write data to the dom: stencils.

Stencils are an extension on JSONML, with regex-like ‘gaps’ that can catch nodes. They can be used much like XPath, but they describe a structure that can be matched much like a schema. For instance, a stencil matching a <figure> element, catching the contents of the <title> element and the value of the @href attribute looks like this:

```
[
  "figure",
  [
    "title",
    [{
      "bindTo": "titleGap",
        "multiple": true,
        "required": false
      }]
    ],
    ["img", { "href": { "bindTo": "hrefGap"}}]
  ]
```

This stencil would allow both reading from and setting the value of the title using the "titleGap" gap.

Stencils fulfil another role as well: matching a DOM structure. The following stencil matches a `<metadata>` element with a `<title>` (disregarding its contents) and a `<related-links>` element, which contains an empty `<link>` element with the `@href` attribute set to a certain value:

```
  [
    "metadata",
    ["title", [{"multiple": true, "required": false}],
    [
      "related-links",
      {"multiple": true, "required": false},
      ["link", {"href": "a certain value"}],
      [{"multiple": true, "required": false}]
    ]
  ]
```

This stencil matches the same structures as the XPath expression `self::metadata[title and related-links/ link/ @href = "a certain value"]`: match all `metadata` elements with a `title`, with a `link` to `"a certain value"`. The stencil is subjectively easier to manage in some cases because it can perform two-way data binding as well test whether a node aligns with it. Obvious downsides of these stencils are related to the complexity of them. Stencils were later enhanced by:

- allowing nested XPath expressions;
- omitting nodeNames, to allow us to match a node regardless of its name;
- requiring the selection to either be fully contained, start before, or end after a given node;
- setting the selection after the command was executed;
- and many more features.

This did make stencils more powerful, and the selection mechanics allowed them to match structures which XPath expressions could not. This did not make them easier to write, and we are working on deprecating this API in favour of more XPath and XQuery usage.

4.4. Extender

'Fuzzy commands' became one of the most powerful features of the FontoXML editor platform. These are primitives like InsertNodes: '*insert a node at the current cursor position, splitting anything that has to be split to allow for this new element to be placed*'. A new related need arose: *insert a new element somewhere under a given element, at any offset.* This can be computed using a new extension upon schema validation and synthesis: extension.

The exact inner workings of the extension component is too complex to cover in this paper and should be covered more in-depth in a future one.

5. Iteration 3 (now)

5.1. XPath

The editor API was quickly growing to using too many proprietary 'standards', which were invented by our own. This makes it difficult to explain to an outsider how to use our platform for their own editor. Far too many configurations were reverting to using custom JavaScript callbacks to express their concerns, for example when assigning elements to a family. This sparked the need of an XPath engine, preferably one that could leverage a number of optimizations we implemented for our current Selectors implementation. We also wanted to use the new features that the XPath 3.1 standard defines, which was in its Candidate Recommendation phase at the time. At that time, there were no usable implementations, so we wrote an open source XPath 3.1 JavaScript engine[3].

5.2. XPath observers

A number of the editors require knowledge of which kind of elements are present in a document, and what information is related to them. This information is used in, for example, a Schematron panel. We provide a way to detect which dom-relations an XPath query traverses, and a way to be notified when these relations have been altered, so we can update our user interface as few times as possible. This process is further explained in our earlier paper[6].

5.3. XQuery & Update Facility

At this moment, we are in the process of writing an XQuery 3.1 + XQuery Update Facility implementation, so that we can port our DOM manipulation algorithms to those standards and expose them from an XQuery API.

We are still learning how we can express concepts that are not present in those standards, like the selection, DOM positions, the blueprint concept, and 'fuzzy' commands. These concepts are what makes writing editor commands different from for instance writing database update queries.

Database update queries are usually fully aware of the restrictions a schema enforces. This contrasts with editor commands where a certain kind of 'just make it work' mentality is required to correctly interpret what an author wants to do.

6. Conclusion

When one writes a platform for an XML editor in JavaScript, they should be prepared to write a lot of the XML tools themselves (though this may have changed in the last seven years). Furthermore, while it does pay off to design your own configuration languages due to the flexibility of them and the ability to keep it simple, using more complete standards like XPath and XQuery will work better in the long run because those technologies will inevitably have a larger community around them.

Bibliography

[1] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. 2005. Taxonomy of XML schema languages using formal language theory. ACM Trans. Internet Technol. 5, 4 (November 2005), 660-704.

[2] Implementing regular expressions: `https://swtch.com/~rsc/regexp/`

[3] A minimalistic XPath 3.1 implementation in pure JavaScript `https://github.com/FontoXML/fontoxpath`

[4] `http://www.jsonml.org`

[5] `https://drafts.csswg.org/selectors-4/#specificity-rules`

[6] Martin Middel. Soft validation in an editor environment. 2017. `http://archive.xmlprague.cz/2017/files/xmlprague-2017-proceedings.pdf`

[7] `https://github.com/bwrrp/whynot.js`