1

Documentation of the Static Site Generator (SSG)

Design of the Static Site Generator (SSG) daino.

August 4, 2023

Contents

I Introduction to the Static Site Generator (SSG) 4

The Static Site Generator 6

My Goals with SSG 6

Installation and test for functionality 6

Installation and basic test for functionality 6

Build your site with SSG 8

Build your own site 8

The overall setup of a site 9

Details of the Settings for a site 9

Topical subdivision of content 10

Landing page 10

Resources directories 10

The structure of the page files 11

The `YAML` header 11

Web page content 11

Index pages 12

Referencing images and other static content 12

Pages rendered as PDF 12

II Rationale for the SSG 13

Some comments on Static Site Generator available in 2018 15

Some comments on other Static Site Generator 15

Impressions of the SSG sprinkles 17

Reasons to try it 17

Observations 17

Button to get clean start 17

Breaking the program in independent services 18

Many small files connected by same parameter names 18

Theme folder 18

Preview not correctly rendered 18

Parameter for pages and posts 18

Archetypes for pages 18

Terminology 18

Conclusion 19

Principles of SSG design 20

Goals 20

Packages available in Haskell 20

Well designe tools combine cleanly 20

Programmable 21

Uniform interfaces for packages used 21

Separate Theme *and* Content 21

Performance 21

Character file based to facilitate backup and version management 22

Let the directory structure reflect the structure of the site 22

Customization 22

Documentation ***not yet done*** 23

Desirable features 23

Support for the development of a web site 23

Allow some parts of a web site to be protected from the public 23

What remains to be designed after deciding on the tools? 24

content as markdown text 24

Build Site Generator around Pandoc 24

Directory structure 25

Description of site source layout in a text file 25

Private and public, publish and publish state 26
Which web pages should be published 26
Separate readiness for publication *and* visible to everybody 26
Coding in the `yaml` head of every blog page 26
Current implementation (version 0.1.5.1 of SSG) 26

Three options to manage a home page 28
Write the page directly in HTML 28
Complete systems for managing a web presence 28
A UNIX tools approach: Markdown and Pandoc 28

III The use of daino, a Static Site Generator (SSG) 30

Architecture of the bake process 32
Command line processing: 32
Processing of Layout (file `settingsN.yaml`) 32
Watching for changes 32
Shake for rebuilding 33
Transformation of filepath to Path 33
Processing 33
Issues : how to organize regression tests 33

Command Line Interface (CLI) for bake 34

30. PackagesUsed.md 35
Pandoc 35
Templates 35
Watching file change : Twitch 35
Caching 35
JSON 35

40 use of pandoc . md 36
Read Markdown 36
Extract all information into `Context` 36
Convert the Data to target format 36
Fill the converted pieces into template. 36

The programs to generate a web site available in 2018 did not satisfy my expectations but I felt that most of the tools required where available. Thus I embarked on building **Yet Another Static Site Generator** with the distinct properties:

- includes an searchable index to my publications for download which is produced from a `bibtex` database.
- produces for all web pages a printable `pdf` file,
- deals reasonably with a homepage using multiple languages (in my case american english[1] and swiss-style german[2]),
- follows the page layout promoted by Edward Tufte,
- adapts to varying screen sizes using the `w3c` templates

[1] technically `en_US`

[2] aka "hauchdeutsch", technically `de_CH` meaning not to use "ss" in stead of "ß"

It is written in Haskell und uses as much as possible well tested tools, e.g.

- `pandoc` to convert the pages to `HTML` and `pdf`,
- `shake` program for changed inputs.
- `w3c` templates

Part I

Introduction to the Static Site Generator (SSG)

Starts with the design goals and a review of the rationale for a web site content manager. It then covers installation and the instructions to adapt the program to serve your own homepage!

The programs to generate a web site[3] available in 2018 did not satisfy my expectations but I felt that most of the tools required to build a static site generator where available. Thus I embarked on building **Yet Another Static Site Generator** adaptet to the needs of an academic[4].

- searchable list of papers ready for download,
- texts readable in a browser but printable as `pdf`.

After a short introduction to the Static Site Generator SSG follow the instruction to download and to adapt the program to produce a personalized homepage. Not much to do other than organizing content pages in directories, include title and abstract to each content page and add a title and abstract tot the `index.md` page in each directory.

[3] a.k.a. content management systems

[4] The web contains a surpsing amount of advice,from - a consultant, or older, from 2012, by publisher, or - current to go to a static site generator and markdown, 2, - yet another commerical service and - another howto

The Static Site Generator

My goals for daino. What is different from other Static Site Generators? The design goals and rationale and functionality testing.

My Goals with SSG

My Goals with SSG

The use case is my own homepage[5] with requirements typical for an academic researcher. and should be built from available packages in Haskell[7].

- Use an (inexpensive) host server[8].
- Allow me to use a page layout following Tufte css.
- Force a strict separation of *content* and *presentation*[9].
- Look for simple handling and long term stability.

[5] I will use the term **site** (or *web site*) for a set of connected **web pages** (or just *pages*) which can be accessed through a web browser using the `world wide web` technology[6].

[7] especially `pandoc` and `shake`, which reduces the effort to maintain the code using using my "uniform" approach to wrap packages in integratable interfaces.

[8] There are some servers free of charge, e.g. github or google, but I prefer indepence and looked for a basic web server, which cost me Euro 3 per month.

[9] here called `dough` and `theme`, which is baked into the web site.

Installation and test for functionality

Installation and test for functionality

I have tried a number of web site generator programs and found that a test installation and checking the methods for customization is the fastest way to identify what suits my requirements.

Such a test consists of two steps: (1) install and check funtionality of installation with a test[10] and (2) build your site.

[10] The `hello world` test for a site generator.

Installation and basic test for functionality

A basic test for functionality is:

- Clone or copy the code from github[11].
- Change into the `ssg` directory and install with `cabal install` (or perhaps better with stack `stack install`) which produces `ssgbake`, the program which converts (*bakes*) the content into a static site.
- Run `ssgbake -tsw` which produces a test site in your home directory `~/bakedhomepage` which is served on port 3000.

[11] `git clone https://github.com/andrewufrank/SSG`

- Open in your browser `localhost:3000` and you should be greeted by the landing page of the test homepage.
- Edit, for example, the file `ssg/docs/site/dough/Blog/01blog1.md` and observe how the web page is adapting (after refreshing the browser cache!).

If you are satisfied that the installation works, you can proceed to build your own site!

Build your site with SSG

The steps necessary to build a site.

Build your own site

Copy the content of the `ssg/docs/site` directory to where you would like to locate your homepage and rename it to `myhomepage` or whatever directory name you fancy.

I would start `git` in this directory to achieve a flexible backup on the site with `git init`. A suitable `.gitignore` is already in the copied directory and may required adaptation.

Adapt the file `ssg/docs/site/settings3.yaml` minimally[12] with a editor for program text files (i.e. not `office`) for:

- the location of folders, at least for
  - dough: the folder with the source of your site
  - baked: the foler where you expect the generated site (could be, for example, `/var/web/` or `~/bakedhomepage`)
- the port the server is using, when run `ssgbake -s` (default is `3001`)[13]
- `menuitems`: the first levell of subdirectories for the web page files.

After adaptation restart with `ssgbake` in the directory of your homepage and the homepage will be produced, adapted to your needs.

Customization is

- in the `settings` file, and in
- web page files in the subdirectories to the `dough` directory.

The example site in `ssg/docs/site/dough` contains examples for the `settings` file and for web pages with solutions for different uses, e.g. references to images, literature.

All easily customizable aspects are in files and no new compilation of `ssg` is needed[14].

Under the `dough` directory you can include content, typically organized in subdirectories. Each web page corresponds to one file, including the files linking other files in subdirectories.

[12] For more details

[13] The possible switches are - `-s` to start a server, - `-q` for quick, meaning not to produce pdf files, - `-w` to watch files changing and re-bake them automatically.

[14] Recompilation may be needed for new versions of `ssg` or new versions of compilers; it is recommended, but probably not required, to delete the baked website and rebuild it completely.

The overall setup of a site

The file describing the overall setup of a site.

Details of the Settings for a site

Details of the Settings for a site

I will use the term **site** (or *web site*) for a set of connected **web pages** (or just *pages*) which can be accessed through a web browser using the `world wide web` technology[15].

The settings are all collected in a single YAML file[16]. The annotated file for the currently running site can probably serve as a concrete example.

The settings start with `siteLayout`, which gives the directories of the sources for

- theme: where the details of the appearances of the content are fixed,
- dough: the source text for the web pages,
- baked: where the converted files for the web site go; this may be `/var/www/html`[17],
- masterTemplateFile: the template which determines the layout of the converted html - probably use the one provided and adapt later if necessary.
- blogAuthorToSupress: name or names of the authors of most of the material on a site, which should not be repeatedly shown as authors

The content must use the keywords that the theme set up; it is possible to produce with the same theme (i.e. the same directory with the same files) different web sites from different source directories. It is likewise possible to produce different `baked directories` which are independently served from different theme and the same content files.

The `localhostPort` gives the port used by the server created with the `-s` switch of `ssgbake`.

The `siteHeader:` needs values for `sitename:`, `byline:`, `banner` (an image[18] to place by default at the top of all pages) with a `bannerCaption`, a text which can be read if the image not visible.

Last, the entries of a *static menu* are given as `menuitems:` which is shown as a ribbon under the banner page. They consist of a

[15] Following the seminal ideas of Tim Berners-Lee

[16] The current specification of YAML, but there are perhaps better explanations

[17] The default web root for NGINX

[18] preferably wide and narrow; 1024 by 330 pixels works well

- `navlink`: wich is a relative adress to a directory, usually within the `dough` folder.
- `navtext`: the text shown for the link.

The settings file is read each time `ssgbake` is started and content is `baked`; changes are burnt into the converted site and after changes, the site should be rebuild[19].

[19] Just delete the `bakedHomepage` directory and rebuild with `ssgbake`.

Topical subdivision of content

Topical subdivision of content

Usually the content of a site is divided in some topics, e.g. `contact`, `publications`, `blog`. The content for each topic, i.e. the markdown files, are collected in these directories.

Additionally an `index.md` file must be added, which serves as a introduction to the content; a sort of `table of content` is appended automatically and facilitates navigation with clickable links.

Landing page

Landing page

The landing page, i.e. the page shown when the URL of the site is opened. It typically contains a general introduction and links to the major pieces - possibly with some explanation.

The *landing page* of the homepage will be produced from the file `index.md` in the root (`dough`) folder of your homepage using the theme given in the settings file; no special rules or provisions!

Resources directories

Resources directories

Directories to include resources[20], e.g. images or pdf files[21], which are references in other web pages and served can be added wherever convenient. Their location are mentioned in the references included in the source texts for the web pages they reference.

[20] `resources` is a reserved name for directories in SSG; these directories are not searched for web content and should only contain static content, which is references from other pages.

[21] currently only files with extensions `jpg`, `JPG` or `PDF` are dealt with, but extension is a simple change in the Haskell source, specifically in `Shake2.hs`.

The structure of the page files

The structure of the source files for the web pages consist of a header (using YAML syntax) and the page content written in markdown.

The `YAML` header

The `YAML` header

The first part of each web page describes the page. It is fenced off from the page content proper by `---` lines above and beyond. It follows the `YAML` syntax:

```
---
title: text which becomes the title of the page
abstract: typically a multi line text describing the page.
        It becomes the abstract of the page and is shown
        together with the title on the index pages.
author: the author of the page,
        there is a mechanism to suppress this
        for the author of a site
        ([see](/Essays/SSGdesign/004settings.html))
keywords: some descriptive keywords.
date: 2019-03-05
image: if present a reference to the image file
        which will become the pages banner
        (if blank, the default site banner image is used).
bibliography: a reference to the `bib` file
version: publish or draft
visibility: public or private
---
```

Web page content

Web page content

It is followed by the text written as markdown.

- titles are marked with `#` and `##`, which give second and third level titles[22].

  For more details of the (Pandoc) markdown syntax see.

[22] The text after the `title:` keyword in the header gives the first level.

Index pages

Index pages

The structure of the site is revealed to the user through `index pages`[23]. They list the titles and abstracts of the web pages included in a directory, starting from the `root` in a hierarchy. The pages are `clickable` and permit navigation[24].

The index pages must be started by the author of the site as a file `index.md` with keywords

- indexPage: true
- indexSort: title

where the `indexSort` field indicates the order in which pages are listed. A sort by `title` sorts the pages by their filename, which permits to use filenames starting with a number to achieve a specific order.

Alternatives are sort by `data` or `reverseDate` (newest first).

[23] `index.html` files

[24] In addition to the ribbon under the banner image which is always linking to the major subdivisions, listed in the `settings` file and clickable `sitename`.

Referencing images and other static content

Referencing images and other static content

The references can be either absolute to the web root[25], i.e. the directory in which the `dough` is placed or relative to the location to the current page file[26].

Remember that the references must include the `.html` extension of the files in the baked form and not the `md` extensio of the original content files.

It is often useful to place the static content in a `resources` directory[27] in the same directory as the pages for a topic.

[25] I.e. starting with "".

[26] The directory name, not starting with "".

[27] with exactly this name!

Pages rendered as PDF

Pages rendered as PDF

For every web page transformed to `html` a corresponding `pdf` is produced, using the `KOMA` tools for `latex` and rendered as a `scartcl`.

The pdf format uses footnotes at the foot of the page, whereas the footnotes in the web output are pushed to the margin[28]. The bibliography in both output formats are at the end of the page.

[28] Tufte style

Part II

Rationale for the SSG

16

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

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[29] and connections to unwanted services[30],
- Missing integration with BibTex to produce references and a list of publications from BibTex files,

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[31].

I was very impressed with static site generators, e.g. SitePipe which demonstrated how much functionality is available in packages (e.g. from Hackage)[32].

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

[29] 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

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

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

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

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[33],
- static site generation is possible, dynamic sites goal,
- issues with caching between invocations (makes development difficult).

[33] 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[34]
- avoid import of qualified `Prelude`[35]

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.[36]

[34] my *uniform* approach

[35] 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.

[36] 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.[37]

[37] 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.[38]

[38] 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` [39].

[39] 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`[40], 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).

[40] 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[41]; 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.[42]

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.

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

[42] 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[43], versioned with `git` and backed up with ordinary tools[44].

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

[44] 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[45].

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.

[45] 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[46] 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.

[46] 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[47] first appeared.[48] Some still write HTML and maintain their web sites[49].

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

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

[49] 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.[50].

[50] 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.[51]. So my homepage is produced

[51] I was not alone with such ideas! [o6rationale/009aboutSprinkles.md]

using `daino`, a package written in Haskell that runs on both AMD and ARM hardware[52].

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

[52] Raspberry Pi 4

Part III

The use of daino, a Static Site Generator (SSG)

33

Design of a Static Site Generator (SSG).

The design of my Static Site Generator is explained here and gives a detailed account of its functioning.

Architecture of the bake process

> The bake process gradually converts the source texts into texts a html server can use (primarily HTML, PDF, JPG) and adds the supplementary files (mostly CSS to describe appearances.)

The architecture, i.e. the combination of implementations of functions to achieve the overall functionality of SSG, can be seen as steps and each step processing an input into some formats which are used by the next.

Command line processing:

Command line processing:

The standard Unix-style command line analyzes the CLI input and passes it to the program. It establishes the directory in which the command was issued.

Processing of Layout (file `settingsN.yaml`)

Processing of Layout (file `settingsN.yaml`)

List of the directory names and locations - to give flexibility on different distribution of the relevant directories. It is possible to have the code, the content (dough) and the directory where the served files are stored in three different locations.

Watching for changes

Watching for changes

The use of `twitch` to watch for changes in the directories where input files exist and triggering the shake organized rebuilding process removes all tests for file changes in one point. If a change is detected, shake is called.

Given that shake is only redoing what is strictly necessary and caches older results, makes false positives — alerts to changes which are not substantiated — not dangerous and can be ignored.

Shake for rebuilding

Shake for rebuilding

Shake is checking for changes in the needed input files with precision and starts redoing what is necessary to update the result - filtering out false alerts from watching for changes.

Shake relies on filenames and specifically extension. It is important that files with different semantics have different extensions; for example, templates must be separated by extension for the specific processor.
Shake is managing all filenames and calls functions in the next (sub-layer). It checks for existence of files and produces error messages when a file is not found — no further error processing for missing files neded.

In cases where files with the same extension (e.g. `html` or `pdf`) are given (in the dough directory) and are produced for some other files given as e.g. `md` files, the processing checks wether a file is given and if not, tries to produce it.

Transformation of filepath to Path

Transformation of filepath to Path

The FilePath typed files are translated to Path type, which differentiates relative and absoulute path to files or directories.

Processing

Processing layers are split again in two: a layer to read or write files (using typed files and typed content) before it is passed to the operations actually manipulating the data.

Issues : how to organize regression tests

In general, testing algebraic properties is difficult for complex data; I have a method to organize regression tests. Results from operations are stored and used for input later. The input and output of the test functions are typed to avoid problems with confusion in types between data written to disk and read from disk.

The construction of a test for a function is limited as another tested function must produce the input data.

Command Line Interface (CLI) for bake

The main program of SSG ssgbake has a command line interface (CLI) which includes switches to direct

The main program is `ssgbake` and it includes some swiches to taylor the run:

- continuous update (`watch`) when filecontent on disk changes update the current produced content to reflect changes applied to the files on disk (`w`)
- start a web server to serve the produced homepage (`s`)
- update the test homepage (`t`), which is included in the code and distributed with it,
- quick run without producing the `pdf` files, which slows down the conversion (`q`)
- help. The swiches include in specific version of SSG are shown (`h`).

30. *PackagesUsed.md*

> The Packages from Hackage used. Primarily pandoc, pandoc-citeproc, doctemplates, but also twich, shake, scotty and aeson, lens, and aeson-lens.

Pandoc

The central component of any modern site generator seems to be Pandoc. At the moment only markdown is used for content and output is html, additionally, `pdf` files for print output.

Pandoc-citeproc allows the inclusion of references and reformat references based on a BibTex file, which includes the details.

Templates

Pandoc includes a template system, [doctemplates] (http://hackage.haskell.org/package/doctemplates). Injects text values from a JSON record (based on labels); it allows conditionals (‘$if(label)$ .. $endif$) and loops.

Watching file change : Twitch

Twitch uses FSnotify to connect programmed actions to activities with files. It can be used to notify the process which bakes files about changes in file content.

Caching

Shake is a Haskell version of `make` and can be used to convert a static site (idee in Slick

JSON

The aeson Haskell implementation of JSON is used, together with aeson-lens for getting and setting values in JSON records.

40 use of pandoc . md

The transformation uses Pandoc, in four steps: - read the file into the pandoc structure - extract from the pandoc file all content into a context - convert the context into the target format - fill the context into a template to produce the result (respective a .tex file to process by lualatex)

Read Markdown

Read Markdown

Reads the YAML header and the text content into a `Pandoc` data type. The formating, in the header and the content, is converted from the input format (e.g. markdown) into the internal Pandoc encoding.

This first step could read essentially any format, Pandoc accepts - likely with minimal or no changes in other steps.

Extract all information into `Context`

Extract all information into `Context`

Extract the information in the `MetaValue` type into a `Context MetaValue`; preserves the formating in the Pandoc format, but separated into pieces.

Convert the Data to target format

Convert the Data to target format

The Pandoc structured formatted data are converted to the target format (either Latex encoded as Text or HTML encoded as Text) - each individual piece.

Fill the converted pieces into template.

Fill the converted pieces into template.

The specific templates for `Daino` must be compiled and are then filled with converted pieces - separately to produce the HTML file to be served and the `.tex` file to be processed by `lualatex`, which produces the final `.pdf`.

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