Static Site Generator

La Salle Campus Barcelona

Javier Mérida

12 02 2023

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 customiza-

tion 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 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

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.

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.

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

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

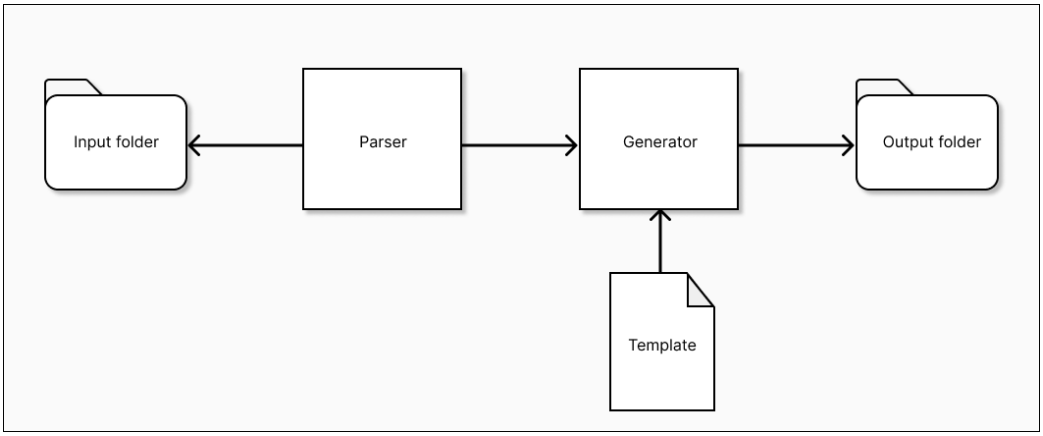

Figure 3.1: Build process using Parser and Generator modules.

folder.

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.

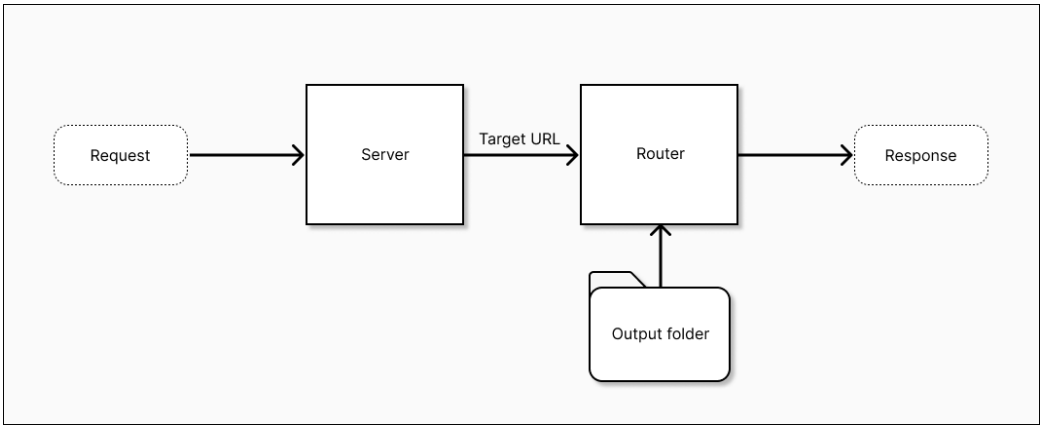

Figure 3.2: Serve process using Server and Router modules

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 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 readability 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 develop-

ment 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.

3.4.6.1 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.