UNIVERSITY OF APPLIED SCIENCES DRESDEN

Bachelor Thesis

Using JavaScript, APIs and Markup (JAM Stack) for Developing an E-Commerce Website Prototype

January 27, 2020

Simon Brandt
Matriculation number: 41084
simon.brandt@htw-dresden.de
Faculty of Informatics / Mathematics
Course of Studies: Business Informatics

1. Supervisor:
Prof. Dr. Thomas Wiedemann
2. Supervisor:
Dipl.-Inf. (FH) Robert Dominik

Abstract

The purpose of this thesis is to investigate the JAM stack paradigm: an approach for developing websites and web applications that promises speed, high security and scalability with a system that's flexible and easy to maintain for developers. The thesis will compare how problems in website development and delivery are solved in the LAMP and MEAN stack and put them into contrast to newer approaches. JAM stack websites are characterized by CDN-hosted static markup files, served directly to a web client without the use of web servers. The static front end is hydrated with a JavaScript layer that handles dynamic capabilities and connects to a distributed, API-driven back end when needed. By showing how an e-commerce website prototype can be built and deployed, the thesis provides a proof of concept for this type of website architecture and supports the techniques that were used by exploring and describing other platforms, built on the same principles. The thesis will furthermore point out the most important advantages and disadvantages of the JAM stack approach in its current state of technology.

Table of Contents

1 Introduction

Invented in the early 1990s, the World Wide Web can be argued to be the most important information sharing system today. Increasingly high demands on this communication platform continue to set high standards for applications running on it. During the past three decades, more and more people and businesses around the world started using the web, incorporating it into their lives in many different ways. Many people are depending on web-based applications privately or in their day to day work, raising the expectations on the capabilities of the medium. Especially in the last few years, users have become increasingly mobile and increasingly impatient. Expectations of immediacy and performance challenge some of the techniques that websites get delivered and displayed with.

This thesis will examine the JAM stack approach for building and serving websites. The approach encompasses tools and workflows that promise easy development and deployment of applications that are fast, secure and highly scalable. A productivity boost for developers and a performance boost for users are said to be achieved by rethinking processes and architecture of previous development approaches. A modular, distributed architecture and delivery system is promised to make JAM stack websites simple to work with as well as resilient to high traffic.

To begin, section 2 of this thesis will shed some light on the rough overall history that web development has gone through in the last three decades. This will include a closer look on the LAMP stack and the MEAN stack, two popular stacks for developing and serving websites and web applications. In section 3, a closer look will be taken into the theoretical concepts, tools and workflows used to build JAM stack projects. Section 4 will describe the implementation of a prototype website, utilizing the techniques discussed in the section before and analyzing the different technologies and service providers that were used to develop this website.

Section 5 will be showing ways for extending the prototype and how they can be implemented. It will furthermore list resources for exploring new tools and services around the JAM stack ecosystem. In extension to this, section 6 investigates two case study projects to show practical possibilities with the JAM stack architecture at a larger scale. The first case study shows how a JAM stack project is used to implement a transparent workflow to allow collaboration on the content of a website, while the second case study will provide insights into the architecture of an article publishing platform built on the JAM stack. Section 7 will tie the previous sections together by discussing advantages of the JAM stack architecture while also describing disadvantages and limitations of this approach.

2 Background

Section Summary: This section looks at changes that web development technology has gone through since the 1990s. It provides a historical context for this thesis and describes two popular technology bundles - the LAMP stack and the MEAN stack - more closely by exploring their basic architecture as well as strengths and weaknesses.

2.1 A Brief History of Web Development

Since the beginning of the World Wide Web in the early 1990s, the technologies and programming languages surrounding the web have gone through a series of evolutionary phases. In the first days of the web there was only Perl on the server side and HTML on the client side. Almost 30 years later a rich ecosystem of languages, frameworks and technologies has evolved and the next breaking changes always seem to be just around the corner. New frameworks, paradigms and technologies are developed constantly and only time alone seems to be able to tell which ones are just fleeting trends and which ones have the ability to solve deeper problems that can bring a long-term change to the environment.

In the beginning being mostly used by universities and academic institutes, the web became increasingly popular around the year 1994 for serving commercial websites leading up to the rise and burst of the ”dot-com bubble” between the years 1995 and 2002 [98, p. 1]. For the companies that survived this period, the World Wide Web, with its ability to bring access to information and services directly to the customer, grew to be a profitable source of revenue. Due to this development the face and structure of websites also went through a number of evolutionary steps from simple text-based, interconnected information systems to the media-rich, socially-driven and high-performing web platforms that we know and use today.

In a corresponding development, great leaps had to be made on the technology side of the World Wide Web to keep up with the increasing requirements of this global, ever-growing information and communication system that would eventually comprise of millions of websites for billions of users.

In the 1990s, for companies getting on board with the World Wide Web a critical factor in choosing the right technologies to run a web presence was cost efficiency. Many of the newly developed business branches around web technology were experimental or were not expected to return substantial value besides sole marketing purposes [44, p. 17].

During the 1980s with the rise of projects like GNU and the GNU General Public License open-source software was on its move and surely found its way into web technologies during the early 1990s [44, p. 13] [17] [18]. With open-source technologies at their hands companies could not only develop web-based software affordably but also had the advantages of being able to tweak and augment those technologies to their particular needs [44, p. 18]. Until today, open-source software plays a very important role in web development technologies.

In order to successfully run web platforms, certain problems needed to be solved. This included data storage, data processing, HTML page generation and delivery to the web client. In the course of evolving technologies, combinations of certain software components often proofed to be particularly well fitted to solve these problems. This resulted in the emergence of specific solution stacks that were commonly used by different companies.

This thesis will focus on the JAM stack but will briefly talk about the LAMP and MEAN stack in order to provide a perspective on the differences of web stacks and the way they evolved. Figure 1 shows how the three stacks discussed in this thesis relate to each other in their focus on back-end or front-end technologies. (All logos used in graphics of this thesis are intellectual property of their respective owners. This includes the logos of Linux, MySQL, Apache, PHP, MongoDB, Express, Node.JS, Angular, JavaScript, Stripe, Netlify, Gatsby, GitHub, OMDb, Formspree and Hugo.)

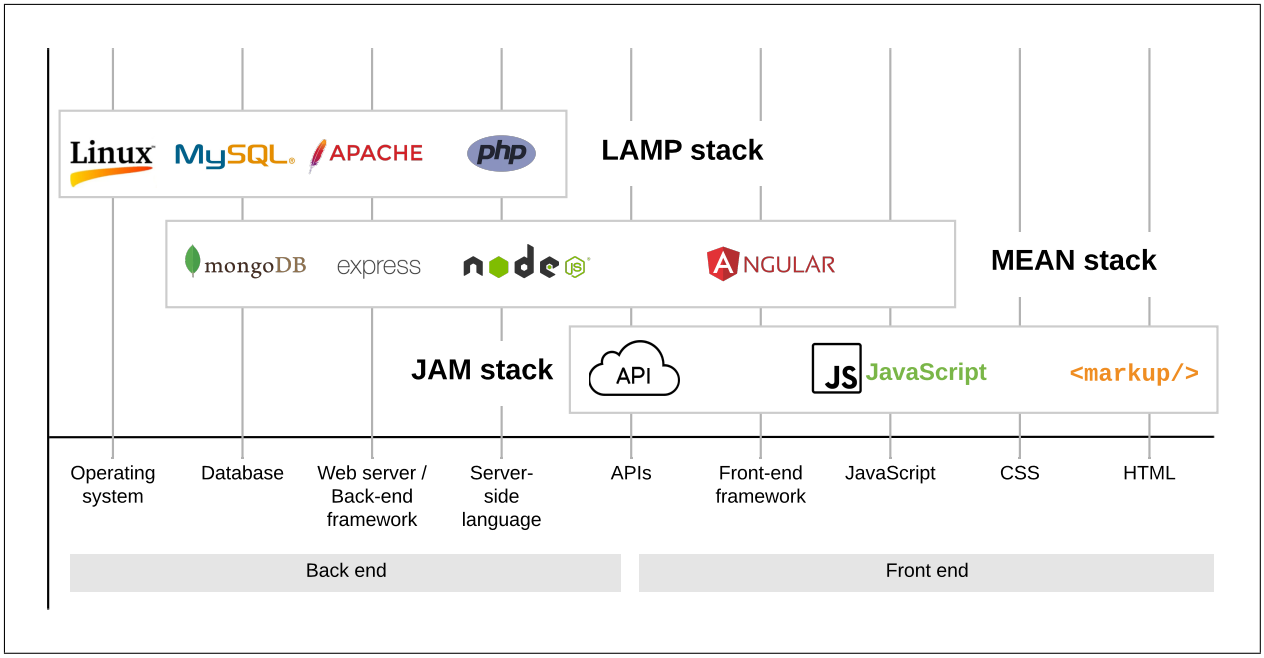

Figure 1: LAMP, MEAN and JAM stack and their relation in regards of back-end or front-end focus [11]

2.2 The LAMP Stack

2.2.1 Architecture

In the early days of web technologies, the individual components that made up a technology stack were independently developed software solutions that gained popularity by proving to be a suitable combination for satisfying newly arising requirements [71, p. 2]. LAMP stack is the combination of Linux, Apache, MySQL and PHP that early adopters of open-source software started using in 1997 [71, p. 2].

As websites and web applications needed a platform to run, the GPL-licensed operating system Linux soon gained popularity in web technology. In combination with the Apache web server, Linux proved to be a very capable system to serve content for the World Wide Web and remained the most popular solution to do so for almost two decades [52]. On the LAMP stack, data is stored in a relational MySQL database. When a user is requesting a resource behind a specific URL, the request is handled by the Apache web server. The request is passed to PHP, the server-side language layer, which loads the file and executes the script contained in the file [45]. If required, the PHP script will establish a connection to the database server where the MySQL system will query the needed data from the database, passing it to PHP. This data is then injected into a template and an HTML file is generated that is passed back to the Apache server who sends the file to the web client (figure 2). When the user activates an internal link on the website a new request is made to the target resource, repeating the process.

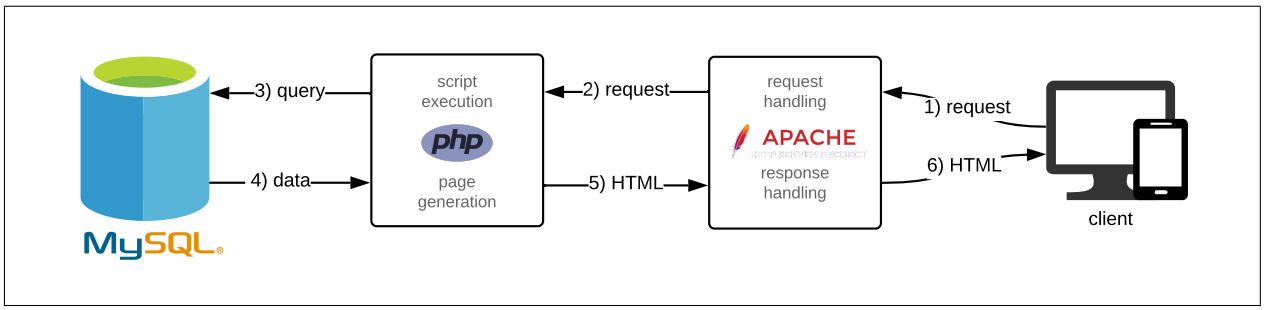

Figure 2: Request-response cycle on the LAMP stack

2.2.2 Shortcomings

This approach in handling user requests brings certain drawbacks with it. A primary bottleneck in the LAMP stack architecture is the multithreaded nature of the Apache web server. In order to

handle incoming requests, a number of idle web server processes have to be standing by, waiting to be assigned a new request - each of the processes requiring a certain amount of main memory [75, p. 11]. Scaling a LAMP stack application can become difficult and expensive very quickly as new server hardware needs to be set up to handle more incoming requests fast enough.

Furthermore the computational steps of request handling, script execution, database querying and template generation, necessary for serving an HTML file to the requesting client, have to be executed every time a request is made. Although caching layers can be integrated on different levels of the stack, this still leaves a number of redundant steps that are executed to serve content to the requesting user [9, p. vi].

While many individual steps can influence the performance of an application, there are also security implications that go along with building on this type of architecture. With Apache, PHP and MySQL there are different dynamic parts involved, each executing code, creating a bigger attack surface that can be exploited and that needs to be secured and kept up to date. A 2019 w3techs survey shows that the LAMP stack based content management system ”WordPress is used by 62.0% of all the websites whose content management system we know. This is 35.2% of all websites” [70]. The survey also shows that over 30% of these installations are running on outdated versions of WordPress making it a favorable target for hackers.

Finally, developing and maintaining LAMP stack applications requires know-how in multiple areas, reaching from front-end and back-end development, to server and database setup as well as DevOps to maintain efficient workflows. For small teams or individual developers, it is difficult to acquire all necessary skill needed to get development started, while for bigger projects different teams, writing different languages are required to handle the individual tasks [9, p. 21].

2.3 The MEAN Stack

2.3.1 Architecture

The MEAN stack is a JavaScript-driven combination of MongoDB, Express.js, Angular and Node.js. Over the years JavaScript as a programming language continuously evolved and was even discussed for its usage as a server-side language as early as 1998 [63]. Yet it was not until 2009 that the first version of Node.js was released - a cross-platform JavaScript runtime environment that significantly increased the popularity of JavaScript as a server-side programming language [13]. More and more JavaScript-based solutions for web development arose, including NoSQL database technology like

MongoDB in 2009 and front-end application frameworks like AngularJS in 2010 (forwardly only referenced as ”Angular”). In a 2013 blog post MongoDB employee Valeri Karpov discussed the usage of SQL-free database technology and how he and his team are integrating MongoDB in their development environment with Express.js, Angular and Node.js thus coining the term MEAN stack [47].

On the MEAN stack, the document-oriented database system MongoDB stores data as JSON, the JavaScript Object Notation [75, p. 8]. Express.js, a small framework built on top of Node.js, is used to organize web applications on the server side and handle tasks like requests processing, talking to the JSON database and revealing data to the network over API endpoints. When a MEAN stack application is requested by a visitor, the web server usually returns a minimal HTML page and a JavaScript bundle file with application code that was written using the Angular front-end framework. The JavaScript code is then executed in the browser, rendering the layout of the application. During runtime, the front-end framework can request data from back-end API endpoints and react to user interaction (figure 3 shows an example request-response cycle for a MEAN stack application). Changes in the user interface like data filtering or page transitions are made dynamically by applying updates to the Document Object Model (DOM) of the HTML page [75, p. 13 sq.].

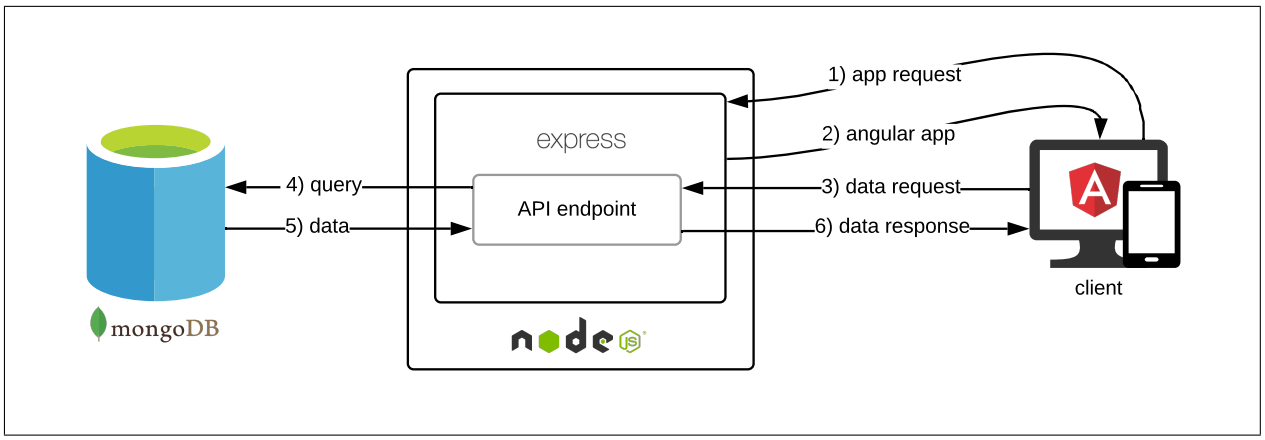

Figure 3: Request-response cycle on the MEAN stack

2.3.2 Improvements over the LAMP stack

One of the most significant innovations the MEAN stack introduced was a common programming language in all parts of the stack - from database over the back end to the front end. The homogeneous JavaScript-driven development environment reduces friction between front end and back end of an application and simplifies communication between development teams [46].

When used as a web server environment, Node.js can leverage its asynchronous, event-driven architecture to handle multiple incoming requests in a single thread thus scaling more gracefully and with far less server hardware compared to the multithreaded approach with Apache on Linux [75, p. 11].

2.3.3 Shortcomings

Still some of the problems addressed in the LAMP stack section remain relevant for the MEAN stack. All network facing parts of a web applications still have to be secured and all code executed at runtime poses a risk for the application to crash [9, p. 29]. Computational redundancy remains as server-side code and database operations are usually executed for every user who requests a resource.

3 The JAM Stack in Theory

Section Summary: In this section, technical background information about the JAM stack architectural paradigm is provided. A brief definition is given and the architecture of JAM stack projects is described by looking at important development tools and workflows. The section covers information about static site generators, headless CMSs, project deployment and approaches for handling back-end logic.

3.1 Definition

JAM stack stands for JavaScript, APIs and Markup. So far, looking at the LAMP or MEAN stack, a stack described a specific set of tools used to develop and serve websites. The term JAM stack however involves a more general component specification and provides less of a sufficient definition to understand how to use it or how it works. It can be described as a community collection of best practices and workflows [9, p. vii].

The term ”JAM stack” was coined by a team of developers around Mathias Biilmann, founder and CEO of the cloud computing company Netlify and described as ”a modern web development architecture based on client-side JavaScript, reusable APIs, and prebuilt Markup” [10]. What most significantly sets the JAM stack aside from previous stacks is that websites and web applications are served directly from a CDN, without the use of dedicated web server infrastructure [14]. According to Mathias Biilmann, the best practices that define a JAM stack project are: 1) everything lives in Git, 2) the entire site/app is hosted on a CDN, 3) builds are automated, 4) changes are published through atomic deploys and 5) deploys go live via instant cache invalidation on the CDN [66]. In order to better grasp the JAM stack architecture it is helpful to look at how the development process works and what types of tools and services are involved, reaching from static site generators (SSGs), over headless content management systems, to cloud hosting providers and the Git-driven workflow connecting the pieces (see figure 4).

3.2 Static Site Generators

The central tools that are required to build JAM stack websites are static site generators. To get an understanding of static site generators it is worth looking at their central purpose, sets of features and the existing variety of different SSGs. The main goal of employing a static site generator is

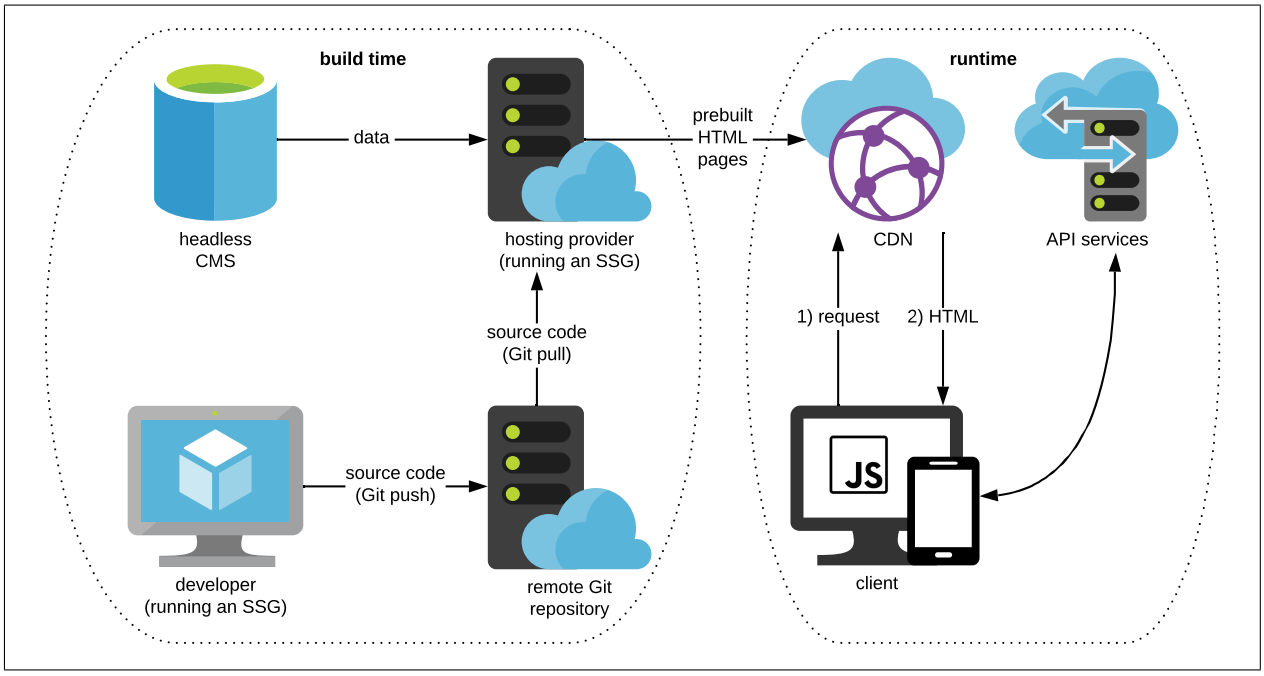

Figure 4: A visualization of the JAM stack architecture and its concise request-response cycle

generating static HTML files. SSGs can be described as a compromise between writing HTML files by hand and having them dynamically generated by a traditional CMS like WordPress. They are used for rendering data against templates to create non-dynamic HTML and CSS files that can be served to a client over the internet.

The process of rendering data into templates to create HTML files is essentially taking place in every LAMP stack project as well during runtime (for every request). The main difference on the JAM stack is that as much as possible work in generating HTML pages is done ahead of time (during build time) - before a page gets requested. To achieve this, static site generators include tool sets for developing layouts for page templates, as well as functionality to pull data into these templates. By running the build process of an SSG, all HTML pages for the project are generated. A blog website with one home page and 10 blog posts would output 11 HTML files (all linking each other) that are ready to be uploaded to a CDN.

To optimize generated websites and reduce load times, another common SSG feature is image optimization - rendering different resolutions for all images on a website during build time and responsively only serving the necessary resolution of an image depending on screen dimensions of the requesting device. Many SSGs also include progressive web app (PWA)-driven features like page-caching to enable offline access [28]. Similar to the LAMP stack process, page transitions are achieved with the full load of a new HTML page. To speed up page transitions, a feature called

link prefetching is often available - reducing load times of linked pages when browsing a website by already requesting HTML pages when their links enter the visitors viewport [24].

Central to the creation of websites is the ability of creating user interfaces in a component-oriented structure that allows flexibility and reusability of interface elements. To achieve this, many SSGs are built as a meta framework around established front-end development frameworks like Vue or React, leveraging their capabilities for creating user interfaces. The JavaScript-based static site generators Next.js and Gatsby for example use React at their core, while Nuxt.js is built around Vue. Other popular SSGs include Hugo (written in Go) and Jekyll (written in Ruby). Section 5.2.2 introduces a resource to find more available SSGs.

Figure 5 shows the popularity of some of the most-used open-source SSGs over time. Stars of the respective GitHub repositories are used as a rough indicator for the popularity of the project. We can see, while Jekyll is by far the oldest project, it has since been bypassed in popularity by a number of newer SSGs like Next.js, Hugo and Gatsby.

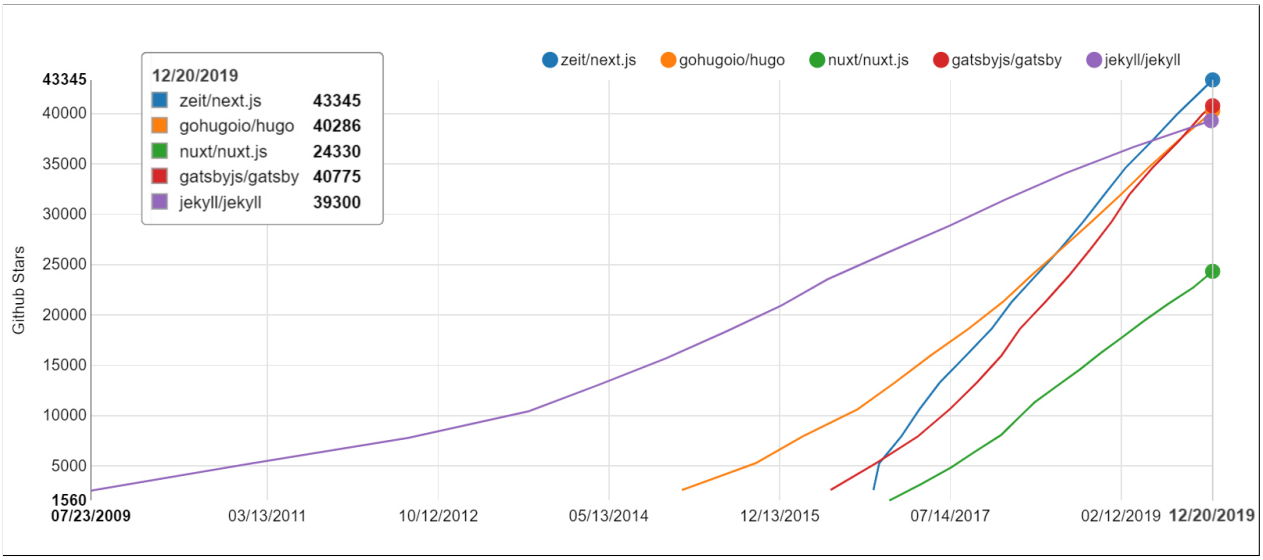

Figure 5: GitHub stars of different SSG repositories over time [69]

3.3 Headless CMSs

To handle data in a JAM stack project, headless content management systems (CMSs) are used. ”Headless” referring to the fact that no front end is connected to the CMS. Unlike traditional, coupled CMSs like WordPress or Drupal, that handle data and inject that data into front-end templates, a headless CMS only handles data and has no knowledge of how this data will be used or where.

Two basic types of headless CMSs can be distinguished: API-based and Git-based CMSs [68]. To understand the difference, it is best to have a look on how data is stored by those systems.

API-based CMSs store data in databases and are the more traditional way of headless content management. They expose API endpoints that can be connected to any platform over the internet. There are content management systems specifically designed to just handle data like the open-source solutions Strapi (www.strapi.io) and Ghost (www.ghost.org), or commercial providers like Contentful (www.contentful.com) or Prismic (www.prismic.io). Another popular approach is to run a traditional CMS decoupled from its front end. WordPress for example can be extended by plugins, enabling headless capabilities [96]. This way editors that are used to the WordPress interface won't have to adjust to a new platform when the change is made from a coupled to a decoupled architecture. Other popular, monolithically designed CMSs can also be witnessed of venturing into the area of headless capabilities in the recent years. This includes the systems TYPO3, Drupal, Magento and Shopify [95] [15] [77] [78].

The second type of headless CMSs, Git-based (or "file-based") CMSs, store data in flat files directly in the file system, often alongside the source code of a project. Usually Markdown is used to store page content where additional meta data can be saved by adding front matter to the top of the document. Front matter is a YAML-based syntax used to store variable information between two triple-dashed lines in a document [64] (see listing 1 for an example). This is a common approach for storing blog posts or news articles. Git-based CMSs are connected to the remote repository of a project, like a GitHub repository. Changes to a document made with the CMS will be directly committed into the project repository. Advantages of this setup are the seamless Git integration and the unified "database" environment shared by developers and content editors. While editors can rely on a graphical interface to work with content, developers can just as easily work on the raw Markdown files themselves without leaving their IDE. Popular examples for Git-based CMSs are Netlify CMS (www.netlifycms.org, open source) or Forestry (www.forestry.io, closed source).

3.4 Deployment

To publish a JAM stack project, a remote Git repository and a hosting provider needs to be added. To host static HTML files (technically) any hosting service able to serve these files would be sufficient. However the capabilities a JAM stack hosting provider should include are: 1) connecting to the remote Git repository of a project, 2) reacting to commits by 3) listening to a build hook, 4) pulling the source code and 5) executing the build command of the static site generator and uploading the generated files to a CDN (figure 6 shows the steps to setup the deployment workflow).

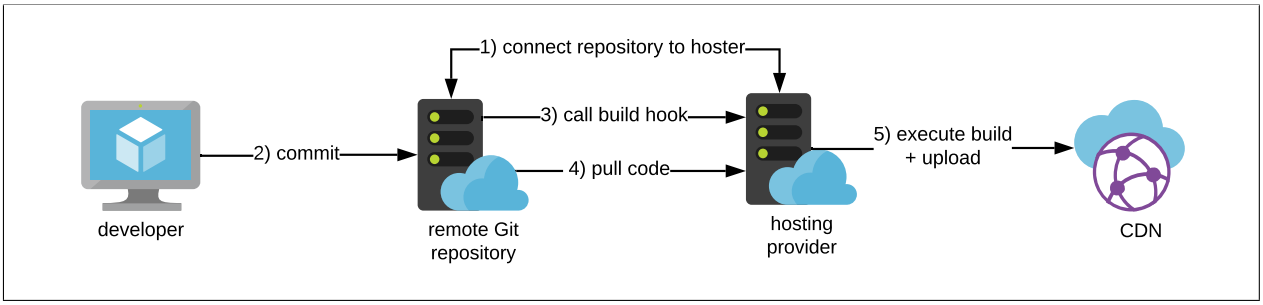

Figure 6: Steps for setting up the JAM stack deployment workflow

Utilizing this set of features a fully automated CI/CD pipeline can be set up. Popular JAM stack hosting providers include Netlify (www.netlify.com), ZEIT (www.zeit.co) or Google Firebase (firebase.google.com).

Different, otherwise unrelated services can be connected to each other by using webhooks: user-defined HTTP callbacks, triggered when an event occurs [7]. One service can notify another service of an event by sending an HTTP POST request to a specific endpoint with payload information about the event that occurred. The targeted server receiving the webhook request uses the payload information to determine what action to take [9, p. 43].

To set up a deployment pipeline, a hosting platform can provide a special build webhook that will trigger its build process. When a commit gets added to the remote Git repository, the repository platform will call this build hook, notifying the hosting provider, who will pull the projects source code into his system. The hoster can then build out the production-ready files, using the build command of the static site generator. The generated files will then be published globally to a CDN, putting them as physically close to the end user as possible [9, p. viii]. Committed changes that get automatically published through this pipeline include all possible code changes like adjustments in the layout or styling.

But rather than just having direct commits to the project repository trigger a new build, it is practice to have all content management tools configured with webhooks. A headless, API-driven CMS (not directly connected to the project repository) can be set up to call the hosting platform's build hook directly whenever data was changed. The hosting service will react to this build hook by triggering a new build of the project. When the build is running it will request the changed data from the CMSs API, injecting that data into the HTML files and deploying them to the CDN. (Git-driven CMSs are connected to this setup automatically. They are configured to commit changes into the remote Git repository, triggering a rebuild in the same way as a code commit by the developer). See figure 7 for this process.

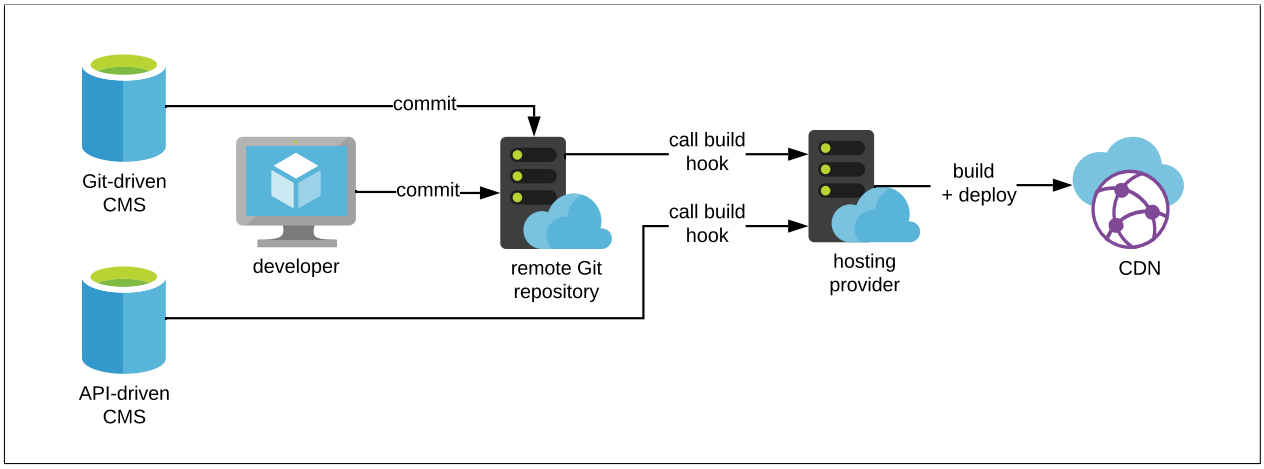

Figure 7: Process of triggering builds from different sources via build webhooks

Depending on the speed of the build process and the number of pages a full rebuild of a large website could take up to several minutes, making it impractical. The right tooling and configuration however will be able to build even extensive websites in seconds. A benchmark test published on the blog of forestry.io analyzed build times of the static site generators Hugo and Jekyll. Table 1 shows differences in build time and Hugo's capability of building a website with 1,000 pages in 0.65 seconds. A project with 10,000 pages will take 7.46 seconds to be built. However both Hugo and Jekyll provide incremental building modules that watch for changes and will rebuild only pages that have changed [49].

SSG / pages	**10 pages**	**100 pages**	**1,000 pages**	**10,000 pages**
Hugo	0.06s	0.12s	0.65s	7.46s
Jekyll	1.38s	3.8s	18.42s	218.61s

Table 1: Comparing build time for Hugo and Jekyll for different sized websites [49]

3.5 Back-End Logic

Since JAM stack websites at their core are just static HTML files, hosted on a CDN, there is no application back end to manage processes like authentication, form handling or data search. These tasks could be achieved by setting up a central back-end system, decoupled from the front end. However a more common way for performing back-end work in JAM stack projects is to connect software as a service (SaaS) providers over their public APIs and let them handle specific tasks. [9, p. 8]. By abstracting common server-side processes like image optimization or payment processing these third-party services are making specific functionality reachable from the front end of any application running on the web. Popular API services are Auth0 for authentication (www.auth0.com)

or Cloudinary for media optimization and delivery (www.cloudinary.com). The prototype section (section 4) will have a closer look at how the payment service provider Stripe can be used to accept payments (www.stripe.com). Section 5.1.2 shows how SaaS provider Algolia makes sophisticated data search available directly from an application front end (www.algolia.com). The advantage of using third-party services is that their high domain specialization can be leveraged and functionality can be integrated comparatively quickly without great implementation cost. If the required services are too specific with no external provider available, custom serverless functions or small microservices can be set up. The best-practice approach for JAM stack projects is to build decoupled, distributed systems, where each service does one thing and one thing well. Section 6.2 will briefly show how www.smashingmagazine.com uses small, individually designed tools for handling commenting, shop orders and login.

However, since no central server runtime is in use on a JAM stack project, there is no user session available for keeping track of things like a clients login state or role-specific permissions. These permissions would be necessary to give web clients access to non-public API systems across different services. This problem can be solved with stateless authentication using JSON Web Tokens (JWT) - an access token standard created by authentication service provider Auth0. JWTs are essentially client-stored strings with user and permission information, generated by an authentication service and signed with a secret key [67, p. 5-6]. An example JWT can state that the user admin@example.com has the role of an administrator. When a user makes a request to a private microservice he will send this token along. By configuring the secret key (that the token was signed with) in the environment of different microservices, they can independently verify the users claim to have administrator permissions and grant access to their APIs. No services have to have knowledge of each other or where the token comes from (see figure 8. Listing 2 shows an example JWT. More information about JWTs can be found at www.jwt.io).

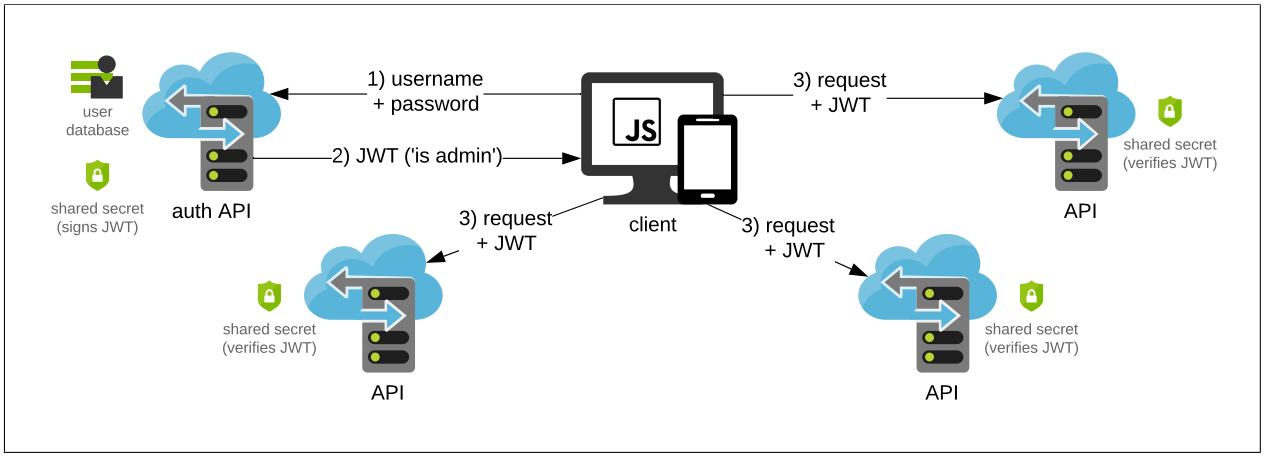

Figure 8: Using a JSON Web Token to give a client access to other services

4 Prototype Development

Section Summary: After looking at the theoretical concepts of the JAM stack in detail, the following section will contain a practical approach by describing the development of a prototype website that uses the JAM stack architecture. This section will be divided into the motivation for the prototype, gathering of requirements, designing the solution (including choosing the tooling needed to fulfill the requirements) and going over the implementation and deployment of the website. The last subsection will look at metrics such as speed, cost and developer experience.

The deployed website can be visited at: https://smbt-gatsby-ecommerce-prototype.netlify.com
The project source code is available at: www.github.com/smbt/gatsby-ecommerce-prototype

The project can be run locally with the Node Package Manager by executing the commands "`npm install`" followed by "`npm start`" from the root of the repository, hosting it a http://localhost:8000. By running "`npm build`", the static pages can be generated. (Only when using the code directly from GitHub, a `.env` file with API keys has to be added at the project root - see listing 9. The install and build are tested with npm version 6.13.1.)

4.1 Motivation

The central subject of the prototype will be an e-commerce website. A webshop is very suitable for a proof of concept of the JAM stack architecture for three main reasons.

Firstly, data in a webshop does not change very frequently. Still, having flexible product data at the core requires dynamic markup generation. This makes it suitable for carrying out the HTML generation ahead of time, after data changes in one distinct build, instead of during runtime on every request.

Secondly, setting up the back end for processing payments can be a huge, if not unsolvable challenge for small and even medium sized companies. Especially when support for international customers needs to be provided, juggling different currencies, accepting a variety of payment methods and complying to regionally varying regulations can be an overwhelming task to accomplish and poses a strong risk if mistakes during implementation are made. However building on a monolithically designed system like Shopify or Magento and relying on their back end to provide the necessary webshop infrastructure can be limiting when specific functionality is not supported or cost-intensive when it has to be added by upgrading the system. Therefore the API-driven JAM

stack approach can solve this dilemma by providing a distributed architecture with loosely coupled parts where the expertise of highly specialized back-end software providers can be leveraged without losing flexibility in the overall design of the project.

Lastly, page speed is an inherently critical factor for the success of e-commerce platforms. In 2018 Google announced that ”[s]peed is now used as a ranking factor for mobile searches” [43], making it an important factor to optimize for mobile devices with limited processing power and potentially unstable network access. On top of that, ”website performance is critical to maintaining customer attention and completing online transactions” [1], CDN provider Akamai writes in connection with a 2017 study on online retail performance, pointing out that ”[j]ust a 100-millisecond delay in load time hurt conversion rates by up to 7%” [2].

A JAM stack driven project setup aims to provide both increased development productivity by automating repetitive tasks and abstracting back-end processes as well as increasing page speed by highly optimizing towards direct delivery of content to the end user on a global scale.

4.2 Requirements

There will be three pages on the website: home, blog and contact. (1) The home page will show a banner with a full-width picture that has to be served responsively for small devices. It will display the shop products with a ”buy now” button that redirects to the payment process. Furthermore there will be a section that dynamically loads content during runtime and displays it on the site. (2) The blog page will contain a list of blog posts, each with a link ”Read article” that links to the blog post in its full length. A CMS will be in place to allow non-technical editors to manage the blog posts. (3) The contact page will contain a contact form that allows visitors to send messages to the shop owner.

4.3 Design

Figure 9 shows an architecture diagram of the prototype website with the tools and services that will be described in this subsection.

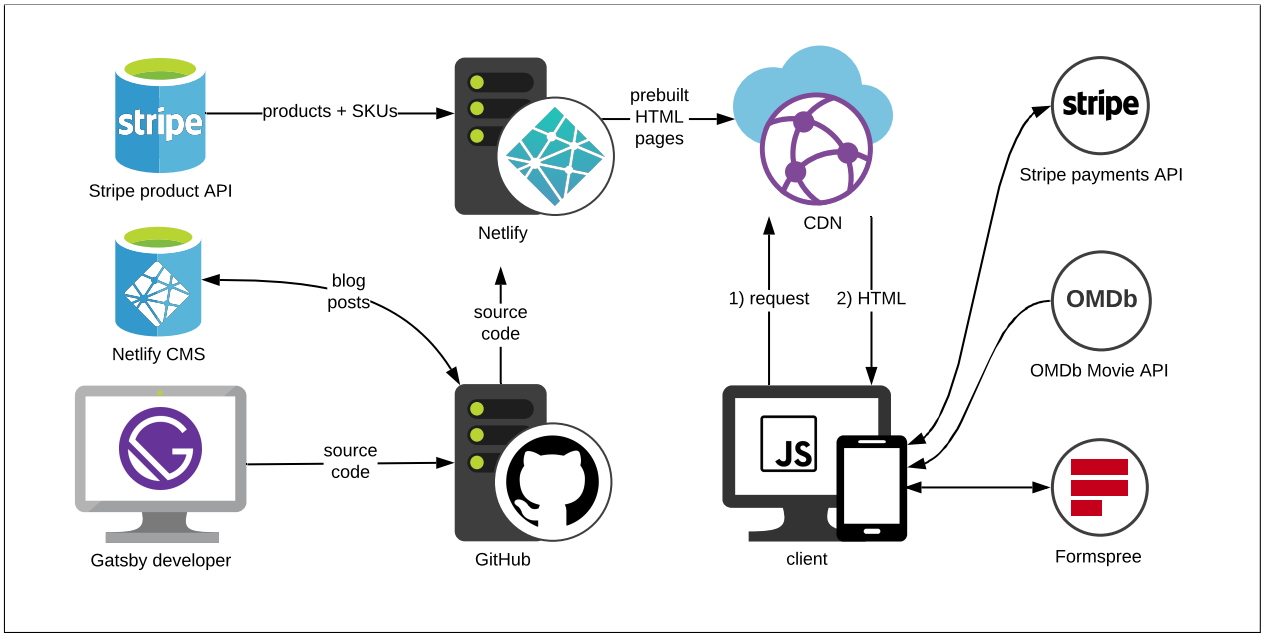

Figure 9: Architecture of the prototype with its different data sources and runtime services

4.3.1 Data Sources and Data Management

Assets, such as images, will be stored directly in the project repository and are integrated with Git version control in the same way as the project code. Shop products will be stored on servers of the payment service provider Stripe. They will be managed using a Stripe-provided GUI (the Stripe dashboard) and pulled into the project code during build time over the Stripe products API [85]. The content dynamically pulled into the site at run time will be media data from www.omdbapi.com (Open Media Database) that can be queried with the help of a search input field. Blog posts will be stored inside the project repository in Markdown files, extended with front matter to allow additional storage of metadata, such as publishing date and tags. The headless, Git-based CMS ”Netlify CMS” will be used to create and manage blog posts.

4.3.2 Back-End Logic

The e-commerce-heavy lifting in the back-end logic of this prototype will be handled by the be-forementioned payment service provider Stripe. The United States based business is a technology company that ”builds economic infrastructure for the internet” [82]. Founded in 2011 in Palo Alto, California, it now has its headquarters located in San Francisco, California, employing over 2000 people in 14 offices world wide. The company claims to serve millions of companies in over 120 countries, including customers like Spotify, Slack, Target, Kickstarter and Shopify [82]. The main

features offered by Stripe are ”Stripe Checkout” and ”Stripe Connect”. Stripe Checkout provides users with a managed payment process available in 14 languages that can easily be connected to any website running JavaScript [88]. Stripe Connect allows customers to build out feature-rich commerce platforms like crowdfunding services, marketplaces or shop builders where Stripe handles the collection of money and payout to third parties [89]. Secondary services offered by Stripe include billing [86], fraud prevention [92] and business data analysis [93].

For the use of this prototype, only Stripe Checkout will be required. Products will be created and managed over the Stripe dashboard, queried over its RESTful product API and displayed within the webshop. The ”buy now” button will redirect to the Stripe-hosted checkout page where customers will input their personal and payment information. The supported payment method will be credit card [91]. When the checkout is finished the customer will be redirected back to the webshop.

The handling of the contact form will be done by Formspree, a form back end and email service for HTML forms. This small service provider enables form integration on websites without the need of personal server-side processes or email servers [16].

4.3.3 Development Environment

The static site generator Gatsby will be used as the central development platform for this prototype. This open-source framework built around React provides a range of features and built-in best practices for creating static websites and is extendable by an ecosystem of plugins [27]. Development of the Gatsby framework was started in May 2015 by American programmer Kyle Mathews [50] who became CEO of the 2018 founded Gatsby Inc. [22]. In 2019 the San Francisco based company raised over 15 Million USD in a series A crowdfunding round [23]. This important budget gives the software project stability by employing core developers while still being a strongly community developed framework that emphasizes its open-source nature and encourages contribution to its repository [25]. On their showcase page Gatsby lists Flamingo, Braun and Airbnb among their users [37].

Gatsby will be used to lay out the user interface of the website using the React framework at its core. Gatsby is the central point of connection for all related data and will integrate the different data sources for this project into its own normalized data layer. To query for data within Gatsby, the API query language GraphQL is used [31]. A second responsibility of Gatsby will be CSS and image optimization and ensuring that images are served to visitors, fit for their screen sizes [35].

Finally, the Gatsby build process will take all source files and create static production-ready HTML files that can be uploaded to a CDN [38].

4.3.4 Publishing

To connect the local development environment to a CDN two main things will need to be in place: a remote repository and hosting provider. For this prototype, GitHub will host the remote repository to make the project-code available over the network (www.github.com). A hosting provider has to be connected, that will pull the projects source code into its own environment, perform the build process and publish the created files to a CDN. In a more traditional approach static resources for a website would be uploaded to a server via FTP, where a web server would take control over handling the distribution of the files. For a JAM stack project however, it is common to use the capabilities of Git and the processes and infrastructure modern hosting platforms provide. This way a fully automated workflow for continuous integration and continuous delivery (CI/CD) can be used with very little manual configuration necessary.

In this project Netlify will be used as a hosting provider (www.netlify.com). Netlify is a company that substantially helped to define the JAM stack paradigms and increase its prominence in web development. Besides offering a platform that provides the functionality for hosting static projects, Netlify is a very active player in the JAM stack community. The company runs a blog that mainly focuses on JAM stack related technologies and workflows [56], maintains popular websites with entry level information about the JAM stack and surrounding technologies [55] [53] [62], published a book about the JAM stack with O'Reilly Media [65] and poses as the main organizer of JAMstack_conf, a conference with events in New York City, London and San Francisco in 2019 and popular personalities of the web development community among their speakers [54]. The San Francisco-based company was founded in 2014 and lists Nike, Atlassian and Samsung among their customers [57]. It provides infrastructure and workflow automation for deploying websites to CDNs and handling certain back-end tasks. The products Netlify offers include "Netlify Build" and "Netlify Edge" that in conjunction provide the beforementioned CI/CD workflow. Moreover, it provides analytics and authentication services as well as handling of forms, large media and serverless functions [60]. A crucial factor for making JAM stack websites work is the way deployment to the CDN is implemented. The process for the developer is described by Netlify as "Develop modularly. Deploy collectively" [59]. This means, while individual, small code changes will be commited to the project repository, the entire project will be published as one unit in an atomic, non-dividable deploy. Either everything will be deployed at once or no deployment will be made at all, avoiding possible mismatches between different deploys in case an interruption would occur.

Instant cache invalidation on the CDN will make sure that there is no stale content from previous deployments available and only the updated content is served [59].

4.4 Implementation

4.4.1 Project Structure and Page Generation

To get a brief overview of the projects source code, figure 10 shows the most important files and folders that are needed to build the main parts of the prototype website.

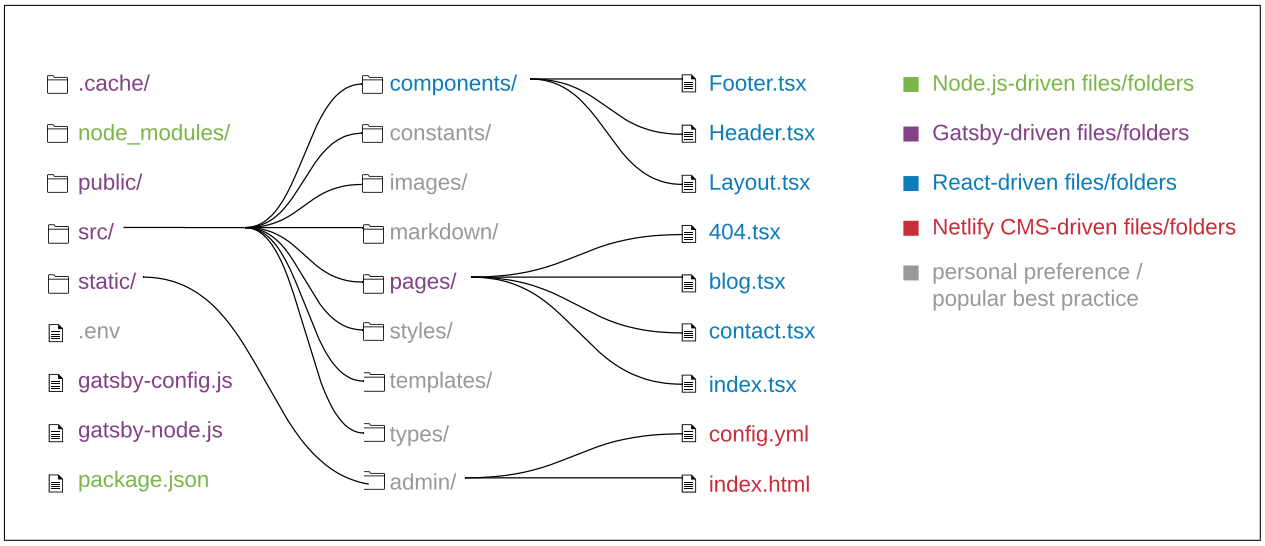

Figure 10: Structure of the most important directories and files in the prototype project repository

Gatsby projects are essentially React projects at their core which in turn are usually set up as Node.js projects. The setup is done with the Node Package Manager (npm) running the command ”`npm init`”. All third-party code (”dependencies”) used for the project is defined in the `package.json` and stored in the `node_modules` folder (figure 10: green elements). To install the Gatsby framework the command ”`npm install gatsby`” is executed, setting up the basic files and folders for a Gatsby project. The basic building blocks for creating the user interface in a Gatsby project are React components (figure 10: blue elements).

To create the core HTML pages for the prototype, files with React components are placed in the folder `./src/pages/`. During the build process the Gatsby framework will pick up these files and render static HTML files. When the build process is finished all pages and assets of the website are saved in the `./public/` folder. For example, the blog page of the prototype has its source in `./src/pages/blog.tsx`. The generated HTML file will be rendered into `./public/blog/`

`index.html` making it available in a browser context at the URL `[www.example.com]/blog`. To view pages during development the command "`gatsby develop`" is executed, hosting the website on the local machine at `http://localhost:8000`.

To flexibly build up the prototype pages, the reusability of React components is used. Recurring elements like header and footer and the general layout wrapper are placed in the `./src/components`-folder. All pages in `./src/pages` that require a header or footer will import these components having them rendered into all respective HTML files likewise. Listing 3 shows a reduced version of the blog page component and how it is using the layout component (listing 4).

4.4.2 Banner Image

After laying out the basic pages, the home page (`index.tsx`) will be filled with content. The first requirement for the home page is a full width responsive banner image. In order use data within Gatsby it has to be pulled into Gatsby's GraphQL data layer. This can be accomplished by installing a source-plugin from the Gatsby plugin library [26]. The physical file for the banner is stored within the project repository on the file system itself. To make it available within Gatsby the plugin `gatsby-source-filesystem` needs to be installed (`npm install gatsby-source-filesystem`). In order to use this plugin, it needs to be referenced in the central configuration file of a Gatsby project, the `gatsby-config.js`, located at the project root. Listing 5 shows the section of the `gatsby-config.js` where the directory `./src/images` is integrated into the Gatsby data layer. This makes all files within the images folder available to be queried in a React component with a GraphQL query. Listing 6 shows the query used to retrieve the banner image. To display this image, the component exported by the Gatsby plugin `gatsby-image` is used (listing 7).

This setup may seem very complicated in order to display a simple image within a website. But looking at what happens behind the scenes will show why this approach can have significant benefits. The problem usually connected to working with images on the web is the optimization for different screen sizes. Serving high resolution images to low resolution screens results in inefficient load times. Long HTML documents may reference many large images that result in elements being pushed around, when the images are loaded. The implementation described above however will solve a number of problems right out of the box. The GraphQL query used to load the banner image will generate multiple files with different image resolutions and file types and store them within the `./public/static` folder. The query used for the prototype outputs a `banner.jpg` with a resolution of 1920x728 pixels at 485 KB file size at the upper end and a `banner.webp` with a resolution of 500x190 pixels and 41 KB file size at the lower end of the spectrum (a total of 6 different banner

images will be generated by Gatsby). Depending on the screen size of the requesting device, only the best fitting file will be sent to be displayed on the website. Moreover the gatsby-image component will reserve a fixed space on the website, avoiding content to be pushed down when the full image is loaded. It further uses a ”blur-up” (or a ”traced-placeholder”) technique, loading a highly compressed version of the image initially to reduce the ”time to first draw” significantly, increasing perceived load speed (figure 11) [29]. This and more processes are done by the Gatsby framework in the background for all images integrated in this way. This includes images from external APIs connected over the network that are queried through Gatsby’s data layer.

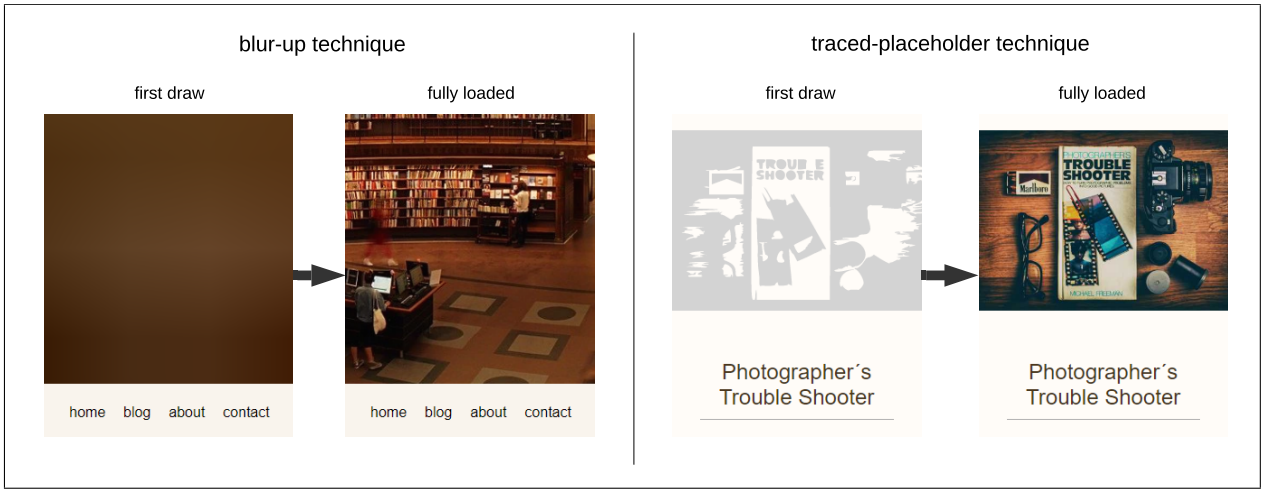

Figure 11: Different techniques used by Gatsby to increase image load speed

4.4.3 Shop Products

As mentioned in the prototype design subsection, all data for the shop products is stored outside of the project repository on servers of the payment service provider Stripe. Corresponding to the approach of pulling data from the file system into Gatsby’s data layer with the usage of `gatsby-source-filesystem`, the way to connect data from Stripe APIs is by using the plugin `gatsby-source-stripe`. (Gatsby plugins are partly community developed and maintained, however a wide variety of popular sources and use cases are covered. The Gatsby plugin library contains 1,537 entries as of the writing of this section.) Listing 8 shows how the Stripe source plugin is included in the `gatsby-config.js`, specifying which object types are needed and the secret key from the Stripe account to connect to. The setup of this project uses an `.env` file at the project root to store environment variables that are kept outside of Git version control. Listing 9 shows an example `.env` file.

Once the Stripe API endpoint is referenced in the `gatsby-config.js` the development server needs to be restarted (”`gatsby develop`”), pulling the data into Gatsby’s GraphQL data layer. Now the products and SKUs can be queried inside react components with GraphQL in the same way as files from the file system. Listing 10 shows the full query to load SKU data into the `index.tsx` page. To be able to create a query like this, it is required to have knowledge about the data structure of the requested objects. This is where the GraphiQL tool that was installed by Gatsby comes into play. GraphiQL is a lightweight GraphQL IDE, developed by the GraphQL Foundation, that can be used to explore Gatsby’s data layer [33]. It provides an interface to build queries and displays the JSON results returned form a query. GraphiQL furthermore shows schema structure documentation for all connected objects in the data layer. With this capability it is possible to work with a third-party API like the SKU API from Stripe without having to consult the API providers documentation directly. Internally, all data that’s added to Gatsby is modeled using nodes. Gatsby automatically infers the structure of these nodes and creates GraphQL schemata [34]. This powerful schema generation is described as ”one of the more complex parts of the Gatsby code base” in the Gatsby documentation, stating that ”as of writing, it accounts for a third of the lines of code in core Gatsby” [36].

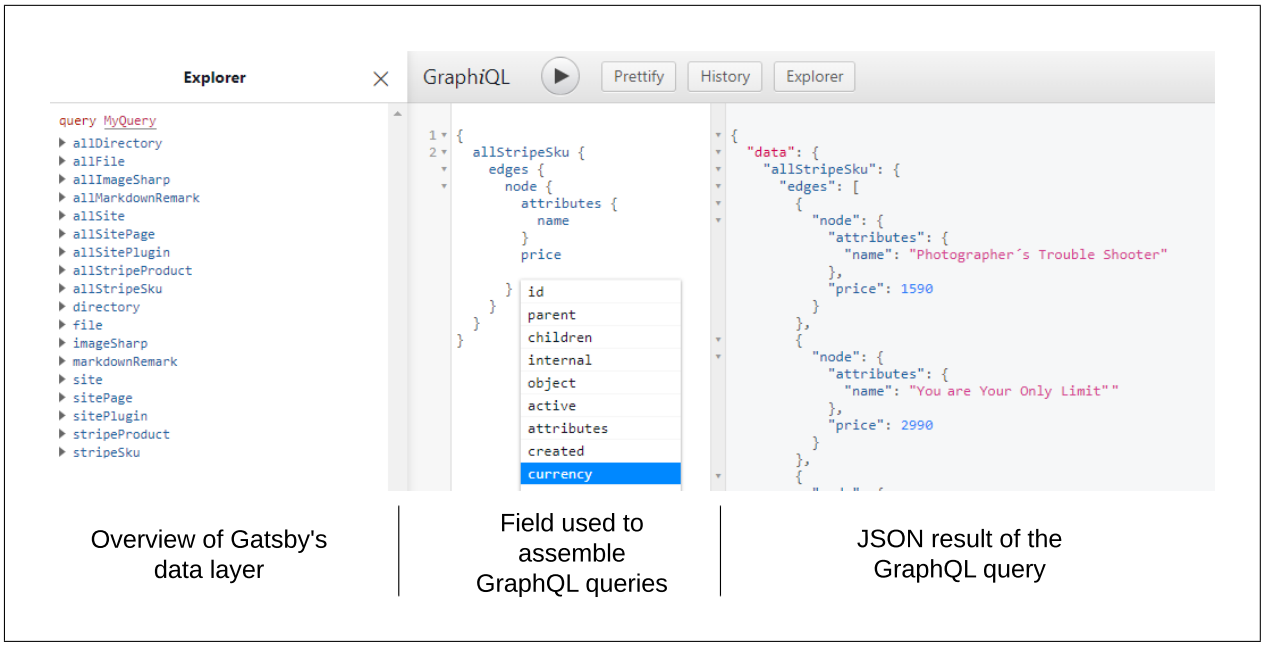

Figure 12: The GraphiQL tool used to explore Gatsby’s GraphQL data layer and assemble queries

Figure 12 shows a screenshot of the GraphiQL interface, creating the query to retrieve Stripe SKUs. GraphiQL provides the user with a context-sensitive auto-complete window showing available fields to help writing queries. This workflow has great advantages when working with APIs. Since the GraphiQL tool is working with live data, the documentation will always be up to date. The API user can ask specifically for the fields needed and leverage GraphQL capabilities like sorting,

filtering and limiting of result data [32]. Being able to view the JSON response right away allows an easy integration into the React components used in this prototype development.

Once the shop products are made visible to the visitor (listing 11) the Stripe Checkout process needs to be connected to be able to handle payments. `Stripe.js`, Stripe's browser-side JavaScript library, offers a straightforward interface to set this up. To include `Stripe.js` into the project codebase the Gatsby plugin `gatsby-plugin-stripe` can be installed: a minimal plugin that will pull the latest version of the library and add it to all pages of the website. Once the Stripe JavaScript object is available in the browser context it can be instantiated with the public key from a Stripe account. Upon click on the "buy now" button of a product the function `redirectToCheckout` will be executed, sending the visitor to the Stripe hosted checkout page of the selected product. Listing 12 shows the implementation of this. The user will be required to fill in personal and credit card information. (Test payments can be carried out using the test card number 4242 4242 4242 4242 with any expiration date in the future and any given verification code.) Stripe will employ its machine-learning tool Radar for every purchase to help prevent fraud by blocking purchases that get classified as not safe by the system. When the payment is found to be valid, Stripe will charge the bank that issued the credit card and redirect the customer back to the shop website. Now the shop owner can work on fulfilling the purchase. Payments received in the shop can be viewed in the Stripe dashboard, however automation processes using plugins or webhooks can be set up to increase efficiency for handling customer purchases [84].

4.4.4 Dynamic Content

To showcase the static and dynamic capabilities of a JAM stack website, content can be displayed in the prototype website dynamically. This is possible due to Gatsby's usage of its underlying React framework, especially the `react-dom` package, providing DOM-specific methods. Static HTML is built using the server-side APIs of ReactDOM while dynamic capabilities, that provide an "app-like" feeling, are kept accessible. Once a Gatsby-generated HTML site is sent to the browser it can be "rehydrated" into a React application by calling the ReactDOM.hydrate method. After this, React takes responsibility of making updates to the DOM, based on changes in data and component properties (this happens in the same way, dynamic behavior is achieved in every React single page application) [39]. Listing 15 shows the implementation of the OMDb search feature, using this process. Whenever an input is made (line 26) the state variable `search` is updated, triggering the `useEffect` method listening to this variable to run. This method will send a GET request with the updated `search` string to the OMDb API and save the fetched result in the `movieResults` variable.

React will then rerender the DOM elements, derived from this variable - in this case rerunning the `movieResults.map()` function, updating the list of movies below the input field.

4.4.5 Blog Posts

To manage the blog posts that are displayed on the blog page of the prototype website, the open-source, Git-based CMS "Netlify CMS" is used. This single page application (SPA) is developed by Netlify and can be integrated into a website by simply adding an index HTML file with a `<script>` tag, that loads the JavaScript bundle for the tool. When working with Netlify as a hosting platform, the authentication for the CMS can be set up in the Netlify admin dashboard of the project (app.netlify.com). This includes user management as well as the setup of a Git gateway that allows the CMS to commit changes to a GitHub repository [58]. An identity widget, handling the authentication on the front end, is provided by Netlify and is also referenced in the SPA `index.html` (see listing 13). To configure the CMS, a `config.yml` is placed next to the HTML file that loads the CMS. (Revisit figure 10 to see both files on the file structure diagram in the `./static/admin` directory. The SPA will be available from https://smbt-gatsby-ecommerce-prototype.netlify.com/admin). The `config.yml` describes Git settings, folder structures and schemata for the content to manage. Listing 14 shows the `config.yml` for the prototype. Whenever a post is edited with the CMS, a commit is made to the project repository, updating the Markdown file that represents the blog post. The hosting provider will get notified of the commit over its build hook, pulls the source code and rebuilds the site. Figure 13 shows a diagram of the Markdown editing workflow with Netlify CMS. The local development environment can be updated with the changed Markdown files by pulling the updated source code from the remote repository ("`git pull`").

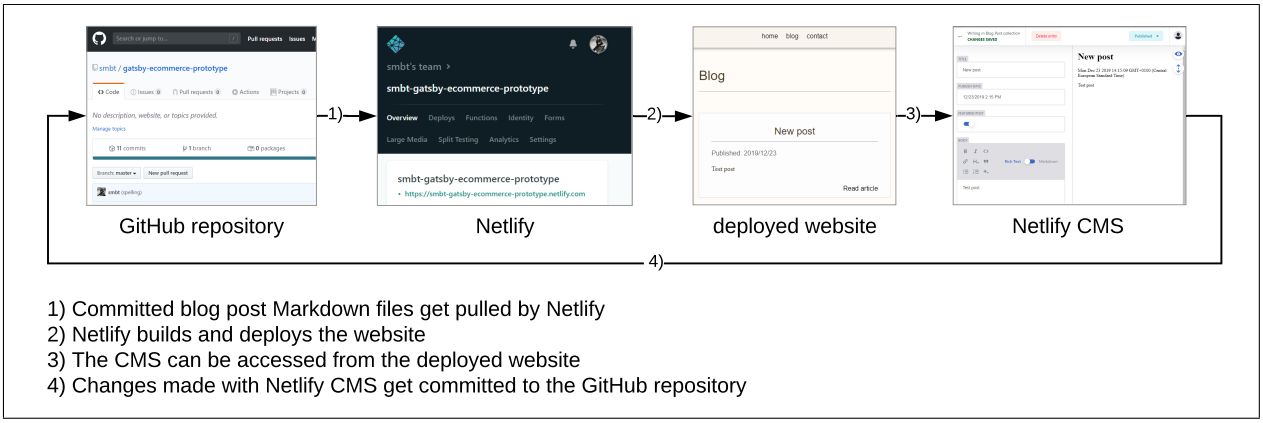

Figure 13: Workflow for editing a blog post

4.4.6 Contact Form

By using Formspree for handling contact form submissions and sending emails a very minimal interface is provided for setting up the needed functionality. After setting up an account with the service provider a standard HTML contact form can be implemented, setting a specific endpoint as the `action` parameter of the form. The information sent by the form will be redirected over Formspree servers where an email is assembled and sent to the email recipient referenced in the `action` parameter. Listing 16 shows a basic implementation for a Formspree contact form.

4.4.7 Publishing

After Git version control is in place in the project repository ("`git init`") the remote GitHub repository needs to be added ("`git remote add origin [reposity-url]`"). Assuming a Netlify account is already registered a "New site from Git" can be added in the personal Netlify dashboard. From there, three steps are required from the developer to get the website live on the CDN (see figure 14). 1) A Git provider needs to be picked. Netlify offers integration with GitHub, GitLab and Bitbucket and will perform the login with the chosen platform. 2) When authentication with the targeted provider was successful a repository can be chosen. 3) Lastly Netlify will try to infer the preferred build settings for the project. In the case of a Gatsby site it will automatically detect "`gatsby build`" as the projects build command and will use the `public` folder as publish directory. After clicking "deploy site" the build and deploy process will be triggered publishing the website to a randomly generated netlify.com-subdomain that can be changed in the "Domain settings" of the Netlify project.

Since the project source code of this prototype relies on environment variables the first build attempt will fail. Environment variables can be added in Netlify's "Build & deploy settings" of the project. Netlify as a platform has a strong focus on usability which reflects in a very clear and easy to understand user interface. The needed settings for this prototype project could be quickly found and set up over the project configuration pages. This includes domain name customization, environment variables and identity functionality used for the Netlify CMS to integrate with the GitHub repository. When the necessary environment variables are set up, a new deploy can be triggered. This time the "`gatsby build`" command will be able to generate all sites, the build will be successful and the website goes live on the CDN.

Additionally to triggering this build manually or by pushing to the project repository, a third way can be set up, using webhooks. In the "Build & deploy settings" of the Netlify project, build hooks

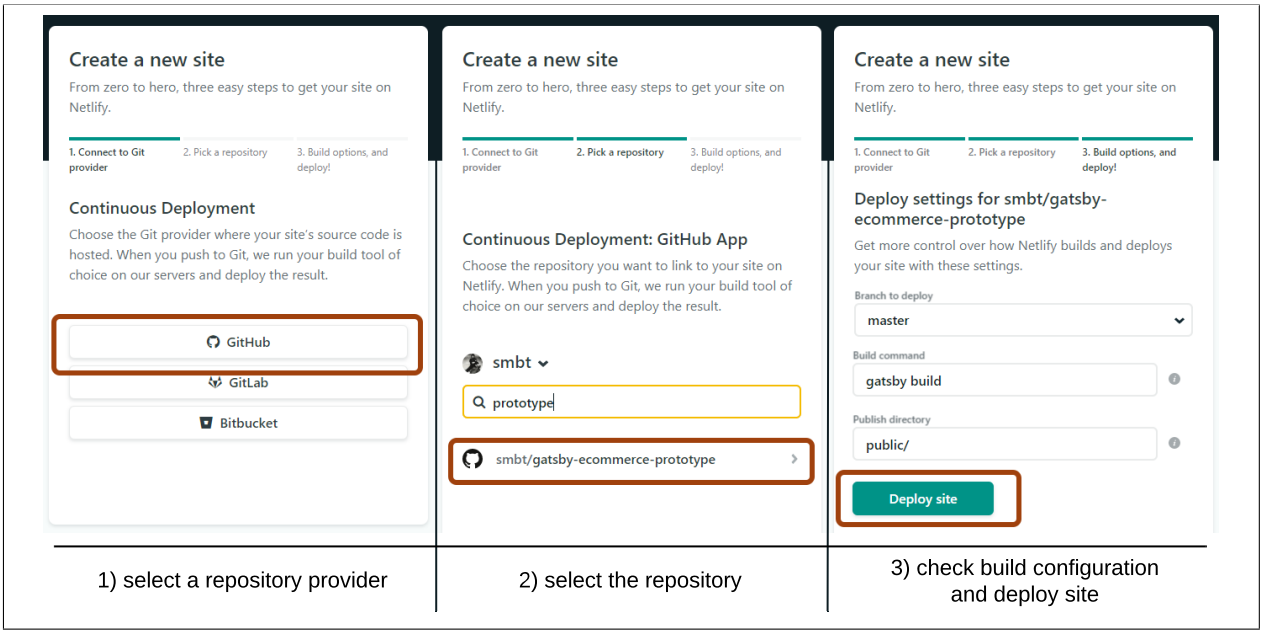

Figure 14: The three steps of adding and deploying a new site with Netlify

can be created in the form of randomly generated API endpoints (for example https://api.netlify.com/build_hooks/5e0f578bc86aff4abc7edf3e). Whenever this hook is called, Netlify will rebuild the website. In a corresponding way, webhook configuration can be set up in the Stripe dashboard, where the shop products are stored. The generated Netlify build hook can be set as a target in the Stripe webhook settings. By specifying the events that will call the webhook (in this case creating/updating/deleting products or SKUs), the build hook is called when product data is changed, notifying Netlify, who will rebuild and redeploy the site reflecting changed product data on the prototype website (figure 7 in section 3.4 showed this process where Stripe can be viewed as the API-driven CMS). Figure 15 shows screenshots of the deployed website.

4.5 Metrics

4.5.1 Speed

In order to measure and compare the global page load speed, the prototype website was hosted on a traditional server infrastructure besides the Netlify hosting. To generate the static assets for all pages the gatsby build process was run locally. The production files were put on a server of the hosting service provider goneo Internet GmbH, where content is served from their data center in Frankfurt am Main, Germany [42].

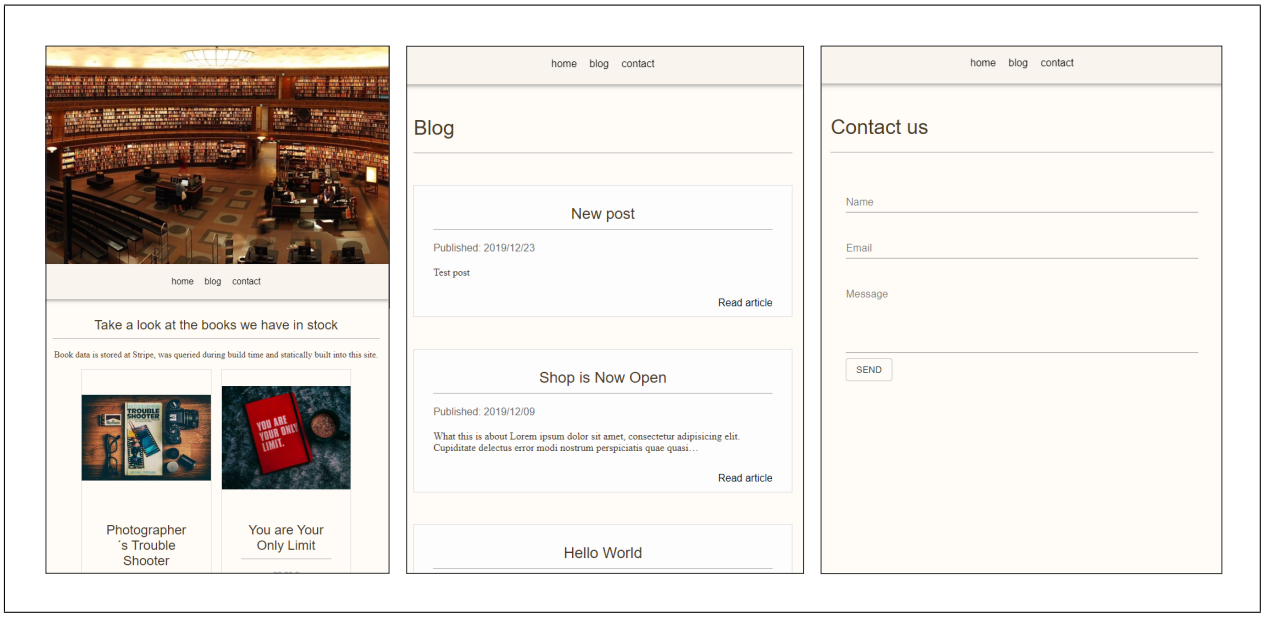

Figure 15: Home page, blog page and contact page of the deployed prototype website

According to a posting from Netlify Head of Support, Chris McCraw, in May 2019, the locations Netlify serves data from are Frankfurt, Singapore, San Francisco, New York, Sao Paolo and Sydney for customers on a ”Regular” plan (this includes the prototype hosting) and additionally Mumbai, Columbus, Des Moines, Tokyo, Dublin and Toronto for customers on an ”Enterprise” plan [51].

To compare the page load speed for both hostings, a website performance testing tool provided by web security company Sucuri Inc., available at https://performance.sucuri.net, was used. This tool allows measurement of page speed from different locations around the world. Figure 16 shows the differences in page load speed from both the Netlify and Goneo hosting of the prototype. The Goneo hosting exceeds Netlify in speed only at their data center location in Frankfurt, suggesting that Goneo's servers are located physically closer to the server that sucuri.net sent their request from. According to this speed test, Goneo can serve the website 285 ms faster than Netlify in this particular scenario in Frankfurt. At locations further away from Frankfurt however, the differences between traditional and CDN-based hosting become more significant. The biggest difference in this small data set is at the location Sao Paulo, Brazil. Due to Netlify's CDN node at this location it can serve the website 1,516 ms faster than the Goneo hosting from Frankfurt.

4.5.2 Cost

For this prototype implementation of a JAM stack website no setup or periodical cost incurred. However the payment service provider Stripe will charge a fee for every processed transaction

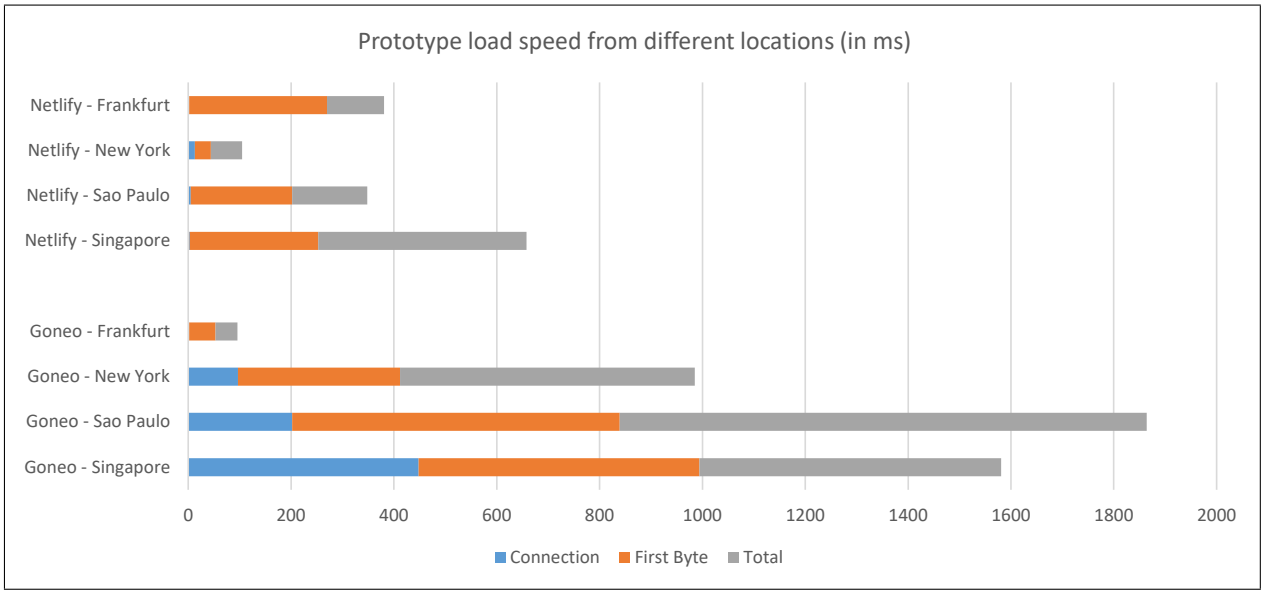

Figure 16: Comparing page load speed for the prototype from different locations, with different hosting providers

outside of the testing mode. Gatsby as an open-source framework is a free to use platform for projects of any size. GitHub and Netlify both provide free accounts with a comparatively wide variety of features, suitable to host small, private projects.

In order to support bigger projects and add more team members to an account, both GitHub and Netlify offer paid plans with a monthly fee. GitHub's smallest paid plan allows to add more than 3 collaborators to repositories and is priced at 7 $ per month [41]. Netlify supports 3 team members at 45 $ per month on its "Pro" plan with a steep increase in pricing for its "Business" plan starting at 1,000 $ per month with full-time support and a 99.99% uptime service level agreement (SLA) among its features [61].

Stripe's payment fees for European cards are 1.4% of the processed transaction + 0.25 € and 2.9% for non-European cards + 0.25€. To calculate an example only European cards are assumed. The small book shop is assumed to sell 50 books every month at the average book price of 18.90€ (945.00€ revenue per month). The monthly Stripe fees would accumulate to 13.23€ (transaction percentage) + 12.50€ (per transaction fix cots) = 25.73€ (2.7% of the shop revenue).

4.5.3 Developer Experience

One of the claims of the JAM stack approach is increased developer productivity and improved overall development experience. So far in the practice of creating the thesis prototype, it has to

be said that, although the JAM stack can be viewed as a comparatively light stack next to the MEAN or LAMP stack, there is still a learning curve that has to be overcome. Some core concepts, significantly different from previous approaches in web development, have to be digested and to start even small projects a set of new workflows, services and tools has to be learned.

The Gatsby framework for example not only requires HTML, CSS and JavaScript knowledge to get development started but at least some familiarity with React, NPM and Node.JS project setup and GraphQL. On top of this, to work productively, theoretical basics of Gatsby processes are required, like the internal data model, the processes of image processing and automated page generation or the interfaces where Gatsby, React and Node.js shake hands to interchange data.

The external services used in a project can have a wide range of enterprise-grade features, abstracted behind their APIs, creating a very broad documentation. This can result in a higher entry barrier, if only a small but individual subset of features are needed, but wide inner processes and workflows have to be understood.

That being said, after the first hurdles are overcome, the JAM stack architecture can open up possibilities that could be much harder to achieve with a different approach. It gives the opportunity to create and safely deploy production-ready full stack applications without much back-end knowledge or expertise in server setup and maintenance. The common best practices used for this prototype support advanced features like deployment pipelines, high delivery speed and application scalability out of the box. Especially the Git-driven workflow benefits make developing and deploying very comfortable. The tools and services involved in the JAM stack ecosystem have a lot of traction and are in rapid development, aiming for a wider feature set and improved usability. The Gatsby framework makes complicated tasks very achievable or even automatic like the process of project setup with webpack and babel to allow writing modern JavaScript. The framework documentation is overall well written and includes many practical tips and tutorials to get work started [30]. Netlify and Stripe too focus on high usability. They provide user interfaces that are clear and intuitive to use and maintain well-organized documentations.

The prototype website heavily relies on third-party software like the Gatsby framework or the internal systems of Stripe and Netlify. It can fairly be stated that the distributed JAM stack architecture is a system strongly based on trust on those third-party solutions. This can be seen as a disadvantage but is a very subjective fact that dependents on personal preference and the nature of a project. For the working on this prototype, it can be stated that, after building some familiarity with the services, companies and the people behind them, it is good to be able to rest knowing they will handle their parts of the project of building, hosting and payment processing properly and reliably.

5 Beyond the Prototype

Section Summary: The prototype in the last section covers very basic functional features. This section will look at options to extend the prototypes functionality and how they can be achieved. Furthermore, different resources will be provided that can be used to explore other available technologies for building JAM stack websites and applications.

5.1 Extending the Prototype

5.1.1 Shopping cart functionality

So far, in the basic implementation of the prototype it is only possible to buy single products and go through their checkout process individually. Having a shopping cart feature available would be a better approach and would allow customers to buy a number of products together. However, there is a basic problem when implementing a shopping cart on a website using the JAM stack architecture: persisting state between pages. Since a static website uses single HTML documents as their most basic UI elements, there is no global application layer above the pages. Adding a product to the cart on the home page would result in the loss of that information when switching pages.

To be able to apply a global state layer, the Gatsby framework offers APIs like the `wrapRootElement` method that allows plugins to wrap around an application for setting persistent UI elements between pages [40]. But an easier and more commonly applied approach is to use the `localStorage` API present in all modern web browsers. This browser-based object storage can be used to store and retrieve data across browser pages and sessions, keeping the data available even after the browser was closed. The `localStorage` interface stores key value pairs. While only strings can be saved as values, it is possible to store nested JavaScript objects and arrays, using the native JSON API. To store a list of items as a string, the JavaScript array can be processed with `JSON.stringify` and converted back using `JSON.parse`. Listing 17 shows a basic reference implementation of storing and retrieving product data with the help of `localStorage`.

A live example of this approach can be seen at the ”Gatsby starter” for e-commerce - a basic implementation of a Gatsby website (in this case an e-commerce website) for bootstrapping Gatsby projects: parmsang.github.io/gatsby-starter-ecommerce [72].

5.1.2 On page search

With a shopping cart in place, bigger shop projects can be realized containing a wide product range with categories spanning over many different pages. With a more complex setup, it will become more difficult for a customer to find a desired product. At this point, an on page search functionality becomes important to help visitors explore the shop offers. While a search feature, showing products filtered by product name, could be sufficient for simple projects and fairly trivial to implement, the task will get more complicated when relevance, fuzzy search or product popularity come into play or complex sorting and filtering needs to be handled. Platforms with highly sophisticated search engines like Google search or Amazon set a high standard for the usability of search that can challenge individual projects to keep up with, in order to stay competitive.

To solve this, Algolia, a third-party API service, can be employed to achieve high-level search functionality. Algolia is a hosted search as a service provider that specializes in consumer grade search, focusing mainly on search speed and relevance of the results [3]. To set searching up, Algolia needs to be provided with the data sets that need to be searched. Data can be sent to Algolias servers using their API clients or its dashboard and can be kept in sync by periodical pushes or by pushing new data whenever it has changed [5]. The data will be indexed by Algolia to allow performant querying and can be accessed from a websites front end by sending queries in the form of search terms. The search client, that sent the query, will receive a paginated array of relevant results, that can be displayed in the web browser (see figure 17; listing 21 shows an example response from Algolia). Algolia offers the integration of a complex and optimized search process with high availability. The searchable client data is hosted on multiple servers to maximize back-end up time and is served over a distributed search network to reduce network latency by connecting the client to a node closest to his location [4]. A live Algolia search can be seen at www.smashingmagazine.com.

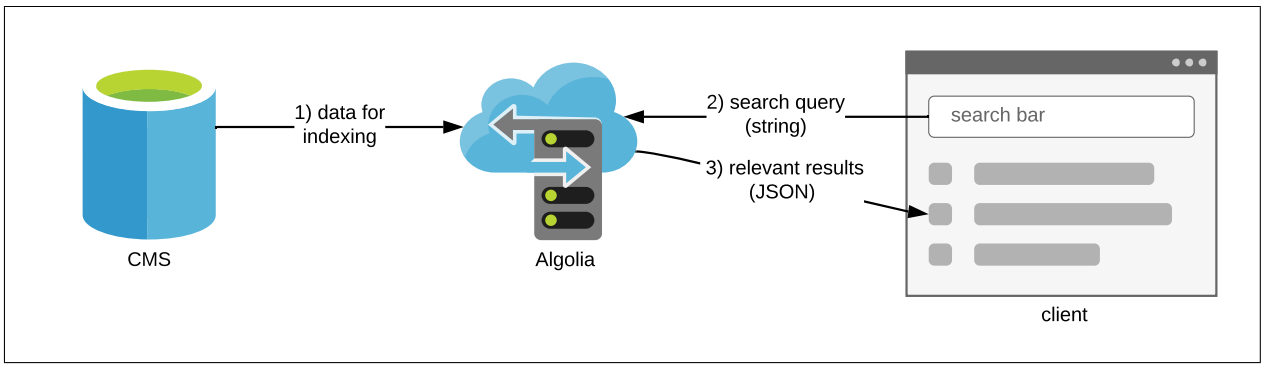

Figure 17: On page search integration with Algolia: the data flow

5.1.3 Process Automation With Webhooks

Managing and fulfilling successful purchases manually over the Stripe dashboard can become impractical at scale. With the help of webhooks, automation processes can be set up to help handle repetitive tasks efficiently. In the same way internal events get fired by the Stripe system when product data was changed, a "`checkout.session.completed`" event is triggered whenever a customer was charged successfully. Instead of connecting a webhook endpoint that will trigger a rebuild of the page, an endpoint can be set up in a custom back-end environment that can invoke a variety of subsequent processes. Stripes internal event system will send checkout session data like client-id and purchased items as webhook payload along to the specified endpoint [84] [94].

Digital products like ebooks for example can be automatically delivered via email. For paperback books, a delivery service API can be connected to automatically print pre-purchased shipping labels. If, besides Stripe, other payment service gateways like PayPal are connected, orders from Stripe can be fed into a layer above Stripe and PayPal where a unified order management is handled.

5.1.4 Reduction of Stripe Coupling

In the implementation of the prototype, Stripe not only handles the payment process but also stores the products and hosts the checkout form where customers enter their personal and credit card information. This setup allows a minimal integration with Stripe, where only client-side code has to be executed in the web browser, while Stripe manages the checkout session. This however creates a stronger coupling to Stripes services than necessary. By managing checkout session and products in custom systems a more flexible set up can be achieved.

Instead of querying product data from the Stripe API, they can be stored in the project repository in markdown files similar to blog posts. Name, price, currency and other needed data can be stored within the front matter section of the Markdown files and content management can be handled via Netlify CMS, alongside the blog posts. Storing product data outside of Stripe gives the ability to implement dynamic price calculation and enables features like sales or discounts.

To avoid having to redirect customers to the Stripe hosted checkout form, a custom checkout process can be implemented. To realize this, Stripe provides small input elements that can be integrated into a website. These elements handle input of credit card information and submit it to Stripe, preventing sensitive information from touching personal systems [83]. This will create a more professional looking checkout flow while maintaining high security and reliability of the process.

5.2 Exploring the (JAM Stack) Ecosystem

5.2.1 Picking the right tools

The range of technologies that can be used for developing modern web applications contains a great variety of different, yet often similar solutions. Technologies relevant for JAM stack projects range from static site generators, over content management tooling to hosting providers and include a wide ecosystem of API providers with specialized, feature-rich functionalities. In order to implement the prototype for this thesis, a combination of tooling and services was used consisting of Gatsby, GitHub, Netlify, Stripe and others. Most of these building blocks could be exchanged by similar technologies without significantly changing the final product. However all platforms, tools and frameworks have their individual strengths and weaknesses. The choice of tooling should be made cautiously and in the end will not be unaffected by personal preference.

Comparing a variety of solutions and picking the best fit is a difficult task, as insights are necessary into technologies that are under active and often rapid development. In a 2019 blog post, published on www.medium.com, the situation is described as ”a big mess [where] selection of a technology to get the job done on a particular project has become more time consuming than to actually develop the project” [76] - probably over exaggerating but pointing out a fact with some truth to it. Increasing performance and a broadening feature range can blur the differences between two options and technological weaknesses existent a year earlier may not be relevant anymore. Blog articles listing pros and cons of new technologies can get out of date quickly and research should be conducted thoroughly, using a variety of sources, including up to date product documentation. When enough time is available, building small test projects should be considered to evaluate usability of a framework or service. In the end not only feature range will be a crucial factor, but also a high quality documentation and the individual learning curve for getting up and running with new tooling.

As an entry point for finding and comparing available JAM stack related technologies, a few frequently referenced web resources will be shown in the following, that hopefully endure longer than the average blog post and remain relevant and up to date in the upcoming years. All three resources are websites that are community maintained. (In the JAM stack way - they are sourced from GitHub repositories, using Markdown files as their data base. Subsection 6.1 in the case studies section explains more in detail how this process of collaboration works.)

5.2.2 Static Site Generators

The website www.staticgen.com contains a comprehensive overview of available SSGs. It lists a number of 279 platforms, as of the writing of this section, even though comparatively few of them will possess mainstream relevance. The entries can be filtered by programming language (JavaScript, C#, Ruby, ...), template engine (Angular, React, Vue, ...) and license (closed- or open-source). Sorting can be done using GitHub stars or Twitter followers to provide basic information about popularity. Where a GitHub repository is available a trend based insight over time can be obtained by using a project called ”GitHub star history” hosted at https://star-history.t9t.io. All of the numbers should only be used as rough indicators for the popularity of the frameworks. Staticgen.com is developed and hosted by Netlify Inc.

Website: https://www.staticgen.com
Source Repository: https://github.com/netlify/staticgen

5.2.3 Content Management Systems

A similar resource to the previous is www.headlesscms.org. It lists headless content management systems and again uses GitHub stars or Twitter followers as popularity indicator for sorting. Entries can be filtered by the CMS type (API-driven or Git-based) and license (open- or closed-source). Headlesscms.org is also developed and hosted by Netlify Inc.

Website: https://headlesscms.org
Source Repository: https://github.com/netlify/headlesscms.org

5.2.4 All Things Serverless

Css-tricks.com, a well-known source for web development content, hosts a manifold collection of technologies for the serverless web ecosystem at its subdomain https://serverless.css-tricks.com. The website is divided into ”services”, ”resources” and ”ideas”, whereas the ”services” part is most relevant to this section. There is an overlap to the beforementioned resources but the listings contain a much wider variety of technologies including hosting, e-commerce, data storage and search providers. The collections may not be as comprehensive in numbers but focus more on the most important players of each field and serve as a useful point of entry for conducting deeper

research on available tooling options. Serverless.css-tricks.com is developed by css-tricks.com founder Chris Coyier.

Website: https://serverless.css-tricks.com/services/major
Source Repository: https://github.com/CSS-Tricks/serverless

5.2.5 Staying up to Date on the JAM Stack

As mentioned above - information about technologies in a rapidly evolving ecosystem can become outdated quickly. To be able to stay in touch with current innovations in a certain field, good blog platforms with regular content on the topic can be useful resources. Following a short number of blogs, centering mainly around the JAM stack architecture:

Gatsby (static site generator): www.gatsbyjs.org/blog
Netlify (hosting provider): https://www.netlify.com/blog
Snipcart (e-commerce service provider): https://snipcart.com/blog/categories/jamstack
Bejamas (web development agency): https://bejamas.io/blog/jamstack-universe

6 Case Studies

Section Summary: To showcase some of the bigger projects that are achievable using the JAM stack architecture, the following section will describe two platforms and how they work. It will show how the project at serverless.css-tricks.com achieves content collaboration through JAM stack workflows and how a popular online magazine - smashingmagazine.com - has build its entire web platform on the JAM stack.

6.1 CSS-Tricks: Transparent Content Collaboration

The website https://serverless.css-tricks.com, hosting collections of different serverless technologies, was mentioned in the last section and described as a GitHub-sourced and community-maintained resource. This subsection will describe its architecture and how collaboration on the project is achieved.

Similar to the thesis prototype, the website is generated with the Gatsby framework. It is hosted on GitHub and deployed over Netlify [12]. Another overlap in technologies is the usage of the Netlify CMS, that integrates easily with GitHub and the Netlify hosting platform. As is the case for all Gatsby projects, the layout of the website is realized using React components while the content itself is stored in Markdown and JSON files. Listing 18 shows the `gatsby.md` Markdown file, rendering the entry for Gatsby in the SSG-section of the website and listing 19 shows the corresponding schema definition in the `config.yml` file for the CMS. The live entry can be seen at serverless.css-tricks.com/services/ssgs.

The access to collaboration on this project is achieved with a link titled ”edit this”, placed on all collection entries. The links will open the Netlify CMS editor, asking the visitor to login with a GitHub account in order to proceed. To be able to edit an entry (a Markdown file, saved in a another persons GitHub repository), Netlify CMS will fork the repository in the background, creating a full copy of the project in the visitors GitHub account. The CMS instance will connect to the visitors copy of the repository where changes can be applied.

When the visual editor has opened the `gatsby.md` Markdown file, changes can be applied and saved, creating a commit in the personal copy of the repository. Since the fork of the original repository was tracked by GitHub, a pull request can be opened, attempting to merge the applied changes back into the CSS-Tricks repository. This pull request can be reviewed and either accepted

or declined by the maintainer of the original repository. Figure 18 shows screenshots of the GitHub user interface for creating a pull request in the described scenario.

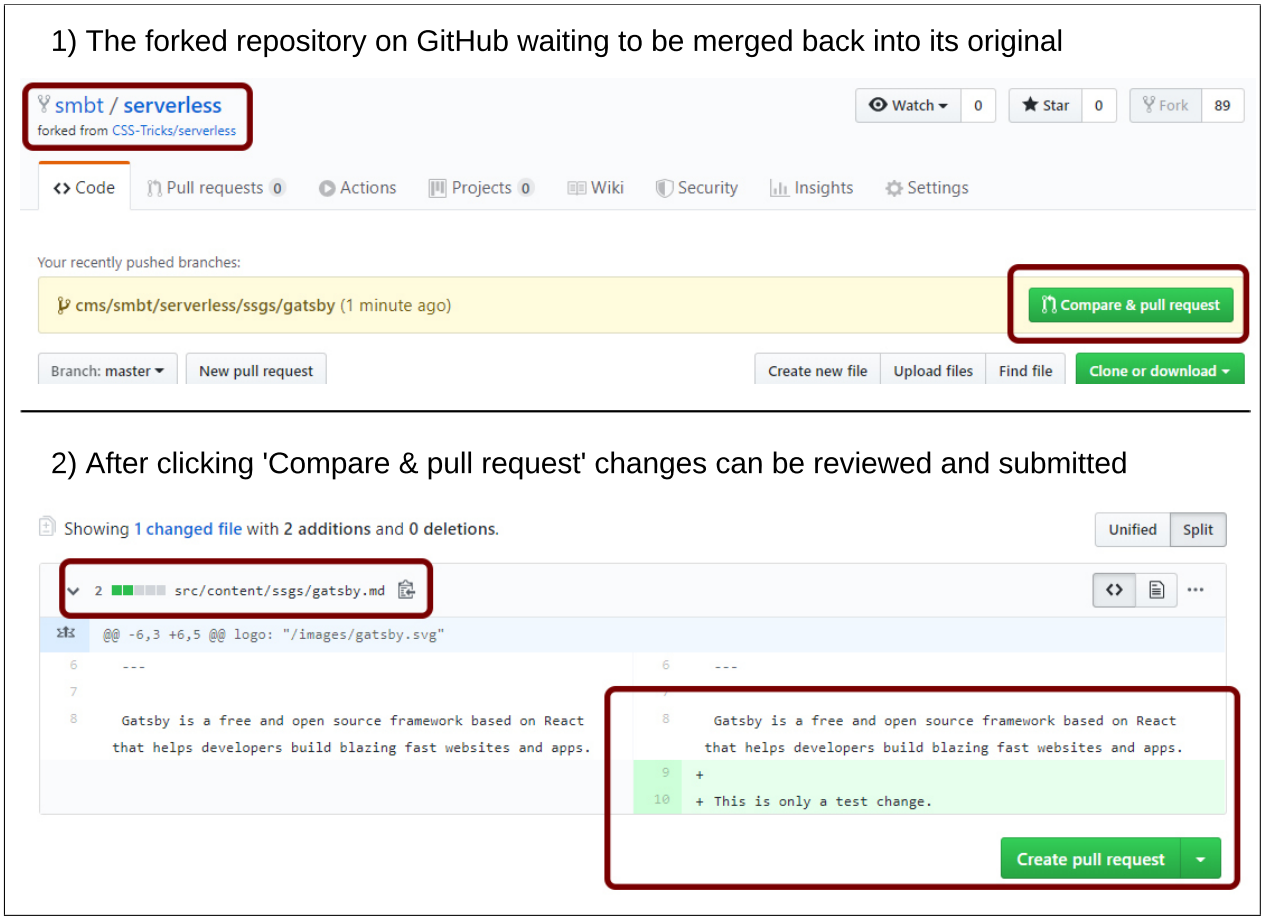

Figure 18: GitHub screenshots showing how changes to a forked repository can be applied

When the pull request was accepted, a merge commit will be applied, updating the original repository. Assuming the very likely case that this repository is connected to its hosting platform in the same way, the thesis prototype is, Netlify will be notified of this repository update an will start to rebuild and deploy the site with the changed markdown file.

The way, modern web technologies and Git-driven workflows are used at www.css-tricks.com shows, how easily collaboration can be achieved on a web project with the JAM stack approach. So far, the process of working directly with GitHub repositories, code commits and pull requests is very developer oriented. However, by improving automation and abstracting the tooling even further away from the source code, this workflow can be made more accessible for non-technical users. All changes made and the person committing them are tracked and visible in the Git version history. Drawing a parallel to the crowdsourced encyclopedia project Wikipedia, JAM stack architecture

and workflows can therefore open the door for a wider variety of high-quality, community-driven web content projects in the future.

6.2 Smashing Magazine: The JAM Stack at Scale

Smashing Magazine is a web platform and ebook publisher with content for web design and web development. It was founded in 2006 as part of Smashing Media AG and serves as an independent source of practical articles with 3,000,000 monthly page views, according to their own metrics [79]. The online platform of Smashing Magazine consists mainly of 4 systems. (1) The main website where authors can publish articles and readers can write comments, (2) an online shop system for selling ebooks, (3) a job board where companies can post job offerings and (4) a smaller system used to inform about the Smashing Conference on www.smashingconf.com.

During the course of over 10 years of operation, www.smashingmagazine.com grew from a basic WordPress blog to a large online presence. In 2017, the magazine decided to rewrite their entire platform because the running systems could not keep up with the demands of the grown audience [9, p. 69]. They decided to undergo a switch from an environment consisting of different monolithic systems to a JAM stack architecture. This switch was achieved in cooperation with Netlify engineers who documented the process in their ebook ”Modern Web Development on the JAM-stack”. This subsection leans on the documentation provided by this ebook which is mainly quoted here.

The old platform consisted of a WordPress blog for the article publication, a Shopify installation for the online shop, a Ruby on Rails app used to power the job board and a smaller system serving the Smashing Conference site based on the flat-file CMS Kirby running on a PHP stack separate from the WordPress installation [9, p. 70].

All four of these systems had their individual front ends. This meant, Smashing Magazine had to build and maintain four implementations of their corporate design separately. Moreover, when an article published on Smashing Magazine went viral, the WordPress systems overloaded and could not handle all incoming requests properly anymore, even though different caching plugins and libraries were installed in the systems. ”Maintaining and developing every platform separately was expensive and time-consuming”, Smashing Magazine wrote in an article on the topic [19]. The WordPress platform with all its plugins and extensions became so complicated that making changes always posed the risk of breaking different parts of the system [9, p. 71].

Their goal was to create a better and uniformed templating process with only one implementation of their design in one repository. Performance and reliability had to be increased, to keep the systems from breaking down under heavy loads [9, p. 71]. The JAM stack architecture posed a very promising approach for realizing the desired goals.

The new system had to be able to build out tens of thousands of pages for articles, products and job offerings on every build. The static site generator Hugo was chosen for this task, giving its speed in page generation (table 1). Netlify CMS was used to manage the different content types like articles, authors, products and job postings. They are saved in the project repository in Markdown files with front matter and picked up by Hugo to build out the static front end. Interactive functionality like signup, login or the checkout process is managed, using the React-compatible mini front-end framework Preact in conjunction with Redux, a popular JavaScript state management library [9, p. 81]. When actions like form submission or order confirmation are taken, an AJAX call is made to a microservice handling the interaction [9, p. 75].

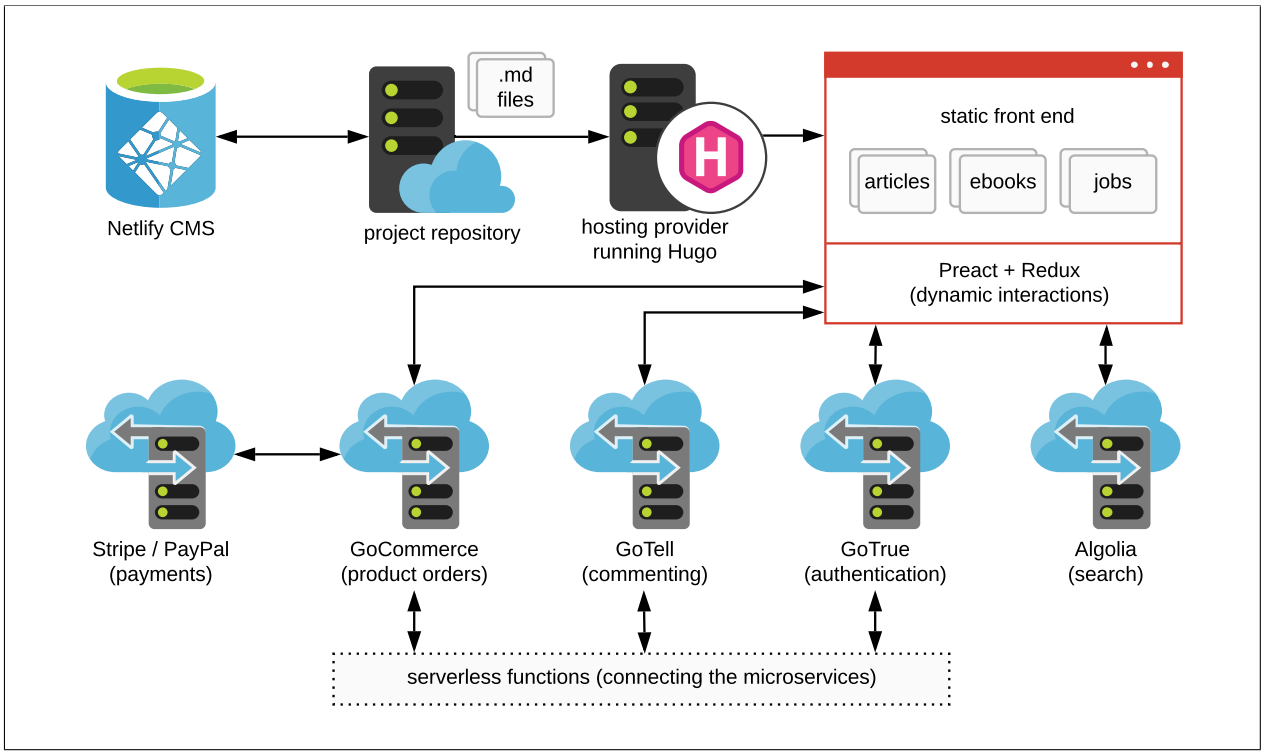

Figure 19: Overview of the Smashing Magazine web platform after their switch to the JAM stack

To enable site search Algolia was integrated, pushing all content to its search index API every time the site was built. In the process of the relaunch, a number of small open-source microservices were developed by Netlify for Smashing Magazine: GoTell is used for handling comments, GoTrue for authentication functionality and GoCommerce serves as a minimal shop back end, man-

aging communication between front end and payment services like Stripe or PayPal. To provide GoCommerce with product information during runtime, AJAX calls are made to the actual HTML detail page of a product where a script tag is placed by Hugo with JSON metadata that was sourced from the Markdown front matter (listing 20 shows an example JSON object, that is injected into every product detail page in that manner). Serverless functions are used to connect the individual services to each other, where needed. ”The total amount of custom, server-side code for this entire project consisted of 575 lines of Java-Script across 3 small AWS Lambda functions”, Netlify stated in their project documentation [9, p. 108].

More in-depth information about this project spanning over 40 pages can be found in the before-mentioned ebook provided on the Netlify website at www.netlify.com/oreilly-jamstack.

Vitaly Friedman, co-founder of Smashing Magazine was quoted in an interview, stating that the ”time to first load [of the website] went from 800 milliseconds to 80 milliseconds” [21]. Smashing Magazine reported a smoother user experience due to the easier integrations, speed, and better performance [8]. Having the control to freely merge different parts of their platform, Smashing Magazine integrated shop products throughout their website, allowing to better promote their own shop and in turn removing external advertising almost entirely [19].

7 Discussion

Section Summary: The JAM stack approach promises certain benefits for web developers and visitors but also has its boundaries. This section will look at advantages, that JAM stack projects provide and will show disadvantages and limitations in the current state of development.

7.1 Advantages

Separation of concerns is probably one of the most important architectural patterns in JAM stack projects, that provide a number of advantages further down the road.

Separating back-end tasks into encapsulated (micro-)services draws clearer borders between different concerns of a project, making it easier to comprehend it as a whole while also allowing easier maintenance [9, p. 18]. Complexity can get outsourced to third-party services, leveraging the expertise of companies who focus exclusively on a single problem domain [9, p. 19]. By providing rich features behind highly-abstracted APIs those service providers can cater their products to a wide range of customers, allowing to scale efficiently and keeping products and services affordable (especially when comparing the cost of custom implementations).

The short request-response cycle on the JAM stack provides global delivery speed, difficult to meet using other web architecture approaches. Time is money, in the modern days of the web, and high performance in tooling and delivery is crucial to meeting the high standards expected by most people and - in the end - search engines. The website www.wpostats.com provides reports and case studies of websites and companies, describing their impact of web performance improvements on business metrics. From a 2018 entry: ”Furnspace [an online furniture retailer] reduced their image payload by 86% resulting in a reduction in load time of 65%. This improved user experience helped double Furnspace’s eCommerce purchase conversion ratio, cut bounce rates by 20%, increase mobile revenue by 7% and dramatically improve SEO” [74]. Less privately maintained server infrastructure also results in less needed specialization and therefore lower developer cost in JAM stack projects.

Their distributed hosting infrastructure allows JAM stack websites to scale efficiently. Spikes in website requests can be handled by cloud service providers, by load-balancing requests to other servers or CDN nodes [9, p. 23]. The otherwise difficult and expensive task of scaling, to meet unexpected high amounts of traffic, is baked into the JAM stack architecture.

Separating the build process from the runtime highly decreases the amount of code, that has to be executed, to serve and run a website or web application. This reduces potential points of failure as well as lowering the attack surface for possible exploits [9, p. 30]. By handling tasks with the help of third-party API providers the responsibility of maintaining infrastructure, securing processes and complying to regulations can be offloaded to specialized companies, who's business models depend on providing stable and secure services. App development teams can focus more on their core objective: creating customer applications.

The trend in less back-end-focusing technology stacks can even be witnessed by looking at propositions of purely front-end stacks like the STAR stack, described by Shawn Wang in a css-tricks.com article. He describes a set of widely used front-end tools consisting of Design *Systems, TypeScript*, *Apollo* and *React* [97].

7.2 Disadvantages and Limitations

The element of trust in third-party solutions was mentioned before, at the end of the prototype section (4.5.3). This can be seen as a major hindrance in choosing to build on this type of architecture, especially when personal or customer data has to go through foreign systems. Data privacy is an important issue when building web-based applications, as laws and regulations have to be followed like the European GDPR (General Data Protection Regulation) or PCI standards (Payment Card Industry Data Security Standards). The application developer has to ensure that no connected third-party systems lack control of their data and comply to these rules. External service providers often provide information about their data privacy standards but careful research needs to be conducted and specialized legal consultation should be considered, if necessary. Exemplary pages about security and data privacy for service providers of the thesis prototype are www.github.com/security, www.netlify.com/security and www.stripe.com/privacy.

Staying in the area of third-party services and tools, choosing the right solution for a project can be challenging. It is often difficult and time consuming to sufficiently understand how a solutions works and to make out their strengths and weaknesses. Research and comparison of services is key and for important projects only established or capital backed service providers may be considered to ensure steady, long-term support. Innovative technology is inherently subject to fast paced development cycles. During the time of work on this thesis alone, changes in third-party features, documentation and user interfaces could be witnessed. API endpoints went deprecated or did not keep up with changed regulatory requirements anymore [90] [87]. In order to be prepared for

change, thorough understanding of a service and the surrounding ecosystem is required and staying in touch with current development may be critical.

With a network of services connected to an application, another difficulty that arises is cost control. Having a variety of different, fixed and variable cost positions to keep track of can become a challenging task. For example, adding PayPal as a payment gateway to the thesis prototype may result in the integration of the service Snipcart, that supports both credit card and PayPal payments (and in turn also uses Stripe among other payment gateway under the hood [80]). Snipcart charges a percentage of every transaction *including* the fees of the underlying payment gateway [81]. User-triggered service requests, depending on other services, triggering side effects served by yet another services can create an opaque network of pending fees, all charged by different service providers. (Figure 20 shows a graphical representation, derived from a dotnetpro article by Alexander Schulze, who also proposed a solution to this problem in his article [73].)

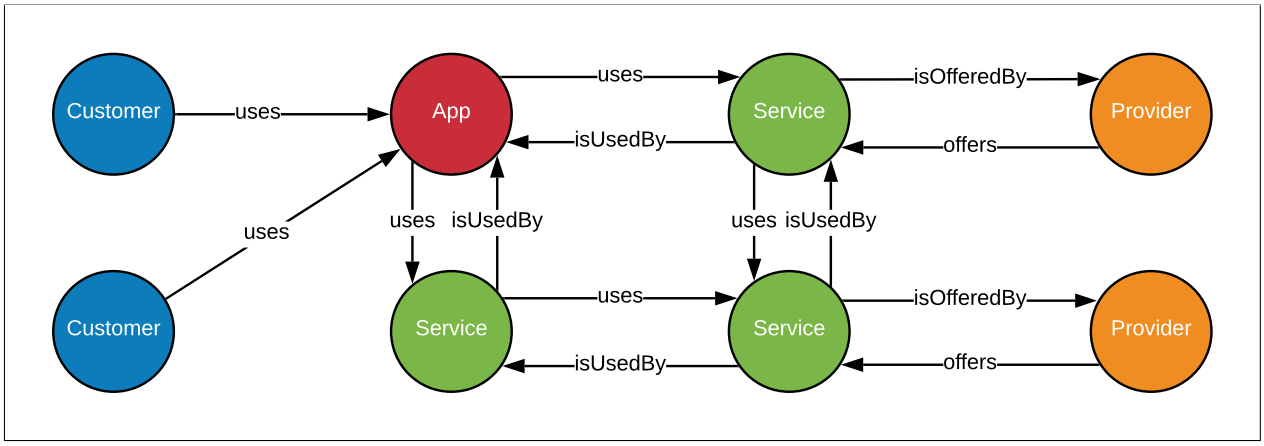

Figure 20: A semantic graph of cloud services and their dependencies [73]

On the more technical side of the JAM stack approach, other factors will come into play that can create boundaries. WordPress, as a LAMP stack based system, gives developers as well as non-technical users the opportunity to create websites and publish content. Its ecosystem of plugins, page builders and graphical theme editors gives people without programming knowledge a wide range of control over their platform. So far, the tooling around JAM stack technology is in its early stage and mostly requires some amount of technical understanding of languages and workflows in order to start building. Git-driven, headless content management systems are still young and may lack customization abilities or proper role management. Additionally, in order to sell a JAM stack project to a customer or other stakeholder, some learning curve and adaption barrier towards the new tooling and architecture might have to be overcome [66].

As the prototype website and the Smashing Magazine case study shows, websites with changes a few times a day or even every few minutes (depending on the duration of the build process) may be well suited to run on the JAM stack architecture. As of the current state of technology however sites or applications with highly dynamic content or websites with millions of pages, like newspaper websites, challenge the concept of statically pre-building an entire platform [20]. Hybrid systems of JAM and MEAN stacks could be suitable for dynamic applications, pre-building as much as possible while still fetching all dynamic parts during runtime. Huge sites can be generated using incremental builds, where only changed parts get regenerated. Those use cases however can open up difficult situations where standard workflows can easily fall apart due to inconsistencies between dynamic and static content or newer and older generated parts of a website [48].

8 Conclusion

In this thesis, we set out to investigate the JAM stack as an approach for building modern websites and web applications. We explored different web development architectures, to gain a perspective on the problems, JAM stack technology aims to solve and looked at the tools and processes used to solve them. By showing the steps necessary to develop and deploy an e-commerce prototype website, we could see that, without much optimization or server-side knowledge, a fast, scalable application could be built, that can handle advanced back-end tasks without a dedicated server environment.

By describing additional features for the prototype and looking at web resources and technology collections, insights were given into the possibilities, the JAM stack ecosystem offers for building modern web applications. It was shown how the use of new workflows can enable collaboration on websites and that the JAM stack architecture is not only suitable for small websites, but can also support major platforms with a wide range of functionalities.

With the help of modern web browsers, global delivery networks and third-party software solutions, a transition could be made from strongly-coupled, server-centered monoliths to decentralized, client-focused applications. By abstracting repetitive back-end tasks, a shift towards front-end technology can be witnessed that gives small teams or individual developers the ability to create feature-rich full-stack applications with high performance, availability and security.

The ecosystem surrounding the JAM stack and overall modern web technologies is in a state of rapid development cycles where high competition breeds high innovation. This both benefits and challenges JAM stack projects, as developers need to follow the trends and changes to stay up to date and keep the individual parts of their systems in sync.

Although the reduced project overhead of setting up and managing server environments promises great improvements in development productivity, there are important factors that have to be taken into consideration before making the switch to a JAM stack architecture. Especially creating platforms for users in the European Union, while consuming software services from mostly United States-based companies can increase the risk of losing control over customer data. As with all innovative technology, small, cautious steps should be taken, to avoid running into unexpected problems. By following the developments of service providers, application developers and the ecosystem as a whole, new possibilities, use cases and best practice solutions will emerge.

A Glossary

AJAX Asynchronous JavaScript + XML: set of web development techniques to dynamically load content into an application without full page refreshes. 40

CDN Content Delivery Network: regionally distributed network of servers for delivering web content connecting an end user to a network node closest to his physical location to reduce network latency. 16, 19

CI/CD Continuous Integration / Continuous Delivery: practice of incrementally integrating small code changes into a central code base and deploying these changes on a short-term basis rather than having only major software versions delivered to the end user. 12, 19

content management system Software used manage the creation, storage and editing of data used in a web project.. 8, 10, 35

conversion rate The percentage of visitors to a website who take a desired action like buying a product or subscribing to a newsletter. 16

DevOps Set of practices used to shorten the development life cycle of a project like code quality control or deployment automation. 5

DOM Document Object Model: API for HTML documents defining a logical structure of elements that can be dynamically manipulated by JavaScript. 6, 24

front matter Syntax used to store variables and metadata within Markdown files. 11, 17, 33, 40, 41

Git A distributed version-control system for tracking changes in source code during software development. 8, 19

GraphQL Data query and manipulation language based on a specification created by Facebook Inc. that is used to communicate with web APIs. 18, 21

IDE Integrated Development Environment. 11

Markdown A lightweight markup language with plain text formatting syntax. 11

Node Package Manager Package manager that is installed with Node.JS and is used to install and manage third party software in many JavaScript projects. 20

progressive web app Web site or application that is progressively enhanced to allow for more native app like features such as offline functionality. 9

React JavaScript-based front-end framework created and used by Facebook Inc. and published under the MIT License. 10, 18, 24

RESTful Architectural constraints for creating URI-based HTTP-APIs to provide interoperability between computer systems on the internet. 18

single page application Web application based on only one HTML page where data fetching and page transitions are done dynamically in the client browser. 24, 25

static site generator Software used as the central development framework for creating JAM stack projects. They handle generation and optimization of HTML sites and related assets. 8, 9, 18, 34

viewport Section of a web page that is visible to the viewer at his current scrolling position. 10

Vue JavaScript-based front-end framework originally created by Evan You and published under the MIT License. 10

YAML YAML Ain't Markup Language. A human-readable data-serialization language. 11

B References

[1] Akamai Inc. *Akamai Online Retail Performance Report: Milliseconds Are Critical*. Accessed: 2019-12-21. 2017. URL: https://www.akamai.com/uk/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp.

[2] Akamai Inc. *The State of Online Retail Performance*. Accessed: 2019-12-21. 2017. URL: https://www.akamai.com/us/en/multimedia/documents/report/akamai-state-of-online-retail-performance-spring-2017.pdf.

[3] Algolia Inc. *Algolia Homepage*. Accessed: 2020-01-07. URL: https://www.algolia.com/.

[4] Algolia Inc. *Distributed Search Network (DSN)*. Accessed: 2020-01-07. URL: https://www.algolia.com/doc/guides/scaling/distributed-search-network-dsn/.

[5] Algolia Inc. *Send and Update Your Data*. Accessed: 2020-01-07. URL: https://www.algolia.com/doc/guides/sending-and-managing-data/send-and-update-your-data/.

[6] Algolia Inc. *Understand the API Response*. Accessed: 2020-01-24. URL: https://css-tricks.com/star-apps-a-new-generation-of-front-end-tooling-for-development-workflows/.

[7] Atlassian Inc. *Webhooks*. Accessed: 2020-01-17. 2019. URL: https://developer.atlassian.com/server/jira/platform/webhooks/.

[8] Chris Bach. *Smashing Magazine just got 10x faster*. Accessed: 2020-01-21. 2017. URL: https://www.netlify.com/blog/2017/03/16/smashing-magazine-just-got-10x-faster/.

[9] Mathias Biilmann and Phil Hawksworth. "Modern Web development on the JAMstack: modern techniques for ultra fast sites and web applications". In: (2019).

[10] Contentful GmbH. *A JAMstack-ready CMS*. Accessed: 2019-12-16. URL: https://www.contentful.com/r/knowledgebase/jamstack-cms/.

[11] Chris Coyier. *ooooops I guess we're full-stack developers now*. Accessed: 2020-01-23. 2019. URL: https://full-stack.netlify.com/.

[12] Chris Coyier. *What is Serverless?* Accessed: 2020-01-08. URL: https://serverless.css-tricks.com/about/.

[13] Ryan Dahl. *Node.js Initial Release*. Accessed: 2019-12-09. 2009. URL: https://github.com/nodejs/node-v0.x-archive/tags?after=v0.0.4.

[14] Sarah Drasner. *Let's Build a JAMstack E-Commerce Store with Netlify Functions*. Accessed: 2019-12-16. 2019. URL: https://css-tricks.com/lets-build-a-jamstack-e-commerce-store-with-netlify-functions/.

[15] Drupal Association. *Headless Drupal*. Accessed: 2019-12-19. URL: https://groups.drupal.org/headless-drupal.

[16] Formspree Inc. *Formspree website*. Accessed: 2020-01-03. URL: https://formspree.io/.

[17] Free Software Foundation Inc. “GNU General Public License, version 1”. In: (1989). Accessed: 2019-12-09. URL: https://www.gnu.org/licenses/old-licenses/gpl-1.0.en.html.

[18] Free Software Foundation Inc. *GNU Project Homepage*. Accessed: 2019-12-09. URL: http://www.gnu.org/.

[19] Vitaly Friedman. *A Little Surprise Is Waiting For You Here*. Accessed: 2020-01-13. URL: https://www.smashingmagazine.com/2017/03/a-little-surprise-is-waiting-for-you-here/.

[20] Vitaly Friedman. *Demystifying JAMstack: An Interview With Phil Hawskworth*. Accessed: 2020-01-22. 2019. URL: https://www.smashingmagazine.com/2019/05/demystifying-jamstack-interview-phil-hawskworth/.

[21] Vitaly Friedman. *JAMstack Fundamentals: What, What And How*. Accessed: 2020-01-21. 2019. URL: https://www.smashingmagazine.com/2019/06/jamstack-fundamentals-what-what-how/.

[22] Gatsby Inc. *About Us*. Accessed: 2019-12-21. URL: https://www.gatsbyjs.com/about/.

[23] Gatsby Inc. *Announcing Gatsby's $15M Series A Funding Round*. Accessed: 2019-12-21. 2019. URL: https://www.gatsbyjs.org/blog/2019-09-26-announcing-gatsby-15m-series-a-funding-round/.

[24] Gatsby Inc. *Code Splitting and Prefetching*. Accessed: 2019-12-20. URL: https://www.gatsbyjs.org/docs/how-code-splitting-works/.

[25] Gatsby Inc. *Contributing to Gatsby.js*. Accessed: 2019-12-21. URL: https://www.gatsbyjs.org/contributing/.

[26] Gatsby Inc. *Gatsby Plugin Library*. Accessed: 2019-12-28. URL: https://www.gatsbyjs.org/plugins/.

[27] Gatsby Inc. *Gatsby Plugins*. Accessed: 2019-12-21. URL: https://www.gatsbyjs.org/plugins/.

[28] Gatsby Inc. *Gatsby vs JAMstack frameworks*. Accessed: 2019-12-20. URL: https://www.gatsbyjs.org/features/jamstack/.

[29] Gatsby Inc. *gatsby-image*. Accessed: 2019-12-28. URL: https://www.gatsbyjs.org/packages/gatsby-image.

[30] Gatsby Inc. *Gatsby.js Tutorials*. Accessed: 2020-01-21. 2019. URL: https://www.gatsbyjs.org/tutorial/.

[31] Gatsby Inc. *GraphQL Concepts*. Accessed: 2019-12-21. URL: https://www.gatsbyjs.org/docs/graphql-concepts/.

[32] Gatsby Inc. *GraphQL Query Options Reference*. Accessed: 2019-12-30. URL: https://www.gatsbyjs.org/docs/graphql-reference/.

[33] Gatsby Inc. *Introducing GraphiQL*. Accessed: 2019-12-30. URL: https://www.gatsbyjs.org/docs/running-queries-with-graphiql/.

[34] Gatsby Inc. *Node Interface*. Accessed: 2019-12-30. URL: https://www.gatsbyjs.org/docs/node-interface/.

[35] Gatsby Inc. *Optimizing images with gatsby-image*. Accessed: 2019-12-21. URL: https://www.gatsbyjs.org/docs/working-with-images/#optimizing-images-with-gatsby-image.

[36] Gatsby Inc. *Schema Generation*. Accessed: 2019-12-30. URL: https://www.gatsbyjs.org/docs/schema-generation/.

[37] Gatsby Inc. *Showcase*. Accessed: 2019-12-21. URL: https://www.gatsbyjs.org/showcase/.

[38] Gatsby Inc. *Understanding gatsby build (build time)*. Accessed: 2019-12-21. URL: https://www.gatsbyjs.org/docs/overview-of-the-gatsby-build-process/#understanding-gatsby-build-build-time.

[39] Gatsby Inc. *Understanding React Hydration*. Accessed: 2020-01-03. URL: https://www.gatsbyjs.org/docs/react-hydration/.

[40] Gatsby Inc. *wrapRootElement function*. Accessed: 2020-01-07. URL: https://www.gatsbyjs.org/docs/browser-apis/#wrapRootElement.

[41] GitHub Inc. *Pricing*. Accessed: 2020-01-20. 2019. URL: https://github.com/pricing.

[42] goneo Internet GmbH. *Rechenzentrum*. Accessed: 2020-01-20. URL: https://www.goneo.de/ueber_goneo/rechenzentrum.html.

[43] Google Inc. *Speed is now a landing page factor for Google Search and Ads*. Accessed: 2019-12-21. 2018. URL: https://developers.google.com/web/updates/2018/07/search-ads-speed.

[44] Gordon Haff. *How Open Source Ate Software*. Apress Media LLC, 2018.

[45] IBM Cloud Education. *LAMP Stack Explained*. Accessed: 2019-12-16. 2019. URL: https://www.ibm.com/cloud/learn/lamp-stack-explained.

[46] Nilesh Kadivar. *LAMP Stack or MEAN Stack — Which One to Choose For Your Next Web Application?* Accessed: 2019-12-10. 2019. URL: https://hackernoon.com/lamp-stack-or-mean-stack-which-one-to-choose-for-your-next-web-application-75dcbb2b69f1.

[47] Valeri Karpov. *The MEAN Stack: MongoDB, ExpressJS, AngularJS and Node.js*. Accessed: 2019-12-10. 2013. URL: https://www.mongodb.com/blog/post/the-mean-stack-mongodb-expressjs-angularjs-and.

[48] Tanner Linsley. *How to scale massive React Static sites with Incremental Builds*. Accessed: 2020-01-22. 2019. URL: https://www.netlify.com/blog/2019/01/17/how-to-scale-massive-react-static-sites-with-incremental-builds/.

[49] Chris Macrae. *Hugo vs Jekyll: Benchmarked*. Accessed: 2019-12-19. 2018. URL: https://forestry.io/blog/hugo-vs-jekyll-benchmark/.

[50] Kyle Mathews. *Initial commit*. Accessed: 2019-12-21. 2015. URL: https://github.com/gatsbyjs/gatsby/commit/24606f5a2d5c85d7b6661403333f34823409bdf3.

[51] Chris McCraw. *Is there a list of where Netlify's CDN pops are located?* Accessed: 2020-01-20. 2019. URL: https://community.netlify.com/t/is-there-a-list-of-where-netlifys-cdn-pops-are-located/855.

[52] Netcraft Ltd. *Web server developers: Market share of all sites*. Accessed: 2019-12-09. URL: https://news.netcraft.com/archives/2019/.

[53] Netlify Inc. *headlesscms.org*. Accessed: 2019-12-27. URL: https://headlesscms.org/.

[54] Netlify Inc. *JAMstack_conf*. Accessed: 2019-12-27. URL: https://jamstackconf.com/.

[55] Netlify Inc. *jamstack.org*. Accessed: 2019-12-27. URL: https://jamstack.org/.

[56] Netlify Inc. *Netlify Blog*. Accessed: 2019-12-27. URL: https://www.netlify.com/blog/.

[57] Netlify Inc. *Netlify Build*. Accessed: 2019-12-27. URL: https://www.netlify.com/products/build/.

[58] Netlify Inc. *Netlify CMS - Add to Your Site*. Accessed: 2019-12-31. URL: https://www.netlifycms.org/docs/add-to-your-site/.

[59] Netlify Inc. *Netlify Edge*. Accessed: 2019-12-27. URL: https://www.netlify.com/products/edge/.

[60] Netlify Inc. *Netlify Products*. Accessed: 2019-12-27. URL: https://www.netlify.com/products/.

[61] Netlify Inc. *Pricing*. Accessed: 2020-01-20. 2019. URL: https://www.netlify.com/pricing/.

[62] Netlify Inc. *staticgen.com*. Accessed: 2019-12-27. URL: https://www.staticgen.com/.

[63] Netscape Communications Corporation. *Server-Side JavaScript Guide*. Accessed: 2019-12-10. 1998. URL: https://docs.oracle.com/cd/E19957-01/816-6411-10/intro.htm#1022274.

[64] Mike Neumegen. *Front Matter*. Accessed: 2019-12-19. URL: https://jekyllrb.com/docs/front-matter/.

[65] O'Reilly Media Inc. *Modern Web Development on the JAMstack*. Accessed: 2019-12-27. URL: https://www.oreilly.com/library/view/modern-web-development/9781492058571/.

[66] Charles Ouellet. *JAMstack for Clients: Benefits, Static Site CMS, & Limitations*. Accessed: 2019-12-16. 2017. URL: https://codeburst.io/jamstack-for-clients-benefits-static-site-cms-limitations-d015336fa6d5.

[67] E. Sebastian Peyrott. *The JWT Handbook*. Auth0 Inc., 2016.

[68] Tamas Piros. *How to Build Fast and Secure Websites With the JAMstack*. Accessed: 2019-12-17. 2019. URL: https://www.shopify.com/partners/blog/jamstack.

[69] Tim Qian. *Github star history*. Accessed: 2019-12-20. URL: https://old-star-history.t9t.io/.

[70] Q-Success DI Gelbmann GmbH (Co. Ltd.) *Usage statistics and market share of WordPress*. Accessed: 2019-12-16. 2019. URL: https://w3techs.com/technologies/details/cm-wordpress.

[71] Eric Rosebrock and Eric Filson. *Setting up LAMP: getting Linux, Apache, MySQL, and PHP working together*. John Wiley & Sons, 2006.

[72] Parminder Sanghera. *AddToCart/index.js*. Accessed: 2020-01-07. URL: https://github.com/parmsang/gatsby-starter-ecommerce/blob/master/src/components/AddToCart/index.js.

[73] Alexander Schulze. *Transparenz in der Cloud*. published in dotnetpro edition 12.2019. Ebner Media Group GmbH & Co. KG, 2019.

[74] ScientiaMobile Inc. *ImageEngine Case Study – Furnspace*. Accessed: 2020-01-21. 2018. URL: https://www.scientiamobile.com/case-studies/imageengine-case-study-furnspace/.

[75] Chris Sevilleja and Holly Lloyd. *MEAN Machine: A beginner's practical guide to the JavaScript stack*. 2015.

[76] Akhil Sharma. *Gatsby Vs. Next.JS Vs. CRA (Create React App)*. Accessed: 2020-01-07. 2019. URL: https://medium.com/design-and-tech-co/gatsby-vs-next-js-vs-cra-create-react-app-59d61bca3774.

[77] Myles Shipman. *The Future Is Headless*. Accessed: 2019-12-19. 2018. URL: https://magento.com/blog/best-practices/future-headless.

[78] Shopify Inc. *Headless commerce*. Accessed: 2019-12-19. URL: https://www.shopify.com/plus/solutions/headless-commerce.

[79] Smashing Media AG. *About Us*. Accessed: 2020-01-13. URL: https://www.smashingmagazine.com/about/.

[80] Snipcart Inc. *List of e-commerce payment gateways for your store*. Accessed: 2020-01-22. URL: https://snipcart.com/list-ecommerce-payment-gateways.

[81] Snipcart Inc. *Pricing*. Accessed: 2020-01-22. URL: https://snipcart.com/pricing.

[82] Stripe Inc. *About*. Accessed: 2019-12-21. URL: https://stripe.com/en-de/about.

[83] Stripe Inc. *Accept a Payment*. Accessed: 2020-01-07. URL: https://stripe.com/docs/payments/accept-a-payment.

[84] Stripe Inc. *After the payment*. Accessed: 2019-12-31. URL: https://stripe.com/docs/payments/checkout/fulfillment.

[85] Stripe Inc. *API Reference - Products*. Accessed: 2019-12-21. URL: https://stripe.com/docs/api/products.

[86] Stripe Inc. *Billing*. Accessed: 2019-12-21. URL: https://stripe.com/en-de/billing.

[87] Stripe Inc. *Charges API*. Accessed: 2020-01-22. URL: https://stripe.com/docs/payments/charges-api.

[88] Stripe Inc. *Checkout*. Accessed: 2019-12-21. URL: https://stripe.com/en-de/payments/checkout.

[89] Stripe Inc. *Connect*. Accessed: 2019-12-21. URL: https://stripe.com/en-de/connect.

[90] Stripe Inc. *Orders API*. Accessed: 2020-01-22. URL: https://stripe.com/docs/orders.

[91] Stripe Inc. *Payment Methods API*. Accessed: 2019-12-21. URL: https://stripe.com/docs/payments/payment-methods.

[92] Stripe Inc. *Radar*. Accessed: 2019-12-21. URL: https://stripe.com/en-de/radar.

[93] Stripe Inc. *Sigma*. Accessed: 2019-12-21. URL: https://stripe.com/en-de/sigma.

[94] Stripe Inc. *The Session object*. Accessed: 2020-01-07. URL: https://stripe.com/docs/api/checkout/sessions/object.

[95] TYPO3-Initiative. *TYPO3 Extension "headless" - JSON content API for TYPO3 PWA solution*. Accessed: 2019-12-19. 2019. URL: https://github.com/TYPO3-Initiatives/headless.

[96] Sufyan bin Uzayr. *Using WordPress as a Headless CMS*. Accessed: 2019-12-19. 2018. URL: https://www.sitepoint.com/wordpress-headless-cms/.

[97] Shawn Wang. *STAR Apps: A New Generation of Front-End Tooling for Development Workflows*. Accessed: 2020-01-24. 2019. URL: https://css-tricks.com/star-apps-a-new-generation-of-front-end-tooling-for-development-workflows/.

[98] Christian Wollscheid. *Rise and burst of the dotcom bubble: causes, characteristics, examples*. GRIN Verlag, 2012.

C List of Figures

D List of Tables

E Listings

Listing 1: `hello-world.md`: Markdown with front matter for a blog post

```
---
path: '/blog/2019/12/17/hello-world'
title: 'Hello World'
date: '2019-12-17'
author: 'Simon Brandt'
---

## What this page is about
Lorem ipsum dolor sit amet, consectetur adipisicing elit.
Cupiditate delectus error modi nostrum perspiciatis quae
quasi quisquam quos sed vitae?
```

Listing 2: JSON Web Token as encoded string (actual token), with short explanations below

```
// JSON Web Token as base64-encoded string following
// the pattern "header.payload.signature":

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
    eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gQ.
    IrUKQwsPxlPvkcyV9kYuaOUqwGlCZwsiMEhc6ipcFA

// decoded header:
{
  "alg": "HS256", // hashing algorithm
  "typ": "JWT" // type
}

// decoded payload (user data + permissions):
{
  "sub": "235", // id
  "name": "John Doe", // name
  "iat": 1516239022, // 'issued at time': timestamp
  "roles": ["admin"] // permissions
}

// The signature is created by hashing header+payload with a 'shared-secret'
    known to all connected services:
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  shared-secret
)
```

Listing 3: `blog.tsx`: The blog page (./src/pages/blog.tsx), using the layout component

```
import Layout from 'componentsLayout'

const Blog = (props) => {

    const {blogPosts} = props

    return (
        <Layout>
            <h1>Blog</h1>
            <div>
                {blogPosts.map(blogPost => (
                    <Card>
                        <BlogPostSnippet
                            blogPost={blogPost}
                            key={blogPost.frontmatter.title}
                        />
                    </Card>
                ))}
            </div>
        </Layout>
    )
}
```

Listing 4: `Layout.tsx`: The layout component (./src/components/Layout.tsx)

```
const Layout = (props: Props) => (
    <>
        <Header siteTitle={data.site.siteMetadata.title} />
            <main>{props.children}</main>
        <Footer />
    </>
)
```

Listing 5: `gatsby-config.js`: Section of the gatsby-config.js, referencing the gatsby-source-filesystem plugin

```
{
    resolve: `gatsby-source-filesystem`,
    options: {
        name: `images`,
        path: `${__dirname}/src/images/`,
    },
},
```

Listing 6: `index.tsx`: GraphQL query to retrieve the banner image from Gatsby's data layer

```
{
    bannerImage: file(relativePath: { eq: "banner.jpg" }) {
        childImageSharp {
            fluid(quality: 90, maxWidth: 2000) {
                ...GatsbyImageSharpFluid_withWebp
            }
        }
    }
}
```

Listing 7: `index.tsx`: Import and usage of the gatsby-image component

```
import Img from 'gatsby-image'
...
<Img
    fluid={props.data.bannerImage.childImageSharp.fluid}
/>
```

Listing 8: `gatsby-config.js`: Section of the gatsby-config.js referencing the gatsby-source-stripe plugin

```
{
    resolve: `gatsby-source-stripe`,
    options: {
        objects: [`Product`, `Sku`],
        secretKey: process.env.STRIPE_SECRET_KEY,
        downloadFiles: true,
    },
},
```

Listing 9: `.env`: Example contents of a .env file (truncated keys)

```
STRIPE_SK_TEST = sk_test_In56OVxDt02DOVYA...
GATSBY_BASE_URL = http://localhost:8000/
GATSBY_OMDB_API_KEY = 474e0...
```

Listing 10: `index.tsx`: GraphQL query to load Stripe SKUs into a React component

```
{
    allStripeSku {
        edges {
            node {
                id
                currency
                price
                attributes {
                    name
                }
                image
                localFiles {
                    childImageSharp {
                        fluid(maxWidth: 500) {
                            ...GatsbyImageSharpFluid_tracedSVG
                        }
                    }
                }
            }
        }
    }
}
```

Listing 11: `index.tsx`: Displaying the SKUs from Stripe in Sku components

```
1
2  export default (props: Props) => {
3
4      return (
5          <>
6              <BackgroundImage
7                  fluid={props.data.bannerImage.childImageSharp.fluid}
8                  style={{ width: '100%', height: 400 }}
9              />
10             <Layout>
11                 <h1>eCommerce Prototype</h1>
12                 <h2>Take a look at the books we have in stock</h2>
13                 <p style={{ textAlign: 'center' }}>
14                     Book data is stored at Stripe, was queried during build
                           time and statically built into this site.
15                 </p>
16                 <div>
17                     <Grid container spacing={2} justify={'center'}>
18                         {props.data.allStripeSku.edges.map(
19                             (edge: { node: SkuType }) => (
20                                 <Sku sku={edge.node} key={edge.node.id}/>
21                             ),
22                         )}
23                     </Grid>
24                 </div>
25             </Layout>
26         </>
27     )
28 }
```

Listing 12: `StripeButton.tsx`: Implementation of the React component used for the 'buy now' button

```
const StripeButton = (props: Props) => {
    const [stripe, setStripe] = useState<any | undefined>(undefined)

    useEffect(() => {
        setStripe(window.Stripe('pk_test_fJeBDpFjDQ3ijdBrBi23i3Dk00RcsVwxMV'))
    }, [])

    return (
        <Button
            variant={'outlined'}
            size={'small'}
            onClick{=event => {
                event.preventDefault()
                if (!stripe) return
                stripe.redirectToCheckout({
                    items: [{ sku: props.sku_id, quantity: 1 }],
                    successUrl: process.env.GATSBY_BASE_URL + 'paymentSuccess
                        ',
                    cancelUrl:  process.env.GATSBY_BASE_URL + 'paymentCanceled
                        ',
                }).then(function(result: any) {
                    if (result.error) {
                        alert('An error has occured.')
                    }
                })
            }}
        >
            buy now
        </Button>
    )
}
```

Listing 13: `./static/admin/index.html`: HTML file used to load the Netlify CMS and Netlify identity widget

```
<!doctype html>
<html>
<head>
    <title>Content Manager</title>
    <!-- Netlify identity widget -->
    <script src="https://identity.netlify.com/v1/netlify-identity-widget.js">
    </script>
</head>
<body>
    <!-- Netlify CMS -->
    <script src="https://unpkg.com/netlify-cms@2.0.0/dist/netlify-cms.js">
    </script>
</body>
</html>
```

Listing 14: `./static/admin/config.yml`: Configuration file for the Netlify CMS

```
backend:
    name: git-gateway
    branch: master
    commit_messages:
        create: 'Create {{collection}} "{{slug}}"'
        update: 'Update {{collection}} "{{slug}}"'
        delete: 'Delete {{collection}} "{{slug}}"'
        uploadMedia: '[skip ci] Upload "{{path}}"'
        deleteMedia: '[skip ci] Delete "{{path}}"'

media_folder: static/img
public_folder: /img

collections:
    - name: 'blog'
      label: 'Blog Post'
      folder: 'src/markdown-pages/blog'
      create: true
      slug: '{{year}}-{{month}}-{{day}}-{{slug}}'
      fields:
          - { label: 'Template Key', name: 'templateKey', widget: 'hidden',
              default: 'blog-post' }
          - { label: 'Title', name: 'title', widget: 'string' }
          - { label: 'Publish Date', name: 'date', widget: 'datetime' }
          - { label: 'Featured Post', name: 'featuredpost', widget: 'boolean' }
          - { label: 'Body', name: 'body', widget: 'markdown' }
          - { label: 'Tags', name: 'tags', widget: 'list' }
```

Listing 15: `./src/pages/index.tsx`: Implementation of the dynamic movie search

```
export default (props: Props) => {
    const [movieResults, setMovieResults] = useState<Movie[]>([])
    const [search, setSearch] = useState<string>('')

    useEffect(() => {
        const url: string = 'http://www.omdbapi.com/?apikey=' + process.env.
            GATSBY_OMDB_API_KEY + '&s=' + search
        fetch(url)
            .then(res => res.json())
            .then(data => {
                setMovieResults(data.Search)
            })
    }, [search])

    return (
        <>
            <Layout>
                <div>
                    <h2>Not a fan of books? Use our live search to find
                        interesting movies to watch</h2>
                    <p>
                        The media data in this section is dynamically pulled
                            at runtime from <a
                        href="www.omdbapi.com">www.omdbapi.com</a>.
                    </p>
                    <div>
                        <input
                            value={search}
                            onChange=event => setSearch(event.target.value)
                            placeholder={'type a movie title'}
                        />
                        <Grid container spacing={2}>
                            {
                                movieResults ?
                                    movieResults.map((movie: Movie) => {
                                        return <MoviePreview movie={movie}/>
                                    })
                                    : search
                                    ? <CircularProgress/>
                                    : null
                            }
                        </Grid>
                    </div>
                </div>
            </Layout>
        </>
    )
}
```

Listing 16: `./src/pages/contact.tsx`: Example for a contact form, implemented using Formspree as a back end

```
<form
   action="https://formspree.io/simon.brandt@htw-dresden.de"
  method="POST"
>
  <label>
    Name:
    <input type="text" name="name">
  </label>
  <label>
    Email:
    <input type="email" name="_replyto">
  </label>
  <label>
    Message:
    <textarea name="message"></textarea>
  </label>
   <input type="submit" value="Send">
</form>
```

Listing 17: Persisting an array of products (shopping cart) across HTML pages with localStorage

```
// define cart
let cart = [
  {
    id: 1,
    name: 'book 1',
    price: '20'
  },
  {
    id: 2,
    name: 'book 2',
    price: '10'
  },
]

// save cart to localStorage
localStorage.cart = JSON.stringify(cart)

// retrieve cart from localStorage
cart = JSON.parse(localStorage.cart)
```

Listing 18: `gatsby.md`: Markdown file that stores information for the Gatsby entry on serverless.css-tricks.com

```
---
path: "services/ssgs/gatsby"
title: "Gatsby"
url: "https://www.gatsbyjs.org/"
logo: "/images/gatsby.svg"
---
Gatsby is a free and open source framework based on React that helps
    developers build blazing fast websites and apps.
```

Listing 19: `config.yml`: section of the Netlify CMS config file, describing the schema for SSG entries on serverless.css-tricks.com

```
collections:
  - name: "ssgs"
    label: "Static Site Generators"
    folder: "src/content/ssgs"
    create: true
    slug: "{{slug}}"
    editor:
      preview: false
    fields:
      - { label: "Path", name: "path", widget: "string" }
      - { label: "Title", name: "title", widget: "string" }
      - { label: "URL", name: "url", widget: "string" }
      - { label: "Body", name: "body", widget: "markdown" }
      - { label: "Logo", name: "logo", required: true, widget: "image" }
```

Listing 20: A script tag with JSON metadata that is injected into product detail pages on smashingmagazine.com

```
<script class="gocommerce-product" type="application/json">
{
 "sku": "a-career-on-the-web-assuming-leadership",
 "title": "A Career On The Web: Assuming Leadership",
 "image": "//cloud.netlifyusercontent.com/assets/3442cef/large.png",
 "type": "E-Book",
 "prices": [
 {"amount": "4.99", "currency": "USD"},
 {"amount": "4.99", "currency": "EUR"}
 ]}
</script>
```

Listing 21: Example response for an Algolia query, showing one hit for the search query ”jimmie paint” [6]

```
{
  "hits": [
    {
      "firstname": "Jimmie",
      "lastname": "Barninger",
      "objectID": "433",
      "_highlightResult": {
        "firstname": {
          "value": "<em>Jimmie</em>",
          "matchLevel": "partial"
        },
        "lastname": {
          "value": "Barninger",
          "matchLevel": "none"
        },
        "company": {
          "value": "California <em>Paint</em> & Wlpaper Str",
          "matchLevel": "partial"
        }
      }
    }
  ],
  "page": 0,
  "nbHits": 1,"
  "nbPages": 1,
  "hitsPerPage": 20,
  "processingTimeMS": 1,
  "query": "jimmie paint"
}
```