字

Glyph

Rapid Document Authoring Framework

v0.1.0

by *Fabio Cevasco*

February 2010

Table of Contents

Introduction

Glyph is a *Rapid Document Authoring Framework*.

Think of it like a sort of Ruby on Rails but for creating text documents instead of web sites. With Glyph, you can manage your documents tidily in *projects* that can be used to generate deliverables in different formats such as HTML or PDF (through Prince).

Main Features

Glyph uses a simple macro system to perform a wide variety of advanced tasks:

- Generate block-level HTML tags not commonly managed by lightweight markups, like `head`, `body`, `div` and `table`.
- Create and validate internal and external links.
- Include and validate images and figures.
- Automatically determine header levels based on the document structure.
- Automatically generate a Table of Contents based on the document structure.
- Store common snippets of text in a single YAML file and use them anywhere in your document, as many times as you need.
- Store configuration settings in a YAML file and use them anywhere in your document, as many times as you need.
- Evaluate Ruby code within your document.
- Call macros from other macros (including snippets), carefully avoiding mutual calls.
- Include text files in other text files.
- Include the contents of configuration settings (author, title) in the document.
- Filter input explicitly or implicitly, based on file extensions when including files.
- Manage comments and todo items.

Installation

`sudo gem install glyph` — simple, as always.

Essential Glyph commands

Glyph is 100% command line. Its interface resambles Git's for its simplicity and power (thanks to the Gli gem). Here are some example commands:

- `glyph init` — to initialize a new Glyph project in the current (empty) directory.
- `glyph add introduction.textile` — to create a new file called *introduction.textile*.

- `glyph compile html` — to compile the current document into a single HTML file.
- `glyph compile pdf` — to compile the current document into HTML and then transform it into PDF using Prince.

Glyph macros in a nutshell

Format your documents using Textile or Markdown, and use Glyph Macros to do everything else:

Glyph Source:

```
section[header[Something about Glyph]
You can use Glyph macros in conjunction
 with _Textile_ or _Markdown_ to
produce HTML files effortlessly.
  section[header[What about PDFs?|pdf]
Once you have a single, well-formatted HTML
file, converting it to PDF is
extremely easy with a 3rd-party
renderer like =>[http://www.princexml.com|Prince].
  ]
]
```

HTML Output:

```
<div class="section">
  <h2 id="h_1">Something about Glyph</h2>
  <p>You can use Glyph macros in conjunction with
     <em>Textile</em> or <em>Markdown</em> to
     produce HTML files effortlessly.</p>
  <div class="section">
   <h3 id="pdf">What about PDFs?</h3>
   <p>Once you have a single, well-formatted HTML
      file, converting it to PDF is
      extremely easy with a 3rd-party renderer
      like <a href="http://www.princexml.com">Prince</a>.</p>
  </div>
</div>
```

Resources

- Project Home Page: http://www.h3rald.com/glyph/.
- Source Code Repository & Bug Tracking: http://www.github.com/h3rald/glyph/
- Book (PDF): http://github.com/h3rald/glyph/tree/master/book/output/pdf/glyph.pdf
- Reference Documentation: http://yardoc.org/docs/h3rald-glyph/

Chapter I – Getting Started

1.1 Creating your first Glyph Project

To install Glyph, simply run `gem install glyph`, like with any other Ruby gem. Then, create a new directory and initialize a new Glyph project, like so:

`mkdir` *test_document*

`cd` *test_document*

`glyph init`

That's it. You just created a new Glyph project in the `test_document` directory.

> **Glyph's dependencies**
>
> Glyph requires the following gems:
> - extlib
> - gli
> - treetop
> - rake
>
> Additionally, some Glyph macros may require additional gems, such as:
> - RedCloth (*textile* macro)
> - Maruku *or* Kramdown *or* BlueCloth (*markdown* macro)
> - Haml (if you want to load .sass files with the *style* macro)

Every Glyph project is comprised of the following directories:

- `images/` — used to store the image files used in your document.
- `lib/` — used to store your custom Glyph macros and Rake tasks.
- `output/` — used to store your generated output files.
- `styles/` — used to store your stylesheets.
- `text/*` — used to store your source text files.

Additionally, the following files are also created at top level:

- `config.yml` — containing your Project Configuration.
- `document.glyph` — containing your Document Structure.
- `snippets.yml` — containing your text snippets.

1.2 Document Structure

Every Glyph project contains a `document.glyph` file that is typically used to define the document structure. The default `document.glyph`
generated automatically when creating a new project is the following:

```
document[
  head[style[default.css]]
  body[
    titlepage[
      title[]
      author[]
      pubdate[]
    ]
    frontmatter[
      toc[]
      preface[header[Preface]
        @[preface.textile]
      ]
    ]
    bodymatter[
      chapter[header[Chapter #1]
        @[chapter_1.textile]
      ]
      chapter[header[Chapter #2]
        @[chapter_2.textile]
      ]
    ]
    backmatter[
      appendix[header[Appendix A]
        @[appendix_a.textile]
      ]
    ]
  ]
]
```

Even without knowing anything about Glyph Language, you can easily figure out that this file defines a document with a Table of Contents, a Preface and some Chapters. `frontmatter[]`, `preface[]`, `chapter[]`, etc. are all Glyph *macros* used to define — in this case — some structural elements. In practice, this means that if you plan to generate an HTML document, they'll be converted into `<div>` tags.

Be aware that other macros, on the other hand, are used to do something completely different, e.g.:

- `toc[]` generates the document's Table of Contents
- `@[]` or its alias `include[]` is used to copy the contents of another file stored anywhere in the `/text` directory.

Let's now analyze this `document.glyph` more in detail.

- The `document[]` macro wraps every other macro. This is necessary to create the initial `<html>` tag.
- Similarly, `head[]` and `body[]` are used to generate the respective HTML tags. Actually, `head[]` already sets some metadata for you, by default (author and copyright).
- Within `head[]`, the `style[]` macro is used to load the `default.css` stylesheet, which is included by default the `/styles` directory of every Glyph project.
- Immediately after the `body[]` macro, the `titlepage[]` macro is used to define (guess…) the first page of your document. `title[]`, `author[]` and `pubdate[]` insert the title of the document, its author and the publication date (retrieved from the project's configuration settings).
- Then, the `frontmatter[]`, `bodymatter[]` and `backmatter[]` macros are used to further divide the portions of your document according to the rules of book design. They are not mandatory, but they can be used, for example, to number your appendixes with letters instead of numbers and similar.
- `preface[]`, `chapter[]`, `appendix[]` are just a way to wrap content in `<div>` tags, from an HTML point of view, but they are also necessary to nest the content of your document and generate the Table of Contents automatically, together with the `header[]` macro.

1.3 Project Configuration

Glyph stores configuration settings in the following YAML files:

- Your *Project Configuration* is stored in the `config.yml` file, included in each Glyph Project.
- Your *Global Configuration* is stored in a `.glyphrc` file in your $HOME (or %HOMEPATH% on Windows) directory (not created by default).
- The *System Configuration* is stored in the source directory of Glyph itself.

When compiling, Glyph loads all these configuration files and merges them according to the following rules:

- A setting configured in the *Project Configuration* overrides the same setting in both Global and System configuration.
- A setting configured in the *Global Configuration* overrides the same setting in the *System Configuration*

Typically, you should use the *Project Configuration* for all project-specific settings and the *Global Configuration* for settings affecting all your projects (for example, you may want to set 'document.author' in the Global Configuration instead of setting it in the Project Configuration of all your Glyph projects). The *System Configuration* is best left untouched.

Instead of editing your configuration settings directly, you can use the `glyph config` command, as follows:

```
glyph config setting [value]
```

If no *value* is specified, glyph just prints the value of the configuration setting, so typing `glyph config document.author` right after creating a project (assuming you didn't set this in the Global Configuration) will print nothing, because this setting is blank by default.

To change the value of a configuration setting, specify a value right after the setting, like this:

```
glyph config document.author "John Smith"
```

In this way, the document author will be set to *John Smith* for the current project. To save this setting globally, add a `-g` option, like this:

```
glyph config -g document.author "John Smith"
```

> ### Regarding configuration values and data types…
>
> Glyph attempts to "guess" the data type of a configuration values by evaluation (`Kernel#instance_eval`) if the value:
> * is wrapped in quotes (`"` or `'`) — String
> * starts with a colon (`:`) — Symbol
> * is wrapped in square brackets (`[` and `]`) — Array
> * is wrapped in curly brackets (`\{` and `\}`) — Hash
> * is *true* or *false* — Boolean
> * If the value is *nil* — NilClass
>
> Note that this guessing is far from being foolproof: If you type something like *{:test, 2}*, for example, you'll get an error.

There are plenty of configuration settings that can be modified, but most of them are best if left alone (and in the System Configuration file). For a complete reference, see Configuration Reference. Normally, you may just want to change the following ones:

Setting	Description
document.author	The author of the document
document.title	The title of the document
document.filename	The document file name

Chapter II – Authoring Documents

2.1 Text Editing

One of the aims of Glyph is streamlining text editing. Glyph accomplishes this through its own macro language that can be used in conjunction with Textile or Markdown.

2.1.1 Introducing Glyph Macros

By now you probably figured out what a macro looks like: it's an identifier of some kind that wraps a value or parameters within square brackets. More specifically:

- The macro identifier can contain *any* character except for: `[`, `]`, `\`, `|` or spaces.
- The delimiters can be either `[` and `]` or `[=` and `=]` (for more information on differences between delimiters, see Escaping and Quoting).
- The value can be anything, even other macros. If a macro supports more than one parameter, they must be separated with `|`. For example, the `link` (=>) macro can take an optional second parameter for the link text: `=>[#link_id|This is the link text]`.

2.1.2 Escaping and Quoting

Glyph doesn't require any special control characters like LaTeX, and its macro syntax is very straightforward and liberal. This however comes with a price: because square brackets are used as delimiters, you must escape any square bracket in your text with a backslash. That's not *too* bad if you think about it, unless you're writing programming code: in that case, escaping every single square bracket can be painful.

If a portion of your text contains an excessive amount of square brackets, you may consider using the `escape` macro (or better, its alias `.`) with `[=` and `=]` as delimiters. By itself, the escape macro doesn't do anything: it just evaluates to its contents, but the special delimiters act as a quote for any square bracket within them. As a consequence, any macro within `[=` and `=]` will *not* be evaluated.

You can use the quoting delimiters with *any* macro identifier. Obviously, using them as delimiters for things like `section` macros may not be a good idea, but they should really be mandatory with the `code` macro, like this:

```
code[=
section[header[A section]

This is a section.

  section[header[A nested section]
This is another section.
  ]
]
=]
```

> **Note**
> Although quoting delimiters allow you to use square brackets without escaping them, you must still escape them if you want to escape quoting delimiter themselves.

Besides square brackets, there are other characters that must or can be escaped with backslashes, as shown in the following table

Escape Sequence	Evaluates to…	Notes
\[	[	Square brackets must be escaped unless used as macro delimiters or within a quoting macro.
\]	]	Square brackets must be escaped unless used as macro delimiters or within a quoting macro.
\\	\	Backslashes do not have to be escaped by default, but an escaped backslash will evaluate to itself.
\=	=	Equal signs do not have to be escaped by default, but an escaped equal sign will evaluate to iself.
|			Pipes must be escaped (even within quoting macros) unless they are used to separate two or more macro parameters.
\.		An escaped dot evaluates to nothing. Useful to separate macro identifiers from other characters: `_=>[#link|This is an emphasized link]_`

2.1.3 Sections and Headers

Glyph documents are normally organized as a hierarchical tree of nested chapters, appendixes, sections, etc. To define a section, use the `section` macro, like so:

```
section[
  header[Section #1]

Write the section contents here...

  section[
    header[Section #2]

This section is nested into the previous one.

  ] --[End of Section #2]
] --[End of Section #1]
```

This example defines two nested sections, each with its own header. The header is *mandatory*: it will be displayed at the start of the section and in the Table of Contents.

Note an important difference from HTML: there is no explicit level for the headers, as it will be determined at runtime when the document is compiled, based on how sections are nested. The previous code snippet (taken as it is), for example, will be transformed into the following HTML code:

```
<div class="section">
  <h2>Section #1</h2>
  <p>Write the section contents here...</p>
  <div class="section">
    <h3>Section #2</h3>
    <p>This section is nested in the previous one</p>
  </div>
</div>
```

By default, in Glyph the first header level is *2*, so the two headers are rendered as h2 and h3, respectively (--[...] macros are *comments*, therefore they are not included in the final output).

There are *a lot* of macros that can be used in the same way as section, one for each element of Book Design. Each one of them is a simple wrapper for a div tag with a class set to its name.

The following table lists the identifiers of all section-like macros, divided according to the part of the book they should be placed in:

Frontmatter	`imprint` [†], `dedication` [†], `inspiration` [†], `foreword` [‡], `introduction` [‡], `acknowledgement`[‡], `prologue`[‡], `toc`[*]
Bodymatter	`volume`, `book`, `part`, `chapter`
Backmatter	`epilogue`[‡], `afterword`[‡], `postscript`[†], `appendix`, `addendum`[‡], `glossary`[**‡], `colophon`[†], `bibliography`[**‡], `promotion`[†], `references`[**‡], `index`[**‡], `lot`[**‡], `lof`[**‡]

*: The `toc` macro is to generate the Table of Contents automatically, and it must be used with no contents (`toc[]`).

**: This macro is likely to be extended in future versions to generate/aggregate content automatically.

†: This section is not listed in the Table of Contents.

‡: Any subsection of this section is not listed in the Table of Contents.

> **Note**
> `frontmatter`, `bodymatter` and `backmatter` are also valid (and mandatory!) macro identifiers, typically already included in the default `document.glyph` file of every project.

2.1.4 Including Files and Snippets

2.1.5 Links and Bookmarks

2.1.6 Images and Figures

2.2 Compiling your project

2.2.1 Adding Stylesheets

2.2.2 HTML output

2.2.3 PDF Output

Chapter III – Extending Glyph

3.1 Creating Glyph Macros

3.2 Creating Rake Tasks

Chapter IV – Troubleshooting

Appendix A – Command Reference

Appendix B – Macro Reference

Appendix C – Configuration Reference