Chapter 1

Pillar

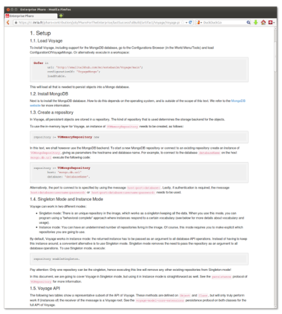
1.1 Introduction

Pillar is a markup syntax and associated tools to write and generate documentation and books. Pillar is currently used to write the Enterprise Pharo book and other projects (see Section 1.1.2). The Pillar screenshot below (see Figure 1.1) shows a part of the Voyage documentation generated by Pillar.

Pillar has many features:

- simple markup syntax with references, tables, pictures, captions...);
- export to HTML, LaTeX, Markdown and Pillar itself;
- customization of the export through a dedicated STON configuration file;
- support of templates using the Mustache templating engine;
- syntax-highlighting of generated code blocks (not yet in LaTeX);
- configurable numbering of section titles and figures;
- ...

Pillar has also:


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.

- a 5-minutes tutorial
- a documentation (in progress)
- a good test coverage (95% with more than a 1150 executed tests)
- a continuous integration job
- a command-line interface
- several existing use cases (see Section 1.1.2)
- a cheat sheet

The contributors are:

- Lukas Renggli created Pier a long time ago from which Pillar's document model, parser and many unit tests are coming;
- Benjamin van Ryseghem did everything on the Mardown exporter and its GitHub Markdown little brother;
- Benjamin van Ryseghem also worked on the command-line interface, the STON configuration interpreter and many other stuff;
- Guillermo Polito fixed a bug;
- Stéphane Ducasse provided continuous feedback from the beginning of the project and got the original idea;
- Damien Cassou for everything else :-).

1.1.1 Editors

These editors have dedicated plugins for Pillar:

- Emacs
- Vim

1.1.2 Pillar users

Pillar is used in several projects. You can have a look at these projects for real use cases of Pillar.

- the Enterprise Pharo book
- the Marina CMS
- the Pier CMS
- the Pillar documentation itself
- the Spec documentation
- the Pharo sound tutorial

1.2 5 minutes tutorial

In this section we give the basic steps to get you started with your first Pillar document and exports. You first need to create a *base directory* inside which we will put all your text, configuration files, and Pillar itself.

```
mkdir mydocument
cd mydocument
```

1.2.1 Installing and exporting your first document

You first need to download Pillar. For that, I recommend executing this script in the base directory.

Then, you can check everything is working fine by creating a first.pier file with this content:

```
!Hello World
```

And finally compiling it from a terminal (see Section 1.6 for more information about the command-line interface):

```
./pillar export --to='html' 'first.pier' > first.html
```

This should generate a first.html file you can open in a web browser. This content of this file will be something like:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>No title</title>
     [...]
  </head>
  <body>
    <div class="container">
      <h1>Hello World</h1>
    </div>
     [...]
  </body>
</html>
```

1.2.2 Configuring a document

As you can see, there is no title in the generated first.html file. This is because we did not specify any. To specify a title, we have to write a configuration file. This configuration is typically named pillar.conf and is written in the STON format (see Section 1.4 for more information about the configuration). Create your first pillar.conf file:

```
{
    "title" : "My first document while reading the 5 minutes Pillar tutorial"
}
```

When you compile using the same command line,

```
./pillar export --to=html first.pier > first.html
```

you should now get a web page with a title:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>My first document while reading the 5 minutes Pillar tutorial</
  </head>
```

1.2.3 Exporting a different content using a template

If you want to tweak the content of the exported file, for example to reference your CSS or to add a footer, you need to create your own template (see Section 1.5 for more information about templating). You must write such template in its own file, e.g., myhtml.template:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>{{{title}}}</title>
  </head>
  <body>
    <div class="container">
      {{{content}}}
    </div>
    <footer>
      <p>Damien Cassou, 2014</p>
    </footer>
  </body>
</html>
```

Then, you need to reference this template from your configuration file. So, edit your pillar.conf configuration file:

```
{
    "title" : "My first document while reading the 5 minutes Pillar tutori
    "configurations" : {
        "html" : {
            "template" : "myhtml.template",
            "outputType" : #html
        }
     }
}
```

Now, compile first.pier

```
./pillar export --to=html first.pier > first.html
```

to generate a first.html that will contain:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>My first document while reading the 5 minutes Pillar tutorial</title>
  </head>
  <body>
    <div class="container">
      <h1>Hello World</h1>
    </div>
    <footer>
      <p>Damien Cassou, 2014</p>
    </footer>
  </body>
</html>
```

This concludes our 5 minutes tutorial.

1.3 Writing Pillar documents

In this section we show how to write Pillar documents by presenting Pillar syntax. You might want to have a look at the cheat sheet and even download and print it.

1.3.1 Chapters & Sections

A line starting with ! becomes a chapter heading. Use multiple ! to create sections and subsections. To refer to a section or chapter, put an anchor (equivalent to \label{chapterAndSections} in Latex) using the @chapterAndSections syntax on a *separate line*. Then, when you want to link to it (equivalent to \ref{chapterAndSections} in Latex), use the *chapterAndSections* syntax. Anchors are invisible and links will be rendered as: 1.3.1.

1.3.2 Paragraphs and framed paragraphs

An empty line starts a new paragraph.

An annotated paragraph starts a line with @@ followed by either todo or note. For example,

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

generates

this is a note annotation.

And,

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

generates a todo annotation that is not visible in the output.

this is a todo annotation

1.3.3 Lists

Unordered lists

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

generates

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

Ordered lists

```
#A block of lines,
#where each line starts with ==#==
#is transformed to an ordered list, where each line is an entry.
```

generates

1. A block of lines,
2. where each line starts with #
3. is transformed to an ordered list, where each line is an entry.

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 a bulleted 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.

1.3.4 Formatting

There is some sugar for font formatting:

- To make something **bold**, write ""bold""
- To make something *italic*, write "italic"
- 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__

1.3.5 Tables

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

```
|!Language |!Coolness
|Smalltalk | Hypra cool
|Java | baaad
```

Language	**Coolness**
Smalltalk	Hypra cool
Java	baaad

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

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

generates

centered	centered	centered
left	right	center

1.3.6 Links

Internal Links and Anchors

To put an anchor (equivalent to \label in Latex), use the @anchorName syntax on a *separate line*. Then, when you want to link to it (equivalent to \ref in Latex), use the *anchorName* syntax. Anchors are invisible and links will be rendered as: 1.3.6.

External Links

To create links to externals resources, use the *Pharo>http://pharo-project.org/* syntax which is rendered as Pharo.

Figure 1.2: This is the caption of the picture.

1.3.7 Pictures

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

```
+Caption of the picture>file://figures/pier−logo.png|width=50|label=pierLog
```

generates Figure 1.2 (this reference has been generated using *pierLogo*).

1.3.8 Scripts

Use scripts when you want to add code blocks to your document.

```
{[}{[}{[}
    foo bar
{]}{]}{]}
```

generates

```
foo bar
```

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

```
{[}{[}{[}label=script1\textbar{}caption=My script that works\textbar{}language=Smallt
    self foo bar
{]}{]}{]}
```

which produces

Script 1.1: *My script that works*

```
self foo bar
```

This script can then be referenced with *script1* (produces ??). Specifying the language (here Smalltalk) will give you syntax highlighting.

1.3.9 Raw

If you want to include raw text into a page you must enclose it in {{{ and }}}, otherwise Pier ensures that text appears as you type it.

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 now only LaTeX, HTML and Markdown are supported). For example, the following shows a formula, either using LaTeX or an image depending on the kind of export.

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

This results in

$$\frac{1+\sqrt{2}}{2} \tag{1.1}$$

Take care: avoid terminating the verbatim text with a } as this will confuse the parser. So, don't write ~~{{{\begin{scriptsize}}}}~~ but {{{\begin{scriptsize} }}} instead.

1.3.10 Preformatted (less used)

To create a preformatted block, begin each line with =. A preformatted block uses equally spaced text so that spacing is preserved.

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

1.3.11 Commented lines

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

1.4 Configuring your output

In this section we show how to configure the export from the Pillar files.

1.4.1 Configuration file

Pillar exporting mechanism can be configured using STON, a lightweight, text-based, human-readable data interchange format (similar to the popular JSON.

1.4.2 Configuration parameters

baseDirectory

Indicate where to look for files with a non-absolute path.

Default value The current working directory.

configurations

Each configuration can define several sub configurations, each of which inherits the properties of its parent.

Default value A dictionary of default configurations from the exporters.

defaultScriptLanguage

Indicate the (programming) language in scripts when none is specified. This language is used for syntax highlighting.

Default value None

headingLevelOffset

Indicate 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 <h3> in HTML.

Default value 0

inputFiles

List the Pillar files that must be exported.

Default value None

newLine

The string that separates lines in the exported document. This is often either LF or CR+LF but any string is possible.

Default value Depend on the operating system.

outputFile

Indicate in which file to export input files.

Default value Standard output.

outputType

Indicate the kind of output desired.

Default value None

separateOutputFiles

Indicate whether each input file must be exported separately or not.

Default value false

startNumberingAtHeadingLevel

Indicate the level of Pillar heading that is going to be numbered with top level numbers. E.g., a startNumberingAtHeadingLevel of value 2 indicates that Pillar heading of level 1 are not numbered.

Default value 2

stopNumberingAtHeadingLevel

Indicate the level of Pillar heading at which Pillar stops numbering. E.g., a stopNumberingAtHeadingLevel of value 4 indicates that Pillar heading of level 4 and more are not numbered.

Default value Infinity (never stop numbering).

template

Indicate the overall structure of the exported documents.

Default value '{{{content}}}' (output the document as is, without any preamble or postamble).

title

Indicate the main title of the document.

Default value 'No title'

verbose

Indicate whether Pillar should write a verbose log when exporting.

Default value false

1.5 Templating

Pillar comes with the Mustache templating engine. This means you can specify a preamble and postamble for your document. Here is an example (bootrap-based) HTML template using Mustache:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>{{{title}}}</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link href="http://netdna.bootstrapcdn.com/bootstrap/3.0.3/css/bootstrap.min.cs
    <link href="http://netdna.bootstrapcdn.com/bootstrap/3.0.3/css/bootstrap-theme.
    <link rel="stylesheet" href="http://yandex.st/highlightjs/8.0/styles/default.mi
    <script src="http://yandex.st/highlightjs/8.0/highlight.min.js"></script>

    <!--[if lt IE 9]>
      <script src="https://oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></scri
      <script src="https://oss.maxcdn.com/libs/respond.js/1.3.0/respond.mi
     <![endif]-->
  </head>
  <body>

    <div class="container">
      {{{content}}}
    </div>

    <script src="https://code.jquery.com/jquery.js"></script>
    <script src="http://netdna.bootstrapcdn.com/bootstrap/3.0.3/js/bootstra
    <script>hljs.initHighlightingOnLoad();</script>
  </body>
</html>
```

In this example, we can see the use of {{{title}}} and {{{content}}} to refer to the title of the document and its actual content (the one exported from Pillar). You have to put such a template in a dedicated file (named chapter.html.template for example) and reference this file from the template configuration parameter (see 1.4.2).

1.6 Command-line interface

In this section we show how to use the pillar command-line interface.

```
$ ./pillar export --to=latex PharoSound.pier > PharoSound.tex
```