Weaki, a desktop application for agile software documentation

Diogo Ferreira - diogoff@fe.up.pt

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Ademar Aguiar - aaguiar@fe.up.pt

February 10, 2017
Weaki, a desktop application for agile software documentation

Diogo Ferreira - diogoff@fe.up.pt

Mestrado Integrado em Engenharia Informática e Computação
Abstract

Software documentation is an important aspect of software development but unfortunately not treated as such most of the time. There are many known ways to document software and these should be adopted by the developers taking their work environment into account. The target audience is the most important factor since it dictates the contents and structure of the documentation and assumes the pre-acquired knowledge of the reader. Documentation for the product’s end-user should be completely different to the one viewed by the development team for example.

Agile development describes a mindset which focuses on doing only what is required when it is required. This can also be applied to documentation and there is a set of guidelines to follow among which reusability and simplicity stand out. These two guidelines can be interpreted as the most basic requirements for agile software documentation tools.

Weaki is a cross-platform desktop application, based on the Electron framework, for agile software documentation meant to extend its web version based on DokuWiki. Its principles are based on weakly-typed wikis which means that the pages are structured but it is not enforced on the user who has the freedom to gradually adopt stricter rules but with benefits. Running natively on the desktop brings many benefits such as direct access to the file system, integration with Git and the ability to customize the application to the user.

The application is developed with the use of agile methods in which at the end of each iteration, one week-long, there’s palpable progress and reports on the situation. Starting by implementing the core features of the web version of Weaki, at the end of 3 months it is expected to start working for the next month on refinements and extra-features such as integrating with Slack, GitHub and Google Drive. The results are then compared to the initial goals and the conclusions are taken.
Resumo

Documentação de software é um importante aspeto no desenvolvimento do mesmo mas infelizmente não é tratado como tal a maioria das vezes. Existem muitas maneiras de documentar e estas devem ser adotadas pelos developers tendo o ambiente de trabalho em conta. O público alvo é o factor mais importante tendo em conta que dita os conteúdos e estrutura da documentação e assume o pré-conhecimento do leitor. Documentação para o utilizador final deve ser completamente diferente da que é vista pela equipa de desenvolvimento por exemplo.

Desenvolvimento ágil descreve uma mentalidade que se foca em fazer apenas o que é necessário quando é necessário. Isto também pode ser aplicado em documentação e existe um conjunto de linhas guia a seguir em que reusabilidade e simplicidade se destacam. Estas duas linhas guia podem ser interpretadas como os mais básicos requerimentos para ferramentas de documentação de software ágil.

Weakí é uma aplicação desktop cross-platform, baseada na framework Electron, para documentação de software ágil que pretende estender a ferramenta web DokuWiki. Os seus princípios são baseados em wikis fracamente tipificadas o que significa que as páginas são estruturadas mas não é forçado ao utilizador que tem a liberdade de gradualmente adotar regras mais estritas mas com benefícios. Correndo nativamente no desktop traz muitos benefícios como o acesso direto ao sistema de ficheiros, integração com Git e a habilidade de personalizar a aplicação ao utilizador. A aplicação é desenvolvida com o uso de métodos ágeis em que ao fim de cada iteração, com a duração de uma semana, existe progresso palpável e relatórios da situação. Começando por implementar as funcionalidades básicas, ao final de 3 meses é esperado começar a trabalhar durante os mês seguinte em refinamentos e funcionalidades extra como a integração com o Slack, GitHub e Google Drive. Os resultados são então comparados com os objetivos iniciais e as conclusão são retiradas.
Acknowledgements

I would like to thank my supervisor, Ademar Aguiar, for the opportunity of doing this project in addition to all the patience and support during the development process.

To my family there’s not enough thanking for all the support they’ve always given me, always believing in me and helping me pushing further.

A heartfelt thank you.

Diogo Ferreira
“How can we make sure we wind up behind the right door when the going gets tough? .
The answer is: craftsmanship.”

Robert C. Martin
# Contents

1 Introduction ........................................... 1
   1.1 Context ........................................... 1
   1.2 Motivations and Goals ............................... 1
      1.2.1 Weaki Design Principles ......................... 2
      1.2.2 Weaki Features .................................. 2
      1.2.3 Methodology ..................................... 4
   1.3 Document structure ................................. 4

2 Wikis and tools ......................................... 5
   2.1 Electron ........................................... 5
   2.2 Source Code Editors ................................. 6
   2.3 Wikis ............................................... 7
   2.4 Project Management Tools ......................... 8
   2.5 Conclusions ....................................... 9

3 Application Overview ................................... 11
   3.1 Interface .......................................... 11
   3.2 User Stories ....................................... 12
      3.2.1 Text editing ................................... 12
      3.2.2 Edit Markdown ................................ 14
      3.2.3 Open files ..................................... 14
      3.2.4 Open repositories .............................. 15
      3.2.5 View workspace ................................ 16
      3.2.6 Git integration ................................ 16
      3.2.7 File history ................................... 17
      3.2.8 Markdown preview ............................... 18
      3.2.9 Templates ...................................... 18
      3.2.10 Code Transclusion ............................. 19
      3.2.11 Content Assist ................................ 20
      3.2.12 Themes ....................................... 21
   3.3 Configuration ...................................... 22

4 Architecture ........................................... 23
   4.1 Technology Stack ................................. 23
   4.2 Rendering and Main processes ..................... 23
      4.2.1 Weaki ........................................ 24
      4.2.2 Window ....................................... 24
   4.3 Rich Text Editing .................................. 24
## CONTENTS

4.3.1 TextEditor ................................................. 25
4.3.2 Decorators ............................................... 27
4.3.3 Suggesters ............................................... 28
4.4 Modules ..................................................... 28
  4.4.1 FileManager ........................................... 28
  4.4.2 Git ......................................................... 28
  4.4.3 FileInterpreter ......................................... 28

5 Conclusions and Future Work .................................. 31
  5.1 Contributions .............................................. 31
  5.2 Future Work ............................................... 31
    5.2.1 Typed Pages .......................................... 31
    5.2.2 Refactoring ........................................... 33
    5.2.3 Structured Views ..................................... 33
    5.2.4 Future Integrations .................................. 33

References ..................................................... 35
List of Figures

3.1 Interface overview .................................................. 11
3.2 Editor component .................................................. 12
3.3 Undo and redo menu items ......................................... 13
3.4 Edit submenu ......................................................... 14
3.5 Open file menu ...................................................... 14
3.6 Open file dialog ..................................................... 15
3.7 Open repository menu ............................................... 15
3.8 Open repository dialog ............................................. 15
3.9 Explorer component ............................................... 16
3.10 GitCommit component ............................................ 16
3.11 GitCommit on the View submenu ............................... 16
3.12 Git commands ...................................................... 17
3.13 FileHistory component .......................................... 17
3.14 FileHistory on the View submenu ............................ 17
3.15 Preview component ................................................ 18
3.16 Preview component on the View submenu .................. 18
3.17 PageTemplates component ....................................... 18
3.18 An applied template .............................................. 19
3.19 PageTemplates on the View submenu ....................... 19
3.20 Different types of file references ............................ 19
3.21 Previewing a file’s content ..................................... 20
3.22 Content assist example .......................................... 20
3.23 Content assist auto-complete example ....................... 20
3.24 Content assist on commit hash ................................. 21
3.25 Default theme ..................................................... 21
3.26 Example theme ..................................................... 21
4.1 Remote method invocation with Electron ..................... 24
4.2 Nested nodes ....................................................... 25
4.3 Overlapping nodes ................................................ 26
4.4 Overlapping nodes with one unbreakable .................... 26
4.5 Two unbreakable nodes overlapping .......................... 27
4.6 Image decorator when hovering ................................. 27
4.7 Image decorator when not hovering ............................ 28
LIST OF FIGURES
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Iterations</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Editors Comparison</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Wikis Syntax Comparison</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Wikis Comparison</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>Project Managers Comparison</td>
<td>8</td>
</tr>
</tbody>
</table>
Abbreviations

API Application Programming Interface
BDD Behaviour Driven Development
CSS Cascading Style Sheets
DOM Document Object Model
HTML HyperText Markup Language
NPM Node Package Manager
OS Operating System
RMI Remote Method Invocation
URL Uniform Resource Identifier
Chapter 1

Introduction

Multiple tools for software development are reviewed and compared against each other in order to determine which features are most common and what would make sense to include in Weaki. Most of these tools are not meant to be used exclusively for documenting but many of their features, interface design and philosophies can be availed to that end.

1.1 Context

The desktop application for Weaki is meant to be used as an extension of its online variant which is a collection of plugins for DokuWiki\(^1\) [CF09]. As an application to be ran directly on desktop instead of a browser, much can done to improve the user experience such as integrating with tools like Git\(^2\) and the file system itself. Furthermore, being a tool for agile software development, simplicity and reusability are of the utmost importance [Agu16].

1.2 Motivations and Goals

Besides supporting the web version of Weaki, the desktop application is planned to extend its features. By combining aspects of development tools such as source code editors and project management platforms, which are already used by the developers, it’s possible to transform the documenting process into a more streamlined and intuitive experience. In addition to that, the integration with the file system and Git empower the user with benefits such as being able to work offline, rollback on changes and having the documentation in a repository, possibly in the same where the code resides.

The vision for Weaki is to support agile methods as best it can and to enable anyone, be it a product owner or a developer, to consume and contribute to the documentation of the project by being as simple as possible.

---

1\(^1\) [https://www.dokuwiki.org/dokuwiki](https://www.dokuwiki.org/dokuwiki)
2\(^2\) [https://git-scm.com/](https://git-scm.com/)
1.2.1 Weaki Design Principles

The main design principles of Weaki merge the intentions of agile methods into the application itself.

Typed Pages  Pages have a type with semantics attached. This makes it possible to programmatically use contents of a page in useful ways, like filtering what is shown according to who’s viewing the page. Besides these immediate benefits, this design principle is the foundation for all the remaining.

Inheritance  By having the option to inherit another page it’s possible to share the same structure and semantics thus boosting reusability. In addition to that, much like object-oriented languages, there are other benefits like type coercion and the ability to express relations between pages.

Weak-typing A page might be compatible with many types. For example, a page of a specific type is automatically compatible with any type higher in the hierarchy. Besides this simplest case, there might be others in which another type with a similar structure makes more sense to that page. In all these cases Weaki has the ability to coerce types or, in other words, assign a different type to a page without changing its content.

Dynamic typing The relationship between the contents and structure of a page is not enforced but instead emerging. In other words, the user can freely type the contents in any way it sees fit and the application automatically, in run-time, assigns the type that makes the most sense to that page.

Everything is a page An important philosophy that makes everything more consistent and predictable with few or none exceptions to the rule. Like operating systems such as Linux where “everything is a file” having one good way to do something is better than having multiple bad ways of doing it.

1.2.2 Weaki Features

The previous design principles make these features possible which intend to simplify the user experience and provide means to maximize reusability thus boosting productivity overall.

Scaffolding  At the moment of a page’s creation, Weaki provides an initial skeleton for the type chosen by the user. This means the user can very easily understand what needs to be done thus lowering the requirements to start being productive and contribute to the documentation.
Introduction

**Content-assist** Having every page coupled with a type makes it much easier to accurately suggest content depending on the context. For instance, the application can provide the user with patterns followed by other contributors for similar types of pages in an attempt to homogenize the wiki’s contents.

**Snippets** With content-assist available there’s basically no reason to not support the use of *snippets* which, much like scaffolding, abolish the need for performing the same task repeatably and promote reusability.

**Code transclusion** Previously, code had to be included either by copying and pasting it directly on the page or by referencing a static document. With git integration there’s the possibility of having a much stronger type of reference, a versioned reference. This means the user can pinpoint to a specific commit or to the latest one available and see it automatically included into the page not having to worry about redundancy or outdated pieces of code.

**Team awareness** Every page has authors, editors and even readers. Weaki promotes transparency with these roles with the intention of promoting team cooperation and discussion.

**Structured views** Sometimes it might be useful to hide certain parts of the documentation that might just confuse the reader. For example, the product owner will not benefit from excerpts of code embedded in a wiki page. Depending on the role of reader some parts of the wiki should be hidden and for that there will be structured views.

**Time labels** Much like commits, each change to the documentation through Weaki will be logged and labelled with the current time.

**Keyboard shortcuts** By now having Weaki separate from the browser there’s a big opportunity to add keyboard shortcuts to all kinds of operations and offer the ability to customize them as the user sees fit. This is one of the most benefiting features for a power user.

**Themes** Requiring an installation on the desktop also means further customization settings are an option. Weaki will have a fully customizable interface made possible by the technology stack used.

**Plugins** A plugin system will be developed so that anyone, including the initial developer, can contribute to Weaki without having to change the source code itself and be able to share it with the community. A plugin might be a simple theme, a collection of shortcuts or even a completely new feature configured in a separate window in the application.

---

3 a small piece or brief extract
4 a knowledgeable and sophisticated user of computers
**Introduction**

**Team Chat** Communication is crucial on any team and Weaki offers that in the form of an integration with Slack, an extremely popular chat application for corporate and personal use.

**File Storage** The ability to include images and files in the documentation is important and supported by linking a file on the repository but there might be times where a file outside of the repository could be included. In these situations Weaki offers a solution by integrating with Google Drive and Slack thus easing the process of uploading a file and linking it.

**1.2.3 Methodology**

The development process follows the agile mindset but not an agile method in specific. Iterations are 1 week-long, as demonstrated in *Table 1.1*, split into three phases: feature development, functional testing and, lastly, review and deployment.

<table>
<thead>
<tr>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
<th>Saturday</th>
<th>Sunday</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature</td>
<td>Feature</td>
<td>Testing</td>
<td>Feature</td>
<td>Feature</td>
<td>Testing</td>
<td>Review</td>
</tr>
</tbody>
</table>

*Table 1.1: Iterations*

There’s a 3 day cycle in which two days of feature development are followed by a day of functional testing. This requires every unit of work to take no more than 2 days of effort to make sure that it’s ready for testing the next day. These small iterations make sure that no amount of work goes to waste and that any bugs and issues with features can quickly be detected.

The days for functional testing do not cover unit testing which is done during the feature development but instead make sure that the requirements are met. This is a good time to detect issues and note them for future handling on review days.

Finally, on the last day of the iteration, there’s a review session to what was done and what can be done to fix the new issues, if any have been detected. Then, all the artifacts, such as executable files and reports, are updated accordingly and deployed on their respective target locations.

**1.3 Document structure**

Besides the current chapter which briefly introduces the reader to the context, motivations and goals there are other 4.

- Chapter 2 provides the research findings for tools in industrial settings such as source code editors, Git, wikis, team communication tools and project management tools.
- Chapter 3 provides an overview of the application, including its interface and features.
- Chapter 4 includes a thorough description of the architecture and its challenges.
- Chapter 5 summarizes what was done, compares to what was initially proposed to do and what is planned for the future.
Chapter 2

Wikis and tools

This chapter presents the research findings for tools in industrial settings such as source code editors, Git, wikis, team communication tools and project management tools. Besides this, it includes an explanation and justification for the chosen framework for Weaki, Electron.

2.1 Electron

The Electron framework was developed by GitHub and it allows the creation of cross-platform desktop applications using web technologies. Supported by the Node.js run-time and the Chromium web browser it can use HTML, CSS and JavaScript to run applications natively on a desktop [Git17]. It was originally called Atom Shell and was developed specifically for the source code editor Atom which is one of the most popular available today alongside Visual Studio Code.

This framework was chosen for Weaki due to its open-source nature, cross-platform capabilities, web-based technology stack and its recent climb in popularity. Being a framework based on Node.js, which uses NPM, there are many community made packages (393,036 available [DeB17] at the time of writing) that can be used.

chai  This package enables the use of the Chai BDD assertion library which is perfect for unit-testing the features developed throughout the project.

mocha  A package that allows the use of the Mocha test framework which is based on asynchronous calls and has very good reporting capabilities like code coverage and test durations.

electron-compile  Enables the use of EcmaScript 61, Less2 and Pug3 on Electron-based applications by automatically configuring the build pipeline to support these technologies.

---

1 the latest specification for JavaScript
2 CSS pre-processor that adds features to the language
3 a template engine for HTML
Wiki and tools

2.2 Source Code Editors

There are many source code editors available for all different type of requirements. Some are the best at small changes to a single file with little overhead when loading like Sublime Text 3 and others excel at project development with many powerful features like code refactoring\footnote{the process of restructuring existing computer code} like Visual Studio.

<table>
<thead>
<tr>
<th>Editor</th>
<th>Refactoring</th>
<th>Embedded Git</th>
<th>Plugins</th>
<th>Electron-Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visual Studio Code</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Atom</td>
<td></td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Visual Studio</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>NetBeans</td>
<td>X</td>
<td>X</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>IntelliJ</td>
<td>X</td>
<td>X</td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Eclipse</td>
<td>X</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>Sublime Text 3</td>
<td></td>
<td></td>
<td></td>
<td>-</td>
</tr>
</tbody>
</table>

On the list of editors in Table 2.1 only Atom and Visual Studio Code are based on Electron. Every single one of these editors have a system of plugins which makes it a good feature to include in Weaki. Not every editor supports Git with the default installation but all do with plugins, an immediate justification to include a plugin system. Finally, only Atom and Sublime Text 3 don’t support code refactoring, besides the simple find-replace feature, which is understandable since these two editors are meant for a wide range of languages. The remaining editors can afford having powerful code refactoring options since they mostly support a finite range of languages.

In conclusion, Visual Studio Code is the best option for further research during development since it’s the only one to cover all these aspects and because it’s open-sourced. If at any point there’s a hurdle that seems impossible to overcome, Visual Studio Code can prove otherwise and offer a possible solution to the problem.
2.3 Wikis

The Weaki desktop application is an extension of its web version which consists of a set of plugins for DokuWiki. In Table 2.2 and Table 2.3 there’s a comparison matrix [Cos17] of other wiki solutions in respects to supported syntaxes and common features, respectively.

<table>
<thead>
<tr>
<th></th>
<th>HTML</th>
<th>Markdown</th>
<th>Textile</th>
<th>Syntax Highlight</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>DokuWiki</td>
<td>X</td>
<td>Plugin</td>
<td>X</td>
<td>X</td>
<td>PHP</td>
</tr>
<tr>
<td>TiddlyWiki</td>
<td>X</td>
<td>X</td>
<td>Plugin</td>
<td>-</td>
<td>JavaScript</td>
</tr>
<tr>
<td>PmWiki</td>
<td>Plugin</td>
<td>Plugin</td>
<td>-</td>
<td>Plugi[n</td>
<td>PHP</td>
</tr>
<tr>
<td>MediaWiki</td>
<td>X</td>
<td>Extension</td>
<td>-</td>
<td>-</td>
<td>PHP</td>
</tr>
<tr>
<td>TikiWiki</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>PHP</td>
</tr>
<tr>
<td>ProntoWiki</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>C#</td>
</tr>
</tbody>
</table>

Table 2.2: Wikis Syntax Comparison

MediaWiki is by far the most popular on this matrix since it’s used on wikis like Wikipedia and yet it’s not the most feature-rich. In terms of syntax it only supports HTML and partial Markdown with an extension. These comparisons were made to check for features not supported by DokuWiki, since it’s the one Weaki is based on, and with the exception of forums support there is no relevant feature that it is not included. The Textile syntax is only supported by DokuWiki while HTML is supported by all and syntax highlight by most. Still, ease of use and interface design are questionable for DokuWiki and TiddlyWiki has, subjectively, the most pleasant one. Finally, most of the researched wikis are based on PHP with the exception of TiddlyWiki and ProntoWiki that use JavaScript and C#, respectively.

<table>
<thead>
<tr>
<th></th>
<th>Themes/Skins</th>
<th>Preview</th>
<th>Page History</th>
<th>Templates</th>
<th>Plugins</th>
<th>Text Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>DokuWiki</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PmWiki</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>TiddlyWiki</td>
<td>X</td>
<td>X</td>
<td>Plugin</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MediaWiki</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>TikiWiki</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>ProntoWiki</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2.3: Wikis Comparison

In terms of relevant features these wikis are far less apart in comparison to syntax support. All are visually customizable, are able to preview the output and have page history which are all features that are supported by Weaki. Then, all but ProntoWiki support templates and plugins which, like was mentioned previously, greatly benefits the desktop application for Weaki. Finally, only DokuWiki and PmWiki and TiddlyWiki use text storage for their pages.

In short, DokuWiki supports all the relevant features and syntaxes including all the benefits from the Weaki plugin set. Besides the differences in usability and interface design there is not much to take from the other wikis into the desktop application for Weaki.
2.4 Project Management Tools

Weaki is meant to be used by agile teams and these teams are probably using a project management tool. There’s much choice when it comes to these tools and Table 2.4 compares some of the most popular.

<table>
<thead>
<tr>
<th></th>
<th>Web-based</th>
<th>Issue Tracking</th>
<th>Team Chat</th>
<th>Git</th>
<th>File Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active Collab</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Wrike</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>GitHub</td>
<td>X</td>
</tr>
<tr>
<td>Zoho Projects</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>GitHub</td>
<td>X</td>
</tr>
<tr>
<td>Trello</td>
<td>X</td>
<td>X</td>
<td>Slack</td>
<td>GitHub</td>
<td>X</td>
</tr>
<tr>
<td>Jira</td>
<td>X</td>
<td>X</td>
<td>Slack</td>
<td>GitHub</td>
<td>X</td>
</tr>
<tr>
<td>PivotalTracker</td>
<td>X</td>
<td>X</td>
<td>Slack</td>
<td>GitHub</td>
<td>X</td>
</tr>
<tr>
<td>Basecamp 3</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Teamwork Projects</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Freedcamp</td>
<td>X</td>
<td>X</td>
<td>-</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Table 2.4: Project Managers Comparison

All of these tools are web-based or supported in a browser which is a good source of inspiration for interface design since the desktop application for Weaki is web-based. Issue tracking is a feature that is supported by all and this could be also supported by Weaki since the objective is team collaboration and issues are bound to happen. With the exception of Freedcamp all the project managers have team chat or an integration for Slack, which is a very popular chat application. Luckily Slack has an excellent web API which makes it easy to build an integration and for that reason it is a good option for Weaki. Most of these project managers support Git and those that don’t support it natively have plugins to that end with GitHub being the common hosting service across all the tools. Finally, file storage is offered by all these platforms which is not something that Weaki is for, but it can easily integrate with storage platforms like Google Drive and even Slack.
2.5 Conclusions

Most source code editors researched embed git natively and support refactoring and all of them have a plugin system. A plugin system makes a lot of sense since these are tools used primarily by software developers and there’s the tendency for making their own plugins if a feature is missing or badly implemented. This can be brought into Weaki for the same reasons. Both embedding git and refactoring are features that fit in Weaki and can increase its value. Having a version control system means having a well-documented page history, rollback capabilities and a repository to include the wiki. Since Weaki uses typed pages, refactoring also becomes a possibility and something the user can benefit greatly from.

The web version of Weaki is based on DokuWiki which is one of the most feature-rich wiki applications. The research performed on other wikis showed that there aren’t relevant features DokuWiki doesn’t support but there are some differences in usability and interface design that can be taken into account.

All of the researched project management tools are web-based which makes them a very good source of inspiration and guidance for interface design. Most of these tools are services paid by companies, so their quality is somewhat assured. All of these platforms have some way of tracking issues which is a good feature for Weaki since it’s meant for team collaboration and issues are bound to surface. Most of these tools support team chat and if not natively then by integrating Slack which is an excellent feature for Weaki as a way to boost team communication. The simple and well designed web API for Slack makes it an even more appealing option. Only Active Collab supports git natively and the rest that support with a plugin integrate GitHub. This turns GitHub integration into an attractive option for Weaki since git support is already planned, there is an opportunity to use GitHub’s issue tracking system and its file storage capabilities can also be leveraged since all the project managers offer native file storage. Besides GitHub, there’s the option to integrate Google Drive and even Slack as an alternative for file storage.

By merging the best parts of all these different tools for agile software development teams, the desktop application for Weaki can maximize team efficiency with its communication capabilities, varied ways to reuse previous work and its familiar feel for developers.
Wikis and tools
Chapter 3

Application Overview

3.1 Interface

The application’s interface is split as seen in figure 3.1 with each section having the name of the component occupying it. Both the left and right components can be resized by the user with the mouse by dragging their right and left bounds, respectively.

Figure 3.1: Interface overview

The top menu includes all the available commands which can be triggered by clicking on them or pressing the keyboard shortcut bound to each one.

The Explorer component on the left is where the user can see the project’s files and folders and open them with a click on the mouse.

The StatusBar component on the bottom shows context data like the path of the currently opened file.
Application Overview

At the centre there’s the **Editor** component which is where the user can edit files with the help of Weaki’s tools.

The **Router** is a special component which shows different components based on a location much like a URI on a browser identifies a resource like a web page. By default, the registered locations are the following:

/\history  Shows all the existing commits where the current file was modified and with a single click restores their state at the time. The default shortcut to bring this into view is **Control+Shift+H**.

/\git/commit  Shows the currently staged files and enables the user to quickly commit with a specific message. The default shortcut to bring this into view is **Control+Shift+G**.

/\preview  Shows the result of compiling the current file as Markdown into HTML as the user types. The default shortcut to bring this into view is **Control+Shift+P**.

/\templates  Shows the existing templates in the repository and offers a quick way to apply specific values to the variables. The default shortcut to bring this into view is **Control+Shift+T**.

3.2 User Stories

The following user stories describe what features the application provides to the end-user and how they work.

3.2.1 Text editing

**Description**  As a user, I want to be able to edit text in the way that’s expected for today’s rich text editors.

![Figure 3.2: Editor component](image)

12
The **Editor** component (figure 3.2) located in the centre panel of Weaki allows the user to write, delete, cut, copy and paste text as any other modern text editor. This entails being able to select text which is possible via the mouse or keyboard through the expected shortcuts. The component automatically shows scroll bars, horizontal and vertical, whenever the content is out of bounds which can be controlled with the mouse. The tabs at the top of the editor let the user manage the currently opened files and check which ones have pending changes, indicated through the little white dot next to the its name. These tabs can be selected with a click on the mouse. To create a new file, shown as ‘untitled’, the user can use the default shortcut **Ctrl+N** and to close it the **Ctrl+W** shortcut.

Another feature which is expected and, with today’s standards, considered necessary is the ability to undo and redo actions which is support in Weaki through the expected shortcuts and optionally via the **Edit** submenu as seen in figure 3.3.
Application Overview

3.2.2 Edit Markdown

**Description** As a user, I want to easily edit markdown text so that my productivity raises.

![Edit submenu](image)

Figure 3.4: Edit submenu

The **Edit** submenu, show in figure 3.4, offers a set of commands for quick markdown snippets including ones as inline HTML like **table**. These snippets can be used on selected text and most of them will simply surround said text but some, like **link** or **image**, interpret it and use it in different ways.

3.2.3 Open files

**Description** As a user, I want to open existing files so that I can edit them.

![Open file menu](image)

Figure 3.5: Open file menu

The **File** submenu contains the command to open file which is bound to the default shortcut **Control+O**, as seen in figure 3.5. This will prompt up a dialog to choose the respective file to open. This dialog changes with each operating system and in figure 3.6 there’s an example of how it might look in a Manjaro distribution of Linux.
3.2.4 Open repositories

**Description** As a user, I want to open an existing repository so that I can use it as a project.

The **File** includes the command (figure 3.7) to open a repository as a whole and is bound to the default shortcut **Control+Shift+O**. Much like the open file command this prompts a dialog (figure 3.8), different for each OS, to choose a folder which must contain, or be part of, a repository. This loads all the files, excluding the ones configured to be ignored, and sets the workspace for the application.
3.2.5 View workspace

Description As a user, I want to be able to view and manage the files which are loaded into the application.

The Explorer component, show in figure 3.9, located in the left panel of Weaki shows the files loaded by the application and sorts them alphabetically and by type (folders first). Clicking on folders collapses or opens them and clicked files are highlighted and opened in the editor.

3.2.6 Git integration

Description As a user, I want to be able to execute some git commands through the application itself so that there’s no need to use the console.

It’s possible to commit inside Weaki through the GitCommit component, in figure 3.10, which has a very simple interface showing the current changes and allowing the user to input a custom message. The component can be opened by using the default shortcut Control+Shift+G or by using the View submenu as seen in figure 3.11.
Application Overview

The Git submenu (figure 3.12) also includes some high level git commands such as push, fetch and checkout.

![Figure 3.12: Git commands](image)

3.2.7 File history

**Description** As a user, I want to restore a file to an older version based on a git commit.

![Figure 3.13: FileHistory component](image)

The FileHistory component located on the right panel of Weaki enumerates the different commits where the current file was modified and allows the user to restore the file’s content at the time with a single click, as seen in figure 3.13. This component can be opened via the default shortcut Control+Shift+H or through the View submenu (figure 3.14).

![Figure 3.14: FileHistory on the View submenu](image)
### 3.2.8 Markdown preview

**Description**  As a user, I want to preview the resultant HTML of a markdown file as I edit it.

![Figure 3.15: Preview component](image)

The **Preview** component, in figure 3.15, allows the user to see current file compiled to HTML as it is changed. This component can be opened with the default shortcut Control+Shift+P or through the **View** submenu, as seen in figure 3.16.

![Figure 3.16: Preview component on the View submenu](image)

### 3.2.9 Templates

**Description**  As a user, I want to use prebuilt templates when building pages so that my time creating one is reduced.

![Figure 3.17: PageTemplates component](image)
Application Overview

The component **PageTemplates** enumerates the available project templates as seen in figure 3.17. By clicking on one of the entries a form appears in which the inputs match the variables inside the template. To apply a template the user clicks on the button below the form which will replace the current text with the appropriate result as the figure 3.18 depicts.

![Templates](image1.png)

**Figure 3.18: An applied template**

To open this component there’s the default shortcut **Control+Shift+T** and the **View** submenu entry (figure 3.19).

![View submenu](image2.png)

**Figure 3.19: PageTemplates on the View submenu**

### 3.2.10 Code Transclusion

**Description**  As a user, I want to reference pieces of code instead of copying and pasting.

```javascript
{app.js}
{app.js@weak1}
{app.js@weak1#B8533}
```

**Figure 3.20: Different types of file references**
Application Overview

It’s possible to reference a file, or part of it, at a specific commit as opposed to copying its contents and posting as plain text. A reference is divided into three parts: **path**, **section** which is preceded by a "@" and **commit** which has the character "#" as a prefix, the last two being optional. If no section is provided then the whole file is taken into account and if no commit is present the latest one is used. Examples of references mixing these parts can be seen in figure 3.20.

![Figure 3.21: Previewing a file’s content](image)

The user can then open the file by clicking on the reference while pressing the **Shift** key. This will open the file at the latest version as to not restore an unwanted older version when all the user wanted was to preview the file. To preview the file as it was in the specified commit the user has two options: hover the reference while pressing the **Control** key (figure 3.21) or open the file and restore it to the same commit via the file history view.

### 3.2.11 Content Assist

**Description** As a user, I want to receive suggestions when typing in order to raise my proficiency.

![Figure 3.22: Content assist example](image)

The application aids the user when writing as to maximize efficiency. For example, when writing a reference to a file, a popup will show up to help selecting the file section. Figure 3.22 shows this feature working on a JavaScript file. Clicking any of the suggestions will fill in the appropriate text.

![Figure 3.23: Content assist auto-complete example](image)

The list of sections will adapt to what the user writes. For example, in figure 3.23 the list is reduced to the sections matching the user input, essentially working as an auto-complete.
Another instance where the popup shows is when the user tries to write a commit hash as shown in figure 3.24.

### 3.2.12 Themes

**Description** As a contributor, I want to easily create colour themes for the application.

The figure 3.25 shows the default colour theme of Weaki. New themes can be created by changing the handful of variables in `/src/styles/theme.scss` and re-compiling the stylesheets. By changing a couple variables it’s possible to very easily change the look and feel of the application. In figure 3.26 there’s an example of a slightly lighter theme with a yellow accent.
3.3 Configuration

Weaki has global and project specific configurations. A folder named **configs** can be found in the installation folder and contains two files: **.weakignore** and **keymaps.json**. The former contains file globs\(^1\) to be ignored by the application and the latter maps shortcuts into commands. The available key combinations can be found in Electron’s documentation page\(^2\).

For project specific configurations there’s a folder named **.weaki** which is automatically created if it doesn’t exist. This folder can contain another folder called **templates** which in turn contains the project specific templates. These templates are then parsed and presented on the **/templates** view.

---

\(^1\) patterns which specify sets of filenames with wildcard characters
\(^2\) https://electron.atom.io/docs/api/accelerator/
Chapter 4

Architecture

4.1 Technology Stack

Weakly is based on the Electron framework which allows the development of cross-platform desktop applications using languages usually found in web development, such as JavaScript and HTML.

The React framework enables a component-based development on web interfaces by merging behaviour and representation together. A component has its own properties and state and its representation automatically updates when a change occurs. This makes it easy to digest complex data without having to do anything directly to the DOM.

Finally, Electron uses Node as its backbone which in turn allows the use of other packages available in NPM. Packages like fs, simple-git and highlightjs handle some of the features themselves.

4.2 Rendering and Main processes

Electron uses the rendering capabilities of Chromium\(^1\) to display browser windows in what are called rendering processes and bridges the gap between the operating system and JavaScript with Node running in the main process which hosts the actual application. While there can be several rendering processes running there is always but one main process.

In order to use Node modules in the browser windows there must be inter-process communication with the main process. This communication can be explicit by sending messages through the appropriate channels or implicit with the RMI pattern. Components on the rendering processes can acquire a reference to the application running in the main process and call its methods as if it was a local object while in reality messages are sent between both processes. React components which integrate with Git or other operating system tools make use of this.

To obtain a reference to the application running on the main process from a rendering process it’s first necessary to import the module `remote` of Electron. This module provides several ways to

---

\(^1\)Open source browser
create or obtain remote objects from the main process. Then the method `getGlobal`, as the name suggests, permits the attainment of global variables living in the remote object. In the context of Weaki, there’s a global variable named `instance` which holds the singleton instance of the application. From here it’s possible to call the remote’s object methods as if it were a local object.

An example of this use can be seen in figure 4.1 where a component intends to run a command in the main process by executing the method `executeCommand`. The flow of `doWork`’s execution is as follows:

1. A message is sent to the main process indicating the object, function and arguments intended.
2. The function executes and the main process sends back a message with the respective return value.
3. The remaining instructions of `doWork` are executed.

The remote function could even return Promises which would still complete the chain back in the rendering process. This feature is extremely useful and used widely in Weaki.

### 4.2.1 Weaki

The class Weaki located in `/app.js` is the starting point of the application and holds everything required to run in the main process such as all the existing commands and the module file-manager for example.

### 4.2.2 Window

The class Window in `/src/window.js` represents the only browser window in Weaki. Here is where the placement of the components is set along with the management of which component is shown on the right sidebar.

### 4.3 Rich Text Editing

There are two components that allow the text editing feature of Weaki. These are the `text-editor` and `editor`. The text-editor component implements all the low level features of text editing, such as
as copy and paste, and offers support for decorators, suggestors and key mappers. The editor component uses this low level component and builds upon it with file tabs for editing sessions, by specifying the decorators, suggestors and key mappers and listening to global events such as 'application:file-loaded' in order to load the new file into the text-editor.

### 4.3.1 TextEditor

The text that is input by the users is not what is directly shown to them there are decorations, popups and suggestions that may surface while editing. Instead, the text editor parses the input and builds a tree of nodes describing its contents so that it can then use this tree to render the HTML which is what the end-user sees.

Every time the state of the editor changes the tree is rebuilt before the rendering step in order to reflect the changes. The generation process is split into the following stages:

1. Generate the root node which includes all the text
2. Generate the syntax highlighting nodes
3. Insert the highlighting nodes at the root
4. Generate the decorator nodes
5. Insert the decorator nodes at the root
6. Generate the selection and cursor nodes
7. Insert the cursor node into the selection node
8. Insert the selection node at the root

While all the nodes are ultimately inserted at the root this does not mean they remain at the same depth. The insertion process is recursive and tries to find the best parent for the nodes. Much like the cursor node has the selection node as its parent, decorator nodes can be inserted into highlighting nodes or even other decorator nodes depending on some attributes. The main factor deciding the parent is the text range the node occupies. If node A occupies the range 0 to 10 and node B occupies the range 3 to 7 the latter will have A as its parent, as seen in figure 4.2.

![Diagram of nested nodes](image)

**Figure 4.2: Nested nodes**
But there are a few cases where this is not as simple. There may be two nodes that overlap regions of each other. For example, if node A occupies range 0 to 10 and B occupies the range 5 to 15 there is no other solution but to split one of them. The node chosen to be split is always the one trying to be inserted. So, if the node A was already present in the tree, B would be the one split and there would be a new node with the range 5 to 10 inserted into A with the same characteristics as B, as seen in figure 4.3.

This works fine for nodes that do not require state and can be broken, such as class decorators, but not for stateful ones. For example, the decorators for images use their text as the source for the images shown as a popup and it could not work if the node would to be split. For such occasions the decorators can specify whether or not they can be broken via the `breakable` property which defaults to true. For example, if the class decorator A, which is already in the tree, occupies the range 0 to 10 and the image decorator B that occupies the range 5 to 15 is inserted, the former is the one to be split because a class decorator is breakable and an image decorator is not, shown in figure 4.4.

If there is an occasion where there is an overlap and both nodes are unbreakable, the node being inserted is simply rejected, as figure 4.5 shows.
Architecture

These nodes when rendered include their ranges in a special property named \texttt{dataSet} on the corresponding DOM elements. This property ensures the data in it will not be wiped whenever the DOM needs to be updated. This made possible to take advantage of the native support of text selection with a cursor instead of listening to all events and figuring out where exactly the cursor is in relation to the text. Upon selecting text with the cursor it’s possible to obtain said selection which contains the start and end DOM nodes. Now it’s only necessary to determine which ranges of text these nodes contain which can be obtained through the previously mentioned property \texttt{dataSet}. With this data it’s possible to construct a more appropriate selection object which is then used for rendering purposes and when using the keyboard to transverse the text.

4.3.2 Decorators

It’s possible to decorate specific ranges of text with new colours, popups or even bind actions to the mouse by extending the base class Decorator in \texttt{/src/decorators/decorator.js} and passing it to the TextEditor component.

A decorator can be something as simple as applying a CSS class to span of text by simply providing the static property \texttt{regex} which identifies the ranges of text and the default prop \texttt{class-Name} for the class. For more complex behaviours it’s possible to use the state properties \texttt{popup}, \texttt{popupVisible} and \texttt{containerProps} of the component. The \texttt{popup} property contains the element to be shown as a popup, as the name suggests, while the property \texttt{popupVisible} controls its visibility. Lastly, the property \texttt{containerProps} contains all the properties passed directly to the DOM element serving as a container.

The image decorator, for example, parses the text to be used as the source for the image which shows as a popup. The popup is visible when the mouse is hovering (figure 4.6) and hidden when not (figure 4.7).

![Figure 4.6: Image decorator when hovering](image.png)
4.3.3 Suggesters

Content suggesters fuel the content assist feature by interpreting the current text, line or word and offering suggestions.

4.4 Modules

The application maintains singleton instances of the following modules which can only be used on the main process.

4.4.1 FileManager

This module is used for file system access with asynchronous operations such as readFile and readDirectory which make use of Promises on top of the Node’s module fs. In addition to the usual methods expected to use on the file system the module also has multiple events, like FileChange and FileAdd, that can be watched. While the fs module does offer similar events it can’t be trusted as it is plagued with bugs and issues. The package chokidar is used to remove most of these issues and simplify the event management. One of these events, FileChange, has a flag named isExternal indicating whether the change was made via the FileManager module or external sources such as git. This is useful for the Editor component, for example, to determine if there should be an update on the displayed text which should only happen if the file was modified externally.

4.4.2 Git

This module is a collection of git operations, such as commit and push, all of which are asynchronous with the use of Promises on top of the simple-git package.

4.4.3 FileInterpreter

In order to be able to reference specific sections of a file there’s the need for file interpretation which is handled by this module. Internally this module contains several code interpreters arranged by type which are then used when the wanted file is loaded. In summary, the module itself does not actually interpret the code inside the files but does offer the structure for code interpreters to register themselves for such use. To interpret a file its path is required and optionally its type and a commit hash. If no file type is provided the interpreter tries to match the file extension, if it exists,

---

2 These issues can be seen in nodejs/node GitHub page on the issues section

---

28
with the corresponding code interpreter. If no code interpreter is found an error is thrown. If a commitment hash is provided, the corresponding version of the file is offered to the code interpreter or else the most recent version is offered instead.

Since the files are loaded by commit, the latest if none provided, their interpretations can easily be cached by the commit hash. This way, the file system is only accessed once per file and commit.

4.4.3.1 CodeInterpreter

The available code interpreters are located in /src/code-interpreters and all extend the base class CodeInterpreter. This base class has one method named getSections which receives text as its parameter and returns sections which have names and start and end indexes. The reason why it is called getSections and not getMethods or getClasses is due to the fact that there are languages where there are no methods or classes, like Markdown. An actual implementation of a code interpreter must decide for itself what it considers a relevant section. Taking JavaScript as an example, a class might be considered a relevant section much like a function or a class method. In order to specify the types of files the interpreter is intended for there are the two static properties named fileType and fileExtension which can be either strings or arrays of strings. These are used by the module FileInterpreter when selecting the code interpreter to be used for a file.
Architecture
Chapter 5

Conclusions and Future Work

The study of the technologies used by software developers introduced a new vision into what is useful in Weaki. By merging the feature-rich environment of a source code editor, the team collaboration aspects of project management software and the familiarity of tools such as Git and Slack, Weaki could improve the efficiency of agile teams on documenting their work.

While unfortunately it wasn’t possible to implement all the initially planned features, a good support for extra development is in place. The features included in the final version allow a more efficient workflow than using a plain text editor and prevent having to use the command line for things like Git. Versioned references are extremely useful when documenting code and Weaki makes good use of them.

5.1 Contributions

Agile software documentation tools for the desktop are not too common and for this reason Weaki can be a good starting point. One of its strong suits being the technology stack which allows the use of widely used languages lowers the entry barrier for contributions for this open-sourced project. In addition, the architecture eases the creation of useful components with the use of the Router component, the decorators structure, code interpreters and suggesters.

5.2 Future Work

The major features that could not be implemented in time are included in this section. While these weren’t included in the final version they were very much thought out beforehand.

5.2.1 Typed Pages

In order to support typed pages the file should not be saved as plain text. Instead they should be saved as a description of the type they extend defaulting to a base type, if none. For example, when the user selects one of the available templates, applies the appropriate values and then saves, what would be stored could be something similar to the listing 5.1 below.
Conclusions and Future Work

Listing 5.1: Example file format

```json
{
  template: "class_descriptor",
  variables: {
    className: "ImageDecorator",
    shortDescription: "Shows an image as a popup",
    location: "/src/decorators/image-decorator",
  }
}
```

This way, the information of the template and its variables isn’t lost whenever the application is closed although opening these files in other programs but Weaki would start to make less sense. When opening these files in Weaki the application would then parse the templates, replace the variables and present the file to the user as it was in plain text.

Inheritance would require more information to the templates but would allow much more reusability. Imagining that listing 5.2 contains the definition of the class-descriptor type, an extension of it could be something similar to listing 5.3.

Listing 5.2: Example base type format

```json
{
  extends: null,
  text: "#{className} - {location}"
}
```

Listing 5.3: Example extended type format

```json
{
  extends: "class-descriptor",
  overwrites: {
    className: "{{className}}.{{functionName}}",
  },
  before: "[Go back] (/../)\n",
  after: "\n---\n{{shortDescription}}"
}
```

The overwrites property enables the replacement of specific parts of the base type. In this case it extends what is shown next to the className variable. The remaining properties, before and after, prepend and append text to the base type, respectively.

32
Conclusions and Future Work

5.2.2 Refactoring

With the typed pages feature complete, refactoring would require nothing more but to change the original template and prompting a refresh on the file’s display.

5.2.3 Structured Views

A view filters what is seen in a template and could simply be stored as ranges from the original template which are visible (listing 5.4).

```json
{
    "name": "productOwnerView",
    "template": "class descriptor",
    "ranges": [{ "start": 0, "end": 10 }, { "start": 25, "end": 50 }]
}
```

Listing 5.4: Example view format

When applying the view the application would have to first determine which variables are included in the ranges and show them accordingly.

5.2.4 Future Integrations

Integrations with platforms such as Slack, GitHub and Google Drive can easily be created as separate components and then registered on the Router component with a corresponding location. For example, to show the current issues on the repository a new component could be created and registered on /github/issues which would then launch the appropriate requests to GitHub and present the information. All this without requiring any extra change on the application itself.
Conclusions and Future Work
References


