XML

Guide for students of Medieval Words, Modern Methods

P. S. Langeslag

revision of August 28, 2025

Contents

Requirements

This document assumes you have installed VS Code with the XML extension, and have obtained the course materials from the GitHub repository.

XML File Structure and the TEI Header

For the purposes of this seminar, you will be working with a single XML file based on `template.xml` from the GitHub repository. Duplicate the file and rename the copy after the text you intend to transcribe, or call it whatever you want to call it for now.

Upon opening the template, you should be looking at the contents of a 60-line XML file. You will find that the entire file is comprised of a series of **nodes** that are instances of specific **elements**, i.e. metadata containers, which are expressed in pairs of **tags** set apart by angular brackets, such as `<title>`, `<editor>`, or `<text>`. All valid elements for our particular framework are defined elsewhere, in the schema, and described in the TEI Guidelines, but some of the ones you should expect to work with actively are covered below, with reference to our specific use case.

The logical structure of XML has it that every instance of an element must be *closed*: thus to write a prose paragraph you start with the `<p>` opening tag, followed by the paragraph content, whose end is signalled by the `</p>` closing tag; the forward slash indicates that the element is being closed. However, an opening tag can also be closed immediately without the need for a separate closing tag. This is typically done where no content is required: when we use `<pb/>` for "page beginning", for instance, the node is "**empty**" and has no text content but serves only as metadata indicating that a new manuscript side starts here (it is a **milestone element**), and so there is only a single tag, with the forward slash that closes it occuring immediately before the closing bracket of the sole or opening tag. In these cases, predefined kinds of information may still be contained in the form of **attributes**: `<pb n="18v"/>` marks the start of the verso of the 18th folio. In either form, the node must be closed with the help of a forward slash.

XML documents accommodate **nesting**: the content of one element, say `<library>` (in a hypothetical situation not covered by our schema), can include further nodes using such elements as `<bookCase>`, which in turn may contain lower structural levels such as `<book>`. It is, however, an unforgiving structure: if you close the containing element (the bookcase) before closing the contained element (the book), your document is invalid and will yield no output when you attempt to transform it into another format, such as PDF or HTML. Likewise, the schema regulates how elements may nest; a sensible schema would prohibit for bookcases to fit inside bookcases, or for bookcases to fit inside books. Our schema forbids for the lower-level element `<s>` (sentence) to contain a higher-level element `<p>` (paragraph), for instance. In the process of drafting XML, you will invariably end up spending time investigating why your code will not validate, or why the output of your transformation is not as expected, but your editor's validator and sometimes the feedback received from the transformation processor will provide the necessary guidance at least with the first of these challenges.

Mention was made above of **attributes**, which are optional or mandatory features contained within the opening (or sole) tag of an element and set off from the element name by a space. (Attributes are identified in the documentation using a preceding at-symbol, e.g. `@type`. While the `@` is in fact used in languages used to *navigate* and *process* XML, it is printed in XML documentation only to avoid confusion with elements; the `@` is not to be used in XML code!) Each attribute in turn has a **value**, which is supplied in (conventionally double, but always straight) quotes and set off from the attribute name by an equals sign. Any further attribute is again set off by a space:

```
<head xml:lang="la" rend="color(red)">DOMINICA PALMARUM</head>
```

Although technically each attribute must have exactly one value, we can use a custom

delimiter to allow us to encode more than one piece of information, which we may separate out when processing the file by relying on that same delimiter. We will use a single space as a delimiter between attribute substrings. Thus if the rubric from the above example is two lines tall in addition to being in red ink, its `@rend` attribute could read `rend="color(red) size(2)"`. Once we have settled on the space as a delimiter, we should ban the use of space characters in attribute values where the material to their left and right is not intended as two separate values; thus we cannot use `@reason="faded ink"` if we are also relying on space to delimit multiple substring values for `@reason`.

XML element and attribute names are **case sensitive**. This means you have to correctly and consistently reproduce the typographical case of the class you mean to reference. In the above example, the element `<bookCase>` uses a convention known as **camelcase**, using a capital only for the initial of each non-initial word element in the name. When working with a schema that registers an element `<bookCase>` but not `<bookcase>`, the latter form will not validate.

You will have noticed in the template that the lines are indented at different distances, like so:

```xml
<library id="Bodley">
  <bookCase name="Junius">
    <book n="11"/>
  </bookCase>
</library>
```

This is an optional but practical way of visualizing the nested structure of the elements: for every new child element, it is best to start a new line and indent it one additional unit. Coding editors will indent your lines for you as you work, and many have explicit formatting commands as well. VS Code has the keybinding `Shift+Alt+F` (Windows), `Shift+Option+F` (macOS) or `Ctrl+Shift+I` (Linux). Or if you forget, open the Command Palette with `Ctrl+Shift+P` and start typing `Indent using tabs`.

The header, which is what we call everything between `<teiHeader>` and `</teiHeader>` (ll. 18–52 of the template), contains metadata about the file, yourself, and the text and manuscripts covered in your transcription. In our template, action is required where a line starts with a complete XML tag but ends with instructions contained within `<!-- -->` tags, as these instructions specify what information to enter between the opening and closing tags. These special brackets signal a **comment**: anything contained within them is ignored by the processor and intended only for the benefit of those editing the raw markup. Code editors will usually render comments in a distinct colour so they are easily told apart.

The XML Ecosystem

XML files often rely on a larger infrastructure that may not be immediately apparent. There is, first of all, the **schema**, which is a document declaring what XML elements are valid and what attributes they may, or must, have. For our convenience, we will be using a remote (i.e. online) schema covering all the elements defined by TEI. The header of your XML file makes reference to this schema (l. 15). This serves two purposes: it informs anyone accessing the file what standard we are using, but it also serves as the benchmark for the **validation** process carried out by your editor to test your syntax against the schema.

In addition to the schema, there is often need for a **(transformation) stylesheet**, i.e. a document declaring what an HTML (or DOCX, ODT, LaTeX) file generated from your XML document should look like. Stylesheets specify not only traditional design features such as the base typeface and text colour, but also what formatting action any given XML element triggers, for instance that a rubric should be rendered in a certain font size, weight, and colour, that foreign-language text should appear in italics, and whether something encoded as appearing in red ink in the manuscript should in fact be reproduced in red. XML files may be styled in CSS, in which case the XML file is loaded directly in a browser but styled using information contained in the CSS stylesheet, and no new file is written to disk. However, for more complex cases we may need to generate new files in a format intended for reading or further processing through a process called **transformation** using an **XSLT transformation processor**. In this course we will first style our work using CSS. Because we want to access both the text and the metadata of our transcription in Python, we will pass over XSLT, instead using a Python library to access the content of our XML nodes directly.

Populating the TEI Header

Returning to your new document, you will see that its larger part is taken up by the TEI header (ll. 15–52). The logical structure of the TEI header is explained in this video on the topic; for our practical work, we'll limit ourselves to entering the necessary details into this basic header template. So go ahead and enter the title of the text or excerpt you are undertaking to edit into the <**title**> field (i.e. in between the <**title**> and </**title**> tags); enter your name into the <**editor**> field; and populate the descendants of the <**msDesc**> node with information on your manuscript. Just remember that the information is to be entered in between the opening and closing tags of each node, not after the latter, and not into the comment.

Starting Your Transcription

Next, scroll down below the template header. <**text**> is where our transcription starts, and in most cases we will encode all text content within <**body**>, as other children of the <**text**> node are for front and back matter, which are more typical of print publications as opposed to manuscripts. Inside the <**body**> node, you will have to format the encoding according to your choice of text: the template assumes verse, consisting of at least one <**lg**> (line group) consisting of a sequence of <**l**> (line) elements, but if you are working on prose, you will want to replace the whole <**lg**> node with <**p**> for the first of your paragraph-like block elements. If your text begins with a rubric, transcribe it as a <**head**> node *before* the corresponding <**lg**> or <**p**> node.

Prose and Verse

The paragraph is hardly an objective unit; if you are transcribing a prose text and are unsure where to start a new paragraph, work with a translation and use your best judgement.

Sequences of verse are not expected to occur within a `<p>` element. Instead, in addition to enclosing each *line* of verse within an `<l>` element, we wrap the full verse *sequence* in `<lg>` for "line group". Though not strictly required, it is best practice to number verse lines explicitly (`<l n="1">`), particularly if your excerpt does not start at line 1 of a poem. The `<lg>` (line group) element likewise is not mandatory, but it is recommended whenever two or more lines of verse form a sequence.

```
<p>The <title level="m">Aeneid</title> opens as follows:</p>

<lg>
  <l n="1">Arma virumque cano, Troiae qui primus ab orus</l>
  <l n="2">italiam fato profugus Laviniaque venit</l>
  <l n="3">litora, . . .</l>
</lg>
```

Validation

While you'll have to get used to understanding what you see in the raw XML, if you are using an XML editor there are a couple of tools at your disposal to help make sure all is well. The first of these is validation, which holds your code against the schema containing the rule set (in our case for the TEI specification of XML) and points out any mismatches. Every advanced XML editor comes with a validation service, and our template links to an online schema file (l. 15). You'll generally rely on *automatic* validation, which draws Word-like squiggly lines under invalid code as you edit your document. For instance, if in your new document you create a new line within the `<l n="1">` element and type `<head>` (to encode a manuscript rubric), it should only take a second or two before underlining appears. In VS Code, upon seeing the squiggly line appear you can retrieve the full text of the error either by opening the Problems panel (`Ctrl+Shift+M`) or with the debugger by hitting `F8`, or you can simply hover the squiggly line for a tooltip. Rubrics are not allowed inside paragraph elements; thus any text-initial rubrics should precede `<lg>` (or `<p>`) in the text body. VS Code shows the location of any errors in red in the scrollbar on the right. Live validation thus helps you maintain valid code as you type. If any issues are still outstanding when you are done transcribing, you may choose to identify and resolve any outstanding issues by initiating the formal validation process or navigating through VS Code's debugger, or you can simply click on or navigate to each of the red positions in the scrollbar and solve them using the cues provided by automatic validation.

Entities

Keybindings inserting special characters directly as glyphs, as recommended in the Software guide, are only advised for standard unicode glyphs that are ideally contained in the character sets of leading fixed-width typefaces as well. The reason we want them to be standard unicode, as opposed to “Private Use Area” characters, is that otherwise the wrong choice of font won’t render them at all, nor will it be clear what glyph they are meant to represent. The reason we want them represented in popular fixed-width fonts is that code editors conventionally use fixed-width fonts (as this allows for predictable formatting e.g. of tables), and if these can’t display your special characters they will come out as **tofus** (□) and you won’t be able to identify their intended value or tell them apart on sight.

In all other cases, you will need to encode your special characters as **entities**. This means you declare and describe them at the top of your XML file, assign them a user-readable identifier, and also provide the corresponding (hexadecimal) identifier of its slot within the unicode character set. You will see that our template `template.xml` has a few such declarations right at the top, contained within the doctype declaration:

```
<!DOCTYPE TEI [
  <!ENTITY et "&#x204A;"><!-- TIRONIAN SIGN ET -->
  <!ENTITY middot "&#x00B7;"><!-- MIDDLE DOT -->
  <!ENTITY punctelev "&#xF161;"><!-- PUNCTUS ELEVATUS -->
]>
```

For each of three special characters, these lines register a short reference (e.g. `et`) and the corresponding unicode point this short reference should expand to (e.g. `&#x204A`), in this case followed by a human-readable description of the character taken from the Menota entities list. Whenever you require further special characters in your transcription work, you’ll want to copy in the corresponding lines from that same entity list into your doctype declaration. Now to use these special characters in your document, you need to take the short reference (in the above example, that’s `et`, `middot`, or `punctelev` respectively) and enter it delimited by an ampersand on the left and a semicolon on the right, e.g. `&et;`. In fact, as soon as you type an ampersand in an XML coding editor such as VS Code with the XML extension, a dropdown menu appears with all the available options, i.e. all the entities declared within the document. This also means that the ampersand itself is a reserved character in XML: if you ever need to print the ampersand itself, you’ll have to use the entity code `&amp;`, which is standard and does not need declaring. If your validator should ever complain about undeclared entities, just make sure you have copied in the correct entity declarations and don’t mistype them anywhere, or use any entities that haven’t been declared.

Just two final notes on the general use of entities. First, whereas we use them here to gain access to special characters, their broader purpose is to be able to map longer or more complicated sequences to short references, so you could e.g. conveniently produce the string ‘North Atlantic Treaty Organization’ by typing simply `&nato;`. Second, entities, like XML element and attribute names, are case sensitive: if you’ve declared an entity as `nato`, you cannot invoke it with `&NATO;` but must type `&nato;`.

Styling and Transformation

To see the result of our work, we either need to reference a CSS stylesheet before opening the XML file in a browser, or we can transform our documents into a format meant for reading or further processing. Let's take these in turn.

CSS (Cascading Style Sheets)

With most XML files, if you open one in a browser, it is likely to warn you that it has no style information associated with it. Odds are it will output its contents nonetheless, but in document-tree format, i.e. much the same as in your code editor. If we now declare a stylesheet, it will look for style information you have provided in your CSS file. For anything not styled there it will fall back on the browser's defaults (this is the "cascade" referred to in the format's name). This means that if you declare a stylesheet that doesn't exist, your browser will simply print the content of your XML file, including the full header, in a default style, in a single line of output, with no further formatting. This is because XML uses custom elements, which the browser defaults can't possibly cover, and it is unlikely to have specifications even for a comparatively widespread standard like TEI. So we'll need to create and declare a stylesheet and populate it with style instructions.

Stylesheet declarations go near the top of your document, ahead of `<TEI>`. Thus the template contains the following stylesheet declaration:

```
<?xml-stylesheet type="text/css" href="tei.css"?>
```

Now if you open your transcription in a browser (opening it as a local file as you would in any other app, e.g. with the `Ctrl+O` keybinding), it should style some of the content, such as the title and the verse lines, though if you have completed more of the TEI header than just the basics, it will probably still output much of it in a default single-paragraph dump. To correct for this, you will have to duplicate `tei.css`, reference your new stylesheet in the XML, and add more CSS rules to format your document in accordance with your needs.

Though it has become remarkably powerful over the years, CSS is not an advanced scripting language. We can only tell it how to display each type of element, or not to display it. So when styling XML, the first step is usually to filter out some of the header information using the instruction `display: none`. After that, we may want to start each major unit of information on a new line using `display: block`, and style various elements using the properties `font-size` and `font-weight`. Look over the rules declared in your duplicate of `tei.css` and extend it with rules of your own; you can learn the basics of CSS at w3schools, or learn it more fully at web.dev.

CSS is not designed to add content (as opposed to style) that is not in the document. We can get around this in minor ways by using pseudoclasses, for instance by adding this instruction:

```
editor:before {
    content: 'edited by ';
}
```

But the bottom line to remember is that CSS is there to style content that is already there, and at most to disable some types of content from being displayed. Anything beyond that requires XSLT.

XSLT (Extensible Stylesheet Language Transformations)

Whereas CSS merely provides style information for documents and does not reshape them in any major way, XSLT can take your XML document and apply a more complex set of transformation rules to each element it encounters, or even mine it selectively for the information it wants and output that in an order and structure it prescribes. As such, its capabilities place it somewhere in between a stylesheet and an interpreted programming language, but by official definition it too is a stylesheet language. XLST processing outputs a new file to disk, which may be anything from plaintext to DOCX or (indirectly) PDF, but perhaps the most common output formats are HTML and XHTML.

Whereas the commercially available editor Oxygen ships with TEI stylesheets for transformation to XHTML, PDF, DOCX, and ODT, VS Code does not offer transformation out of the box. You may nevertheless wish to transform to HTML or XHTML as you work in order to ensure your work renders as intended. In VS Code with the `tei-publisher-vscode` extension installed, `Ctrl+Shift+A` will give you a predetermined list of stylesheets to choose from (after retrieving it online); if you select one, it will render a transformation preview in a new pane. Please note that the stylesheets provided by TEI Publisher do not account for many of our elements, so the results will not be as intended e.g. if you encode abbreviations or textual variants.

For the purposes of this class, after encoding our transcriptions in XML we will want to retrieve the text and metadata we have encoded in Python. Because we want to access the metadata as well as the text itself, we will have to rely on an XML library such as `lxml.etree`. If we were only interested in the text itself, we could alternatively use XSLT to transform our transcription to plaintext and read that into our Python interpreter.

If you are interested in learning more about XSLT, the w3schools tutorial could be a place to start. You will also have to obtain an XSLT processor, such as Saxon HE.