APRIL 16, 2019 / #WRITING

We need a new document markup language — here is why

by Christian Neumanns

Introduction: What's the Problem?

There are many document markup languages available already. Wikipedia lists over 70 variations in its List of document markup languages — among them HTML, Markdown, Docbook, Asciidoctor, reStructuredText, etc.

Why, then, does the title of this article suggest we need *yet another one*???

What's the problem?

There are two fundamental problems with the existing document markup languages: Either they are not easy to use, or they are not well suited to write complex documents, such as technical articles, user manuals, or

easy to use, but not suited for complex documents" would be Markdown.

Of course, the above categorization is simplistic. But it should serve as a good starting point to get the gist of this article which aims to delineate the kind of problems that occur in practice. You'll see many representative examples of markup code that illustrates what's wrong, complemented by links to more information.

You'll also discover a *new* markup language. Lots of examples will demonstrate how a new syntax can lead to a language that is "easy to use and suited for complex documents". A *proof-of-concept* implementation is already available. More on this later.

Preliminary Remarks

Please note:

- This article is about document markup languages used to write *text documents*, such as books and articles published on the net. There are other markup languages used to describe specific data, such as mathematical formulas, images, and geographic information, but these are out of scope of this article. However, some ideas presented in this article might be applied to other kinds of markup languages as well.

are also important in the choice of a suitable markup language, such as: support on your OS, ease of installation and dependencies, the tool chain available to create final documents, the quality of documentation, price, customer/user support, etc.

- Readers of this article should have some basic experience with a markup language like HTML, Markdown, Asciidoctor, or similar.
- Readers not aware of the *many* advantages of document markup languages might first want to read:
  Advantages of Document Markup Languages vs WYSIWYG Editors (Word Processors)

Inconveniences / Part 1

Let us first consider some well-known markup languages and have a look at some inconveniences.

HTML

HTML is the language of the web. So, why not write everything in HTML? The reasons to discard this option are well known. Let's quickly recapitulate them.

HTML is cumbersome to write. Nobody wants to write XML code by hand, although editors with HTML/XML

code.

Suppose we want to display a horizontally centered image with a simple black border and a link. The HTML code an inexperienced user would expect to work could look like this:

```
<img src="ball.png" align="center" border="yes" link="ht
```

But the code he or she actually has to write is cumbersome and there are different ways to do it. Here is one way:

```
<div style="text-align: center">    <a href="http://www.
```

HTML lacks “productivity features for writers”, such as:

- Automatic generation of a table of contents, index, glossary, etc.
- Variables used to hold recurring values
- Splitting a document into different files

Other inconveniences will be shown later.

It is easy to learn and use, and well suited for short and simple texts, such as comments in forums, readme files, etc.

However, it suffers from the following problems that make it unsuitable for complex or big documents (e.g. technical articles, user manuals, and books):

- The original Markdown defined by John Gruber lacks many features expected by writers, such as tables (only embedded HTML tables are supported), automatic generation of table of contents, syntax highlighting, file splitting, etc.
- There is no unique, unambiguous specification for Markdown. Many flavors of Markdown exist, with different rules and different features supported. This leads to incompatibility issues when markup code is shared. CommonMark is an attempt to solve this problem. However, the specification is huge and not completed yet (at the time of writing, April 2019, version 0.28, dated 2017–08–01, is the latest one).
- Markdown has similar problems and limitations to those shown later in chapter “Inconveniences / Part 2”. These flaws can quickly become an annoyance when you use Markdown for anything else than short, simple texts.

Markdown's shortcomings:

- Why You Shouldn't Use "Markdown" for Documentation
- Sundown on Markdown?
- Why Markdown Is Not My Favourite Language

Docbook

Docbook is an XML-based markup language that uses semantic tags to describe documents.

It has probably the most complete set of features among all markup languages. It has been used by many authors, is pre-installed on some Linux distributions, and is supported by many organizations and publishers. Docbook has been successfully used to create, publish, and print lots of big documents of all kinds.

But it has the following drawbacks:

It uses XML and a verbose syntax. Look at the following example, borrowed from Wikipedia:

```
<?xml version="1.0" encoding="UTF-8"?><book xml:id="simp
```

Now compare the above code with the following one, written in a modern markup language like Asciidoctor:

```
= Very simple book== Chapter 1Hello world!I hope that yc
```

Docbook is also complex, and therefore hard to learn and use.

Output produced by Docbook, especially HTML, looks old-fashioned (see examples on its website). Of course, presentation can be customized, but this is not an easy task.

LaTeX

LaTeX is a high-quality typesetting system. It is widely used in academia to create scientific documents. It is considered to be the best option for writing PDF documents containing lots of mathematic formulas and equations.

I never used LaTeX myself, because I don't write scientific documents — just articles and books to be published on the web. Therefore, I don't want to comment on it too much. However, it is important to mention it because of its popularity in academia.

LaTeX's unique syntax seems verbose to me, and a bit

```
\documentclass{article}\usepackage{amsmath}\title{\LaTe›

\begin{document}    \maketitle    \LaTeX{} is a document

    % This is a comment    \begin{align}        E_0 &= m
```

The article Conversion from (La)TeX to HTML states that converting LaTeX math to HTML is “a challenge”.

Some markup languages allow LaTeX snippets to be embedded in their markup code, which can be very useful if you need the power of LaTeX for maths. There are other options to display maths on the web, such as Mathjax or MathML (an ISO standard and part of HTML5).

Popular for Big Documents

A impressive number of markup languages have emerged. Many of them use a syntax similar to Markup, and are therefore easy to learn and use. Some have more features than Markdown and are even extensible. However, as soon as we start writing complex documents, corner-cases and

Asciidoctor (an improved version of Asciidoc), and reStructuredText (an improved version of StructuredText). We will have a look at them soon.

Practical Markup Language (PML)

Before moving on to the most interesting part of this article, let me briefly introduce the new markup language I mentioned already in the introduction.

The language is called ***Practical*** *Markup Language (PML)*.

> "fitting the needs of a particular situation in a helpful way; helping to solve a problem or difficulty; effective or suitable"
>
> *— definition of 'practical' in the Cambridge Dictionary*

I started the PML project a few months ago because I couldn't find a markup language that was easy to use *and* well suited for big, complex documents, such as a user manual.

In the next section we'll look at examples of markup code written in PML, compared to code written in other languages. So let's first mention two basic PML syntax rules needed to understand the upcoming examples.

can contain text or child nodes.

For example, here is a node containing text that will be rendered in italics:

```
{i bright}
```

This node starts with `{i` , and ends with `}` . `i` is the tag name. In this case `i` is an abbreviation for `italic` , which means that the node's content will be rendered in *italics*. The content of this node is the text `bright` . The above PML markup code will be rendered as:
bright

Some nodes have attributes, used to specify additional properties of the node (besides its tag name).

For example, the title of a chapter is defined with attribute `title` , as follows:

```
{chapter title=A Nice Surprise    Once upon a time ...}
```

There is not much more to say about the basic concept of

Manual.

You can download and play around with a free implementation of PML. But please note: PML is a *work in progress*. There are missing features, you might encounter bugs, and backwards compatibility is currently not guaranteed.

I use PML myself to write all my web documents, such as this article. For links to more real-life examples please visit the FAQ.

Inconveniences / Part 2

I this section we'll look at examples that reveal *some* problems encountered with markup languages. This is by no means an exhaustive enumeration of all troubles and corner cases. The aim is to just show a few examples that demonstrate the kind of inconveniences and limits encountered in the real world.

For each example the markup code will be shown in HTML, Asciidoctor, reStructuredText, and PML.

If you want to try out some code, you can use the following online testers (no need to install anything on your PC):

- HTML
- Asciidoctor

An online tester for PML is not yet available. You have to install PML on a Windows PC if you want to try it out.

Font Styles

Font styles (*italic*, **bold**, `monospace` , etc.) are often used in all kinds of documents, so good support is essential.

But as we will see, surprises and limits can emerge, as soon as we have to deal with non-trivial cases. Let's look at *some* examples to illustrate this.

Part of a Sentence in Italics

Suppose we want to write:
They called it *Harmonic States*, a good name.

This is a trivial case, and all languages support it.

HTML:

```
They called it <i>Harmonic States</i>, a good name.
```

Asciidoctor:

```
They called it _Harmonic States_ a good name
```

reStructuredText:

```
They called it *Harmonic States*, a good name.
```

PML:

```
They called it {i Harmonic States}, a good name.
```

Part of a Word in Italics

We want to write:

She *un*wrapped the challenge first.

HTML:

```
She <i>un</i>wrapped the challenge first.
```

Asciidoctor:

```
She __un__wrapped the challenge first.
```

She _un_wrapped the challenge first.

reStructuredText:

```
She *un*\wrapped the challenge first.
```

Note that the letter w has to be escaped (preceded by a backslash) for reasons explained here. If the letter is not escaped then a warning is displayed and the result is:
She *un*wrapped the challenge first.

PML:

```
She {i un}wrapped the challenge first.
```

Text in Bold And Italic

We want to write:
They were all ***totally flabbergasted***.

HTML:

```
They were all <b><i>totally flabbergasted</i></b>.
```

```
They were all *_totally flabbergasted_*.
```

reStructuredText:

Combining bold and italic is not supported in reStructuredText, but there are some complicated workarounds.

PML:

```
They were all {b {i totally flabbergasted}}.
```

Real-Life Example

Here is an example inspired by an Asciidoctor user who asked how to display:
_id *optional*.

Let's make the exercise a little bit more interesting by also displaying:
_id **optional**.

HTML:

No surprise here. It just works as expected.

Asciidoctor:

Intuitive attempt:

```
*_id* _optional___id_ *optional*
```

The first line doesn't work, it produces:
***id** _optional*

However, the second line works, which is a bit counterintuitive.

If normal text includes a character that is also used for markup (in our case the `_` preceding `id`), then the character must be escaped. This is a fundamental rule in pretty much all markup languages. For example in HTML a `&` lt; must be escaped wi `th &` lt;. Many languages (including Asciidoctor and PML) use a backslash prefix (e. `g .` \r) to escape. So let's rewrite the code:

```
*\_id* _optional__\_id_ *optional*
```

This doesn't work in Asciidoctor. It produces

_id _optional_

and

_id **optional**

Here is a correct version, as suggested in an answer to the user's question:

```
*pass:[_]id* _optional__pass:[_]id_ *optional*
```

Another answer suggests this solution:

```
*_id* __optional_____id__ *optional*
```

More edge case are documented in chapters Unconstrained formatting edge cases and Escaping unconstrained quotes of the Asciidoctor User Manual.

reStructuredText:

```
**_id** *optional**_id* **optional**
```

There is no problem here, because the character _ is not

However, suppose we wanted to write:
id **optional**.

Here is the code:

```
*\*id\** ***optional***
```

Note that the `*` s in `id` must be escaped, but the `*` s in `optional` don't need to be escaped.

PML:

```
{b _id} {i optional}{i _id} {b optional}
```

Nested Font Styles

Nested font styles of the same kind (e.g. `<i>...<i>...</i>...</i>`) occur rarely in text written by humans, but they could be more or less frequent in auto-generated markup code. If they are not supported then the tool that generates the markup code becomes more complicated to implement, because it must track the font styles that are active already, in order to avoid nesting them.

```
<i>This is <i>excellent</i&gt;, isn't it?</i>
```

No problem, it produces
This is excellent, isn't it?

Asciidoctor:

```
_This is _excellent_, isn't it?_
```

The above code is obviously ambiguous: Are the italics nested or do we want to italicize "This is " and ", isn't it?". When I tested it, the result was neither of it:
This is _excellent, isn't it?_

As far as I now, Asciidoctor doesn't support nested font styles of the same kind.

reStructuredText:

The reStructuredText specification states: "Inline markup cannot be nested." However, no error is displayed if it *is* nested, and the result is unspecified.

RML:

```
{i This is {i excellent}, isn't it?}
```

Font styles of the same kind can be nested in PML. The above code results in:
This is excellent, isn't it?

Nested Chapters

Suppose we are writing an article titled "New Awesome Product" that contains four chapters. The structure looks as follows:

```
New Awesome Product    Introduction    More features
```

Later on we decide to insert chapter "Advantages" as a parent of the three last chapters. The structure now becomes:

```
New Awesome Product    Introduction    Advantages
```

What are the *changes* required in the markup code to pass from version 1 to version 2? Can we simply insert the code for a new chapter? Let's see.

first version:

```
<h1>New Awesome Product</h1>
<h2>Introduction</h2>
<h2>More features</h2>
<h2>Faster</h2>
<h2>Less resources</h2>
```

second version:

```
<h1>New Awesome Product</h1>
<h2>Introduction</h2>
<h2>Advantages</h2>
<h3>More features</h3>
<h3>Faster</h3>
<h3>Less resources</h3>
```

Note: Code *changes* are displayed in bold.

As shown above, besides inserting the new chapter, we have to change the markup for the three child chapters: `h2` must be changed to `h3`.

Asciidoctor:

first version:

```
= New Awesome Product

== Introduction

== More features

== Faster

== Less resources
```

second version:

```
= New Awesome Product

== Introduction

== Advantages

=== More features

=== Faster
```

Again, we have to change the markup for the three child chapters: `==` must be changed to `===`

Note: The blank lines between the chapters are required, otherwise the document is not rendered correctly.

reStructuredText:

first version:

```
********************
New Awesome Product
********************
Introduction
============
More features
=============
Faster
======
Less resources
==============
```

second version:

```
********************
New Awesome Product
********************
Introduction
============
Advantages
==========
More features
-------------
Faster
------
Less resources
--------------
```

The markup for the three child chapters must be changed: All occurrences of `=` must be changed to `-`

PML:

first version:

```
{doc title=New Awesome Product
    {ch title=Introduction}
    {ch title=More features}
    {ch title=Faster}
    {ch title=Less resources}
}
```

second version:

```
{doc title=New Awesome Product
    {ch title=Introduction}
    {ch title=Advantages
        {ch title=More features}
        {ch title=Faster}
        {ch title=Less resources}
    }
}
```

In PML, there is no need to change the code of the three child chapters.

Bottom Line:

In all languages, except PML, the markup code of all child chapters must be adapted if a parent chapter is inserted.

This is not a deal-breaker in case of small articles with few chapters. But image you are writing your next big article or book with lots of chapters and frequent changes. Now, the necessity to manually update child chapters can quickly turn into a daunting, boring, and error-prone task.

Note: Asciidoctor provides a `leveloffset` variable that

unneeded complexity, as can be seen here and here.

A more serious kind of problem can arise in the following situation: Imagine a set of different documents that share some common chapters. To avoid code duplication, the common chapters are stored in different files, and an `insert file` directive is used in the main documents. This works fine as long as the levels of all common chapters are the same in all documents. But troubles emerge if this is not the case.

It is also worth to mention that HTML, Asciidoctor and reStructuredText don't protect us against wrong chapter hierarchies. For example, you don't get a warning or error if a chapter of level 2 contains a direct child chapter of level 4.

In a language like PML, the above problems simply don't exist, because the level is not specified in the markup code. All chapters (of any level) are defined with the same, unique syntax. The chapters' tree structure (i.e. the level of each chapter) is automatically defined by the parser. And wrong hierarchies in the markup code, such as a missing `}` to close a chapter, are flagged by an error message.

Lists

In Asciidoctor the kind of problems we have seen with

chapters. Asciidoctor lists use different markup code to explicitly specify the level of list items (`*` for level 1, `**` for level 2, etc.). Moreover, there are a number of complications you have to be aware of when working with complex list content.

In reStructuredText, nested lists are created using indentation and blank lines. This works fine for simple nested lists, but creates other problems in more complex cases (not discussed here). Using whitespace (e.g. blank lines and indentation) to define structure in markup code is a bad idea, as we'll see soon.

In HTML and PML, the above problems don't exist with lists because the syntax for parent- and child nodes is the same.

Whitespace And Indentation

At first, using whitespace to define structure seams like a good idea. Look at the following example:

```
parent    child 1    child 2
```

The structure is very easy to read *and* write. No noisy special markup characters are needed.

Unfortunately, this works well only for simple structures, and has other inconveniences we'll see soon.

Therefore, a simple, but important rule must be applied in markup languages designed to work well with complex content:

> "Whitespace doesn't change the structure or semantics of the document."
>
> — *whitespace-insignificant-rule*

What does this mean?

First, let us define *whitespace*: Whitespace is any set of one or more consecutive spaces, tabs, new lines, and other Unicode characters that represent space.

In our context, the above rule means that:

Within text, a set of *several* (i.e. more than one) whitespace characters is treated the same as a *single* space character.

For example, this code:

```
a beautiful    flower
```

```
a beautiful flower
```

Between structural elements, a set of whitespace characters is insignificant.

For example, this code:

```
<table>    <tr>
```

... is identical to this one:

```
<table&gt;<tr>
```

A special case of whitespace is *indentation* (leading whitespace at the beginning of a line). The above rule implies that indentation is insignificant too. Indentation doesn't change the result of the final document.

Applying the *whitespace-insignificant* rule is important, because it leads to significant advantages:

- There is no need to learn, apply and worry about

markup specification adds unneeded complexity, and can lead to markup code that is ugly, error-prone, and difficult to maintain, especially in the case of nested lists.

- Whitespace can freely be used by authors to format the markup code in a more understandable, presentable and attractive way (pretty printing). For example, lists (and lists of lists) can be indented to display their structure in a visually clear and maintainable way, without the risk of changing the underlying structure.
- Text blocks can be copy/pasted without the need to adapt whitespace.
- If shared text blocks (stored in different files) are imported into several documents with different structures, there is no risk of changing or breaking the structure.
- There is no unexpected or obscure behavior if the whitespace is not visible for human readers. Some examples:
  - a mixture of whitespace characters, such as spaces and tabs, especially when used to indent code
  - whitespace at the end of a line
  - whitespace in empty lines
  - visually impaired (blind) people who can't read

display-whitespace mode.

- Tools that generate markup code, as well as markup parsers are generally easier to create.
- In some situations it is useful to reduce whitespace to a minimum (e.g. no new lines), in order to save storage space and improve performance.

If you want a few examples demonstrating the kind of technical problems that arise if whitespace is significant, you can read:

- What are the downsides to whitespace indentation rather than requiring curly braces?
- F# syntax: indentation and verbosity
- Issue in nodeca/js-yaml

So, how is whitespace handled in the languages we are discussing in this article?

HTML:

HTML applies the *whitespace-insignificant* rule.

For a thorough explanation, look at this excellent article written by Patrick Brosset: When does white space matter in HTML?.

This can lead to surprising behavior and problems with no easy or no satisfying solution. Some examples can be seen here and here.

reStructuredText:

reStructuredText has whitespace rules that are 'a bit surprising'.

For example, writing `*very*` results in *very* (text in italics, as expected). However, `* very*` results in * very* (no italics!), because of the whitespace preceding "very". To understand why, the answer might be found in chapter Whitespace of the specification.

PML:

Similar to HTML, PML applies the *whitespace-insignificant* rule.

There is one exception: For practical reasons, a blank line between two text blocks results in a paragraph break. This means that instead of writing:

```
{p text of paragraph 1}{p text of paragraph 2}
```

```
text of paragraph 1text of paragraph 2
```

Note: Sometimes, whitespace *is* significant in text. For example whitespace must be preserved in source code examples. Or, in verbatim text, several consecutive spaces or new lines must be preserved in the final document. All languages support this. However, in reStructuredText it's not always obvious how to it, as shown here.

Other Inconveniences

As seen already, some markup languages systematically use opening and closing tags. An example would be < ;i> ; an d </i> in HTML. All XML-based languages, as well as PML belong to this class of languages.

Without digging into details, here are some drawbacks that can occur in languages that do *not* (or not always) use pairs of distinct opening/closing tags (e.g. Markdown, Asciidoctor, and reStructuredText):

Editor support

Creating good, reliable editor support is more difficult to develop. Examples of useful editor features are:

- syntax highlighting for markup code

- visualizing pairs of block start/end marks (e.g. { and its corresponding })
- block collapsing/expanding
  In the case of languages that use distinct opening/closing tags, the two last features work out-of-the-box in some editors. For example, PML uses { and } for node boundaries. This is also used in many programming languages (C, Java, Javascript, etc.) and therefore block features implemented for programming languages will also work for PML.

Document validation

Fewer syntax and structure errors can be detected automatically. This can lead to more time spent on debugging documents. Or, even worse, there might be silently ignored errors that end up in wrong output (Did I really fail to spot the missing warning block on page 267 of my 310 pages book?).

Parsers

It is more difficult to create parsers (i.e. programs that read markup code) that work well in all cases. If different parsers read the same markup code, there is an increased risk of getting different results for corner-cases.

Auto-generated markup code

significant, or font styles cannot be nested, then additional state must be updated and tracked, in order to respect these rules.

My Own Experience

When I started writing technical documents a few years ago, I used Docbook. It took me some time to learn it, but after that I never stumbled on anything I couldn't do. Docbook is powerful. However, I disliked typing verbose XML code. I tried some XML editors, but gave up. Finally I just wrote complete text blocks unformatted in Notepad++, and then adorned the text with the necessary markup code. It was frustrating and time-consuming. Moreover, I couldn't find a stylesheet that produced good-looking web documents, and I didn't have the patience, motivation, and experience to fiddle around with big, complex CSS files and adapt them.

Later on I discovered Asciidoctor. What a relief. Everything was so much simpler and the web documents were beautiful, out of the box. Asciidoctor's documentation is great, and I think the community is helpful and active. However, when I started to write more complex and bigger documents, I had to deal with problems similar to those described in the previous sections. At one point, I had to develop a specific pre- and post-processor to solve a problem for which I couldn't find a solution in Asciidoctor/Gitbook.

not exist in Docbook.".

To make a long story short, I concluded that we need a new markup syntax. The key points to success would be:

- easy to learn: few, simple, consistent and predictable rules (no exceptions)
- easy to write and read
- well-structured documents with no ambiguities
- powerful enough to create big, complex documents without the need for "special rules, tricks, or workarounds"

After a period of investigating, pondering, programming, testing and improving, the Practical Markup Language (PML) was born. Since then, I never looked back again. Today I write all my web documents in PML (including this article).

Of course, when I started to create PML, it was to cover my own needs. So, I am probably biased. Hopefully this article contains enough factual examples, but I encourage you to leave a comment if you see anything wrong, unfair, or missing. I appreciate constructive feedback of any kind, and I will update the article if needed.

Conclusion

languages vanish with the PML syntax.

Now we should come together to improve PML and make it more powerful, so that it covers more use cases and more people can benefit from it.

Please help to spread the word. Or try out PML and send feedback, so that we know what needs to be refined. Your voice counts!

The vision is to create the best possible document markup language and all necessary tools, so that writers can focus on writing and enjoy creating beautiful documents in a minimum of time — without worrying about unneeded complexity.
