the cover page

Table of Contents

Chapter 1

Introduction

the topic, the background, why the topic is relevant or of interest to you, what what you hoped to achieve, the aims and objectives of the project.

Chapter 2

Literature review and Context

the setting of the project in the context of other relevant work or theories or results. How this setting influenced the project.

Chapter 3

Research/Development Method

the overall approach and rationale. Why the project was tackled in the chosen way, and why others were ruled out.

Chapter 4

Data/Findings/Designs

the project outcome. This might be data collected and tabulated or the design of a program, or whatever outcome was obtained.

Chapter 5

Analisys/Evaluation/Testing

assessing or testing the project outcome. If the project is of type 2 are the results plausible? If the project is of type 3 or 4 then any computer code shold be tested using range inputs.

Chapter 6

Conclusion/Recommendations

as a result of the project. The project does not need to have a positive conclusion. For example, it might prove that some system was not successful. You should indicate to what extent your objectives have been achieved.

Chapter 7

Review/Reflections

this is often missed out

Chapter 8

...

Chapter 9

...

Chapter 10

Appendix - About this documentation

The dilemma

At the time of this writing, I faced a dilemma, which word processor is the best processor for writing this documentation.

The most obvious choices were:

- Microsoft Word
- Apple Pages
- Google Docs

Surely any of the choices above would be suitable, however, in my humble opinion, such options can be somewhat 'evil' when it comes to document formatting, turning most documentation written in one platform unreadable in the other or to the very least almost certainly not looking as intended.

In general, converting the document to a PDF format remediates this problem, however, I am now left with the fact that my content is tightly coupled to the editor that created the documentation. Modification is difficult, for example, someone else may wish to edit the document, or perhaps if at some point in time in the near future I decide to present my documentation as an HTML document on the web, or perhaps as a deck of slides or even publish it in a book format, I may have to revise the entire text looking for any 'evil' formatting issues that was not visible in the former format.

Finally, as a writer, when writing documentation, I should concentrate on writing the documentation, and not about auto generated formatting issues that may arise and drag productivity in a typical writing session. The writer should only worry about the semantics and the content of his/her writing, formatting should be done separately, perhaps not even by the writer himself, or better, simply choose a new format from predefined options written by a talented designer.

LaTeX is a really good option to move away from the formatting problems mentioned above, and beyond, however, it does trap the writer with a little clutter to tinker with in terms of settings and so on. What I am trying to say is, once the document is finalised, it no longer consists of the content and the content only, but also carries several formatting tags. This is one hand demonstrates how powerful LaTeX can be, but in the other, may confuse and distract the writer.

The solution

Since this course is about computer science, I set off to find a solution that would allow me and any other developer to run away from the masses and write simple interchangeable documentation with ease. Writing a solution that could potentially be further enhanced to the point a non programmer could also benefit from.

The requirements were:

- Write content and content only, without distractions

- Formatting should be written separately and be interchangeable/themeable
    - This also means that if some time in future I want to publish my content to a different media, I should be able to do so without too much effort
- It should not be coupled to any specific text editor
- It should have an automated building and deployment solution

The answer was always there, Markdown. Markdown is a really simple and easy to use markup language, it is the de facto standard for readme files in software development and widely used in blogs throughout the web.

A short introduction to Markdown syntax can be found in the link below:

http://daringfireball.net/projects/Markdown/syntax

Although Markdown may not be as powerful as LaTeX, the community around it is immense and really keen in providing further enhancements to it, the following link lists many projects and plugins that address some of those short-comes:

https://github.com/cben/mathdown/wiki/math-in-Markdown

The community also provides an excellent open source Markdown editor called Mou, although Mac specific, all other major operating systems have free alternatives to Mou:

http://25.io/mou/

Finally, there is an open source project which is now maintained by Jakob Truelsen and Ashish Kulkarni, called wkhtmltopdf, this headless command line tool, allows any HTML document to be converted to PDF, and since Markdown is easily convertible to HTML, we now have all the tools we need in order to create our documentation.

http://wkhtmltopdf.org/

The workflow

To make this work, we will need to automate every step of the process, so that we can only write Markdown, then compile/deploy our work with only one command and in the process, if we wish to do so, add some personalised styles to our document.

The application that holds the documentation has been scaffolded using Yeoman, more specifically, using a generator called generator-jekyllrb.

https://github.com/robwierzbowski/generator-jekyllrb

Although Yeoman is an NPM package, backed by Node.js, it combines three simple but powerful Ruby Gems:

- Jekyll: A static blog generator created by the GitHub Team. It provides an easy to use templating language called Liquid as well as giving the means to transform Markdown files into HTML, finally compiling the entire application into deployable static websites.
- Redcarpet: A Markdown to HTML converter.
- Compass: A CSS pre-processor extension of SASS.

Moreover, Jekyll makes it really easy to deploy and host your application directly on Github free of charge.

This Yeoman generator also leverages the power of Grunt and Bower into the workflow.

Most of the Grunt tasks used for this documentation have also been used for the main IOU application, however, a few notable differences are mentioned below:

- grunt-build-control: Allows deployment to Github via Git.
- grunt-jekyll: integrates Jekyll with Node.js.
- grunt-wkhtmltopdf: Compiles HTML files to PDF.

Developing locally

If you wish to use this workflow for other projects or simply try out what has been created so far, please follow the following instructions and make sure all dependencies are met and available on your path:

Dependencies:

- Git
- Ruby and Ruby Gems
  - SASS
  - Compass
  - Jekyll
- Node.js and NPM
  - Grunt
  - Bower
- wkhtmltopdf

Git clone this repository and `cd` into it:

```
$ git clone https://github.com/jbonigomesbbk/jbonigomesbbk.github.io && cd $_
```

Install the NPM packages:

```
$ npm install
```

Install the Bower dependencies:

```
$ bower install
```

Serve the app:

```
$ grunt serve
```

Other than `serve`, you may wish to run:

```
$ grunt serve:dist
```

The latter will first build the application then serve the optimised code, the former, may be a better option when debugging as files will not be minified and changes will be automatically refresh in the browser using 'browser sync'.

Once displayed in the brower, you may wish to explore the documentation and view the PDF generated version too, link found right below the left navigation bar.

The directory structure will look as follow:

- .jekyll
- .sass-cache
- .tmp
- app
- dist
- node_modules

And the root directory hold the following files:

- .bowerrc
- .csslintrc
- .editorconfig
- .gitattributes
- .gitignore
- .jshintrc
- _config.build.yml
- _config.yml
- bower.json
- Gemfile
- Gruntfile.js
- package.json

The top most directories, prefixed with a dot '.' only hold temporary files required by Jekyll to serve the application locally.

All files in the root directory are common configuration files for Git, Jekyll and Node.js.

The node_modules directory holds all the Grunt tasks.

The app directory is where we, developers will actually work.

The dist directory holds the last build from our application, that is, the result of our code. For every build, Grunt will delete this directory and re-create it based on our changes. The dist directory holds the optimised code that is ready for production.

The app directory tree structure looks like this:

- _bower components
- _includes
- _layouts
- _posts

- _scss
- docs
- fonts
- img
- js
- pdf

The top most directory `_bower_components` holds the Bower dependencies. All the directories prefixed with an underscore '_' will not be coppied to the dist directory during the build process.

- _includes: holds the cover page and the xsl file used to generate the table of contents in the PDF.
- _layouts: hold the layout for the main web page, the layout for the documentation page and the layout for the PDF file.
- _posts: holds the actual Markdown that makes up this documentation, they have been separated by chapters.
- _scss: the SASS styles for the app and the print.scss, used by the PDF file.
- docs: holds the documentation landing page.
- fonts: the web fonts used in this project.
- img: the images.
- js: the JavaScript.
- pdf: a placeholder HTML file containing all chapters and extending the PDF layout, as well as the actual generated PDF file.

During your workflow, you may notice that the PDF file looks well formated when running `grunt serve:dist` however, this statement does not hold true when running `grunt serve`, this is because Jekyll compiles the temporary CSS file in a different location when running on debug mode, to avoid too much overhead, when testing the PDF's layout you may use this short-hand:

```
$ grunt build
```

This will build the latest changes and save them under the dist directory, including a well formatted PDF file. You may now open the PDF directly:

```
$ open dist/pdf/index.pdf
```

If you wish to deploy this app to GitHub, you will need to update the remote option in the buildcontrol task, found in the Gruntfile.js, swap it with a valid GitHub Page repository path, more details about GitHub Pages, Jekyll and BuildControl can be found in the link below:

- https://pages.github.com/
- http://jekyllrb.com/
- https://github.com/robwierzbowski/grunt-build-control

Once set up, you may deploy your code using:

```
$ grunt deploy
```

Roadmap

- Keep improving this workflow and gathering feedback from others on how to improve it.
- Create a more generic version to use as a starting point for future documentations, perhaps a Yeoman generator.
- Fix the issue with no styles in the PDF's generated with `grunt serve`.
- Create new themes for the Markdown's, so far it is using the generic GitHub flavour styles.

Details

The source code for this documentation can be found at:

https://github.com/jbonigomesbbk/jbonigomesbbk.github.io

The latest build of this documentation can be found here:

http://jbonigomesbbk.github.io/docs