Technologisches Gewerbemuseum
Höhere Technische Lehranstalt für Informationstechnologie
Schwerpunkt Medientechnik, Systemtechnik

Diploma Thesis

TGM Rebranding

Development of the TGM Webseite

Project management and communication process
Raphael Schlager 5AHIT

The page's stack and structural concept
Maximilian Mairinger 5AHIT

System interfaces and blog structure
Georg Felber 5AHIT

User experience and interface design
Farid Goldmann 5AHIT

Content management system and system engineering
Moritz Meier 5AHIT

Supervisor : Gottfried Koppensteiner, Christoph Brein, Christoph Roschger
Performed in the school year 2020/21

Submission note:
21.04.2021 Approved:

Statutory declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Location, Date	Raphael Schlager
Location, Date	Maximilian Mairinger
Location, Date	Georg Felber
Location, Date	Farid Goldmann
Location, Date	Moritz Meier

Abstract

One of the most essential parts of any organisation's outward representation is its website. As a school, attended by no less than 2500 students, the TGM marks no exception. The project TGM Rebranding is a multi-team effort aimed at developing a new website for the TGM, as well as an information-platform for internal use and creating related multimedia content. The focus of this diploma thesis lies on the development of the website itself. Particularly the main goal can be described as a progression from the formerly used website in terms of design, usability, security, speed and content. The thesis reviews both the development process itself, as well as describing the accompanying research of technologies and management methodologies.

Die eigene Website ist einer der relevantesten Aspekte der Repräsentation einer Organisation. Als Schule, die von mehr als 2500 Schülern besucht wird, bildet das TGM keine Ausnahme. Das Projekt TGM Rebranding ist eine teamübergreifende Arbeit, die zum Ziel hat, eine neue Website für das TGM zu entwickeln, weiters eine Informationsplattform für den internen Gebrauch zu erstellen und entsprechende multimediale Inhalte aufzunehmen. Der Fokus dieser Diplomarbeit liegt auf der Entwicklung der Website selbst. Das Hauptziel kann als Weiterentwicklung der bisher genutzten Website in Bezug auf Design, Usability, Sicherheit, Geschwindigkeit und Inhalt beschrieben werden. Die Arbeit gibt sowohl einen Überblick über den Entwicklungsprozess selbst, als auch über die begleitende Recherche von Technologien und Management-Methoden.

Contents

0.1 Performance optimization

To avoid having to choose from a subset of platforms when developing Graphical user interfaces (GUIs), corporations and independent developers have always been striving to build an abstraction layer over these. One big player in the aforementioned realm is Oracle, which's subsidiary Sun Microsystems released the first version of their multi-paradigm programming language Java in 1995 [86, p. 348]. Java aims to promote a 'one fits all' approach, with its ability to compile executables to all modern platforms. Using cross-platform environments enables businesses to simplify their human resources efforts drastically, hence making them economically immensely attractive [86, p. 39] [68, p. 1]. This historically led to the wide adoption of Java in enterprise software [56][98] and, more recent, to the emerging of competitor software designed to serve a similar goal.

Over the last few years, the web has surfaced as the most prominent cross-platform environment. Today, we can leverage a variety of different ways to distribute web-based applications to end-users.

First, the most apparent one is through the various modern web browsers [68]: Google Chrome, Safari, Firefox, Opera and Edge. Today, all of these support Progressive Webapps (PWAs) [7], to allow access to features traditionally reserved for installed apps, such as background synchronization, resource caching (for offline apps), or home screen links. This is made possible by adding a web-manifest [32], containing metadata about the application, and a service worker (SW) [31], a client-side stored executable that runs as a proxy between all network requests of a site. Android offers to promote "trusted PWAs" via their playstore [19]. Apple has hitherto distanced itself from in-browser installable PWAs. Safari users can add links to any website to their home screen, although not via a popup as is common on every other platform [66], but rather through the same workflow as adding a bookmark. Apple also rejects the idea of publishing web-apps on their App Store [11]. Nevertheless, safari fully supports service workers, as well as the majority of web-manifests features [66], thus PWAs in their most basic form.

Second, to mitigate all compatibility issues, a Webapp can also be deployed in a traditionally more App-like environment. Electron (one of those frameworks) wraps any app into standalone executable, which can be installed on any platform, without the prerequisite of a modern web-browser [16]. Such "[...] frameworks differ significantly in basic approach and paradigm, and in scope. The basic paradigmatic possibilities have been summarized in figure **??**. Although the origin of this depiction reach back to 2012 [50, p. 63], it still serves well for its purpose. In fact, only with PWAs an extension might soon be necessary [...]" [68, p. 3]. Some of the most popular ones include React-Native, Ionic, Cordova, NativeScript and Electron. [68, p. 7]

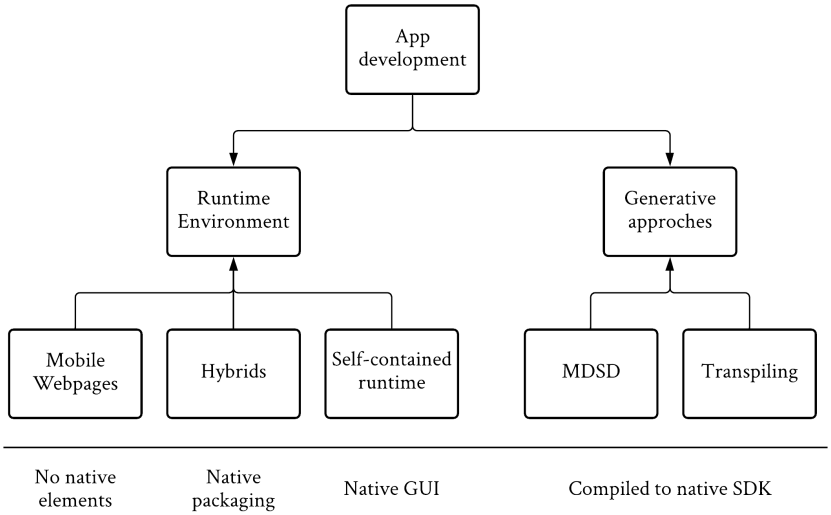

Figure 1: Webapp - native deployment paradigms (adapted from and by [69, p. 5], originally from [50, p. 63])

These possibilities entailed the rapid development of elaborate procedures supporting the development of scalable software. An active community of developers is shaping the open-source ecosystem used by millions of dependents [25][103, p. 2]. A variety of tools and languages are being developed by different organizations and independent web-advocates, which compile to the primitives of the web (*js*, *css*, *html*). Among the biggest players in this field are: *pug* and *Handlebars* (compiled to *html*), *scss* and *less* (compiled to *css*) and *TypeScript* and *CoffeeScript* (compiled to *js*) [79, p. 2].

0.1.1 Bundlers

Bundlers are conceptually designed to aggregate all these different resources and compile them into a distinct number of executables. In addition to their core features, bundlers may take advantage of plugins for a variety of transformation applications (see above), thus acting as a sort of task runner for compilers [63, p. 13-14].

Well-known bundlers are Browserify, Webpack and Rollup [63, p. 18] (see figure **??**). Browserify was one of the first bundlers to offer the functionality mentioned above, but is now getting outdated as some more modern features (listed below) are not or only scarcely supported [63, p. 20]. The following chapter will compare the approach of the latter two in-depth.

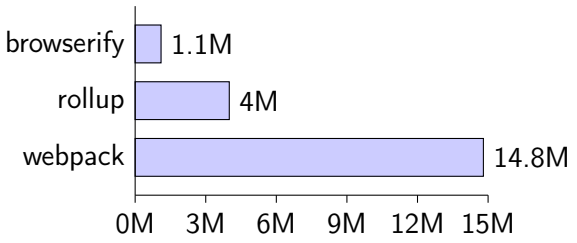

Figure 2: Downloads per Week [15][27][35]

In addition to being task runners for compilers, bundlers also manage import resolution by calculating a dependency graph. Having the entry script at the very top, the modules are injected in reverse order (bottom to top) into the output bundle, thus ensuring that dependencies are always defined before being used [63, p. 14, 15]. This is important for these three applications:

- For the web, as until recently, browsers did not support any kind of multi-file modularisation with a corresponding import/export syntax [65]. Bundlers can be used to support modularisation in legacy browser environments.
- Even though there has been recent push towards native support of modularisation by modern browsers, as of now "our general recommendation is to continue using bundlers before deploying modules to production."[78]. Whereas letting browsers resolve all dependencies could improve caching, there are a variety of performance concerns supporting the argument of using bundlers. Especially the potential for optimizations at compile time (as described below) outweighs the cons drastically [84].
- *Node.js* was originally designed with a Common *js* (CJS) module resolution. Only since version 13.2.0, released on the 21.11.2019 [92], does core *Node.js* natively support Ecmascript (ES) Modules (ESs) [93]. The implementation, however, is still lacking full interoperability with traditional CJS modules. While it is, strictly speaking, backward compatible, there are some issues concerning forward compatibility. At the time of this writing, it is not possible to CJS require an ESM module. This behavior is specifically written in the release specification [91], hence unlikely to change in the future. Bundlers can be used to circumvent this issue.

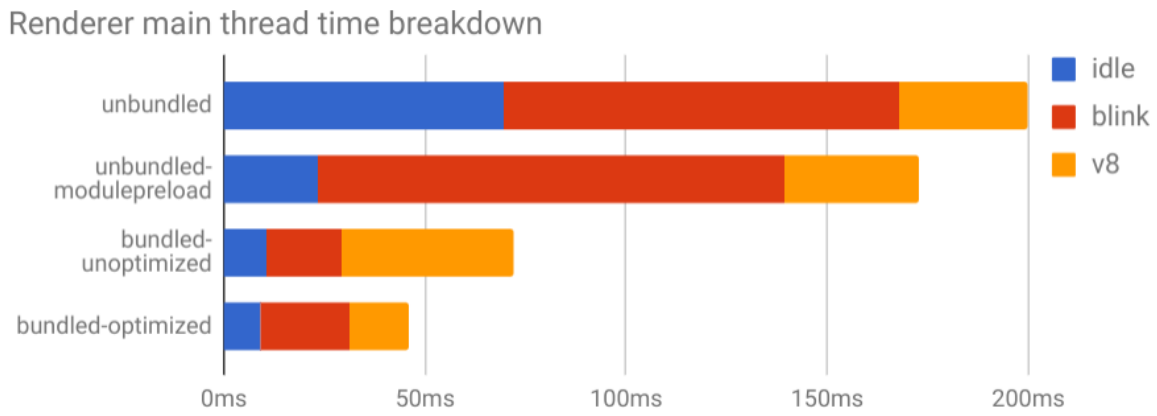

Figure 3: moment.js (composed of ~300 modules) [24] webpack-bundled and unbundled. "'Idle' means the main thread was idle [resources were downloaded]. 'V8' is the aggregated time spent in v8 functions, 'blink' is everything else"[84]

Notable here is that webpack and rollup have drastically different approaches on how to resolve dependencies. Take two modules 'main.js' and 'mod1.js' as sourcecode where main.js is the entry point.

```
// module.js
let localVar = "initial state module"
export function getVar() {
  return localVar
}
export function setVar(to) {
  localVar = to + " !changed!"
}
```

Listing 1: mod1.js source code

```
// main.js
let localVar = "initial state main"
console.log(localVar) // "initial state main"
import { setVar, getVar } from "./module"
console.log(getVar()) // "initial state module"
setVar("something else")
console.log(getVar()) // "something else !changed!"
```

Listing 2: main.js source code

Webpack utilizes Immediately invoked function expressions (IIFEs) very similar to *Node.js*, to create an enclosed scope. Such an IIFE gets wrapped around every module at compile time [63, p. 15-16].

```
((module, global, require) => {
  // module.js
  let localVar = "initial state module"

  module.exports = {
    getVar() {
      return localVar
    },
    setVar(to) {
      localVar = to + " !changed!"
    }
  }
})(/*webpack magic*/)

((module, global, require) => {
  // main.js
  let localVar = "initial state main"
  console.log(localVar)   // "initial state main"

  const mod = require("./module")
  console.log(mod.getVar())   // "initial state module"
  mod.setVar("something else")
  console.log(mod.getVar())   // "something else !changed!"
})(/*webpack magic*/)
```

Listing 3: IIFE module scope

This architecture allows for sophisticated code splitting (explained below), as it is simply a collection of IIFEs. Since enclosing a scope into a function is a core feature of *JavaScript*, utilizing IIFEs to achieve modularisation is naturally easy to handle. [63, p. 39, 40][51]

A drawback here would be the increased bundle size for each module (the IIFE and the small amount of `webpack magic` **??** to register it) as well as the chunk initializing the module index and its surrounding infrastructure (the webpack runtime). [63, p. 21]

Rollup, on the other hand, has a drastically different approach to the same problem. It tries to resolve all imports without the usage of enclosing scopes, instead relying on scope hoisting. "Scope hoisting means that instead of wrapping each module in a function and then chaining the functions, all the module code can be hoisted into a single function . This removes boiler plate code remarkably. It also reduces the amount of chained function calls, which makes the code run faster. When hoisting, the bundler however needs to take care of preserving the original behavior and preventing naming collisions."[63, p. 15]. Note how `localVar` in main.js is mangled into `localVar$1` (in listing **??**).

```
// module.js
let localVar = "initial state module";

function getVar() {
  return localVar
}
function setVar(to) {
  localVar = to + " !changed!";
}

// main.js
let localVar$1 = "initial state main";
console.log(localVar$1);    // "initial state main"
console.log(getVar());   // "initial state module"
setVar("something else");
console.log(getVar());   // "something else !changed!"
```

Listing 4: Scope hoisted module

This technique has, in addition to the elimination of boilerplate, one key advantage regarding bundle size. While the traditional webpack-like approach makes the **minifier** responsible to detect and omit dead code, here the bundler abandons unused code on a global scope. As rollup has access to the raw source code, which is generally much less ambiguous regarding side effects, it can be much more confident in aggressive dead code detection [51]. The minifier could, in theory, do just as well, but is traditionally limited to the already bundled code, by its position in the build cycle. Hence cannot make as bold of assumptions about its input's intent, as it works on a lower abstraction level [85].

While deploying gigantic bundles on the server-side will merely cause negligible performance penalties, the bundle size is of crucial concern when targeting client-side web browsers [73, p. 1]. Loading a whole application upfront will, no matter the minification, lead to unacceptable initial load times. Hence, all modern bundlers natively support code-splitting [95][94], as a mean of loading resources at runtime. When code-splitting, multiple bundles are output by the bundler, containing a composed part of the application. These chunks can be independently lazy-loaded at any time during the app's live-cycle. Smart utilization

of this technology can immensely improve First contentful paint (FCP) times, while the bundler does the heavy lifting of determining which resources to include in what bundle at compile-time [63, p. 16]. This computation is extensively more complex than reversing a dependency graph, hence code-splitting is exclusively supported by modern bundlers [63, p. 19-21].

Due to the distinct features of webpack and rollup, we oftentimes see webpack being used for large-scale applications, while rollup is being used to bundle standalone *JavaScript* modules or packages [51]. Wildly popular libraries such as react, vue and D3 use rollup to bundle their own executables, while popular frameworks for building applications such as react or vue utilize webpack to bundle the subject application [26][30][51].

0.1.2 Minification algorithms

Independent of the bundler used, utilizing a minifier in the build process, especially for client-side web projects, is beneficial, as it can dramatically reduce the download size of each output chunk. "Minifying means making the code shorter and reducing the amount of characters, whitespace, comments, semicolons and other extra markup used and renaming variables and functions with shorter ones"[63, p. 16], also known as variable mangling. There is a multitude of popular minifiers/tokenizers for *html*, *css* and *js*. The most popular ones include html-minifier (*html*) [74][57], css-nano (*css*) [88][28] and terser (*js*) [34][38], which all support webpack and rollup integrations.

Much like any other transpiler the majority of minifiers state as one of their design goals, to not change the runtime flow [29] [47]. One notable exception here is the Closure Compiler's 'advanced mode'. It tries to utilize runtime flow analysis to determine where the interpreter will go and thus optimize runtime performance, as well as bundle size [17]. To make this possible it requires the source to use explicit object indexing, as property names will be mangled [18].

Hitherto the Closure Compiler is limited to ES5 input (with some ES6 features supported) [18], hence may not be suitable for modern production applications. A possible workaround would be to transpile ES6+ source code to ES5 (via e.g. babel [14]) and then chain the Closure Compiler (advanced mode) to the build cycle. This may result in smaller output bundles, but one would still have to refrain from using runtime computed strings as object indices [18]. Under this limitation, however, one might find the Closure Compiler to output significantly better results than a traditional minifier.

0.1.3 Image compression

"A picture is worth a thousand words, but byte-wise, they often cost an order of magnitude or two more." [80]. "Unoptimized images are often the greatest contributor to page bloat. Looking at the 90th percentile of the distribution of page weight, images account for a whopping 5.2 MB of a roughly 7 MB page [see figure **??**]. In other words, images comprise almost 75% of the total page weight."[46]

Percentile	Overall	Image	JS	CSS	HTML
90	7	5.2	1.1	0.2	0.1
75	3.8	2.4	0.7	0.1	0.1
50	2	1	0.4	0.1	0
25	1	0.3	0.2	0	0
10	0.4	0.1	0.1	0	0

Figure 4: Page weight on desktop broken down by resource type. Approximation in MB. [46]

'Image', being the heaviest resource across all percentiles indicates great potential for performance gains through optimisation [52]. The following chapter will compare the four popular compression algorithms for natural rasterized images, concerning modern web development: JPEG, PNG, WebP and AVIF. GIF, although popular, will intentionally be left out of this comparison, as it cannot be considered a general purpose image codec, due to it's limitation to 256 colors [97, p. 1][9].

First JPEG, released in 1992 [55, p. 15][36, p. 1], enjoys universal browser support to this date [21]. "The JPEG compression algorithm was clearly focusing on natural image or realistic scenes. Its performances were best on this class of images. For graphics and text, JPEG was not as well suited, especially at very low bit rates where artifacts appear at the boundaries of high contrast areas"[55, p. 13]. This marks a remarkably good match for common web imagery. This, in part, lead JPEG, long after being superseded by webp, to still retain its place as the most popular image codec on the web [80][3], with around 70% of websites using it today [36, p. 1].

Second PNG, released in 1997 [82], enjoys similar to JPEG universal browser support to this date [21]. It was originally designed to be a lossless compression algorithm but was later extended to feature a lossy option. While lossy PNGs generally perform much better on natural images than their lossless counterpart, both merely excel when benchmarking images with many sharp edges [20]. As such an assumption cannot be made across the board for natural images of any category, in fact, the opposite tends to be more true, PNG is not suited very well for natural image compression on the web. Nevertheless, due to its historical nature, PNG is the second most popular image codec on the web today [80].

Third WebP, developed in 2010 [71], has just at the time of this writing received full support by all modern web browsers [67]. Apple's Safari has until very recently not supported WebP, but has shipped Safari 14 [12] on September 16, 2020 [13] with WebP rendering capabilities. WebP is developed by Google, "[...] based on intra-frame coding from the VP8 video coding format [...]"[71] and is designed to supersede both JPEG and PNG [67][9]. Generally speaking, it performs much better with a lossy file-size reduction over JPEG of 25%-34% [33][36] and a lossless file-size reduction over PNG of 23%-42% [1] respectively. Therefore WebP should in almost all cases be preferred over both prior image codecs. Only the late adoption by Safari is holding it back from being the 'default' codec to recommend [9]. Until a sufficient percentage of Safari users can view WebP images, it cannot be sustainably used as sole image format on the web. Setting up an architecture to serve fallback images based on the client's browser capabilities [21] will most certainly be worth the extra effort [6], especially at scale. At this point, one should also consider supporting AVIF, for even better compression across the board [71].

Fourth AVIF, finalized in 2019 [60], is currently only supported on desktop browsers running on chromium [21][64]. Thus Chrome and Opera (desktop), which comprise about 26% of the global browser market share [64].

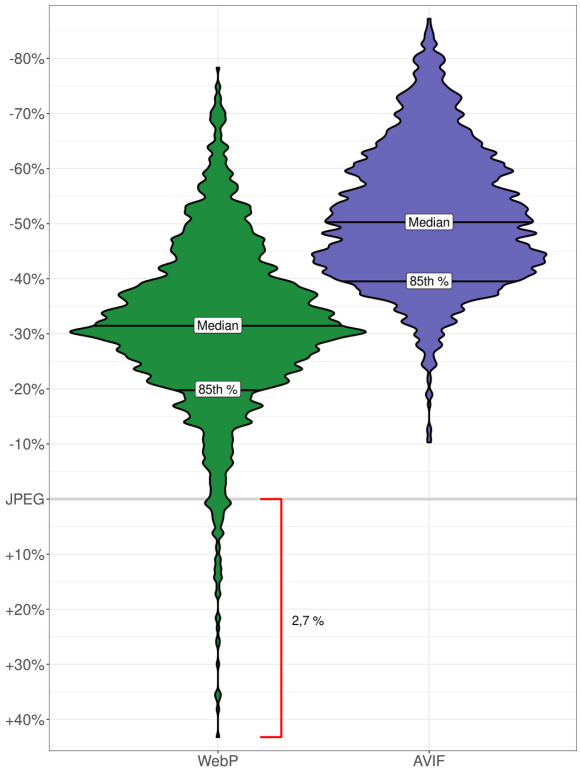

Figure 5: "Image file size savings in WebP and AVIF compared to the reference JPEG at the same DSSIM."[2].

AVIF is being developed by the non-profit [77] organisation Alliance for Open Media [75], a collaboration of Google, Apple, Mozilla, Intel and other tech giants [76][87], and "allows encapsulating AV1 intra-frame coded content"[36] AVIF is set to supersede WebP, alongside PNG and JPEG [67], with a median file-size saving of 50% over JPEG and 27% over WebP (derived from figure **??**). Therefore AVIF should in almost all cases be preferred over every other image codecs, when supported. At the current browser implementation coverage AVIF, similar to WebP but worse, cannot be considered to be a standalone image codec [64]. When targeting anything outside the very high end of client environments, maintaining fallback resources rendered in a more widely supported codec is a must have [21]. Granted, "unless it's automated, offering up 3 versions of the same image is a bit of a pain, but the savings here are pretty significant, so it seems worth it, especially given the number of users that can already benefit from AVIF."[6]

To emphasize this point, we will compare the same image encoded with JPEG and AVIF, both at 45KB, side by side. WebP sits somewhere in the middle between AVIF and JPEG in visual fidelity [2] as analyzed in figure **??**.

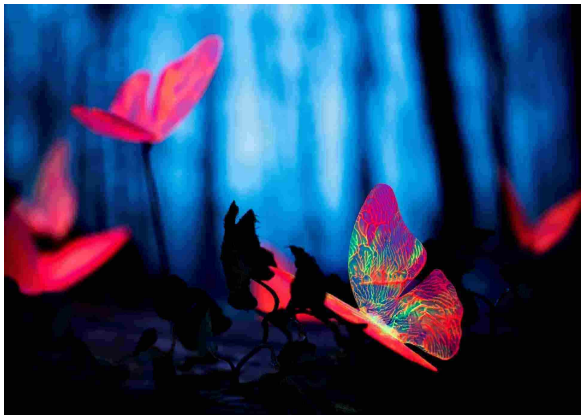

Figure 6: JPEG reference image at 45KB. An interactive demo can be found at [87].

Figure 7: AVIF reference image at 45KB. An interactive demo can be found at [87].

For an interactive demo of the difference between figure **??** and **??** please view [87], or visit [6] for an in-depth analysis, comparing even more codecs for different use-cases. A introduction to how this remarkable quality gain conceptual works can be found here [9].

Another previously untouched aspect of **perceived** loading performance is progressive image decoding. An image can, generally speaking, be encoded with one of two decoding features: baseline and progressive or none at all. Baseline encoded images load from top to bottom in full detail, while progressively encoded ones are decoded in multiple passes, adding detail with each iteration [3] Figure **??** tries to illustrate this process. Progressive, and to a lesser extend baseline, encoding can add tremendous benefits to the perceived loading performance of images [3]. This technology has similar psychological effects as progress indicators [100]. They are in many ways even better, as the loading process is virtually being shown instead of an abstract representation of progress.

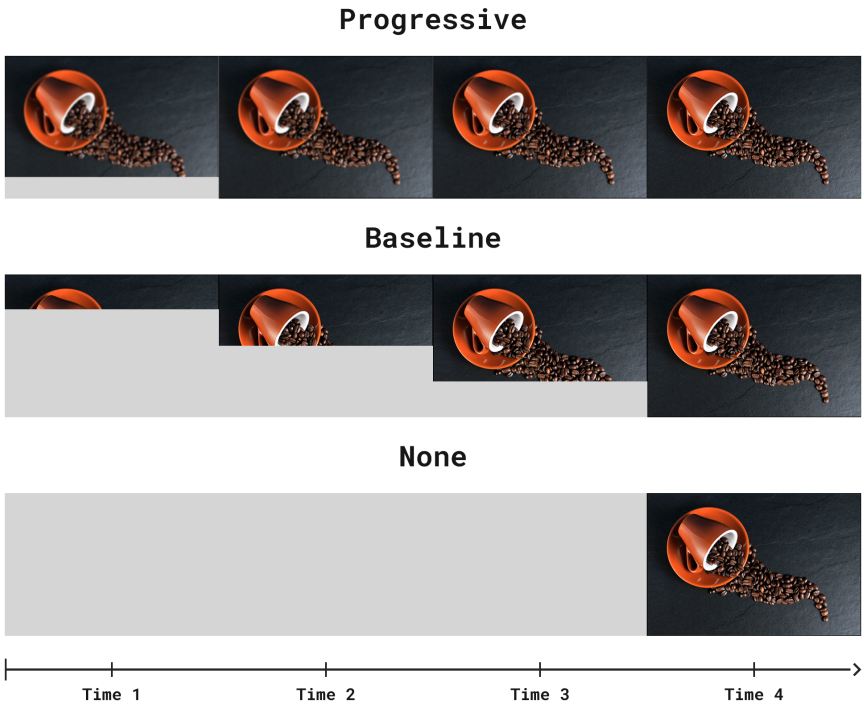

Figure 8: Loading states over time (x-axis from left to right). Note how the first frame of 'progressive' is significantly blurred. A demo can be found at [58] and [6] (Figure adapted from [3])

At the time of this writing, merely JPEG encoders support progressive decoding. WebP supports baseline decoding and AVIF currently none at all. The complete lack of support for enhanced decoding features by AVIF is mostly due to the young age of the algorithm and the fact, that the image codec is originally based on a video codec, which by nature do not need to render a partial frame [6]. This might change with time though, as the addition of progressive decoding to the AVIF spec is under discussion [8]. Meanwhile, this could be solved by rendering a preview image at compile time in drastically reduced resolution. Which can consequently be displayed as an intermediate placeholder, while the full image is loading. This would leave creative control over how the preview looks and behaves to the developer (maybe adding filters like a blur), but on the other hand add unnecessary network requests and consume bandwidth [6]. A proof of concept can be found at [6], showing the evident UX virtue.

0.1.4 Conclusion

0.1.4.1 Bundlers

While browserify served its years well, now it is simply missing crucial features, deeming it inconsiderable as bundler for the TGM-website. Rollup and webpack originate from very different methodological approaches, both heaving a compelling feature set for a specific use case. Although they grow closer together with age, we can still confirm that webpack is better suited for large scale web-application cores consuming many different resources, while rollup features better tree shaking and is therefore better suited for pure *JavaScript* / *TypeScript* projects.

0.1.4.2 Minification algorithms

Terser proves to be a very solid and modern *JavaScript* minifier, with an active community and amazingly responsive developers [85]. It might, however be worth to develop an extensive build stack utilizing the unique features of the Google Closure Compiler at a huge project scale. As the TGM-website is still reasonable in scope, we opted for terser.

0.1.4.3 Image compression

The clear winner in terms of performance (regarding compression size) is unsurprisingly AVIF, followed by WebP. We do however also see some significant benefits favoring JPEG. Progressive image decoding should really be considered in this comparison, beside the ever accompanying browser compatibility argument of course. As progressive image decoding is currently still under discussion in the AVIF spec, we discussed a possible workaround for this lacking feature.

0.2 Build stack

We decided to opt for a very traditional project architecture, in which we separate the frontend from the backend insofar that the frontend is using a JSON-based API as the only means to communicate with the server. This strict separation allows us on the one hand to develop both applications independently, once a common API specification has been defined. On the other hand, as this disjuncture is realized by maintaining two independent git repositories, we were able to hook up continuous integration (CI) and continuous deployment (CD) scripts to every commit registered by a GitLab instance. This opened up the possibility for a three-tier deployment architecture where the Development (DEV) branch on the repository is automatically deployed to a contained environment, while only receiving dummy data. The master branch can, analog to DEV, be automatically deployed to an enclosed environment called User acceptance testing (UAT), as in user acceptance testing, which contrary to DEV works with production data. The production environment alone is by design not automatically deployed by any hook. The always up-to-date DEV- and to its extension UAT environments are intended to help the communication process by making it effortless to communicate half-finished ideas without impacting production.

As the project's source code is not directly used as distribution, at times for performance reasons and sometimes out of necessity, since the browser does not read the high-level programming language used for development, we engineered an intricate building pipeline or build stack. This pipeline gets executed on every developer's machine as well as the CI / CD servers and when deploying to production, using different presets. The following section will give insights into the development process concerning this project's build stack.

0.2.1 Concept

Today, a modern web application's frontend can be divided into two separate runtimes. A service worker thread handling all background tasks and acting as a network proxy, and the core frontend source which controls all visual representation of a site.

Therefore we opted for a dual build architecture. Rollup will be used to transpile all service worker source files, resolve their dependencies and finally minify the code using terser. As the *sw* source is by nature confined to *TypeScript* / *JavaScript* we can expect to yield great results in terms of bundle-size without needing an elaborate configuration.

Contrary, the core frontend code will be comprised of a multitude of vastly different file types, which will all have to be minified, some may even require to be transpiled first by different build tools. To name just a few applicable languages we are using besides the *TypeScript* core: *pug* (*html*), *css* and *svg*. This, in combination with the requirement of lazy loading and multi-threading capabilities, induced us to consider webpack the better choice here.

Both build scripts pipe their output to a dedicated */public* directory. This distributable can consequently be statically served by an *nginx* server for production or a local development server powered by Node.js. Former is designed to provide an easy-to-change dummy API and features some development-focused tooling. Changes made in the source of either the frontend or the server itself are instantly reflected in the browser by automatically reloading when necessary.

Another, heretofore untouched aspect of building the website is image compression. sub-section '**??**' found images to be generally speaking the largest contributor to overall page size and therefore concluded that the most potential compression gains are to be found here. During research, we made it a priority to capitalize on the aforementioned prospects. Due to current browser limitations in terms of image codec

support, we envisioned a stack, capable of distributing images of different codecs with different resolutions depending on the requestee's screen size and codec support. Another proposed feature was the universal application of progressive image decoding, polyfilled if not possible otherwise. This approach should, however, not penalize server runtime performance nor encourage development friction in a significant way. This proved to be a difficult undertaking especially considering the long compilation times of the AVIF codec at large resolutions.

0.2.2 Implementation

Our initial approach to solving the issue revolved around the well-known webpack add-on *img-loader*, which works on the basis of declaring compression algorithms for specific file types. *Img-loader* performs static analysis on *css* files in search for `background-image: url("...")` declarations pointing to a relative file location. Those will be replaced depending on the file size by a base64 URL containing the image or a hash associated with the compressed distribution file. This concept quickly proved to be inadequate for our application context, as utilizing *css* as the only declaration point, tremendously limits our feature detection capabilities. Hence this premade loader merely allows for same-codec compression and leaves the codec choice together with the polyfill responsibility to the developer. This will, on a project scale, either lead to unoptimized images across the board or hours of developer time wasted recompiling images by hand, which are both not viable.

Another option we considered was writing our own webpack-plugin, scraping all rasterized images from the project, compressing them with different codecs and at different resolutions using *sharp* and finally reflecting them at a central directory. All images would have to be uniquely identifiable by name as static analysis of imperative languages like *JavaScript* or *TypeScript* is vastly out of scope for this endeavor. This way, a client-side proxy responsible for all image requests, may then, at runtime, inquire which image codecs are supported by the current browser and continue with the best possible one. This proxy could further specify the desired resolution, to better adapt to the client's window size - again saving bandwidth. Even our last proposed feature, progressive image decoding, could be implemented, using this technology by polyfilling. An image compiled to an absurdly low resolution could be preloaded, blurred, and used as a preview while the full image is being loaded. While this approach is perfectly suitable for production, as it does not cut corners during the client-side runtime, we see significant processing time at compile-time due to the number of images being used. Waiting approximately 20 minutes on each deployment cycle is bearable but not trailing every development session nor say, whenever a new image is added. Therefore we required an applicable caching mechanism, not recompiling already compressed images on a local machine.

Thus, we opted for another approach, the one we actually used. We incorporated all lessons learned from the two prior attempts and developed a standalone nodejs script. For better reusability, we decided to publish it as an npm package under the name of *image-web*, where the source code can be found today. This way we could offer streamlined installation alongside a *cli* to ease the entry barrier without compromising on advanced *api* operability. We initially envisioned a caching algorithm based on the filenames already compressed, which worked reasonably well, until images were changed (and their name was not updated). As an image having the specified name may already have been compiled, thus may exist in the destination directory, updated images were not reflected in the compiled application. Although this issue could be circumvented by prefixing an incriminating number so that the filename would stay unique, we decided to resolve it by pragmatically creating a dotfile at the destination, storing the source images' last changed dates and diffing those to the current ones. This final approach worked wonders as it could easily be added to the project's build script and thus be independently executed by developers and *CD* servers alike.

The rest of this endeavor was comprised of fine-tuning compression levels of the various codecs to achieve similar human perceived visual fidelity and optimizing compile times by multi-threading the compilation processes. Another interesting finding during development was the 'effort'- or its inverse 'speed' parameter available on each used compression algorithm. Compromising on compression effort during development could greatly improve initial compilation time. To avoid issues similar to the mentioned rename phenomenon, we preemptively chose to include the last specified effort into the *dotfile*. Another unexpected problem arose during production when an admin wanted to upload a specific image, which mysteriously rotated when piped through our compression script. It turned out to be configured with an *exif-orientation* property, which was unknown to the developers and thus overlooked until the issue arose.

This implicitly transitions towards the topic of the other context in which our script found application: the admin interface. *Ghost*, the Content Management System (CMS) hosting the admin interface, uses, at the time of this writing, their own *sharp* powered image compression framework to automatically compress uploaded images at runtime. Their framework is naturally, by our standards, far inferior to our solution, therefore we modified ghost's source to utilize our library over their implementation. Here, the decision to opt for a standalone script, published to npm, paid off a second time.

This leaves merely one question unanswered: How does the client-side image proxy check for codec compatibility on the fly? This was surprisingly easy to solve. The HTML *picture* element, as in listing **??**, provides a simple API to declare fallback options from top to bottom, in case one or more sources cannot be rendered by a subject browser. Note how the last child, and thus the last fallback, is a traditional *img* element. This ensures virtually complete browser compatibility as if the *picture* and to its extension the *source* elements themselves are not supported, which is already very unlikely, then we simply fallback to the *img* element without any codec checks. Since the *picture* element may, in such cases, be ignored, we direct all style declarations to the *img* element. This usage is as such intended by the specification, in fact, the *source* elements are even in supporting environments considered as shadow elements, overwriting the *src* attribute of the main *img* [23].

```
<picture>
  <source type="image/avif" srcset="filename.avif">
  <source type="image/webp" srcset="filename.webp">
  <img src="filename.jpg">
</picture>
```

Listing 5: Codec support detection with the *picture* element

Compliance to this implementation is realized by providing a custom *image* component taking an image identifier without file-type annotation, which creates the aforementioned *picture* template inside its body accordingly. The now nested *img* element implicitly inherits all applicable styles from its *image* parent, since those properties do not propagate the DOM tree by default nor are the elements directly accessible from outside the local scope.

0.3 Load strategy

Early in the project, we made the decision to develop using a client side rendering architecture. Although the project evolves around a website in the traditional sense in contrast to the team's previous work, where we built a web app, we decided to migrate the client-side page router and strongly adapt it. This allows us to stay in control of the site's state during navigation, enabling intricate animations elevating the user experience, in addition to the performance enhancing lazy and preloading capabilities this opened up. The following section will focus on the load strategy envisioned and engineered by the frontend developers.

0.3.1 Concept

Many web-pages are distinctly separated into pages and further sections within them. We commonly see the prior design choice being leveraged for lazy loading optimizations, whereas the former is only seldomly considered. The frontend engineers together with the UX designers of our team chose to break this convention by developing a section-based page architecture that vertically integrates into the website as a whole. Both, the UX and the loading strategy heavily utilize this methodical approach accordingly. Figure **??** illustrates this separation in blue taking the Industrial engineering (as in the german Wirtschaftsingenieure) (WI) Page as an example. Every section is uniquely resolved by its own sub-directory on the website's URL, depicted on the left-hand side of figure **??**. So if, say the base url of the WI-page reads 'tgm.ac.at/tagesschule/wirtschaftsingenieure/', we would see the hero section. Now, when scrolling towards the next section, the url will change to 'tgm.ac.at/tagesschule/wirtschaftsingenieure/info/' and further to 'tgm.ac.at/tagesschule/wirtschaftsingenieure/hightlights/', always matching the current scroll position. Additionally, the header, depicted in figure **??**, will highlight the current section on the right-hand side. Of course, this method works bidirectionally, thus clicking onto a section name or changing the url directly will resolve to the according section.

Figure 9: Page sections illustrated - on the WI department page

We use this separation not only for structural guidance but also to power the aforementioned section-based lazy loading functionality. The ability to load sections individually enables us to specifically target the requested one first in our loading strategy. So that if let's say a user is directly referred to the '/info' sub-directory of a department page, we merely have to wait for this individual section to load before displaying the first-contentful-paint. The rest of the sections available on the page will be loaded as part of our preloading strategy.

To illustrate the preloading strategy, we will consider the following load stages, which form the basis of

all preloading efforts. We envisioned three loading stages, applicable to pages and sections alike.

- **Minimal content load**

  This stage primarily focuses on resolving as quickly as possible, while still providing some sort of minimal content to interact with. Before this stage is reached no section nor page will ever be visible to the user. During the minimal content load stage, we expect requests concerning the layout and style resources of the component and maybe some rudimentary text-based content to populate it, however only if deemed absolutely necessary. The design goal of waiting for this loading stage to complete, before considering a component ready, is to display a coherent loading indication as long as there is no reason to switch view.

- **Full content load**

  This stage will resolve as soon as all 'low-effort' content has been fetched and all functionality is available to the user. During the full content load stage, we expect requests to advanced scripting content, the full text source and low effort imagery like svgs or previews. The design goal of awaiting this loading stage is to indicate up to which point preloading may make sense. Going beyond, thus loading the next stage will not result in any improvements concerning the usability of the page, but may consume significantly more resources than the previous stages.

- **Complete load**

  The complete load resolvement indicates, as the name implies, the successful loading of all available resources of a subject frame. Since this is the last stage being called in the invocation chain all resources skipped at the previous load stages must be requested here. Oftentimes this merely concludes to images and similar artifacts being loaded here.

0.3.2 Implementation

This nested lazy- and preloading strategy was realized by declaring a common denominator of both the section and the page class called 'frame'. Every lazy-loadable frame must comply to the aforementioned guidelines by populating those three stage functions, which may only start loading as they are called and must return a Promise indicating its load status. The manager of such frames must invoke the functions in order, but is free to pause at any point in the invocation chain. The ability to pause is especially interesting for the preloading implementation. Preloading adjacent pages until their full content load stage can immensely improve perceived performance while preserving bandwidth. The complete loading stage will only be called on navigation to a specific site.

The individual loading of User interface (UI) components work on a low level by importing at compile-time split chunks via the webpack provided `import("./component.ts")` API. The staggered request for text-based content was equally effortless. In sharp contrast to the work required for the division between image source declaration and the actual loading event. To solve this issue, we implemented a versatile recording library, which allows interested parties to record all intended image sources into an array composed of setter functions executing the request instead of loading the images as soon as possible. This uses in fact the same proxy as used for the codec support checks. A very similar record-based approach was taken at the inquiry of adjacent pages. The only difference being instead of setter functions were URLs recorded.

Another noteworthy aspect of development was the deployment of an intersection observer to detect the currently active section. It provides intricate control over its behavior and gives precise access to the raw

intersection properties, all using a modern API, illustrated in figure **??**. Together with the history API and some state management, we were able to realize our conceptual vision without compromise. Through this fast-paced development process, we were able to do some user-testing and were able to fine-tune some values with the gathered feedback. It was however until late in development when we finally changed our history push / replace strategy from 'A click justifies a history push' to 'A navigation outside of the current context justifies a history push', as we noticed some users were getting confused in some cases by the prior ruling.

```
let options = {
  root: document.querySelector('#scrollArea'),
  rootMargin: '0px',
  threshold: 1.0
}

let observer = new IntersectionObserver((entries) => {
  entries.forEach(entry => {
    // Entry describes an intersection change for one observed
    // target element:
    //   entry.boundingClientRect
    //   entry.intersectionRatio
  })
}, options)

let target = document.querySelector('.listItem')
observer.observe(target)
```

Listing 6: Intersection observer usage example adapted from [22]

Bibliography

[1] Jyrki Alakuijala and Vincent Rabaud. *Lossless and Transparency Encoding in WebP*. 09.03.2021. Google. URL: https://developers.google.com/speed/webp/docs/webp_lossless_alpha_study.

[2] Daniel Aleksandersen. *Comparing AVIF vs WebP file sizes at the same DSSIM*. 10.03.2021. Ctrl blog. URL: https://www.ctrl.blog/entry/webp-avif-comparison.html.

[3] Daniel Aleksandersen. *Progressive JPEGs make a meaningful impact on perceived performance*. 10.03.2021. Ctrl blog. URL: https://www.ctrl.blog/entry/jpeg-progressive-loading.html.

[4] Izzat Alsmadi, Chuck Easttom, and Lo'ai Tawalbeh. *The NICE Cyber Security Framework*. Springer, 2020, pp. 67–69, 132–133. 262 pp. ISBN: 9783030419868.

[5] *API Reference - Ghost*. URL: https://ghost.org/docs/api/v3/ (visited on 11/24/2020).

[6] Jake Archibald. *AVIF has landed*. 09.03.2021. URL: https://jakearchibald.com/2020/avif-has-landed/.

[7] Jake Archibald. *Is Service Worker Ready Yet?* 15.02.2021. Google. URL: https://jakearchibald.github.io/isserviceworkerready/.

[8] Jake Archibald. *Optional 'progressive' download frame | av1-avif*. 12.03.2021. AOMediaCodec. URL: https://github.com/AOMediaCodec/av1-avif/issues/102.

[9] Jake Archibald and Surma. *Image compression deep-dive*. 09.03.2021. Google Chrome Developers. URL: https://www.youtube.com/watch?v=F1kYBnY6mwg.

[10] Stacie Arellano. *Understanding Web Accessibility Color Contrast Guidelines and Ratios | Stacie Arellano*. 04.03.2020. css-tricks.com. URL: https://css-tricks.com/understanding-web-accessibility-color-contrast-guidelines-and-ratios/.

[11] Apple developers authors. *App Updates for HTML5 Apps*. 16.02.2021. Apple. URL: https://developer.apple.com/news/?id=09062019b.

[12] Apple developers authors. *Safari 14 Release Notes*. 09.03.2021. Apple. URL: https://developer.apple.com/documentation/safari-release-notes/safari-14-release-notes.

[13] Apple developers authors. *webp browser support table*. 09.03.2021. Apple. URL: https://support.apple.com/en-us/HT211845.

[14] Babel authors. *Babel*. 04.03.2021. GitHub. URL: https://github.com/babel/babel.

[15] Browserify authors. *Browserify*. 16.02.2021. Npm. URL: https://www.npmjs.com/package/browserify.

[16] Electron authors. *Electron - Homepage*. 05.03.2021. Electron. URL: https://www.electronjs.org/.

[17] Google closure compiler authors. *Closure Compiler Wiki*. 02.03.2021. Google. URL: https://github.com/google/closure-compiler/wiki.

[18] Google closure compiler authors. *Understanding the Restrictions Imposed by the Closure Compiler*. 02.03.2021. Google. URL: https://developers.google.com/closure/compiler/docs/limitations.

[19] Google developers authors. *Trusted Web Activity*. 15.02.2021. Google. URL: https://developers.google.com/web/android/trusted-web-activity.

[20] ImageAlpha authors. *PNG can be a lossy format*. 09.03.2021. ImageAlpha. URL: https://pngmini.com/lossypng.html.

[21] MDN authors. *Image file type and format guide*. 09.03.2021. MDN. URL: https://developer.mozilla.org/en-US/docs/Web/Media/Formats/Image_types.

[22] MDN authors. *Intersection Observer API*. 19.04.2021. MDN. URL: https://developer.mozilla.org/en-US/docs/Web/API/Intersection_Observer_API.

[23] MDN authors. *Picture Element*. 19.04.2021. MDN. URL: https://developer.mozilla.org/de/docs/Web/HTML/Element/picture.

[24] Moment authors. *Moment*. 02.03.2021. Npm. URL: https://www.npmjs.com/package/moment.

[25] Npm Authors. *About npm | npm Docs*. 05.03.2021. Npm. URL: https://docs.npmjs.com/about-npm.

[26] React Authors. *React | Github*. 08.03.2021. Facebook. URL: https://github.com/facebook/react.

[27] Rollup authors. *Rollup*. 16.02.2021. Npm. URL: https://www.npmjs.com/package/rollup.

[28] Rollup plugin-postcss authors. *Rollup plugin postcss*. 02.03.2021. Npm. URL: https://www.npmjs.com/package/rollup-plugin-postcss.

[29] Typescript authors. *TypeScript design goals*. 02.03.2021. Typescript. URL: https://github.com/Microsoft/TypeScript/wiki/TypeScript-Design-Goals.

[30] Vue Authors. *Vue | Github*. 08.03.2021. VueJs. URL: https://github.com/vuejs/vue.

[31] W3C authors. *Service Workers 1*. 15.02.2021. W3C. URL: https://www.w3.org/TR/service-workers/.

[32] W3C authors. *Web App Manifest*. 15.02.2021. W3C. URL: https://www.w3.org/TR/appmanifest/.

[33] Webp authors. *WebP Compression Study*. 09.03.2021. Google. URL: https://developers.google.com/speed/webp/docs/webp_study.

[34] Webpack authors. *Terser Webpack Plugin*. 02.03.2021. Webpack. URL: https://webpack.js.org/plugins/terser-webpack-plugin/.

[35] Webpack authors. *Webpack*. 16.02.2021. Npm. URL: https://www.npmjs.com/package/webpack.

[36] N. Barman and M. G. Martini. "An Evaluation of the Next-Generation Image Coding Standard AVIF". In: *2020 Twelfth International Conference on Quality of Multimedia Experience (QoMEX)*. 2020, pp. 1–4. DOI: 10.1109/QoMEX48832.2020.9123131.

[37] Wilfred Van Casteren. "The Waterfall Model And The Agile Methodologies : A Comparison By Project Characteristics-Short The Waterfall Model and Agile Methodologies". In: *Academic Competences in the Bachelor* February (2017), pp. 10–13. DOI: 10.13140/RG.2.2.10021.50403. URL: https://www.researchgate.net/publication/313768860.

[38] Bogdan Chadkin. *Rollup plugin terser*. 02.03.2021. Npm. URL: https://www.npmjs.com/package/rollup-plugin-terser.

[39] David Cohen, Mikael Lindvall, and Patricia Costa. "DACS State-of-the-Art/Practice Report Agile Software Development". In: (). URL: http://users.jyu.fi/$\sim$mieijala/kandimateriaali/Agilesoftwaredevelopment.pdf.

[40] Mike Cohn. *Agile Needs to Be Both Iterative and Incremental*. en. URL: https://www.mountaingoatsoftware.com/blog/agile-needs-to-be-both-iterative-and-incremental (visited on 04/20/2021).

[41] Erika Corona and Filippo Eros Pani. "A review of Lean-Kanban approaches in the software development". In: *WSEAS Transactions on Information Science and Applications* 10.1 (2013), pp. 1–13. ISSN: 17900832.

[42] *CVE - Common Vulnerabilities and Exposures (CVE)*. URL: https://cve.mitre.org/ (visited on 11/23/2020).

[43] Ghost developers. *Ghost GitHub*. 25.11.2020. URL: https://github.com/TryGhost/Ghost.

[44] Caler Edwards. *How to use Negative Space in UI Design | Caler Edwards*. 26.02.2019. URL: https://www.youtube.com/watch?v=A0Ev_4zto4Y.

[45] Caler Edwards. *Typography - UX Tips | Caler Edwards*. 07.01.2020. URL: https://www.youtube.com/watch?v=ptqe5cfRg-8.

[46] Tammy Everts and Katie Hempenius. *Page Weight | Web Almanac*. 08.03.2021. http archive. URL: https://almanac.httparchive.org/en/2019/page-weight.

[47] Anthony Van de Gejuchte. *Terser doesn't do variable flow analysis*. 04.03.2021. GitHub. URL: https://github.com/terser/terser/issues/690#issuecomment-625197365.

[48] *hackerone: 793704 Server-Side Request Forgery (SSRF) in Ghost CMS*. URL: https://hackerone.com/reports/793704 (visited on 11/24/2020).

[49] Orit Hazzan and Yael Dubinsky. "The Agile Manifesto". In: *SpringerBriefs in Computer Science* 0.9783319101569 (2014), pp. 9–14. ISSN: 21915776. DOI: 10.1007/978-3-319-10157-6_3.

[50] Henning Heitkötter et al. *Business Apps: Grundlagen und Status quo*. 2012.

[51] Katie Hempenius. *Katie Hempenius on 'Improving Page Performance in Modern Web Apps'*. 08.03.2021. Smashing Magazine. URL: https://vimeo.com/254858694.

[52] Katie Hempenius. *Page Weight Percentiles: overall & by resource type*. 08.03.2021. Google. URL: https://twitter.com/katiehempenius/status/1103005053353369600.

[53] Hoffman and Andrew. *Web Application Security: Exploitation and Countermeasures for Modern Web Applications*. eng. Sebastopol: O'Reilly Media, Incorporated, 2020. ISBN: 9781492053118.

[54] *How to run Scrum efficiently in 2019? Quick guide for beginners*. en. URL: https://habr.com/en/company/hygger/blog/455022/ (visited on 04/20/2021).

[55] Graham Hudson et al. "JPEG-1 standard 25 years: past, present, and future reasons for a success". In: *Journal of Electronic Imaging* 27.4 (2018), p. 040901.

[56] Charles Humble. *The Future of Java in the Enterprise - InfoQ's Opinion*. 15.02.2021. Infoq. URL: https://www.infoq.com/articles/enterprise-java-opinion/.

[57] juliendargelos. *Rollup plugin html-minifier*. 02.03.2021. Npm. URL: https://www.npmjs.com/package/rollup-plugin-html-minifier.

[58] Pooya Karimian. *Progressive JPG Demo*. 12.03.2021. Pooya Karimian. URL: http://pooyak.com/p/progjpeg/.

[59] Eric D. Kennedy. *7 Rules for Creating Gorgeous UI*. 13.11.2014. medium.com. URL: https://medium.com/@erikdkennedy/7-rules-for-creating-gorgeous-ui-part-1-559d4e805cda.

[60] Nimrod Kramer. *AVIF — the next-gen image format you need to know about*. 09.03.2021. Gitconnect. URL: https://levelup.gitconnected.com/avif-the-next-gen-image-format-you-need-to-know-about-631e4476dc71.

[61] Steve Krug. *Don't Make Me Think: A Common Sense Approach to Web Usability Voices That Matter*. 2013.

[62] Craig Larman. *Agile and Iterative Development: A Manager's Guide*. en. Google-Books-ID: 76rnV5Exs50C. Addison-Wesley Professional, 2004. ISBN: 9780131111554.

[63] Sonja Laurila. "Comparison of JavaScript Bundlers". In: (2020).

[64] Caniuse maintainers. *AVIF browser support table*. 09.03.2021. Caniuse. URL: https://caniuse.com/avif.

[65] Caniuse maintainers. *es6-modules browser support table*. 25.02.2021. Caniuse. URL: https://caniuse.com/es6-module.

[66] Caniuse maintainers. *Web App Manifest browser support table*. 16.02.2021. Caniuse. URL: https://caniuse.com/web-app-manifest.

[67] Caniuse maintainers. *webp browser support table*. 09.03.2021. Caniuse. URL: https://caniuse.com/webp.

[68] Tim A Majchrzak, Andreas Biørn-Hansen, and Tor-Morten Grønli. *Progressive web apps: the definite approach to Cross-Platform development?* Hawaii International Conference on System Sciences, 2018, p. 1.

[69] Tim A Majchrzak and Jan Ernsting. *Achieving business practicability of model-driven cross-platform apps*. 2015.

[70] Ilaria Matteucci. "Synthesis of secure systems". PhD thesis. Citeseer, 2008.

[71] Aditya Mavlankar et al. *AVIF for Next-Generation Image Coding*. 09.03.2021. Netflix. URL: https://netflixtechblog.com/avif-for-next-generation-image-coding-b1d75675fe4.

[72] *Metasploit | Penetration Testing Software, Pen Testing Security | Metasploit*. URL: https://metasploit.com/ (visited on 11/23/2020).

[73] Javad Nejati. "Characterizing Bottlenecks in Web Performance". PhD thesis. State University of New York at Stony Brook, 2020.

[74] T. Nishino. *HTML-minifier Loader*. 02.03.2021. Npm. URL: https://www.npmjs.com/package/html-minifier-loader.

[75] Alliance for Open Media authors. *Alliance for Open Media*. 09.03.2021. Alliance for Open Media. URL: http://aomedia.org/.

[76] Alliance for Open Media authors. *Alliance for Open Media*. 09.03.2021. Alliance for Open Media. URL: http://aomedia.org/membership/members/.

[77] Alliance for Open Media authors. *Vimeo Joins the Alliance for Open Media*. 09.03.2021. Alliance for Open Media. URL: http://aomedia.org/press\%20releases/vimeo-joins-the-alliance-for-open-media/.

[78] Addy Osmani and Mathias Bynens. *JavaScript modules*. 25.02.2021. V8. URL: https://v8.dev/features/modules.

[79] Andrew Pilsch. "Translating the Future: Transpilers and the New Temporalities of Programming in JavaScript". In: (2018).

[80] Tamas Piros, Ben Seymour, and Eric Portis. *Media | Web Almanac*. 09.03.2021. http archive. URL: https://almanac.httparchive.org/en/2020/media.

[81] Henny Portman. *What's The Difference Between Iterative And Incremental Development?* en-US. Mar. 2021. URL: https://thedigitalprojectmanager.com/iterative-and-incremental-development/ (visited on 04/20/2021).

[82] Greg Roelofs. *History of the Portable Network Graphics (PNG) Format*. 09.03.2021. Linux Gazette. URL: http://www.libpng.org/pub/png/pnghist.html.

[83] Kenneth S. Rubin. *Essential Scrum: A Practical Guide to the Most Popular Agile Process*. en. Google-Books-ID: 3vGEcOfCkdwC. Addison-Wesley, July 2012. ISBN: 9780321700377.

[84] Kunihiko Sakamoto. *ES module loading: Bottleneck analysis and Optimization plans*. 02.03.2021. google. URL: https://docs.google.com/document/d/1ovo4PurT_1K4WFwN2MYmmgbLcr7v6DRQN67ESVA-wq0/pub.

[85] Fábio Santos and Maximilian Mairinger. *Issue: Dead code detection for static class members not working; breaking tree shaking*. 27.02.2021. Terser. URL: https://github.com/terser/terser/issues/724.

[86] Herbert Schildt. *The complete reference Java*. 7. Auflage. McGraw Hill, 2018, pp. 348, 39.

[87] Justin Schmitz. *AVIF Converter*. 09.03.2021. avif.io. URL: https://avif.io/.

[88] Andrey Sitnik, Michael Ciniawsky, and Alexander Akait. *Postcss Loader*. 02.03.2021. Npm. URL: https://www.npmjs.com/package/postcss-loader.

[89] spacemonkey. *Joomla first commit*. URL: https://github.com/joomla/joomla-cms/commit/5169e6a03c4013c911293fef82413817fec04635.

[90] Gary Stoneburner, Clark Hayden, and Alexis Feringa. *Engineering principles for information technology security (a baseline for achieving security)*. Tech. rep. Booz-Allen and Hamilton Inc Mclean VA, 2001.

[91] Node.js Team. *Node.js v15.10.0 Documentation*. 25.02.2021. node.js. URL: https://nodejs.org/api/esm.html.

[92] Node.js Team. *Previous Releases | nodejs*. 25.02.2021. node.js. URL: https://nodejs.org/en/download/releases/.

[93] Node.js Module Team. *Announcing core Node.js support for ECMAScript modules*. 25.02.2021. node.js. URL: https://nodejs.medium.com/announcing-core-node-js-support-for-ecmascript-modules-c5d6dc29b663.

[94] Rollup Team. *Code Splitting*. 27.02.2021. Rollup. URL: https://rollupjs.org/guide/en/#code-splitting.

[95] Webpack Team. *Code Splitting*. 27.02.2021. Webpack. URL: https://webpack.js.org/guides/code-splitting/.

[96] *W3 school - CSS Selectors*. URL: https://www.w3schools.com/CSS/css_selectors.asp (visited on 03/30/2021).

[97] Yang Wang et al. "GIF2Video: Color Dequantization and Temporal Interpolation of GIF Images". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019.

[98] Stephen Watts. *State of Java in Programming Today*. 15.02.2021. BMC. URL: https://www.bmc.com/blogs/state-of-java/.

[99] *What is Agile Kanban Methodology? Learn the Methods & Tools*. URL: https://www.inflectra.com/methodologies/kanban.aspx (visited on 04/21/2021).

[100] Sara Willermark, Nikola Pantic, and Hannah Pehrson. "Subjectively Experienced Time and User Satisfaction: An Experimental Study of Progress Indicator Design in Mobile Application". In: *Proceedings of the 54th Hawaii International Conference on System Sciences*. 2021, p. 4476.

[101] WordPress. *WordPress*. URL: https://wordpress.org/.

[102] *xp*. URL: https://www.researchgate.net/figure/Extreme-programming-software-life-cycle_fig7_279415421.

[103] Markus Zimmermann et al. "Small world with high risks: A study of security threats in the npm ecosystem". In: *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 2019, pp. 995–1010.