Publishing Documents with Pillar 7.0

Stéphane Ducasse and Guillermo Polito

April 29, 2020

Layout and typography based on the `sbabook` $\LaTeX$ class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

About Pillar

Pillar is a system to manage documents (books, presentations, and web sites). From a common format, it is able to generate documents in multiple formats (html, markdown, latex, AsciiDoc). It is composed of several modules such as importers, transformers, document model and outputers.

This book describes Pillar in its current version 7.0. Pillar is currently developed and maintained by Stéphane Ducasse and Guillermo Polito. The original author of Pillar was Damien Cassou. Many people have also contributed to Pillar: Ben Coman, Guillermo Polito, Lukas Renggli (original author of the PierCMS from which a first version of Pillar has been extracted), Benjamin van Ryseghem, Cyril Ferlicot-Delbecque, Thibault Arloing, Yann Dubois, Quentin Ducasse and Asbathou Sama Biyalou. Special thanks to Asbathou Sama Biyalou!

This book adapts, extends, and clarifies the chapter explaining Pillar in the *Enterprise Pharo: a Web Perspective* book.

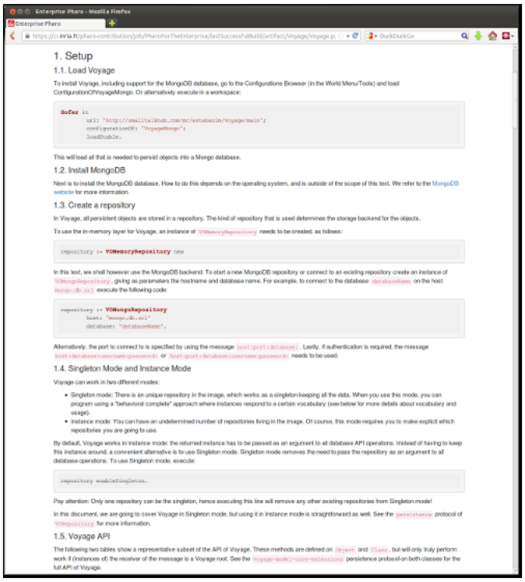
Pillar was sponsored by ESUG[1].

1.1 Introduction

Pillar (hosted at http://github.com/pillar-markup) is a markup language and associated tools to write and generate documentation, books (such as this one), web sites, and slide-based presentations. The Pillar screenshot in Figure 1-1 shows the HTML version of chapter Voyage.

Pillar has many features, helpful tools, and documentation:

[1] http://www.esug.org


Enterprise Pharo - Mozilla Firefox
Enterprise Pharo
https://ci.inria.fr/pharo-contribution/job/PharoForTheEnterprise/lastSuccessfulBuild/artifact/Voyage/Voyage.pi
DuckDuckGo

1. Setup

1.1. Load Voyage

To install Voyage, including support for the MongoDB database, go to the Configurations Browser (in the World Menu/Tools) and load ConfigurationOfVoyageMongo. Or alternatively execute in a workspace:

```
Gofer it
        url: 'http://smalltalkhub.com/mc/estebanlm/Voyage/main';
        configurationOf: 'VoyageMongo';
        loadStable.
```

This will load all that is needed to persist objects into a Mongo database.

1.2. Install MongoDB

Next is to install the MongoDB database. How to do this depends on the operating system, and is outside of the scope of this text. We refer to the MongoDB website for more information.

1.3. Create a repository

In Voyage, all persistent objects are stored in a repository. The kind of repository that is used determines the storage backend for the objects.

To use the in-memory layer for Voyage, an instance of `VOMemoryRepository` needs to be created, as follows:

```
repository := VOMemoryRepository new
```

In this text, we shall however use the MongoDB backend. To start a new MongoDB repository or connect to an existing repository create an instance of `VOMongoRepository`, giving as parameters the hostname and database name. For example, to connect to the database `databaseName` on the host `mongo.db.url` execute the following code:

```
repository := VOMongoRepository
        host: 'mongo.db.url'
        database: 'databaseName'.
```

Alternatively, the port to connect to is specified by using the message `host:port:database:`. Lastly, if authentication is required, the message `host:database:username:password:` or `host:port:database:username:password:` needs to be used.

1.4. Singleton Mode and Instance Mode

Voyage can work in two different modes:

- Singleton mode: There is an unique repository in the image, which works as a singleton keeping all the data. When you use this mode, you can program using a "behavioral complete" approach where instances respond to a certain vocabulary (see below for more details about vocabulary and usage).
- Instance mode: You can have an undetermined number of repositories living in the image. Of course, this mode requires you to make explicit which repositories you are going to use.

By default, Voyage works in instance mode: the returned instance has to be passed as an argument to all database API operations. Instead of having to keep this instance around, a convenient alternative is to use Singleton mode. Singleton mode removes the need to pass the repository as an argument to all database operations. To use Singleton mode, execute:

```
repository enableSingleton.
```

Pay attention: Only one repository can be the singleton, hence executing this line will remove any other existing repositories from Singleton mode!

In this document, we are going to cover Voyage in Singleton mode, but using it in Instance mode is straightforward as well. See the `persistence` protocol of `VORepository` for more information.

1.5. Voyage API

The following two tables show a representative subset of the API of Voyage. These methods are defined on `Object` and `Class`, but will only truly perform work if (instances of) the receiver of the message is a Voyage root. See the `voyage-model-core-extensions` persistence protocol on both classes for the full API of Voyage.

Figure 1-1 An example Pillar output

- simple markup syntax with references, tables, pictures, captions, syntax-highlighted code blocks;
- export documents to HTML, LaTeX, Markdown, AsciiDoc, ePuB and Pillar itself, and presentations to Beamer and Deck.js;
- many tests with good coverage (94% with more than a 2100 executed tests), which are regularly run by a continuous integration job[2]
- a command-line interface and dedicated plugins for several text editors: Emacs[3], Vim[4], TextMate[5], and Atom[6]
- a cheat sheet (see Chapter 5).

1.2 Pillar users

This book was written in Pillar itself. If you want to see how Pillar is used, have a look at its source code (http://github.com/SquareBracketAssociates/Booklet-PublishingAPillarBooklet), or check the following other real-world projects:

[2]https://ci.inria.fr/pharo-contribution/job/Pillar
[3]https://github.com/pillar-markup/pillar-mode
[4]https://github.com/cdlm/vim-pillar
[5]https://github.com/Uko/Pillar.tmbundle
[6]https://github.com/Uko/language-pillar

- the Updated Pharo by Example book (https://github.com/SquareBracketAssociates/UpdatedPharoByExample),
- the Pharo MOOC - Massive open online course (https://github.com/SquareBracketAssociates/PharoMooc,
- Any of the Pharo booklets (https://github.com/SquareBracketAssociates/Booklet-XXXX,
- the PillarHub open-access shared blog (http://pillarhub.pharocloud.com).

1.3 Pillar future features

Pillar 70 saw some major refactorings and cleaning: it does not rely on Grease and Magritte anymore. Its architecture is a lot cleaner.

Still some issues are missing. Here is a little list of features that we are working on or will soon:

- Incremental recompilation. Since we remove the use of make (so that Windows users can use Pillar) we should introduce a way to avoid to recompile complete book when just one chapter changed.
- Markdown syntax.
- Release of Ecstatic. Pillar supports the deployment of web sites named Ecstatic and we are working on a second version of Ecstatic.
- Better table support.

1.4 Conclusion

Pillar is still in active development: maintainers keep improving its implementation. The current version of Pillar is Pillar 70. This booklet only documents Pillar 70. This booklet will be synchronised with future enhancements.

CHAPTER 2

Getting Started

In this section we give the basic steps to get you started with your first Pillar project. We show first how to install Pillar in different platforms and how to test your installation. This chapter covers also the installation of LaTeX, which is required to generate pdf documents. Installing LaTeX is not needed if you use the Travis build integration to automatically generate your documents once committed.

Once Pillar is installed, we will proceed to create a first Pillar project. A Pillar project is a directory that will contain several `.pillar` files with your content, a `pillar.conf` configuration file and a `_suppert` directory containing several template-related files which will manage the style of your document.

2.1 Installing the zip

On https://github.com/pillar-markup/pillar/releases, Pillar comes up on all platforms as a zipped distribution.

This is the easiest way to get started.

Installing it in your System

You can then proceed to install that pillar build where you want. For example, you can place it in a hidden directory in your home directory:

```
# move the pillar directory to your HOME
$ mv pillar ~/.pillar
```

Then add that directory to the PILLAR_HOME and HOME environment variables, for example, by modifying your .bashrc (or .zshrc) with:

```
export PILLAR_HOME="$HOME/.pillar/build"
export PATH="$PATH:$PILLAR_HOME"
```

Now you are ready to use Pillar.

2.2 A First Pillar Project

A Pillar project is a directory containing several `.pillar` files with your content, a `pillar.conf` configuration file and a `_support` directory containing several template-related files which will manage the look-and-feel of your document. The complete project structure is setup by Pillar itself, you do not need to do it manually.

Setup the Project

Let's then procceed to create a new empty directory for our project and tell Pillar that we want to use the book archetype on it.

```
$ mkdir my-pillar-project
$ cd my-pillar-project
$ pillar archetype book
```

The `archetype` command sets up the project structure for you: it installs a configuration file *pillar.conf* prod with default values, it creates several Pillar files with some sample content, and it imports the basic styling files inside *_support*.

Build your Project

At this point, you can already generate your book using the command `pillar build [output_format]` followed by the desired output. For example, to generate your project in html:

```
$ pillar build html
```

Or in pdf using your system's LaTeX installation:

```
$ pillar build pdf
```

Inspect your Project's Output

Once `build` has finished, the build results are written into the `_result` directory. The `_result` directory contains the output in different subdirectories depending on the used format, and is structured as follows:

```
your_project\
  pillar.conf
  file1.pillar
  file2.pillar
  _results\
    format1\
      file1.format1
      file2.format1
    format2\
      file1.format2
      file2.format2
```

For example, if you generated your files in html and pdf you will see

```
your_project\
  pillar.conf
  file1.pillar
  file2.pillar
  _results\
    html\
      file1.html
      file2.html
    pdf\
      file1.pdf
      file2.pdf
```

Building a Single File

Pillar generates all `.pillar` files it finds in your project by default. We can instruct it otherwise by specifying the files to build as argument. For example, if you want to build only `index.pillar` to html you can use the `pillar build` command as follows:

```
$ pillar build html index.pillar
```

This will generate only the output related to the `index.pillar` file.

2.3 Versionning your Project

In this section we explain how to share your project in a source code repository such as Git. This section guides you through the setup of a Git repository, the selection of the correct files to commit, and the creation of a `.gitignore` file to avoid auto-generated files.

Setting up a Git Repository

There are two main ways to setup a Git repository:

- we can clone an existing repository and add our files there, or
- we can create a new repository and push it to an existing remote

This first option is the more practical in case you previously created a Git repository in GitHub. If that is the case, or if you prefer to start by creating a repository on GitHub or your favorite Git repository hosting, then this step consists on cloning that existing repository in a new directory and adding your files to it.

```
$ git clone git@github.com:[your_username]/[your_repository].git
$ cp path_to_your_project
```

```
cd your_repository
```

Versioning your Book

Now to version simply you book you should version all the files except `_result`. Later we will show you how to add travis and bintray support for automatic compilation so that you do not need to install locally LaTeX on your machine.

```
$ git add pillar.conf
$ git add _support
$ git add Chapters
$ git add book.pillar
```

Adding a .gitignore

Not all the downloaded files are worth versionning. We suggest the following .gitignore configuration.

```
more .gitignore
/_result/
```

You can remove the folder figures and the chapter samples. Now you are ready to commit your files.

2.4 Building Pillar from Sources

This section explains how to build and install Pillar both in Unix based systems (like OSX and Linux) and Windows. The first part of this section covers how to download an already built version of pillar or how to build one yourself.

Build Requirements

Building Pillar from sources requires that you have some dependencies installed in your system:

- **Git**: a Git installation is required to get the Pillar repository from `git@github.com:pill` `markup/pillar.git`.

- **wget**: is required to do several file downloads. For OSX users, you might need to install `wget` via `brew` following for example the following instructions:

```
# The following lines will install both Homebrew and wget, ignore
    the first one if Homebrew is already installed
$ ruby -e "$(curl -fsSL
    https://raw.githubusercontent.com/Homebrew/install/master/install)"
$ brew install wget --with-libressl
```

1. @@note Windows is not able to deal with `.sh` and makefile scripts natively, so you have to install tools providing shell-like capabilities #such as https://www.cygwin.com/. If you decide to install cygwin you should be able to follow the Mac OS X and Linux installation section.

Downloading and Building Pillar

You can build Pillar from sources on Unix-like systems such as OSX and Linux. This approach is also valid using msys and cygwin like terminals on Windows. To get Pillar source code, we can use a `git clone` command as follows:

```
$ git clone git@github.com:pillar-markup/pillar.git
```

Once we got the repository, we should execute the `build.sh` script found in the *scripts* directory:

```
$ cd pillar
$ ./scripts/build.sh
```

Installing it in your System

You can then proceed to install that pillar build where you want. For example, you can place it in a hidden directory in your home directory:

```
# Go back to the previous directory and
# move the pillar directory to your HOME
$ cd ..
$ mv pillar ~/.pillar
```

Then add that directory to the PILLAR_HOME and HOME environment variables, for example, by modifying your .bashrc (or .zshrc) with:

```
export PILLAR_HOME="$HOME/.pillar/build"
export PATH="$PATH:$PILLAR_HOME"
```

Test your Installation

To test your pillar installation, open a new terminal and execute the pillar `--version command`. If everything is ok, that should print out (as in the current version) the version of the Pharo VM. For example:

```
$ pillar --version
M:    CoInterpreter VMMaker.oscog-eem.2380 uuid: c76d...
```

2.5 Getting Pillar Dependencies

If you're going to build pdf documents using Pillar, then you will required to install LaTeX also. We recommend that you install a complete TeX Live installation, like that different LaTeX templates can reuse existing modules. You can get a full installation from https://tug.org/texlive/ or by using your favorite package manager on Linux. For example using apt-get you can do:

```
sudo apt-get install texlive-full
```

Also, a manual LaTeX installation can be done by downloading TeX live's default distribution as a tarball and installing several packages on top. This installation procedure is already distributed with Pillar as it is used by our continuous integration jobs. You can find our installation script in the `scripts/ci/ensureLatex.sh` file that you can find in:

```
scripts/ci/ensureLatex.sh
```

```
  #!/bin/bash
  # From
    https://github.com/y-yu/install-tex-travis/blob/master/install-tex.sh

  #Enable to not exit on error
  #set -o errexit

  #Enable to trace bash execution
  #set -o xtrace

  DIRNAME=tl-`date +%Y_%m_%d_%H_%M_%S`

  echo "make the install directory: $DIRNAME"
  mkdir $DIRNAME
  cd $DIRNAME

  wget
    http://mirror.ctan.org/systems/texlive/tlnet/install-tl-unx.tar.gz
  tar zxvf install-tl-unx.tar.gz
  cd install-tl-*

  BASE_PROFILE=$(cat << EOS
  selected_scheme scheme-small
  TEXDIR $HOME/texlive/2017
  TEXMFCONFIG $HOME/.texlive2017/texmf-config
  TEXMFHOME $HOME/texmf
  TEXMFLOCAL $HOME/texlive/texmf-local
```

```
TEXMFSYSCONFIG $HOME/texlive/2017/texmf-config
TEXMFSYSVAR $HOME/texlive/2017/texmf-var
TEXMFVAR $HOME/.texlive2017/texmf-var
option_doc 0
option_src 0
EOS
)

if [[ $TRAVIS_OS_NAME == 'osx' ]]; then
  echo "$BASE_PROFILE\nbinary_x86_64-darwin 1" > ./small.profile
  export PATH=$PATH:$HOME/texlive/2017/bin/x86_64-darwin
else
  echo "$BASE_PROFILE\nbinary_x86_64-linux 1" > ./small.profile
  export PATH=$PATH:$HOME/texlive/2017/bin/x86_64-linux
fi

./install-tl -profile ./small.profile
tlmgr init-usertree
tlmgr install latexmk
tlmgr install luatex85

cd ../..

echo "remove the install directory"
rm -rf $DIRNAME
```

Now you are ready to write your document in Pillar and get them exported in PDF and HTML.

Once the basic installation is installed you can procceed to install all your required dependencies using TeXLives `tlmgr` package manager. You'll find an up-to-date bash script that performs the installation of the minimal required packages in:

```
scripts/ci/ensure\_book\_dependencies.sh
```

This script installs packages to manage fonts and better LaTeX rendering such as fira, gentium-tug, footmisc, tcolorbox, environ, trimspaces, import, multirow and ifetex.

CHAPTER 3

Pillar core syntactic elements

In this chapter, we present the Pillar syntax. Have a look at the Chapter 5: it contains a two pages summary. We use the current text as example, notice however, that some results may be different on other support like HTML because of different common practices (for example creating an anchor with the reference instead of a footnote for an external link).

It is important to see that since Pillar is extensible, plugins may extend the default markup. The extension mechanisms allow one not to learn new syntax but just new tags and parameters as will present later.

3.1 Headings: Chapters & sections

A line starting with `!` represents a heading. Use multiple `!` to create sections and subsections.

```
!!!Headings: Chapters & Sections
```

is the way the current section is created. The exporter and template may interpret the section the way they want (.i.e., for example outputing `subsection` instead of `section`).

Some exporters support the possibility to specify a level of interpretation for `!`. `headingLevelOffset` controls how to convert from the level of a Pillar heading to the level of heading in your exported document. For example, a `headingLevelOffset` of 3 converts a 1st level Pillar heading to an `<h4>` in HTML.

```
"headingLevelOffset" : 3
```

This feature does not work in Pillar 7.0 and it is planned to reintroduce it.

3.2 References and links

Anchor definition

To refer to a section or chapter, put an anchor (equivalent to `\label{chapterAndSections}` in LaTeX) using the `@chapterAndSections` syntax on a *separate line*.

```
@chapterAndSections
```

Anchor reference

When you want to refer to an anchor (equivalent to `\ref{chapterAndSections}` in LaTeX), use the `*@chapterAndSections*` syntax. Anchors are invisible and links will be rendered as: 3.1. If you get a Fig. without a number this is probably that you forgot the `@` in the reference.

```
*@chapterAndSections*
```

Alias

You can create alias for links. This is especially adapted for web sites containing reference to other pages.

```
*Chapters>chapterAndSections*
```

creates the link Chapters. Again the template may interpret this differently.

To create a link to another pillar file, use the `*Alias>File.pillar@AnchorName*`. The alias and anchor parts are optional but you will need them in some cases (for example if you have an inter-file link and you export in LaTeX, or if you have an inter-file link and you export all your file in the same html file).

External links

To create links to external resources, use the `*Pharo>http://pharo.org/*` syntax which is rendered as Pharo[1].

```
*http://www.pharo.org*
```

The same syntax can also represent email addresses: write `*dupond@free.fr*` to get dupond@free.fr.

3.3 Lists

Pillar offers three default lists: Bullet, numbered and labelled.

[1] http://pharo.org/

Bulleted lists

```
-A block of lines,
-where each line starts with ==-==
-is transformed to a bulleted list
```

generates

- A block of lines,
- where each line starts with `-`
- is transformed to a bulleted list

Numbered lists

```
#A block of lines,
#where each line starts with ==#==
#is transformed to a numbered list
```

generates

1. A block of lines,
2. where each line starts with `#`
3. is transformed to a numbered list

Definition lists

Definition lists (*aka.* description lists) are lists with labels:

```
;blue.
:color of the sky
;red.
:color of the fire
```

generates

blue. color of the sky

red. color of the fire

List nesting

```
-Lists can also be nested.
-#Thus, a line starting with ==-#==
-#is an element of an unordered list that is part of an ordered list.
```

generates

- Lists can also be nested.
    1. Thus, a line starting with `-#`
    2. is an element of a bulleted list that is part of an ordered list.

3.4 Commented lines

Lines that start with a `%` are considered comments and will not be rendered in the resulting document.

3.5 Escaping characters

Special characters (e.g., `+` and `*`) must be escaped with a backslash: to get a `+`, you actually have to write `\+`. The list of characters to escape is:

```
^, _, :, ;, =, @, {, |, !, ", #, $, %, ', *, +, [, -
```

3.6 Formatting

There is some syntax for text formatting:

- To make something **bold**, write `""bold""` (with 2 double quotes)
- To make something *italic*, write `''italic''` (with 2 single quotes)
- To make something `monospaced`, write `==monospaced==`
- To make something ~~strikethrough~~, write `--strikethrough--`
- To make something $_{\text{subscript}}$, write `@@subscript@@`
- To make something $^{\text{superscript}}$, write `^^superscript^^`
- To make something underlined, write `__underlined__`

3.7 Pictures

To include a picture, use the syntax `+caption>file://filename|parameters+`:

```
+Caption of the
    picture.>file://figures/pharo-logo.png|width=50|label=pharoLogo+
```

generates Figure 3-1 (this reference has been generated using `*@pharoLogo*`).

Parameters are pairs `"label"="value"` separated by `|`.

3.8 Tables

To create a table, start the lines with `|` and separate the elements with `|`. Each new line represents a new row of the table. Add a single `!` to let the cell become a table heading.

Figure 3-1 Caption of the picture.

```
|!Country |!Capital
|France | Paris
|Belgium | Brussels
```

Country	**Capital**
France	Paris
Belgium	Brussels

The contents of cells can be aligned left, centered or aligned right by using `|{`, `||` or `|}` respectively.

```
||centered||!centered header||centered
|{ left |} right || center
```

generates:

centered	**centered header**	centered
left	right	center

3.9 Preformatted

To create a preformatted block, begin each line with `=`. A preformatted block uses equally spaced text so that spacing is preserved. In general you should prefer code block over preformatted blocks. Code blocks more powerful.

```
= this is preformatted text
= this line as well
```

3.10 Code block

Use code blocks when you want to add code snippets to your document. Code blocks are delimited by `[[[` and `]]]`.

```
[[[
foo bar
]]]
```

generates

Listing 3-2 My code block that works

```
self foo bar
```

```
foo bar
```

Code block with a label or caption

If you want either a label (to reference the code later) or a caption (to give a nice title to the code block), write the following:

```
[[[label=foobar|caption=My code block that works|language=smalltalk
self foo bar
]]]
```

which produces script 3-2 (this reference is produced with `*@foobar*`).

Code block syntax highlighting

To specify the syntax a code block is written in, you need to use the `language` parameter. For example on 3-2 we used the `smalltalk` value for the `language` parameter.

> **Note** The currently supported languages are bash, css, html, http, json, javascript, pillar, sql, ston, shellcommands and smalltalk

If you don't want syntax highlighting for a particular script, specify `no language` as value to the `language` parameter.

Code block with line numbers

If you need to explain a long piece of code, you may want a code block to have line numbers:

```
[[[lineNumber=true
self foo bar.
self bar foo.
]]]
```

produces

```
self foo bar.
self bar foo.
```

Code block from an external file

If you want you can also include a code block from a external file. For example if you have a file ‘myProject.html‘ and you want to take the code from line 15 to line 45, instead of copy/pasting the code you can use:

```
[[[language=html|fromFile=myProject.html|firstLine=15|lastLine=45
]]]
```

The `firstLine` and `lastLine` parameters are optional.

Code block whose contents is computed

Sometimes you would like to program the contents of a code block. You can do it using the `eval=tag` and the system offers you a stream variable that you can use to emit the contents of the code block.

=[[[eval=true

```
| languages |
stream nextPutAll: '@@note The currently supported languages are '.
languages := PRRealScriptLanguage withAllConcreteClasses collect: #standardName.
...
```

=]]]

Note that since this feature is a potential security holes it has to be enable via a plugin declaration.

```
  {
    "title":"A simple reflective object kernel",
    "attribution":"Stéphane Ducasse",
    ...
    "plugins": ["PRScriptEvaluator"],
    ...
  }
```

3.11 Annotated paragraphs

An empty line starts a new paragraph.

An annotated paragraph starts with @@ followed by a keyword such as `todo` and `note`. For example,

```
@@note this is a note annotation.
```

generates

Note this is a note annotation.

And,

```
@@todo this is a todo annotation
```

generates a todo annotation

To do this is a todo annotation

By default the templates support `@@todo` and `@@note`.

3.12 Raw

If you want to include raw text into a page you must enclose it between `{{{` and `}}}`, otherwise Pillar ensures that text appears as you type it which might require transformations.

A good practice is to always specify for which kind of export the raw text must be outputted by starting the block with `{{{latex:` or `{{{html:`. For example, the following shows a formula, either using LaTeX or plain text depending on the kind of export.

```
{{{latex:
\begin{equation}
   \label{eq:1}
   \frac{1+\sqrt{2}}{2}
\end{equation}
}}}
{{{html:
(1+sqrt(2)) / 2
}}}
```

Take care: Avoid terminating the verbatim text with a } as this will confuse the parser, better add a space or two. So, don't write `{{{\begin{scriptsize}}}}` but `{{{\begin{scriptsize} }}}` instead.

3.13 Meta-Information

Meta-information of a particular file is written at the start of the file between curly braces using the STON syntax (a super set of JSON). A meta-information starts with a word between quotation marks acting as a key, is followed by a colon `:`, and finishes with a value.

For example, the following Pillar file,

```
{
  "title": "A first document",
  "author": "John Doe"
}

!Hello World
```

represents a Pillar document with the title and author set. You can use whatever keys you like. The metadata keys can be used in templates (see Chapter **??** for more information about templating).

For example the template for the current book uses the following template variables: attributions, title, series, and keywords. You can see this in the file main.mustache in the _support/template/latex folder.

3.14 Annotations: supporting extensions

Pillar proposes two extensions mechanisms: tags and parameters. Pay attention that each plugin should handle the tag or parameter processing.

Annotations are the Pillar way to have extensible syntax. An annotation has this syntax:

```
${tag:parameter=value|parameter2=value2}$
```

InputFile annotation

You can include a file into another pillar file. The `inputFile` annotation takes as parameter the path of the file relative to `baseDirectory` (if you don't change the base directory, it is your working directory). In this example, 2 files are included:

```
${inputFile:path=test.pillar}$

${inputFile:path=chapter2/chapter2.pillar}$
```

Footnote annotation

You can add footnotes to explain or annotate words. The `footnote` annotation takes as parameter the note which will appear at the end of the document. In this example, one footnote is added.

```
Foo${footnote:note=Some Explanation for Foo}$
```

In addition we should add the plugin as follows in the pillar.conf file.

```
  "plugins": ["PRFootnoteTransformer"],
```

Citation annotation

You can add citations to refer to publications or books. The `citation` annotation takes as parameter the reference using the `ref`.

```
In this chapter we will explore a minimal reflective class-based
    kernel, inspired
from ObjVlisp ${cite:ref=Coin87a}$.
```

You have to declare the plugins as follow.

```
  {
    "title":"A simple reflective object kernel",
    "attribution":"Stéphane Ducasse",
    "series": "The Pharo TextBook Collection",
    "keywords": "bootstrap, reflective, meta system, Pharo,
    Smalltalk",
    "latexWriter" : #'latex:sbabook',
    "plugins": ["PRCitationTransformer"],
    "bibFile": "others.bib",
    "newLine": #unix,
    "htmlWriter": #html
  }
```

Parameters

Pictures and scripts are both using parameters to specify different options and variations. Parameters are pairs `"label"="value"` separated by `|`.

Here are two examples within the same expressions: Code blocks and captions use parameters.

```
[[[label=foobar|caption=My code block that works|language=pillar
+Caption of the picture.>file://figures/pharo-logo.png|width=50|label=pharoLogo+
]]]
```

CHAPTER 4

Specific markup

This chapter presents some extensions specific to a given support such as LaTeX or slides.

4.1 Generate a part of your document with a script

If you want you can also evaluate a script to generate a part of your document. Notice that some pillar version may disable it for security reasons.

For example if you write a project's documentation and want to give some metrics about its code, you can write something like this:

```
[[[eval=true
| packages classes |
packages := RPackageOrganizer default packages select: [ :each |
               each name includesSubstring: 'Pillar' ].
classes := packages flatCollect: [ :each | each classes ].
stream
   nextPutAll: 'The Pillar project contains:';
   lf;
   nextPutAll: '- ==';
   print: packages size;
   nextPutAll: ' packages==.';
   lf;
   nextPutAll: '- ==';
   print: classes size;
   nextPutAll: ' classes=='.
]]]
```

will generate:

The Pillar project contains:

- `41 packages.`
- `455 classes`

To enable this feature, you should declare that you want to use the corresponding plugin in the `pillar.conf` (`"plugins": ["PRScriptEvaluator"]`).

```
{
  "title": "Publishing Documents with Pillar 7.0",
  "attribution": "Stéphane Ducasse and Guillermo Polito",
  "series": "Square Bracket tutorials",
  "keywords": "Pillar, HTML, PDF, Markdown, EPUB, AsciiDoc, Beamer,
    Pharo, Smalltalk",
  "newLine": #unix,
  "plugins": ["PRScriptEvaluator"],
  "htmlWriter": #html
  }
```

4.2 LaTeX

Citations

Citations are only available for LaTeX. You can add citations to your document to reference an element in a LaTeX bibliography. The `cite` annotationtakes as parameter the key of the reference in the bibliography.

```
${cite:ref=Duca17a}$
```

The example above will render as `cite{Duca17a}`

If you want to use other type of citations like `citep` or `citet`, please overwrite the command in your LaTeX template: `renewcommand{cite}{citep}`

Future versions will support also the definition in pillar.conf of the bibliography file. Right now you should harcode it in the template.

```
\bibliographystyle{alpha}
\bibliography{rmod,others,new}
```

You have to add a plugin in the pillar.conf file to manage the creation of bibliography.

4.3 Slides

Pillar offers the possibility to define slides. Slides requires the use of the presentation archetype.

Slide annotation

This annotation is used to create slides structure for a `beamer` or a `deck.js` export. The parameter **title** is required. The **label** parameter can be used to refer to this slide in another slide:

```
${slide:title=My slide|label=sld:mySlide}$
```

Columns

With Pillar you can put text and other contents in columns. To do that, you need to delimit an environment with the `columns` and `endColumns` annotations. Then you can create columns with the `column` annotation. The column annotation takes 1 required parameter: the width of the column. Here is an example:

```
${columns}$

   ${column:width=60}$
      bla
   ${column:40}$
      bla

${endColumns}$
```

> **Note** The column annotations currently works only for the beamer, HTML and Deck.js export.

CHAPTER 5

Pillar cheatsheet

Headings

`! Heading`	Heading 1	`!!! Heading`	Heading 3
`!! Heading`	Heading 2	`!!!! Heading`	Heading 4

Anchors and links

`@anchor`	anchor
`*@Internal Link*`	Internal link
`*http://www.pharo.org*`	External link
`*Pharo>http://www.pharo.org*`	Alias external link

Lists

`-bullet`	Creates one bullet
`#numbered`	Creates one numbered item
`;description :definition`	Creates a description definition item

Formatting

`""bold""`	**bold**
`''italic''`	*italic*
`==monospaced==`	`monospaced`
`--strikethrough--`	~~strikethrough~~
`@@subscript@@`	$_{\text{subscript}}$
`^^superscript^^`	$^{\text{superscript}}$
`__underlined__`	underlined

Characters to be escaped using \ are

```
^, _, :, ;, =, @, {, |, !, ", #, $, %, ', *, +, [, -
```

`%` commented line	each line starting with `%` is ignored

Pictures

```
+Caption.>file://figures/pharo-logo.png|width=50|label=pharoLogo+
```

Tables

`|! Column1	Column2`	Two column header
`||`	centered
`|{ left`	left
`|} right`	right

Annotated paragraphs

==@@note==	Produce a note paragraph
==@@todo==	Produces a todo

Code blocks

```
[[[
self foo bar
]]]
```

```
[[[label=foobar|caption=My code block that works|language=smalltalk
self foo bar
]]]
```

Raw

```
{{{latex:
latex code here
}}}
{{{html:
plain html here
}}}
```

CHAPTER 6

Building automatically your document with Travis

Travis is a service similar to Jenkins that is useful to automate your processes. Once configured, a travis job will run each time you commit to your book github repository. We will show now how we can use Travis to automatically build the pdf. In addition we will show how we can automatically publish the pdf on the github repository itself.

6.1 Configuring your github account

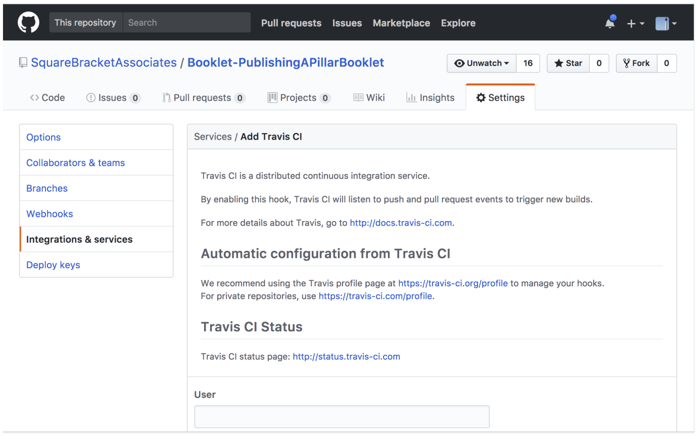
The first to start with is to enable the travis integration from your github repository. You will find the integration in Settings and the menu integration services. You should look for Travis CI and enable it as shown in Figure 6-1.

Once this is done, you should go to your travis account http://travis-ci.com and enable your project as shown in Figure 6-2.

Note that this setup may take a while to sync. So do not worry if these systems start to off right in the minute. It may take a while, so let us pass to the next item.

6.2 Add and edit the .travis.yml file

Here is the `.travis.yml` of this booklet. You can find it online at https://github.com/SquareBracketAssociates/Booklet-PublishingAPillarBooklet:



SquareBracketAssociates / Booklet-PublishingAPillarBooklet

Services / **Add Travis CI**

Travis CI is a distributed continuous integration service.

By enabling this hook, Travis CI will listen to push and pull request events to trigger new builds.

For more details about Travis, go to http://docs.travis-ci.com.

Automatic configuration from Travis CI

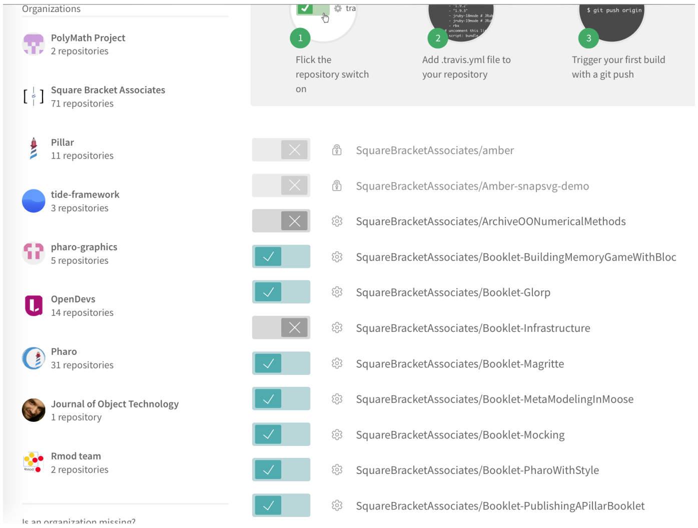
We recommend using the Travis profile page at https://travis-ci.org/profile to manage your hooks.
For private repositories, use https://travis-ci.com/profile.

Travis CI Status

Travis CI status page: http://status.travis-ci.com

User

Figure 6-1 Enabling the travis service in your github account.


Organizations
1
Flick the repository switch on
2
Add .travis.yml file to your repository
3
Trigger your first build with a git push
$ git push origin
Is an organization missing?

PolyMath Project
2 repositories

Square Bracket Associates
71 repositories

Pillar
11 repositories

tide-framework
3 repositories

pharo-graphics
5 repositories

OpenDevs
14 repositories

Pharo
31 repositories

Journal of Object Technology
1 repository

Rmod team
2 repositories

✗	SquareBracketAssociates/amber
✗	SquareBracketAssociates/Amber-snapsvg-demo
✗	SquareBracketAssociates/ArchiveOONumericalMethods
✓	SquareBracketAssociates/Booklet-BuildingMemoryGameWithBloc
✓	SquareBracketAssociates/Booklet-Glorp
✗	SquareBracketAssociates/Booklet-Infrastructure
✓	SquareBracketAssociates/Booklet-Magritte
✓	SquareBracketAssociates/Booklet-MetaModelingInMoose
✓	SquareBracketAssociates/Booklet-Mocking
✓	SquareBracketAssociates/Booklet-PharoWithStyle
✓	SquareBracketAssociates/Booklet-PublishingAPillarBooklet

Figure 6-2 Enabling your github project on travis.

```
language: smalltalk
sudo: false

os:
  - linux

smalltalk:
  - Pharo-7.0

install:
  - git clone https://github.com/pillar-markup/pillar.git -b v7.4.1
# Run pillar build script. Pillar will be built in `pwd`/build!
  - cd pillar && ./scripts/build.sh && cd ..
# Install latex
  - . pillar/scripts/ci/ensure_latex.sh
  # Install latex dependencies required by pillar
  - ./pillar/scripts/ci/ensure_book_dependencies.sh

script:
  - ./pillar/build/pillar build pdf

after_success:
  - wget -c
    https://github.com/probonopd/uploadtool/raw/master/upload.sh
  - mv _result/pdf/book.pdf pillarBooklet-wip.pdf
  - bash upload.sh pillarBooklet-wip.pdf

branches:
  except:
    - /^(?i:continuous)$/
```

Add a similar configuration file to your repository. Once the travis github get synchronised and travis is kicked in, you will be able to check on the travis log that your project has been successfully built.

6.3 Some explanations

Let us explain some parts: the `after_success` section of the configuration is using a script to release continuously on each green commit. The documentation is available at https://github.com/probonopd/uploadtool/.

```
after_success:
  - wget -c
    https://github.com/probonopd/uploadtool/raw/master/upload.sh
  - mv _result/pdf/book.pdf pillarBooklet-wip.pdf
  - bash upload.sh pillarBooklet-wip.pdf

branches:
  except:
    - /^(?i:continuous)$/
```

6.4 Using badges to indicate build status

You can now use the status of a travis build right in the github repository using markdown badges in the project `README.md` file.

```
# A booklet explaining how to build a booklet

[![Build status][badge]][travis]

[travis]:
    https://travis-ci.org/SquareBracketAssociates/Booklet-PublishingAPillarBooklet
[badge]: https://travis-ci.org/SquareBracketAssociates/
Booklet-PublishingAPillarBooklet.svg?branch=master

## To contribute
- Fork
- Do pull Request

## To latex it locally

```
pillar build pdf
```
```

At the stage you should get already important feedback since you will know if your project fully builds or not.

How to add a new released file in your git hub account

To release a pdf that will be stored on github in the booklet repository, we should issue an annotated tag as follow:

```
git tag -a v1.0-Pharo50
git push --tags
```

6.5 Conclusion

Now you are ready to manage and automatically build documents in various formats. s

CHAPTER 7

Testing Your Documents

What does it mean to 'test a book' and why would you do it? Testing a book means checking the code it displays is up to date. This can be done by checking if examples are correct or verifying that method and class definitions compile without raising an error in the last Pharo version. Being able to test a book means updating it more easily and therefore more frequently. On the long term, easier updates means more frequent updates but also a documentation improvement along with more ressources available for the community.

Pillar comes with a book tester.

7.1 Typical codeblocks

Pillar syntax allows you to write codeblocks, showing a part of code. The body of a codeblock allows one to display any part of code you might want to. For instance, example, method and class definitions can be tested.

Examples

```
[[[
  1 + 1
>>> 2
]]]
```

Method definition

This codeblock presents a method definition.

```
[[[
YourClass >> yourMethod
  ^ 'bla'
]]]
```

Class definition

This codeblock presents a class definition.

```
[[[
Object subclass: #YourClass
     instanceVariableNames: ''
     classVariableNames: ''
     package: 'YourPackage'
]]]
```

As we show next, using parameters we can indicate that the expression inside a codeblock should be validated

7.2 Testing codeblock examples

Using the parameter `example` and the >>>= helps you to validate your examples. For example

```
[[[example=true
3 + 4
>>> 7
]]]
```

The book tester interprets the previous parametrized codeblock as an assertion `self assert: (3 + 4 ) equals: 7`.

The book tester follows the following pattern with only one >>> per codeblock.

```
[[[example=true
   Action
   >>> ExpectedResult
]]]
```

7.3 Testing method and class definitions

The booktester checks that class and method definitions are valid (do not raise syntactical errors at compilation). You should use the following parameters: `methodDefinition=true` or classDefinition=true as code block. It makes sure your book is up to date with the latest Pharo version.

Method definitions

Method definitions should follow this format:

```
[[[methodDefinition=true
  YourClass >> yourMethod
  methodBody
]]]
```

with an emphasis laid on the >> between your class and method.

It is important to understand that the class (here `YourClass`) should be present in the system producing the Pillar book. To load code, see the `${loader}$` annotation presented in the following chapter.

Class definitions

Class definitions should follow this format:

```
 [[[classDefinition=true
   YourSuperClass subclass: #YourClass
        instanceVariableNames: ''
        classVariableNames: ''
        package: 'YourPackage'
]]]
```

In addition to validate that the class definition is valid, the parameter defines the class in the environment in which the book is defined. It supports the definition of testing other methods.

As a general principle, any method definition refering to classes not defined in the system, should be defined after by a class definition or a loading instruction (see `$loader$` as shown in the following).

7.4 Practical testing

There are currently two ways of testing your book: (1) using the command line and (2) from within Pharo. Now from the command line two options are supported.

Check file

```
> $PHARO_VM $IMAGE clap checkFile yourfolder/f1.pillar
```

When a path is not explicitely stated, the book tester takes the first file matching `*.pillar`.

To do Add the help output.

Check all files

```
> $PHARO_VM $IMAGE clap checkRepository yourfolder/f1.pillar
```

When a path is not explicitely stated, the book tester takes the first file matching `*.pillar`.

```
> $PHARO_VM $IMAGE clap checkRepository --help
```

Check full book

```
> $PHARO_VM $IMAGE pillar build checkBook --baseDirectory=`pwd`
```

This second version uses the already existing `build` command for Pillar. Pay attention the key difference is that the build version works on expanded text version where annotations have been processed while the clap version only on the pillar textual input.

However, using one method or the other leads you to obtaining a report on all tests made. The passed ones are not displayed, however the failed ones are shown by stating first the explanation of the failure (name of the exception or if the assertion failed). Note that only codeblocks with one of the three following parameters will be tested: `example`, `methodDefinition` and `classDefinition`. And they will be tested using the methods used above.

To do Add a report

7.5 From within Pharo

You can also test a pillar file from within Pharo itself. To do so you should use a `PRBookTesterVisitor` to check a file using `checkFileNamed:`. This can be useful to inspect each of the results. In particular the successful one.

```
PRBookTesterVisitor new checkFileNamed: aFile
```

7.6 What can not be tested

Some codeblocks can not be tested. For example, nested results to show what happens after some iterations such as:

```
  [[[
    1 + 1
    >>> 2
    >>> 3
    >>> 4
  ]]]
```

Local variable definitions are not validated:

```
[[[
  | tmp |
  tmp := 0.
  tmp >>> 0
]]]
```

Or examples that show an error should be raised such as:

```
[[[
  String \+
  >>> Error
]]]
```

Or examples without concrete values

```
[[[
  Date today
  >>> aDate
]]]
```

CHAPTER 8
Improving Book Writability

BookTester can do a lot more for you. Indeed you can load a specific version of your code and display it instead of copying it from your code repository to your pillar. In addition you can run your tests.

8.1 Loading your code

How to test something you have defined but not loaded? This certainly is the question, especially when using class or methods that you just defined (which happens in nearly every book you might want to write). To answer that question, the following annotation has been implemented:

```
${loader:account=YourGitAccount|
  project=YourGitProject|
  tag=YourGitTag|
  baseline=YourBaselineName}$
```

This annotation loads your git project at the given tag. The tag is useful to specify your chapter's prerequisites (precedent chapter for example). Note that a proper use of this annotation requires you to tag every step of your code to create milestones. The baseline parameter takes by default the name of the project, but it is present in case they are different.

8.2 Support test-driven development writing

When writing book to teach TDD, authors often place test first and only after method definitions. This is why validating a test just after its definition may be not work.

This is why every codeblock with the parameter `methodDefinition` has its compilation tested but a test will not be run until the `run` annotation is encountered.

```
${run:testClass=YourTestClassName}$
```

The `run` annotation executes every test defined with `methodDefinition` within a given test class and report the results in the booktester report.

Typical TDD booklet

A typical sequence will be then:

```
   [[[classDefinition=true
     TestCase subclass: #MyTestCase
   ]]]

   [[[methodDefinition=true
       MyTestCase >> testFoo
             self assert: Foo new name equals: 'foo'
   ]]]

   [[[methodDefinition=true
      MyTestCase >> testFoo2
        self assert: Foo new friendName equals: 'bar'
   ]]]

    [[[classDefinition=true
    Object subclass: #Foo
   ]]]

   [[[methodDefinition=true
     Foo >> name
        ^ 'foo'
  ]]]

     ${run:testClass=MyTestCase}$

   [[[methodDefinition=true
       Foo >> friend
          ^ 'bar'
    ]]]

    ${run:testClass=MyTestCase}$
```

About method definition

It is possible to define a method that will be used in a test but without showing its definition in the text using the `hidden` parameter. For example, when the author does not want to show accessors but want to use them in tests.

The following method will be compiled to the class `Foo` but will not be shown in the book output.

```
  [[[methodDefinition=true,hidden=true
    Foo >> addedButNotShown
      ^42
  ]]]
```

8.3 Helping Tools

In addition to the previous annotations, book tester offer three other annotations: `showClass`, `showMethod`, and `screenshot`.

showClass

The `showClass` annotation allows you to display the class definition of your choice and is used as follows

```
${showClass:class=YourClassName}$
```

For example, using the annotation like this:

```
${showClass:class=Integer}$
```

displays the following in the text

```
Number subclass: #Integer
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Kernel-Numbers'
```

Note that currently the displayed class definition is not parametered as `classDefinition` because we supposed there is no need to test the compilation of it. Whether you defined it somewhere else and want to show it again or it is already defined in the system and therefore does not need to be compilation-tested.

showMethod

The `showMethod` annotation allows you to display the class definition of your choice and is used as follows

```
${showMethod:method=isPowerOfTwo|class=Integer}$
```

displays the following method in the document:

```
Integer>>isPowerOfTwo
  "Return true if the receiver is an integral power of two."
  ^ self ~= 0 and: [(self bitAnd: self-1) = 0]
```

For the same reasons as `showClass`, the displayed method definition is not parametered as `methodDefinition`.

screenshot

The `screenshot` annotation is used as follows:

```
${screenshot:class=YourClassName|method=yourMethodName|caption=yourCaption|width=your
```

This annotation uses the class and method names to create both:

- **PNG File** corresponds to a System Browser opened on the given method definition found in the given class method dictionary ; this file is stored under YourChapter/figures/screenshot/YourClassName»YourMethodName-Date.png
- **Figure Reference** replaces the annotation with the fresh created PNG file and given caption/width/label

The default value of `caption` is `YourClassName>>YourMethodName`. The default value of `label` is `lbl_yourMethodName`. The default value of width is `50`.

8.4 Conclusion

This chapter shows that Pillar offers strong support to synchronise your code and its documentation. First it provides ways to test that examples or definitions are correct. Second it offers ways to link your versioned code can be loaded and displayed. Finally, even tests in the code loaded can be verified.

CHAPTER 9

Migrating from Pillar 50 to Pillar 70

9.1 Uninstall Pillar 50

```
rm Pharo.image
rm Pharo.changes
rm pharo
rm pharo-ui
rm -r pharo-vm
rm mustache
rm pillar
rm Makefile
rm -r support/makefiles
rm download.sh
```

9.2 Install Pillar 70

```
git clone git@github.com:pillar-markup/pillar.git -b newpipeline
     _pillar
./_pillar/scripts/build.sh
```

To keep a local pillar

```
mv build _pillar/bin/
export PATH=_pillar/bin:$PATH
```

9.3 Updating the templates

```
pillar updateTemplate book
```

9.4 Converting the pillar.conf

Old pillar.conf

```
  {
    "metadata" : {
      "title": "The Pillar Super Book Archetype",
      "attribution": "The Pillar team",
      "series": "Square Bracket tutorials",
      "keywords": "project template, Pillar, Pharo, Smalltalk"
    },
    "outputDirectory": "build",
    "mainDocument": "book",
    "latexTemplate": "support/templates/main.latex.mustache",
    "latexChapterTemplate":
    "support/templates/chapter.latex.mustache",
    "htmlTemplate": "support/templates/html.mustache",
    "htmlChapterTemplate": "support/templates/html.mustache",
    "chapters": [
    "Chapters/Chapter1/chapter1",
    "Chapters/Chapter2/chapter2"],
    "newLine": #unix,
    "configurations": {
      "LaTeX" : {
        "outputType": #'latex:sbabook',
        "separateOutputFiles": true
      },
      "HTML" : {
        "outputType": #html,
        "separateOutputFiles": true
      }
    }
  }
```

- We need to first flatten the metadata tag from this:

```
{
  "metadata" : {
    "title": "The Pillar Super Book Archetype",
    "attribution": "The Pillar team",
    "series": "Square Bracket tutorials",
    "keywords": "project template, Pillar, Pharo, Smalltalk"
  },
  ...
}
```

to this:

```
{
  "title": "The Pillar Super Book Archetype",
  "attribution": "The Pillar team",
  "series": "Square Bracket tutorials",
  "keywords": "project template, Pillar, Pharo, Smalltalk",
  ...
}
```

- We then need to replace the configurations entries configure a writer for each of the possible outputs we will use:

```
{
  ...
  "latexWriter": #'latex:sbabook',
  ...
}
```