Static Site Generators

Group 2

Reinhard Egger, Daniel Geiger, Lorenz Leitner and Peter Oberrauner

706.041 Information Architecture and Web Usability WS 2019/2020
Graz University of Technology

02 Dec 2019

Abstract

Static Site Generators are tools to automatically create fully-fledged HTML web pages that can be served statically to each client with no need for regeneration. In an age of per-client dynamically created web pages, static web pages offer a valuable alternative, the benefits of which are discussed in this survey paper. The way in which Static Site Generators work and what technologies they use is described, as well as how content can be rendered on the web in general. An overview of the most relevant Static Site Generators and their features is given, although not all can be mentioned due to the sheer number of Static Site Generators in existence. A concluding recommendation for people looking to choose a tool suited to their liking is provided as well.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Static Site Generators are increasingly popular tools to generate Hypertext Markup Language (HTML) web pages. Before static site generators existed, web developers wrote HTML web pages by hand, which is time-consuming and does not adhere to the Don't Repeat Yourself (DRY) principle, as much of the HTML code is essentially the same, most often only differing in content, while the structure code necessary for the page setup remains the same.

1.1 Basics

Static site generators take care of much of the leg work web developers used to have to do. Only the content and some optional metadata need to be entered manually, the surrounding HTML code is created automatically by the generator. Even fully-fledged websites can be created like this, for example some additional menus, headers or footers to accompany the content.

This generation of web pages can occur only once on the server, when new content is added, or existing content is changed. The generated HTML is then stored on the server and can be served to each connecting client as-is, which is the definition of static web pages. In contrast to that, there are dynamic web pages, which are generated (differently) for each connecting client on the fly. Static web pages have the benefits of faster delivery and less workload for the server, but they are by definition the same for each client, that means there can occur no per-client customization or other dynamic content without some additional workarounds. See Chapter 2 about all possibilities of rendering content on the web and Section 2.3.3 for elaborations on how dynamic content can be rendered into static websites.

1.1.1 How They Work

Most static site generators use some form of markup language, like Markdown, for the creation of content [Gruber 2004]. A templating engine is used to create the HTML. See Chapter 3 for further information about templating engines. A simplified illustration of this can be seen in Figure 1.1.

1.1.2 Example

Listing 1.1 shows a simple example of an input file used in *Hugo* [French 2019], a static site generator written in the Go programming language. The resulting HTML web page generated from this input file can be seen in Figure 1.2.

1.2 Alternatives

Content Management Systems are usually used when hosting a blog with a full blown interface to work on. Flat-file Content Managament Systems are an alternative if the use of any command line or writing code is unwanted.

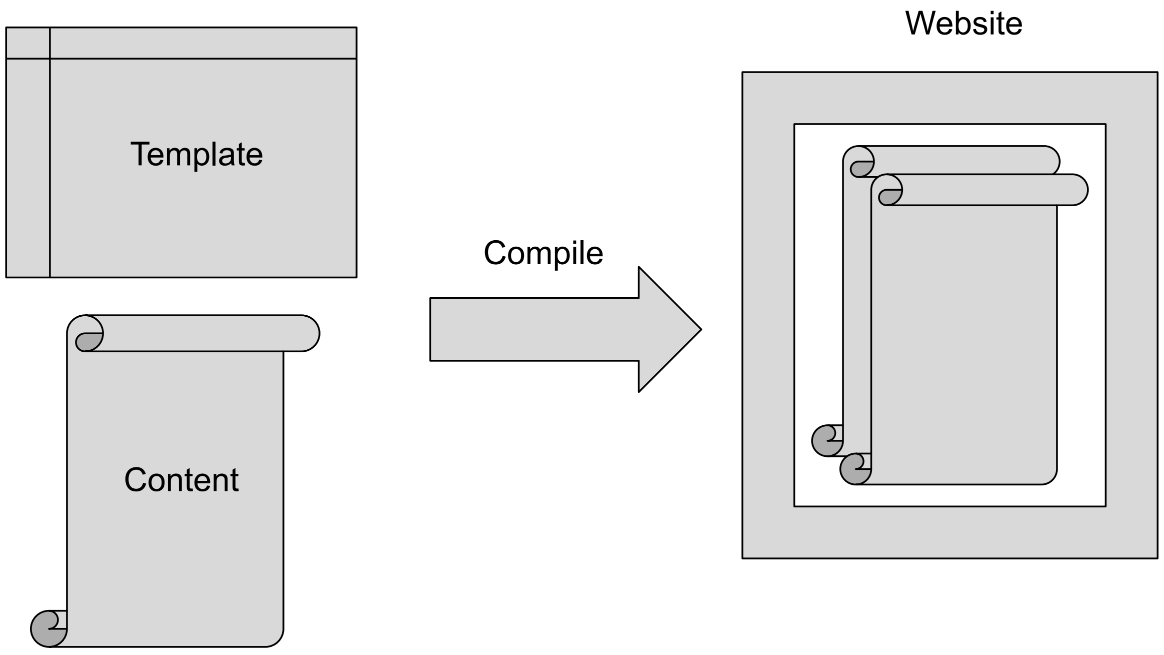


Figure 1.1: The content is taken by the static site generator, which combines it with a templating engine to compile it into a fully-fledged static HTML web page, or if needed, a website. [Redrawn from Kato [2012]]

```
---
author: "Jon Doe"
title: "Test Post"
date: 2019-11-14T11:04:53+02:00
---
# Hello
Lorem ipsum dolor sit amet...
```

Listing 1.1: Hugo Example input file. Metadata is formatted in YAML. Below that is content formatted in Markdown.

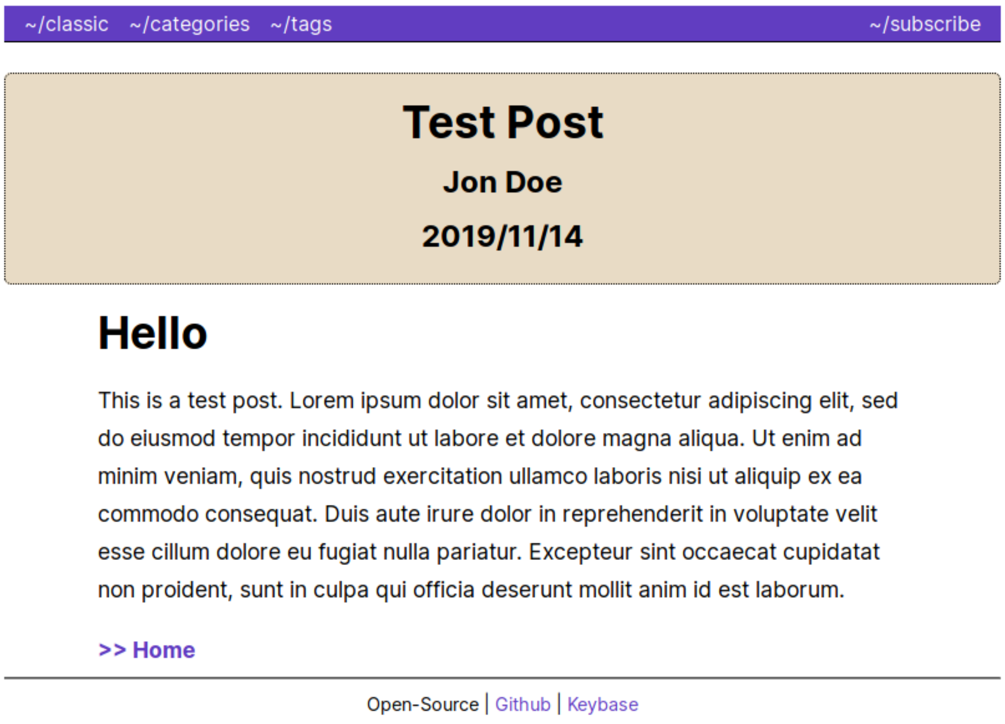
1.2.1 CMS

Content Management Systems are widely known, *WordPress* [Automattic 2019] is the most popular one. In a CMS the project is put on a Webserver and a Database is configured in which all the data is. The configuration of the website, new posts be created, selection of a theme and installing plugins and many more can done via the web interface.

1.2.2 Flat-File CMS

In a Flat-File CMS the project is uploaded to the webserver. Markdown files can also be used which the Flat-File CMS will recognize and automatically create the posts from. Some Flat-File CMSs also have a complete CMS included in which you can create new posts. The advantage of a Flat-File CMS is that everything is saved in files and not in a sperate database.

In our survey we looked at 3 different Flat-File CMSs:



Test Post

Jon Doe

2019/11/14

Hello

This is a test post. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Figure 1.2: Example of Hugo. The input seen in Listing 1.1 is compiled to the rendered HTML seen here. [Screenshot taken by the authors of this paper using the hugo-classic theme [Kellen 2012] under the MIT License [MIT].]

- *Kirby* [Bastian Allgeier GmbH 2019]
- *Grav* [Miller 2019]
- *Pico CMS* [Pellegrom 2019]

While Kirby includes a CMS, it is paid and costs 99€ per site. Grav includes a CMS and a CLI to use it. Pico CMS only works with files, but doesn't have any community forum to ask questions.

Chapter 2

Methods of Rendering Content on the Web

The basic procedure of how content appears in a user's browser is simple:

1. The user requests a page on the server.
2. The server responds with some content that is displayed in the user's browser.

These steps always happen, and they invariably occur in this sequence. However, there are different methods of rendering which influence the timing of the rendering and the entity that takes care of it. These methods are described and compared in the following sections.

2.1 Server-Side Rendering (SSR)

Server-Side rendering means that pages are dynamically generated every time they get requested. That means that the server takes care of fetching the necessary data and the subsequent rendering of a page. When the server is done, a fully rendered page is transferred to the client, which then only has to take care of displaying it [Błaszyński 2018].

2.1.1 Strengths

Rendering content on the server at request time has the following advantages:

- **Search engine optimization (SEO) can be performed with little effort** because crawlers receive an already rendered web page that can easily be indexed. Some crawlers are not able to execute JavaScript code, and therefore everything that's not already rendered in the server's response can't be indexed by them [Góralewicz 2017].
- **Dynamic content can be rendered without client JavaScript** because the pages are generated at request time, and the server can always bake the most up-to-date data into the HTML response.
- Given that the major work of rendering is done on the server, this method puts only **very light load on clients**.

2.1.2 Weaknesses

Having a central server that handles all the rendering tasks, of course, comes with some disadvantages which are listed as follows:

- Server-Side rendered websites **don't scale well with traffic**. As the amount of traffic grows, the server has to handle more and more requests. Because of this, it acts as the bottleneck of the whole setup.
- The application **infrastructure is much more complicated** in comparison to other rendering methods because a server, including a server-side rendering framework, has to be set up and maintained.

2.2 Client-Side Rendering (CSR)

A client-side rendered web page is only fully rendered on the client via JavaScript. When such a page is requested, the server mostly just replies with an empty page containing a single placeholder element and some attached scripts. The scripts are then executed on the client, and they take care of fetching necessary data from potential APIs and rendering the final HTML [Błaszyński 2018].

2.2.1 Strengths

Doing all the rendering on the client has some strengths, which include the following:

- A major strength of this method is that it generally leads to an **increase in usability** because asynchronous updates of the content can easily be integrated. All the rendering already happens on the client, and therefore a rerendering can easily be triggered when desired. That leads to greatly improved usability when compared to full page reloads whenever something changes.
- An application based on CSR **scales very well with increasing traffic**. A big chunk of the work, the rendering, is done by the individual clients, and therefore no chokepoint on a central rendering server exists. In most cases, an API will need to be hosted on a server that could potentially become a performance bottleneck of such a setup. However, it's much easier to perform load optimization of API calls than it is to mitigate performance issues caused by SSR.
- **Dynamic content can be integrated with little effort**. That is because the pages are generated dynamically, and the client can fetch the newest data from an API and consider it when rendering.

2.2.2 Weaknesses

The relocation of work to the clients also leads to some disadvantages which are listed as follows:

- **SEO can be problematic** because some crawlers don't understand/execute JavaScript code, and therefore they would only see an empty page with no content. An empty page can not be indexed, which leads to the page not being ranked in the search engine. According to Góralewicz [2017] only the Google [Google LLC 2019] and Ask [Ask Media Group, LLC 2019] crawlers are able to properly index content that's rendered via JavaScript. Even though Google alone makes up for 62.5% of all search engine queries in the US [Clement 2019], ignoring the other providers still leads to a great loss in visibility.
- This method puts **high load on clients** because the rendering has to be performed by them. That can lead to laggy user experience on weaker devices, especially when the rendering becomes more complex.
- The rendering of content is **highly dependent on JavaScript**, which means that the page is more prone to failures.

2.3 Static Rendering

In this method of rendering the pages are generated in a separate build step, which happens before deploying the website to a server. The server then receives the already fully rendered HTML pages and just has to distribute them to the individual clients when requested [Błaszyński 2018].

2.3.1 Strengths

The strengths of static rendering are similar to those of server-side rendering and can be summarized as follows:

- Statically rendered web pages **display very fast**. The reason for this is that the expensive task of rendering has already been done at the time the pages are requested. They just need to be transferred to the clients without any further transformations. Such pages can also be hosted on a Content Delivery Network (CDN), which makes the transfer even faster.

- This method **takes the heavy load of rendering from the server and the clients** because the content is already prerendered in a separate build step. That means that a potential server is free to focus on handling API calls, and less performance is demanded of client devices.

- Similar to server-side rendered websites, static rendering also results in **effortless SEO**. The search engine crawlers receive already fully rendered HTML content that can be indexed without any JavaScript-related limitations.

2.3.2 Weaknesses

The weaknesses of static rendering all stem from the fact that every page is generated ahead of time. They can be listed as follows:

- **Dynamic content requires more effort** than other methods. That makes sense because the nature of dynamic content conflicts with static rendering. Different approaches on how dynamic content can be rendered into statically rendered pages are elaborated in Section 2.3.3.

- Another weakness of static rendering is that **build times can get quite long** on large websites. This issue can potentially be mitigated by using a high performing static site generator, but even then, it can become a burden when the amount of content grows large.

2.3.3 Rendering of Dynamic Content

There are two fundamentally different approaches to how dynamic content can be rendered into websites generated with a static site generator:

1. Static rendering of dynamic content

2. Dynamic rendering of dynamic content

The first approach is **static rendering of dynamic content**. That means that the SSG fetches all the required data at build time and renders it into static pages. The advantages of this approach are the same as those of static rendering in general, which can be seen in Section 2.3.1. A limiting factor of statically rendered dynamic data is that the underlying data should only change very rarely as every change would require a subsequent rerendering of all pages using it. Another potential issue is that, depending on the amount of data and how it's rendered, build sizes can quickly become exceedingly large.

The second approach is **dynamic rendering of dynamic content**. Here, everything that is static is built into a static frame, which is then filled with dynamic content on the client via JavaScript. That is a reasonable compromise as it still takes advantage of static rendering while using client-side rendering for content that changes frequently. In comparison with the static approach, this approach doesn't suffer from large build sizes and recurring rebuilds. However, it brings a dependency on client JavaScript execution, which means that SEO can be problematic.

In summary, neither approach is better than the other. The decision of which one to choose has to be made in the context of the particular situation. There are, however, static site generators that are better suited to integrate dynamic content than others. Most of the time, these are SSGs which are based on JavaScript frontend frameworks.

Chapter 3

Templating Engines

Templating engines are generating a result document out of a template which fetches some data and includes it into the template.

3.1 Advantages

The advantages of a templating engine are:

- Variables and functions
- Text replacement
- Conditional evaluation and loops

3.2 Example

Listing 3.1 shows the example of a template file of the templating engine Pug. The code gets converted to HTML code seen in Listing 3.2.

3.3 Types of Different Templating Engines

There exist many different templating engines and for each programming language there is one most popular one. For JavaScript there is Pug [Pugjs 2019], which was originally called Jade. For Ruby exists the default one for Ruby, the ERB (Embedded RuBy) engine [Britt and Neurogami 2019]. For Python the most popular is Jinja [Ronacher 2019] and for PHP exists Twig [Potencier 2019]. In contrast to other programming languages there are many more for JavaScript than for any other. As is expected as most templating engines work for the web. Other popular templating engines for Javascript are Handlebars [Katz 2019], EJS [Eernisse 2019], Haml [Clarke 2019] and Nunjucks [Mozilla Corporation 2019].

3.4 Consolidate.js

Concolidate.js [Holowaychuk 2019] is a templating engine framework, where a function signature can be used and one can easily switch between different templating engines, thus it is easy to use another template engine and not change any code.

```
doctype html
html(lang="en")
head
    title= pageTitle
    script.
    if (foo) bar(1 + 5)
```

Listing 3.1: Example syntax of a template in Pug.

```
<!DOCTYPE html>
<html lang="en">
    <head>
    <title>Pug</title>
    <script>
        if (foo) bar(1 + 5)
    </script>
    </head>
```

Listing 3.2: Example of generated code of a template in Pug.

3.5 Syntax Similarities

Certain templating engines are based or inspired by another templating engine. For example Nunjucks is inspired by Jinja2 which is based on Django [Django Software Foundation 2019]. Switching between Nunjucks and Jinja2 is therefore very easy, because they have identical syntax. However, there shouldn't be any native features for JavaScript in use in the Nunjucks template, so the existing template can easily be used for Jinja2 [*FAQ Nunjucks* 2019]. Pug (Jade) is also inspired by Haml and the Twig Syntax originares also from Jinja2/Django.

Chapter 4

Overview of Static Site Generators (Standout Features)

Since there are too many static site generators in existance, in various states of maintenance and popularity, describing all of them in this survey would be an insurmountable task and no benefit to anyone. Some of the more popular ones have been picked out and the features that make them distinct from the others are described in more detail. To get a more comprehensive list of many static stite generators, the website StaticGen [Netlify 2019] could be taken into consideration.

4.1 Jekyll (Ruby)

Jekyll [Maroli et al. 2019] was one of the first static site generators and the one responsible for the reflux of static websites in the world wide web of today, according to Biilmann [2015].

Jekyll was also the first static site generator to introduce FrontMatter, which is the YAML formatting used to specify metadata on top of content files, which many other static site generators now also use. A small example can be seen in Listing 4.1.

4.1.1 Input Languages

- Markdown - For writing the content
- YAML (FrontMatter) - For specifying metadata in posts
- CoffeeScript - To generate JavaScript
- Sass - To generate CSS

4.1.2 Main Use

Jekyll's main use case is suited to blogging, it even calls itself "blog-aware". Jekyll also pioneered the use of GitHub as a hosting platform for static web pages. The statically generated web pages can be pushed to certain GitHub repositories and the rendered HTML content can then be seen on respective `github.io` URLs. This can be done in theory with any static web pages, but Jekyll was the first to streamline the process and make many aware of it.

Another useful thing that can be easily done with Jekyll is transforming existing HTML pages into Jekyll projects. The existing HTML elements can be swapped out for Jekyll's templating elements and from that point on used as templates for further plain text content used as input.

```
---
author: "Jon Doe"
title: "Test Post"
date: 2019-11-14T11:04:53+02:00
---
```

Listing 4.1: Metadata formatted in YAML, called FrontMatter.

4.1.3 Templating Engine

The templating engine use by Jekyll is Liquid [Shopify Inc. 2019], which is the one created and used by Shopify. This templating engine is built around safety due to circumstances about how it is used at Shopify. This means it is not possible to use custom code in the templates used for a Jekyll project. There are however plugins that can handle this.

4.1.4 Content Model

Posts are stored in a directory called `_posts`. All posts must be named according to a naming scheme of the format `yyyy-mm-dd-title.md`. This is more restrictive than it could be, in for example Hugo, the posts can be named in any of `title.<format>`. A different directory called `_data` is available to store data files that can be accessed in post via `site.data` from the template syntax.

4.1.5 Asset Pipeline

The asset pipeline of Jekyll is rather simple, however it does support Sass and CoffeeScript. Many larger Jekyll projects opt to use a third-part build-system such as Gulp or Grunt to handle larger content systems and have features like live-reloading working, which does not work out-of-the-box, in contrast to other static site generators like Hugo, where this is built-in.

4.1.6 Extension Possibility

Plugins do exist for Jekyll and are in fact used in most Jekyll projects. They offer support for progressive web applications, advanced templating features, search engine optimization, et cetera. Self-made Ruby plugins can simply be added to the `_plugins` directory.

4.2 Hugo (Go)

Hugo's [French 2019] main source of advantage and disadvantage is that is is a statically compiled binary executable. Due to that, is is exceptionally fast when building websites, because there is no running interpreter, the installation is easy, as one does not need a Go development environment installed. However, it also makes it virtually impossible to extend the functionality, since there is no easy access to change the binary executable.

4.2.1 Input Languages

- Markdown - For writing the content
- YAML (FrontMatter) - For specifying metadata in posts
- TOML - For writing configuration in the config file
- External helpers (Asciidoc, reStructuredText, or pandoc)

4.2.2 Usability

Hugo is very easy to install, update, and run. A pre-built binary can be downloaded without having to have Go installed, in contrast to most other static site generators. Using it itself is also very easy: Setting up a website only takes a handful of commands in the command line - No coding required.

4.2.3 Community

There is a large community around custom themes people build and share. These themes not only influence the looks of the website, such as the CSS, menus, headers, footers, et cetera, but also offer templates of the content model, already implement responsive web-design and take much of the initial setup work away, if one were to use no theme at all.

4.2.4 Build Performance

Building HTML from the input content files is exceptionally fast. One reason why is because Go is a statically compiled language. No interpreter is run when building websites, which is the case in most other static site generators, which use interpreted languages like JavaScript or Ruby.

4.2.5 Main Use

The main use case of Hugo is for content-driven websites, that is if there is a lot of content that needs to be generated, as Hugo is very fast to do this. Freedom of dependencies and ease-of-use is also a main attraction for users who want an easy-startup and do not want to manage development environments, or do programming themselves.

4.2.6 Templating Engine

Hugo uses the Package template from Go's standard library, but supports also Amber and Ace. The default package templating engine is similar to Liquid, which is used frequently in other static site generators.

4.2.7 Content Model

Hugo has the most powerful content model out of the box out of all static site generators, according to Biilmann [2015] from Smashing Magazine. The content is grouped into a tree structure, and sub directories are URL sections. For example: `http://website.com/posts/2019-11-28-blog-entry.html` would be the local file `content/posts/2019-11-28-blog-entry.md`. The content model supports tags and categories, which can be used to display all posts in a category, or all posts with a specific tag.

4.2.8 Asset Pipeline

The asset pipeline is rather weak in Hugo - There is not even a real "pipeline". When Hugo builds the web pages, it uses assets like pictures from a directory called "static". Live reload works with Hugo's local server, so when changes are made locally the browser automatically displays the newly generated HTML.

4.2.9 Extension Possibility

There is no support for plugins. Hugo comes as a binary, so one cannot easily add functionality to it, but external helpers exist, for use cases like supporting AsciiDoc and reStructuredText instead of Markdown. Also, many of the features that other static site generators pull in via plugins are already offered in Hugo by default, for example dynamic data sources, menus, syntax highlighting, themes, tables of contents, shortcodes, et cetera.

4.3 Metalsmith (JS)

Metalsmith [Segment.io, Inc. 2019] is a SSG written in JavaScript that uses the Node JavaScript Runtime. Metalsmith advertises itself as using plugins for almost every task, simple blog posts for example can be generated using only two plugins, while more intricate solutions will use more plugins. It is only required to install the functionality that is needed for a project.

4.3.1 Input Languages

Metalsmith needs plugins to add input languages:

- Markdown - For writing the content
- YAML (FrontMatter) - For specifying metadata in posts
- JSON - For specifying metadata in posts
- LESS - To expand upon CSS

There are also plugins for other sources available, for example a way to import data from Excel files.

4.3.2 Usability

Metalsmith can be installed with the Node Package Manager. Metalsmith doesn't have any graphical interface and uses the command line and it also requires editing of configuration files and a bit of coding in order to use the plugins. Like many of our examples Metalsmith is also under a MIT-license [MIT], making it free to use, even for commercial purposes.

4.3.3 Community

The community of Metalsmith is not large, but active and provides additional plugins as well as templates and some tutorials.

4.3.4 Build Performance

The build performance of Metalsmith isn't as fast as Hugo, but for most purposes it should suffice, since the advantage of SSGs is that the site is not generated anew with every request.

4.3.5 Main Use

Most examples found during research use Metalsmith to create blogs or smaller websites.

4.3.6 Templating Engine

By default Metalsmith suggests Nunjacks as its templating engine, which also needs to be installed separately, but it can also use other templating engines such as Handlebars or Pug.

4.3.7 Extension Possibility

As already stated Metalsmith is built on the premise that functionality is added with plugins, as such there is a relatively large amount of extensions available. There also exists a section on Metalsmith's website's frontpage that lists a number of plugins.

4.4 Wintersmith (JS)

Wintersmith [Nordberg 2019] is a SSG written in CoffeeScript, which compiles to JavaScript, that uses the Node Javascript Runtime. It is inspired by Blacksmith, another SSG. Like Metalsmith and many other SSGs it heavily builds on plugins. Most functionality can be added as a plugin. Starting with or switching to Wintersmith from other SSGs is made easy, because it uses no special metadata or file structure.

4.4.1 Input Languages

Wintersmith needs plugins to add input languages, but comes bundled with:

- Markdown - For writing the content
- YAML (FrontMatter) - For specifying metadata in posts

Other available input languages are for example:

- LESS - To expand upon CSS
- CSV - For data

4.4.2 Usability

Wintersmith is easy to install with the Node Package Manager (npm) that comes with the Node Runtime. The user interface is based on the command line and no graphical interface is available. It is easy to use and for simple sites not much previous knowledge is required to use Wintersmith. Additionally Wintersmith is under a MIT-license [MIT], making it free for use.

4.4.3 Community

Wintersmith has a decently sized community that provides plugins, templates as well as tutorials.

4.4.4 Build Performance

Wintersmith should be sufficiently fast for most purposes [KS 2017] even if it is not as fast as Hugo.

4.4.5 Main Use

Wintersmith advertises itself to be able to do more than just blogs and can also provide plugins to help creating modern web applications.

4.4.6 Templating Engine

Wintersmith comes with the Jade (now known as pug) templating engine plugin, but it can also use other Javascript templating engines like Nunjucks or Handlebars and more through community plugins.

4.4.7 Extension Possibility

Wintersmith can be expanded with the many available plugins. It is possible to add new input languages as well as choose a prefered templating engine from the choices that are provided. There is also a list provided on Wintersmith's GitHub page that shows some of the most useful plugins.

4.5 Sculpin (PHP)

Sculpin [Dragonfly Development 2019] is written and can be expanded with PHP. The functionality does not stand out much, as it uses a command line interface and the templating engine Twig to generate static sites from Markdown and YAML-Frontmatter. In the space of SSGs using PHP is rare, which is why if working with PHP is prefered, Sculpin is one of a few choices that are available.

4.5.1 Input Languages

- Markdown - For writing the content
- YAML (FrontMatter) - For specifying metadata in posts

4.5.2 Usability

Sculpin is reasonably easy to install and requires Composer, a dependency manager for PHP. While Sculpin can only be used with the command line, usage is relatively simple compared to other SSGs. Thanks to being under a MIT-license [MIT], Sculpin is free to use and expand upon, even for commercial use, which lowers the barrier of entry.

4.5.3 Community

The community of Sculpin is small and only provides a comparatively small number of tutorials, themes and even plugins. If a large community is prefered, Sculpin is not the right choice.

4.5.4 Build Performance

There is not much information on the performance as a SSG, but according to some reviews it does not seem slow compared to the average SSG [Urevc 2017].

4.5.5 Main Use

Sculpin is mainly used for smaller projects like Blogs or personal websites, but there are also some company sites built with it. Since it is free and easy to use but lacks any powerful features it is a good fit for smaller scale projects.

4.5.6 Templating Engine

Sculpin can only use one templating engine which is Twig, a templating engine for PHP.

4.5.7 Extension Possibility

Sculpin can be extended in PHP by the user, since it is open source, or by installing available plugins.

4.6 Frontend-Framework-Based SSGs (JS)

Many SSGs are based on popular JavaScript frontend-frameworks such as React or Vue. All of the examples of frontend-framework-based SSGs listed in this survey use the Node JavaScript Runtime, which needs to be installed to run. These SSGs have a lot in common in terms of usability and also functionality. One field where these SSGs usually excell is the creation of Progressive Web Apps and Single Page Applications. While they seem to be quite similar in many aspects, they are designed and best suited for slightly different applications and workflows. In the end choosing one of these comes down mostly to personal preference. The examples in this survey are Gatsby, React Static, Next, Nuxt and Gridsome.

4.6.1 Input Languages

Like Metalsmith or Wintersmith most frontend-framework-based SSGs need plugins for input languages. Markdown and YAML-frontmatter are basic input languages that all of the examples cab work with, while many other forms of input can be added. These SSGs also support adding external sources for data, for example CMSs.

```
    - components
      - layout.js
    - markdown-pages
      - post_2019-11-25.md
      - post_2019-11-26.md
    - pages
      - about.js
      - index.js
    - styles
      - global.css
    - templates
      - blogTemplate.js
```

Listing 4.2: An example for a file structure working with Gatsby.

4.6.2 Usability

The examples of frontend-framework-based SSGs listed in this survey are not very user friendly as they only have a command line Interface and also require some form of coding in JavaScript in order to create templates and actually generate static sites. One example for a template in Gatsby can be seen in the Listing 4.3. Without previous experience in working with the frameworks they are based on, it is quite hard to learn and might not be the best choice for every use case or user. One advantage for users is that all examples listed here are under a MIT-license [MIT] and can be used and modified for free, even for commercial use.

4.6.3 Community

Most of the frontend-framework-based SSGs have strong communities that are also built on the community of the framework they are based on. The communities provide templates, tutorials as well as plugins to add features.

4.6.4 Build Performance

Build performance varies between the different SSGs but is generally not slow. For most purposes these SSGs should be fast enough.

4.6.5 Main Use

Frontend-framework-based SSGs have an advantage to other SSGs in terms of creating single-page applications and progressive web apps, thanks to the frameworks they use, which provide the functionality and plugins needed without using any external tools.

4.6.6 Extension Possibility

Frontend-framework-based SSGs offer the possibility to install and write plugins to add support for new input languages and data sources or add functionality to the generated sites.

4.6.7 Gatsby

Gatsby [Gatsby, Inc. 2019] is an extension of the React framework that also provides functionality as a SSG. Gatsby can only use GraphQL to query data and doesn't leave any other options. See Listings 4.2 and 4.3 for a Gatsby example.

```
import React from "react"
import Layout from "../components/layout"
import { graphql } from "gatsby"

export default function Template({
  data, // this prop will be injected by the GraphQL query below.
}) {
  const { markdownRemark } = data // data.markdownRemark holds your post data
  const { frontmatter, html } = markdownRemark
  return (
    <Layout>
      <div class="container">
        <div>
          <h2>{frontmatter.title}</h2>
          <h3>{frontmatter.date}</h3>
          <div
            class="content"
            dangerouslySetInnerHTML={{ __html: html }}
          />
        </div>
      </div>
    </Layout>
  )
}

export const pageQuery = graphql`
  query($path: String!) {
    markdownRemark(frontmatter: { path: { eq: $path } }) {
      html
      frontmatter {
        date(formatString: "YYYY MM DD ")
        path
        title
      }
    }
  }
```

Listing 4.3: An example of a template to generate a site from markdown in Gatsby. Based on code provided on https://www.gatsbyjs.org/docs/adding-markdown-pages/

4.6.8 React Static

Similar to Gatsby, React Static [Linsley 2019] is based on React and can generate static sites. The biggest difference is that it does not force the user to use GraphQL to handle data. Compared to Gatsby the community appears to be much smaller and not as invested.

4.6.9 Next

Next [ZEIT, Inc. 2019] is also based on React. Like React Static Next also gives the user the choice to not use GraphQL. The main difference to Gatsby and React Static is that Next is designed to run on a server and provide server side rendering [Bedford 2019], while Next also can provide static rendered pages, it isn't the main focus.

4.6.10 Nuxt

Nuxt [A. Chopin and S. Chopin 2019] is a framework expanding on Vue, that provides static site generation. Like Next Nuxt also provides good support for server side rendering.

4.6.11 Gridsome

Gridsome [H.-J. Vedvik and T. Vedvik 2019] is also based on the Vue framework. Compared to Nuxt it claims to be more optimized for static content. GraphQL is used to pull data from sources, similar to Gatsby. The Community is small compared to Nuxt, Gatsby or Next.

4.7 Overview of SSG Characteristics

All the characteristics of the Static Site Generators can be seen in the following tables: 4.1, 4.2, 4.3 and 4.4.

**	Hugo	Metalsmith	Next.js	Jekyll
Website	https://gohugo.io/	https://metalsmith.io/	https://nextjs.org/	https://jekyllrb.com/
Language	Go	JS	JS/React	Ruby
Note	*	"Everything is a plugin"	Based on React	“a simple, blog-aware, static site generator”, any normal static page can be a Jekyll project
Type	SSG	SSG	SSG	SSG
Input Languages	HTML, Markdown, YAML (FrontMatter), JSON, TOML	Markdown, and many more through plugins	Markdown, JSON, CSV and many more through plugins (none onboard)	Markdown, YAML: Introduced FrontMatter
Usability	Very easy	As simple as you want, but more advanced usage requires coding	Hard, requires a lot of coding	Simple
PWA	only manually	Not out of the box	Yes	Via plugin
SPA	*	*	Yes	*
SEO	Yes	Via plugin	Yes	Via plugin
Community	Community around themes	Small	Tutorials, Plug-ins, templates	*
Build Performance	Very fast	*	*	*
Main use	Content-driven websites	More than simple blogs	PWA, Blogs	Blogs, hosting on GitHub Pages
Input Interface (CLI or CMS)	official CLI (3rd party Frontend interfaces available)	CLI, JavaScriptAPI	*	*
Templating engine	Package template from Go’s standard library	Nunjucks, Handlebars.js, Twig, add template engine via Consolidate.js (many different template engines)	None	Liquid (i.e. no custom code in templates)
Content model	Most powerful content model out of the box out of all SSGs	Take content from source files -> manipulate via plugins (chaining) -> Write to output directory	*	_posts/yyyy-mm-dd-title-of-the-post.md, custom collections, _data folder
Asset Pipeline	Weak	*	*	no built-in support for live reloading, minification or asset bundling, uses Sass and CoffeeScript to generate CSS and JS files
Extension possibility	Plugins: No, but external helpers exist	Plugin-based overall logic - Can be extended to do more than just basic static site generation	Plug-ins available	Simple to extend, many plugins available. Add ruby plugins to the _plugins folder.
License	Apache License 2.0	MIT	MIT	MIT

Table 4.1: Characteristics overview of Hugo, Metalsmith, Next.js and Jekyll

**	**Nuxt**	**Mikser**	**Harp**	**Wintersmith**
Website	`https://nuxtjs.org/`	`https://github.com/almero-digital-marketing/mikser`	`https://harpjs.com/`	`https://wintersmith.io/`
Language	JS/Vue	JS	JS	JS
Note	Based on Vue	Node.js like React	Node.js	Node.js like React
Type	SSG	SSG	SSG	SSG
Input Languages	*	Jade, Eco, Ect, Ejs, Swig, Nunjucks, Twig, Markdown, Textile, YAML, TOML, ArchieML, CSON, JSON5, more through plug-ins	Jade, Markdown, EJS, CoffeeScript, Sass, LESS and Stylus	Pug, Mardown and others through plug-ins
Usability	*	Moderate, requires coding	Easy, CLI	Easy, CLI
PWA	Yes	*	No	No
SPA	Yes	*	No	No
SEO	Yes	*	Yes	Via plugin
Community	Tutorials, Plug-ins, templates	non existent	Tutorials	Tutorials, Plug-ins, templates
Build Perform-ance	*	very fast	fast	fast
Main use	Landing pages, SPA	*	*	Blogs
Input Interface (CLI or CMS)	*	CLI	CLI	CLI
Templating engine	*	Pug aka Jade, Eco, Ect, Ejs, Swig, Nunjucks, Twig, Markdown, Textile, YAML, TOML, ArchieML, CSON, JSON5, support for new engines through plug-ins	Jade, EJS	Jade, others via community plugins available
Content model	*	*	*	*
Asset Pipeline	*	*	*	*
Extension possib-ility	Plug-ins available	Plug-ins available	Only by modifying code	Plug-ins available
License	MIT	Not clear	Freeware and Open Source; own license model	MIT

Table 4.2: Characteristics overview of Nuxt, Mikser, Harp and Wintersmith

**	Roots	Middleman	Kirby	Grav
Website	`https://github.com/jescalan/roots`	`https://middlemanapp.com/`	`https://getkirby.com/`	`https://getgrav.org/`
Language	JS	Ruby	PHP	PHP
Note	No active development	*	*	*
Type	SSG	SSG	Flat-file CMS	Flat-file CMS
Input Languages	Markdown	Markdown, YAML and JSON (config)	Markdown, Kirby-Text	Markdown, YAML
Usability	*	*	Very easy, just copy files on a webserver and use CMS	"Very easy, copy files on webserver and sue CMS"
PWA	No	*	No	No
SPA	*	*	Combine pages on a single page	Combine pages on a single page
SEO	No	*	No	Yes
Community	*	*	Kirby Forum	Grav Forum
Build Performance	*	*	*	*
Main use	Worfklow that's similar to Carrot (NY based company)	more advanced marketing and documentation websites (design-savvy companies)	Blogs, static websites	Blogs, static websites
Input Interface (CLI or CMS)	*	*	CMS	CMS, CLI
Templating engine	Pug/Jade, switch to EJS possible	Ruby (ERB) templates, can be swapped out for Liquid or Haml	PHP template engine, and other via plugins (e.g. Twig)	Twig
Content model	Input in views/, output to public/, i.e. no "content model" more than that	sitemap, data folder, collections, source and corresponding destination files, built on Sprockets	*	*
Asset Pipeline	built-in asset pipeline for CoffeeScript and Stylus	External pipelines (since v4)	*	*
Extension possibility	Asset pipeline and content model easily extensible	Powerful API, creating new plugins is not well documented	Plugins, Themes available	Plugins, Themes, Skeletons available
License	*	*	Paid, 99€ per site, testing free	MIT

Table 4.3: Characteristics overview of Roots, Middleman, Kirby and Grav

**	**Pico CMS**	**Gatsby**	**React Static**	**Sculpin**
Website	`http://picocms.org/`	`https://www.gatsbyjs.org/`	`https://github.com/react-static`	`https://sculpin.io/`
Language	PHP	JS/React	JS/React	PHP
Note	*	Based on React	Based on React	Based on Symfony framework
Type	Flat-file CMS	SSG	SSG	SSG
Input Languages	Markdown, YAML	Markdown, JSON, CSV and many more through plugins (none onboard); can also source from CMSs	Markdown, JSON; can also source from CMSs	Makdown
Usability	Very easy, only copy fules and create/edit files	Hard, requires a lot of coding	Hard, requires a lot of coding	Moderate, requires coding
PWA	No	Yes	Yes	No
SPA	No	Yes	Yes	No
SEO	No	Yes	Yes	No
Community	Not really, only github	Tutorials, Plug-ins, templates	Tutorials, Plug-ins, templates	Themes, Plug-ins
Build Performance	*	fast	very fast	fast
Main use	Blogs, static websites	PWA, Blogs	PWA, SPA	Blogs
Input Interface (CLI or CMS)	Only creating files	CLI	CLI	CLI
Templating engine	Twig	None	None	Twig
Content model	*	*	*	*
Asset Pipeline	*	*	*	*
Extension possibility	Plugins, themes	Plug-ins available	Plug-ins available	Plug-ins, Themes available
License	MIT	MIT	MIT	MIT

Table 4.4: Characteristics overview of Pico CMS, Gatsby, React Static and Sculpin

Chapter 5

Concluding Remarks

A few conclusions can be made considering the investigated static site generators and their surrounding technologies, although it is virtually impossible to define static site generators simply as being better or worse than others.

5.1 Ranking

As can be seen in Chapter 4, most static site generators are able to produce similar results. They do this in various ways, but in the end, similar HTML websites can be produced in any case. Some do offer benefits over others, but then again those have different benefits. Consider for instance Hugo [French 2019], which offers fast build performance and ease of use but has no way of including plugins, in contrast to Metalsmith, which offers great support for plugins but might be more difficult to use.

Due to this mostly even spread of advantageous features there is no definitive “best” static site generator, at least from a purely objective point of view. Some people might prefer one over the other, due to features that more align to their use case or which might be the biggest difference, the underlying technology stack, which is the language in which the static site generator is written, as well as the frameworks it is using, for example React as a front-end framework.

5.2 Recommendation

On account of the above mentioned facts, the simplest recommendation that can be made for people wanting to use a new static site generator is to choose one that is based on a technology that they already know and like. This makes it easier to familiarize oneself with the static site generator and how it works and is being used. Apart from that, the standout features can also be taken into consideration. Again, someone looking for speed might want to opt for Hugo [French 2019]. Someone looking to transform their existing HTML pages into static site generator projects can use Jekyll [Maroli et al. 2019].

Since not all existing static site generators could be described in Chapter 4, it is worth mentioning the website StaticGen [Netlify 2019] again, which offers a comprehensive list of many if not all static site generators in existence, as well as short descriptions of each.

Bibliography

Ask Media Group, LLC [2019]. *Ask.com - What's Your Question?* `https://www.ask.com/` (cited on page 6).

Automattic [2019]. *WordPress.com - Create a Free Website or Blog*. `https://wordpress.com/` (cited on page 2).

Bastian Allgeier GmbH [2019]. *Kirby - The file-based content management system*. `https://getkirby.com/` (cited on page 3).

Bedford, James [2019]. *Gatsby vs Next.JS - What, Why and When?* 25 Sep 2019. `https://dev.to/jameesy/gatsby-vs-next-js-what-why-and-when-4al5` (cited on page 18).

Biilmann, Matt [2015]. *Static Site Generators Reviewed: Jekyll, Middleman, Roots, Hugo*. 16 Nov 2015. `https://smashingmagazine.com/2015/11/static-website-generators-jekyll-middleman-roots-hugo-review/` (cited on pages 11, 13).

Błaszyński, Łukasz [2018]. *Client and Server Side Rendering Static Site Generators*. 21 Dec 2018. `https://espeo.eu/blog/client-and-server-side-rendering-static-site-generators/` (cited on pages 5–6).

Britt, James and Neurogami [2019]. *ERB - Ruby Templating*. `https://ruby-doc.org/stdlib-2.6.5/libdoc/erb/rdoc/ERB.html` (cited on page 9).

Chopin, Alexandre and Sebastien Chopin [2019]. *Nuxt.js*. `https://nuxtjs.org/` (cited on page 19).

Clarke, Norman [2019]. *Haml*. `http://haml.info/` (cited on page 9).

Clement, J. [2019]. *Share of search queries handled by leading U.S. search engine providers as of July 2019*. 16 Oct 2019. `https://www.statista.com/statistics/267161/market-share-of-search-engines-in-the-united-states/` (cited on page 6).

Django Software Foundation [2019]. *Django*. `https://www.djangoproject.com/` (cited on page 10).

Dragonfly Development [2019]. *Sculpin*. `https://sculpin.io/` (cited on page 15).

Eernisse, Matthew [2019]. *EJS*. `https://ejs.co/` (cited on page 9).

FAQ Nunjucks [2019]. 26 Nov 2019. `https://mozilla.github.io/nunjucks/faq.html#can-i-use-the-same-templates-between-nunjucks-and-jinja2-what-are-the-differences` (cited on page 10).

French, Renée [2019]. *Hugo - The world's fastest framework for building websites*. `https://gohugo.io/` (cited on pages 1, 12, 25).

Gatsby, Inc. [2019]. *GatsbyJS*. `https://gatsbyjs.org/` (cited on page 17).

Google LLC [2019]. *Google*. `https://www.google.com/` (cited on page 6).

Góralewicz, Bartosz [2017]. *Going Beyond Google: Are Search Engines Ready for JavaScript Crawling and Indexing?* 29 Aug 2017. `https://moz.com/blog/search-engines-ready-for-javascript-crawling` (cited on pages 5–6).

Gruber, John [2004]. *Markdown*. 17 Dec 2004. `https://daringfireball.net/projects/markdown/` (cited on page 1).

Holowaychuk, TJ [2019]. *Consolidate.js*. `https://github.com/tj/consolidate.js/` (cited on page 9).

Kato, Masakuni [2012]. *Blogging on Jekyll*. 11 Apr 2012. `https://slideshare.net/mackato/blogging-on-jekyll` (cited on page 2).

Katz, Yehuda [2019]. *Handlebars*. `https://handlebarsjs.com/` (cited on page 9).

Kellen [2012]. *Hugo Classic Theme*. 11 Apr 2012. `https://github.com/goodroot/hugo-classic` (cited on page 3).

KS, Ashutosh [2017]. *10 Best Static Site Generators for Bloggers*. 13 Apr 2017. `https://www.hongkiat.com/blog/static-site-generators/` (cited on page 15).

Linsley, Tanner [2019]. *React Static*. `https://react-static.js.org/` (cited on page 18).

Maroli, Ashwin, Frank Taillandier, and Matt Rogers [2019]. *Jekyll - Simple, blog-aware, static sites*. `https://jekyllrb.com/` (cited on pages 11, 25).

Miller, Andy [2019]. *Grav - A Modern Flat-File CMS*. `https://getgrav.org/` (cited on page 3).

Mozilla Corporation [2019]. *Nunjucks*. `https://mozilla.github.io/nunjucks/` (cited on page 9).

Netlify [2019]. *StaticGen - Top Open Source Static Site Generators*. `https://staticgen.com/` (cited on pages 11, 25).

Nordberg, Johan [2019]. *Wintersmith*. `http://wintersmith.io/` (cited on page 15).

Pellegrom, Gilbert [2019]. *Pico - A stupidly simple and blazing fast, flat file CMS*. `http://picocms.org/` (cited on page 3).

Potencier, Fabien [2019]. *Twig*. `https://twig.symfony.com/` (cited on page 9).

Pugjs [2019]. *Pug*. `https://pugjs.org` (cited on page 9).

Ronacher, Armin [2019]. *Jinja*. `https://www.palletsprojects.com/p/jinja/` (cited on page 9).

Segment.io, Inc. [2019]. *Metalsmith*. `https://metalsmith.io/` (cited on page 14).

Shopify Inc. [2019]. *Liquid*. `https://shopify.github.io/liquid/` (cited on page 12).

The MIT License [2019]. Massachusetts Institute of Technology, 26 Nov 2019. `https://opensource.org/licenses/MIT` (cited on pages 3, 14–17).

Urevc, Janez [2017]. *Playing with the Sculpin static site generator*. 29 Jan 2017. `https://janezurevc.name/playing-sculpin-static-site-generator` (cited on page 16).

Vedvik, Hans-Jørgen and Tommy Vedvik [2019]. *Gridsome*. `https://gridsome.org/` (cited on page 19).

ZEIT, Inc. [2019]. *Next.js*. `https://nextjs.org/` (cited on page 18).