Rationale for the SSG

Observations during testing of other static site generators available a few years ago lead to a set of goals for my own.

August 4, 2023

Contents

I experimented with a number of static site generators to build a web site for my own use in 2018. The observations lead to a set of requirements, which are detailed here.

Some comments on Static Site Generator available in 2018

Why yet another static site generator? Pandoc provides nearly everything and gives the desired functionality (code in Haskell, markdown as primary text input, backup with git).

Some comments on other Static Site Generator

Constructing a simple homepage, perhaps with a blog and some images can be done in a few hours, possibly days with a tool like wordpress. To make a homepage satisfying some special requirements takes a bit of planning of content and how it can be served.

I have tried a few tools to produce an homepage for an university researcher, but I observed in late 2018 some limitations where I desired simpler handling or more flexibility for my use:

- Not easily (i.e. out of the box) to make work with `markdown` (e.g. the often used WordPress),
- Hard to exclude commercial interests[1] and connections to unwanted services[2],
- Missing integration with BibTex to produce references and a list of publications from BibTex files,

[1] especially from the companies and agencies which are trying to convince us that they constantly improve their service "to serve us better" and force us to change our code accordingly

[2] (for example, WordPress seems to link Google Analytics and similar by default

I prefer to work in the language I know already and not to learn a new set of obscure quirks (e.g. WordPress, Sprinkles), and shy away from approaches "batteries included" (e.g. Jekyll or Hakyll; a similar comment) by the author of an Haskell based approach, forcing users to learn lots of detail, and look for *extensible* and *composable* tools, following the initial Unix philosophy[3].

[3] whatever *composable* means for a site generator?

I was very impressed with static site generators, e.g. SitePipe which demonstrated how much functionality is available in packages (e.g. from Hackage)[4].

[4] The predecessor `SitePipe` proveded some inspiration for SSG.

I later learned of the Multimarkdown-CMS, which seams also to show how much can be achieved with current, existing packages. Unfortunately, it converted into a closed system for the Mac.

In searching for how to adapt the Tufte style to the design of a home page I found Jekyll, which uses the Ruby environment but includes pandoc and Tufte packages adapted to pandoc. It demonstrated that a webpage *Tufte style* is possible.

Impressions of the SSG sprinkles

My impression with trying to use sprinkles, which I hoped could satisfy all my needs.

Reasons to try it

Reasons to try it

I spend some time with trying Sprinkles as a foundation for my own homepage. It was attractive because it seemed to fulfill most of my requirements:

- it is written in Haskell,
- uses `Ginger`, the Haskell implementation for the `Jinja2` theme languages,
- it builds a static site which can be uploaded to a host of my choice,
- is design and setup with `YAML` files,
- clean design using a compiled, site independent engine; all site design is in files,
- support for `markdown` (using `pandoc`).

Some points I perceived as rather negative:

- small (very small) developer and user base[5],
- static site generation is possible, dynamic sites goal,
- issues with caching between invocations (makes development difficult).

[5] Is this better than *roll you own*?

Observations

Observations

During my tests, I found some things which could be improved and which became also ideas which were later included in my SSG design.

Button to get clean start

Provide a method to force a clean restart and, if possible, force it automatically when certain files change. For example, when `project.yml` changes, do a clean restart without user intervention.

Breaking the program in independent services

The monolithic nature with a very large number of dependencies makes it hard to maintain and probably difficult to attract others to contribute.

For example:

- factor out, e.g. string conversion, error handling, file IO[6]
- avoid import of qualified `Prelude`[7]

[6] my *uniform* approach

[7] Despite the well known issues with the current Prelude; I prefer to use it unchanged and work arount the issues, mostly to make code integratable with code written by others.

I think it could be possible to have a separate program for `bake` and one for `serving`, with a common sprinkles-core.

With `ginger` I had the impression that part of the complexity comes from using an extended version of JSON.Value.[8]

[8] I understand the reasons for adding specific types but feel it would be better to wrap the type for `Http` around a JSON.Value and not produce a separate value which then needs conversions etc. etc.

Many small files connected by same parameter names

The structure of the different YAML files to define the design, together with the templates (in `Jinja2`), the `css` styles and others are hard to grasp and keep aligned.

Theme folder

I liked the `Pelikan` approach to have a `theme` folder, which separates the theme from the remainder.

Preview not correctly rendered

The preview operation seems not to properly treat markdown.[9]

[9] The three tick style, which should give a typewriter font, gives spaced text.

Parameter for pages and posts

I would prefer a structure for the pages as two `yaml` docs, separated by `----`, where the first contains proper `yaml` with key - value pairs and the second the markdown text.[10]

[10] Pandoc has adopted this approach and I use it for SSG.

Archetypes for pages

I liked the idea to define *archetypes* for pages and a function, which defines the values and an `archetype` to the proper place and fill the values correctly.

Terminology

I think

- `theme` for the overall layout arrangements (including styles) is a nice word, and I prefer it over blueprints.
- `byline` seems to be an established term for the subtitle of a newspaper and in analogy to a blog.

Conclusion

Conclusion

I was impressed with `sprinkles` enough to try to combine it with `sitepipe` and later `slick` which then moved to use `shake`. I eventually decided to go directly to build on top of `pandoc` and `shake`[11].

[11] all packages can be found on hackage.

Principles of SSG design

Why yet another static site generator? Pandoc provides nearly everything and gives the desired functionality (code in Haskell, markdown as primary text input, backup with git).

Goals

Goals

- Separate the presentation from the content.
- The structure of the site should map directly to the directory and file structure of the stored content.
- All content should be ordinary text (UTF-8) files, which gives a choice of tools for editing, version management, backup and guarantees long term viability.
- Connect the tools with a full programming language.
- Build a SSG from available packages in Haskell, to reduce the code which requires maintenance.
- Demonstrate integration using the "uniform" approach to wrap packages in integrable interfaces.

Packages available in Haskell

Packages available in Haskell

A number of tools are available from Hackage:

- Pandoc
- shake
- twich
- git
- citeproc
- doctemplates

They should be used as far as possible!

Well designe tools combine cleanly

Well designe tools combine cleanly

Tools which survive the test of time provide abstractions which combine without unexpected interactions and confusions.

Programmable

Programmable

It is my experience when adaptation is needed anything short of a complete (and well designed) programming language leads to an infinite sequence of special case additions bolted on with some ugly screws; I have the impression that this is a imitation of e.g. Sprinkles.

Uniform interfaces for packages used

Uniform interfaces for packages used

Wrap packages into a small interface layer to locally hide differences between packages which hinder integration. In Haskell, such differences tend to show up as multiple options to achieve more or less the same ends. I tend to think this is preferably over the use of a `special prelude` which makes collaboration with others difficult.

My **uniform** approach

- use `Text` as the primary representation and use `uniform-strings` to convert to and from other representations (with local deviations from the rule)
- all functions are pure or in the `ErrIO` monad (`ErrorT Text a IO`); operations in other monads are wrapped.
- represent path to files as with `Path`[12], use `uniform-fileio` for all operations to use the same interface for different file types and use `TypedFiles` to connect file extensions to semantics (i.e. code to transform from the external to the internal format),
- write top level code in a basic form of Haskell and eschew use of special features, especially not relying on `Template Haskell`(see Haskell style).

[12] Avoid the more general `FilePath` type!

Separate Theme *and* Content

Separate *Theme* and *Content*

The data should be separated from the style of presentation, to

- allow changes in the data without entanglement in the typically tricky descriptions of style,
- changes in the style must be applied regularly on all content and should not require editing already existing data.

The theme (templates, css etc.) and the content should be separated, with a documented interface. Default locations for theme and content can be adapted to needs.

Performance

Performance

SSG is mostly a proof of concept and demonstration for small academic homepages, it is not optimized for humongous web pages for large organizations. Performance is not designed in[13]; if performance is too slow for a specific use, localize the issue and tweak the code were performance is an issue.

The use of twich and shake gives a nearly dynamic behavior: changes are reflected quickly in locally served pages, when `ssgbake` is run with the `-w` switch.[14]

The conversion of the markdown to latex is, as currently implemented, slow; it requires separate processes for different steps in the conversion in a suboptimal fashion and could be improved.

[13] The caching mechanism of `shake`, however, should make even large web sites viable

[14] The mechanism works well for local changes in a page but requires some tweaking when an index page is produced during `bake` by collecting data from multiple other pages in a directory; typically delete at least the `index.html` page in the baked homepage and re-run ssgbake.

Character file based to facilitate backup and version management

Character file based to facilitate backup and version management

The storage of content should be in text files which can be edited with any editor[15], versioned with `git` and backed up with ordinary tools[16].

[15] my preference is currently `VScode`, but no special features are relied on

[16] Git operations can conflict with other tools to synchronize file content between installations (e.g. `syncthing`).

Let the directory structure reflect the structure of the site

Let the directory structure reflect the structure of the site

The primary structure of a site should be reflected in the directory structure where the files for the pages are stored.

It is acceptable that the file names can be restricted (e.g. must not include spaces) and facilities to translate the file names into readable titles must be provided.

Many site generator, especially flexible content management systems, use databases like SQL for storage of date; for small sites, full function databases are too complex and notoriously difficult to integrate, backup and vulnerable to attacks from hacker.

Customization

Customization

A web site is infinitely customizable; the difficulty is to decide which customizations are provided at a *common user* level, which others are accessible only through tweaking the underlying technology and how accessibly such tweaks are.

I have opted for a minimal set of options to customize:

- a banner image with two lines of text,
- a ribbon of links to directories,

and most everything else is simply text in sources for web pages.

Any further adaption relies on some understanding of the underlying technology; accessible is

- the way text from web pages is arranged as `html` file in a template,
- layout is changeable in the `css` files.

Customization beyond will likely require forking the source code and changes in it.

Documentation ***not yet done***

Documentation **not yet done**

Desirable features

Desirable features

Support for the development of a web site

It should be possible to add new pages without them immediately going live.

Allow some parts of a web site to be protected from the public

No everything in a web site should be automatically visible to everybody; it should be possible to protect some pages or groups of pages with passwords or some similar means.

What remains to be designed after deciding on the tools?

What remains to be designed? The description with in the abstract will be followed up with the specifics.

content as markdown text

content as markdown text

Pandoc allows a metadata block in YAML before the text written in markdown[17].

The design must fix the information to include in the metadata block and decide on the markdown extension to be included.

A decision on the editor is not required; it is recommended that the editor used should include spell checking tools for multiple languages. I think that spell checking is part of editing the source and not part of the baking of a site.

I wish tools to systematically process text from input on an US-keyboard where input of accented characters and similar (umlaut, "ñ", "¿" etc.) is difficult.

It should be possible to collected several shorter markdown texts and produce a booklet.

[17] Pandoc markdown has many extensions. There is an effort underway to specify markdown more precisely as `commonmark`. The extensions must include at least - footnotes - references to be included from bibtex - images - hyperrefs - table of content

Build Site Generator around Pandoc

Build Site Generator around Pandoc

Pandoc translates many different text file formats into `html`, it can handle BibTex references in text and produce publication lists (with pandoc-citeproc now with citeproc.

It works well with doctemplates, which is a small template system[18] with just conditionals and loops. It is produced by the same author as `pandoc`.

Pandoc works with conversion of files into value. Files can include metadata as YAML blocks in the text source.

It is possible to produce `pdf` files from the blog entries, which gives a printable format (better than printing the `html` document).

Pandoc uses internally JSON, which in Haskell means using aeson.

[18] similar to `moustache`

Directory structure

Directory structure

The design fixes the file structure: theme and content (dough) is separated from the baked site. The source for the web pages(`dough`), the layout and appearances (`theme`) are stored in a directory. The produced web pages go into a different directory (`baked` site).

The resulting `html` files served are stored elsewhere. (Storage of Site Data)

Description of site source layout in a text file

Description of site source layout in a text file

The layout of the source directories and the target directory can be set out in a YAML file (`settings.yml`). In the same file, other general parameters of the web site can be included as well.

Principle: only one settings file in structured text format (YAML).

Private and public, publish and publish state

Only pages which are not containing private material (i.e. are public) and which ready to publish (i.e. not a publish)

Which web pages should be published

Which web pages should be published

The publicly visible homepage should only include material which the author deems *public* and *ready for publication*. It must be possible to collect in the directories material which is in preparation and not yet ready for publication but also material which is of a private nature and not visible in the finished homepage.

Separate readiness for publication *and* visible to everybody

The web is in principle a medium where material is accessible to everybody - unless restricted. It must be possible to produce version of the homepage for public visibility and other versions which are restricted.

There are two reasons for excluding material from public visibility:

- not yet fulfilling quality standards, e.g., spelling not checked, incomplete content, only a first idea sketched, etc.
- private material which for should not be available to everybody or must not be included in a published version for, e.g., lack of copyright (i.e. material where somebody else has the copyright and is not public domain).

Coding in the `yaml` head of every blog page

In the head of every blog page two keywords are included:

- visibility: which can have values `private` or `public`.
- version: which can have values `publish`, `draft`, `idea`.

Current implementation (version 0.1.5.1 of SSG)

Only markdown files with `public` and `publish` are baked into the site. Two switches `-d` and `-p` are included to include files which are not intended for public view or not ready for publication to include such files in the bake process. Be careful not to have the produced site served on a publicly visible port!

Three options to manage a home page

Some of my friends confronted the same question and found different solutions. I see three options discussed here.

There are probably many ways to build and maintain a home page. I see three models that some of my friends use:

- Write HTML directly,
- write `markdown` text and transform it into a web page, or
- use one of the ready-made *complete* programs for managing a web presence.

Write the page directly in HTML

Write the page directly in HTML

Colleagues who started web sites very early and learned HTML when MOSAIC[19] first appeared.[20] Some still write HTML and maintain their web sites[21].

[19] https://en.wikipedia.org/wiki/Mosaic_(web_browser)

[20] I remember having MOSAIC on my Macintosh then, but was busy with other things and didn't see the potential.

[21] e.g. https://web.eecs.umich.edu/~kuipers/opinions/old-web-page.html

Complete systems for managing a web presence

Complete systems for managing a web presence

The large market for tools to help with web presence has spawned a number of more or less complete packages, of which `Wordpress` is perhaps the best known. I have found that such complete systems, advertised as *batteries included*, are easy to get started with, but then have a steep learning curve to figure out all the parts you have no intention of ever using and sometimes the solutions offered still do not include the one you want.

Often the systems are easy to get into but hard to get out of: content is in a proprietary format and the user is trapped.[22].

[22] Getting out of Wordpress was all right, but still a loss of investment in non-portable tricks

A UNIX tools approach: Markdown and Pandoc

A UNIX tools approach: Markdown and Pandoc

Unix has been successful at building tools that can be combined and reused. I thought that combining the `markdown' language, which helps the author focus on content, and`pandoc' to translate content into HTML, with a few more tools to manage the site, could be an interesting project.[23]. So my homepage is produced

[23] I was not alone with such ideas! [06rationale/009aboutSprinkles.md]

using `daino`, a package written in Haskell that runs on both AMD and ARM hardware[24].

[24] Raspberry Pi 4

The combination of these tools allows to produce PDFs that give nicer printed pages and I plan to experiment with this.

Produced with 'daino' (Version versionBranch = [0,1,5,3,3], versionTags = []) from /home/frank/Desktop/myHomepage/Essays/SSGdesign/06rationale/index.md with latexTufte81.dtpl arguments booklet