Pillar: A Versatile and Extensible Lightweight Markup Language

Thibault Arloing, Yann Dubois, Damien Cassou, Stéphane Ducasse

Accepted to IWST'16

Pillar: A Versatile and Extensible Lightweight Markup Language

Thibault Arloing Yann Dubois Damien Cassou Stéphane Ducasse

thibault.arloing@etudiant.univ-lille1.fr yann1.dubois@etudiant.univ-lille1.fr damien.cassou@inria.fr stephane.ducasse@inria.fr

RMod team, Inria Lille Nord Europe, University of Lille, CRIStAL, UMR 9189, 59650 Villeneuve d'Ascq, France

Abstract

There is a plethora of languages to write documentation and documents. From extremely powerful and complex such as LaTeX to extremely simple such as Markdown. In this technical article we present Pillar a versatile and extensible lightweight markup language. Pillar's document model and open architecture support exporting from Pillar to various formats such as ASCIIDoc, HTML, LaTeX and Markdown. Pillar is currently used to write books, documentation, websites and slide decks (through Beamer and DeckJS). Pillar specially shines when advanced features are needed such as multiple exports (e.g., a printed book and web pages), internal references (e.g., links to figures with captions) and content generation (e.g., to give an up-to-date code size of a documented software).

Keywords Pillar, Markup, Markdown, ASCIIDoc, LaTeX, Documents, Open-Architecture, Pharo

1. Introduction

There are many different markup languages to describe documents. LaTeX(Goossens et al. 2007) is extremely powerful and can even be extended directly inside LaTeX documents. One downside of this power is that LaTeX takes time to learn: we have seen much more contributions from the community to our technical books when we switched their language from LaTeX to Pillar. Another downside of LaTeX's power is that it is difficult for tools to generate something else (e.g., HTML) than what is natively supported by LaTeX engines. Simpler alternatives to LaTeX emerged over the years such as Markdown.[1] Markdown is a lightweight markup language with many implementations in many tools. This strength is also a weakness because each implementation has features not available in other implementations or available with a different syntax.

In this technical article we present Pillar, an extensible and versatile lightweight markup language.[2] Pillar history started in 2000 with SmallWiki, a predecessor of the Pier Content Management System.[3] Pillar has a lightweight markup syntax similar to the one of Markdown. Beyond what is available in Mardown, Pillar has advanced features such as:

Figure 1. Main syntax of Pillar

[1] https://daringfireball.net/projects/markdown

[2] http://www.smalltalkhub.com/#!/~Pier/Pillar

[3] http://piercms.com/

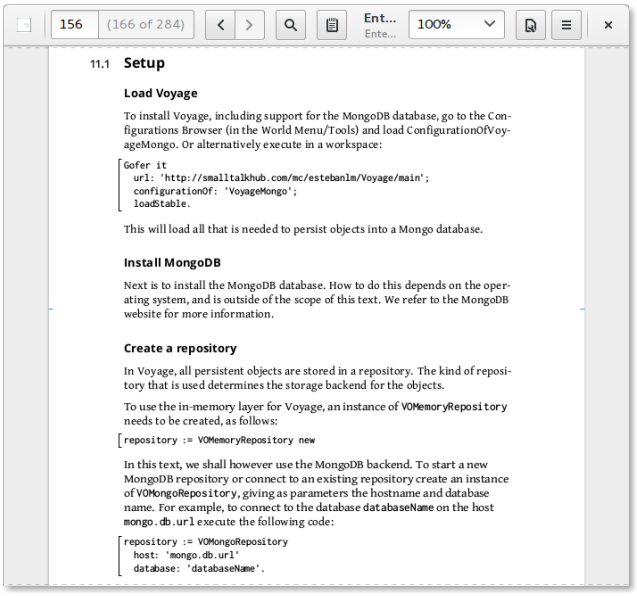

Figure 2. Example of Book Generated with Pillar

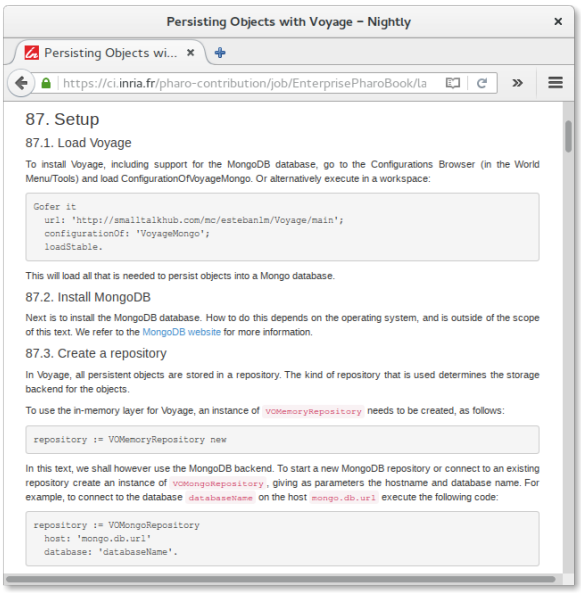

Figure 3. Example of HTML Generated with Pillar

- many export formats (*e.g.*,, HTML, Markdown, LaTeX, AsciiDoc, Beamer) (see Figure 2 and 3 for example exports);
- configuration of the export (*e.g.*,, titles can be automatically numbered with Roman numerals);
- templates with the Mustache template engine;
- syntax-highlighting of code blocks;
- lightweight markup syntax (see Figure 1) with support for tables, figures and scripts with labels and captions, internal references to sections, figures and scripts, and content generation;
- configurable automatic capitalization and numbering of titles;
- text editor plugins (Emacs, Vim, TextMate, and Atom).

Pillar is designed to be easily extensible by a Pharo developer. This is another reason why Pillar is used to write books (Cassou et al. 2015), booklets,[4] technical documentation, websites[5] and lecture presentations.[6]

2. Extensibility Mechanisms

Contrary to LaTeXwhere extensions can be written in the same language as the document, extending Pillar requires writing Pharo code. We wanted to make Pillar extensible to handle several use cases (*e.g.*,, book writing and slide-based teaching) without overwhelming the users with complex syntax.

In this section, we first discuss environments, *i.e.*, how to identify a portion of the document to handle differently (*e.g.*, code blocks are not parsed). Then we present how annotations were introduced to permit the addition of new constructs (*e.g.*,, footnotes) without adding new syntax. We cover implementation of such extensions in the next section.

2.1 Supporting Environments

Some portions of a document must be handled differently from the rest. Code blocks, for example, must be analyzed by the Pillar parser in a different way than paragraphs. There is thus a need to delimit regions of text: Pillar uses the syntactic constructs `[[[` and `]]]` as in:

```
[[[language=smalltalk|caption="hello world"
   Transcript show: 'hello World'
]]]
```

An environment can have arguments (specified after `[[[` and separated by pipes `|`) as is exemplified here with `language` and `caption`.

Environments can be used beyond code blocks. Often repeating structures emerge in documents. For example, every Wikipedia article about a city has a table summarizing the city characteristics (*e.g.*,, coordinates, country and population). Such structures, called Infoboxes[7] in Wikipedia, have the same visual appearance to facilitate identification. Structures separate data from appearance to allow semantic analyses and other manipulations.

[4] `http://books.pharo.org`

[5] `http://guillep.github.io/ecstatic/`

[6] `http://mooc.pharo.org/`

[7] `https://en.wikipedia.org/wiki/Help:Infobox`

Such structures are often expressed with a JSON-like[8] language as follows:

```
{
   "name"        : "Bordeaux",
   "latitude"    : 44.84,
   "longitude"   : -0.58,
   "country"     : "France",
   "population"  : 241287
}
```

Letting users write JSON directly between two paragraphs is not desired as this would require adding new syntax to the parser which might break existing documents and add more restricted characters users must protect. We needed a way to have a JSON structure in a Pillar file without interpreting the JSON as Pillar syntax. To do this, we used environments whose contents is JSON:

```
[[[structure=city
{
   "name"        : "Bordeaux",
   "latitude"    : 44.84, [...]
}
]]]
```

Environments are the only Pillar elements which keep their contents unparsed. This solution encapsulates new syntax within an existing one thus avoiding problems with existing documents.

2.2 Annotations and Arguments

Beyond delimiting regions of text between paragraphs, it is sometimes useful to delimit regions of text *inside* a paragraph. This is for example useful for footnotes and citations. The cornerstone of this extension mechanism of Pillar is the support for annotations whose syntax is the following:

```
${myAnnotation|arg1=true|arg2=Hello world}$
```

An annotation is specified between `${` and `}$`. After the opening brace, a keyword identifies the annotation type (here `myAnnotation`). An annotation may take arguments which are specified the same way as for environments. Using the same syntax to mean the same thing over and over again gives an impression of coherence to the user and helps memorization. The syntax is the same for all element declarations with arguments such as figures:

```
+A Legend>file://fig.png|width=50|label=myFig+
```

Each annotation has a particular behavior that is defined by a plugin within the Pillar architecture. For example, citing a research work is done through `${cite:CitationReference}$` and the class `PRCitationAnnotation` while showing a footnote is done through the annotation `${footnote:Some text}$` and `PRFootnoteAnnotation`.

[8] `http://json.org/`

3. Open Architecture

Pillar open-architecture is based on a document model with many visitors forming a pipeline. After discussing the Pillar document model with its AST, visitors and tests (see Section 3.1), we discuss the implementation of structures (see Section 3.2). To facilitate extension, Pillar extensions are covered by a test framework which makes it easy to check if a new extension follows the extension protocol. All in all, unit tests cover 90% of the Pillar source code.

3.1 The Pillar Document Model

Care has been put to ease Pillar extension. There are a lot of helpers which makes sure adding a new feature is easy.

After parsing the Pillar file, the document is represented by an AST (instance of `PRDocument`). 52 classes (subclasses of `PRDocumentItem`) are responsible for modeling every piece of a Pillar document (*e.g.,*, bold text, links, figures, list items, table rows and cells). The generated AST can then be visited by several visitors (instances of `PRVisitor`'s 59 subclasses).

3.1.1 AST

The AST can be edited and enhanced with information such as properties for each node it contains. These properties can be easily exploited by visitors. For example, code blocks can have properties such as `language` (for syntax highlighting), `caption` (for a legend) and `hideable` (to hide some scripts when students should not be shown solutions exercises).

3.1.2 Visitors

A document writer (subclass of `PRDocumentWriter`) is responsible for exporting an AST in a dedicated output format. Document writers are implemented as visitors. Here is how, for example, bold is handled in both HTML and LaTeX exports:

```
PRHTMLWriter>>visitBoldFormat: aFormat
  canvas tag
    name: 'strong';
    with: [ super visitBoldFormat: aFormat ]

PRLaTeXWriter>>visitBoldFormat: aFormat
  canvas command
    name: 'textbf';
    parameter: [ super visitBoldFormat: aFormat ]
```

For a Pillar input of `""Pharo""`, the HTML writer would write `<strong>Pharo</strong>` whereas the

LaTeX writer would write `\textbf{Pharo}`. Implementing a new document writer is then easy and can be done step by step while being guided by the Pillar test framework.

Visitors are used for many other tasks beyond export. For example, visitors are also used to transform ASTs. Each such visitor is then responsible for a simple transformation. For example, `PRRemoveHideableScripts` is responsible for removing some code blocks from the AST (so that teachers can have solutions in their document but not students):

```
PRRemoveHideableScripts>>visitScript: aScript
  aScript isHideable
    ifTrue: [ self replace: #() ]
```

When a script is hideable, this visitor will replace its node by nothing, thus removing it from the AST.

Parsing, transforming and exporting form a pipeline of phases. Each phase has a small responsibility. Phases are ordered according to a notion of priority and category (among `init`, `parse`, `check`, `transform` and `export`).

About Visitors. The visitor design pattern (Alpert et al. 1998) is particularly suited when the model does not change often as each change in the model requires updating every visitor. This seems to contradict the extensibility philosophy behind Pillar. Nevertheless, this is not a problem in practice for two main reasons. The first reason is that Pillar is organized around a huge set of very specific visitors which are not affected by most changes to the model. For example, the class `PRRemoveHideableScripts` only contains a 2-line method (see above). Second, using inheritance of visitors and AST nodes, when an element is not explicitly handled by a visitor, a default implementation is often acceptable. For example, if a document writer does not implement `visitBoldFormat:`, the text will still be rendered, ignoring the bold markup.

3.1.3 Tests

Automated tests guide the extension of Pillar by validating the visitors behavior. In the context of document writers, unit tests automatically verify the generation of documents. For example, testing that both the HTML and LaTeX writers correctly export bold texts is done with the implementation of these two methods only:

```
PRHTMLWriterTest>>boldFoo
    ^ '<strong>foo</strong>'

PRLaTeXWriterTest>>boldFoo
    ^ '\textbf{foo}'
```

As a result of the Pillar test framework, testing a new document writer is just a matter of provided example outputs: the test framework takes care of instantiating the document writer, creating the example ASTs, executing the necessary parts of the pipeline, and comparing the result with the provided example output.

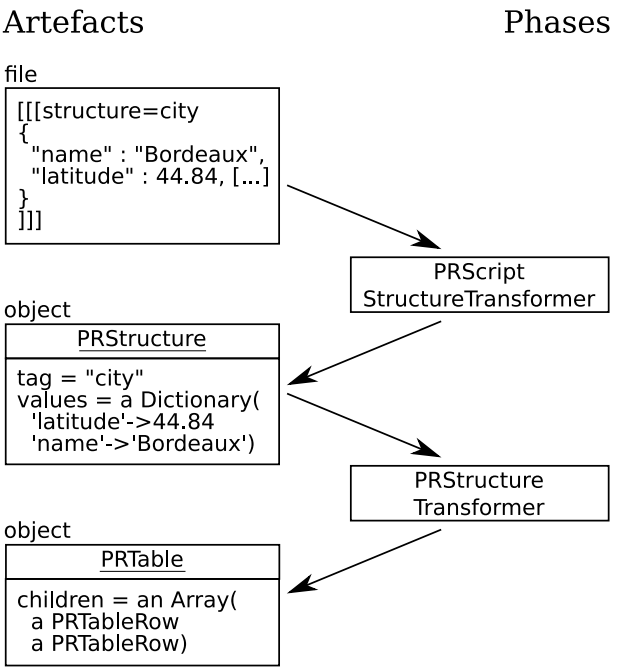

Figure 4. Pipeline transforming a city structure into a table through an instance of the class PRStructure.

3.2 Open Architecture at Work: Structures

In Section 2.1 we introduced the notion of environments and explained how we used them to design the new structure construct. We designed structures so that new ones can easily be used without requiring new Pharo code. Figure 4 represents the pipeline to transform a city structure in a Pillar document into a table. The `PRScriptStructureTransformer` is a visitor which takes care of replacing each structure environment by an instance of `PRStructure`. In essence, this visitor is implemented like this:

```
PRScriptStructureTransformer>>visitScript: aScript
  (aScript parameters includesKey: 'structure')
    ifTrue: [
      self replace: {
        PRStructure new
          tag: aScript structureName;
          values: (Json readFrom: aScript text);
          yourself } ]
```

When a script has a parameter named `structure`, this visitor replaces it by an instance of `PRStructure`.

The class `PRStructureTransformer` is responsible for transforming a structure into an appropriate representation through a structure renderer. There are currently

two renderers (subclasses of `PRStructureRenderer`): one which represents a structure as a table and one as a definition list. Here is the essence of the table renderer implementation:

```
PRTableRenderer>>renderFor: aStructure
  | table |
  table := PRTable new.
  aStructure keys do: [ :each |
    table add: (
      PRTableRow new
        add: (PRTableCell with: (
          PRText content: each));
        add: (PRTableCell with: (
          PRText content: (aStructure at: each)));
        yourself) ].
  ^ table
```

Adding a new renderer is just a matter of creating a new class and implementing a the method `renderFor:`. The decision to use a particular renderer for a particular kind of structure is currently hard-coded but is supposed to be done through the Pillar configuration file.

4. Conclusion

In this paper we presented Pillar and its capacity to be easily extended by a Pharo developer. This paper rapidly compares Pillar with other markup languages and shows that Pillar is a good compromise between power and simplicity.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020.

References

Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison Wesley, Boston, MA, USA, 1998. ISBN 0-201-18462-1.

Damien Cassou, Stéphane Ducasse, Luc Fabresse, Johan Fabry, and Sven Van Caekenberghe. *Enterprise Pharo: a Web Perspective*. Square Bracket Associates, 2015.

Michael Goossens, Frank Mittelbach, and Alexander Samarin. *The Latex Companion*. Addison Wesley, 1994. ISBN 0-201-54199-8. ordered but not received.

Michel Goossens, Sebastian Rahtz, and Frank Mittelbach. *The LaTeX Graphics Companion: Illustrating Documents with TeX and Postscript(R) (Tools and Techniques for Computer Typesetting)*. Addison-Wesley, 2007. ISBN 0321508920 978-0321508928. ordered but not received.

JSON. Json (javascript object notation). `http://www.json.org`. URL `http://www.json.org`.