Static Site Generator

La Salle Campus Barcelona

Javier Mérida

12 02 2023

Contents

Chapter 1

Trying out popular SSGs

The following SSGs has been chosen based on their popularity in order to be tried to gather information on what the basic features for SSG are. It's important to notice that these are being tried in a Linux environment (Linux inside Windows, thanks to WSL), installing them from scratch, using vim as text editor to minimize external tools assistance to have an unbiased appreciation of each one of them.

- NextJS (https://nextjs.org/)
- Hugo
- VuePress
- Eleventy
- Astro

1.0.1 NextJS

NextJS is a NodeJS with ReactJS based SSG framework, meaning that in order to use it's needed to install the following dependencies.

1. First, install NodeJS: `sudo apt install nodejs`
2. Then, install npm (Node package manager): `sudo apt install npm`
3. Finally, using npm to install NextJS: `npx create-next-app@latest nextjs-blog -use-npm`

This command will install the additional dependencies required to develop a NextJS application, such as React. The pages of this SSG are located in the /pages folder, and each page entrypoint has the same name as its corresponding file. As a result, routing is handled automatically and everything is well-defined within the project.

There is no need to switch to a different file to view the routing pages or the names, as one folder contains all of this information.

In addition, the fundamental feature of NextJS is that it uses React as a foundation framework to generate content via components, and the page itself is a React component, so it's really simple to comprehend and begin working with if you' re already familiar with React.

The styles, on the other hand, are covered by basic CSS files in a styles/ folder, as is customary in web development.

Overall, I found NextJS to be incredibly user-friendly, simple to design, utilize, and scale, and well-structured.

1.0.2 Hugo

Hugo is a highly efficient, portable, and easy-to-use content management system that comes with a variety of design themes.

As the testing environment is based on the Linux Debian distribution, one of the installation methods described on their main website was chosen for this test, which was sudo apt install hugo.

However, it is important to note that there are multiple methods and sources for obtaining the Hugo framework, and each has its own pros and

cons, which are not discussed in this document because its primary purpose is to examine the framework's primary features after installation.

To get started with Hugo, simply execute the following command `hugo new site new_site`. Hugo will create a folder with the name of the project, in this case `new-site`, and a new theme can be added to this folder for styling. Notice that there are a large number of already-created and freely-distributed themes, which saves you development time on styles and aesthetics, you could install a theme (using `git submodule add [REPOSITORY NAME]`), add it to the Hugo configuration file `config.toml`, and focus on adding content to the website.

Nonetheless, it is essential to recognize Hugo strategy's limitations. The first is the existence of paid themes, which creates a financial gap within the open source community. This is understandable, however, as adding themes and structures requires time, effort, and a certain level of understanding of the dynamics of user design and experience.

The second drawback is the added difficulty to modify a theme as the framework relies on **overriding** to change specific configurations on styling, which obviously violates the SOLID principles in programming as per the inability to extend these configurations wihtout overriding/changing the entire file, although this is not always applicable on styling due to the intrinsic nature of how these are defined and approached in Web via CSS, however there is either a straightforward method to perform tweaks to specific values, such as exposed variables which can provide the ability to make changes within the same authored skeleton or component. For instance, exposing a variable that allows the user to specify a desired padding between list entries, or a desired color for all secondary buttons.

1.0.3 VuePress

VuePress is self-defined as a minimalistic static site generator, powered by Vue to build the theme system, optimized to focus on content creation such as technical documentation via markdown files for metadata, content, configuration, and more. It renders pages with static HTML and turns the page

to a Single Page Application.

In order to use it, NodeJS and Vue are required. Once these dependencies are up-to-date, one can start creating and working on a site by using the Node CLI command `npx create-vuepress-site [optionalDirectoryName]`, and it will ask for further information on the site such as project name, description, and more.

Once done, it will create a basic documentation site under the `docs` folder. In order to see the website live, the Node modules within `docs` must be installed: `npm install`, and then it can be served locally doing `npm run dev`.

The development experience with VuePress is generally undemanding, as it handles all tasks required to build a website, leaving the emphasis on the content creation via markdown files, and the static content (images, videos) integration is also seamless, as all that is required is to leave them in a given folder and specify the media path within the config markdown, while the theme handles all styling and structure.

However, the themes depend on the Vue framework, which adds an additional layer to the learning curve, and the theme styling can be exposed as variables following the config structure, although the developer has total control over how this structure is organized. The issue with this approach is that it makes it difficult to learn different themes, as each author is free to organize them as they see fit, and one must put in the time and effort to comprehend the theme in order to make further customizations. The lack of established and opined structure may result in disorderly configurations, but this is the price of such freedom.

After using VuePress and NextJS, it is important to note that using NodeJS as a package and dependencies manager increases the developer's pain and work load. NodeJS is notorious for causing pain among developers due to the complications it can lead to when a dependency is not met or is out-of-date, as the solution is never straightforward and further investigation is required to determine what could be happening in t. During this testing, conflicts between the NodeJS version and the VuePress required dependencies caused a number of issues.

1.0.4 Eleventy

Eleventy is a highly adaptable website site generator that enables you to create a website using a variety of template languages with a multitude of settings and pre-installed features to enhance and personalize the working process and final results.

It works with JavaScript, specifically NodeJS, making its installation a primary requirement. Once it's available in the environment, use `npm` to install it by doing `npm install @11ty/eleventy`.

In order to start working on the new project, it's required to create a template file, which can be accomplish by using HTML, JavaScript, Markdown, Nunjucks, and more. Once a template is already established, create the content, called index file which can be done using markdown, and the content of this file will be mapped to the specified location in the given template, generating the output files following this structure.

Eleventy is easy-to-use, framework agnostic as it does not rely on a specific framework, unlike Next.JS and VuePress, and comes with a plethora of features, template languages, flexibility, and the ability to add multiple well-known plugins and third-party tools, such as Sass for styling, to enhance the development experience. This flexibility enables the developer to begin working on the site with the specific set of desired tools, without additional boilerplate or the burden of learning new technologies, while also adapting itself to the needs of the site to be generated, i.e. bringing the ability to use and implement already known tooling to address very specific situations in accordance with the industry standard.

Nonetheless, it is well-known that this flexibility increases the cognitive load associated with starting a new project, facing a blank sheet of paper (or in this case, a blank IDE), and deciding which tools to use to optimize both the work process and the final product/output. The absence of an established structure can be problematic when deciding which features to include, and it makes it more difficult to work on a project that has already begun because the tools used may be entirely different from those used on a previous project.

1.0.5 Astro

Astro promotes itself as an all-in-one web framework, as opposed to a simple static site generator, despite serving the same purpose with full batteries features specialized to focus on content creation and management rather than pure development, and designed to be fast and performant.

To utilize it, simply visit their website (*http://astro.new*) to play with a fully functional browser version of Astro. Similarly, to use it in a local environment, NodeJS and `npm` are required, and a new project will be created using `npm create astro@latest` to begin development.

In contrast to other Single Page Application frameworks where everything is loaded at client-side, and most server-side render applications where it takes a considerable amount of time to load the page when the project starts growing, Astro uses a Partial Hydration, which means that the content (which is always, or at least it will try to always render it in the server) is only loaded when it is needed. For example, some content can be loaded concurrently with scrolling once it becomes visible in the browser.

This method, in conjunction with the island architecture, which divides the website into distinct sections, enables the framework to have significantly improved performance compared to other methods.

Therefore, Astro focuses on the serving sites' performance, allowing the developer to focus on content creation without worrying about execution details, while providing several features to customize the serving process and also enabling the customization of the working environment as it has hundreds of integrations to choose from, such as front-end frameworks (being framework agnostic in the sense that it does not rely on any particular tool), preprocessing toolings, such as Sass, and so on.

Finally, Astro is a highly effective framework with numerous customization options from which to study. It allows for a great deal of freedom and flexibility while remaining simple to use. Nevertheless, for a simple static site generator it may be an overkill tooling system to use, as it is designed to compete against single page applications with their multiple page application

approach, and it lacks an opinionated or strict structure to adhere to, which makes sense as it must fulfill the needs of a complete website.

1.1 Comparison Table

The following table outlines the features that has been considered as most important to take into account when considering static site generator. Note that the purpose of this document is not to benchmark any of these features, thus there is not a workload test to compare the performance of the explored SSGs. Instead, this is a simple exploration on what features are stand out in order to consider as requisites to build a new SSG.

Also, it is important to note that the lack or the presence of any of these characteristic do not make any SSG better than others, as each one of them are made for a specific purpose and shine in terms of addressing the obstacle they are meant to. In fact, this section must be taken as an exploration of requisites to be taken into account.

	NextJS	Hugo	VuePress	Eleventy	Astro
Easy to use	Yes	Yes	No	Yes	Yes
Flexible	No	No	No	Yes	Yes
Strict structure	No	Yes	No	No	No
Template system	No	Yes	No	Yes	Yes
Themes	No	Yes	No	Yes	Yes
Performance oriented	Yes	Yes	NA	Yes	Yes
Framework dependant	Yes	No	Yes	No	No
Easy to install	Yes*	Yes	Yes*	Yes*	Yes*
File-based routing	Yes	Yes	NA	Yes	Yes

Note*: The installation difficulty is strickly dependant of NodeJS and how ‘npm‘ addresses any missing dependency, which can be (sometimes) a painful issue to solve.

Chapter 2

Software requirements specifications

2.1 Introduction

2.1.1 Purpose

The primary objective of this chapter, which is referred to as the Software Requirements Specification (from this point forward, SRS), is to define the requirements and, as a result, the goals that need to be accomplished by the Static Site Generator that is described and implemented throughout the entirety of this project. This will be accomplished by providing a detailed explanation of how the system as a whole works as well as the various components that make up the system.

The person in charge of the creation of the SSG, the developer and/or implementer, is the intended audience for this SRS. Given that it outlines the considered implementation details and design specifications, it serves as a guide for the developer to know exactly what to do along with the definition of done for each task and/or component.

2.1.2 Scope

The Static Site Generator described in the document will be known as `VaGo`, and will henceforth be referred to as such. The nomenclature of this tool is derived from its implementation in the Go programming language, as expounded upon in subsequent sections. Additionally, the name incorporates a homograph word in Spanish, "vago," which connotes indolence or sloth in English. This serves as a playful allusion to the minimal exertion demanded of prospective users in generating a static website through utilization of this software.

VaGo is a tool designed to facilitate the creation and distribution of static content on the web. Its primary function is to interpret and translate markdown files into web content, which includes HTML, CSS, and necessary JavaScript. Furthermore, the system will facilitate the implementation of a theming mechanism, which will enable the creation and dissemination of particular style guidelines for reuse and adherence by the content after its translation from markdown. This theming system will also allow for a flexible and open approach to effectuate specific author-provided modifications to the style of the pages, such as the inclusion of particular padding between elements, the adjustment of text size based on the heading hierarchy, the selection of colors, the application of specific button styles, and other customizable features, which will be contingent upon the preferences established by the theme author.

The provision of flexibility enables prospective users of designated themes to make necessary adjustments to suit their individual requirements, without the need of extensively search through the styling boilerplate and navigating through numerous CSS files to locate the particular parameter to modify.

Similarly, VaGo will offer a straightforward command-line interface (CLI) to facilitate interaction with the program to serve the purpose of translating content into static web pages. This will be possible once the content has been provided in the form of markdown, along with theme styling. Additionally, VaGo will enable users (via the command line) to serve the content as websites through a specified port, using Go's native HTTP methods.

Conversely, the system won't offer support for alternative interpretation formats, such as YAML or JSON, nor can be expected it facilitates the translation of Go code to JavaScript or vice versa. Hence, it is not to be anticipated that it would provide backing for JavaScript reactivity, akin to what is observed in other front-end frameworks. VaGo will exclusively restrict its capabilities to the delivery of static content through fundamental and native CSS and JavaScript. Its primary objective is to facilitate the development process, with a singular focus on this goal to ensure optimal performance. This approach aims to alleviate the burden of web development for individuals who seek to share text or images without delving into the intricacies of the field.

2.1.3 Definitions, acronyms and abbreviations.

- SSG: Static Site Generator.
- MD: Markdown files.
- JS: Javascript.
- CSS: Cascading Style Sheets.
- HTML: HyperText Markup Language.
- CLI: Command-line interface.
- Theme/Style theme(ing): A set of established reusable guidelines and rules to provide style to the web content (website) via HTML and CSS for the sake of consistency.

2.1.4 References

The following collection provides a selection of research documents aimed at facilitating comprehension of the foundational principles that motivate the prerequisites for VaGo. Emphasis is placed on elucidating the rationale behind the necessity of these requirements.

- *What is a static site generator?*. Cloudflare. (n.d.) . https://www.cloudflare.com/learning/performance/static-site-generator/
- Khalid, F. S. (2022, April 18). *How to choose the right static site gener-*

ator. GitLab. https://about.gitlab.com/blog/2022/04/18/comparing-static-site-generators/
- Wikimedia Foundation. (2023, August 12). *Static Site Generator*. Wikipedia. https://en.wikipedia.org/wiki/Static*site*generator
- Wikimedia Foundation. (2023a, April 11). *Hugo (software)*. Wikipedia. https://en.wikipedia.org/wiki/Hugo_(software)
- *What is Hugo*. Hugo. (2023, July 13). https://gohugo.io/about/what-is-hugo/
- *Docs. What is Next.js*. Next.js. (n.d.). https://nextjs.org/docs
- *VuePress guide. Introduction*. VuePress. (n.d.). https://v2.vuepress.vuejs.org/guide/
- *Accessibility principles*. Web Accessibility Initiative (WAI) . https://www.w3.org/WAI/fundamentals/accessibility-principles/
- MozDevNet. (n.d.). *What is accessibility?* Learn web development | MDN. https://developer.mozilla.org/en-US/docs/Learn/Accessibility/What*is*accessibility
- MozDevNet. (n.d.-a). *HTML: A good basis for accessibility*. Learn web development | MDN. https://developer.mozilla.org/en-US/docs/Learn/Accessibility/HTML
- Sun, Y. (2019). *Server-Side Rendering. In: Practical Application Development with AppRun*. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-4069-4_9
- Taufan Fadhilah Iskandar et al (2020). *Comparison between client-side and server-side rendering in the web development* https://iopscience.iop.org/article/10.1088/1757-899X/801/1/012136/pdf

2.1.5 Overall description

This chapter comprises several sections that present a series of requirements according to certain demands. These requirements are derived from an analysis of existing tool scopes, previous work conducted on these tools, potential areas for improvement under specific conditions, the current state of web development, and key considerations for enhancing the user experience.

Subsequently, the requirements are presented in a structured manner, delineating the essential aspects of these requirements, including the intended input and output, the format in which these capabilities are being built, and

the precise requisites for each individual functionality.

Therefore, the primary objective of this section is to present a comprehensive list of explicit requirements associated with the functionalities to be developed. These requirements will be supported by detailed explanations and justifications for their necessity. The aim is to establish a solid and comprehensive set of needs, prioritized according to their importance. This will facilitate a better understanding of the development tasks required to consider the project as completed.

2.2 Overall description

2.2.1 Product perspective

`VaGo` is not the first SSG, nor does it claim to be unique or break any industry standards. Instead, it focuses on providing a simple interface for both the content creator (intended to work primarily in markdown files) and the theme author to create and style websites (meant to provide default styles while deciding editable parameters).

In this regard, it is very similar to, and in fact inspired by, HuGo, another SSG written in the Go programming language. Despite this, `VaGo` attempts to adopt a comparable theming methodology while incorporating simplified modification capabilities. This would enable prospective users to reuse and personalize pre-existing themes without the need to delve into CSS files. Furthermore, the system in question lacks a focus on performance and does not aim to rival HuGo in this regard. HuGo is renowned for its exceptional performance in generating and delivering static content.

In contrast to NextJS and VuePress, which utilize React and Vue, respectively, for component creation and usage, `VaGo` does not employ a framework-based component system. In contrast, `VaGo` restricts itself to utilizing only native HTML, JS, and CSS. However, this does not necessarily preclude the platform from leveraging Go to facilitate or circumvent boilerplate when

converting content into the aforementioned web technologies.

Additionally, the product is not intended to incorporate any reactive techniques or external frontend frameworks in order to achieve similar results. The entirety of the content is intended to be loaded and produced on the server side, with no additional features on the client side, unless the user chooses to incorporate them using JavaScript. It is noteworthy that the utilization of opinionated simplicity fulfills the objective of facilitating content creation for users, as they are not required to attend to such particulars. However, this approach is accompanied by the disadvantage of restricted customization options.

Furthermore, it should be noted that initially, there will be no available modules or tools for integration with this particular static site generator (SSG) framework, unlike other frameworks such as Astro. This is because the development of such modules is not a primary focus of the framework's feature development. Additionally, there are no plans to introduce a module customization feature, as outlined in the accompanying documentation.

Ultimately, it is imperative that end-users possess the capability to engage with the product through a straightforward Command Line Interface (CLI) to produce static content and subsequently distribute it online, with minimal parameters and concealed features. Ideally, the system should be designed to facilitate user comprehension and operation without necessitating the perusal of a cumbersome 30-page manual replete with convoluted directives and extraneous verbiage. Thus, it is apparent that a minimal number of commands, typically two or three, accompanied by fundamental parameters such as port, testing mode, and designated source folder, should suffice to initiate the task at hand.

2.2.2 Product functions

This Static Site Generator's primary objective is to convert markdown files' text to plain HTML, mapping the markup elements to understandable web elements in order to maintain the Web Standards' accessibility.

However, it should also be able to offer a theme system built on CSS files that gives the HTML elements styling. This will be made possible by also producing CSS with a set of additional parameters that follow a pre-established structure and should be simple to alter using Go variables.

Last but not least, `VaGo` will have a brief set of parameters to enable communication with the user via CLI, to enable them to create static content (translation of markdown to HTML with the addition of a theme via CSS), and to serve content via a particular port. A set of features for exporting and importing themes from authors whose works are shared online and/or in repositories may be added in the future.

2.2.3 User characteristics

`VaGo` is designed to be utilized by individuals possessing a basic understanding of web development and terminal operations. However, this does not necessarily demand that they must hold a degree in Computer Science in order to effectively employ the system. The primary objective of the system is to facilitate ease of use for individuals with limited knowledge of computer technologies, including academic scientists seeking to disseminate their knowledge, blog posts, papers, and ideas on the internet, without requiring extensive knowledge of web technologies.

Therefore, a computer-based academic background is not needed for this system to be used, as long as there is a bit of knowledge about markdown and how to use a terminal, it should be entirely enough. Henceforth, individuals possessing a greater depth of knowledge would be capable of executing more intricate undertakings by leveraging the available adaptability while adhering to the prescribed limitations.

2.2.4 Constraints

Considering that `VaGo` is programmed using the Go language, which inherently supports cross-platform functionality, the development efforts will be

concentrated solely on Linux to streamline the implementation of CLI features and circumvent potential complications arising from newline interpretation and other cross-platform limitations.

In addition, the system does not account for security considerations related to delivering content over the internet, and therefore it cannot guarantee any level of safety in this regard beyond what is already provided by the Go programming language.

However, while there exist fundamental performance considerations for its implementation, it neither guarantees nor rivals other frameworks with respect to performance, velocity, or optimal resource utilization.

Consequently, the implementation and development of `VaGo` will primarily rely on the pre-existing features of the language. As such, it is not anticipated that `VaGo` will establish its own mechanisms for managing HTTP requests, signaling protocols for handshaking, thread management through Go routines, or other hardware-related dependencies and controls for the host computer. Therefore, any inherent limitations within the Go programming language will inevitably impact the system, and no resources will be allocated towards attempting to circumvent them.

2.2.5 Assumptions and dependencies

The complete functioning of the SSG is contingent upon the availability of the Linux operating system for its execution. Furthermore, the accurate interpretation of Markdown files is contingent upon the existence of well-crafted documents, without which complications may arise.

In addition, it is anticipated that any additional alterations to the styling will be heavily reliant on one's proficiency in CSS. Consequently, it is imperative that these modifications are accurate, as the system will not undertake any measures to verify the correctness of CSS files.

Moreover, in the event that the system is utilized for the purpose of delivering and sharing files, notably static content, it is anticipated that an

internet connection will be accessible to facilitate communication with client-side requests. It is noteworthy that there will be no verification implemented on this communication, nor will there be any specific error handling beyond what is inherent to the language.

Alternatively, the system's potential expansion could be facilitated through the utilization of native JavaScript, thereby introducing interactive elements to the otherwise static content. However, it is important to note that `VaGo` does not assume responsibility for this aspect, and any modifications or augmentations are solely at the discretion of the user.

Vago's proper usage and functioning necessitate a collection of dependencies, including the Go HTTP module for internet traffic serving. These dependencies are managed by the Go modules import mechanism. Therefore, it's assumed that these dependencies are available in the host machine to be used by the system, and a default error message for the missing dependencies will be thrown in case these are not installed, without fancy or added complexity.

2.2.6 Apportioning of requirements

It is possible that in the future, the system may incorporate supplementary functionalities for the development and utilization of Web Components using conventional web technologies, devoid of any frameworks. The inclusion of these components may introduce an additional level of intricacy and consequently enhance adaptability in generating unique elements and segregating styling. This could potentially enable an approach akin to the Astro islands architecture, featuring isolated CSS. However, a noteworthy advantage lies in the utilization of solely standard technologies, thereby eliminating the need for supplementary tooling or increased size to support the same.

In addition, `VaGo` has the potential to provide fundamental capabilities for generating said components, by eliminating the redundant code required to construct a rudimentary web component, and furnishing a platform to associate specific functionalities to them, without necessitating extensive knowledge of JavaScript coding to achieve the same outcome. It is possible to write

the components using Go language with the aid of VaGo, and subsequently interpret and convert them into JavaScript for the purpose of generating the Web Component.

Nonetheless, it should be noted that the act of translating may result in a loss of flexibility due to its dependence on the system's interpretation. Despite this, it may prove to be a suitable solution for a majority of component types commonly found in the industry. To gain a deeper understanding of the current state of such technologies and their relevance to existing needs, a comprehensive analysis of popular frameworks and their approaches is necessary.

2.3 Specific requirements

2.3.1 External interfaces

Users will be able to interact with the system using CLI commands to carry out simple operations that are built into the system. Here, two fundamental commands should be noted:

- `Build`.
- `Serve`: To serve and share content via a specific port.

Additionally, the system ought to alert the user any time a command it has issued contains an error because it is possible for them to accidentally type the wrong command or misspell any of these. (using buidl instead of build). Notably, unlike other systems, it will lack a recommendation system to offer hints of similar words that are actually interpreted as to assist the user in finding alternative commands, even though it will inform the user about the existence of an error that prevented the system from performing the required task.

2.3.1.1 Build

The task at hand involves the conversion of Markdown files into web-based content, specifically in the form of HTML, JS, and CSS.

The program is designed to utilize the Command Line Interface (CLI) as an input mechanism, whereby it recognizes the command name to execute its designated task. Its output function involves generating the HTML and CSS content into an `out` folder, in accordance with the pre-existing Markdown files. It is noteworthy that each MD file ought to generate an individual page to allow for the adaptability of transforming every file into a distinct output for the website.

Throughout the translation process, it is imperative that the user is kept informed of its progress. In the event of any errors, the system should prompt error messages to ensure the accuracy of the MD files, while simultaneously verifying that all necessary parameters for content and styling have been provided. Upon completion, the system should explicitly indicate the out folder path to the user.

Therefore, it should use the following format for each of the steps considered for the translation and building process: [Date][Page Name]: Step, where each one of these detail token corresponds to:

- Date: Specific date and time using the ISO 8601 format 1[1] without specifying the timezone as `VaGo` will limit itself to use the same timezone from the host machine. For instance: *2023-04-09T14: 22:05* should be used to represent the year 2023, month April (04), day 09 at 14 hours, 22 minutes and 05 seconds.
- Page Name: To specify the exact page name for the URL that takes to that specific page, which is the same as the MD file name.
- Step: To specify the building step it's currently on, whether it is identifying the MD tokens, performing translation to HTML and/or obtaining the parameters set for the theming and styling of the given page.

[1] `https://en.wikipedia.org/wiki/ISO_8601`

On the other hand, additional parameters can be added to the command for specific output capabilities. For instance, `no-log` should be using to omitting logging (no prompt message for every building process step) except for error and finalization message. Also, `no-time` should be used to omit the timestamp from the steps, prompting a message with the following structure: *[Page name]:Step*, with the purpose of avoiding overloading logs and make them easier to read.

2.3.1.2 Serve

The second command serves the purpose of keep the system continuously executing, hence no accepting other commands in the meantime, in order to open a port and send the already built files (by means of the previous command, `build`) via HTTP.

It is important to note that this command will exclusively serve content available in the `out` folder, therefore if the previous command hasn't been run, or if there are no files in this folder, then it won't be able to serve any content at all. Because of it, it is required that the user must first build content in order to then serve it.

Moreover, in addition to the behavior noted above, it must output logs to inform the user of the current status of the serving process, starting by indicating as soon as it starts reading content and initiating listening for requests, while also informing of any issue that may occur by prompting error messages.

Once it starts listening for requests, it should inform of any incoming request/response using the following format: [Date]: Sending [Page] to [requester IP], where each one of these token corresponds to:

- Date: Same as the one outlined for `build` command, using the ISO 8601 format without timezone.
- Page: Page name as stated in the MD filename.
- Requester IP: Specific IP address of the incoming IP.

Additionally, an extra (optional) parameter can be provided to outline the specific port the system should open to listen for requests, instead of the default one.

2.3.2 Functions

The following provides a more in-depth explanation of the functional areas, including full requirements.

2.3.2.1 Generate page

2.3.2.1.1 Description and priority

The task at hand involves generating a page document through the process of translating the contents of a Markdown file into HTML. This requires the utilization of appropriate tags to accurately map the entities present within each Markdown block. Priority: HIGH.

2.3.2.1.2 Stimulus/Response sequences

Stimulus: This feature must obtain the markdown files from a `source` folder.

Response: Translate or generate a corresponding HTML file for each one of these MD files, outputting the result pages into an `out` folder.

2.3.2.1.3 Functional requirements

REQ-1: The MD blocks must be interpreted and translated to HTML tags to generate a page accordingly to what the user has described initially. It should, at least, implement the following elements to ensure a fully accessible page as per best practices for web standards:

- Heading level 1 (`#`): Heading 1 (`<h1></h1>`).
- Heading level 2 (`##`): Heading 2 (`<h2></h2>`).
- Heading level 3 (`###`): Heading 3 (`<h3></h3>`).
- Heading level 4 (`####`): Heading 4 (`<h4></h4>`).
- Heading level 5 (`#####`): Heading 5 (`<h5></h5>`).
- Heading level 6 (`######`): Heading 6 (`<h6></h6>`).
- Text: Paragraph (`<p></p>`).
- Bold text (`**text**`): Strong (`<strong></strong>`).
- Italic text (`_text_`): Emphasis (`<em></em>`).
- Blockquote (`>`): Blockquote (`<blockquote></blockquote>`).
- Unordered list (`-`): Unordered list (`<ul></ul>`) surrounding list items (`<li></li>`).
- Ordered list (`1.`, `2.`, `3.`, and so): Ordered list (`<ol></ol>`) surrounding list items (`<li></li>`).
- Code (`` `` ``): Code (`<code></code>`).
- Fenced code block (` ``` ``` `): Preformatted text (`<pre></pre>`) surrounding a code block (`<code></code>`).
- Link (`[text](URL)`): Anchor tag along with the hyperlink attribute (`<a href="URL">text</a>`).
- Image (`![Alt text](URL)`): Image tag along with alt text and source attributes (`<img alt="Alt text" src="URL"></img>`).

REQ-2: It must verify whenever there is an input error or incorrect usage of the MD blocks, and thereby let the user know about these via error messages and halt the translation process immediately.

REQ-3: Each generated page will have the same file name as the input MD file.

2.3.2.2 Theme styling creation and customization

2.3.2.2.1 Description and priority

This feature is responsible for enabling the construction of a customizable theme for styling purposes. As long as the author (the theme creator) enhances these capabilities via exposed variables, it will manage CSS files

that will provide styles to the pages and provide functionalities to perform tweaks and changes to the theme. Therefore, subsequent users of the same theme would be able to make certain modifications to suit their needs, while retaining the already provided set of opinionated styles. Priority: MEDIUM.

2.3.2.2.2 Stimulus/Response sequences

Stimulus: Single or multiple files with preprocessed CSS attributes that will receive the values of the exposed variables to be processed into a final CSS file. Notice that the format of this file is not strictly CSS as it should be noted that it is waiting for the building stage to add the variables, therefore it's suggested to use a temporal file name adding `.go.css` to differentiate it from its final version.

Response: Once processed, the file or set of files will be merged into a single one, following an established structure and adding the exposed variables via Go.

Stimulus: Authored parameters in the form of Go variables. There is also the possibility of exposing such variables via YAML or JSON format, to avoid overwriting Go code.

Response: The parameters are added to the final CSS file.

REQ-1: The exposed parameters should be added within the preprocessed style file using the pattern `${[variable name]}` in the same place where it will be replaced.

REQ-2: These variables are then added or modified within a specific Go module specifically designed for this purpose. Thereby, the author will be able to expose these in this file, and add a default value in case the user decides to not change anything. It should be noted that the system won't take care of the compilation or verification of the CSS file, hence it is completely up to the author to verify whether it is correctly built.

REQ-3: Multiple style files could be added together in a single final one,

providing the author the flexibility to split them into different modules for readability and maintainability purposes, thus these must be specified in a Go module specifically designed for this purpose, or within a main preprocessed style file using the nomenclature `${#module: [file name]}`.

REQ-4: The system must verify and prompt a clear error message when there is a theme variable that has not been set. More specifically, it should display a message using the following structure: "*Error: [variable name] has not been initialized.*", where the variable name will be replaced with the specific variable missing its initialization.

2.3.2.3 Templating

2.3.2.3.1 Description and priority

The proposed system will incorporate a functionality that enables the establishment of a predetermined framework or outline, in addition to the primary template, for organizing the content into web pages. It is noteworthy that the utilization of Go language's templating capabilities is integral to the handling of this task. However, the system must furnish a mechanism for retrieving the primary information components from the markdown files. This will enable easy recognition and referencing of said components in the template, thereby facilitating customization. Priority: MEDIUM.

2.3.2.3.2 Stimulus/Response sequences

Stimulus: Single or multiple HTML files with Go templating variables to be filled with, from which the user can decide to use custom variables (from custom Go code) or the main, already provided variables from the markdown file components (headings, text, bold, italic, lists, etc).

Response: When built, a final HTML file should be prompted with the provided content via markdown files and following the provided structure or skeleton as in the template.

2.3.2.3.3 Functional requirements

REQ-1: The processed format must follow the established skeleton from the template. If there is no provided template, the system must use a base one already provided with the generator.

REQ-2: Each markdown component should be accessible via Go variables following this naming convention:

- Heading level 1 (`#`): `H1`
- Heading level 2 (`##`): `H2`.
- Heading level 3 (`###`): `H3`.
- Heading level 4 (`####`): `H4`.
- Heading level 5 (`#####`): `H5`.
- Heading level 6 (`######`): `H6`.
- Text: `B`.
- Bold text (`**text**`): `Strong`.
- Italic text (`_text_`): `Em`.
- Blockquote (`>`): `Blockquote`.
- Unordered list (`-`): `Ul`.
- Ordered list (`1.`, `2.`, `3.`, and so): `Ol`.
- Code (`` `` ``): `Code`.
- Fenced code block (```` ``` ```` ````): `Pre`.
- Link (`[text](URL)`): `A`, `A.URL`.
- Image (`![Alt text](URL)`): `Img`, `Img.Alt`, `Img.Src`.

It is important to note that the naming convention and HTML elements are identical. This is intentional, as it should direct the user to the correct element the component should be used in, so as to maintain best practices for web accessibility and search engine optimization, avoiding the use of meaningless `div` tags everywhere.

2.3.2.4 Configuration files

2.3.2.4.1 Description and priority

Through the configuration files the user will be able to set up certain parameters that will change the way the system works to adapt it to given needs and setup, such as determine specific input and output folders, theme name, template file name, author information (name, website name, creation date), and others. It's worth noting that these configuration files are not strictly necessary as the system must initiate with a set of given default parameters, which can be overwritten for extended customization. Priority: LOW.

2.3.2.4.2 Stimulus/Response sequences

Stimulus: A Go, JSON or YAML file with the set of parameters to be read from, so the system can use them as variables for its configuration. The type of file to be used will depend on the development time, as reading directly from a Go file or module it's easier and faster to implement than a JSON or YAML file, as it requires a parsing step first.

Response: The system will produce and/or build the static content following these parameters accordingly.

2.3.2.4.3 Functional requirements

REQ-1: The system will use a set of default values if the configuration files are not provided/changed, thus it is not strictly needed for the system to work properly.

REQ-2: The following configuration files must be scanned (found via specific filename) and used for its respective purpose: Theme config (`theme.*`,) to specify styling parameters as well as the theme name, template config (‘ layout.*) to specify the files entries for templating, main configuration (main.*) for the main parameters for VaGo. File extension to be decided depending on development time as noted on previous point.

REQ-3: For each configuration file, the system must be able to interpret and apply the following parameters:

1. Theme config (`theme.*`):

- Theme name (`name`): The name of the given theme. This name will be accessed for further setup steps.
- Parameters(`params`): An open field to add a list of the open parameters for styling.

2. Template config (`layout.*`):

- Entries (`entries`): A list of template filenames to be included in the templating system.
- Final layout (`final`): A nested ordered list of the templates to be merged for the final composition layout.

3. Main config (`main.*`):

- Author name (`author`): The author name of the website to generated content for.
- Input folder (`input`): Input folder to get the files to read content from, prior to the website generation.
- Output folder (`output`): Output folder to write the static generation results.
- Theme (`theme`): The name of the theme to be used for this website.

It is worth noting that these configuration files and parameters are not final, and yet other may get included in the future depending on the needs that may arise.

Chapter 3

Software Design

3.1 Overview

The VaGo Static Site Generator, as presented in this project, is designed to be utilized by fellow developers for the purpose of creating straightforward and user-friendly websites, prioritizing content over any additional functionalities offered by the site. Hence, the paramount consideration lies in the user-friendly nature of VaGo, which allows for enhanced concentration on content during usage.

In this manner, VaGo should possess the capability to parse markdown files and convert them into web pages. Additionally, it should be able to serve this content through a designated port, while implementing automatic routing based on the file name to URL convention. Furthermore, VaGo should ensure that the user remains informed about each step executed by the Static Site Generator (SSG) through consistent and informative logging.

Therefore, the design of VaGo is significantly dependent on and focused on the following essential elements: **Parse**, **construct**, **serve**, **route**, and **log**. This section will now propose a recommendation for the implementation of these components and an approach to construct their interconnection,

in accordance with fundamental programming principles, with the aim of developing software that exhibits both robustness and maintainability. It is important to acknowledge that the content presented in this chapter is merely a proposal, subject to potential changes or modifications throughout the implementation phase. This flexibility is necessary as additional challenges and requirements may emerge.

3.2 Context

As seen in previous chapters, VaGo is not the first static site generator available in the market. Numerous frameworks provide a plethora of features, surpassing the mere provision of static content. These frameworks offer extension capabilities through the utilization of various programming languages, support input format files other than markdown, incorporate plugins, and present additional functionalities.

With that being stated, VaGo does not aim to revolutionize existing practices or challenge the current state of the field through novel technologies or approaches. Instead, it seeks to investigate the potential of developing a static site generator (SSG) independently. This exploration involves determining the extent to which one can achieve desired outcomes while maintaining simplicity. Additionally, VaGo offers users the advantage of incorporating specific features tailored to their projects without necessitating the acquisition of new skills or adaptation to unfamiliar technologies or frameworks that may not fully meet their requirements.

Furthermore, this study aims to investigate the potential introduction of novel elements that deviate from the conventional norms of similar projects. It seeks to evaluate the use of these features and engage in a discourse regarding potential areas for enhancement.

Considering this perspective, it is imperative that the project's development remains focused on minimalism and simplicity, while also allowing room for both maintainability and extension. Therefore, while it may lack the capability to provide plugins like other frameworks, the code base should

possess a high level of comprehensibility and extensibility, allowing for the creation and integration of new features.

3.3 Goals (and non goals)

Following is a list of essential points that must be specified and elaborated upon in this design document. In an effort to isolate the development process on the principal features and prevent wasting time, it also includes functionalities that are not expected to be developed further.

Goals

- The code is characterized by its minimalistic design, simplicity, ease of use, and readability.
- The decoupling of functionalities into several files, or modules, serves to isolate the code according to its intended purpose.
- It is imperative that each module maintains a singular focus on a certain purpose or functionality. Hence, the fundamental functionalities will be divided into one or more modules, namely Parse, Construct, Serve, Route, and Log.
- The code will possess a certain degree of opinionation due to the project's inherent characteristics and software requirements. This design choice limits the extent of customization available to users, hence facilitating ease of use.
- The project's development will be closely aligned with the limitations and recommended methodologies of the programming language.
- Additional features can be incorporated if deemed necessary, following a thorough examination of the rationale behind their implementation.

Non goals

- Extensibility via plugins or external modules.
- Multiple language development. Only the one chosen will be used.
- Integration with other known technologies, except for the ones included in the standard web industry.

- Usage of design patterns that may over-complicate the code base, unless they are strictly required and proven to be useful for the project needs.

3.4 Proposed Solution

3.4.1 Parser

Based on the prioritization of VaGo's features, the initial module to be built pertains to the parsing of markdown file content. Hence, the proposed methodology entails the process of reading, detecting, and storing tokens for various text elements such as titles/headings, links, bold formatting, italic formatting, and others. These tokens are then stored in variables based on their respective content types.

Moreover, in order to enhance convenience, it is imperative to consolidate these variables into a singular variable, such as a map or array, since this approach can facilitate the process of relocation, interpretation, and retrieval. The use of a struct variable type enables the accomplishment of this task, given that the programming language employed is Go.

As a result, via the accomplishment of this task, the users of this module are able to effectively interact by accessing the information included within this variable alongside the information retrieved from the markdown. This extracted material can subsequently be utilized to construct the pages in accordance with a specified template.

In this manner, the suggested architecture will incorporate the designation `Out` to facilitate the identification of its intended functionality. The primary function responsible for extracting the content will be denoted as `ParseMarkdown`. This function will accept the markdown file content as a parameter and ultimately yield an instance of the `Out` struct, populated with the parsed content.

3.4.2 Constructor/Generator

After the extraction and parsing of the material, the system is required to identify and interpret the template files provided by the user. This process enables the generation of HTML pages that incorporate the extracted text. Hence, the chosen directory for input files will be iteratively accessed within a loop, with each file being processed according to the provided specifications to generate the corresponding pages.

The utilization of the Out structure within the templating system allows users to determine the appropriate structure by adhering to the pre-established naming convention for the Out variables. After the establishment of this configuration, the system will utilize these templates in accordance with the previously outlined methodology.

Therefore, the system will not only retrieve each input file, analyze its content, extract the templating, and generate the newly parsed HTML content, but it will also generate new files specifically for this purpose within the designated output folder. The encapsulation of the described functionality will be implemented within the `Build` function, which will accept the input folder location, template file, and output folder as parameters.

Both the Parser and Generator will work together in the same phase named as "Build", which consists mainly on taking the Markdown content and template in order to build the HTML content. This same name will be then used by the CLI to indicate the software to perform these actions.

The image belows provides a graphical explanation about the interaction between these modules along with the input folder, template and output folder.

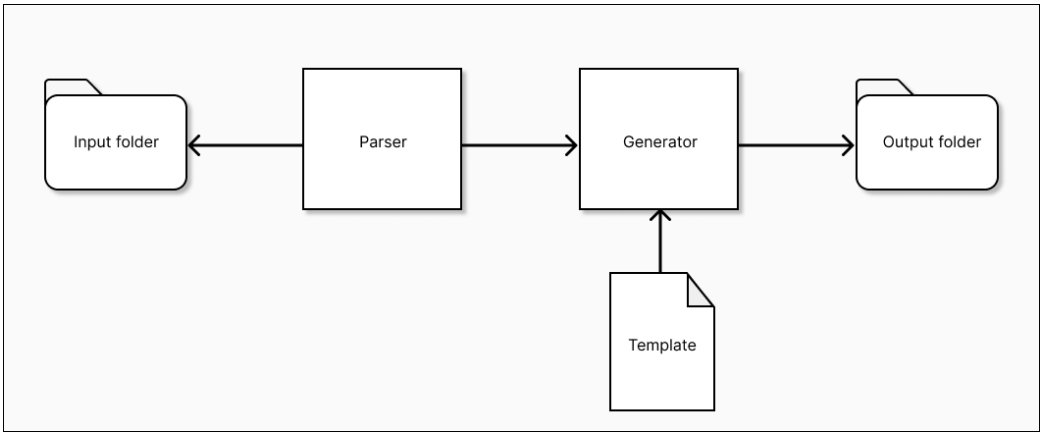

Figure 3.1: Build process using Parser and Generator modules.

3.4.3 Server

This module is pretty simple as its main purpose is to open a port on localhost to start listening to requests, which will be then managed by a router using the `/[target]` URL format.

Moreover, the function `Serve` will be in charge of exposing this behavior to start serving files, with the port number and the folder with the content to be sent as parameters.

3.4.4 Router

The basic objective of the system is to efficiently handle incoming requests by receiving a designated folder containing the content to be served. It then automatically generates and delivers the appropriate pages based on the file names and the corresponding target URLs. In order to do this task, it is necessary to iterate through the files included within the designated folder. During this process, the program should identify each file name and afterwards locate the file that corresponds to the requested URL. Once located,

the information will be delivered back to the IP address of the individual making the request.

If a file name is not discovered, the system should generate a 404 page. This page can either be provided by the user in advance or a default page can be used.

Similarly to previous modules (generator and parser) used in the Build stage, Server and Router are meant to be used together to accomplish another task: Serve. This will be on charge of managing every incoming HTTP request and handle them accordingly to the target URL, whether this exist as a page in the output folder or not. The following diagram provides more context on the relationship between these as to provide guidance on their interaction and desired outcome.

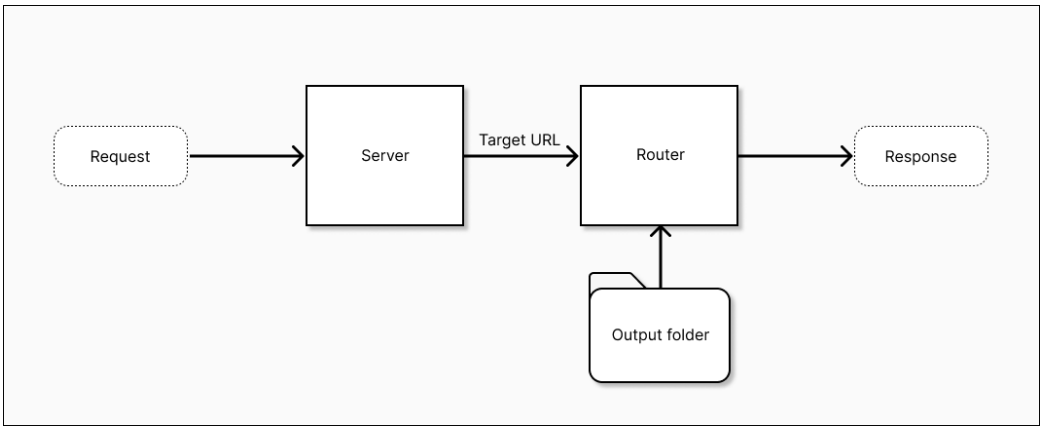

Figure 3.2: Serve process using Server and Router modules

3.4.5 Logger

It is imperative that during the entirety of the aforementioned modules, the system maintains constant communication with the user, providing updates and notifications regarding all ongoing activities. This is necessary to guarantee the user's awareness and facilitate comprehensive understanding of each

individual activity.

According to this, it is necessary to implement an adaptive logging system that is tailored to each phase, as each step possesses distinct information to offer and certain requirements to fulfill. Specifically, it is necessary for one logger to furnish details pertaining to the construction of each page, while another logger should supply information regarding requests, encompassing the IP address of the requester, the requested page, and the status of the router.

Furthermore, it is imperative for the loggers to accurately record the precise date and time of each event that takes place.

Hence, it is appropriate to designate the logger responsible for the generation and parsing phases as `BuilderLogger`, and the logger responsible for serving and routing as `ServeLogger`. On the contrary, in order to enhance simplicity and enhance readability, both loggers will employ a uniform interface to provide an equivalent Log function, which serves the aim of displaying information pertaining to the ongoing activity.

3.4.6 CLI

The command line interface of VaGo is designed to ensure user-friendliness, clarity, and comprehensive documentation. This objective will be achieved by utilizing the urfave/cli package (as discussed in the subsequent chapter), which offers functionalities for implementing command-line interface (CLI) commands with flags, storing parameters in variables for future utilization, abbreviated commands, aid with documentation help, and further capabilities.

According to this, the package will incorporate two primary commands: Build and Serve. The initial option, "–no-log," will indicate to the system that it should refrain from generating any informational logs. Additionally, the flag " –no-time" will be employed to exclude timestamps from the logs, hence preventing an excessive display of distracting information and numerical values on the screen. Subsequently, the system will execute all the nec-

essary procedures to construct webpages from markdown files, as observed in the generator and parser phases.

On the contrary, the serve command will incorporate a flag, namely –port, which allows for the specification of a particular port to be utilized by the system for serving files. The default port for this will be 8080. Subsequently, the system will execute the necessary operations during the Serve phase.

It is noteworthy that for both commands, the system will retrieve information from the configuration files in order to access the necessary specifics for input and output files, styles, theming, homepage, and any other pertinent information required for the effective and adaptable execution of the software.

Additionally, the subsequent graphic illustrates an overview of the procedure undertaken by the Command Line Interface (CLI) to obtain the user's request, parse the configuration file (in yaml format), and choose the appropriate phase to execute, incorporating the user-specified flags as variables.

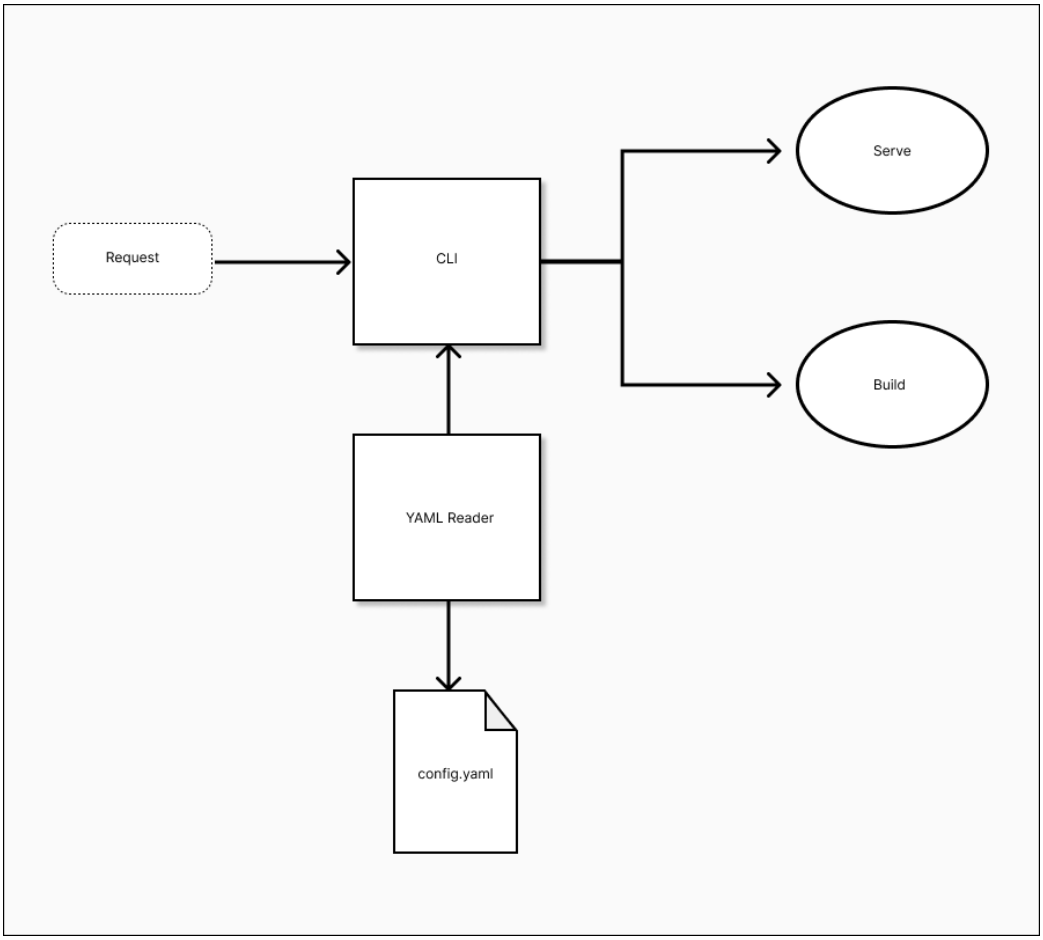

Figure 3.3: CLI processing provided command and reading configuration YAML file.

3.4.7 Development strategy

This project is meant to be very simple and minimalistic in terms of development. And on top of that, the language used, Go, aims for simplicity on its syntax and usage.

There are multiple advantages for this, such as keeping an easier read-

ability and enhanced capacity to understand what is happening in a certain piece of code, but it also keeps the consistency and concrete structure over the whole project, which will benefit later on when multiple pieces of code (modules) are connected, and the consumer of further modules must be able to easily understand its interface and purpose, as it will allow the development process to not only go faster but also to minimize the number of errors, and while there may appear some issues, the concreteness and simplicity will make it easier to overcome and conquer the upcoming challenges.

Moreover, besides the already provided simple syntax from Go, and its recommended best practices, the development process will make use of the Single-responsability principle[#], to encapsulate the multiple functionalities in different modules/files, hence each one must serve to one single purpose.

As per this, if a structure is created, and there are some methods attached to this struct, these must remain in the same file. On the other hand, if the struct does not have any methods, a single file must be defined for it, although it won't have any other methods or functions.

Moreover, multiple functions that share or serve a single purpose, must be created in the same file.

Similarly, it is absolutely forbidden the creation of modules with ambiguous names such as 'util', where its purpose and functionalities are not clear for the consumer. The module's name must explicitly reflect its purpose, and if the collection of variables, functions and methods within it are not coherent, then the whole existence of this module must be re-evaluated in order to follow a semantic meaning.

It is important to note that during the description of these strategies, the word "class" has not been mentioned, and there is a relevant reason for that, which will be explained in the next topic.

3.4.7.1 Go vs OOP

Go, the programming language, by design, is meant to be Object Oriented Programming and at the same time it is not. This limitation arises from the fact that, while it is feasible to construct a class-like abstraction of a tangible entity using structures and methods, it lacks support for hierarchical relationships. Consequently, this necessitates that developers employ abstractions at a single layer, simplifying the code by avoiding the accumulation of excessive levels of complexity, and instead favoring the usage of composition.

Although certain teams may view this as a drawback, it is important to note that this particular attribute aligns seamlessly with the objectives of the project, which prioritize simplicity and minimalistic code. This assertion is supported by the aforementioned reasons.

Furthermore, it provides support for dynamic interfaces, which implies that there is no need to statically declare the objects that implement specific interfaces. Instead, the interpretation of these interfaces occurs dynamically when the corresponding methods are implemented.

In order to maintain simplicity in software interfaces, it is imperative to adhere to the design principle of limiting the number of methods to a maximum of five. Additionally, the Interface Segregation principle should be observed to prevent the creation of unnecessary interfaces. For an interface to be justified, it must possess two essential qualities: a semantic purpose or meaning for its existence, and a clear enhancement to the code that cannot be achieved through any other means.

Chapter 4

Used technologies

This section will present a review of the external technologies, modules, and frameworks that have been incorporated into the project. These components have been interconnected to address specific issues, streamline the development process, and meet the defined requirements.

4.1 gomarkdown

https://github.com/gomarkdown/markdown

External Go module that provides enhanced features to extract the content of a markdown file in tokens following the Abstract Syntax Tree from the file, useful with the extraction that are included within other tokens. For instance, a bold text that is within a bigger piece of text from a list.

This is useful as the context is preserved and can be then used to follow a given structure from the provided template at the generation stage, providing both flexibility and ease to use for the end user.

It is important to note that this same library provides also functionali-

ties to automatically parse markdown files to HTML. However, this is not used in the project given that it provides limited flexibility, and it does not capabilities to produce HTML from a provided template, which is one of the main functionalities of VaGo. Similarly, it does not have any HTTP service to serve and route requests.

Therefore, this module is only meant to be used in the Parser stage on the information extraction step, with pertinent modifications to adapt it to VaGo requirements.

4.2 Go standard libraries

Multiple modules from the Go standard have been used in the implementation of this project, as it has been limited to the usage of standard libraries as much as possible to avoid overloading from external sources, which are prompt to unexpected changes and failures.

It is important to note that this section is not meant to go through all the modules available and used during the development of this project, as most of them provide trivial functionalities such as string formatting, but some of the most important modules that provides functionalities required for the development of VaGo, which would require extended amount of time to develop otherwise, and are not the main focus of this project.

4.2.1 HTML/Template

https://pkg.go.dev/html/template

The package template (html/template) is designed to facilitate the creation of data-driven templates that generate HTML output. These templates are specifically designed to mitigate the risk of code injection, ensuring the security of the resulting HTML content. The package in question offers an interface that is identical to that of text/template. Therefore, it is recom-

mended to utilize this package in favor of text/template whenever the desired output is in the form of HTML.

Moreover, by means of this package, VaGo can provide extended flexibility and customization capabilities to the end user when creating their own site implementing their desired skeleton and structure to be followed by the markdown content.

4.2.2 Go YAML

gopkg.in/yaml.v3

The third iteration of the YAML package for the Go programming language introduces enhanced functionality enabling effortless encoding and decoding of YAML files.

The primary purpose of this tool is to extract the values from configuration files and organize them into predefined structures. These structures are subsequently utilized to get specific information needed for various activities, including input and output folder management.

4.3 urfave/cli

https://github.com/urfave/cli

This package provides a straightforward and user-friendly solution for constructing Command Line Interface (CLI) tools in the Go programming language. It has several features, including the ability to add commands and subcommands, support for alias and prefix matching systems, an automatically created help system with documentation in markdown format, and additional functionalities.

The purpose of this package is to facilitate the construction of VaGo

as a command line tool. It accomplishes this by implementing commands for the Build and Serve stages, which include input arguments, flags, and a help command to offer guidance on usage. The package aims to provide a straightforward implementation and usage experience for the end user. Additionally, it serves as a convenient method for installing VaGo on the user's system.

4.4 HTTP

Chapter 5

Implementation

This chapter will cover the implementation details of VaGo, reviewing the proposed software design and discussing whether it was followed, or if some modifications were required and why.

5.1 Parser

The following files are contained in the extractor package.

The parser will obtain information from the markdown file and store it in the proposed Out structure type. Therefore, the first step here would be to define the shape of this struct, which contains the following attributes:

- H1: A string.
- H2: String array.
- H3: String array.
- H4: String array.
- H5: String array.
- H6: String array.
- Content: Full HTML content, containing all the information from the

markdown file in one variable.

- P: String array.
- Ul: String array.
- Ol: String array.
- Link: String array.
- Image: String array.

This struct will allow the user to navigate over the multiple input tokens, iterating over them to follow the same order it was provided initially. Moreover, the user can obtain multiple headings and paragraphs using index navigation. This is done with the purpose of providing flexibility for templating when giving structure to the content.

On the other hand, if the user decides to use the automatic, same structure from markdown, they can simply output Content which will display everything in HTML format already.

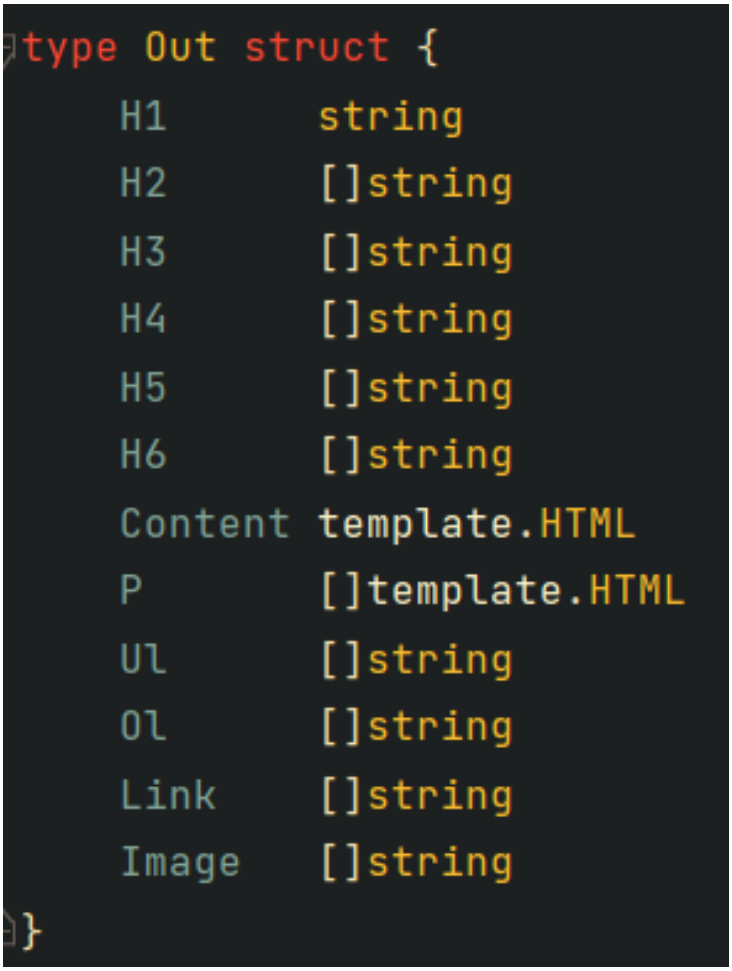

Figure 5.1: Out structure type.

Next, the parser will make use of this structure to store values. Moreover, as mentioned previously, VaGo makes use of the `gomarkdown` package to construct the Abstract Syntax Tree, with some slights modifications. These modifications are required because, otherwise, all the HTML content will be stored in a single variable, and this does not offer the possibility of building a flexible template system obtain values from different variables.

Therefore, the navigation of the AST must be modified to extract the

given content as desired. This navigation is done via the `gomarkdown` function named `ast.WalkFunc`, which offer the possibility to get through every child node of the tree. During this walk, the titles will be obtained first as no additional logic is required besides detecting whether the given token is a heading or not, and what specific heading is being read.

Nonetheless, for the rest of the content, further logic must be applied. This is due to the fact that, although there are multiple paragraphs, these can be grouped in a single section given the agreement that every section is split by a title. Hence, the array `P` will store multiple paragraphs in one single index following this logic.

It is important to note that this does not include the first heading, given that it is assumed there will be only one per page (as main title) to follow the W3C accessibility norms for titles.

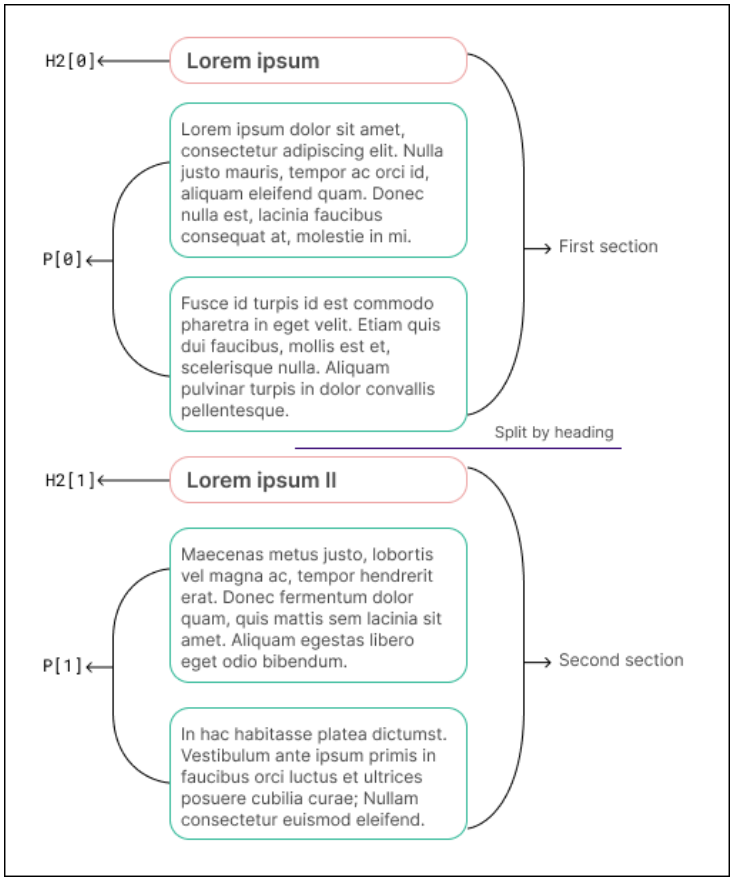

Figure 5.2: Content separation logic.

This logic is implemented as follows: If a heading is detected, a new section upcoming, meaning that the index for `P` must be incremented. Otherwise, store the content in the current index for `P`.

However, this approach is context-awareness lacking, leading to problems when deciding what's the current index, what would be the next one, and knowing whether the AST walker is already in a section. The lack of awareness can be problematic in certain circumstances, for instance, what happens if there are two contiguous titles, meaning that no paragraphs are added but the index shouldn´t increment either.

For this reason, a new structure is defined to track these process, named ContentTracker, which stores the current index amount, and a boolean specifying whether the tracker is inside a section or not.

```
type ContentTracker struct {
    IsInside bool
    Index    int
}
```

Figure 5.3: Content tracker structure type

Consequently, the logic for parsing the content will be as follows:

When a heading is identified, the program should examine the tracker boolean to determine if it is located within the heading. If the boolean value is true, it should be changed to false. Additionally, the index should be incremented by one if the current paragraph is not empty. If it isn't inside, set it to true.

In the event that a non-heading is identified, the tracker should be updated to indicate that it is inside (i.e., set to true). Subsequently, the markdown text should be retrieved and saved to the variable `P` at the current index of the tracker.

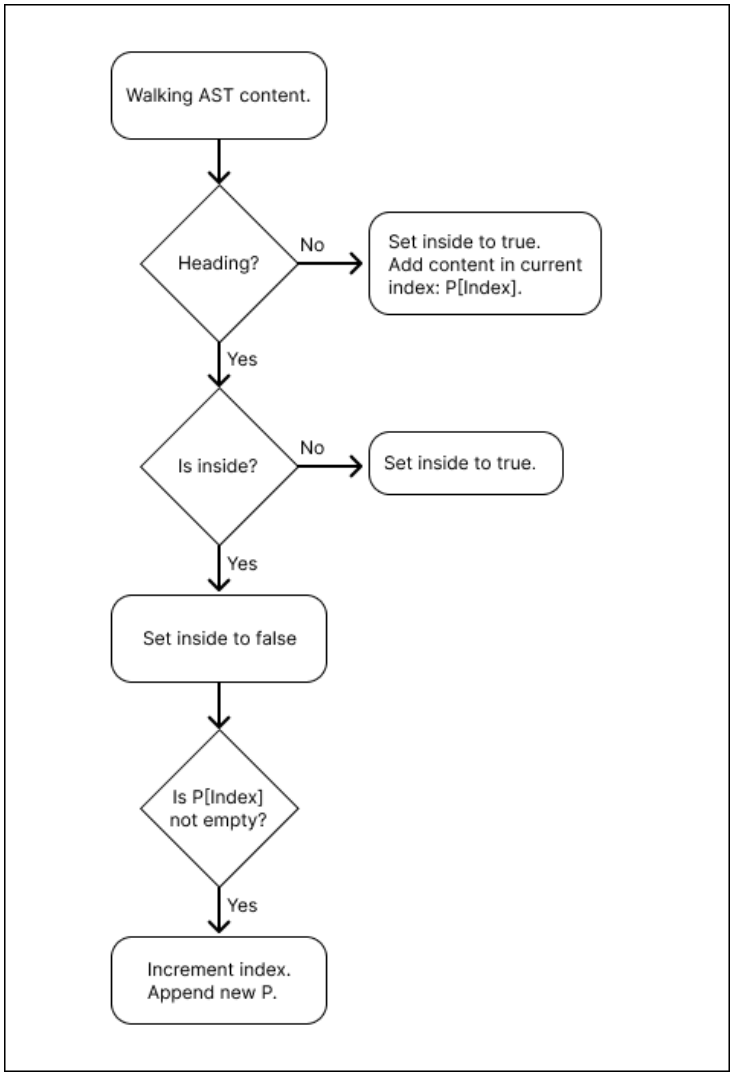

Figure 5.4: Parser logic.

As a result of the gomarkdown AST walking mechanism's behavior, the parser is required to execute a skip children operation when a content token (excluding headings) is identified. This is done to prevent redundant readings of inner children within the paragraphs, such as bold, italic, and links. These elements have already been stored in the fully parsed HTML representation

of the paragraph.

This is accomplished by evaluating the content extraction function, which will return true if it's in presence of content, and false in case of a heading.

Finally, the AST extraction function ends up with the following anatomy, returning the provided document and the filled Out structure variable.

```
func extractAst(doc ast.Node) (ast.Node, Out) {
    var out Out
    out.P = append(out.P, elems...: "")
    var tracker ContentTracker

    ast.WalkFunc(doc, func(node ast.Node, entering bool) ast.WalkStatus {
        if entering {
            extractTitles(node, &out)
            if extractContent(node, &out, &tracker) {
                return ast.SkipChildren
            }
        }

        return ast.GoToNext
    })

    return doc, out
}
```

Figure 5.5: AST extraction function.

This all is encapsulated in a public function named ParseMarkdown, where it will set up the gomarkdown to begin the AST walk, use previous function to build the Out structure, and finally render everything to be stored in Out.Content.

```go
func ParseMarkdown(md []byte) Out {

    extensions := parser.CommonExtensions
    p := parser.NewWithExtensions(extensions)
    doc := p.Parse(md)

    doc, out := extractAst(doc)

    htmlFlags := html.CommonFlags
    opts := html.RendererOptions{Flags: htmlFlags}
    renderer := html.NewRenderer(opts)
    result := markdown.Render(doc, renderer)

    out.Content = template.HTML(result)

    return out
}
```

Figure 5.6: Parse markdown function

5.2 Generator

The following files are contained in the generator package

The generator process is fairly simple as it will limit itself to obtain the parsed content and construct files through a template on the specified output folder.

5.2.1 Build pages

First, it will build the pages, taking every markdown file from the input folder, obtain its content, parse it using the ParseMarkdown function from previous package, take the provided template, create a new file with the

same name but changing the extension to .html, and output the result of this template with the Out content. The last two steps are encapsulated in the function buildPage for simplicity.

This same process is repeated for each input file to create its HTML version accordingly.

```
func buildPage(templ string, content extractor.Out, filename string) {
    t := template.Must(template.ParseFiles(templ))
    filename = changeFileExtension(filename, extension: "html")
    file := createFile(filename)
    err := t.Execute(file, content)
    if err != nil { err *

    if err := file.Close(); err != nil { err *
}
```

Figure 5.7: Build a single page with akin name

```go
func buildPages(config input.IOPath, noLog bool, noTime bool) {
    files, err := os.ReadDir(config.InFolder)
    if err != nil {
        log.Fatal(err)
    }

    var logger generator.GeneratorLogger
    for _, file := range files {
        if file.IsDir() {
            continue
        }

        logger.Init(file.Name(), noLog, noTime)
        logger.Info( text: "Starting page")

        md, err := os.ReadFile(config.InFolder + file.Name())
        if err != nil {
            logger.Error(err)
        }

        logger.Info( text: "Parsing markdown")
        out := extractor.ParseMarkdown(md)

        logger.Info( text: "Building HTML output")
        buildPage(config.InTemplate, out, config.OutFolder+file.Name())
        logger.Info( text: "Page built in output folder %s", config.OutFolder)
    }
}
```

Figure 5.8: Build pages function, repeated for every input file

5.2.2 Build styles

Secondly, the same process is repeated for the theme styles, except for a few modifications as some steps are not required: Instead of parsing the content, it will get it from a YAML file with the variables to adapt, and these will be added to a given CSS file, generating a new one in the output folder. This last step is encapsulated in function named buildStyle.

This same process is repeated for all files contained in the given styles folder.

```go
func buildStyle(styles string, theme map[string]interface{}, filename string) {
    t := template.Must(template.ParseFiles(styles))
    filename = changeFileExtension(filename, extension: "css")
    file := createFile(filename)
    err := t.Execute(file, theme)
    if err != nil { err }

    if err := file.Close(); err != nil { err }
}
```

Figure 5.9: Build a single CSS file with akin name

```go
func buildStyles(config input.IOPath, noLog bool, noTime bool) {
    var logger generator.GeneratorLogger
    theme := input.GetTheme(config.InTheme)
    styles, err := os.ReadDir(config.StylesFolder)
    if err != nil {
        panic(err)
    }

    for _, style := range styles {
        if style.IsDir() {
            continue
        }

        logger.Init(style.Name(), noLog, noTime)
        logger.Info( text: "Parsing theme.")
        buildStyle(config.StylesFolder+style.Name(), theme, config.OutFolder+style.Name())
        logger.Info( text: "Style built in output folder %s", config.OutFolder)
    }
}
```

Figure 5.10: Build pages function, repeated for every CSS file

It is worth noting that throughout this whole process, a logger is in place and being used to keep the user informed of every step performed, using the established structure with timestamp, page name and logging information. Logger implementation is explained later on following sections.

5.3 Server

The server implementation is characterized by its simplicity, as it primarily consists of two core components: the listener, responsible for actively moni-

toring incoming requests, and the router, which directs these requests based on their respective target URLs.

The implementation utilizes the net/http package in Go to establish a server and manage incoming requests in a suitable manner. The following code snippet demonstrates the creation and utilization of a router handler within the ListenAndServe function, which will create file server from a provided directory.

This function accepts a specified port and the router handler as parameters, enabling the determination of appropriate actions for each incoming request. It is noteworthy that the router necessitates an input directory from which to retrieve the files for serving. This input directory is acquired through the configuration variables supplied by the user.

The Serve function, which is the primary public function utilized by the remaining components of the system, is responsible for initiating the server.

```go
func Serve(port int, config input.IOPath) {
    var logger server.ServerLogger
    directory := config.OutFolder
    homepath := config.OutFolder + config.Home
    logger.Init()

    logger.Info( text: "Starting...")
    logger.Info( text: "Listening for requests on port %s", strconv.Itoa(port))
    err := http.ListenAndServe(":"+strconv.Itoa(port), RouterHandler(http.Dir(directory), homepath))
    if err != nil {
        logger.Error(err)
    }
    logger.Warning( text: "Closing server and port...")
}
```

Figure 5.11: Server entrypoint function.

Subsequently, the router is equipped with supplementary logic to decide the appropriate files to transmit. This is particularly important in cases where the requested file from the destination URL cannot be found, necessitating the transmission of a 404 page.

Furthermore, in cases where a destination URL is not explicitly mentioned, it is imperative for the system to generate the home page. The user is required to submit the home page information in the configuration file,

and the path to the home page must be set as a function parameter for the RouterHandler.

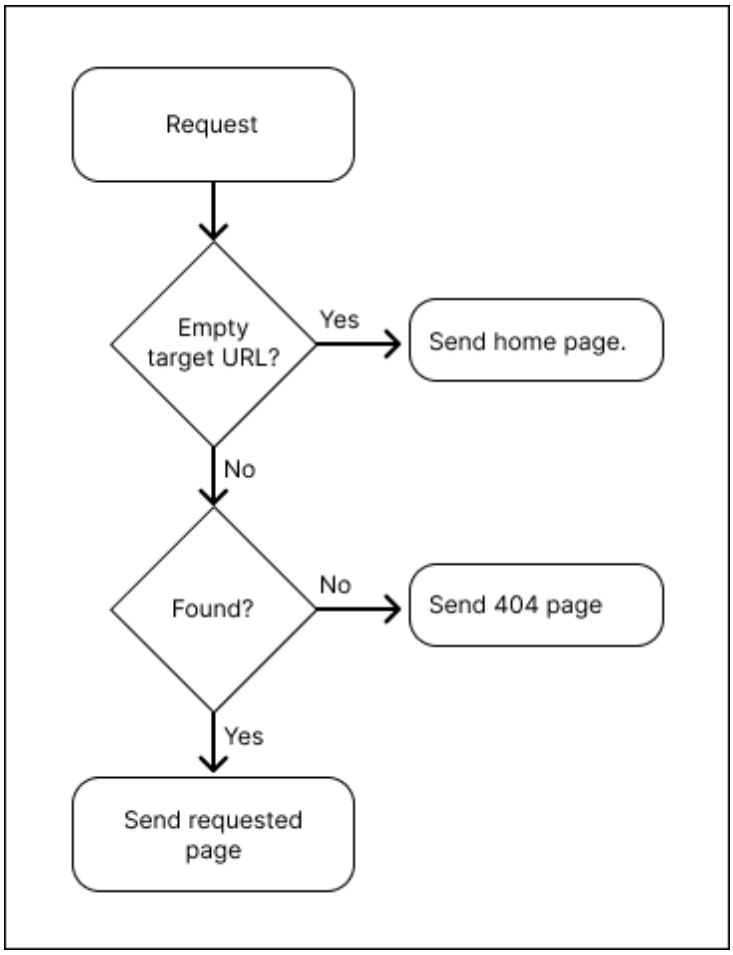

Figure 5.12: Router handler logic.

Figure 5.13: Router handler function.

5.4 Logger

As described in previous chapters, the project will need two types of loggers with their clear differences: Generator Logger and Server Logger. Notwithstanding, these both share similar functionality, which includes display information, have different level of logging (information, warning, error) and include a prefix.

Therefore, to ensure the same structure is followed, Go interfaces has been put in place, enforcing the development to follow the functions described on each, and reducing the cognitive load:

- Logger interface, with three functions for each logging level: Info, Warning and Error. The idea is that the developer can simply use each function accordingly and the system will output the information sent as parameters (similar to fmt.Println) with a prefix establishing the given level.

```
type Logger interface {
    Info(string, ...any)
    Warning(string, ...any)
    Error(error, ...any)
}

const (
    INFO    = "INFO"
    ERROR   = "ERROR"
    WARNING = "WARNING"
)
```

Figure 5.14: Logger interface.

- Prefixer interface, short and simple with only one function: Prefix. This helps to encapsulate the prefixing logic, and build prefixes independently of the logger.

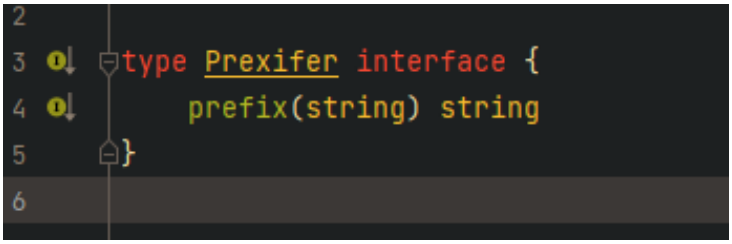

Figure 5.15: Prefixer interface.

Then, the implementation of each logger is dedicated to its requirements and functionality.

5.4.1 Generator Logger

Instantiated by an Init function, where the developer can specify in advance the pagename to log as a prefix, and whether it should include timestamps or no logs at all. These last two parameters are taken by the prefixer, which will take care of the prefixing log information.

Moreover, the logging functions, Info, Warning and Error will simply take the prefixer and use the Prefix function to add the logging level, followed by the logging text, which may or not have arguments to add to the text, similar to what is done in fmt.Print, adding extra formatting options for flexibility.

```go
type GeneratorLogger struct {
    pagename string
    logger   *log.Logger
    prefixer generatorPrefixer
}

func (bl *GeneratorLogger) Init(pagename string, noLog bool, noTime bool) GeneratorLogger {
    bl.pagename = pagename
    bl.logger = log.New(os.Stdout, prefix: "", flag: 0)
    bl.prefixer = generatorPrefixer{
        NoLog:    noLog,
        NoTime:   noTime,
        Pagename: pagename,
    }

    return *bl
}
```

Figure 5.16: Generator logger implementation.

5.4.2 Generator Prefixer

Here the prefixer will take care of the timestamp in RFC822 format, whether the logs should be printed or not and the prefixed log level, and the file/page name to be logged. Doing so allow the prefix to be flexible according to the established configuration.

```go
type generatorPrefixer struct {
    NoLog    bool
    NoTime   bool
    Pagename string
}

func (bp *generatorPrefixer) prefix(logLevel string) string {
    if bp.NoLog {
        return ""
    }

    if bp.NoTime {
        return fmt.Sprintf(format: "[%s] %s: ", bp.Pagename, logLevel)
    }

    now := time.Now().Format(time.RFC822)
    return fmt.Sprintf(format: "[%s][%s] %s: ", now, bp.Pagename, logLevel)
}
```

Figure 5.17: Generator prefixer implementation.

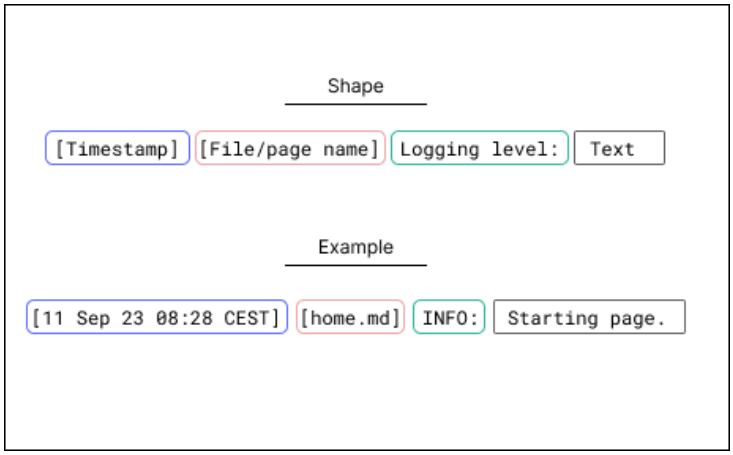

Figure 5.18: Generator logger and prefixer result.

5.5 CLI

As mentioned in previous chapters, the CLI is developed with urfave/cli package, which allow customization for commands using typical Linux CLI format, displaying the information required from the user to execute them successfully. This information includes the following:

- Build
  - Name: build.
  - Flags: *config* (–c), *no-log* (–nl) and *no-time* (–ct).
  - Description: Build/Generate web content from markdown files on indicated folder (default: config.yaml).

```
Name:  "build",
Usage: "Build/Generate web content from markdown files on indicated folder (default: config.yaml).",
Flags: []cli.Flag{
    &cli.StringFlag{
        Name:        "config",
        Value:       "config.yaml",
        Aliases:     []string{"c"},
        Usage:       "Load configuration from `FILE`.",
        Destination: &configFile,
    },
    &cli.BoolFlag{
        Name:        "no-log",
        Value:       false,
        Aliases:     []string{"nl"},
        Usage:       "Don't display logs for every page.",
        Destination: &noLog,
    },
    &cli.BoolFlag{
        Name:        "no-time",
        Value:       false,
        Aliases:     []string{"ct"},
        Usage:       "Don't display timestamps when logging.",
        Destination: &noTime,
    },
},
```

Figure 5.19: VaGo CLI build definition.

- Serve
  - Name: serve.
  - Flags: *port* (–p) and *config* (–c).
  - Description: Start serving generated content via specific port (default: 8080).

```
Name:  "serve",
Usage: "Start serving generated content via specific port (default: 8080)",
Flags: []cli.Flag{
    &cli.IntFlag{
        Name:        "port",
        Value:       8080,
        Aliases:     []string{"p"},
        Usage:       "Serve files on port `PORT`",
        Destination: &port,
    },
    &cli.StringFlag{
        Name:        "config",
        Value:       "config.yaml",
        Aliases:     []string{"c"},
        Usage:       "Load configuration from `FILE`.",
        Destination: &configFile,
    },
},
```

Figure 5.20: VaGo CLI serve definition.

For both commands, the common flag, config, is meant to be used to indicate the entry configuration file, which contains information about input and output folders, templates, styles, theme and homepage. For Build, no-log and no-time are used to indicate the system to not print logs at all and to not display timestamps, accordingly. For Serve, the port flag is used to specify the port number to be used.

Moreover, the package provides the utility to display help usage for both commands, including the main VaGo application. This documentation provides information about the command's description, usage of every flag, input type, default values, short command, and how to correctly structure the operation.

Figure 5.21: VaGo CLI help.

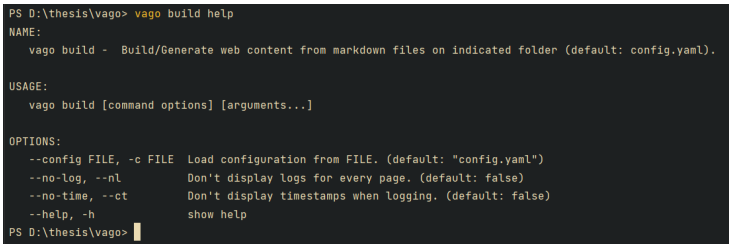

Figure 5.22: VaGo CLI build help.

```
PS D:\thesis\vago> vago serve help
NAME:
   vago serve - Start serving generated content via specific port (default: 8080)

USAGE:
   vago serve [command options] [arguments...]

OPTIONS:
   --port PORT, -p PORT    Serve files on port PORT (default: 8080)
   --config FILE, -c FILE  Load configuration from FILE. (default: "config.yaml")
   --help, -h              show help
PS D:\thesis\vago>
```

Figure 5.23: VaGo CLI serve help.

Finally, when a Build command is called, it will execute the Build function from the generator package, sending the configuration file (reading it through a helper function to get YAML files), no-log and no-time as parameters.

On the other side, the Serve command will execute the Serve function from the server package, sending the port number and configuration file as parameters.

Both commands are specified in the Action field for urfave/cli package command type.

```go
Action: func(*cli.Context) error {
    config := input.ReadYAML(configFile).AsIOPath()
    generator.Build(config, noLog, noTime)
    return nil
},
```

Figure 5.24: VaGo CLI build action definition.

```
Action: func(*cli.Context) error {
    config := input.ReadYAML(configFile).AsIOPath()
    server.Serve(port, config)
    return nil
},
```

Figure 5.25: VaGo CLI serve action definition.

Chapter 6

Results

The implemented methodology led to the development of a system that possesses the ability to efficiently produce and deliver static content. This system is characterized by its user-friendly nature, requiring minimal effort to learn and operate. Furthermore, it is equipped with comprehensive logging capabilities, enabling users to gain insights into the system's operations at any given time.

The first step to follow in order to use VaGo, is to install the software using the native Go command *go install*. This will get all the required dependencies, compile the code, build an executable and install the CLI for the system to recognize it as a command-line tool.

Moreover, to perform the most basic tests, a markdown file with all the types of tokens has been used to make sure everything is being correctly parsed. The file contains a structure similar to the following (whole file not included for simplicity), named as *first .md*:

```
## Emphasis

**This is bold text**
```

```
__This is bold text__

*This is italic text*

_This is italic text_

~~Strikethrough~~


## Blockquotes


> Blockquotes can also be nested...
>> ...by using additional greater-than signs right next to
each other...
> > > ...or with spaces between arrows.


## Lists

Unordered

+ Create a list by starting a line with `+`, `-`, or `*`
+ Sub-lists are made by indenting 2 spaces:
- Marker character change forces new list start:
* Ac tristique libero volutpat at
+ Facilisis in pretium nisl aliquet
- Nulla volutpat aliquam velit
+ Very easy!

Ordered

1. Lorem ipsum dolor sit amet
2. Consectetur adipiscing elit
3. Integer molestie lorem at massa
```

```
1. You can use sequential numbers...
1. ...or keep all the numbers as `1.`

Start numbering with offset:

57. foo
1. bar


## Code

Inline `code`

Indented code

// Some comments
line 1 of code
line 2 of code
line 3 of code
.
.
.
```

This way, it can be used to test the parsing and generation mechanism. Adding this file to the input file, and using the following configuration (*config.yaml*), the building process can be started.

```
# config.yaml

input: "./source/"
template: "./template/index.html"
output: "./out/"
```

Where *./source/* is the input folder where the markdown file is stored, *./out/* is the output folder where the result will be saved, and *./template/index.html* is the template file with the following structure:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<link rel="stylesheet" href="styles.css">
<title> {{ .H1 }} </title>
</head>
<body>

<section> {{ .Content }} </section>

</body>
</html>
```

The template structure is rather simple, but it serves the purpose of testing the parsing mechanism. It's important to note that the *.Content* variable is being used, meaning that all the content will be displayed in one single place, without much customization.

Then, the Build process is executed by using the command *vago build*. This will start the parsing process, displaying logs of every steps, and finally the new file generation will be performed to create an HTML version file of the provided input, following the given template.

```
PS D:\thesis\vago> vago build
[11 Sep 23 22:17 CEST][first.md] INFO: Starting page
[11 Sep 23 22:17 CEST][first.md] INFO: Parsing markdown
[11 Sep 23 22:17 CEST][first.md] INFO: Building HTML output
[11 Sep 23 22:17 CEST][first.md] INFO: Page built in output folder ./out/
```

Figure 6.1: VaGo build execution with first.md

Now, inspecting the */out/* folder, a new file can be seen with the same name as the input file, but with the HTML extension: *first.html*. Reviewing the content of this file, it can be verified that it has been correctly parsed, as

each token is correctly mapped to its equivalent HTML tag (whole file not included for simplicity):

```
<section>
<h2>
    Blockquotes</h2>
    <p><strong>This is bold text</strong></p>
    <p><strong>This is bold text</strong></p>
    <p><em>This is italic text</em></p>
    <p><em>This is italic text</em></p>
    <p><del>Strikethrough</del></p>

    </section>



    <section>
    <h2>Lists</h2>
    <blockquote>
    <p>Blockquotes can also be nested&hellip;</p>

    <blockquote>
    <p>&hellip;by using additional greater-than signs right
    next to each other&hellip;</p>

    <blockquote>
    <p>&hellip;or with spaces between arrows.</p>
    </blockquote>
    </blockquote>
    </blockquote>

    </section>



    <section>
    <h2>Code</h2>
    <p>Unordered</p>
```

```
<ul>
<li>Create a list by starting a line with <code>+</code>,
<code>-</code>, or <code>*</code></li>
<li>Sub-lists are made by indenting 2 spaces:

<ul>
<li>Marker character change forces new list start:

<ul>
<li>Ac tristique libero volutpat at</li>
<li>Facilisis in pretium nisl aliquet</li>
<li>Nulla volutpat aliquam velit</li>
</ul></li>
</ul></li>
<li>Very easy!</li>
</ul>
<p>Ordered</p>
<ol>
<li><p>Lorem ipsum dolor sit amet</p></li>

<li><p>Consectetur adipiscing elit</p></li>

<li><p>Integer molestie lorem at massa</p></li>

<li><p>You can use sequential numbers&hellip;</p></li>

<li><p>&hellip;or keep all the numbers as <code>1.</code>
</p></li>
</ol>
<p>Start numbering with offset:</p>
<ol>
<li>foo</li>
<li>bar</li>
</ol>

</section>
```

```
<section>
<h2>Tables</h2>
<p>Inline <code>code</code></p>
<p>Indented code</p>
<pre><code>// Some comments
line 1 of code
line 2 of code
line 3 of code
</code></pre>
```

Nonetheless, in order to verify that it's working as expected with all the tags correctly set, it is imperative to test it out in a browser, to determine if the content is being displayed as it should. Therefore, this can be tried out by simply starting a server and accessing this file using URL path */first.html*. On a side note, the home page is not being set in the configuration file, hence the server will lack of a homepage, although it doesn't impose an issue for the purpose of this test. The test can be started by using the command *vago serve*:

```
PS D:\thesis\vago> vago serve
[11 Sep 23 22:44 CEST] INFO: Starting...
[11 Sep 23 22:44 CEST] INFO: Listening for requests on port 8080
```

Figure 6.2: VaGo serve execution with first.md

Then, accessing *localhost:8080/first.html* the following page can be seen in the browser:

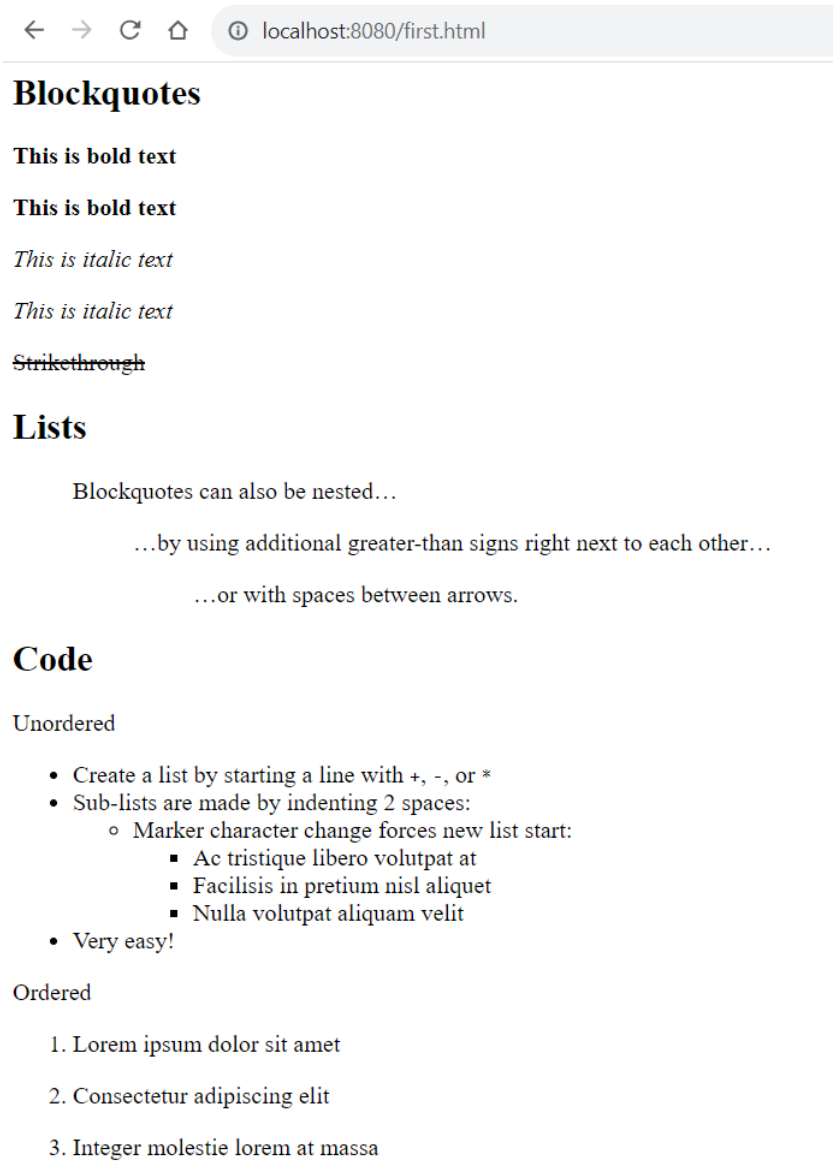

Figure 6.3: first.html displayed in browser.

With this simple test, the main functionality of parsing, generating, serving and routing has been successfully demonstrated, proving VaGo to be a static site generator capable of performing the most simple tasks.

Notwithstanding, there is left to try the templating and theming system, to ensure its flexibility capabilities. For this, a new input file is constructed with several sections divided by titles. The new file will be named as lorem.md, and it has the following structure (not all the content is displayed for simplicity):

```
# Lorem Ipsum

## What is Lorem Ipsum?

Lorem Ipsum is simply dummy text of the printing and
typesetting industry. Lorem Ipsum has been the industry's
standard    dummy text ever since the 1500s, when an
unknown printer took a galley of type and scrambled it
to make a type specimen book. It has survived not only
five centuries, but also the leap into electronic
typesetting, remaining essentially unchanged. It was
popularised in the 1960s with the release of Letraset
sheets containing Lorem Ipsum passages, and more
recently with desktop publishing software like Aldus
PageMaker including versions of Lorem Ipsum.

## Why do we use it?

It is a long established fact that a reader will be
distracted by the readable content of a page when
looking at its layout. The point of using Lorem
Ipsum is that it has a more-or-less normal
distribution of letters, as opposed to using
'Content here, content here', making it look like
readable English. Many desktop publishing packages
and web page editors now use Lorem Ipsum as their
default model text, and a search for 'lorem ipsum'
will uncover many web sites still in their infancy.
Various versions have evolved over the years,
sometimes by accident, sometimes on purpose
(injected humour and the like).
```

Since the content of this new input file has defined sections, the previous template can be modified to take advantage of this and provide a more customized structure, iterating over second headings *H2* and using the same index to iterate over *P* values. This is done by using the *range* function from Go templates and storing a reference to *P* to avoid losing its content within the range context, then providing the same iterator *$i* on the function *index*.

These functionalities are all part of the native Go template system, so VaGo takes advantage of the extended already provided features to provide flexible customization options.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<link rel="stylesheet" href="styles.css">
<title> {{ .H1 }} </title>
</head>
<body>

<h1> {{ .H1 }} </h1>
{{ $p := .P }}
{{
    range $i, $H2 := .H2 }}

<section>
<h2>{{ $ H2 }}</h2>
{{ index $p $i }}
</section>

{{ end }}

</body>
</html>
```

In the same way, a new CSS file is added to the */styles/* folder, making use of the theme variables that will be defined later:

```
section{
background-color: {{ .background }};
margin: {{ .margin }};
padding:  {{ .padding }};
}
```

Following this, the variables *background, margin, padding* are defined in the *theme.yaml* file, and further users can modify these variables to match their styling needs. For the moment, the content of this file will be the following:

```
background: "#696969"
margin: "5px"
padding: "5px"
```

Consequently, a few modifications to the configuration file (config.yaml) are required to add the new theme, and the new file *lorem* is added as a homepage too, in order to try out the home page feature.

```
input: "./source/"
template: "./template/index.html"
output: "./out/"

# Theme:
styles: "./styles/"
theme: "./theme.yaml"

home: "lorem.html"
```

Once everything is set up, a new building process can be started using the same command *vago build*, and this will provide context on the files parsed and created, including the style files.

```
PS D:\thesis\vago> vago build
[11 Sep 23 23:14 CEST][first.md] INFO: Starting page
[11 Sep 23 23:14 CEST][first.md] INFO: Parsing markdown
[11 Sep 23 23:14 CEST][first.md] INFO: Building HTML output
[11 Sep 23 23:14 CEST][first.md] INFO: Page built in output folder ./out/
[11 Sep 23 23:14 CEST][lorem.md] INFO: Starting page
[11 Sep 23 23:14 CEST][lorem.md] INFO: Parsing markdown
[11 Sep 23 23:14 CEST][lorem.md] INFO: Building HTML output
[11 Sep 23 23:14 CEST][lorem.md] INFO: Page built in output folder ./out/
[11 Sep 23 23:14 CEST][styles.cssgo] INFO: Parsing theme.
[11 Sep 23 23:14 CEST][styles.cssgo] INFO: Style built in output folder ./out/
PS D:\thesis\vago>
```

Figure 6.4: Building with new file, template and theme.

After this, the files *lorem.html* and *styles.css* has been added to the output folder */out/*. Then, to verify the web content and its new styles added, a new server has to be started using the same command *vago serve* (output omitted for simplicity). Following, the URL *localhost:8080* is accessed, and the content of lorem.html is correctly displayed as home page, with a set of styles provided as expected:

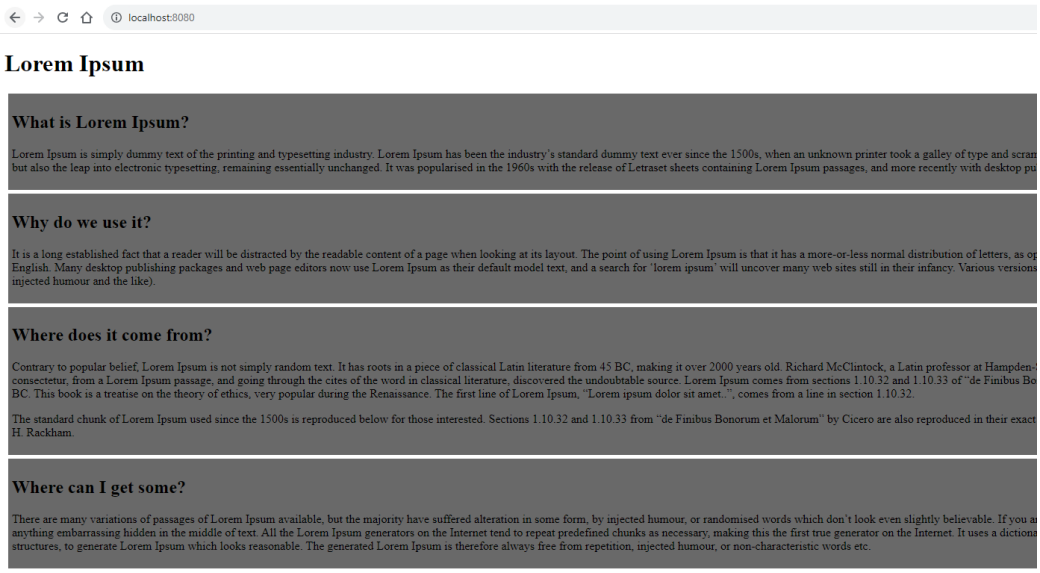

Figure 6.5: Serving lorem.html as homepage with styles.

The provided color scheme and separation between sections may not be desirable. However, this can be easily modified by accessing the theme file and adding the desired characteristics. A cyan like color should provide better readability, and an increased margin and padding should be enough to split the sections.

```
background: "#C0FFEE"
margin: "18px"
padding: "24px"
```

After building and starting the server once again (ouput omitted for simplicity), the following content is displayed in the browser:

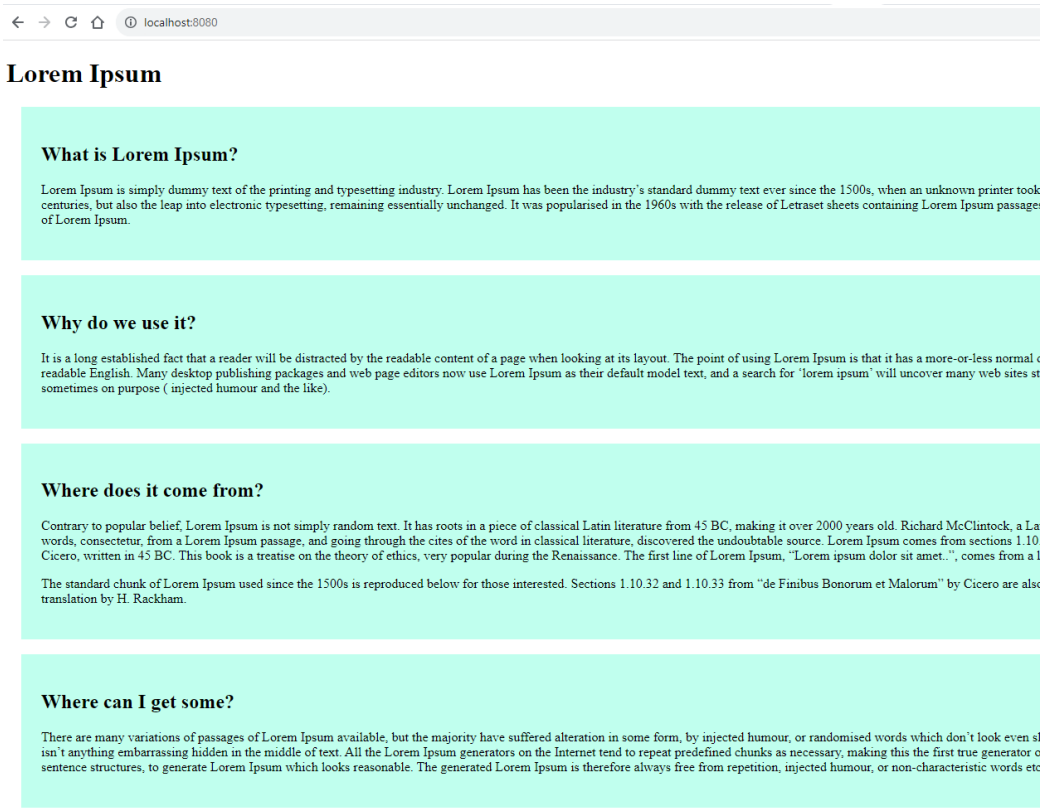

Figure 6.6: Serving lorem.html after modifying the theme.

Chapter 7

Conclusion

As demonstrated on previous chapter, VaGo is a software capable of obtaining input markdown files from an established folder, read each file content, navigate through the abstract syntax tree, obtain the required tokens to build up a web page, parse it into respective HTML tags, create HTML files accordingly to the input content using the same name, build up dedicated styles using a theme system reading customization variables, read configuration file to adapt system to user specifications, provide a listener server on a provided port, route requests to map the same file name convention to target URLs, display insightful logging and an intuitive command interface with discerning usage documentation.

As a result, it has been shown that the implementation of VaGo successfully meets the requirements outlined in previous sections. Furthermore, it can be regarded as a favorable alternative to existing Static Site Generators, as it possesses the ability to execute a majority of the characteristics commonly observed in the present market. This is without accounting for some other measures that are not the primary emphasis of this project, such as its lightning-quick creation and serving of files, which opens up new possibilities beyond merely serving static information: The rapid pace of building construction facilitates the expansion of serverside rendering capabilities.

What's more, VaGo has been developed using a minimalistic approach,

reducing the code and the cognitive load to the lowest possible, with a balance between fulfilling requirements and following the proposed design with the desired simplicity. This minimalistic philosophy has demonstrated an overall increased productivity in terms of implementation, while reducing the amount of complications and challenges to face when dealing with bugs and errors, with the addition of being an open door to extension given that the code base is very simple and easy to read and understand, allowing future collaborators to improve its functioning and/or add new features.

All in all, VaGo has been capable of meeting the requirements established at the beginning, following the minimalistic approach, providing a fully functional static site generator. Hence, this project serves as a proof that selecting the correct tools, frameworks, language, libraries and modules, by setting a realistic set of requirements, with the right approach that ties up the development process without limiting its creativity, and understanding the balance between proposed design and implementation changes, any software project can be developed with ease, ending up with a robust, flexible and open to extension results.

From the point of view of users, VaGo is ready to be used for those looking for a simple software that can allow them to focus on content creation rather than software development details. Create markdown content, input a couple of commands and have your static file served site ready to go.

Chapter 8

Future lines

Although VaGo is capable of performing the task evaluated in this project, it is far from perfect, as there are some improvements and features that can be added as a static site generator.

For instance, the software requires the whole code to be in the same parent folder as the input and output content, styles, template, configuration and more. Thus, it would be very handy to have a feature that allows the creation of a new VaGo project folder from scratch, only with the required folders and files needed to start crafting content, without the code being in the same directory, as it would allow the creation of multiple projects with the same code folder in another place in the system, allowing the user to have a separation of concerns between the content and VaGo code base, useful for further arrangement and manipulation.

On the other hand, given that it uses the concept of themes for styling, a theme package manager can be created along with the main project to provide extended capabilities for theme publication and sharing, as well as download and install other themes. This will allow the user to follow the philosophy of only focusing on content creation and theme customization, as it would subtract the need of crafting CSS styles files from scratch.

Finally, in order to meet the current state of the art of the most popular

static site generators, VaGo must have the capability of extending its functionalities via plugins, which must be easy to find, install, use and manage. An example of a handy plugin, is the capability of adding LaTeX-like content writing, allowing the user to not only create files on markdown format, but also extending these to use LaTex nomenclature, which is useful for mathematicians as they can easily compose complex formulas. This would require a plugin manager, to navigate through existing plugins and install them on command.

Chapter 9

Bibliography

- *What is a static site generator?*. Cloudflare. (n.d.) . https://www.cloudflare.com/learning/performance/static-site-generator/
- Khalid, F. S. (2022, April 18). *How to choose the right static site generator*. GitLab. https://about.gitlab.com/blog/2022/04/18/comparing-static-site-generators/
- Wikimedia Foundation. (2023, August 12). *Static Site Generator*. Wikipedia. https://en.wikipedia.org/wiki/Static*site*generator
- Wikimedia Foundation. (2023a, April 11). *Hugo (software)*. Wikipedia. https://en.wikipedia.org/wiki/Hugo_(software)
- *What is Hugo*. Hugo. (2023, July 13). https://gohugo.io/about/what-is-hugo/
- *Docs. What is Next.js*. Next.js. (n.d.). https://nextjs.org/docs
- *VuePress guide. Introduction*. VuePress. (n.d.). https://v2.vuepress.vuejs.org/guide/
- *Accessibility principles*. Web Accessibility Initiative (WAI) . https://www.w3.org/WAI/fundamentals/accessibility-principles/
- MozDevNet. (n.d.). *What is accessibility?* Learn web development | MDN. https://developer.mozilla.org/en-US/docs/Learn/Accessibility/What*is*accessibility
- MozDevNet. (n.d.-a). *HTML: A good basis for accessibility*. Learn web development | MDN. https://developer.mozilla.org/en-US/docs/Learn/Accessibility/HTML
- Sun, Y. (2019). *Server-Side Rendering. In: Practical Application Development with AppRun*. Apress, Berkeley, CA. https://doi.org/10.1007/978-

1-4842-4069-4_9
- Taufan Fadhilah Iskandar et al (2020). *Comparison between client-side and server-side rendering in the web development* https://iopscience.iop.org/article/10.1088/1757-899X/801/1/012136/pdf