

XML document authoring with *emacs* `nxml` - mode



John W. Shipman

2011-03-22 13:44

Abstract

Describes an XML editing module for the *emacs* text editor.

This publication is available in Web form¹ and also as a PDF document². Please forward any comments to tcc-doc@nmt.edu.

Table of Contents

1. The problem of XML validity: why <code>nxml</code> - mode?	1
2. Configuring your <code>.emacs</code> file	2
3. How <i>emacs</i> finds your schema	3
4. Format of a schema locating file	3
4.1. Schema locating rules	3
4.2. Type identifiers and how they work	5
4.3. The default <code>schemas.xml</code> file	6
5. Special <i>emacs</i> commands	7
5.1. Schema commands	7
5.2. The completion command	8
5.3. Validation commands	8
5.4. Markup commands	9
5.5. Cursor motion commands	9
6. Toward faster work: template files	10
6.1. Templates for block elements	10
6.2. Templates for inline elements	11
7. Decluttering your project directory	11

1. The problem of XML validity: why `nxml` - mode?

Anyone who works with XML files has to deal with validation. How do we make sure that the file is *well-formed*, that is, legal under the rules of XML?

More importantly, how do we insure that the file is *valid*, that is, conforming to the *schema* that describes what elements and attributes are allowed for the given document type?

One solution is to use the *emacs* text editor, with the optional `nxml` - mode package, for document creation and maintenance.

To use `nxml` - mode, you'll need to be familiar with these tools:

¹ <http://www.nmt.edu/tcc/help/pubs/nxml/>

² <http://www.nmt.edu/tcc/help/pubs/nxml/nxml.pdf>

- The basics of XML. See the Tech Computer Center's *XML help page*³.
- The *emacs* text editor. See *The emacs text editor*⁴.

If you're using an existing document type like DocBook, XSLT, RDF, or XHTML, that's all you'll need.

However, if you're going to invent your own XML document types, instead of using a DTD (Document Type Definition) or the XML Schema language, we recommend that you take a look at the Relax NG Schema language, especially when expressed in the delightfully concise compact syntax (RNC). For a comparison of several current schema languages, and why we prefer Relax NG at this time, see *Relax NG compact syntax*⁵.

Warning

These notes describe the 20060901 version of `nxml-mode`. This package works best with *emacs* version 21.3 or later. It will not work with version 20 or with *xemacs*.

2. Configuring your `.emacs` file

In order to use the `nxml` package under *emacs*, you must add these lines to your `.emacs` file, so that *emacs* knows where to find the `nxml` package:

```
;; /usr/share/emacs/site-lisp/tcc-nxml-emacs: Add these lines
;; to your .emacs to use nxml-mode. For documentation of
;; this mode, see http://www.nmt.edu/tcc/help/pubs/nxml/
;;--
;; Add the nxml files to emacs's search path for loading:
;;--
(setq load-path
      (append load-path
              ('("/usr/share/emacs/site-lisp/nxml/"))))
;;--
;; Make sure nxml-mode can autoload
;;--
(load "/usr/share/emacs/site-lisp/nxml/rng-auto.el")

;;--
;; Load nxml-mode for files ending in .xml, .xsl, .rng, .xhtml
;;--
(setq auto-mode-alist
      (cons '("\\.\\(xml\\|xsl\\|rng\\|xhtml\\)\\'" . nxml-mode)
            auto-mode-alist))
```

If adding these lines breaks *emacs* for other uses, you can set up a separate customization file just for `nxml-mode`.

For example, you can make a copy of your `.emacs` file named `nxml.emacs` and add the customizations shown above. Then to run *emacs* with `nxml`, use this command:

```
emacs -q --load ~/nxml.emacs
```

³ <http://www.nmt.edu/tcc/help/xml/>

⁴ <http://www.nmt.edu/tcc/help/pubs/emacs>

⁵ <http://www.nmt.edu/tcc/help/pubs/rnc/>

The `-q` option prevents *emacs* from reading your `.emacs` file, and the `--load ~/nxml.emacs` option causes it to read your customized setup file instead.

3. How *emacs* finds your schema

emacs has to know where your schema lives so that it can help you build and maintain a valid document. This is a two-stage process:

1. *emacs* has an internal variable named `rng-schema-locating-files` that tells it all the places to search for *schema locating files*. This variable's default value is `('schemas.xml', 'DIST-DIR/schema/schemas.xml')`, where `DIST-DIR` is the distribution directory where `nxml` is installed.

The `schemas.xml` file in the distribution directory handles many of the common cases such as XSLT files or DocBook documents.

You can also have a `schemas.xml` file in the same directory as the document you are working on. This file is also a schema locating file.

2. Each schema locating file specifies rules for associating files with schemas. The rules can use the file's extension (such as `.xsl`) or the file's root element (such as `article`) to infer the correct schema. If all else fails, it can simply specify `file X uses schema Y.`

When *emacs* opens your file, it looks through the schema locating files in order until it finds a match.

4. Format of a schema locating file

Schema locating files are in XML format. The Relax NG compact schema for this document type lives in file `schema/locate.rnc` in the `nxml` install directory.

Here's a simple example:

```
<?xml version='1.0'?>
<locatingRules xmlns="http://thaiopensource.com/ns/locating-rules/1.0">
  <uri resource="myfile.xml" uri="myschema.rnc"/>
</locatingRules>
```

The first two lines, and the last line, are the same in every file. Between them are *rules* that specify various ways to find schema files for a given XML document.

The third line is a rule that says the Relax NG compact schema for file `myfile.xml` lives in file `myschema.rnc` in the same directory.

Relative URI (Universal Resource Identifiers) values are interpreted as relative to the same directory as the schema locating file.

When *emacs* is looking through the rules in a schema locating file, it looks at each rule until it gets a match, and then stops. So the order of the rules is important: place the more important rules first.

The various rules that can appear in schema locating files are described below.

4.1. Schema locating rules

Here is a list of the rules that can appear in a schema locating file:

<uri resource="f" uri="s"/>

Associate resource *f* with schema file *s*. The resource may be a full URI or a local file name.

For example, this rule would associate file `abc.xyz` with schema `xyz.rnc`:

```
<uri resource="abc.xyz" uri="xyz.rnc"/>
```

<uri pattern="p" uri="s"/>

Like the previous form, but *p* may contain one or more wild-card characters `"*"`.

For example, this rule would associate all files ending in `.xyz` with schema `xyz.rnc`:

```
<uri pattern="*.xyz" uri="xyz.rnc"/>
```

<uri pattern="p" typeId="T"/>

A rule of this form associates any URI matching pattern *p* with type identifier *T*. See Section 4.2, "Type identifiers and how they work" (p. 5).

<documentElement prefix="N" localName="E" uri="s"/>

If a document's root element is *E* in namespace *N*, use schema file *s*.

If you omit the namespace identifier `prefix="N"`, this rule matches by document element regardless of the namespace.

If you omit the element name `prefix="E"`, this rule will match all root elements in namespace *N*, if given. If both `prefix` and `localName` attribute are omitted, it matches all documents. Example:

```
<documentElement localName="horses" uri="horse.rnc"/>
```

This will associate any file whose root element is `horses` with schema file `horse.rnc`.

<documentElement prefix="N" localName="E" typeId="T"/>

Like the preceding form, but it associates the matching files with type identifier *T*; see Section 4.2, "Type identifiers and how they work" (p. 5). Examples:

```
<documentElement localName="stylesheet" typeId="XSLT"/>
<documentElement prefix="xsl" localName="transform" typeId="XSLT"/>
```

The first example will associate any file whose root element is `stylesheet` with type `XSLT`. The second example associates any file whose root element is `xsl:transform` with that same type code.

<transformURI fromPattern="p" toPattern="q"/>

If a file has a name of the form `"p"`, try to match a schema file whose name is `"q"`. The patterns may contain one or more `*` characters. For each `*` in pattern *p*, the text that matches the `*` will be substituted for the corresponding `*` in pattern *q*.

Here's an example. The following rule means that for any file ending in `".xml"`, if there is a schema by the same name ending with `".rnc"`, use that schema.

```
<transformURI fromPattern="*.xml" toPattern="*.rnc"/>
```

So for example if you have a file named `"horses.xml"` and a schema file `"horses.rnc"`, the above locating rule would associate them.

<namespace ns="N" uri="s"/>

If the file's root element has the same namespace URI as *N*, use schema *s*.

<namespace ns="N" typeId="T"/>

Like the previous form, but files in namespace *N* will be associated with type id *T*.

<typeId id="T" uri="s"/>

Defines a new *type ID T*, and says to use schema *S* for rules that refer to that type ID. A type ID name is an arbitrary string you make up to describe a document type (schema).

In any of the rules that have a `uri="s"` attribute, you can instead use a `typeId="T"` attribute. Assuming you have defined the type ID *T*, the rule will use the schema related to that type ID.

<typeId id="T" typeId="U"/>

Defines a type ID named *T* as the same as type ID *U*.

<include rules="F"/>

Includes another schema locating file *F*.

<applyFollowingRules ruleType="R"/>

This rule instructs *emacs* to immediately search all the remaining schema locating files in its list (the value of its `rng-schema-locating-files` variable), and apply all rules of type *R*.

For example, suppose you want to use the transform rewrite rules of type `transformURI` that are in the files after this one in the sequence of schema locating files. You can't just use the `include` rule, because if any rule in the later files matches, it will win. So instead you use this rule:

```
<applyFollowingRules ruleType="transformURI"/>
```

If *emacs* gets to this rule without a match, it will then go and apply all the `transformURI` rules from later files, using them to rewrite the pathname if appropriate. Then matching continues in the current file, using the rewritten pathname.

4.2. Type identifiers and how they work

If you are writing a general-purpose schema locating file, type identifiers declared using `typeId` rules can be most useful.

Here's an example. Suppose you want all files ending in `.html`, and all files that use the XHTML namespace URI, to use the XHTML Strict schema. You could use `<uri pattern="*.html"/xhtml-strict.rnc/>` and `namespace` rules that refer directly to the XHTML Strict schema. However, consider this fragment of a schema locating file:

```
<uri pattern="*.html" typeId="XHTML" />
<namespace ns="http://www.w3.org/1999/xhtml"
  typeId="XHTML" />
<typeId id="XHTML" typeId="XHTML Strict"/>
<typeId id="XHTML Strict"
  uri="xhtml-strict.rnc"/>
<typeId id="XHTML Transitional"
  uri="xhtml-transitional.rnc"/>
```

The first rule associates all files ending in `.html` with type ID "XHTML".

The second rule associates all files with default namespace `"http://www.w3.org/1999/xhtml"` with type ID "XHTML".

The third rule defines type ID "XHTML" as the equivalent of another type ID "XHTML Strict".

The last two rules relates type IDs "XHTML Strict" and "XHTML Transitional" to specific schema files.

The advantage to this indirect linking of rules to schema is that, if you want to, you can change from XHTML Strict to XHTML Transitional by changing the third line to:

```
<typeId id="XHTML" typeId="XHTML Transitional"/>
```

4.3. The default schemas .xml file

Here is schema/schemas.xml, the default schema locating in the nxml installation directory.

```
<locatingRules xmlns="http://thaiopensource.com/ns/locating-rules/1.0">
  <transformURI fromPattern="*.xml" 1
    toPattern="*.rnc"/>

  <uri pattern=".xsl" typeId="XSLT"/> 2
  <uri pattern=".html" typeId="XHTML"/>
  <uri pattern=".rng" typeId="RELAX NG"/>
  <uri pattern=".rdf" typeId="RDF"/>

  <namespace ns="http://www.w3.org/1999/XSL/Transform"
    typeId="XSLT"/> 3
  <namespace ns="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    typeId="RDF"/>
  <namespace ns="http://www.w3.org/1999/xhtml"
    typeId="XHTML"/>
  <namespace ns="http://relaxng.org/ns/structure/1.0"
    typeId="RELAX NG"/>
  <namespace ns="http://thaiopensource.com/ns/locating-rules/1.0"
    uri="locate.rnc"/>

  <documentElement localName="stylesheet" 4
    typeId="XSLT"/>
  <documentElement prefix="xsl" localName="transform" typeId="XSLT"/>

  <documentElement localName="html" typeId="XHTML"/>

  <documentElement localName="grammar" typeId="RELAX NG"/>

  <documentElement prefix="" localName="article" typeId="DocBook"/>
  <documentElement prefix="" localName="book" typeId="DocBook"/>

  <documentElement localName="RDF" typeId="RDF"/>
  <documentElement prefix="rdf" typeId="RDF"/>

  <documentElement localName="locatingRules" uri="locate.rnc"/>

  <typeId id="XSLT" uri="xslt.rnc"/> 5
  <typeId id="RELAX NG" uri="relaxng.rnc"/>
  <typeId id="XHTML" uri="xhtml.rnc"/>
  <typeId id="DocBook" uri="docbook.rnc"/>
  <typeId id="RDF" uri="rdfxml.rnc"/>
</locatingRules>
```

- 1** If a file's name is *f.xml*, and there is a Relax NG compact schema file *f.rnc*, use that schema. For example, if your file is named *kites.xml* and its Relax NG compact schema lives in file *kites.rnc*, that schema will be used.

- 2 This line and the three that follow set up rules for the standard file extensions for XSLT, XHTML, Relax NG, and RDF (Resource Definition Framework) files. Each rule refers to a type ID defined further down. For example, any file whose name ends in `.xsl` is referred to type ID "XSLT", which is later connected to the XSLT schema in `xslt.rnc`.
- 3 These five elements set up rules for files by namespace URIs: XSLT, RDF, XHTML, Relax NG (in the XML format), and RNC (Relax NG in the compact format).
- 4 These elements set up default schemas for XSLT, XHTML, Relax NG, DocBook, RDF, and RNC document types, based on the name of the root element. For example, any file whose root element is `article` is assumed to be a DocBook article.
- 5 These lines connect the type IDs used above to the actual schema files. Because local URIs are resolved relative to the schema locating file in which they live, these rules will find the RNC schema files that are located in the same directory as the default `schemas.xml` file.

5. Special *emacs* commands

The previous section discusses how schema location files work to help *emacs* find the right schema for your file. If the default rules work for your case, you can proceed immediately to edit your file.

However, if you want to create a file that does not match any of the rules in the default schema location file, you have two choices.

- If you already a schema location file named `schemas.xml` in the same directory, you can add a new rule to it that associates your file with the schema. If you don't have such a file, you can create one, and then add the new rule. See Section 4, "Format of a schema locating file" (p. 3) for rule formats.
- There are special keystroke sequences that will connect the current file with a schema that you specify. This will create a `schemas.xml` file if there isn't one already, and then add a rule to that file that makes the connection.

When you create a new XML document file, if your file doesn't have one of the file suffixes that automatically triggers `nxml-mode` (such as `.xml` or `.xsl`), you can put *emacs* into `nxml-mode` manually with `M-x nxml-mode`.

Below are all the special keystrokes available in `nxml-mode`. As in the *emacs* reference guide⁶, the prefix `C-` means control and `M-` meta (either combine with *alt* or prefix with *esc*).

Also, these terms have their usual *emacs* meanings:

- *Point* is the location of the cursor.
- *Mark* is the location marked with `C-@`.
- The *region* is the text between point and mark.

5.1. Schema commands

<p><code>C-c</code> <code>C-s</code> <code>C-t</code></p>	<p>If you are creating a new document in one of the supported standard document types, you can start your document with this command. You will be prompted in the minibuffer for the document type string. Type one of these strings for the supported standard types: <code>DocBook</code>, <code>RDF</code>, <code>RELAX NG</code>, <code>XHTML</code>, or <code>XSLT</code>. You can use <i>tab</i> for completion in the minibuffer. Then <i>emacs</i> will ask you if you want to save that schema association in your local <code>schemas.xml</code> file.</p>
---	--

⁶ <http://www.nmt.edu/tcc/help/pubs/emacs>

C - c C - s C - w	Asks what schema you are using. If your document is connected with a schema, the path to the schema appears in the minibuffer. If no schema is in effect, you will get the message "Using vacuous schema".
C - c C - s C - f	Manually selects a schema for the current document. You will be prompted in the minibuffer with "Schema file: " followed by the current directory's path. Edit the minibuffer to reflect the path to the schema you want, then press <i>enter</i> .
C - c C - s C - a	If you are using a root element that is defined in the schema selection files, you can connect your document to the schema by typing just the "<" and tag name for the root element, then typing C - c C - a. This tells <i>emacs</i> to go through the schema selection process again, and if the root element matches any of the rules, <i>emacs</i> will know what schema you want.
C - c C - s C - l	Add a rule to the local schema locator file <code>schemas.xml</code> that connects the current document to its schema.

5.2. The completion command

C - <i>enter</i> or M - <i>tab</i>	Completion. If <i>emacs</i> is running in its own window, use C - <i>enter</i> , otherwise use M - <i>tab</i> .
---------------------------------------	---

The action of this command is somewhat complex. This command can be used to finish typing a tag or attribute name once you have entered an unambiguous substring.

For example, if you are writing XHTML and you enter "<tab" somewhere that a `table` element is allowed, and then request completion, *emacs* will add the rest of the characters of the start tag so you will then have "<table".

As another example, suppose you have typed "<table b" and you then invoke completion. Since the only attribute of a `table` element that starts with `b` is `border`, after completion you will see "<table border="". Typing the attribute value and the closing ">" sequence are up to you.

If the characters you typed can be the initial characters of more than one element or attribute name, the screen will split and you will see a list of the permissible names in the other half of the screen. You will be prompted in the minibuffer with "Tag: "; type enough additional characters to make the choice unambiguous and then press *enter*, and the completion will be added to your document.

If you can't remember a tag name, you can type just the "<" symbol and then invoke completion. The split screen will display all the permissible tags at the cursor's location.

If you are creating a start tag and can't remember an attribute name, just type one space and invoke completion. For example, if you have typed "<table " (note the space after the tag name) and you invoke completion, the split screen will show you all the possible attributes of the `table` element.

5.3. Validation commands

Normally, as soon as *emacs* associates your document with a schema, `nxml-mode` constantly validates your document against the schema with every keystroke.

If validation is in effect and your document is not valid against the schema, the point where the document becomes invalid is shown with thin red underlining.

These commands will assist you in validating the document:

C - c C - n	<p>Move to the next location where the document structure is not valid. If the document isn't valid, the cursor will jump to the probable error, and display a message in the minibuffer explaining what it doesn't like.</p> <p>If the document is valid from the cursor to the end of the file, the message "No more errors" appears in the minibuffer.</p> <p>To validate the entire document, move to the top of the document with M-< and then use this key sequence. If it says "No more errors" with point at the top of the file, the entire file is valid; otherwise it will jump the cursor to the next invalid content.</p>
C - c C - v	<p>Turn validation on or off. If validation is turned on, in the status line's mode area you will see either "nXML Valid" or "nXML Invalid". If validation is turned off, neither word will appear after "nXML" in the mode line. You may want to turn validation off during serious document surgery, then turn it back on when you think it's valid again.</p>

5.4. Markup commands

<i>tab</i>	Indent the current line according to the level of nested block tags. The indentation is two spaces per level.
M-C-\	Indent all the lines in the region using the same process as for <i>tab</i> .
C - c C - f	Insert an end tag for whatever element the cursor is in. This works whether you are still inside the start tag or in the content.
C - c C - i	Used when you have finished the start tag of an inline element, up to but not including the closing ">". This command adds the closing ">" and an end tag, and then places the cursor between the tags so you can type the content.
C - c C - b	Like C - c C - i, but used with block elements. The command adds the closing ">", then a blank line, then an end tag on yet another separate line. The cursor is left indented at the proper level on the central blank line.
M - q	Reformat the paragraph containing the cursor. This works best if the content does not start on the same line as the start tag.
C - c C - x	<p>Inserts an XML processing instruction at the top of your file:</p> <pre><?xml version="1.0" encoding="utf-8"?></pre>

5.5. Cursor motion commands

M - C - f	Move forward over tag. If point is not inside a tag, it moves to a position just before the next tag. If point is inside a start tag, it jumps to a position just before the closing ">". If point is inside an end tag, it moves just past the end tag.
M - C - b	Move backward over tag. If point is not inside a tag, it moves just after the previous start tag. If point is inside a start tag, it jumps to a position just after the starting "<". If point is inside an end tag, it moves just before that tag.
M - C - n	Move the cursor to the end of the next element.
M - C - p	Move the cursor before the previous element.
M - C - d	Move the cursor to the next included element after point, to a position just after the start tag; d is for "down."

M-C-u	Move the cursor to a position just before the start tag of the element containing point; u is for “up.”
C-c C-o C-d	Hide the children of the current element, as in <i>emacs</i> outline-mode.
C-c C-o C-s	Reverses the action of C-c C-o C-d, revealing the children of the current element.

6. Toward faster work: template files

When you are writing an XML document with this tool, it is useful to accumulate a set of “template” files—small fragments of XML skeletons that you can paste into your document with the *emacs* command `insert-file` (usually bound to `C-x i`). The author prefers to maintain these templates in a central location, such as a first-level subdirectory of his home directory named “~/i”.

6.1. Templates for block elements

For example, in a DocBook document⁷, a procedure has this structure:

```
<procedure>
  <step>
    <para>
      First step....
    </para>
  </step>
  <step>
    <para>
      Second step....
    </para>
  </step>
  ...
</procedure>
```

If you are going to be writing a lot of DocBook procedures, it saves time to have two small template files lying around that you can insert into your document. The first—call it `proc`—contains the skeleton of a procedure with one step:

```
<procedure>
  <step>
    <para>

    </para>
  </step>
</procedure>
```

In this skeleton, the line inside the `para` element is empty, so you can fill in the text of the step.

The second template file contains just the skeleton of a step. Let's call this file `step`:

```
<step>
  <para>
```

⁷ <http://www.nmt.edu/tcc/help/pubs/docbook42/>

```
</para>
</step>
```

So the workflow for writing a procedure becomes:

1. Insert a copy of the `proc` file.
2. Fill in the text of the first step.
3. For each additional step, insert a copy of the `step` file, and fill in its text.

If you are finicky about maintaining proper indentation, and the template file isn't indented correctly for your current context, that's easy to fix. Just after inserting the template, issue the command `indent-region (M-C-l)`. In `nxml-mode`, this command re-indent all the lines in the region; since the region is set to the inserted portion just after using `insert-file`, only the text just inserted will be re-indented.

6.2. Templates for inline elements

Inline elements can also benefit from the template technique.

Another example of a frequently used element in DocBook is the `code` element, used to indicate text that is typed by the user. Here is a template file, named `us`, for this element:

```
<code >
</code >
```

The workflow for marking up a bit of text as user input goes like this:

1. Insert the `us` template file.
2. Move to the end of the start tag (with `C-e`) and type the text to be marked up.
3. Use `C-d` to delete the newline so that the end tag abuts the end of the marked-up text.

Note the extra spaces before the closing `>` of the tags in the template. Because XML allows whitespace before the closing `>` of a tag, this furnishes more places where long lines can be broken by the `fill-paragraph (M-q)` command, or the action of `auto-fill-mode`.

7. Decluttering your project directory

Once you accumulate a lot of template files, they tend to clutter up the directory where you are building your XML document. Follow this procedure to move them to a subdirectory out of sight and use them from there.

1. Start in the directory where the XML document lives.
2. Make a subdirectory `i/`:

```
mkdir i
```

3. Move your templates to that directory. For example:

```
mv proc step i
```

4. Add these lines to your `.emacs` file:

```
;; Define a command to insert a fragment from the i/ subdirectory.
;;
(defun insert-frag (name)
  "Like insert-file but prepends 'i/' to the path given."
  (interactive "MTemplate: ")
  (insert-file (concat "i/" name)))
;;
;; Bind the above command to C-c C-e.
;;
(global-set-key "\C-c\C-e" 'insert-frag)
```

5. To insert a template named *F*, use the key sequence “C-c C-e *F*”.