A Selective Rendering Concept for Static Site Generators

Sascha Zarhuber

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Interactive Media

in Hagenberg

im Juni 2017

Declaration

I hereby declare and confirm that this thesis is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

Hagenberg, June 26, 2017

Sascha Zarhuber

Contents

Abstract

Whereas modern content management systems provide a solid environment for dynamic content creating and editing, their dependency on external services like database services or authentication providers often complicate their abilities to scale. Additional duties for keeping them responsive in larger ecosystems are therefore often responsible for slowing down the overall workflow of a developer.

Static site generators on the other hand offer an easy solution for steadily growing websites. Their only task is to create a full-featured file structure, which contains browser-readable HTML files that do not require any on-the-fly rendering upon request. However, static site generators contain a significant drawback, as the rendering mechanism normally cannot distinguish between already present files and new content. In fact, the build time increases every time a new file is added to the source directory.

This Master's thesis therefore tries to compensate the full extent of a complete rebuild every single build cycle by providing a caching mechanism based on a selective approach, together with a remotely working REST API as wrapping interface for user-friendly interaction and improved division of work.

Kurzfassung

Während moderne Content Management Systeme ein gut entwickeltes Umfeld für Online-Inhaltsverwaltung und -erstellung bereitstellen, sind sie doch auf eine Anzahl externer Services angewiesen. Dazu zählen vordergründig Datenbanksysteme, aber auch verschiedene Login-Mechanismen, um Zugang zu einem gesperrten Editierbereich freizugeben. Durch die Abhängigkeit von derartigen Services entsteht bei zunehmender Größe des jeweiligen Projekts ein Anstieg des Aufwands für die Verwaltung dieser Erweiterungen.

Static Site Generatoren andererseits benötigen keine externen Erweiterungen, da ihre einzige Aufgabe darin besteht, die Website-Quellcodes in eine für Webbrowser lesbare Version zu konvertieren. Allerdings haben diese Static Site Generatoren einen erheblichen Nachteil; da sie nicht zwischen bereits vorhandenen und neuen Inhalten unterscheiden können, ist jedesmal ein vollständiger Neubau der Website-Quellen notwendig.

Diese Masterarbeit soll daher einen Lösungsweg für dieses Problem aufzeigen. Durch einen selektiven Algorithmus sollen am Ende nur die wirklich notwendigen Inhalte gebaut werden und in eine vorhandene Dateistruktur eingebunden werden. Zusammen mit einer REST API soll zusätzlich eine benutzerfreundliche Interaktion und verbesserte Arbeitsteilung möglich sein.

Chapter 1

Introduction

Back in the early 1990s, when the internet made its first steps towards a broader public use, a group of students at the University of Illinois created "Mosaic", the first publicly available Browser [3, p. 11]. At that time, websites consisted of just HTML and probably some images, whereas the release of "Netscape Navigator" led to the introduction of *Brendan Eich*'s JavaScript engine. Additionally, Netscape also introduced a web server software called "Netscape Enterprise Server", thus making the Internet available for the first web developers [3, p. 12].

Since then, a lot has changed; content management systems were published, the internet was turning to what was called "Web 2.0" and the common user was not just a content recipient anymore, but also a content creator without requiring deeper understanding of web technologies [3, p. 19]. This has affected not only private users, but also whole enterprise structures until today.

However, the most important part stayed the same; steadily providing content which is deliverable on request. To do so using content management systems requires not only a web server and the client's browser, but also a properly set up chain of interacting services for assembling HTML code on the fly. While this kind of architecture may surely be fitting smaller blogs very well, the necessary effort of managing constantly growing enterprise sites is likely to grow exponentially.

Therefore, systems which are not dependent on such a chain are constantly on the rise over the last yars. They especially make sense in environments, where content is constantly added, but hardly ever deleted or changed. Lastly, by constructing static websites in plain HTML and mostly avoiding any dynamic features, a trend reversal back to the internet origins is clearly noticeable in some fields of modern web development.

1.1 Problem statement

Static site generators are growing fast and are more and more used as a replacement for common content management systems. The main advantage is their independence of external services, like database systems, session caching services, etc. Also, they seldomly consist of complicated backend systems and are mostly created in pure HTML or simple markup languages like Markdown (see Sec. 3.1.2).

One of the biggest drawbacks however is the fact, that static site generators have to preprocess every bit of information they contain. This is the complete opposite compared to other content management systems, which process information on request. This means, that user-readable content is fetched and rendered "just in time" it was requested from the client.

Therefore, depending on the setup, a dynamically growing amount of time needed for a build cycle might be the case. For being able to work against this fact, a working approach has to be found, which saves time by leaving out information, which has not changed since the previous build.

1.2 Goals

To find a suitable solution, a service which contains a build pipeline including a caching mechanism has to be implemented. The caching mechanism should thereby act as the core part, as it is responsible for fetching data between the current development state and a previous build cycle. Furthermore it should determine the build extent by selecting the respective files for rendering based on the fetched commit diff. The research question is therefore the following:

How to speed up static site generation by a selective approach?

The implemented solution covers the necessary steps for working with cacheable content in a way, that a remote-only building process is possible. Together with the precondition of having a GitHub account, any repository consisting of a Metalsmith project should be able to get rendered on this service's REST API.

By introducing this service early into a project workflow, the user should notice a significant improvement concerning the rendering time per build cycle. Furthermore, it should take a considerable amount of workload out the developers hands.

1.3 Structure

To express the considerations which led to the finished solution, the following pages are structured into several chapters.

Chapter 2 – Shows the current state of the art, culminating in the presentation of three selected static site generators: *Jekyll*, *Hexo*, *Metalsmith*, as well as a comparison between them.

Chapter 3 – Explains the most important terms concerning the operation of static site generators (including code samples); *build pipelines*, *frontmatter*, *markdown*, *templates* and *diff*.

Chapter 4 – Gives an understanding of the initial theoretical approach behind this project. Challenges and solution strategies are examined prior to general considerations towards the implementation are being unveiled.

Chapter 5 – Describes the full extent of the implementation including the whole REST framework, which was built around the build pipeline. Different graphics are showing examples of how various parts were realized.

Chapter 6 – Evaluates the project using different testing approaches. The REST API was put under high load testing, whereas the build pipeline and the caching mechanism were compared using the timespan needed for an operation. Furthermore, an outlook shows possible future improvements.

Chapter 7 – Shows the conclusion of this project work and unveils difficulties, as well as enhancement strategies for productive use.

Chapter 2

State of the Art

The roots of the most well-known modern Content Management Systems (*CMS*) date back to the early 2000s, when PHP was (and *still is*) the dominant factor in terms of server-side programming languages (see Fig. 2.1).

While the *typical* CMS was starting out as mostly just a "dynamic online tool", it also shows that with seamlessly integrating new features gained through the development of its underlying programming languages, as well as steadily adding new functionalities (mostly requested by the community), a transition towards a fully-manageable and customizeable, semi-automatic web application was possible [3, p. 17].

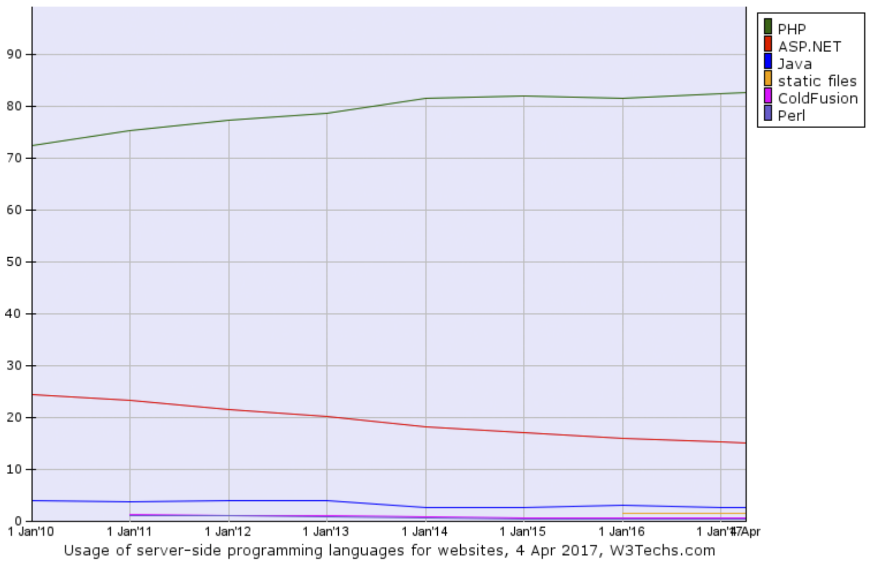


Figure 2.1: A graphic showing the global share of *server-side programming languages* from January 2010 to April 2017. *PHP* remains the dominant language with a share growing from 72.5% to 82.6% [43].

The advantages are clear: a web designer or content editor does not automatically have to be a web developer, who needs profound knowledge about software or server architecture. Instead, most of the time it is enough to know how to operate an FTP-client and to know the credentials of the built-in database service, submitted by the web hosting provider upon registration. As a summary, it can be said: *Less people for less responsibilities* – the web application does the rest.

However, this progression also caused a few drawbacks, especially when it comes down to comparing the amount of workload needed before actually being able to create content for the World Wide Web. Due to the fact, that most CMSs evolved to their own sort of powerful admin panels, developers who are not keen about keeping their site updated, risk its defacing or other embarrassing attacks through unmanaged security holes in the code [3, p. 23]. So, how is it possible to bridge the gap between a fairly secure system and creating content whenever it is desired?

In 2008, *Tom Preston-Werner* created *Jekyll* out of his frustration of having the need of "styling a zillion template pages" and "moderating comments all day long" before even being ready to create content on his blogging engine [39]. Furthermore he mentioned, that one of the other main reasons were the lack of possibility for publishing his posts on his own server, when subscribing to a fully-managed online hosting service like wordpress.com[1]. Services like this offer only limited customization options, where there likely is no access to online storage and database without using the built-in admin panel. Even then, access might be very restricted.

The intended core functionality of Jekyll narrows down its mode of operation to handle three main components found in most static site generators today [3, p. 24]:

Core language – The language a static generator is written in, for example JavaScript or Ruby.

Templates – The templating language to be used through the blog and posts.

Plug-ins – All static site generators allow for additional functionality through some sort of a plug-in system.

In contrast to common dynamic CMSs, a static site generator outputs plain static HTML. It does it in a way, that a certain *distribution* folder holds the complete web root, without the need of binding it to external services like databases or session management tools. Occasionally, different plug-ins also allow the generation of client-side *JavaScript* or *Cascading Style Sheet* files through their respective pre-processing tools.

[1] https://en.wordpress.com – a hosted version of Wordpress.

2.1 Jekyll

As already explained, Jekyll was created out of the need for avoiding to service the blogging engine before writing and publishing content. Since it is deeply integrated into *GitHub*, it is considered as the probably most-popular static site generator.

2.1.1 History

Tom Preston-Werner, co-founder of GitHub[2], announced it in October 2008 in one of his blog posts [39]. Already in December 2008, it was introduced as build engine for the then newly featured GitHub Pages service, allowing owners of repositories to publish a static website by just pushing to a certain *master* or *gh-pages* branch [40], which is still available for free to this day.

All of this happened just 6 (respectively 8) months after GitHub was launched [41] and is now even being used by technology-leading companies to showcase their open-source efforts[3].

2.1.2 Technology

Jekyll was entirely written in *Ruby*, as Tom Preston-Werner rather saw himself as a software developer in the first place, than as a content author [39]. Until now, the repository for Jekyll still consists mainly of Ruby code at a share of roughly 77.5%.

Advantages

One of the main advantages is the modular structure of its code base. By inheriting different Ruby classes, it is quite easy to extend and add features to fit the developer's needs. Due to its wide-spread usage initiated through the GitHub universe, Jekyll also has an accordingly huge user base and is therefore well documented [3, p. 26].

Furthermore, its website[4], which mainly acts as starting basis for documentation, is not only available as open-sourced git repository, it is also built using `Jekyll` to prove its universality.

Starting from scratch, the command `jekyll new my_project` installs a blog environment for starters in the `./my_project` folder. The basic install consists of an elementar blog post structure, *Sass* source files, and a few template files written for Shopify's *Liquid*[5] engine.
Using this starting environment, the unexperienced developer quickly gets

[2]https://github.com – GitHub Inc.
[3]https://github.com/showcases/github-pages-examples – GitHub Pages examples.
[4]http://jekyllrb.com – Jekyll website.
[5]https://help.shopify.com/themes/liquid – Shopify's Liquid template engine.

a sufficient overview of what is generally possible using Jekyll, whereas the content author is able to fully concentrate himself on writing content, as the used *Markdown* markup language requires little to no prior syntax knowledge. Furthermore, Jekyll already ships with a built-in webserver for quickly reviewing the rendered static output.

Disadvantages

As powerful as Ruby might have been designed, many unskilled developers are facing difficulties right from the beginning, as most of them experience a steep learning curve. Nearly every single bit of customizing Jekyll requires Ruby knowledge, especially if it is desired to move along the "predefined" way and not including third-party extensions like *Node.js* tools or else.

Additionally, its template language, Liquid, offers customization on a very high level, so it might happen to confuse business logic[6] with template logic[7]. To make things worse, different template constructions might also evolve over time and therefore causing a parallel coding universe when trying to surpass difficulties in the business logic.

2.2 Hexo

Hexo understands itself as counterpart to Jekyll, mostly by covering the same ideas of static site generation, but building up completely on Node.js. It even offers a migration service for Octopress- and Jekyll-users who are willing to switch.

2.2.1 History

Version 1.0.0 was originally released in March 2013[8], although development on GitHub dates back to September 2012 as the first commit was published using the message "init".

Tommy Chen, its creator, first used *Octopress*[9] but quickly became dissatisfied with its performance, as the rendering of 54 blog posts already took more than a minute of compile time [14]. Since he assumed Ruby might be the cause for the lack of performance of his primarily used blogging framework, and further development on this case was not likely to happen any time soon, he decided to look for something which got his attention shortly before: Node.js.

[6]How Jekyll processes data into programmatically readable structures.

[7]How Liquid transforms these structures into browser-readable HTML.

[8]https://github.com/hexojs/hexo/releases/tag/1.0.0 – Hexo v1.0.0 release page on GitHub.

[9]http://octopress.org – Octopress website.

However, Node.js was not really a big player back at that time, so the offer of blogging frameworks written in JavaScript was very dense and not really fitting the needs of Tommy Chen. In his announcement article for Hexo [14], he references a blog post of *Boris Mann*, also an `Octopress` user at that time, listing a few Node.js-based blogging frameworks, which were already around in June 2012 [35]. Interestingly, only two of all the mentioned ones, *Wintersmith* and *DocPad* are still actively maintained today.

2.2.2 Technology

As already stated above, Hexo primarily consists of JavaScript, thus making it easier to start for developers with a frontend web development background. In fact, its GitHub repository shows JavaScript holding a share of 100% on the source code[10].

Advantages

Right from the start, Hexo presents itself using its feature-rich command-line interface (*CLI*), similar to Jekyll. Once it is installed, `hexo init my_project` scaffolds a new starter template into the `./my_project` folder.
As Tommy Chen himself wanted an easy-to-use replacement for Octopress, Jekyll and Hexo share a lot of common features; their content files use both *YAML* frontmatter and *Markdown* by default, even the main configuration file uses a very similar structure in both frameworks. This should make it extremely easy switching from Jekyll to Hexo.

First and foremost, programming-unaware content authors might especially like its CLI, as it also offers to create files based on the `hexo new` command. Depending on other submitted command-line arguments, Hexo may automatically put the new file in the according sub-folder, whether it is a *draft, page* or *post.* Publishing a draft is as easy as `hexo publish`. When creating content, Hexo also contains a feature-rich, Octopress-inspired custom tag selection for including content from *YouTube, Vimeo* or *GitHub Gists.*

Additionally, its plugin collection is also constantly growing and mostly community supported. A special naming convention using `hexo-` as prefix helps by determining which plugins to auto-load out of the `node_modules` folder. Using this way, Ruby's *convention over configuration* mantra is ported to JavaScript as well and especially supports beginners by not having to define the usage of a certain plugin.

[10] https://github.com/hexojs/hexo – Hexo repository on GitHub.

Disadvantages

Hexo might look like as an ideal replacement for Jekyll, but since both share so much similarities, they also share some disadvantages. Whereas Jekyll ships with Liquid and Sass as standard, Hexo does with *EJS* and *Stylus*. Although clearly stated, that both of these plugins might be easily uninstalled later on [15], the whole setup pre-installation seems as opinionated as Jekyll's.

In addition to the already mentioned plugin system, a missing configuration option might as well turn out to be misleading in terms of customization options, especially when being dependent on the CLI. If customization is necessary, the developer often is forced to switch to the JavaScript API[11] or to add a plugin to the project to make the build pipeline fit the customization's needs.

When it comes to caching, Hexo uses a homebrew version of *JSON memory caching* called *Warehouse*, also created by Tommy Chen[12], initially mentioned in the release notes for version *3.2.0-beta.2* [13]. Using this plugin, a mode called "hot processing" should enable faster rebuilds. The main drawback here might be the caching speed, which is on the one hand filling up the memory when working on bigger projects, whereas the persisting of the database is fully dependent on file input/output write speeds of the underlying hard disk. Furthermore, a constantly growing database file is hardly transferrable when trying to implement a decentralized building system out of Hexo.

2.3 Metalsmith

Compared to the already described static site generators, *Metalsmith* is to be considered the youngest project. It might also be the most radical project, as it was designed to consist of *nothing but plugins* [3, p. 31]. Therefore, in terms of still being a static site generator, it tries hard to push the limits much further than previously mentioned Jekyll and Hexo.

2.3.1 History

Initially developed by *Segment*[13] for their internal needs, such as *documentation, help* and *blog pages* [46], Metalsmith was finally open-sourced and made publicly available around February 2015 – its commit history on GitHub dates back to February 4th, 2014. Most of the commits at that time were published by *Ian Storm Taylor*, co-founder of Segment, although his contri-

[11] https://hexo.io/api/ – Hexo's JavaScript API documentation.
[12] https://github.com/tommy351/warehouse – Warehouse repository on GitHub.
[13] https://segment.com – Segment's website.

bution to the project ends after releasing *v2.1.0* on September 24th, 2015[14] at the moment.

Like Hexo, Metalsmith's repository completely consists of JavaScript code, as its developers also were unsatisfied with the then existing static site generators. According to Chris Sperandio, the Metalsmith developers desired pure flexibility for their "wide array of use cases", while other frameworks all asked for a certain structure on the content [46].

2.3.2 Technology

Since Metalsmith consists of only plugins, specifically written for this very framework, there is no real standard setup provided. Although there are a few tutorials and best practices listed in its GitHub repository [45], as well as in a repository called "*awesome-metalsmith*"[15], the initial dive-in might scare a few people away, since Metalsmith might not be as well documented as the previously mentioned frameworks. Moreover, most developers seem to experience a very steep learning curve at first, given the amount of customization options and the requirements for understanding the blog engine infrastructure [3, p. 31].

Advantages

Every developer is able to shape Metalsmith exactly to his/her needs, once he knows about the basic usage. It ships with a CLI, as well as a JavaScript API, where the "real hacking" is possible. The CLI gets easily configured via a `metalsmith.json` file, stored in the project directory. It consists mainly of general project configurations, placed in the object's root, as well as an array of used plugins, respectively combined with their configuration.

It neither contains a pre-installed template engine, nor any other preprocessing tools, like Sass or Less. However, the available plugins support most of them to a satisfying extent. As an example, the metalsmith-layouts plugin is a wrapper for *Consolidate.js*, which per se acts as a wrapper for the most common template engines[16]. Therefore, the developer is able to select the tools based on his/her preferences and may initialize a project from scratch, without needing to clean up any pre-installed demonstration files first.

Using the built-in JavaScript API, it is also possible to invoke the needed modules programmatically, which is one of the core topics of this Thesis.

[14] https://github.com/segmentio/metalsmith/commits/master?author=ianstormtaylor – Contributions of Ian Storm Taylor to the Metalsmith repository on GitHub.

[15] https://github.com/metalsmith/awesome-metalsmith – "Awesome" Metalsmith resources list.

[16] https://github.com/tj/consolidate.js#supported-template-engines – Consolidate.js-supported template engines on GitHub.

Program 2.1: config.js

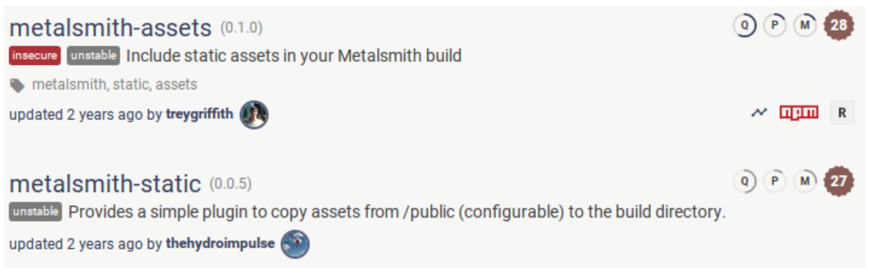
```
 1 const Metalsmith = require('metalsmith');
 2 const layouts = require('metalsmith-layouts');
 3 const markdown = require('metalsmith-markdown');
 4
 5 Metalsmith(__dirname)          // Invoke with "___dirname" as CWD
 6 .metadata({                    // define globally available variables
 7   name: 'Test site',
 8   url: 'http://localhost:3000'
 9 })
10 .source('_src')                // the content directory
11 .destination('_site')          // the output directory for compiled content
12 .clean(true)                   // delete output directory first? yes!
13 // (...)                     // Additional plugin configuration
14 .use(layouts({
15   engine: 'handlebars',
16   pattern: '*.html'            // Plugin would look for .html files, although
17 }))                            // not yet processed by markdown renderer
18 .use(markdown())               // <— Here markdown gets processed to HTML
19 .build((err) => {
20   if (err) {
21     throw err;
22   }
23   console.log('Success!');
24 });
```

Disadvantages

Such an amount of freedom in designing a project may also cause some dangers. In this case, one of the most crucial things is the arrangement of plugins in the configuration. Since Metalsmith acts as a streaming build system, every transformation of the content must happen at its time to not interfere with any upcoming plugins. This is especially important when a plugin might alter the underlying code in a way, that a following plugin becomes useless, as it might not be able to succeed in its predefined task. *Andy Jiang* gives a good example about a sample structuring of Metalsmith plugins on the Segment blog [34].

As an example, Program 2.1 shows one bold example of misconfiguration (see line 16):

Moreover, the available plugins may seem as not as popular as the plugins from Hexo, given the average amount of stars received on GitHub. This might be due to the often missing maintainance, or simply because of the fact, that there seem to be multiple plugins for one single task (see Fig. 2.2).


metalsmith-assets (0.1.0)
insecure unstable Include static assets in your Metalsmith build
metalsmith, static, assets
updated 2 years ago by treygriffith

metalsmith-static (0.0.5)
unstable Provides a simple plugin to copy assets from /public (configurable) to the build directory.
updated 2 years ago by thehydroimpulse

Figure 2.2: A screenshot showing some of the results for the search query "*metalsmith static assets*" on https://npms.io. Both of the shown entries describe an identical mode of operation within the Metalsmith build pipeline. Also notice the very unstable *semver* versions: *0.1.0* and *0.0.5*.

Table 2.1: A comparison of static site generators

	Jekyll	*Hexo*	*Metalsmith*
Language	Ruby	JavaScript	JavaScript
Foundation	Oct. 2008	Sept. 2012	Feb. 2014
Contributors	~700	~100	~50
Plugins	~800	~600	~590
Customizability	Mediocre	Low	High
Opinionatedness	High	High	Low
Standard templates	Liquid	Swig	*none*

2.4 Comparison

To conclude the summaries of different static site generators, a short overview using a table is given below (see Table 2.1). The comparison is built up on advantages and disadvantages, as well as the most important features they consist of. Therefore overall popularity and customizability play an as important role, as the language they are written in. However, Metalsmith is standing a little bit out, as it does not provide a full-featured framework – it merely serves as the basis for further plugin setup, whereas Jekyll and Hexo already contain some sort of standard setup. The numbers of plugins in Table 2.1 were taken from search queries on https://npms.io and https://rubygems.org.

Chapter 3

Technical Foundations

Before any explanation of the theoretical approach behind this research, there is a need of describing the technical foundations, on which the general thoughts were built on.

Being mostly a back-end web developer, one surely gets confronted with a lot of changes in this field. Changes which were mostly caused by the ongoing progression of general web development, but also caused by constantly coming and going "hypes". Changes which might leave minor traces, but sometimes also having a major impact on the way some developer processes things during his/her work on different projects.

One of these major turning points was the introduction of EcmaScript 6 (*ES6*), which provides a more sophisticated application flow for software developers compared to the old standards, where the code architecture quickly got out of hand, especially if not fastidiuosly checked using *JSLint* [16]. ES6 introduced a lot of new features, pushing JavaScript more and more towards the definition of an "object-oriented" scripting language, thus not only by providing "real classes", instead of the old and cumbersome approach of inheritence by setting a constructor function into the *prototype* object [2, p. 47].

However, the probably most beneficial functions are *Promises* [37] and *Arrow functions* [36], both saving significant amounts of code – especially when working with asynchronous environments, like HTTP requests.

A dynamic web project often demands lots of preparation before receiving any visible outcome, sometimes even when using a predefined framework. Additionally, though many projects in the Node.js universe are already matured to an extent, where they may be even used for enterprise projects, there is still a remaining risk for failing a client due to the amount of dependencies on external services like databases, session storages or user management tools.

Compared to a dynamic CMS, a static site generator takes out the complexity of a web project, as it only produces the very basic parts for serving

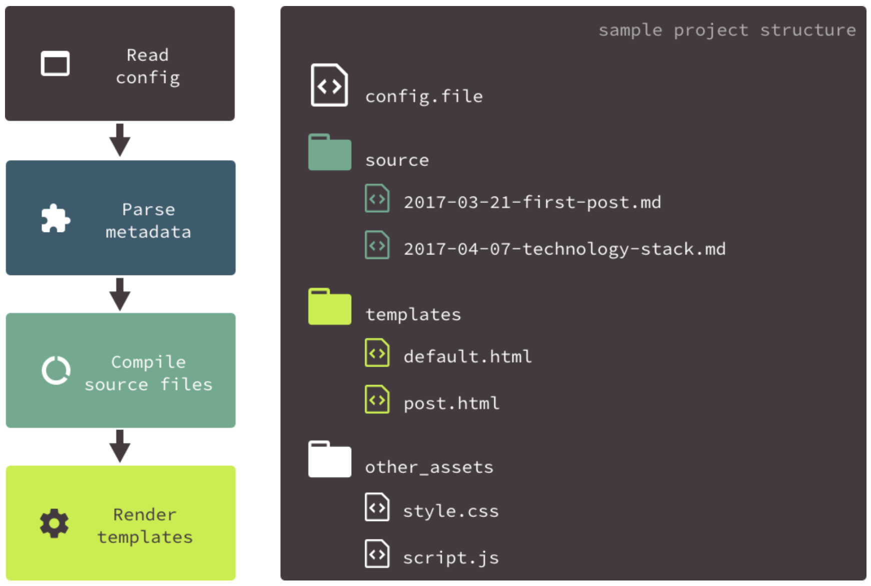


Figure 3.1: A graphic showing the basic flow of a *build pipeline*. First, the configuration file is read and necessary modules invoked. Second, the global metadata, as well as metadata from within the content, is parsed and stored in the global configuration. Third, the content sources get compiled into basic *HTML* markup. Lastly, the compiled content gets rendered into predefined templates, to apply a given structure which is used commonly throughout the website.

information to the client. Furthermore, most of these projects are quickly scaffolded and provide a huge amount of variety in different tools they use. When using an environment for automatic deployment, an updated version may even be uploaded to the web server without a developer's intervening.

3.1 Build pipelines

A static site generator mostly consists of a build pipeline, which handles the workflow needed for bringing the content into shape. This goes from setting the boundaries, determined by a configuration file, to finally producing a web root, consisting of HTML, CSS and JavaScript files, as well as images.

Normally, the major part of it happens sequentially, as nearly all content files are facing a series of transformations on them [34]. Although the amount and extent of conversions may differ significantly from pipeline setup to pipeline setup, it can be broken down to the following core parts (see Fig. 3.1):

Metadata parser – Parses global metadata, found in the configuration file or in the YAML frontmatter of content files.

Program 3.1: frontmatter.md

```
---
author: Sascha Zarhuber
title: Frontmatter demo
date: 2017-04-07
tags:                     # Array of tags for use later on
  - some
  - tags
  - here
template: false           # Order plugin to not render this file
---

Here starts the normal content
```

Markdown compiler – Used to convert easily read- and writeable Markdown files in browser-readable HTML.

Template renderer – Responsible for bringing the very basic content structure in shape. The goal should be a common appearence, enriched with additional elements (like navigation, breadcrumbs, etc.).

Of course, the list above overlaps at some point with the list mentioned by Vikram Dhillon [3, p. 24], as a build pipeline may be considered only as a part of the given static site generator (although the main part), not as the generator itself. One of the reasons is its independence of programming languages: A build pipeline doesn't care which programming language it consists of, as long as it knows how to interpret the content sources and templates. Therefore, it merely should be called a concept, not a framework.

3.1.1 Frontmatter

Program 3.1 shows a sample usage of frontmatter inside a Markdown file. Bounded by three dashes above the main content source, it allows certain per-file metadata definitions, which will be parsed at build time and provided for the template rendering engine.

As an example, the selected template for this sample file may also hold a list for the mentioned *tags*, as well as a placeholder for the *author*'s name and/or *title*. The main content gets then rendered into the respective placeholding tag, already self-containing a basic structure.

Using a *template: false* declaration, some plugins may prevent rendering the content into a template. This might be interesting in cases, where different partials should be included in the DOM by some sort of asynchronous JavaScript later on.

Program 3.2: markdown.md

```
# This text gets wrapped in a h1 tag
## This text gets wrapped in a h2 tag

*italic text*
**bold text**

[This is a hyperlink text](http://www.fh-ooe.at)
![This is an image caption](http://localhost/assets/image.png)
```

3.1.2 Markdown

Markdown consists of shorthand conventions, which should be easier to type for content creators [3, p. 38]. Therefore, it makes understanding HTML not a necessary precondition anymore, as a basic content structure may be easily achieved when prepending/surrounding text with certain special characters like `#`, `*`, `_`, etc (see Program 3.2).

Originally created by *John Gruber* as a plugin for *Movable Type* and *Blosxom* blogging engines in March 2004 [22][23], it should be a supportive tool for users against the complexity of formal markup languages (e.g., HTML5) [8, p. 4]. According to Gruber's intention, there is no "invalid" `Markdown`, as he suggests the author should either "keep on experimenting" or "change the processor", if the output happens to fail his/her expectations [8, p. 5].

GitHub finally adapted Markdown to its own version, called "GitHub Flavored Markdown" (*GFM*), somewhere around April 2009[1]. The people behind it enhanced its original functions to also support *code blocks, tables, strike-through text*, as well as *auto-linking* URL structures within the content [8, p. 18]. Additionally, also GitHub-specific functions, such as *user mentions, commit references* or *emojis*, may be used [29].

Since then, GitHub renders browser-friendly versions of general descriptions written in Markdown, for providing fast and easy overviews of the respective repository. As an example, an existing `README.md` always appears below the root file tree section on a repository front page [4, p. 5].

3.1.3 Templates

Templates are the frames of each content page, caring for a common, browser-readable HTML structure. A uniform design layout allows site-wide look and feel using a global CSS style sheet, as well as certain events triggered by user behaviour, handled by a single JavaScript file.

[1] https://github.com/mojombo/github-flavored-markdown/issues/1 – GitHub Flavored Markdown examples by Tom Preston-Werner.

Program 3.3: post.hbs

```
<!-- include header partial -->
{{> header.hbs}}

<!-- set title of post in h1 -->
<h1>{{title}}</h1>
<!-- insert post author & date metadata -->
<p>by {{author}} on {{date}}</p>

<div>
  <!-- insert the main content here -->
  {{{contents}}}
</div>

<!-- include footer partial -->
{{> footer.hbs}}
```

Born out of the need to fail-safe produce HTML on the server, as the produced data chunks – initiated by a client HTTP request – steadily grew, the goal behind templating engines is primarily to separate business logic from data display. Ideally, at the end of the day, there should be no code in HTML, and no HTML in code [12, p. 225].

A simple template file example for a blog post is shown in Program 3.3. It is written in *Handlebars*, a very basic templating language, offering only a very limited amount of template logic. Included are *loops, if/else, partials, ...* – however, additional "*helper*"-functions may be added by the developer.

Although such template logic may come in handy for the most part, as some business logic decisions seem to be rather taken during rendering, instead of being hard-coded before, the concept of data separation is therefore often unknowingly violated [12, p. 228]. Thus, the choice of the "optimal" templating engine for a given project is crucial, as different engines offer a different range of built-in logic. This could go from very conservative *Mustache*[2] to very powerful ones like *Liquid*[3] (see Sec. 2.1.2) or *EJS*[4].

3.2 Git

Today, hardly any software project is started without any form of version control system (*VCS*). It supports developers as a back-up system and living archive of their work, as data generally is ephemeral and can be lost easily

[2] https://mustache.github.io/ – Mustache website.
[3] https://help.shopify.com/themes/liquid – Shopify's Liquid template engine.
[4] http://www.embeddedjs.com/ – Embedded JavaScript website

[9, p. 1]. Although there are many different variations offered, some of the most popular ones today are *Git, Apache Subversion, Mercurial* and *Bazaar.*

3.2.1 History

Git was initially published by *Linus Torvalds* on April 7th, 2005 [9, p. 6]. This was necessary, as the “free” version of the then used VCS for the *Linux* kernel development, *BitKeeper*, was restricted in a way it was not suitable any longer for the community behind it. Furthermore, the search of an already available alternative to the BitKeeper system failed due to an unsatisfying combination of needed features, so *Torvalds* came up with his own VCS flavor, containing all desired functionalities for further Linux development (among others) [9, p. 4]:

- distributed development,
- handle thousands of developers,
- efficient performance,
- support branched development, as well as
- free, as in freedom.

While it was merely a tool for kernel hackers in the beginning, its simplicity quickly made developers use it for other projects too. However, the CLI still scared off people with less programming background, until GitHub came and introduced its Desktop client[5]. Today, it is one of many GUIs, which are available for different operating systems[6] and completely omits the necessity of terminal emulator knowledge when working with Git.

3.2.2 Technology

Having decided to use Git as VCS, everything begins with setting up a new respository. This can be made remotely on a service like GitHub, or locally using `git init`. When initialized as a local repository, it can still be published remotely through setting the correct URL using `git remote` later on [9, p. 198].

Commits

The next step would be to work with the repository. For the most part, this should not influence the developer’s workflow in any way, as the `.git` directory should be automatically hidden in most modern editors to not distract him/herself from the actual work. If the developer succeeded in a sub-task or simply wanted to save the current project state, he/she would create a *commit.*

[5]https://desktop.github.com/ – GitHub Desktop client.

[6]https://git-scm.com/downloads/guis – Git GUI clients on the Git website.

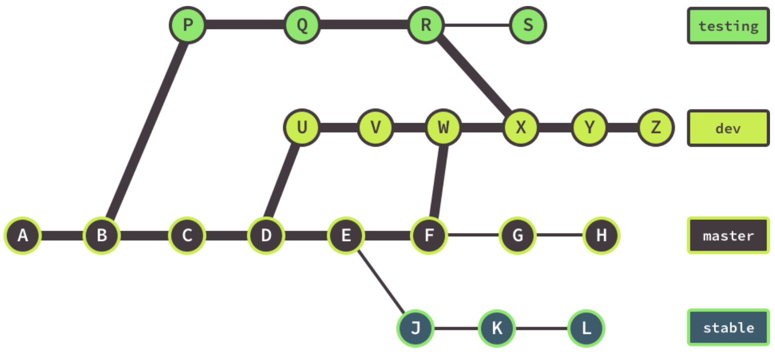


Figure 3.2: A graphic showing the basic structure of branches in Git. The *testing* branch was created out of project state “B”, whereas *dev* was branched away from state “D” and currently holds the *head commit* at “Z”. In the meantime, *testing* was merged into *dev* at state “R”. In the end, *dev* contains the commit history of all states connected with the bold line [9, p. 92f].

A commit is a snapshot of the current repository’s state. While it does not contain a copy of every file and directory in the project tree, it rather compares the current condition with its predecessing snapshot and creates a list of affected files and their changes. Blobs[7] are either reused, if they remain unchanged or created new, if they were altered [9, p. 65].

Every commit is organized in a way, that it is chained to its predecessor (*parent*), thus representing a singly linked list with the ability of gaplessly going back from the current state (*head*) to the initial commit [3, p. 204][9, p. 65]. This is necessary, as it may happen to fix a bug or improve a design decision only by reworking a certain snapshot in the past [9, p. 151].

Branches

Since its early days, Git was used not only as back-up or archive system, but also as code management system. This is possible due to built-in functions, such as the so-called *branched development*, where a current state of the actual development is duplicated and worked on separately. It allows for development to continue in multiple directions simultaneously [9, p. 89] (see Fig. 3.2).

When being part of a remote team, a developer may also *push* his/her own local branches for providing it to others, as well as keeping them for local development and *merging* them back to the main branch after succeeding in his current task [3, p. 207]. Through the use of these possibilities, a responsible administrator keeping track of the development structure is not necessarily required to be appointed in any repository settings. Furthermore,

[7] *BLOB* – Binary Large OBject, a file which does not consist of queryable source code.

the team may decide members in charge for merges in an agile way, based on the current need.

Usage in static site development

The advantages of using Git in static site development are mainly its possibilities of providing a full-featured archive of the content and source code of each project. Seamlessly going back and forth in the website development history makes it easy to navigate between every single content edit, without loosing track of previous and future revisions. In this case, it may work significantly better than using a common database system for content storage.

Furthermore, it is also possible to make use of Git *hooks*. Once a new commit is created, a hook might take care of invoking the build pipeline as a "post-receive" hook – thus, the new website version gets built automatically without requiring any additional user interaction. However, hooks should be only used with caution, as they are not distributed the same way as the files tracked in a given repository and also may harm the integrity of the respective Git repository [9, p. 285f].

3.3 GitHub

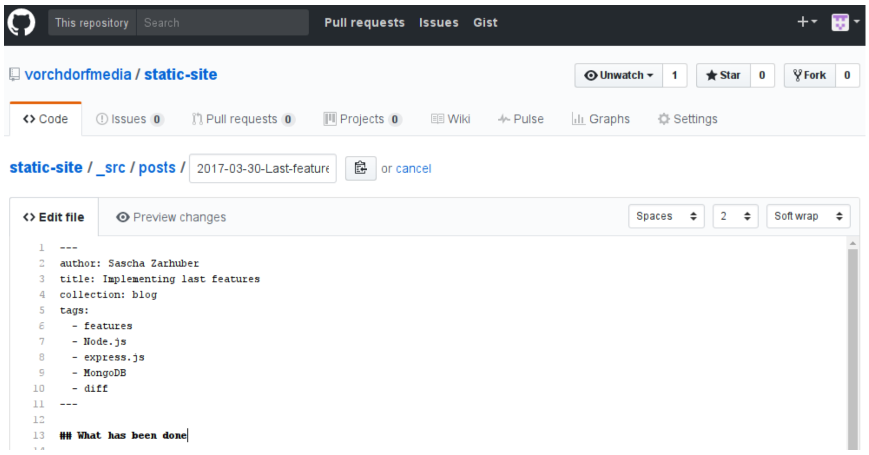
As already mentioned a few times before (see p. 6 or 16), *GitHub* is currently probably the most popular online collaboration platform, hosting not only the source code for the Linux kernel[8], but also for other huge projects like Google's *TensorFlow*[9], Microsoft's *.NET*[10] or Facebook's *React*[11].

3.3.1 History

Tom Preston-Werner, a Ruby programmer from San Francisco and creator of earlier mentioned Jekyll (see Sec. 2.1) and *Chris Wanstrath* started developing GitHub in October 2007. After releasing a private beta in January, they released the site to the public on April 10th, 2008 [41].

Since then, GitHub grew very fast and quickly gained on popularity throughout the developer landscape, hosting more than 56 million projects today[12]. Thanks to their generous freemium pricing model, collaborating in open source projects still is for free: A free tier account may hold unlimited open source repositories, working together with unlimited contributors[13].

[8] https://github.com/torvalds/linux – Linux kernel repository on GitHub.
[9] https://github.com/tensorflow/tensorflow – Tensorflow repository on GitHub.
[10] https://github.com/Microsoft/dotnet – .NET repository on GitHub.
[11] https://github.com/facebook/react – React repository on GitHub.
[12] https://github.com/about – GitHub's "about" page.
[13] https://github.com/pricing – GitHub's pricing page.



Spaces: 2, Soft wrap

```
---
author: Sascha Zarhuber
title: Implementing last features
collection: blog
tags:
  - features
  - Node.js
  - express.js
  - MongoDB
  - diff
---

## What has been done
```

Figure 3.3: A Screenshot showing the *In-Page Code Editor* of GitHub. An existing file is selected and ready to be edited. When finished, the user may commit the changes into the repository, so that other contributors also benefit from his/her adjustments.

All in all it seems, that GitHub turned coding into a truly social activity [9, p. 416].

3.3.2 Technology

The main use case for creating a repository on GitHub is the fact, that unlike other privately hosted remote repositories, it offers a wide range of additional services. Services like *Issue tracking, Pull requests* and *Code reviews* leverage the maintainability of source code in a repository, making it easy for each developer to discuss and manage the current project state without the need of switching to a third-party application.

Especially for content authors without programming knowledge, the *In-Page Code Editor* might be a very supportive tool, as it provides a clean and easy-to-use frontend for directly adding content to the repository (see Fig. 3.3). Additionally, the just edited file may not only be committed into the currently selected branch, but also in a newly created branch. Therefore, the source branch stays clean, whereas the edited file may get reviewed by an assigned supervisor, before being ready to get merged. Furthermore, also developers might make use of this feature, especially when a small hotfix is to be made, where it would be too time-consuming to *pull, commit* and *push* from/to the repository [9, p. 405].

Pushing to a *gh-pages* branch, or creating a "<username>.github.io" repository, enables the use of GitHub's built-in website hosting service. From there, either a Jekyll project is freshly built, or already compiled static HTML are automatically published to the web – additionally, custom domains may be used when adding a `CNAME` file [3, p. 171f].

3.3.3 REST API

Another significant advantage is the access of GitHub's REST API. Currently existing in its third major release, it almost offers every feature also available graphically in its web interface, as an equivalent JavaScript Object Notation (*JSON*) upon programmatical request. Some services even feature more advanced data through the API than through the UI [9, p. 410].

When having the need of including data from a GitHub account into a third-party service, a single authenticated HTTP request does the trick. Due to many available endpoints, a developer may quickly find the type information he/she needs to further process data directly from a repository. As an example, a complete listing of a repository's file tree is also possible, without needing to download and unpack an archive file. For one thing, this saves quite some time, for another thing, the requested data is already processed and presented, so that not only file paths are unveiled, but also their direct links and data types.

Furthermore, the API not only offers access to informational data about a given repository, instead, its manipulation through creating commits or uploading a file may also happen. To sum everything up, very well documented examples are available on the API page on GitHub, where developers catch a good glimpse, of what is possible overall[14] [9, p. 401].

3.4 Diff

"*diff* reports file differences between two files, expressed as a minimal list of line changes (...)" [6, p. 1]. Existing more than 40 years now, it has been an essential tool for file comparison throughout the history of computing – furthermore, it is also a core component of Git, which contains its own version called *git diff* [9, p. 108].

3.4.1 History

Initially published by *James W. Hunt* and *Malcolm D. McIlroy* in July 1976 when working at *Bell Labs*, the algorithm was later used in *UNIX* as application called diff. *Paul Eggert* and *Richard Stallman* (among others) also wrote the diff application as part of their *GNU diffutils*[15], which is nowadays mainly distributed in Linux derivatives, MacOS, as well as part of Git. They used an improved algorithm published by *Webb Miller* and *Eugene W. Myers* in 1985 [10, p. 3], who proved that the original "Hunt-McIlroy

[14] https://developer.github.com/v3/ – API v3 documentation on GitHub.

[15] http://manpages.ubuntu.com/manpages/zesty/en/man1/diff.1.html – Manpage for GNU diff.

Program 3.4: sample.diff

```
0a1
> w
3,4c4,6
< c
< d
---
> x
> y
> z
6,7d7
< f
< g

```

algorithm" is inefficient on certain special cases. As a test case, they used a file containing 1000 blank lines, and a second file, consisting of the initial file, but with a single non-blank line added on both ends. As a fact, using other experiments performed on typical files, Miller's and Myers' algorithm ran roughly four times faster [11, p. 1034f].

3.4.2 Technology

diff's core task is finding the "shortest sequence of insertions and deletions that will convert the first string to the second" [11, p. 1025] together with finding the longest common subsequence occurring in both files [6, p. 2]. Combined with the mathematical algorithm, it should provide an easily understandable format for humans, consisting of line numbers joined with *a, c* or *d* (append, change, delete), as well as $<$ and $>$ line prefixes, showing the affiliation either to the initial or compared file. This is called the "Normal Format" [10, p. 12]. Program 3.4 shows a sample output, comparing the strings `a b c d e f g` and `w a b x y z e` (one line per letter) [6, p. 1f].

The Unified Format

To provide a more readable user experience, GNU diff contains an improved format, called Unified Format, removing redundancy by using a more compact syntax. It can be selected as output format by executing diff together with a `-u` flag [10, p. 16], whereas git diff uses this as standard format to show changes within the current working tree [20].

As an example, Program 3.5 shows the same diff output as Program 3.4, only as Unified Format using the following components: `--- {filename} {timestamp}` indicates the initial file together with the timestamp it was created, whereas `+++ {filename} {timestamp}` is the same as above, but for

Program 3.5: unified_format.diff

```
--- oldfile 2017-04-13 09:42:47.474769553 +0200
+++ newfile 2017-04-13 09:43:13.898566935 +0200
@@ -1,7 +1,7 @@
+w
 a
 b
-c
-d
+x
+y
+z
 e
-f
-g

```

Program 3.6: file.diff

```
 1 --- oldfile 2017-04-13 09:33:18.092200876 +0200
 2 +++ newfile 2017-04-13 09:34:11.256529061 +0200
 3 @@ -1,8 +1,10 @@
 4  Hello,
 5 -this line gets deleted in newfile.txt,
 6 -as well as this, so both get prefixed with a minus
 7 +
 8  This line stays the same, but is preceded
 9  by a blank line, which gets prefixed with a plus
10 +Oh yes, and this line got added in newfile.txt
11  All other lines staying the same as in oldfile.txt
12  are prefixed with a blank space.
13     (such as this, which should be a blank line)
14 +   (but not this, as it also got added in newfile.txt)
15 +Good bye (also added)
```

the compared file. `@@ -{intial line range} +{compared line range} @@` shows the affected line range, where `-1,7` indicates the following 7 lines, starting from the first line of the initial file. Lastly `+` marks a line as added in compared file and `-` marks a line as deleted in the compared file. To make the above explanation a little bit more clear, an additional example with a text speaking for itself is shown in Program 3.6.

It can be clearly seen, that line 3 shows a growth of *newfile* by two lines: `-1,8` vs. `+1,10`. Having also a color-coded representation, it would boost the readability of such diff outputs once again.

Program 3.7: A snippet of a file called "manual.txt", which is affected by a conflict. Content between `HEAD` and `=======` contains the local version, content below contains the foreign conflicting version.

```
<<<<<<< HEAD:manual.txt
I (the developer) am right!
=======
The branch_name is right!
>>>>>>> branch_name:manual.txt
```

Usage with Git

As already stated, diff is one of the core components of Git. Not only does it support determining changes in the source code between a snapshot and another, it may also reveal merge conflicts, if segments are mutually exclusive and therefore preventing a flawless propagation of development. Thus, a varying development history of different origins (e.g. branches) not compatible to each other might be indicated. Furthermore, a conflict may also happen, if a developer forgot to *pull* the latest changes before committing his/her current development progress. These conflicts may only be handled through human guidance [9, p. 124].

A conflict presents itself primarily through a message similar to:

```
CONFLICT (content): Merge conflict in file
Automatic merge failed; fix conflicts and then commit the
result.
```

If anything like the above happens, the affected files by the conflict also contain a structure like shown in Program 3.7. A conflict can then be resolved by removing its markers and picking the appropriate resolution of either side of the `=======` delimiter [31], as well as mixing them to the developers needs [9, p. 126]. A single file may also contain multiple conflicts.

Usage with GitHub

Especially when interacting with GitHub's REST API, it is very easy to generate and import file diffs of a given repository. Whether two branches or two commits using their *SHA* values are compared, a single HTTP request suffices for programmatically retrieving data, which is normally only accessible using a terminal emulator.

Depending on the requested *media type* in the appropriate HTTP header field, either a full-featured diff, patch or JSON containing per-file patches is emitted by the API. If the latter was used, the underlying diff is translated into a JSON object, containing information like the number of additions, deletions and changes, as well as the mentioned *patch* for each file.

As a consequence, a repository does not necessarily have to be *cloned*, as it may be patched constantly using the API to keep it up to date – using this method, patch is even able to create and delete files, if necessary [10, p. 57]. The only prerequisite is to keep track of the single commit hashes the patches are applied from.

Chapter 4

Theoretical Approach

Once a decision is made in favor of a project using a static site generator, first challenges may already arise:

- What kind of media is being used? Images, videos, or just text?
- How is the project structured? How many independent permalink structures are there?
- How many content authors are there? How often is content added or altered?
- How steady is the site design? Does the site have more than one independent design flow?

Some content-related issues may not be improved in a predictable amount of time, but even in projects where only a high amount of content productivity is pursued, developers are forced to keep the build pipeline performant and responsive to the content author's needs.

4.1 Challenges

As already stated above, nearly every web project bears challenges to solve, both for developers on the one hand, and for content creators on the other hand. While content creators mostly need to solve structural issues in the published content, developers are mainly responsible for supporting authors in technical questions, as well as constantly keeping an eye on the backend development. This might go from always keeping the underlying modules updated, to populating the source code with design or template changes, to finally maintaining the build pipeline and deployment setup.

4.1.1 Distributed development

When it comes to administer a static site project, it is very likely that there will not be any possiblity of working on the same project in the same

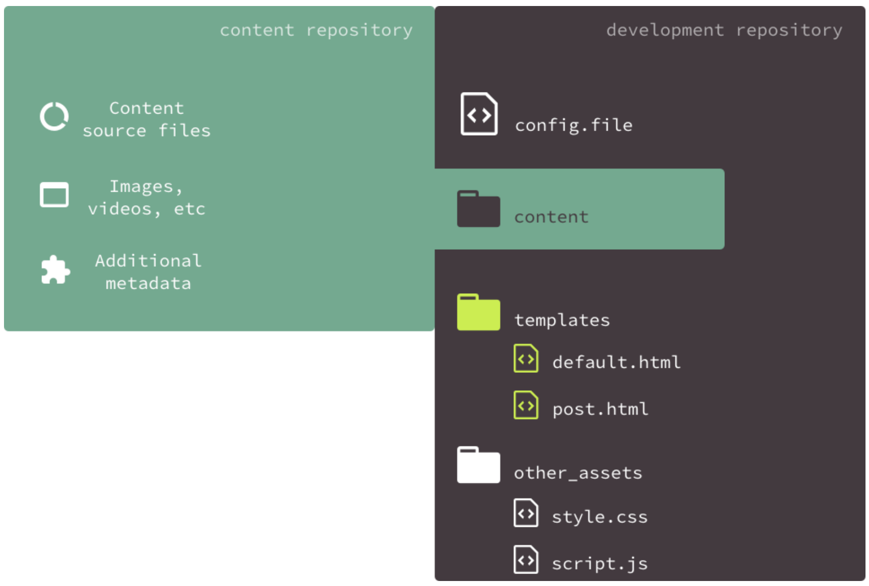


Figure 4.1: A graphic showing the stylized separation of a project into a *content* and a *development* repository. The content authors may only be granted access to the content repository, while developers should be granted access to both, thus providing a seamless integration for the content into the build pipeline flow (see Fig. 3.1).

environment in a linear way. Instead, every developer will have to have a local install of the used generator at his/her disposal, together with access to a remote repository of a version control system for exchanging the current development status with other project maintainers. The main reason for that is the fact, that unlike content authors, developers do have the obligation of installing or maintaining the project's dependencies [3, p. 85].

Content Editors on the other hand may, if using for example GitHub, make use of the remote "In-Page Code Editor" (see Fig. 3.3), which also provides an optimistically rendered version of the current content, although without making use of the project's style sheet.

Separating content from code

As constantly growing static site projects may sooner or later come to a point, where content progression differs from development progression, it might be useful to separate both parts into independent repositories (see Fig. 4.1). This especially makes sense, if the content editor team is also locally separated from the development team and therefore an additional level of security against accidental branch intermixture is needed.

Merging only using pull requests

However, if this kind of strong division is not desired anyhow, another option would be to limit access to the development repository in a way, that everyone may *fork* a repository, but only certain project users are allowed to *merge* external branches into the main development tree. On GitHub, *pull requests* may be used. These pull requests allow any user to announce his/her contribution to the project using a commit history of a forked repository. The source project owners may then decide whether or not to merge the announced changes and also have the chance to express their point of view via comments directly on the pull request to its creator [9, p. 394f].

The point in time, when a pull request is created is subsidiary, as further development on the specific branch is automatically included as well. Thus, per-line comments are also removed, once the specific line of code has been modified in a following commit [27]. Furthermore, the possibility to merge is checked after every commit pushed to the respective branch, so the responsible users always know, if a merge operation may be successfully executed. Otherwise, a merge is only possible after locally checking out the pull request and processing it using the command line [30].

Staging versions

Working with separate remote branches on a version control system like Git also allows for staging environments and therefore testing different versions of the projects concurrently. The public version however, visible to all clients visiting the website, remains stable and should receive only well-tested or well-considered updates as the very last step in the ongoing development.

To achieve this goal, a testbed is necessary and may be realized using another branch besides “master”. Sometimes an additional “bleeding edge” version is also likely to be included in the build process. Based on this strategy, it is easy to control and maintain different revisions at the same time and nevertheless infer the functionality of different proof of concepts for merging them into the public version later on.

4.1.2 Build cycles

One of the major challenges using static site generators remains the issue of providing a “real website look and feel” to the content editor. Whereas authors in dynamic CMSs are presented with an already pre-rendered version of the newly added content (since the underlying system is not dependent on any template rendering before deployment), static site generators only offer a glance of the author’s work, after the whole build pipeline process succeeded in its render flow, unless other pre-caching methods are used. Yet, most static site generators do not include such kind of tools by default.

Based on the size and the amount of files in the website source code, this

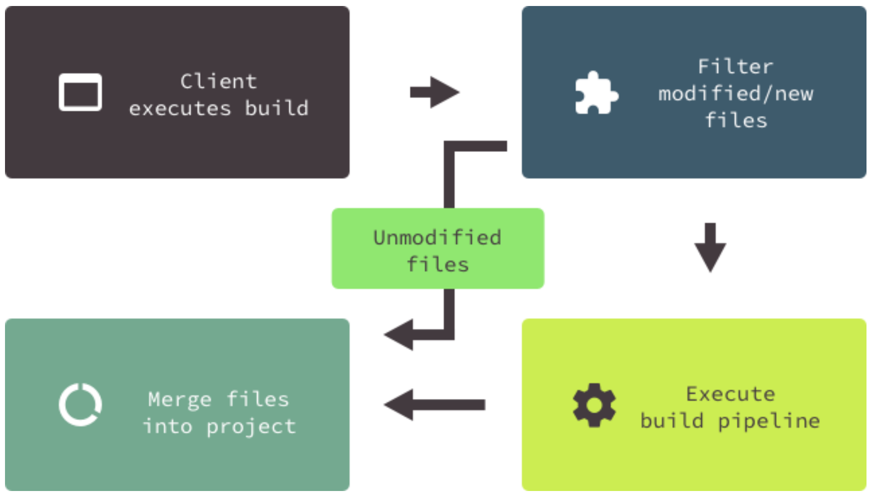


Figure 4.2: A graphic showing the theoretical approach of a build process flow, supported by caching. After the *client* (content creator, developer) executes a build, the included caching mechanism should filter modified or added files and send them to the build pipeline. After the build succeeded, the newly built files should be merged with the already existing files to form an updated version of the website root.

time frame can easily grow linearly. If there are also additional tasks added to the build process, such as resizing images to different screen sizes for providing a responsive user experience, the computational effort may easily get out of hand and therefore the duration until the content editor first sees the result of his/her work simply gets unacceptable.

Possible problems of long-lasting build processes

Waiting for the completion of the build pipeline can cause severe recesses in the work performance of a content editor or developer, as further work depends on its success, while a failure is often combined with time loss beforehand and intensive bug hunting afterwards – probably resulting in even more consecutive build pipeline failures. This assertion may not only be linked to crucial modifications in development, also the smallest hotfixes might as well provoke a full rebuild without justifying the whole effort.

Furthermore, being forced to wait in line may cause a developer to loose track on the development process, thus the introduction of hidden bugs (although not resulting in a build failure) is more likely. Additionally, mindlessly executing build cycles may even lead to data loss or blocking the workflow of other contributors.

4.1.3 Caching

Speeding up a build process can be done via *caching*. The right caching method should differ between unmodified content and files which actually have been reworked or were newly introduced into the project source. Using this sort of information, the algorithm might only choose the latter files, send them to the build pipeline and merge the outcome to the already existing project build (see Fig. 4.2). Although a few static site generators already include some sort of caching methods – although most of them only work locally (like Hexo's Warehouse, see Sec. 2.2.2), a first step is made. It should significantly improve the build duration for local development, as long as the optimal cache storage is being used. Hexo's Warehouse uses a JSON file as persistant storage, while the temporary storage lies in the Computer's RAM. This is fine for smaller projects, but could also lead to critical memory issues when used in projects containing a huge amount of files. For bigger data sets, it would be possible to use caching in conjunction with databases like SQLite using the *JSON1*[1] extension.

As the local cache may not easily be transferred to other contributors or deployment engines, the first build after a pull does not take advantage of any speed up technique. Moreover, the methods mentioned above all use a significant amount of local computing power to provide a useable cache, which could lead to problems and unwanted slowdowns on portable devices.

Determination of cacheable contents

The next step would be to select cacheable files, as not every file has the same impact on the project source. While normal blog posts are mostly self-contained – unless there is probably a preview of a post featured somewhere, a template file on the other hand is often a dependency for many blog posts. A working cache is more important for a commit only containing content data, than for a commit which only contains template files, as a template file acts as a dependency for possibly multiple content files. Therefore a template file invalidates the cache, which leads to a complete rebuild.

Since the relations between files, as well as their dependencies, may appear a bit obscure at first sight, it needs some kind of tracking system for ensuring a persistent overview of cacheable contents. This does not stop at comparing the file path within the project tree, as for example, all template files are stored within one special directory. For future considerations, also the source code would have to be examined, as the frontmatter in content files may contain information about dependencies to external files (see Sec. 3.1.1). Therefore, the challenge here would be a reverse lookup mechanism for spotting file relations in the project source.

[1]https://www.sqlite.org/json1.html – JSON1 documentation on SQLite's website.

4.2 Solution strategies

A primary task would be the focusing on critical issues to at least boosting the project's overall performance noticeably without losing too much track. This may be worked on in terms of collaboration, as well as in the project's setup, where on the one hand the team's performance and on the other hand the build engine's performance should be improved.

A lot of issues can be covered using GitHub's API, although some project specific adaptions are still necessary. Nevertheless, the API provides enough information for quickly perceiving a sufficient overview of the respective repository.

4.2.1 Distributed development on GitHub

Based on GitHub's API and the advantages of using Git as version control system, it is definitely a significant benefit to equip all project contributors with a GitHub account. While the public, open-source model is free of charge (see Sec. 3.3.1), there are also different pricing models offered for privately held projects, which should be hidden to the public[2].

Although dependent on financial expenditures, the additional value of working on a project with closed source (though it may be released as open source somewhere in the future) may be worth considering. Yet, full-featured access to the API is also included in the free tier though.

Not only GitHub offers a queryable API, also *Bitbucket* provides an API with similar response data[3] – although certain features are missing, compared to GitHub. These missing features include for example certain project download functions, among others.

In addition to the API, GitHub's built-in In-Page Code Editor plays an important role for choosing it as core support tool for this project. Therefore, merging using pull requests and a continuous branching model for supporting staging versions qualify as proposed strategies to developers.

4.2.2 Build cycles

Supporting content authors in their workflow also means to not require them to install unnecessary build tools manually, unless critically needed. Due to the possibility of using GitHub's In-Page Editor, the whole Git checkout, commit and push process becomes in a way redundant too. Moreover, the online editor automatically creates pull requests on demand, so that the respective project owners should get notified automatically, if a merge is possible and therefore an update of the currently published project may be

[2]https://github.com/pricing – GitHub's pricing models.

[3]https://developer.atlassian.com/bitbucket/api/2/reference/ – Bitbucket's API documentation.

initiated.

Normally, a responsible user would pull the new state after a merge of the pull request, then execute the build pipeline. After the build process succeeded, he/she then has to take care for updating the webroot on the server, so that the newest version of the website gets delivered to the client upon request. However, this practice may easily get cumbersome, as the respective developer might get distracted by checking out new branches and possibly leaving behind his/her own work for the moment. Moreover, if the deployment has to be done manually, additional mistakes may happen during the whole action (see Sec. 4.1.2).

In this case, it makes sense to remotely outsource the build service and to possibly even automatically execute the render cycle. Used in conjunction with GitHub's *Webhooks*, an external service would receive a build execution order via a HTTP POST request, based on certain predefined events in the GitHub repository [32]. Apart from the information the webhook provides, the service would even accept custom build options issued by responsible users, as the endpoint has to be publicly available anyhow. Once a build succeeds, the service should then notify a predefined list of users about the render cycle result and provide the outcome via download possibility.

Existing remote services for static site generators

CloudCannon[4] is probably the most popular online static site generator and offers a commercial external building service for Jekyll projects, together with source access using a GitHub or Bitbucket repository. According to its documentation, it currently supports Jekyll projects running *v2.4.0* or newer[5]. It features a project file explorer and presents every new project as opinionated as Jekyll usually does (see Sec. 2.1.2), as well as an automatic deployment service on their own subdomains. However, access to the rendered website files is not included, so every customer is dependent on using their hosting service.

BowTie[6] is similar to CloudCannon and is also offering a commercial online service. It seems to be a much more standalone service than CloudCannon, though it also offers GitHub integration, as well as custom Webhooks for event-based actions on external services.

Pancake[7] is a free service for externally building static sites. It features an engine auto-detection and currently supports *Jekyll, Wintersmith, Pelican, Sphinx, Hyde* and *Middleman*[8]. Due to its non-commercial version, several

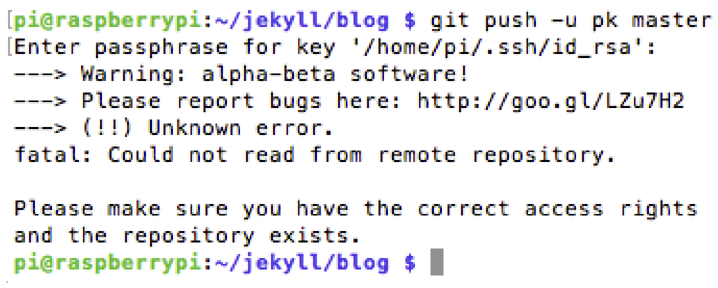
[4] http://cloudcannon.com – CloudCannon, the Cloud CMS for Jekyll.

[5] https://docs.cloudcannon.com/building/versions/ – Supported versions on CloudCannon documentation.

[6] https://bowtie.io – Website of BowTie.

[7] https://www.pancake.io – Website of Pancake.

[8] https://github.com/pancakeio/detect/blob/master/heuristics.go – Currently supported



```
pi@raspberrypi:~/jekyll/blog $ git push -u pk master
Enter passphrase for key '/home/pi/.ssh/id_rsa':
---> Warning: alpha-beta software!
---> Please report bugs here: http://goo.gl/LZu7H2
---> (!!) Unknown error.
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
pi@raspberrypi:~/jekyll/blog $ 
```

Figure 4.3: A screenshot of an approach to pushing a Jekyll project to *Pancake*. As a result, the operation failed with an "Unknown error".

restrictions are to be considered [38].

However, the service does currently not run stable, as an initial project setup failed (see Fig. 4.3). It seems that Pancake uses a post-receive hook for automatically trying to build a project, once it detected the underlying engine type. This causes waiting time for the developer on the one hand, but on the other hand informs whether a build was successful or failed. During another push attempt, it failed, as *bundler*[9], a gem dependency management tool required by Jekyll, was not mentioned in the "Gemfile" contained in the repository.

4.2.3 Caching

As stated before, most static site generators do not contain any form of caching mechanism by default – if they do, caching is limited to the local machine a build is executed on. Since there probably is no easy way of providing a form of remote caching, as this largely includes the necessity of external services to exchange a common status, as well as an index containing information about source and destination files for later rebuilds, it needs an equivalent strategy, which merely contains these information from a certain point in the past to the present, without relying on physical file structures to be exchanged.

Furthermore, such a caching strategy must be universally useable across all operating systems and ideally does not require any additional setup from the user. Moreover, it should also feature hassle-free integration into any project without depending on an external, yet unused service.

Keeping all of these issues under consideration, not every suggestion might get featured equally in the final solution – the main reason is, that a kind of transformation like the one caused by a build pipeline always needs an existing status to build up from. So, tradeoffs are likely to accompany

static site generators by Pancake in raw source file on GitHub.

[9]http://bundler.io – Bundler, a gem dependency manager.

any form of decision to be made in this case.

Caching based on diff

As Git was chosen as version control system, diff is already part of the development suite. Therefore, a gapless detection of development progress between two arbitrary commits is possible. The diff format can be parsed to JSON and makes it easy for use in JavaScript. Thus, its usability for further processing on application level is assured[10].

The most important parts of a diff representation in this context are the file paths, as well as the type of modification on each file affected in the respective time span. Considering this kind of information, an existing repository might be quickly divided into unaffected and affected files – where affected files, as well as their dependents possibly need to be selected for a rebuild. The final decision of the rebuild extent based on the diff, however, should be based on heuristics.

To conclude the consideration of using diff, the approach explained above is different from "classical" caching. Such a mechanism, founded on diff, is not dependent on support-files produced on its own (like a caching catalogue), but it requires a consistent and strict git workflow, otherwise it has no control over untracked files.

4.3 Considerations towards implementation

After looking at challenges and possible solutions, the following topics can be identified as being essential for:

Remote – Outsorce long-lasting actions to an external service.

Caching – Speed up builds by making use of already finished work.

Versioning – Keep track on development and possibly revert, if necessary.

Branching – Let different parts of development evolve to their own speed.

Git qualifies as core companion to any website project, especially when the project itself is maintained by multiple developers, designers and/or authors. Being aware of GitHub as social code management tool and moreover the benefits of its API, it serves well as foundation for the solution (see Sec. 4.2.1).

Since the tool should also be remotely accessible, it makes sense to also design it as RESTful API, for handling programmatical access as well as access from possible frontend apps lying on top. Furthermore, its main work cycle might get detached for neither distracting users due to ordering them to wait until it finished, nor blocking access in between (see Fig. 4.4). However,

[10] https://runkit.com/saschazar21/diff-parsing-demo – An interactive example for fetching and parsing a diff-file.

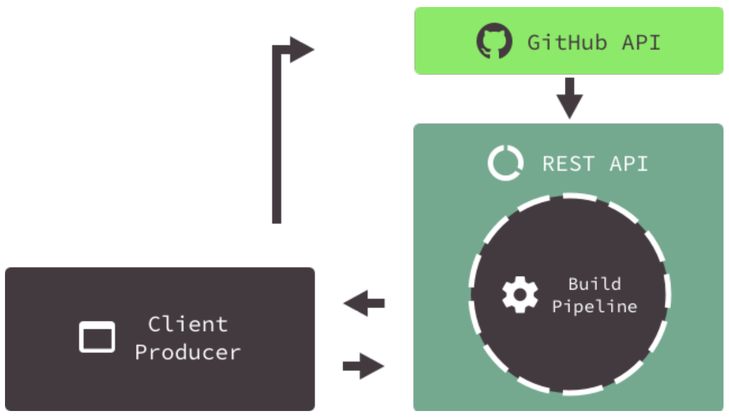


Figure 4.4: An abstract flow visualization of the planned request cycle. The client (developer, content creator) manages his/her code on GitHub. Based on the respective configuration, a build cycle may be triggered automatically using a GitHub webhook, or manually by sending a POST request to a certain endpoint. This creates an instance of the build pipeline. The pipeline requests data of the project from GitHub and sets up the project configuration. After a successful build, the REST API provides a downloadable archive of the newly built webroot.

the most important part behind these thoughts is the choice of the ideal static site generator system.

4.3.1 Choosing a static site generator

The evaluation needs to cover the usability, pluggability, customizability and overall maintenance, as well as the level of its general support of the candidate systems. First and foremost, the installation process should be as easy as possible and not rely on too many third-party dependencies, which are probably not needed afterwards. This improves the maintainability of the system.

The programming language of the chosen static site generator does have to be considered well, as it has to fit seamlessly into a planned REST API, in the best case without any further adapter in between. This should provide an easy way to hook additional code into the configuration step, if needed. Ideally, it emits events as well, so any host process knows when a detached process is finished.

All in all, the currently best solution seems to be *Metalsmith* (see Sec. 2.3), as it offers a pluggable module ecosystem, and also provides access to a JavaScript API, among others. Together with some custom tweaks (e.g. dynamic module loading), an independent build setup for each project may be injected using only a specific configuration file.

Program 4.1: An example for a basic express.js setup, roughly taken from http://expressjs.com/en/starter/hello-world.html. In this case, a web application listens for a *GET* request on its root path "/" and responds with a "Hello World!" message.

```
const express = require('express');              // require base module
const app = express();                           // Create instance

app.get('/', function rootRoute(req, res) {      // Listen for GET request
  return res.send('Hello World!');               // Send response
});

app.listen(3000);                                // Listen on port 3000
```

4.3.2 Constructing a REST API

JavaScript proved its universality due to its usage both on client- and server-side. Node.js is a server-side implementation for JavaScript, backed by Google's V8 engine, which directly translates the scripting language into machine code [1, p. 4]. A seamless integration of Metalsmith into the API service may therefore happen without much hassle.

The easy installation is supported by several third-party apps like Node Version Manager (*NVM*)[11] and mostly will not need any admin rights, which makes it ideal to use on hosting environments without root access (unlike PHP or Ruby for example). Although not equally well supported among the most popular operating systems, at least MacOS and Linux provide a stable enough environment for NVM.

Looking for a framework for setting up an API, *Express*[12] is well suited for this task, as it only consists of a very basic setup – similar to Metalsmith – but may be easily enhanced using different node modules, thus providing a uniquely shaped web application in contrast to conventional, monolithic frameworks like *Django* or *Ruby on Rails* [1, p. 176].

As a result, the main purpose of such an express application would be acting as a web-based infrastructure for the underlying build pipeline. Based on different REST endpoints, as well as their request parameters, executing a uniquely configured Metalsmith process on a project directory within the API's file tree should be possible without any further external interaction requirement. The project directory would be provided using an appropriate GitHub repository, requiring only a basic, as little opinionated configuration as possible, together with a public access possibility to the repo.

Additionally, to benefit most from any remote outsourcing, a current production-ready version of the website may be held available at a special

[11] https://github.com/creationix/nvm – NVM's repository on GitHub.
[12] http://expressjs.com/ – Website of Express.

endpoint for downloading at any time. In this case, deployment is enabled without waiting for completion of any build cycle beforehand, unless the source code received any updates through development.

4.3.3 Caching and selective rendering

As the technical foundations for a project covering the use case of static site generation are now defined, a constantly growing amount of necessary build time still remains as one of the core problems. Caching across remote machines is likely to be impossible, especially if local computers also have to be added to the caching network and not every node is working on the same project revision at the same time. Moreover, an additional distribution mechanism would also have to be implemented, acting primarily as supply tool for providing already rendered revisions of different steps in development.

Concentrating on basic improvements of speeding up a build cycle, a solution without exchanging complete file trees might be possible. To make sure all required data is accessible, the respective GitHub API credentials are mandatory. The main reason behind that is the gapless availability of every commit and its underlying file tree via HTTP – therefore a separate `git checkout` on server level is not needed, GitHub provides the according file tree as immediately obtainable tarball or zip archive.

Together with the development history between two individual commits resulting in a diff and a downloadable file tree representing a certain step in the current progress, a build log has to keep track of the ongoing rendering actions and their results. The idea behind that is the possibility of remembering any render action in the past and incrementally building up on the latest positive result by only selecting the modified files for use in the build pipeline and leave out any other. Relying on an already available bugless result of a previous build cycle, any successful outcome of an upcoming action based on a later commit may be easily merged (see Sec. 4.2.3).

Chapter 5

Implementation

After shaping a basic theoretical approach and probably sketching different considerations, it is time to define the needed tools and setting up a project repository. Also, it has to be considered, that the finished project should depend on as little third-party software as possible, but also be as deployable as possible.

At the very beginning, a clear structure has to be adopted, which concurrently serves as navigation guide for later development. Having a folder structure ready at development start, it may seem rigid and even narrow down the developer's freedom in creating his/her part of the application logic during some part of the process, but nevertheless it is definitely an important and easy supporting tool for projects being maintained using an asynchronous collaboration workflow. Furthermore, it may even help in creating tasks focusing on certain parts of the development process.

5.1 Foundation

Since Node.js is very suitable for providing an instant development environment on most popular operating systems[1], as well as quickly leveraging a basic application, which provides immediate feedback to its creator – without depending on any precompilation steps – it may be considered as basic framework for any further development.

Together with the achievements of ES6, a clean code foundation marks the base structure for further module introduction into the project. Step by step, a modular web service is going to be raised and formed according to its designed operation mode.

[1]http://nodejs.org/dist/latest/ – Precompiled versions of the latest Node.js release, for different operating systems.

5.1.1 Express.js for REST

Starting with Express.js, the sample code in Sec. 4.3.2 shows a reasonable example on how to easily provide an API endpoint. While the example only returns a string containing “Hello World!”, a JSON structure may also be used and is probably a better choice for working programmatically on the response data later on.

Furthermore, a good advice would be to use a modular form of route definitions, since the main source file will soon get too bloated and may grow a lot of spaghetti-code in it. This may be achieved in outsourcing the routes in specific files and/or folders and importing them via a *require*-statement. As a bonus, an external source file containing route definitions also allows for custom logic and middlewares, which may be hidden to the rest of the application by default [1, p. 220f].

Middleware

Especially when depending on advanced application logic (e.g. user authentication, database management, etc. . .), further tasks containing validation checks or user definitions may get necessary. If these tasks are required by more than one route, it makes sense to abstract their logic into reusable components for use as middleware in these specific routes [1, p. 223]. Optionally, more than one middleware may be used on a single route, where their placement stands for their execution order – from first to last.

Listing 5.1: An example for middleware ordering, where *firstMiddleware* gets called right before *secondMiddleware*. Both middlewares have to succeed (e.g. return *done*-callback function) in order to grant access to the “/secret” route.

```
const firstMiddleware = function (req, res, done) {
  // Application logic ...
  if (false) {
    return res.send('Not allowed');   // Middleware failed
  }
  return done();                      // Call done, if middleware succeeded
};

const secondMiddleware = function (req, res, done) {
  // More application logic ...
  return done();                      // Call done, if this middleware also
      succeeded
};

router.get('/secret', firstMiddleware, secondMiddleware, function
      callback(req, res) {
  // Finally access to route logic , if both middlewares succeeded
});
```

OAuth 2.0

OAuth 2.0 stands for an open authorization framework, which grants limited access to a certain HTTP service, either on behalf of a resource owner (e.g. allow access to user data of a social network account), or by allowing a third-party application to obtain access on its own behalf [5, p. 1]. In this case, the latter is more interesting, as a programmatical access may be achieved by issuing an access token via a "client credentials" grant type. Therefore, an application-only access is possible without depending on any user interaction.

The whole authentication process is necessary, as the final web application will hold different user accounts, as well as their registered projects. Thus, every client (human or non-human) may interact with the application's API only via certain issued tokens, which ideally are only valid for a specific amount of time before they expire [5, p. 43].

5.1.2 MongoDB

Every account or project data has to be stored on a non-volatile type of memory to faithfully provide any requested information at any desired point in time. Moreover, these data requests may not interfere with each other, nor cause inconsistencies or conflicts within the storage, even if accessed at the same time. As a consequence, a memory solution depending on files will not likely fulfill every crucial requirement, especially when a service is constantly and fast growing.

A good choice is therefore to use *MongoDB*, since it stores the entries already as formatted JSON and is not depending on a fixed table schema beforehand. As a result, the structure most likely does not have to be excessively administered during development and stays as adaptive as possible until a final schema has evolved.

Additionally, MongoDB also features the possibility of administration via JavaScript-files on the server-side. These files may not only query the database for entries, but also contain predefined tasks for manipulating the contents – critical commands therefore should be rather automated by a *Cronjob* or executed by the user from within the *mongo* shell by only using this form of interaction method [33]. This supports preventing mistyped commands as well as reducing the risk of data loss on the database. Moreover, it may also help in constantly keeping a second database updated, which is intended for testing purposes and is assumedly relying on a production-like ecosystem.

In this case, the database is used for holding user data together with data logged by the API during every single build process. Thus, the user may get provided with a seamless reproduction of the build history of his/her project at any time.

Schemas

Besides holding user data and information about the build cycles, the database also is the first section to be addressed in terms of information about the currently processed repository itself. This saves time and reduces the access rate to the GitHub API, as the most important information about the repository and its contributors gets stored in the database upon registration using the REST API.

However, most of the required data during a build cycle still is fetched from GitHub, as the database's core task is the user management, as well as the storage of build logs. It would be a bit redundant to also constantly update repository changes on GitHub into the database.

Therefore, the schema structure on the database can be listed as follows:

users – Holds the plain user data; users have to sign up using the e-mail addresses they are also using on GitHub. The schema then contains the user's name, the hashed password, as well as the e-mail address. Additionally, the respective GitHub user data is also stored using a *github* property.

clients – Registered users have to register a client for OAuth 2.0 token exchange. Client ID and password are generated automatically by the REST API. Besides the client secret, there is also a reference to the respective user entry present.

accesstokens – Holds access tokens issued by the OAuth 2.0 framework. The tokens are only valid for 60 minutes and are checked for validity upon every request (see Sec. 5.1.1).

projects – Once new repositories are registered using the REST API, data from GitHub is fetched and stored together with references to the user schema. Furthermore, every time a build log was created, the respective array in the project schema grows by one reference to the *builds* schema.

builds – Every time a build cycle was triggered and succeeded in validating the repository information on GitHub, an intermediate build entry is created. Finally, after the process finished, the success property in the build log is updated accordingly (either *true* or *false*). To provide precise, usable information, the schema also contains the base and head hash of the current commit range, the start and end timestamp of the process, as well as an array containing rendered file paths. If this array is empty, it indicates either an initial build, or a full rebuild. Furthermore, the *filename* property stores the name of the generated tar.gz archive (see fig 6.3).

5.1.3 GitHub API

Since the project will require every suitable static site source to be hosted on GitHub, the GitHub API is able to provide short and fast overviews of their current state. For providing an automated static site builder, the project needs to rely on some crucial information before being able to set everything in place.

Because of the fact, that speed and performance are the dominant factors in such considerations, it is unlikely to look up information from a "physical" file tree on the disk, unless it cannot be done using a service, which already delivers data in an understandable form. Hence the GitHub API offers a wide range of information about a hosted repository and furthermore is able to cope with a constantly high amount of load, it should be a good fit while even having the need of sending multiple requests until receiving the right extent of data.

The most important API calls for this project are listed below, their description may be found on GitHub's API Reference[2].

- `/repos/:owner/:repo/git/trees/:sha` – Returns the file tree of a repository at a given commit hash.
- `/repos/:owner/:repo/contents/:path` – Returns the contents of a file or directory.
- `/repos/:owner/:repo/:archive_format/:ref` – Returns the link for a repo tarball at a given reference (e.g. *master*, but can also be a commit hash).
- `/repos/:owner/:repo/commits` – Returns commits for the given repository.
- `/repos/:owner/:repo/compare/:base...:head` – Returns affected files between a *base* and *head* commit. Comparison between branches and/or forked repositories is also possible.

Get a Tree

One of the most important steps prior to working with a foreign website source is the examination of its file structure, especially when dealing with static site sources. Unlike any other dynamic project, which probably updates itself using certain online sources under predefined circumstances, a static site nearly always has to be rebuilt from scratch.

Although it is also possible to unveil the desired state from the past by reverting to that commit, multiple actions have to be taken before succeeding in that task. This goes from checking out the branch the commit belongs to, triggering a `git revert` command – which reverts the changes made up

[2] https://developer.github.com/v3/ – API Reference for version 3 on GitHub.

to the current commit [21] – to reading in the file tree using any third-party plugin like *node-klaw*[3].

When calling the “Get a Tree” endpoint on the GitHub API although via a GET request, the response contains an array of objects, including file paths, as well as information whether it is a file or directory (among others). Based on the provided parameters like *recursive*, the endpoint filters every file and directory down to the lowest level, instead of returning only the files living in the top level of the repository. If the response also bears a *truncated* key with value “true”, the repository exceeded the file limit and not every file could be filtered. In this case, a manual checkout would be necessary[4].

Get contents

After having caught a glimpse of the file structure, it is easy to look up certain file names and therefore check for their existence. Also, it allows for failing early in case of any user defined error concerning the availability of certain files before downloading its contents. Whenever the existence of a desired file is assured by having found its position in the file tree, the API is able to send its raw contents upon requesting the “Get contents” endpoint together with the appropriate *Accept* header[5].

The use of this endpoint is as simple as valuable, as it lets the project’s engine parse single files (e.g. configurations), without being dependent on downloading whole tarballs or zip archives. This completes the fail-early stage, as it not only checks for existence, but also for parseability of crucial files before requesting and processing heavy project archives. On the other hand, it is also necessary for setting up the required build steps, which should enable a parallel workflow, once everything comes into play.

One of the core limitations, which should be considered here, is a maximum file size of 1MB. Whenever a file exceeds this limit, it should be downloaded as a buffer and read in manually afterwards.

Get archive link

Once all the preceding checks have passed, one of the heavier tasks may be executed. Since the possible integrity of the project was assumed using the successfully parsed configuration file, the subsequent step would be to download the static site repository, in fact at a certain commit state in the past, probably.

However, this can only be done using GitHub’s API, as the “Get archive link” endpoint might be one of the very few, which are not available in the

[3] https://github.com/jprichardson/node-klaw – Klaw’s repository on GitHub.

[4] https://developer.github.com/v3/git/trees/#get-a-tree – “Get a Tree”-section in the GitHub API Reference.

[5] https://developer.github.com/v3/repos/contents/#get-contents – “Get contents”-section in the GitHub API Reference.

Browser version yet. After the endpoint received a request by the client, it produces an archive according to the provided parameters and prepares a download possibility at a URL, which gets included in the response header, together with a "302 Found" HTTP status. Either the framework is configured to automatically follow the issued URL, or a second request to the location included in the *Location* header field, becomes necessary. Private repository archives are issued together with a quickly expiring token[6].

After the download succeeded, a final step would be to extract the archive in a project working directory – therefore, any further command line interaction for this task becomes obsolete.

List commits on a repository

For keeping track on a project, git commits contain a lot of information about the project – however, probably the most important information is the list of affected files since the last commit [9, p. 65]. Normally, git projects include these snapshots automatically, but since the downloaded archive from the GitHub API does not contain any git-specific files, but only source code immediately belonging to the project, the commits have to be obtained from another source.

Due to the fact, that the project will store build results into the database, there might always be a reference to a previous build process and the commit hash used as state of development progress. This will therefore be a starting point from which to query commits from the GitHub API. The "List commits on a repository" endpoint provides all commit information corresponding to the included parameters in the request, up to a certain amount, before subsequent pagination requests become necessary[7].

The call to this endpoint especially makes sense, if a certain time frame has to be queried and only timestamps are available, as the call may also include *since* and *until* parameters in ISO 8601 format.

Compare two commits

Following the download of the website source repository and its commit data, the affected files of the development progress since the last build are the next task in determining the build extent. Because of the caching approach, this is one of the most critical steps, as the affected files mark the beginning of a list of files which need to be considered for a rebuild task.

One of the biggest advantages of this API endpoint is the fact, that parsing the diff manually may be fully omitted, as the response already contains

[6]https://developer.github.com/v3/repos/contents/#get-archive-link – "Get archive link"-section in the GitHub API Reference.

[7]https://developer.github.com/v3/repos/commits/#list-commits-on-a-repository – "List commits on a repository"-section in the GitHub API Reference.

a JSON including all the necessary data concerning the repository modifications within the given time frame. Again, based on the *Accept* header, a custom response in patch or diff format can also be enforced, if it needs to.

After fetching the diff data, the list of affected files only has to be filtered based on the modification type (status) of each included file (modified, added, deleted) and synchronized with the existing file tree. All other files may be safely deleted before the build process starts, unless they are critically needed by the static site generator or share any form of dependency to the files present in the diff.

5.1.4 Metalsmith

As already stated in Sec. 4.3.1, Metalsmith would be a good fit for a project like this. The most important part after setting up a REST API using Express, is to include Metalsmith's JavaScript API as a module for on-demand building – first and foremost although, a strong focus should be kept on maintaining its customizability in terms of loadable modules.

Metalsmith instance

Creating a Metalsmith instance marks the beginning of any further build pipeline interaction, as it holds all the necessary modules, configurations, as well as the asynchronous *build*-function, caring for the overall work and furthermore emitting different events for signaling errors or warnings during the process itself.

It is included the same way as any other ordinary Node.js module, therefore the first step towards a dynamic setup is taken – basically all that needs to be done in the first place, is the provision of a callback function, containing a reference to Metalsmith, as well as taking a configuration object as parameter.

Before being ready to get exported, the Metalsmith instance has to handle the configuration object provided by the function parameter to form at least a basic shape of a useable static site generator. Compared to Program 2.1, this may look like Program 5.1.

After the Metalsmith instance was returned from the module, the receiving function only has to execute `metal.build()` for the build process to start. If acting from a foreign working directory, a smart advice would also be to change the CWD to the folder Metalsmith should perform his actions in. Some of the community-built modules tend to ignore the working directory set in Metalsmith's configuration and use the current one instead, which could lead to unwanted side effects of not handling certain tasks at all.

Program 5.1: A sample file showing a Metalsmith setup handled as a module. The *module.exports*-function marks the entry point from outside. The *return*-statement is everything a function from outside will see.

```
 1 const metalsmith = require('metalsmith');   // Require metalsmith module.
 2
 3 module.exports = function metalsmithSetup(obj) {
 4   const config = obj;                       // Set obj as config variable.
 5   const metal = metalsmith(config.cwd)      // Set current working directory.
 6     .source(config.source)
 7     .destination(config.destination)
 8     .metadata(config.metadata);             // Set global metadata
 9
10   // Loop through config.plugins array of objects (name, configuration).
11   config.plugins.forEach(function getPlugin(plugin) {
12     // Dynamically require() plugin name.
13     const loadedPlugin = require(plugin.name);
14
15     // Append plugin to metalsmith instance.
16     metal.use(loadedPlugin(plugin.configuration));
17   });
18
19   return metal;                            // Return the configured instance.
20 };
```

Dynamic module loading

Line 13 of Program 5.1 already shows what dynamic module loading is all about. Of course, the required modules differ from website project to website project, therefore all needed modules have to be listed in a configuration object – in the best case, they are included in an iterable form, so that the setup function may process one module after the other.

However, one of the biggest challenges is the verification of available modules, otherwise missing modules need to be downloaded prior to inclusion on runtime. Together with setting up the instance, these two steps probably require the most background checks for not being responsible to crash the application if any error occurs.

5.2 Application structure

A basic approach of the project structure may be seen in Fig. 5.1. Although this might look still very abstract, the core packages are already clearly visible, while neither the access to the GitHub API, nor the access to the MongoDB is yet visualized. The graphic can be interpreted as follows:

- *Base Application* – The base structure, consisting of a Node.js environment, together with necessary supporting packages, such as a

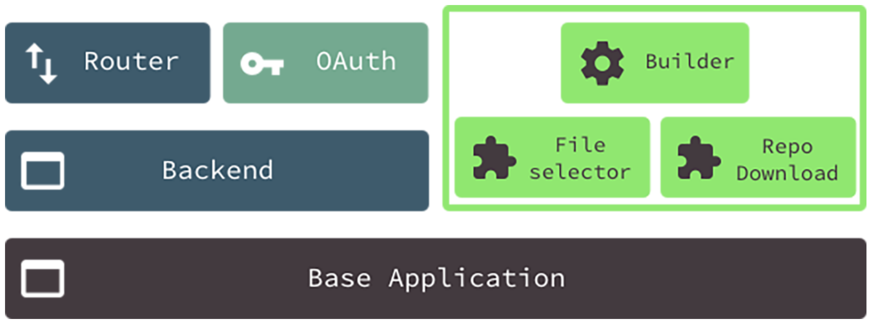


Figure 5.1: A graphic showing the base structure of the implemented application. The *base application* layer serves as foundation, containing necessary libraries for implementing the *HTTP* specifications. The *routing* and *OAuth* layer are responsible for authenticated requests to the endpoints, while the *builder package* is designed as a partly autonomous, loosely coupled rendering service.

MongoDB driver and a GitHub API implementation.

- *Backend* – The Express.js ecosystem, responsible for controlling the HTTP subset.
- *Router* – The Express.js router instance, providing all the necessary endpoints for accessing the application's functions.
- *OAuth* – The authentication framework, as theoretically explained in Sec. 5.1.1.
- *Builder* – The main build pipeline package consisting of many small plugins for asynchronous handling the process from parsing the configuration to actually building the website.

5.2.1 Basic setup

The base application layer is more or less a Node.js stack, covering necessary support features, like reading environment variables or creating various instances of needed modules for the main application flow. It also cares for connecting the service to a MongoDB database, as well as providing a connection framework to the GitHub API.

Furthermore, it holds different database models for user registration, OAuth tokens and build logs. These models are necessary for maintaining a consistent structure on the database collections, thus avoiding custom value checks after fetching entries. Using *Mongoose*[8], additional features like manipulation functions and automatic population may be used without depending on other toolsets. One example would be the automatic hashing

[8]http://mongoosejs.com – Mongoose, "elegant mongodb object modeling for node.js"

and comparison of passwords, which is enabled using *pre* hooks on schemas at a certain event (e.g., "save")[9].

Express.js

On top of the base application layer, an Express.js setup works as a REST API service. It is configured as first instance in the application's main entry point and is bootstrapped right after the launch of the project. Extending the core module of Express is easy due to the built-in middleware pluggability. A middleware function may get added to the application by binding it to an instance of the app object using an `app.use()` call [26].

One of the additional middlewares used to extend the app instance is a logging mechanism called "morgan"[10], which allows a fully customizeable output format for logging HTTP requests and the duration until a response was sent. Another important extension is "method-override"[11]. This module allows the consideration of a *X-HTTP-Method-Override* field in the request header, sent by clients, which are not supporting request types like PUT or DELETE.

Router

The middleware concept is designed as a sequential flow of callback functions. Once a request is coming in, the instance is forwarding the data from middleware to middleware until either a response is returned and the middleware chain gets interrupted, or no additional function is left and the instance throws an error.

Thus, the routing mechanism is nothing more than a built-in middleware of Express. It allows for dividing incoming requests based on their URL structure and subsequently assigning them to their respective predefined tasks. These functions again may behave like middleware functions (e.g. for checking authorization, including abstracted functions, imposing pre-conditions, etc.) and therefore expand the callback cycle by additional functionality [25]. A sample implementation of routing middleware may be seen in Program 5.1.

Authentication

Several routes require authentication before being able to access, as a consequence, an automated mechanism handling all necessary steps for securely exchanging user details is inserted in front of the respective routes. Before

[9] http://mongoosejs.com/docs/api.html#schema_Schema-pre – "Pre" hook documentation for MongooseJS.

[10] https://github.com/expressjs/morgan – Morgan repository on GitHub.

[11] https://github.com/expressjs/method-override – Method-override repository on GitHub.

Program 5.2: A basic router configuration showing the use of an authorization service as a middleware, thus dividing the routes into fully accessible ones (*/all*) and ones with limited access (*/limited* and */secret*).

```javascript
const express = require('express');
const router = express.Router();

// Include OAuth configuration from ./oauth.js in middleware format.
const oauth = require('./oauth');

// fully accessible route without authentication
router.get('/all', function allRoute(req, res) {
  // route logic ...
});

// Every route from now on demands authentication via the following middleware.
router.use(oauth);

router.get('/limited', function limitedRoute(req, res) {
  // route logic ...
});

router.get('/secret', function secretRoute(req, res) {
  // route logic ...
});

// Export router instance for using as middleware in app's main entry point.
module.exports = router;
```

doing so, the OAuth stack has been implemented – the *OAuth2orize* package provides an authorization server toolkit for setting up a service implementing the OAuth 2.0 protocol.

The skeleton coming with the package needs to be configured based on the current project's setup, then the instance exposes a middleware, which may be mounted in certain routes [24]. After this has happened, the use of the framework may divide the routing configuration into fully accessible routes on the one hand and routes with limited access on the other hand (see Program 5.2).

From this point on, the client has to provide an access token in the *Authorization* HTTP header using the "Bearer" authentication scheme for gaining access [7, p. 5]. The service will deny further processing if either an inexisting or already expired access token was provided. In this case, the client has to exchange his/her correct client credentials for a new access token on the authorization endpoint prior accessing the desired endpoint again [5, p. 41] (see Sec. 5.1.1).

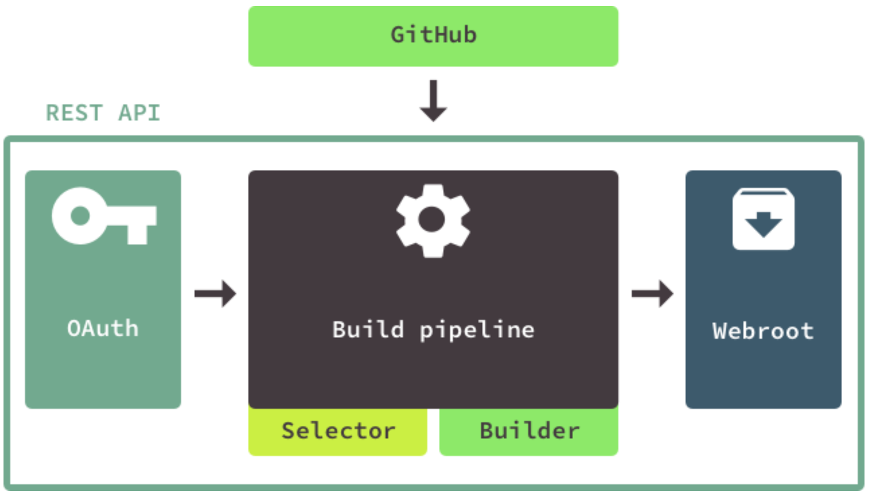


Figure 5.2: A graphic showing the first draft of the then proposed REST API cycle. At first the OAuth step should care for authentication, then the build pipeline should be initiated and orchestrate its services to interact with the GitHub API, select files to build and run the build task before saving a rendered version of the webroot, ready for deployment.

5.2.2 Build pipeline

After the HTTP- and authentication service was built as a user interaction possibility for the upcoming build pipeline realization, the full extent of the necessary API endpoints needed to be defined. Since the OAuth 2.0 framework was already implemented and the access to the GitHub API was prepared as includeable module, only the endpoints responsible for managing and triggering builds, needed to be reserved for the final build service development.

API definition

From the first draft of an API cycle (as seen in Fig. 5.2) to the final structure, a lot of details needed to be tidied up. This was mainly due to the complexity of the build pipeline itself, since one of the biggest challenges was to provide a real non-blocking event loop. By realizing such a non-blocking loop, the client receives an instant, intermediate response, instead of having to queue beforehand, and/or wait until the whole operation finishes. Furthermore, a recurring request for obtaining the build status was enabled using an own endpoint returning information from the database entry. The endpoints, which were implemented in favor of user projects, were the following:

- `POST /api/project` – Creates a project in the database, together with reference to its GitHub data.

- `POST /api/project/:owner/:repo/delete` – Deletes all references to the project from the database.
- `POST /api/project/:owner/:repo/build` – Trigger a new build cycle. Returns the reference to the database entry of its build log.
- `GET /api/project/:owner/:repo/status` – Get the status of the latest build (pending, failed, success).
- `GET /api/project/:owner/:repo/download` – Download the latest successful build as tar.gz archive. (e.g. for automated deployments)

Tasks

As already explained, the build pipeline is realized as a modular concept, consisting of different sub-tasks, bound together in a network of various dependencies to and from each other. Most of these modules are dynamically interlinked with API calls to GitHub, but also with partially strong data modifications of the respective responses.

One after the other, all necessary modules are loaded on request – to not confuse them by their actual functions, they are divided into three action levels: *engine*, *module* and *support*.

- Engine-specific tasks are immediately working for and on the compile actions (e.g. setting up the build pipeline, loading necessary modules, etc.),
- Module-specific tasks care for external assistance (e.g. installing modules, compressing rendered output, etc.) and
- Support-specific tasks care for engine-specific assistance (e.g. parsing configurations, creating new database entries, etc.).

5.3 Engine

The build pipeline engine basically wraps a common interface around the static site generator. Together with already mentioned supporting modules, it forms a nearly standalone ecosystem within a service, which happens to expose a REST API.

Other than pure HTTP services, the build engine not only has to cope with database queries – its main purpose is to handle file input/output management based on various configurations, ideally asynchronous and possibily even in parallel. Especially the latter may cause trouble at some point, because of JavaScript's single threaded model. Though Node.js may handle asynchronous operations well, it requires its event loop to continue running, when non-blocking operations (like input/output) are executed concurrently. This differs heavily from other programming languages, which are likely to create additional threads for such kind of tasks [19].

5.3.1 Asynchronous work

Possibly one of the most important requirements of the project is to work with asynchronous calls, as well as processing them as performant as possible. As already explained, the API depends on a significant variety of tasks for fetching data to create an instance of the build pipeline according to a certain configuration.

Most of them are realized using the JavaScript Promise API, where on the one hand subsequently nesting callback functions are avoided and on the other hand, various *then*-functions are not only getting chained to each other ("Promise chain"), but also returning a Promise themself [37]. This allows to keep an asynchronous flow in the same block – some operations even allow handling more than one asynchronous function concurrently within a Promise construct.

As a result, all API calls to GitHub are realized using Promises – as a matter of fact, the contained *then*-functions are acting as necessary backbone, as the returned data often needs to be altered or even merged with the response of a second, concurrent request. One example would be the comparison of the existing file tree versus the affected files by the commit range.

5.3.2 Child processes

For keeping the API responsive to requests while a build process is running, it makes sense to decouple the heavy rendering task into an own process ("Child process"). The child process will balance the work load, so that the rendering will happen in its own V8-process on an additional processor core [1, p. 335], only able to communicate to the host process via emitting events on its built-in communication channel [17]. The host process will reside in its initial thread and only receive a message, if the child process emits one or exited – therefore enough information will be distributed to keep the project's status in the database up to date.

There is a critical thing to consider though; since every child process gets equipped with an own memory and V8 instance, constantly allocating resources by spawning a large amount processes may lead to unexpected server crashes [17]. Virtual private servers (*VPS*) with a significant amount of RAM, as well as up to 20 processor cores and more will handle such heavy tasks of course better than local machines with often less than 4 cores, but also have to be managed well in terms of resource usage. Every child process is likely to occupy a minimum of 10 megabytes of RAM by default – though this amount surely increases. The final extent is based on the task it has to handle [44].

5.3.3 Storage

Because of the fact, that the project requires different stages of every repository to be stored for making use of caching, a significant amount of data has to be stored for quick access. To not loose track on the constantly growing extent, it possibly would be best to export them to long-lasting storage services like Amazon S3 buckets[12] – however, due to the many locally hosted services (first and foremost the REST API), the decision was made in favor of also storing repository data locally in the mean time during development.

The first kind of data to outsource would surely be the rendered archive, containing the webroot. As this needs to be accessed at all time, a downtime is simply not acceptable and a constant uptime cannot be guaranteed on a service like this, unless it is also run on multiple failsafe instances around the globe.

5.3.4 Realization

After the necessary technologies for providing a non-blocking event flow were defined, the build pipeline module set could be brought in shape. Together with the wrapping REST API and the predefined endpoints (see Sec. 5.2.2), the main entry point should be able to carry the heavy load of instantiating the static site generator, as well as fetching data from GitHub. Furthermore, it should act as control interface for managing the internal communication between Metalsmith and the API.

As seen in Fig. 5.3, much of the build pipeline's workflow actually happens without the user knowing about (grayish area). Basically the only thing happening before sending an HTTP response, is the parsing of the configuration file, as well as filtering files, affected by the commit range. If both succeeds, the database entry is created and the user may then query the API for getting to know the current status, while the main task possibly is still running in a forked child process. So, the REST API stays responsive the whole time during build, without causing any lack of performance.

Forking a child process

According to the Node.js documentation, the `child_process.fork()`-function is used specifically to spawn new Node.js processes. This means, that every child process creation happens without breaking the event loop of the parent process [17]. Therefore, this step is crucial before invoking any heavy task, which may result in blocking the API's responsiveness to handle additional HTTP requests.

Since the sub process is completely decoupled from the main task, it likely will not do its job in the same current working directory as its parent

[12]https://aws.amazon.com/de/s3/ – Amazon's S3 cloud storage service.

process. Whenever that happens, a *cwd* option may be set in the configuration object upon fork [17]. The Metalsmith instance requires this feature, as every repository it will be working on is nested in a certain sub folder in the project's file tree. Most of the Metalsmith plugins are moreover designed to only work in the actual CWD, unlike its API initially proposed (see line 5 in Program 2.1) [45].

However, equipping the child process with a cwd option was not possible at this point, as the child process also was responsible to install missing node modules. These would have been stored at the repository level – thus, they got deleted in a subsequent build cycle, as the repository folder gets cleared prior to every download action. By omitting the cwd flag, the node modules are getting installed at the level of the REST API to make them also accessible for other website projects. Finally, the CWD gets set to the building directory via `process.chdir()` after the Metalsmith instance was configured, but before the build function was triggered.

What is additionally happening in the child task, is the following; the respective repository archive is being fetched from the GitHub API – this happens in parallel to setting up the build pipeline:

1. Creating a Metalsmith instance by setting the global API options according to the global configuration object, provided by the parent process.
2. Filter required Metalsmith module names from the configuration object.
3. Sequentially invoke the modules to the Metalsmith instance.
   (a) Check if the current module is already available in the main project's node modules folder,
   (b) append it by default,
   (c) or catch any error by appending the module name into an array of yet missing node modules.
   (d) `npm install` the missing modules and repeat.
4. Return the completed Metalsmith instance to the child process.

Everything that is still missing, is waiting until both tasks (archive fetching, build pipeline setup) have succeeded, before the `build()`-function is ready to be triggered and the rendering process will be initiated. No matter if it fails or succeeds, the parent process receives a message containing necessary build information in any case.

Finishing the task

When the build pipeline was fully executed and nothing is left to do in the external JavaScript file, the process should normally exit automatically. This can be determined by listening to the *close* or *exit* events – moreover, it is also possible to *kill* it right from the parent process using `child.kill()` [18].

After the build pipeline finished, the CWD gets reset to its initial value, before the file tree of the rendered website is compressed into a tar.gz archive. This currently overwrites any previous archive of the same commit data – nevertheless, any previous version may be held available by appending the respective commit hash to the file name. From this point on, any future HTTP requests trying to fetch the website archive will get provided the latest version.

To conclude the build task, the database entry finally gets updated with information about the outcome of the rendering cycle.

5.4 Cache

Since the structure of the application flow is now in shape, the last major thing missing is the caching method, although already largely prepared using the GitHub API calls. As seen on Fig. 5.3, the first action after fetching the file tree and commit data, respectively, is to filter the unchanged files. But to really make caching work using diff, there are a few additional preconditions to be aware of; one of them is a functional folder structure at certain levels, another one is some crucial project-based information, which should be contained in the repository's configuration file.

5.4.1 Preconditions

Basically the most important place for gaining information about cacheable data is the structure of the respective repository itself. To know which files should be considered for caching presumes to know which files are critical for the project. It is very unlikely though, to perceive the full extent of this data.

A first approach however would be to getting to know the exact opposite; files which are unnecessary to cache. Metalsmith already cares for a first step of file separation, as the content file directory is announced in the *source*-property of its configuration, whereas everything on the same level of its CWD is treated as a supporting file (e.g., templates, partials, etc. . .). Since the relations between certain supporting files (or system files) and content files remain unclear – unless there is already some kind of network, which is able to sketch these relationships – the developer may announce ignoreable files right in the build pipeline configuration.

As the content files more or less are likely to not have any dependents, it can be safely assumed, that commits only featuring content files cause the build pipeline to only include these files and leave out every other content file (see Sec. 4.1.3). This marks the ideal scenario for this caching approach.

1. To make the approach work, the source repository has to be placed in the future working directory of Metalsmith, deleting any remains of past build cycles.
2. Then, the website has to be built to an intermediate file path, which remains until a full rebuild becomes necessary (e.g., a template file has changed).
3. Any further builds are either merged into this intermediate directory, or completely rebuilt after deleting the file tree.
4. Finally, this directory is holding the latest website version, so that a tar.gz archive is created out of it.

5.4.2 Filtering files

Because of the fact that both Metalsmith's source directory and the ignoreable files declared by the developer are known after parsing the repository's configuration file, it is easy to filter any possibly cacheable file out of the affected files by the given commit range. Without already having to deal with physical files, the respective file path is enough to compare it against the file path of the declared source directory.

If the list only contained content files, every other content file, which was unmentioned in the commit result, is being put on the ignore list, as it remained unchanged. This only works, if there was at least one successful build log stored in the database and the build engine is able to rely on an already present intermediate build folder. Otherwise, the result would be an incomplete webroot.

After the ignore list was populated, it is also added to Metalsmith's build instance, after the child process was forked. Therefore it stays in the global configuration object, which is handed from one task to the other. In the end, the ignore list is being stored in the database entry, so that the caching extent of every build may be looked up using the respective build log.

Ignore vs. delete

Initially, the ignore list was inexistent, as one of the first tasks of the build pipeline was to slim down the build directory in deleting all unchanged files. On the one hand, this supports the engine in precisely only rendering altered files, without having to deal with different globbing patterns[13], or an array of file paths.

However, there are a few downsides; the first is the inevitability of returning to files which have been deleted. If any future task needs to access a file deleted by the prior selection, it would have to download the whole archive again, which would be very costly in terms of time. Additionally, file

[13] https://github.com/isaacs/minimatch#usage – Minimatch matching library on GitHub.

input/output is always a computational expensive operation – physically deleting a file by overwriting it even more.

Therefore, populating an array with file path strings, without having to read from disk, seems the best possible solution for this ecosystem after all.

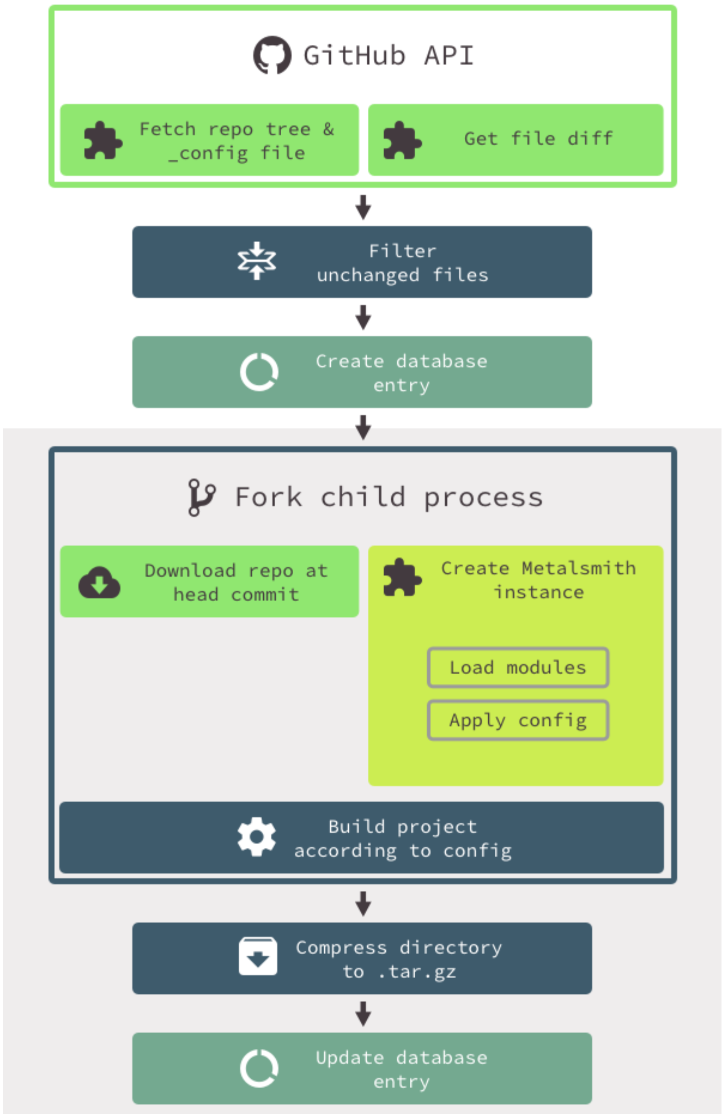

GitHub API
Fetch repo tree & _config file
Get file diff
Filter unchanged files
Create database entry
Fork child process
Download repo at head commit
Create Metalsmith instance
Load modules
Apply config
Build project according to config
Compress directory to .tar.gz
Update database entry

Figure 5.3: A graphic showing the main application flow of the build pipeline from top to bottom. At first, the *repo tree* and *file diff* are fetched from the GitHub API in parallel. After file filtering and creating a database entry, a child process is forked, which cares for executing the heavy tasks for building. After a successful build, the resulting files are compressed into a *tar.gz* archive, then the database entry is updated accordingly. Afterwards, the child process is terminating gracefully.

Chapter 6

Evaluation

To prove the project's usability, it has not only to be tested against a "real" setup; additionally, it also has to be tested against different settings. Since it was designed to be as unopinionated as possible, there are still challenges to face concerning the overall support of different third-party tools, like template engines, etc.

Nevertheless, a base repository for future building using the REST API is set up without much hassle. As the project features a generic Metalsmith instance for building and rendering the different website projects, a local installation setup might as well be useful prior to handing over the repository to the REST API. This might support fixing bugs, which are probably much easier detected, if the source code is at hand.

6.1 Minimal requirements

For even being able to build a repository successfully, it has to consist of a "valid" Metalsmith project. This means, that a few requirements have to be met, such as the following:

- A folder structure, which consists of at least a source folder inside the project root,
- a configuration file, either in YAML or JSON format and named *_config.**,
- and finally being hosted on GitHub as public repository.
- Furthermore, it must not rely on any other build tools (e.g. *Gulp*[1], *Webpack*[2], etc...), only Metalsmith is supported at this time.

[1] http://gulpjs.com – Website of Gulp.js

[2] https://webpack.js.org – Website of Webpack.

Program 6.1: _config.yml – a sample configuration file, containing some global configuration data, as well as a few Metalsmith plugin definitions.

```
global:
  source: '_src'            # source folder
  destination: '_dist'      # distribution folder
  metadata:
    # define metadata

exclude:                    # list of ignoreable files

# Metalsmith plugin configuration starts here

drafts: true                # metalsmith-drafts plugin
markdown: true              # metalsmith-markdown plugin
excerpts: true              # metalsmith-excerpts plugin

collections:                # metalsmith-collections plugin
  blog: posts/*.md
  repos: repos/*.md

# To be continued (...)
```

6.1.1 Configuration file

The configuration file is probably the most critical part in the repository's contents, as it is the only source for the build pipeline to obtain the setup instructions from. Since the Metalsmith CLI is able to render a project based on a single JSON configuration file and the API setup doesn't really differ, the format of the configuration needed by the REST API is nearly identical. Therefore, the configuration for a local Metalsmith installation and the one used for the project's build pipeline are very well interchangeable (see Program 6.1).

Since the REST API is able to parse both YAML and JSON notations, it is up to the developer to choose what fits his/her needs best. Since Metalsmith only understands JavaScript, any YAML configuration is parsed to JSON by the API, prior to forking the child process. This makes sense in a way, as the build setting information is getting included in the general options object, which is handed over from the REST API to the build pipeline, where parts of it get stored in the database together with the build log.

6.1.2 Local testing

Having a local Metalsmith install at stake may not only support the developer in finding and fixing bugs, it also helps to constantly pursue a clean build setup. The remote build pipeline neither is configured to inform about any installed modules, nor is it able to independently draw any conclusions of the provided configuration file. The only way to communicate with any responsible developer, is to send E-Mails containing status messages, or to respond build log information from the database upon request.

Although caching is not available when testing locally, it is often the only way to fix the build tree in a way, that Metalsmith is able to produce a successful outcome again. The reason behind that is the fact, that developers often try to fix a bug using subsequent small code changes – this requires multiple rebuilds to check if the effort succeeded. However, it is also possible to patch the code base by adding one commit after another and analyze the messages of the build log entries.

6.2 Comparison

When trying to compare the project's build pipeline to standalone static site generators like Jekyll or Metalsmith, it has to be stated, that neither one of those requires a git repository, nor any sort of authorization (besides during their installation process possibly). Furthermore, Jekyll also provides a command line argument for setting up a base project (see Sec. 2.1.2), so that hardly any time is lost before a content author actually is being able to start writing.

As this project initially was designed to support porting Jekyll projects to Metalsmith, it already requires a basic structure for being able to work with. However, when starting from scratch, the probably best advice is to set up a local project, which makes use of the Metalsmith CLI and then start porting the configuration to fit the REST APIs standards.

6.2.1 Jekyll

A Jekyll project, as already explained in Sec. 2.1.2, is entirely written in Ruby and sets up on the Liquid templating engine. As a configuration file, it requires a YAML file called *_config.yml* in the project root and supports parsing data from YAML, JSON and CSV files to usable site variables out of the box [3, p. 76]. A functional Jekyll project also has to be equipped with Liquid templates in the *_layouts* folder, together with a few more directories holding different contents resulting in various recycleable parts during the build process. As of Jekyll 3.2, most of these parts have gotten outsourced in different Ruby gems, thus being hidden to the public and making it harder to port them to other generator applications like Metalsmith [42].

Jekyll source repository

Parallel to the development of this project, the Jekyll docs sources[3] were ported the best possible for setting a benchmark for the usability of the REST API. To make this happen, a few major changes needed to be made in order to get a positive build result:

- The root folder was put in the source folder, but *_data*, *_layouts* and *_includes* were left out. Additionally, all asset-containing folders were moved into the *_public* directory.
- Necessary plugins were installed and configured for correctly setting the render flow:
  - metalsmith-date-in-filename,
  - metalsmith-sass,
  - metalsmith-collections,
  - metalsmith-permalinks,
  - and a few more.
- Due to the limitations of *TinyLiquid*[4], especially file paths in include and extend statements had to be adjusted.
- Some functionality still does not work without further engineering, for example the YAML files in the `_data` directory, or gem-dependent tasks like generating the sitemap or managing redirects.

To conclude; it is possible to successfully render a Jekyll project using Metalsmith, although there are a lot of adjustments necessary beforehand, not to speak of the deficits of the TinyLiquid package. Therefore, the expected behaviour of the outcome is likely to differ heavily from the reality. The test repository for this evaluation was obviously uploaded on GitHub[5] and may be tested locally using the Metalsmith CLI.

6.2.2 Metalsmith

Since Metalsmith was chosen for use as static site generator within the REST API (see Sec. 4.3.1), it should be used as the foundation framework for any website project, which should be built using the REST API in the future. Due to the fact that Metalsmith is yet another npm module built for Node.js, it also offers to act as part of any available build tool, such as Gulp, Webpack or else – however, this is not supported (see Sec. 6.1).

[3] https://github.com/jekyll/jekyll/tree/master/docs – Jekyll docs section in the Jekyll repository on GitHub.

[4] https://github.com/leizongmin/tinyliquid – TinyLiquid repository on GitHub.

[5] https://github.com/vorchdorfmedia/jekyll-docs – Test repository for porting a Jekyll project to Metalsmith.

The main reason behind this limitation is, that automatically detecting a different or additional build setup is error-prone and may easily slow down the render process. Furthermore, Metalsmith offers a feature-rich API and the possibility of writing plugins[6] to fit the developer's needs, thus making nearly any additional build tool obsolete. All that is left to do, is to publish any written plugin publicly available to npm and append it to the repository's configuration file for future use in the build pipeline. As a result, other developers may also download and/or enhance the plugin to keep it up to date.

6.3 REST API

The REST API was built and designed as remote-acting, standalone web application, running on Node's servers-side JavaScript engine. Apart from specifically reserved firewall ports, the project is only dependent on a Node.js installation, every other dependency may be installed using npm, regardless on the used operating system.

By extending any desired Node.js Dockerfile[7], it may be even run as autonomous Docker container and therefore multiplied for better load balancing, based on the current HTTP load extent.

6.3.1 Load testing

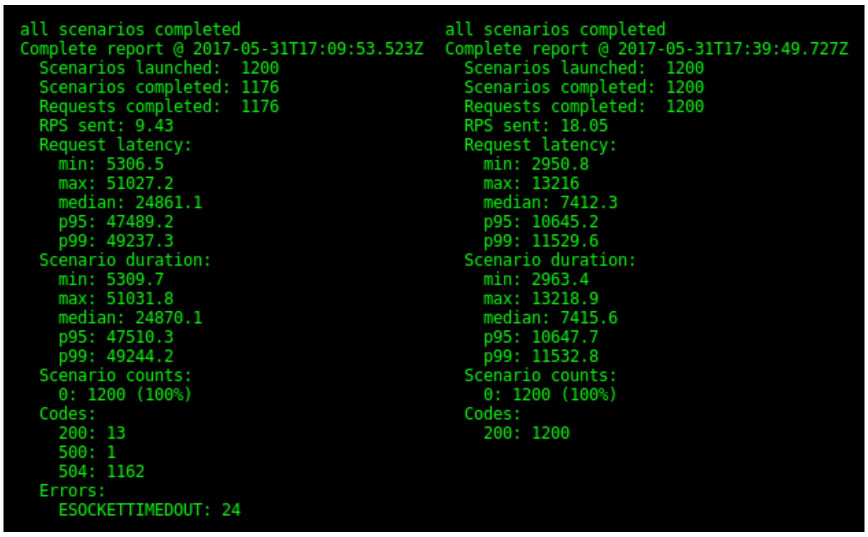
To evaluate the basic stability while handling multiple concurrent requests, the API was put under a high load test using Artillery[8] (see Figs. 6.1 and 6.2). Without any load balancing, nor any other high load supporting tool, 1200 POST requests triggered the build pipeline for a total duration of 60 seconds. As explained in the graphic's caption, the penetration test showed a success rate of roughly 1%, a reasonable minimal response time of about 5 seconds but a terrible maximum response time of 51 seconds. Of course, this data may not be interpreted as a successful result in the first place, but it has to be stated, that the test was run on the endpoint causing the heaviest task in the system and most failure responses were effected by GitHub blocking most requests due to their rate abuse checking system.

The same test running on a much lighter task is showing a different picture; 1200 GET requests for receiving information about the latest build cycle had a response rate of 100%. The response time ranged from 3 to 13 seconds, which is again a sign to not let a single application handle such an amount of requests without load balancing beforehand. In the end, it is safe to say, that the REST API may handle a reasonable amount of

[6]http://www.metalsmith.io/#writing-a-plugin – "Writing a plugin" section in Metalsmith's documentation.

[7]https://hub.docker.com/_/node/ – Official repository for Node.js on Docker Hub.

[8]https://artillery.io – Website of Artillery, a load testing toolkit.



```
all scenarios completed                   all scenarios completed
Complete report @ 2017-05-31T17:09:53.523Z  Complete report @ 2017-05-31T17:39:49.727Z
  Scenarios launched:  1200                 Scenarios launched:  1200
  Scenarios completed: 1176                 Scenarios completed: 1200
  Requests completed:  1176                 Requests completed:  1200
  RPS sent: 9.43                            RPS sent: 18.05
  Request latency:                          Request latency:
    min: 5306.5                               min: 2950.8
    max: 51027.2                              max: 13216
    median: 24861.1                           median: 7412.3
    p95: 47489.2                              p95: 10645.2
    p99: 49237.3                              p99: 11529.6
  Scenario duration:                        Scenario duration:
    min: 5309.7                               min: 2963.4
    max: 51031.8                              max: 13218.9
    median: 24870.1                           median: 7415.6
    p95: 47510.3                              p95: 10647.7
    p99: 49244.2                              p99: 11532.8
  Scenario counts:                          Scenario counts:
    0: 1200 (100%)                            0: 1200 (100%)
  Codes:                                    Codes:
    200: 13                                   200: 1200
    500: 1
    504: 1162
  Errors:
    ESOCKETTIMEDOUT: 24
```

Figure 6.1: Screenshots of two command line outputs showing the results of the REST API being put under high HTTP load. During 60 seconds, the API had to face *1200 requests* of 20 virtual clients created by Artillery. The test on the left was defined to include a single POST request triggering a new build cycle (with a full rebuild option) every time the API accepted a new connection. The results show the following: *24* could not be handled at all, *1162* resulted in gateway timeouts (GitHub blocks), but *13* were handled successfully and returned with a *200 OK* HTTP status code.

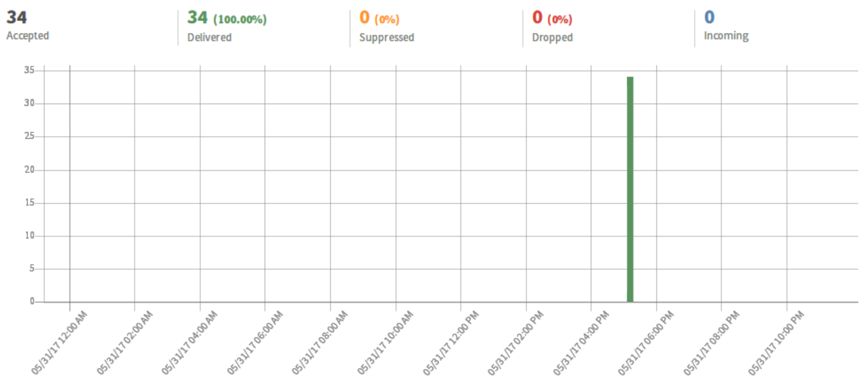


Figure 6.2: A screenshot showing the extent of the previous load-test (see Fig. 6.1) in the Mailgun dashboard. Obviously, the build pipeline was triggered *34* times, leading to the same number of E-Mails being sent. Out of this 34 E-Mails, *8* showed a success message, whereas the others mostly failed due to other concurrent requests deleting the CWD as a preparation step prior to downloading the repository archive.

requests quite well on its own (e.g. requests triggered by GitHub webhooks), but consisting of multiple instances may be the best option for handling a significant amount of requests every once in a while.

6.4 Caching

In terms of caching, the build pipeline is best evaluated when assuming the best possible, as well as the worst possible case. As already explained in Sec. 4.1.3, such scenarios would be on the one hand a commit only containing content changes (e.g. new or modified blog posts) and on the other hand a commit containing a modification of the default template. Since the default template is very likely to act as a dependency of nearly all content files, a full rebuild is inevitable.

6.4.1 Initial build

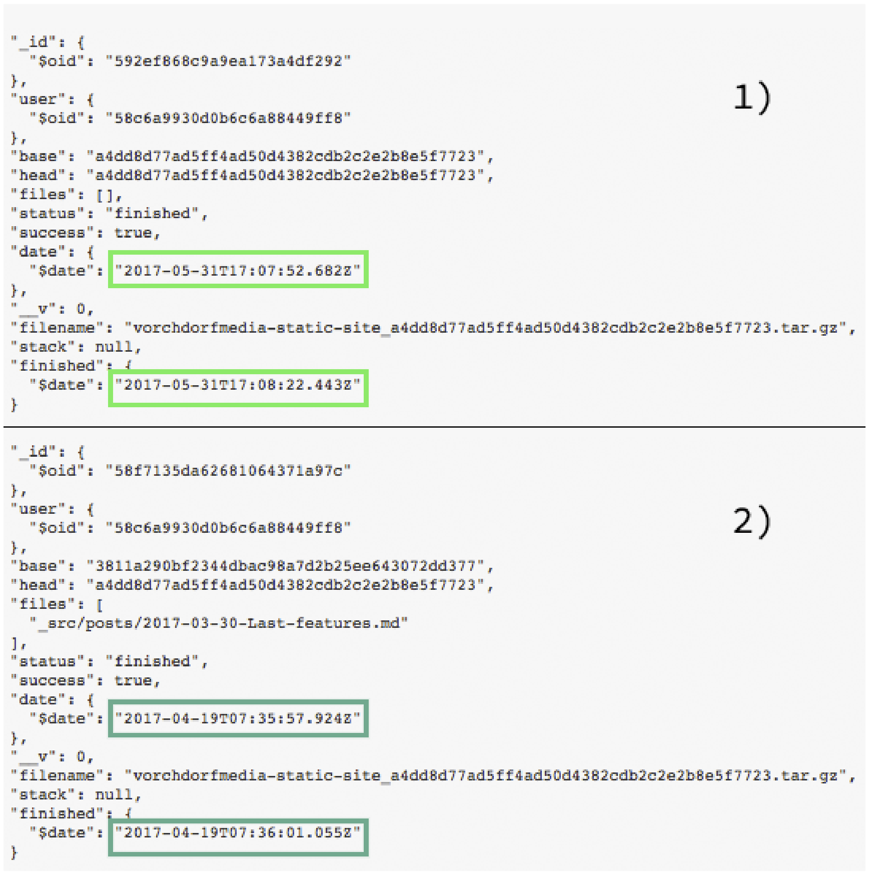
An initial build is necessary every time a repository was registered using the REST API, or the repository's previous build attempts constantly failed and no successful outcome was produced yet. Not only caring for the required folder structure, a successful build cycle also provides information for a subsequent rendering process by storing its head commit hash value in the build log on the database. Any following build attempt is able to forge upon the last successful build files.

Therefore it is a good advice to have a successful initial build ready as soon as possible, as future build cycles profit from an early render history and a best possible caching structure. By omitting an early registration to the REST API, any initial build cycle in the future will last a significant amount of time longer, due to continuous progression which is not able to make use of any cached file structure.

6.4.2 Caching strategy

Since caching works most effectively if subsequent commits only contain content changes, the commit culture should be focused towards a content-only development, to make use of a long-lasting series of performant build cycles. Normally, this would be the standard for steady sites containing a significant amount of various information (e.g., FAQ-, support-, or documentation-sites), where constantly changing design decisions are not likely to play an important role. Concerning the need of a templating- or design change, the probably best advice is to collect commits containing such system files for as long as possible before actually merging them into the main branch and causing a longer lasting rebuild task, resulting in a major redesign.

Blogs are also likely to follow this kind of commit pattern, as usually a theme is set once, before subsequent blog posts are published. Using this type



```
"_id": {
  "$oid": "592ef868c9a9ea173a4df292"
},
"user": {
  "$oid": "58c6a9930d0b6c6a88449ff8"
},
"base": "a4dd8d77ad5ff4ad50d4382cdb2c2e2b8e5f7723",
"head": "a4dd8d77ad5ff4ad50d4382cdb2c2e2b8e5f7723",
"files": [],
"status": "finished",
"success": true,
"date": {
  "$date": "2017-05-31T17:07:52.682Z"
},
"__v": 0,
"filename": "vorchdorfmedia-static-site_a4dd8d77ad5ff4ad50d4382cdb2c2e2b8e5f7723.tar.gz",
"stack": null,
"finished": {
  "$date": "2017-05-31T17:08:22.443Z"
}
```

```
"_id": {
  "$oid": "58f7135da62681064371a97c"
},
"user": {
  "$oid": "58c6a9930d0b6c6a88449ff8"
},
"base": "3811a290bf2344dbac98a7d2b25ee643072dd377",
"head": "a4dd8d77ad5ff4ad50d4382cdb2c2e2b8e5f7723",
"files": [
  "_src/posts/2017-03-30-Last-features.md"
],
"status": "finished",
"success": true,
"date": {
  "$date": "2017-04-19T07:35:57.924Z"
},
"__v": 0,
"filename": "vorchdorfmedia-static-site_a4dd8d77ad5ff4ad50d4382cdb2c2e2b8e5f7723.tar.gz",
"stack": null,
"finished": {
  "$date": "2017-04-19T07:36:01.055Z"
}
```

Figure 6.3: Two screenshots of build log entries on the database showing the extent of ideal caching. Example 1) shows a forced rebuild during the load test (see Fig. 6.1). Obviously there already happened some successful rendering cycles in the past, as *base* and *head* show the same commit hashes. The build lasted roughly 30 seconds and resulted in a successful archive file. Example 2) shows an earlier build, which made use of an available cache. The entry listed in the "files" array was the only file, which was rendered and added to the existing file structure. Therefore the build cycle lasted only 4 seconds.

of scenario, the extent of ideal caching may be seen on Fig. 6.3. A rebuild, as well as an initial build at a later time is a lot more time consuming for resulting in a sane file structure, than a selectively rendered build result, which is able to get merged into an existing website root. Thereby it is not important to check for any existing file structure, if any previously failed build attempt forced a rebuild due to its commit history, as the current build is always able to rely on the rendering result of the last successfully logged attempt.

6.5 Outlook

Since this project has to be merely seen as a proof of concept, a few open points still remain. One of them would be the possibility of including it into a project workflow as a continuous integration service. By acting as an interface between GitHub and any deployment service, which is able to decompress a tar.gz archive, it would be the missing link for an automated update process to the web hosting service.

However, for extending the precision and usability of the REST API, some additional enhancements are necessary, where each of them is likely to form the extent of a project of its own.

6.5.1 User experience

To not only offer access to the REST API, but also a certain level of project management without relying on pure HTTP requests, it needs a graphical user interface. Furthermore, the current setup consists of a hard-coded access token for accessing certain data on GitHub – this is not feasible for a multi-user system.

Graphical user interface

Through providing a graphical user interface (*GUI*), a repository owner may not only have the possibility of a quick overview of his/her project, also managing a repository by adding/removing contributors authorized for triggering build cycles, as well as adjusting settings for any possible deployment strategy surely leverages the overall productivity. Moreover, build messages may be examined much easier and clearer.

Because of the REST API already being present, such a GUI may easily be built on top using different frontend libraries based on JavaScript. In the end, the API below will have to be extended for a few endpoints more. This not only enhances the overall functionality of the GUI, but also enables to provide the same functions to low level HTTP requests.

GitHub authorization

For making it possible to interact with repository data of any logged in user, he/she has to grant access somehow [28]. Normally this is done via a dialog in the browser, then the REST API receives an access token for making future requests without permanently asking the user for authorization, similar to the implemented OAuth 2.0 framework (see Sec. 5.1.1).

6.5.2 Cache improvement

Caching currently works by only comparing file paths, thus differentiating between system- and content files. Whereas this works fine for the vast majority of used repositories to a reasonable extent, the performance, as well as the reliability also depend on future improvements concerning the selective rendering algorithm.

Frontmatter parsing

The first approach for improving the overall caching performance would be an analysis of the frontmatter. As it is written in YAML and delimited using three dashes on the top and on the bottom [3, p. 77], it should be very well parseable. Metalsmith already does that in order to provide different plugins with additional per-post metadata. However, since the list of cacheable files already has to be declared prior to creating the Metalsmith instance, making use of data parsed by Metalsmith would conflict the actual caching process.

This is aggravated by the fact that frontmatter does not always follow a fixed schema, so that every repository owner would have to introduce certain keys into a kind of tracking system on the REST API for supporting the caching algroithm with additional data.

Machine learning

A much more performant approach would be the constant tracking of processed files and thereby creating a virtual network of dependencies within a respository. Based on the individual data of every website project, the conjunctions to various dependent files of a system file (e.g., a template) could be revealed. A precise detection of cacheable files could be refined more and more, resulting in the most performant build process possible.

Chapter 7

Conclusion

The main interest for static site generators evolved during my work at a digital performance monitoring company based in Linz, Upper Austria. I was impressed by the simplicity of generating HTML content without having to construct an extensive interface before even getting to the point of actually creating content. One of the major drawbacks although was the idle time I had to face during a rendering cycle.

One of the projects I used to work with was initially based on Jekyll, but with some strong customizations added to the build pipeline setup. Consisting of a reasonable amount of content files, a build cycle sometimes lasted more than 20 minutes – mostly due to heavy tasks, such as picture resizing, etc.

Therefore, this project was originally designed only as supporting tool for local development, but it quickly grew out of hand as I figured out, that this type of development is facing too many limitations. One of the very first approaches was to use the GitHub API, as I had already gained some experience in using it while working on a few projects in the past. Concerning the amount of information needed, and first and foremost where to actually fetch it, GitHub is the best possible tool to use, unless a strictly local solution is preferred. Soon after, it was clear to build something, which is able to act remotely and as automated as possible.

However, the major premise for this project was to provide an unopinionated tool for rendering a website with a caching solution included. Although this may sound fairly understandable in the first place, it soon turned out, that this mixture is also going to be the biggest challenge in finding a suitable way of solving this problem statement.

As a conclusion, I can now say, the most interesting part about my research was not only to find ways to overcome those local performance issues during rebuilds, but also trying to leverage the common workflow in moving as many local tasks to a remote workspace as possible. This should support content authors and developers in focusing on their core jobs by taking unnecessary responsibilities off their hands.

Soon after my initial project setup, I was already forced to balance the importance of the core principles and therefore I had to compromise over some of them. First, the project is not as unopinionatedly usable as originally planned, as by all forms of customizability, Metalsmith needs at least a core structure of parameters in its configuration file. To not interfere with the standard configuration file, I designed an adjusted format, by also allowing it to be written in YAML. This especially should support developers switching from Jekyll.

Second, by providing an automated remote workspace, the question of long-term storage has to be reconsidered, as the tar.gz archive currently only gets written to the same file structure the REST API lies in. This saves time by always having the latest version at hand, however, using an external storage like Amazon S3[1], the file distribution would scale significantly better (especially when using multiple instances in Docker containers, etc. . .) and is also a lot cheaper in the long run.

Third, the caching algorithm currently appears very basic, as the variety of future repositories cannot be correctly evaluated by now. Therefore it needs some kind of machine learning, which is able to virtualize a dependency graph throughout a single repository (see Sec. 6.5.2). However, this would cover the extent of a project on its own.

Lastly it can be said, that the overall performance of the REST API handling different smaller demo projects during my tests was quite the same, as both a repository containing 24 files and a repository containing roughly over 300 files needed little over 30 seconds for an initial build. A much more interesting examination would be testing the REST API in a productive ecosystem, as the "real" needs of a comparable development team would be revealed much faster and much more precise. Depending on these informations, future development could be led towards fields which really matter.

To sum everything up, the outcome is quite the initially expected extent; a proof of concept, which is able to produce a usable website on the one hand, but on the other hand should as well demonstrate the difficulties of providing and running an unopinionated, semi-automated system, which should be able to work with highly diverse source repositories together with as many requests as possible at the same time. This project shows that all of this is possible, if the end user is able to compromise on his expectations and development is pushed further towards a more user-friendly surrounding.

[1]https://aws.amazon.com/s3/?hp=tile&so-exp=below – Amazon S3 website.

Appendix A

Contents of the CD-ROM

Format: CD-ROM, Single Layer, ISO9660-Format

A.1 PDF files

Path: /

S1510629021_Zarhuber_Thesis.pdf	Master's Thesis with instructions (entire document)

Path: /online

Hexo 3.2.0-beta.2.pdf	[13]
Hexo debut.pdf	[14]
Hexo Documentation - Setup.pdf	[15]
JSLint Documentation.pdf	[16]
Node.js Documentation.pdf	[17][18]
Overview of Blocking vs Non-Blocking.pdf	[19]
git-diff.pdf	[20]
git-revert.pdf	[21]
Introducing Markdown.pdf	[22]
Markdown.pdf	[23]
OAuth2orize.pdf	[24]
Routing.pdf	[25]
Using middleware.pdf	[26]
About pull requests.pdf	[27]
Basics of Authentication.pdf	[28]
Mastering Markdown.pdf	[29]
Merging a pull request.pdf	[30]

Resolving a merge conflict using the command line.pdf	[31]
Webhooks.pdf	[32]
Write Scripts for the mongo Shell.pdf	[33]
Building Technical Documentation with Metalsmith.pdf	[34]
Node based static site generators.pdf	[35]
JavaScript reference - Arrow functions.pdf	[36]
Using promises.pdf	[37]
Git Projects.pdf	[38]
Blogging Like a Hacker.pdf	[39]
GitHub Pages.pdf	[40]
How I Turned Down USD 300,000.pdf	[41]
Directory structure.pdf	[42]
Usage of server-side programming languages for websites.pdf	[43]
Setting up a Node.js Cluster.pdf	[44]
Metalsmith Repository.pdf	[45]
Building Building Blocks	[46]

Path: /source

Technical Documentation.pdf	LaTeX version of the project's README.md

A.2 Source code

Path: /source

v1.0.1.zip	Source code of the project

A.3 Graphics

Path: /images

*.sketch	Source files
*.svg	Vector graphics
*.png	Rendered images & Screenshots

References

Literature

[1] Mike Cantelon et al. *Node.Js in Action*. Greenwich, CT, USA: Manning Publications Company, Oct. 2013 (cit. on pp. 37, 40, 53).

[2] Douglas Crockford. *JavaScript: The Good Parts*. Sebastopol, CA, USA: O'Reilly Media, May 2008 (cit. on p. 13).

[3] Vikram Dhillon. *Creating Blogs with Jekyll. Build elegant and minimalistic static blogs*. Orlando, FL: Apress, 2016 (cit. on pp. 1, 4–6, 9, 10, 15, 16, 19, 22, 28, 62, 69).

[4] Christopher Gandrud. "GitHub: A tool for social data set development and verification in the cloud". *The Political Methodologist* 20.2 (May 2013), pp. 7–16 (cit. on p. 16).

[5] Dick Hardt. *The OAuth 2.0 Authorization Framework*. Ed. by Dick Hardt. Request for Comments. Oct. 2012. URL: https://rfc-editor.org/rfc/rfc6749.txt (cit. on pp. 41, 50).

[6] James Wayne Hunt and Malcolm Douglas McIlroy. *An algorithm for differential file comparison*. Tech. rep. 41. Murray Hill, NJ, USA: Bell Laboratories, June 1976. URL: https://nanohub.org/infrastructure/rappture/export/3582/trunk/gui/src/diff.pdf (cit. on pp. 22, 23).

[7] Michael Jones and Dick Hardt. *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. Ed. by Michael Jones and Dick Hardt. Request for Comments. Oct. 2012. URL: https://rfc-editor.org/rfc/rfc6750.txt (cit. on p. 50).

[8] Sean Leonard. *Guidance on Markdown: Design Philosophies, Stability Strategies, and Select Registrations*. Ed. by Sean Leonard. Request for Comments. Mar. 2016. URL: https://rfc-editor.org/rfc/rfc7764.txt (cit. on p. 16).

[9] J. Loeliger and M. McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. Sebastopol, CA, USA: O'Reilly Media, Aug. 2012 (cit. on pp. 18–22, 25, 29, 45).

[10] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files with GNU diff and patch.* Bristol, UK: Network Theory Ltd., June 2003 (cit. on pp. 22, 23, 26).

[11] Webb Miller and Eugene W. Myers. “A file comparison program”. *Software: Practice and Experience* 15.11 (1985), pp. 1025–1040 (cit. on p. 23).

[12] Terence John Parr. “Enforcing Strict Model-view Separation in Template Engines”. In: *Proceedings of the 13th International Conference on World Wide Web.* WWW ’04. New York, NY, USA: ACM, 2004, pp. 224–233. URL: http://doi.acm.org/10.1145/988672.988703 (cit. on p. 17).

Online sources

[13] Tommy Chen. *Hexo 3.2.0-beta.2.* Dec. 2015. URL: https://github.com/hexojs/hexo/releases/tag/3.2.0-beta.2 (cit. on pp. 9, 72).

[14] Tommy Chen. *Hexo debut!* chinese. Oct. 2012. URL: https://zespia.tw/blog/2012/10/11/hexo-debut/ (cit. on pp. 7, 8, 72).

[15] Tommy Chen. *Hexo Documentation: Setup.* URL: https://hexo.io/docs/setup.html (cit. on pp. 9, 72).

[16] Douglas Crockford. *JSLint Documentation.* URL: http://www.jslint.com/help.html (cit. on pp. 13, 72).

[17] Node.js Foundation. *Node.js Documentation – Asynchronous Process Creation.* URL: https://nodejs.org/api/child_process.html#child_process_child_process_fork_modulepath_args_options (cit. on pp. 53–55, 72).

[18] Node.js Foundation. *Node.js Documentation – child.kill([signal]).* URL: https://nodejs.org/api/child_process.html#child_process_child_kill_signal (cit. on pp. 56, 72).

[19] Node.js Foundation. *Overview of Blocking vs Non-Blocking.* URL: https://nodejs.org/en/docs/guides/blocking-vs-non-blocking/ (cit. on pp. 52, 72).

[20] Git. *git-diff - Show changes between commits, commit and working tree, etc.* URL: https://git-scm.com/docs/git-diff (cit. on pp. 23, 72).

[21] Git. *git-revert - Revert some existing commits.* URL: https://git-scm.com/docs/git-revert (cit. on pp. 44, 72).

[22] John Gruber. *Introducing Markdown.* Mar. 2004. URL: http://daringfireball.net/2004/03/introducing_markdown (cit. on pp. 16, 72).

[23] John Gruber. *Markdown.* 2004. URL: http://daringfireball.net/projects/markdown/ (cit. on pp. 16, 72).

[24] Jared Hanson. *OAuth2orize.* URL: https://github.com/jaredhanson/oauth2orize (cit. on pp. 50, 72).

[25] StrongLoop, IBM and other express.js contributors. *Routing.* URL: http://expressjs.com/en/guide/routing.html (cit. on pp. 49, 72).

[26] StrongLoop, IBM and other express.js contributors. *Using middleware.* URL: http://expressjs.com/en/guide/using-middleware.html (cit. on pp. 49, 72).

[27] GitHub Inc. *About pull requests.* URL: https://help.github.com/articles/about-pull-requests/ (cit. on pp. 29, 72).

[28] GitHub Inc. *Basics of Authentication.* URL: https://developer.github.com/v3/guides/basics-of-authentication/ (cit. on pp. 68, 72).

[29] GitHub Inc. *Mastering Markdown.* 2014. URL: https://guides.github.com/features/mastering-markdown/ (cit. on pp. 16, 72).

[30] GitHub Inc. *Merging a pull request.* URL: https://help.github.com/articles/merging-a-pull-request/ (cit. on pp. 29, 72).

[31] GitHub Inc. *Resolving a merge conflict using the command line.* URL: https://help.github.com/articles/resolving-a-merge-conflict-using-the-command-line/ (cit. on pp. 25, 73).

[32] GitHub Inc. *Webhooks.* URL: https://developer.github.com/webhooks/ (cit. on pp. 33, 73).

[33] MongoDB, Inc. *Write Scripts for the mongo Shell.* URL: https://docs.mongodb.com/manual/tutorial/write-scripts-for-the-mongo-shell/ (cit. on pp. 41, 73).

[34] Andy Jiang. *Building Technical Documentation with Metalsmith.* Oct. 2015. URL: https://segment.com/blog/building-technical-documentation-with-metalsmith/ (cit. on pp. 11, 14, 73).

[35] Boris Mann. *Node based static site generators.* June 2012. URL: http://blog.bmannconsulting.com/node-static-site-generators (cit. on pp. 8, 73).

[36] Mozilla Developer Network and individual contributors. *JavaScript reference – Arrow functions.* URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions (cit. on pp. 13, 73).

[37] Mozilla Developer Network and individual contributors. *Using promises.* URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises (cit. on pp. 13, 53, 73).

[38] Pancake.io. *Git Projects.* URL: http://docs.pancake.io/docs/git/basics (cit. on pp. 34, 73).

[39] Tom Preston-Werner. *Blogging Like a Hacker*. Nov. 2008. URL: http://tom.preston-werner.com/2008/11/17/blogging-like-a-hacker.html (cit. on pp. 5, 6, 73).

[40] Tom Preston-Werner. *GitHub Pages*. Dec. 2008. URL: https://github.com/blog/272-github-pages (cit. on pp. 6, 73).

[41] Tom Preston-Werner. *How I Turned Down $300,000 from Microsoft to go Full-Time on GitHub*. Oct. 2008. URL: http://tom.preston-werner.com/2008/10/18/how-i-turned-down-300k.html (cit. on pp. 6, 20, 73).

[42] Tom Preston-Werner and Jekyll contributors. *Directory structure*. URL: http://jekyllrb.com/docs/structure/ (cit. on pp. 62, 73).

[43] Q-Success. *Historical yearly trends in the usage of server-side programming languages for websites*. Apr. 2017. URL: https://w3techs.com/technologies/history_overview/programming_language/ms/y (cit. on pp. 4, 73).

[44] Scott Robinson. *Setting up a Node.js Cluster*. Jan. 2016. URL: http://stackabuse.com/setting-up-a-node-js-cluster/ (cit. on pp. 53, 73).

[45] Segment. *Metalsmith Repository*. URL: https://github.com/segmentio/metalsmith/blob/master/Readme.md (cit. on pp. 10, 55, 73).

[46] Chris Sperandio. *Building Building Blocks*. Feb. 2015. URL: https://segment.com/blog/building-building-blocks/#metalsmith (cit. on pp. 9, 10, 73).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —

Breite = 100 mm
Höhe = 50 mm

— Diese Seite nach dem Druck entfernen! —