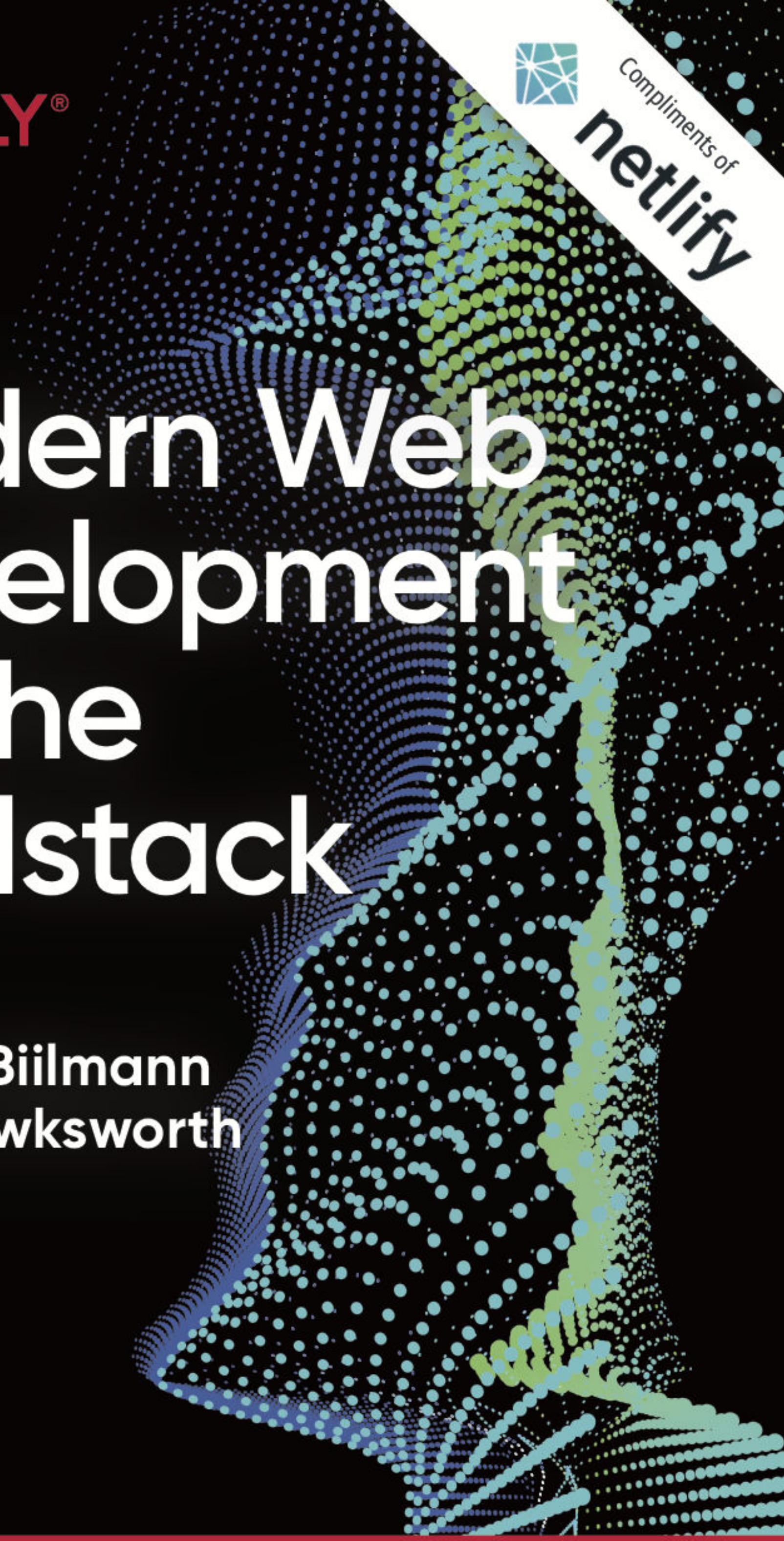
O'REILLY®

Compliments of netlify

Modern Web Development on the JAMstack

Mathias Biilmann & Phil Hawksworth

REPORT



Really pause and think about how much time and effort web teams around the world have spent building and managing infrastructure. For many years, launching a site or web application has been as much about deploying complex server environments as it's been about building actual application code. The cloud made provisioning all these resources faster but no less complicated.

The JAMstack was born of the stubborn conviction that there was a better way to build for the web. Around 2014, developers started to envision a new architecture that could make web apps look a lot more like mobile apps: built in advance, distributed, and connected directly to powerful APIs and microservices. It would take full advantage of modern build tools, Git workflows, new frontend frameworks, and the shift from monolithic apps towards decoupled frontends and backends.

At Netlify, we've watched enthusiasm for the JAMstack grow well beyond our wildest expectations. It now powers every site we deploy for over 500,000 developers and companies. We've seen firsthand how the JAMstack improves the experience for both users and developers. Most importantly, we've seen how increases in site speed, site reliability, and developer productivity can contribute to the continued health and viability of the open web.

We're thrilled that you, too, want to explore the JAMstack and hope this book serves as a useful guide. Welcome to the community. Welcome to a new way to develop, deploy, and power web content and applications. Welcome to the JAMstack.

Sincerely,

Matt Biilmann
CEO, Netlify

netlify

Modern Web Development on the JAMstack

Modern Techniques for Ultra Fast Sites and Web Applications

Mathias Biilmann and Phil Hawksworth

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Modern Web Development on the JAMstack
by Mathias Biilmann and Phil Hawksworth

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Jennifer Pollock
Development Editor: Angela Rufino
Production Editor: Elizabeth Dayton
Copyeditor: Octal Publishing, Inc.
Proofreader: Sonia Saruba
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

June 2019: First Edition

Revision History for the First Edition
2019-06-17: First Release

978-1-492-05854-0

[LSI]

Table of Contents

Introduction

In just the past few years, a flurry of advancements has greatly strengthened the web as a content and application platform. Browsers are much more powerful. JavaScript has matured. WebAssembly is on the horizon. It certainly feels like the beginning of a new chapter for the web. You've likely felt this as you've witnessed the explosion of new frontend frameworks and API-based services.

Although what's technically possible in the browser has advanced, so too have expectations for immediacy. Videos must play instantly. Browser applications must launch faster than their desktop counterparts. Users have become increasingly mobile and increasingly impatient—we are all fed up with slow pages and we vote angrily against them with the back button. (Google also seems to be losing patience, and now factors site speed into its famous ranking algorithms.)

Broadly speaking, this book covers new, modern approaches to building websites that perform as fast as possible. More concretely, this book shows you how to run any web property, from simple sites to complex applications, on a global Content Delivery Network (CDN) and without a single web server. We introduce you to the JAMstack: a powerful new approach for deploying fast, highly scalable sites and applications that don't require traditional frontend infrastructure. If you tend to feel delivering great websites should be more about the craft of markup and JavaScript than server setup and administration, you've found your book.

And, if you've ever struggled with any monolithic system or framework—wrestling with hosting, deploying changes, securing, and scaling everything—you already understand why the JAMstack is becoming so popular. It's one of those rare shifts in the landscape that delivers a productivity boost for developers *and* a large performance boost for users. The JAMstack is helping to bring about a more secure, stable, and performant web that's also more fun to develop and create for.

Under this new approach, there isn't a "server environment" at all—at least not in the traditional sense. Instead, HTML is prerendered into static files, sites are served from a global CDN, and tasks that were once processed and managed server side are now performed via APIs and microservices.

We realize that seasoned web developers might eye anything new with quite a bit of skepticism. There have certainly been countless other new ideas on how to build the web before. For nearly three decades, the developer community has explored ways to make the web easier and faster to develop, more capable, more performant, and more secure.

At times, though, the effort has seemed to trade one goal for another. Wordpress, for example, became a revolution in making content easier to author—but anyone who's scaled a high-traffic Wordpress site knows it also brings a whole set of new challenges in performance and security. Trading the simplicity of HTML files for database-powered content means facing the very real threats that sites might crash as they become popular or are hacked when nobody is watching closely.

And dynamically transforming content into HTML—each and every time it's requested—takes quite a few compute cycles. To mitigate all the overhead, many web stacks have introduced intricate and clever caching schemes at almost every level, from the database on up. But these complex setups have often made the development process feel cumbersome and fragile. It can be difficult to get any work done on a site when you can't get it running and testable on your own laptop. (Trust us, we know.)

All these challenges have led the developer community to begin exploring the JAMstack as a modern refactoring of the way websites are developed and served. And like any good refactoring, you'll find that it both advances and simplifies the stack.

Being web developers ourselves, the authors of this book are more excited by the JAMstack than we've been about any emerging trend for quite a while. That's because it uniquely solves for all these common problems—developer experience, performance, and security—all at the same time, and all without one compromising the other. To us, it feels like the logical future of the platform.

It just makes sense.

Developer Jonathan Prozzi said it best on Twitter: "My learning journey leading to #JAMstack has re-ignited my interest and passion for web technology." We think the JAMstack can do a lot to rekindle your own enthusiasm, too, and we're eager to welcome you to the growing community of JAMstack practitioners. You'll find developing for the web has become fun and fearless all over again.

The JAM in JAMstack

The JAMstack brings together JavaScript, APIs, and markup, the three core components used to create sites that are both fast and highly dynamic. JAMstack sites are well suited to meet the demanding requirements of today's mobile-first web (where load times urgently matter and quality bandwidth can never be guaranteed).

We should pause to say what the JAMstack isn't: it's not any one specific technology in and of itself; nor is it driven by a large company; nor is there any standards body that controls or defines it.

Instead, the JAMstack is a movement, a community collection of best practices and workflows that result in high-speed websites that are a pleasure to work on.

As you dive in, you'll find a few common guidelines but also lots of choice, including your pick of languages. JavaScript is called out specifically as the language of the browser, but you can use as much or as little JavaScript as your project requires. Many JAMstack sites also use Python, Go, or Ruby for templating and logic. Lots of open source software has sprung up to help you create JAMstack sites, and we spend some time in this book going over a few of the more popular options.

Not everything about the JAMstack is a new idea, but it's only very recently that we've had the technology required to make the approach possible, especially on the edge of the network and in browsers. This first section of the book should give you a working understanding of the JAMstack, why it's an important new development, and how to reason about it.

A Workflow for Productivity and Performance

By nature, JAMstack sites are the following:

- Globally distributed and resilient to heavy traffic
- Centered around a developer friendly, Git-based workflow
- Designed modularly, consuming other services via APIs
- Prebuilt and optimized before being served

It's this last point that deserves special attention. Think for a moment about today's most common approach to serving web content: for each and every request made to a website, data is pulled from a database, rendered against a template, processed into HTML, and finally pushed across the network (and maybe even an ocean) to the browser that requested it.

When web servers repeatedly build each page for each request, it begins to feel like a lot of work happens in the wrong place at the wrong time. After all, a general rule for maximum performance is to perform the fewest steps possible. Shouldn't new HTML be produced only when content or data changes, not every time it is requested?

As it turns out, that's exactly how the JAMstack operates. Here's what happens under the JAMstack workflow:

1. The source for the site is a hosted repository that stores content and code together as editable files.
2. Whenever a change made, a build process is triggered that prerenders the site, creating final HTML from templates, content, and data.
3. The prepared, rendered assets are published globally on a CDN, putting them as close to end users as physically possible.

This approach eliminates large amounts of server and network latency. Given the efficiency and simplicity of serving prerendered content directly from a CDN, it's no surprise JAMstack sites tend to acheive the highest possible scores on speed tests like Google Lighthouse.

Globally distributed content isn't entirely new, but the speed at which you can update CDN files directly from a repository *is* new. No more dreaded delays waiting for the CDN's cache to expire—we can now build highly distributed sites right on the edge of the network and remove the need for any type of frontend servers to process each request.

Version Control and Atomic Deploys

In the JAMstack, it's possible and common for the content of the site, blog posts and all, to live in a Git repository right alongside the code and templates. This means that no content database is required, making JAMstack sites dramatically easier to set up, run, deploy, branch, and modify.

And in this version-control-centric world of the JAMstack, every deploy of the site is *atomic*, making it trivial to roll back to any state, at any time, for any reason. Complex staging environments are no longer needed as previewing and testing changes use the branching system built into the heart of Git. With the workflow for changes made fast, simple, and safe, most JAMstack sites are far from "static" and are often updated just as often as their more complex counterparts, sometimes hundreds of times daily.

Contributing on the JAMstack

As you picture a JAMstack workflow, you probably imagine making code changes in an editor and then running a command to build the site and deploy it to production. Quite a bit of development on the JAMstack happens in this exact way, especially early on.

But for most JAMstack sites, the contributors aren't just developers. New updates to the site are also triggered by content authors using a CMS as well as by automated actions, just as you'd expect from any modern website.

Instead of happening locally, the build process now often runs on a hosted service in the cloud. Push a change to GitHub (or another repository service) and a new build is automatically triggered on special-purpose build servers, sending the final result directly to the CDN. It's an amazingly addictive way to develop, and we show you how to get it going.

But what about those authors who aren't developers and might not be familiar with Git? The JAMstack has spawned a clever new generation of authoring tools that look and function like a normal Content Management System (CMS) but actually check changes into version control behind the scenes. In this way, everyone participates in the same workflow, enjoying the safety, branching, and rollbacks of modern version control software—even if they are unaware it's happening. It's a nice improvement over content requiring a database that you need to manage and version separately.

You also can contribute changes to the site in a third way: programmatically, via automation. For example, you can run a script daily that incorporates the latest press articles and Twitter mentions into the home page. Or take what happens on *Smashing Magazine*'s JAMstack site: each time a user comments on an article, a simple function commits the comment to the site's repository and triggers a new build (as long the comment passes moderation).

Starting to see the power of this workflow? Developers, content authors, and automated processes are all saving a living history of the site right to the same place—the repository—which acts as one source of truth. Even if the site is updated hundreds of times in a day or more, every state is preserved. And, most important, the site stays fast and responsive because every page is built and optimized before being served.

APIs to Process and Personalize

It takes more than HTML, though: web applications need to perform work and have state. Traditionally, web servers were required to store a user's session and allow the application to do things like remember items in a shopping cart or power the checkout experience.

In the JAMstack, these personalized experiences are done using JavaScript to make API calls that send and receive data. Many of the APIs used are third-party services. Stripe, for example, is a popular service for processing payments. Algolia is a popular API for powering search. (Your authors have a deep love for almost all technology, but would never go back to manually handling payments or running Apache Solr search clusters.)

Other APIs can be custom-built functions unique to each application. Instead of a monolithic and complex framework that manages everything, sites can now be designed around microservices: simple functions that perform specific tasks, executing once when called, and then exiting cleanly. It's a very scalable approach that's easy to reason about.

But with no servers to store user sessions, how do we connect all of these discrete API calls? How do we handle authentication and identity? To power user accounts, a JAMstack application will often create a secure ID token stored in the browser. (This type of token is called a JavaScript Web Token, or JWT.) The identity is then passed with each API call so that services are aware of the user. We cover this setup in more detail later, but we introduce it here to help you understand the power of the platform. JAMstack sites can do much more than serve simple content, and many are rich applications with all the features you'd expect, like ecommerce, membership, and rich personalization.

Static sites may be, well, static, but you'll find that JAMstack sites are anything but. In fact, you'll find prerendered markup for web UI combined with APIs and microservices to be one of the fastest and most reliable platforms available for advanced web applications.

Bringing It All Together: A *Smashing Magazine* Case Study

Smashing Magazine has been a steady presence in web development for more than a decade, running a massively popular website and publishing high-quality content covering themes like frontend development, design, user experience, and web performance.

At the end of 2017, *Smashing Magazine* completely rewrote and redesigned its site to switch from a system based on several traditional, monolithic applications to the JAMstack. This final section of the book showcases how it solved challenges like ecommerce, content management, commenting, subscriptions, and working with a JAMstack project at scale.

Ready? Time to Get Our JAM Session Started

The best way to learn the JAMstack is to dive right in. The JAMstack uses the skills that you already have but pushes on your conventions about how websites are run and what's possible without servers. We suggest you bring the same open mind and sense of adventure that brought you to explore web development in the first place. There's never been a better time to be a web developer—unburdening ourselves from administering websites is bringing the focus back to creating the content and interfaces that make them compelling.

CHAPTER 1

The Challenges of Modern Web Development

In technology, any kind of change—even one that's gradual or starts with smaller projects—won't be made lightly. These shifts involve evaluation of past investments in infrastructure, acquisition of new knowledge, and thoughtful consideration of feature trade-offs.

So, before we can convince you that there's value in considering the JAMstack as yet another approach to web development, we'll spend some time digging into the most pervasive approaches to web development today and the challenges they introduce. Put simply, let's look first at the risk of sticking with the status quo before diving into the JAMstack and the benefits of making a switch.

The Drawbacks of Monolithic Architectures

Web sites have traditionally been powered by monolithic architectures, with the frontend of the application tightly coupled to the backend.

Monolithic applications are opinionated about how tasks on the server take place. The opinions they enforce have shaped how web development is approached and the environment within which we now work.

Popular and trusted as the *de facto* way to build for the web, these tools have had a profound impact not just on how sites will perform, but also on how efficiently they can be developed and even on the mindset of the developers.

The lock-in created by this is tangible. These applications dictate the approach to many facets of a web development project in ways that are not always appealing to the web development community at large. The selection of one monolithic app over another is often based on how long their lists of features are. A tendency to select tools based on what is best in class, or the most popular in the market, can often overlook a fundamental consideration: what does this *project* actually need?

So, does this sound familiar? Have you worked on projects where what should be simple is actually rather complex? Perhaps this complexity has been accepted because the project is for a high-profile site or being worked on by a large team?

Does the complexity you encounter feel truly necessary?

By regularly asking ourselves these questions, we have come to believe that there is a level of overengineering present in web development today. This overengineering impedes the use of proven, effective processes, technologies, and architectures.

Can we take action to improve this situation? Can we attack the limited flexibility, performance concerns, scaling challenges, and security woes imposed on us by legacy architecture?

Limited Flexibility

The freedom to do good work—to design the exact experiences we want for our users—is regularly compromised by complex monolithic applications.

Over the past few years, there has been a rising appreciation of the dramatic performance gains made by optimizing frontend code. Yet, while recognition of the value of strong frontend development has grown, the platforms developers use have often not afforded the freedom to bring frontend skills fully to bear.

The architectures imposed by monolithic applications can undermine our ability to utilize the latest technologies and even affect such things as our development workflows. This inhibits our efforts to fundamentally protect important aspects of software development such as a healthy development experience to unlock the potential in our talented development teams.

Performance Concerns

Website performance is integrally important to the success of internet-delivered content. Many studies exist that conclude that performance and conversion are tightly linked. One such study concluded that a single-second delay in load time can hurt the conversion of an ecommerce site by 7%.

As it happens, monolithic apps are rarely conducive to superior site performance. They need to generate and deliver HTML every time a new visitor arrives on the site. This significantly slows down page load time.

But performance isn't dictated by speed alone. Because monolithic apps are so large, it can be difficult to define architectural boundaries. The code is so interconnected that fixing a bug, updating a library, or changing a framework in one part of the app can break another part of it.

Caching is another important part of overall site performance—and it's one that's notoriously difficult to get right with a dynamic site. If a site is built on a monolithic app, it's possible that the same URL can return different content depending on a variety of parameters, including whether a user is logged in or the site was previously running a split test. Being able to deliver a predictable experience to end users is vital.

Scaling Challenges

Because monolithic architecture requires the page view to be generated for every visitor, infrastructure needs to be scaled in anticipation of site traffic. Not only is that expensive, it's also difficult to get right. Teams end up over-provisioning their infrastructure to prevent downtime or risk a crash because there is no clear separation between the infrastructure required to generate and manage the site and that required to serve the site.

When the same application that builds the page views or lets authors manage content also needs to be scaled to handle traffic spikes, it's not possible to decouple these facilities in order to scale and protect each piece of the infrastructure according to its needs.

In this scenario, the considerations that influence the technical architecture of a site suffer from being lumped into one large system. In computer science, we recognize that the costs of writes and reads can be very different, and yet monolithic apps bundle these features together into the same application, making them difficult to design for appropriately.

The approach to designing a system that allows for hundreds of millions of read operations is very different to that of a system that allows a similar number of write operations. So, is it wise to combine these capabilities into the same infrastructure and demand that the risk profile of one feature influence that of another? And are those features likely to encounter similar traffic levels?

There are ways to put distance between the users visiting your site and the complexity that generates and delivers that site. Without such separation, scale can be difficult to achieve.

Security Woes

We want to ensure that as we're evaluating the reasons that monolithic apps are no longer serving the growth of the web, we're careful not to insult the hundreds of thousands of development teams that chose to implement them. We've been that team. You might be on that team right now. One of the ways to avoid this is by simply looking at the facts. And when we talk about security, the facts are difficult to dismiss.

According to recent estimates, Wordpress powers 29% of the web. Joomla and Drupal follow as the second and third most popular content management systems, respectively. That makes them a prime target for bad actors.

The availability of these products as hosted services can help to absolve us from some of the responsibilities for securing the underlying infrastructure, but in self-hosted instances, we must shoulder all of that burden ourselves. In either case, there are many attack vectors, and the introduction of third-party plug-ins can expose us to further risk and make things even more difficult to secure.

Monolithic apps like Wordpress, Drupal, and Joomla combine every single component and plug-in of a web project's architecture into a single codebase. In turn, it creates a massive surface area for malware to penetrate. Not only is the attack surface area extremely large, it's also exposed every single time the site is built because any plug-in the site uses must execute each time the page loads for a new site visitor, magnifying the risk.

With that volume of distribution, injecting malware into a single plug-in can mean massive distribution. A recent attack caused 12 million Drupal sites to require emergency patching.

And there's one more gotcha: in monolithic apps, plug-ins are tied directly to the core framework. Because they're notoriously insecure and require frequent (and often buggy) updates, hastily updated plug-ins run the risk of breaking the entire site. Maintainers are left to choose between security patches and the risk of breaking site functionality.

The Risk of Staying the Same

Developers and business decision-makers alike have recognized the risks of continuing to develop web projects using monolithic apps. But the risk of change often outweighs the risk of staying the same. For large companies, massive investment in architecture and talent combines with fear of giving up the flexibility, features, and tooling of a more mature ecosystem. For small businesses, rebuilding web infrastructure pulls resources away from critical growth initiatives.

However, it's our belief that the time has come when continuing to invest in the status quo is a risk too great to perpetuate. And there's an army of developers who are backing this up. The first whispers of change came as a wave of personal blogs, open source project documentation sites, and time-bound projects like events and conference sites migrated to the JAMstack.

These whispers have grown louder as larger sites with greater audience numbers and more content also began embracing the JAMstack, challenging previously accepted limitations by delivering comments, subscription content, ecommerce, and more.

As more of the world's population comes online, it's essential that we deliver sites that can handle huge traffic spikes, perform quickly and predictably under any traffic conditions, support different languages and localizations, and be accessible and usable on all the devices we know and many that we don't. And, sites need to do this all while being secure and protected from the malicious attacks that are inherent in an open, global network.

This sounds like a job for the JAMstack.

CHAPTER 2

Introducing the JAMstack

What's in a Name?

The JAMstack at its core, is simply an attempt to give a name to a set of widely used architectural practices. It gives us one word to communicate a large range of architectural decisions.

It came about in conversations between people involved in the communities around static site generators, single-page application (SPA) frameworks, build tools, and API services as we realized much of the innovation in each of these categories was connected.

An entire community grew up around the first mature generation of static site generators like Jekyll, Middleman, Metalsmith, Cactus, and Roots. Jekyll found fame as the static site generator supported natively by GitHub Pages, and others were developed in-house by agencies or startups that made them a core part of their website stack. The tools brought back a feeling of simplicity and control from the complex database-based Content Management Systems (CMSs) that they replaced.

At the same time, single-page app frameworks like Vue and React started a process of decoupling the frontend from the backend, introducing build tools like Grunt, Gulp, and later, Webpack and Babel, which all propagated the idea of a self-standing frontend with its own build pipeline and deployment process. This brought a proper software architecture to the frontend workflow and empowered frontend developers to iterate much faster on the user interface (UI) and interaction layer.

When these approaches to building websites and applications intersected, a new API ecosystem began to emerge. Tools like Stripe, Disqus, and Algolia made payments, comments, and search available directly from the frontend via API-driven services.

A large set of terms for the different pieces of these new software architectures began to appear: progressive web apps, static sites, frontend build pipelines, single-page applications, decoupled web projects, and serverless functions. However, none of them really captured the move to a new architecture that encompassed content-driven websites, web applications, and all the interesting hybrids between the two.

The JAMstack put a name to all of it.

In today's web development environment, the "stack" has moved up a level. Before, we used to talk about the operating system, the database, and the web server when describing our stack (such as the LAMP stack and MEAN stack), but with the emergence of new practices, the real architectural constraints became the following:

- **J**avaScript in the browser as the runtime
- Reusable HTTP **A**PIs rather than app-specific databases
- Prebuilt **m**arkup as the delivery mechanism

These components became the JAM in JAMstack. Let's take a look at each of these elements and how they've each evolved to enable this new architecture.

JavaScript

In May 1995, when Brendan Eich wrote the first prototype of JavaScript (originally called Mocha) during a famous 10-day coding spree, few could have imagined that it would someday be the most important runtime language for the widest-reaching publishing stack ever.

Today, JavaScript is nearly omnipresent in browsers. It has grown from Eich's initial creation of a small, scheme-inspired language with a C-like syntax into the most highly optimized interpreted language in the world. It includes advanced constructs for asynchronous actions, flexible class and module system, elegant syntactic constructs like destructuring assignment, and an object syntax that's become the most commonly used data exchange format on the web and beyond (JSON).

More than just a standalone programming language, JavaScript has become a compilation target for *transpilers*, which translate variations of JavaScript into vanilla JavaScript runnable in all modern browsers, and *compilers* for new or existing languages like Elm, ReasonML, ClojureScript, and TypeScript.

In many ways JavaScript is now the universal runtime that Sun Microsystems dreamed of when it built the Java Virtual Machine (JVM).

This "Virtual Machine" for the web is the runtime layer for the JAMstack. It's the layer we target when we need to create dynamic flows that go above the content and presentation layer, whether we're writing fully fledged applications or just adding extra functionality to a content-based site. JavaScript is to the JAMstack what C was to Unix, as the browsers have become the operating system of the web.

APIs

The World Wide Web is fundamentally just a UI and interaction layer built on top of a stateless protocol called the HyperText Transfer Protocol (HTTP). The most fundamental concept of the web is the Universal Resource Locator (URL).

As humans, we're used to being able to type a URL into a browser or follow a link from a website and end up on another website. When building websites or applications, we should always think carefully about our URL architecture and structure. But URLs also give programs running in a browser the power of reaching any programmatic resource that's been exposed to the web.

Modern web APIs were officially born with Roy Fielding's dissertation, "Architectural Styles and the Design of Network-Based Software Architectures," in 2000. It is a seminal paper that defined Representational State Transfer (REST) as a scalable and discoverable architecture for services exposed through HTTP.

The first web APIs were meant to be consumed from server-side applications. Without using tricks like proxying through an intermediary server or using plug-ins like Flash or Java Applets (ugh!), there was no way for a program running in a standard browser to talk to any web API outside of its own domain.

As JavaScript grew from a scripting language mainly meant to do minor progressive enhancements on top of server-side rendered websites into a fully fledged runtime environment, new standards like CORS, WebSocket, and PostMessage emerged. Paired with other new standards like OAuth2 and JSON Web Token (JWT) for authorization and stateless authentication, any modern web API suddenly became reachable from any JavaScript client running in a browser.

Only recently have we begun to understand the effects of this massive innovation. It is one of the drivers propelling the JAMstack as one of the most relevant architectural styles for websites and applications. Suddenly, all of the web has become something like a giant operating system. We're now seeing APIs exposed for anything—from payments or subscriptions over advanced machine learning models running in massive cloud environments, to services that directly affect the physical world like shipping or ride-sharing services, and just about anything else you can imagine. Think it up, and there's probably some API for it out there.

Markup

At the core of the web is HTML. The HyperText Markup Language, known and understood by all browsers, defines how content should be structured and distributed. It defines which resources and assets should be loaded for a site and presents a Document Object Model (DOM) that can be parsed, presented, and processed by anything from standard web browsers over screen readers, to search engine crawlers, to voice-controlled devices or smart watches.

In the early days of the web, a website was simply a folder with HTML files exposed over HTTP by a web server. As the web evolved, we began moving to a model in which a running program on the server would build the HTML on the fly for each visit, normally after consulting a database.

This was a much slower, more complex process than serving static assets, but it was also the only viable way to work around the fact that browsers were simple document viewers at the time. Responding to any user interaction, be it a click or form submission, required an entirely new page of HTML to be assembled on the server. This was true whether you were building a comments feature for a blog, an ecommerce store, or any kind of web application. Because of this document-centric architecture, web experiences often felt much less responsive than desktop applications.

Of course, this was before the rise of JavaScript and the emergence of cross-domain web APIs available from the browser. Today the constraints that led to the legacy architectures of the web have gone away.

Prebuilding Markup

Markup on the JAMstack is delivered by a different model. It is not delivered from traditional frontend web servers tasked with building pages at runtime.

Instead, the JAMstack approach is to prebuild all the markup up front and serve it directly to the browser from a CDN. This process typically involves a build tool (sometimes a static site generator like Hugo or Gatsby; sometimes a frontend build tool like Webpack, Brunch, or Parcel) where content and templates are combined into HTML, source files are transpiled or compiled into JavaScript, and CSS runs through preprocessors or postprocessors.

This creates a strict decoupling between the frontend and any back-end APIs. It allows us to remove a large set of moving parts from the infrastructure and live system, which makes it much easier to consider the frontend and individual APIs involved in isolation. The more we can bake, not fry, the content layer of our site, the more simple deployments, scaling, and security become, and the better the performance and end-user experience will be.

In many ways this decoupled architecture is similar to the architecture of mobile apps. When the iPhone introduced mobile apps (originally because traditional web apps were not performant enough), it was simply not a consideration to build a model in which the full UI would be reloaded from a server every time the user took some action. Instead, IOS introduced a model in which each app could be distributed to users as an application package with a declarative UI, including any needed assets to communicate with JSON or XML-based web APIs from there.

The JAMstack approach is similar. The frontend is the app. It is distributed to browsers directly from a CDN on a globally distributed network. Through techniques like service workers, we can even install our frontend directly on end users' devices after their first visit.

Microservices and serverless functions

Another thing happened in tandem with starting to split the frontend and API layers: microservice-based architectures. Backend teams began moving away from one large monolithic API toward much smaller services with more narrowly defined responsibilities. This move toward microservices has culminated in Amazon Web Services (AWS) Lambda and the concept of cloud functions, whereby simple functions doing just one thing and nothing more can be exposed as microweb API endpoints.

This has triggered an important architectural shift. Before, a typical SPA had just one API through which it would communicate. Now each SPA will typically communicate with many different APIs or microservices. As the surface area of each of these APIs shrinks, and standards like OAuth and JWT allow developers to tie them together, the individual API tends to become more and more reusable.

The amount of fully managed, reusable APIs available as managed services will only keep growing. We're now beginning to see the first function app stores, helping to lift the burden of maintenance involved in running a modern application.

Types of JAMstack Projects

Development teams all over the world have deployed a wide variety of JAMstack-powered projects, from smaller sites and landing pages to complex web applications, to large enterprise web properties with thousands of pages.

HTML Content

The simplest JAMstack sites are pure static sites: a folder with HTML and supporting assets (JavaScript, Cascading Style Sheets [CSS], images, font files) and no moving parts; plain-text files that can be edited directly in an editor of choice and kept under version control.

As soon as there is much more than a single page, however, it makes sense to extract common elements like navigations, headers, footers, and repeated elements into their own templates and partials. For anything but the simplest JavaScript, you can pull modules from npm and use more modern ES6 features. After you have a slightly more complex stylesheet, you can introduce automatic vendor prefixing when needed, support for CSS variables across browsers, and nesting of rules.

A proper build tool chain solves all of these while keeping the basic simplicity of a static site. Everything lives in text files in a Git repository, is under version control, and can be manipulated with all of our text-centric developer tools. With a modern continuous deployment (CD) workflow, publishing is as simple as a Git push.

Content from a CMS

For most of the life of the web, we've taken for granted that our CMS determines our entire development platform and couples rendering, content editing, and our plug-in ecosystem tightly together. But the advent of *headless CMSs*—whether they are API-driven or Git-based—means that we can think of content editing UIs as completely separated from the frameworks and tools which we use to render and deliver our websites.

Content management can also be handled by Git-centric content editing tools like Netlify CMS, Prose, Forestry, or CloudCannon. Or, you can use the build tool to make the first build step a content synchronization with an external content API like Contentful, Prismic, or DatoCMS.

Web Applications

In the early 2000s, Microsoft discretely added a homegrown API to Internet Explorer called XMLHttpRequest (XHR). Before this happened, the only way a browser could initiate a page load was by navigating to a new URL that would load a completely new web page from scratch.

After years of XHR going more or less unnoticed, Jesse James Garrett from the agency Adaptive Path wrote an article in February

2005 that would change the web: “Ajax: A New Approach to Web Applications.” In the article, Jesse coined the term “Ajax” and gave web developers a way to communicate about the new patterns that emerged after XHR was used to fetch data from servers directly from JavaScript—similar to how the JAMstack terminology now lets us talk about a new architecture under the banner of a unifying nomenclature.

In today’s world of ubiquitous frontend-heavy web applications, it’s difficult to absorb just how revolutionary it was to move from relying on complete page refreshes as the only way of truly interacting with a web application to suddenly having the option of initiating HTTP requests straight from JavaScript. The realization that Ajax requests could be used to build web applications without the reliance on full page loads changed the way we thought about the web and also pushed the browser market to start fundamentally reinventing itself after years of IE6-based stagnation.

Initially, Ajax was mostly used with principles like progressive enhancement in which a web application would be built in a way that all state was handled on the server through full page transitions but where browsers with JavaScript support could enhance the interactions through partial page refreshes instead.

Separating the frontend from the backend

True SPAs came about when JavaScript performance became good enough that we could invert the existing model and handle all page transitions directly in the browser without any server-side round trips. The first generation of SPAs was typically built by developing a frontend inside large, monolithic database-driven applications. It was common to work on stacks with Ember frontends inside Rails apps, using the Rails asset pipeline for bundling and minification (even if this flow was mostly painful to work with).

Modern SPAs began to separate the frontend and backend completely, and a whole new generation of frontend build tools emerged to make working with the frontend in isolation truly joyful—after the initial setup and configuration hurdles have been overcome, of course. Webpack is the most commonly used build tool for SPAs today, offering a flow that supports JavaScript and CSS transpiling, postprocessing, integrated bundling and code splitting, as well as real-time browser refreshes on any save.

Large Web Properties

When it comes to prebuilding markup, there are obvious constraints in terms of the number of pages it is practical to generate. The build and deployment cycle simply becomes too long to allow for a viable workflow. If you need to publish a story before a competitor, but you have a one-hour build and deployment cycle between the moment you press publish in your CMS and the moment the story is live, your stack has fundamentally let you down.

For some kinds of publishing, this is such a tight constraint that even a 10-minute build and deploy cycle would be better served by another stack work.

However, build tools have become surprisingly fast, and the boundaries of what can be achieved with a build cycle is shrinking. A static-site generator like Hugo can process thousands of pages a second and provide much faster build cycles. Such impressive performance improvements in these tools, along with emerging techniques to tackle very large sites by carefully splitting site generation into separate branches so that only a small subset of the site is rebuilt for each update, combine to provide great results.

In the end, the trade-off will need to be based on what's most important: performance, uptime, and scalability, or shortening the publishing cycle.

For the majority of content-driven sites, as long as a deploy can be scheduled to go live at a certain time, being able to run a build cycle in less than 10 minutes is likely not of real importance. Unless your site is truly gigantic (pages counted in millions rather than thousands), a JAMstack approach might be a more viable option than you would have once thought.

Hybrid Experiences

It's common for a company to require a variety of web experiences: a primary website, a blog, a documentation site, a web application, an ecommerce store, and so on. Maintaining a common look and feel across all these applications used to be extremely challenging for web teams. That's because on traditional architectures, the frontend of each application is tightly coupled to the backend used to generate it on the server.

The JAMstack solves for this in an elegant, highly scalable way. We can build one, decoupled frontend that spans across all applications. You'll see this in action later when we present the *Smashing Magazine* case study.

Summary

The list of projects that wouldn't be a fit for a JAMstack architecture is continually getting shorter. Improvements in the tooling and automation, the workflows, best practices, and the ecosystem as a whole are creating more possibilities for what a JAMstack site can achieve.

Now that we've covered what the JAMstack is and whether it's right for your project, it's time to talk about the advantages that the JAMstack approach can provide.

CHAPTER 3

Advantages of the JAMstack

Simplifying Systems and Simplifying Thinking

Thanks to the innovators who have built layers of abstraction over the low-level bits and bytes that computers use to operate, the amount of work that is possible by a single developer is now astronomical.

Abstractions, however, are leaky. Sometimes, the lower level of the stack suddenly makes itself known. You could be working on a high level of object-oriented programming, developing a database-driven application with an Object-Relational Mapping (ORM) when suddenly a low-level memory management issue, deep in a compiled dependency, opens up a buffer overflow that a clever malicious user can abuse to hijack the database and completely take over.

Unless we find ways to build solid firewalls between the different layers of the stack, these kinds of issues will always be able to blast through from the lower layers or the far points in the dependency chain.

Fortunately, the JAMstack creates several such firewalls to manage complexity and focus attention on smaller parts in isolation.

Better Comprehension and Mental Agility

With server-side rendering, *all* possible layers of the stack are always involved in any of the requests a user makes to a site or application.

To build performant and secure software, developers must deeply understand all of the layers that requests flow through, including third-party plug-ins, database drivers, and implementation details of programming languages.

Decoupling the build stage completely from the runtime of the system and publishing prebaked assets constructs an unbreakable wall between the runtime environment end users interact with and the space where code runs. This makes it so much easier to design, develop, and maintain the runtime in isolation from the build stage.

Separating APIs into smaller microservices with a well-defined purpose means that our ability to comprehend the particulars of each service in isolation becomes much simpler. Borders between services are completely clear.

This helps us to establish clearer mental models of the system as a whole while being liberated from the need to understand the inner complexities of each individual service and system. As a result, the burden of development and maintenance of the system can be significantly lightened. The areas where we must direct our attention can be more focused with confidence that the boundaries between the services being used are robust and well defined.

There's no need to understand runtime characteristics of JavaScript in different browsers as well as the flows of an API at the same time.

You Don't Need to Be an Expert at Everything

In recent years, the complexity of frontend architectures has exploded as browser APIs have evolved and HTML and CSS standards have grown. However, it's very difficult for the same person to be both an expert in client-side JavaScript performance and server-side development, database query constraints, cache optimization, and infrastructure operations.

When we decouple the backend and frontend, it makes it possible for developers to focus on just one area. You can run your frontend locally with an autorefreshing development server speaking directly to a production API, and make switching between staging, production, or development APIs as simple as setting an environment variable.

And with very small and super-focused microservices, the service layer becomes much more reusable and generally applicable than when we have one big monolithic application covering all of our needs.

This means that we're seeing a much broader ecosystem of ready-made APIs for authentication, comments, ecommerce, search, image resizing, and so on. We can use third-party tools to outsource complexity and rest easy in the knowledge that those providers have highly specialized teams focusing exclusively on their problem space.

Reduce Moving Parts at Runtime

The more that we can prebuild, the more that we can deploy as prebaked markup with no need for dynamic code to run on our servers during a request cycle, and the better off we will be. Executing zero code will always be faster than executing some code. Zero code will always be more secure than even the smallest amount of code. Assets that can be served without any moving parts will always be easier to scale than even the most highly optimized dynamic program.

Fewer moving parts at runtime means fewer things that could fail at a critical moment. And the more distance we can put between our users and the complexity inherent in our systems, the greater our confidence that they will experience what we intended. Facing that complexity at build time, in an environment that will not affect the users, allows us to expose and resolve any problems that might arise safely and without a negative impact.

The only reason to run server-side code at request time ought to be that we absolutely cannot avoid it. The more we can liberate ourselves from this, the better the experience will be for our users, operations teams, and our own ability to understand the projects we're working on.

Costs

There are a variety of costs associated with designing, developing, and operating websites and applications, and they are undoubtedly influenced by the stack we choose. Although financial costs are often most obvious, we should also consider costs to the developer experience, creativity, and innovation.

Financial Costs

Any web development project of significant scale will likely include an exercise to estimate the anticipated or target traffic levels and then a plan for the hosting infrastructure required to service this traffic. This capacity planning exercise is an important step for determining how much the site will cost to operate. It is difficult to estimate how much traffic a site will receive ahead of time, so it is common practice to plan for sufficient capacity to satisfy traffic levels beyond even the highest estimates.

With traditional architectures, in which page requests might require activity at every level of the stack, that capacity will need to extend through each tier, often resulting in multiple, highly specified servers for databases, application servers, caching servers, load balancers, message queues, and more. Each of these pieces of infrastructure has an associated financial cost. Previously, that might have included the cost of the physical machines, but today these machines are often virtualized. Whether physical or virtual, the financial costs of these pieces of infrastructure can mount up, with software licenses, machine costs, labor, and so on.

In addition, most web development projects will have more than one environment, meaning that much of this infrastructure will need to be duplicated to provide suitable staging, testing, and development environments in addition to the production environment.

JAMstack sites benefit from a far simpler technical architecture. The burden of scaling a JAMstack site to satisfy large peaks in traffic typically falls on the Content Delivery Network (CDN) that is serving the site assets. Even if we were not to employ the services of a CDN, our hosting environment would still be dramatically simplified when compared to the aforementioned scenario. When the process of accessing the content and data and then populating page templates is decoupled from the requests for these pages, it means that the demands on these parts of the infrastructure is not influential to the number of visitors to the site. Large parts of the traditional infrastructure either do not need to be scaled or perhaps might not even need to exist at all.

The JAMstack *dramatically* reduces the financial costs of building and maintaining websites and applications.

Team Efficiency

The size and complexity of a project's architecture is directly proportional to the quantity of people and range of skills required to operate it. A simplified architecture with fewer servers requires fewer people and far less specialization.

Complex DevOps tasks are largely removed from projects with simpler environments. What were once time-consuming, expensive, and critical procedures—like provisioning new environments and configuring them to faithfully replicate one another—are replaced with maintenance of only the local development environments (required with a traditional approach, anyway) and the deployment pipeline to a productized static hosting service or CDN.

This shift places far more power and control in the hands of developers, who are in possession of an increasingly widespread set of web development skills. This reduces the cost of staffing a web development project (through a reduced demand for some of the more exotic or historically expensive skills) and increases the productivity of the developers employed. By having working knowledge of a larger portion of the stack and fewer discipline boundaries to cross, each developer's mental model of the project can be more complete and, as a result, each individual can be more confident in their actions and be more productive.

This also lowers the boundaries to innovation and iteration.

The Price of Innovation

When making changes to a site, we need to be confident of the effect our changes might have on the rest of the system.

The JAMstack taked advantage of APIs and an architecture of well-defined services with established interface layers, moving us toward a system that embraces the mantra of "small pieces, loosely joined." This model of loose coupling between different parts of a site's technical architecture lowers the barriers to change over time. This can be liberating when it comes to making technical design decisions because we are less likely to be locked into one particular third-party vendor or service, given that we have well-defined boundaries and responsibilities across the site.

This also gives development teams freedom to refactor and develop the particular parts of the site that they control, safe in the knowledge that as long as they honor the structure of the interfaces between different parts of the site, they will not compromise the wider project.

Whereas monolithic architectures stifle the ability to iterate, with tight coupling and proprietary infrastructures, the JAMstack can break down these boundaries and allow innovation to flourish.

Of course, it is still possible to design a JAMstack site in a way that creates a tightly coupled system or that has many interdependencies that make change difficult. But, we can avoid this because the simplification of the required infrastructure leads to greater clarity and understanding of the constituent parts of the site.

We just discussed how reducing the moving parts of the system at runtime leads to greater confidence in serving the site. This also has a significant impact on reducing the cost of innovation. When any complexity can be exposed at build time rather than at runtime, we are able to see the results of our modifications in far greater safety, allowing for greater experimentation and confidence in what will ultimately be deployed.

From simplification of the system to richer mental models and greater empowerment in development teams, with the JAMstack, the cost of innovation can be significantly reduced.

Scale

The ability for hosting infrastructure to be able to meet the demands of a site's audience is critical to its success. We talked about the cost associated with planning and provisioning hosting infrastructure on traditional stacks and then the much simpler demands when using the JAMstack. Now, let's look more closely at how the JAMstack can scale, and why it is so well suited for delivering sites under heavy traffic loads.

Even with the most rigorous capacity planning, there are times when our sites (hopefully) will experience even more attention than we had planned for, despite all of our contingency and ambition.

The use of the word "hopefully" in the previous sentence is a little contentious. Yes, we want our sites to be successful and reach the widest audience that they can. Yes, we want to be able to report record visitor levels from our analytics and have a wild success on our hands. But infrastructure teams regularly fear that their sites could become too popular. That instead of a healthy, smooth level of demand, they receive huge spikes in traffic at unexpected times. The fear of being at the top of *Hacker News* features heavily on capacity planning sessions, although it is repeatedly being changed for whichever site or social media property might be the latest source of excessive traffic levels due to a site going viral.

The advent of virtualized servers has been a huge step forward in addressing this challenge, with techniques available for monitoring and scaling virtual server farms to handle spikes in traffic. But again, this adds complexity to technical design and maintenance of a site.

Aping Static in Order to Scale

One technique employed by many sites designed and engineered to cope with high traffic levels (both sustained and temporal) is to add a caching layer and perhaps also a CDN to their stack.

This is prudent. And it's also where the JAMstack excels.

Sites built on traditional stacks need to manage their dynamically created content into their various caching layers and CDNs. This is a complex and specialized set of operations. The complexity and resulting cost has lead to a perception that CDNs are the domain of large, enterprise sites with big teams and big budgets. The tooling they put in place effectively takes what is dynamic and propagates that to caching and CDNs as sets of static assets so that they can be served prebaked without a round trip to the application server.

In other words, dynamic sites add an extra layer of complexity just to allow them to be served with static hosting infrastructure to satisfy demand at scale.

Meanwhile, with the JAMstack we are already there. Our build can output exactly the kinds of assets needed to go directly to the CDN with no need to introduce additional layers of complexity. This is not the domain of large enterprise sites, but within reach of anyone using a static site generator and deploying their sites to one of a multitude of CDN services available.

Geography

In addition to the volume of traffic a site might receive, we also must consider the geographical location of our site's visitors. If visitors are situated at the opposite side of the world to where our servers live, they are likely to experience diminished performance due to latency introduced by the network.

Again, a CDN can help us meet this challenge.

A good CDN will have nodes distributed globally, so that your site is always served to a visitor from the closest server. If your chosen architecture is well suited to getting your site into the CDN, your ability to serve an audience anywhere in the world will be vastly improved.

Redundancy

When designing web architectures, we aim to minimize the single points of failure (SPoFs). When sites are being served directly from a CDN, our own servers no longer act as a SPoF that could prevent a visitor from reaching the site. Moreover, if any individual node within a global CDN fails (in itself a pretty major and unlikely event), the traffic would be satisfied by another node, elsewhere in the network.

This is another example of the benefit of putting distance between build environments and serving environments, which can continue to function and serve traffic even if our own build infrastructure were to have an issue.

Resiliency, redundancy, and capacity are core advantages of delivering sites with the JAMstack.

Performance

Performance matters. As the web reaches more of the planet, the developers building it must consider varying network reliability and connectivity speeds. People expect to be able to achieve their online goals in an increasing variety of contexts and locations. And the types of devices that are being used to access the web have never been more diverse in terms of processing power and reliability.

Performance can be interpreted to mean many things. In recent years, the value of serving sites quickly and achieving a rapid *time to first meaningful paint* and *time to interactive* has been demonstrated to be critical to user experience (UX), user retention, and conversion. Simply put, time is money. And the faster websites can be served, the more value they can unlock.

There have been many case studies published on this. You can find some staggering results from web performance optimization listed on *https://wpostats.com/*, some showing marginal performance yielding impressive improvements, like these:

- "COOK increased conversion rate by 7% after cutting average page load time by 0.85 seconds. Bounce rate also fell by 7% and pages per session increased by 10%."

  —NCC Group
- "Rebuilding Pinterest pages for performance resulted in a 40% decrease in wait time, a 15% increase in SEO traffic, and a 15% increase in conversion rate to signup."

  —Pinterest Engineering
- "BBC has seen that they lose an additional 10% of users for every additional second it takes for their site to load."

  —BBC

While others boast staggering results:

- "Furniture retailer Zitmaxx Wonen reduced their typical load time to 3 seconds and saw conversion jump 50.2%. Overall revenue from the mobile site also increased by 98.7%."

  —Google and Zitmaxx Wonen

Where to Focus Optimization Efforts

For a long time, performance optimizations for websites focused primarily on the server side. So-called backend engineering was thought to be where serious engineering happened, whereas frontend code was seen by many as a less-sophisticated field. Less attention was paid to performance optimizations in the frontend, client-side code.

This situation has changed significantly. The realization that efficiencies in the architecture and transmission of frontend code could make a staggering difference to the performance of a site has created an important discipline in which measurable improvements can be made. Much proverbial low-hanging fruit was identified, and today there is an established part of the web development industry focused on frontend optimization.

In the frontend, we see incredible attention being given to things like the following:

- Minimizing the number of HTTP requests required to deliver an interactive site
- Page architectures designed to avoid render blocking or dependencies on external sources
- Use of image and visualization techniques that avoid heavy videos and images
- Efforts to avoid layout thrashing or slow paint operations

With these, and countless other areas, the work of optimizing for performance in the browser has gained tooling, expertise, and appreciation.

But the performance of the web-serving and application-serving infrastructure continues to be critical, and it's here where we see efforts to optimize through initiatives like the following:

- Adding and managing caching layers between commonly requested resources
- Fine-tuning database queries and designing data structures to minimize bottlenecks
- Adding compute power and load balancing to increase capacity and protect performance under load
- Creating faster underlying network infrastructure to increase throughput within hosting platforms

Each area is important. Good web performance requires a concerted effort at all levels of the stack, and a site will only be as performant as its least performant link in the chain.

Critically though, the JAMstack removes tiers from that stack. It shortens that chain. Now operations on which teams used to spend large amounts of time and money to optimize in an effort to speed them up and make them more reliable don't exist at runtime at all.

Those tiers and operations that do remain can now receive more scrutiny than before because fewer areas exist to compete for attention and benefit from our best optimization efforts.

However, there is no code that can run faster than zero code. This kind of simplification and separation of the *building* of sites from the *serving* of sites yields impressive results.

Consider a typical request life cycle on a relatively simple, but dynamic, site architecture:

1. A browser makes a request for a page.
2. The request is handled by a web server that inspects the URL requested and routes the request to the correct piece of internal logic to determine how it should be satisfied.
3. The web server passes the request to an application server that holds logic on which templates should be combined with data from various sources.
4. The application server requests data from a database (and perhaps external systems) and renders this into a response.
5. The response is passed back from the application server to the web server and then on to the browser where it can be displayed to the user.

Figure 3-1 shows this process.

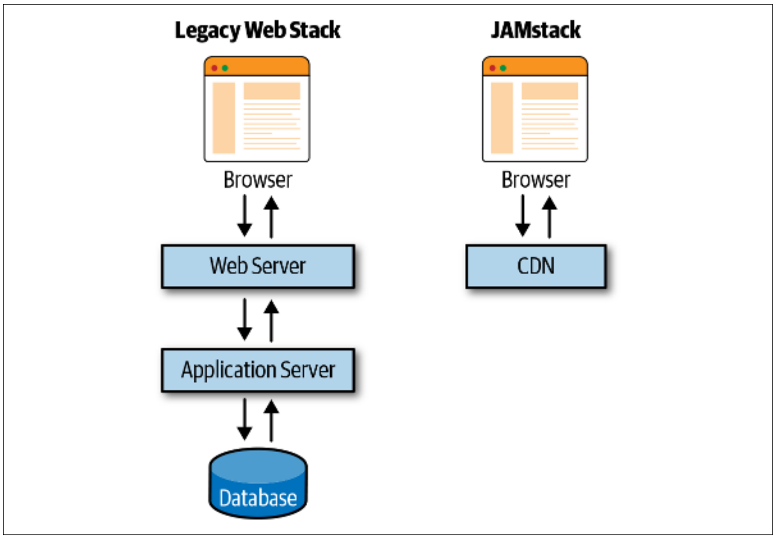


Figure 3-1. Legacy web stack versus JAMstack

In systems like the one just described, with a dynamic backend, it has become common to introduce additional layers in order to help improve performance. We might see the introduction of a caching layer between the webserver and application server, or indeed between the application server and databases. Many of these layers actually introduce operations that behave as if that part of the system were static but need to be managed and updated by the system over the course of operation.

Now let's consider the same scenario for the JAMstack, also shown in Figure 3-1, in which the compilation of page views is not carried out on demand, but ahead of time in a single build process:

1. A browser makes a request for a page.
2. A CDN matches that request to a ready-made response and returns that to the browser where it can be displayed to the user.

Right away we can see that a lot less happens in order to satisfy each request. There are fewer points of failure, less logical distance to travel, and fewer systems interacting for each request. The result is improved performance in the hosting infrastructure *by default*. Assets are delivered to the frontend as quickly as possible because they are ready to be served, and they are already as close to the user as is possible thanks to the use of CDN.

Only Turning the Gears When Necessary

This significantly shorter request/response stack is possible because the pages are not being assembled per request; instead, the CDN has been primed with all of the page views that we know our site will need.

Again, we are benefitting from decoupling the generation and population of pages from the time that they are needed. They are ready to be served at the very moment they are requested.

In addition, we can enjoy being forewarned of any issues that might arise during the generation of these pages. Should such an issue occur in a site where the pages were being generated and served on demand, we would need to return an error to the user.

Not so in a JAMstack site for which the generation of pages happens ahead of time during a deployment. There, should a page generation operation fail, the result would be a failed deployment which would never be conveyed to the user. Instead, the failure can be reported and remedied in a timely fashion. Without any users ever being affected or aware.

Security

The reduced complexity of the JAMstack offers another advantage: a greatly improved security profile for our sites.

Reducing Surface Area

When considering security, the term *surface area* is often used to describe the amount of code, of infrastructure, and the scale of the logical architecture at play in a system. Fewer pieces of infrastructure means fewer things to attack and fewer things on which you need to spend effort patching and protecting. Reducing surface area is a good strategy toward improving security and minimizing avenues for attack.

As we have already learned, the JAMstack benefits from a smaller, simplified stack when compared to traditional architectures with their various servers and databases, each needing the ability to interact with one another and exchange data. By removing those layers, we also remove the points at which they interact or exchange data.

Opportunities for compromising the system are reduced. Security is improved.

Reducing Risk with Read-Only

Beyond improving security by reducing the surface area, JAMstack sites enjoy another benefit. The pieces of infrastructure that remain involved in serving our sites do not include logical code to be executed on our servers at request time. As much as is possible, we avoid having servers executing code or effecting change on our system.

In *read-only* systems, not only are scale and performance considerations improved, but our security profile is improved.

We can return to the discussion of Wordpress sites as a useful comparison. A Wordpress site combines a database, layers of logic written in PHP, templates for presentation, and a user interface for configuring, populating, and managing the site. This user interface is accessed over the web via HTTP, requiring that a Wordpress site is capable of accepting HTTP POST requests, and consuming and parsing the data submitted to it in those requests.

This opens a popular, and often successful, attack vector to Wordpress sites. Considerable effort has been invested in attempting to secure this route to attack, and adhering strictly to establish good practices can help. But this can't ensure total security.

Speculative, hostile traffic that probes for poorly secured Wordpress administration interfaces is rife on the internet. It is automated, prolific, and continues to improve in sophistication as new vulnerabilities are discovered.

For those looking to secure a Wordpress site, it is a difficult battle to permanently, confidently win.

The JAMstack is different. Instead of beginning with an architecture that allows write operations for servers to execute code, and then trying to secure this by keeping the doors to such operations tightly guarded, a JAMstack site has no such moving parts or doors to guard. Its hosting infrastructure is read-only and not susceptible to the same types of attack.

Except, we're cheating here. We're talking about JAMstack sites as if they were static—both in hosting infrastructure and in user experience. Yet we're also trying to convey that JAMstack sites can be just as dynamic in experience as many other architectures, so what gives?

We have been simplifying a little and skipping over aspects of JAMstack sites which might interact with other systems. The JAMstack certainly includes all manner of rich interactive services that we might use in our sites. We talk about a growing ecosystem of services at our disposal, so how do we reconcile that with regard to security?

Let's look.

Outsourced or Abstracted Services

So far, we've focused on the infrastructure that we, as site owners, would need to secure and operate ourselves. But if we are to create dynamic experiences, there will be times when our sites will need to interact with more complex systems. There are some useful approaches available to us to minimize any negative impact on security. Let's examine some of them.

Outsourcing complexity

Recalling that the "A" in JAMstack stands for APIs, it shouldn't be a surprise that we'd describe advantages in using APIs to interact with external services. By using APIs to allow our sites to interact with a discrete set of services, we can gain access to a wide variety of additional capabilities far beyond what we might want to provide ourselves. The market for capabilities *as a service* is broad and thriving.

By carefully selecting vendors that specialize in delivering a specific capability, we can take advantage of their domain expertise and outsource to them the need for specialist knowledge of their inner workings. When such companies offer these services as their core capabilities, they shoulder the responsibilities of maintaining their own infrastructure on which their business depends.

This model also makes for better logical separation of individual services and underlying capabilities, which can be advantageous when it comes to maintaining a suite of capabilities across your site. The result is clear separation of concerns in addition to security responsibilities.

Abstraction and decoupling

Not all capabilities can be outsourced and consumed via APIs. There are some things that you need to keep in-house either because of business value or because external services aren't able to satisfy some more bespoke requirements.

The JAMstack is well placed to minimize security implication here, too. Through the decoupling of the generation of sites through prerendering and the serving of our sites after they have been generated and deployed, we put distance between public access to our sites and the mechanisms that build it. Earlier, we urged caution when providing access to Wordpress sites backed by a database, but it is highly likely that some parts of our sites might need to access content held in a database. If access to this database is totally abstracted away from the hosting and serving of our sites, and there is no public access to this resource (because it is accessed through a totally decoupled build process), our security exposure is significantly reduced.

JAMstack sites embody these principles:

- A minimal surface area with largely read-only hosting infrastructure
- Decoupled services exposed to the build environment and not the public
- An ecosystem of independently operated and secured external services

All of these principles improve security while reducing the complexity we need to manage and maintain.

For the Developer; For the Project; For the Win

Throughout this chapter we've talked about the advantages that come from reduced complexity. We've talked about how removing common complexity can reduce project costs, improve the ability to scale and serve high volumes of traffic, and improve security.

All of these are good news for a project. They are good for the bottom line. They improve the likelihood of a project being approved to proceed and its chances of prolonged success. But if this comes at the cost of a sound development experience, we have some weighing up to do.

Really? Isn't that being overdramatic?

Developer experience is an important success factor. The ability for a developer to effectively deliver on the promise of the design, the strategy, and the vision can often be the final piece of the puzzle. It's where ideas are realized.

A poor development experience can be devastating for a project. It can impede development progress and create maintenance implications that damage the long-term health of a project. It can make it more difficult to recruit and to retain the people you need to deliver a project. It can generate frustrations at slow progress or poor reliability. It can choke off any ability to innovate and make a project great.

A good developer experience can help to create high productivity, a happy team, and innovation at all levels of a project. It is something to strive for.

But of course, we can't sacrifice user experience for developer experience. Jeremy Keith has deftly articulated this on a number of occasions, notably here:

> Given the choice between making something my problem, and making something the user's problem, I'll choose to make it my problem every time.
>
> —Jeremy Keith, Needs Must, 2018

What we need is an architecture and an approach to development that somehow satisfies both of these criteria. The JAMstack can deliver this for us.

The simplification that we have heralded so many times in this chapter does not come at the cost of clarity. It does not introduce obfuscation or opaque magic. Instead, it employs tools, conventions, and approaches that are both popular and increasingly available among web developers. It embraces development workflows designed to enhance a developer's ability to be effective and to build things in familiar but powerful environments. It creates strong logical boundaries between systems and services, creating clear areas of focus and ownership. We need not choose between an effective developer experience and an effective user experience. With the JAMstack we can have both.

How? Let’s move on to look at some best practices.

CHAPTER 4

Planning for the JAMstack

This section covers the JAMstack end-to-end setup and highlights the choices you'll make along the way to create the appropriate setup for your team. These decisions include the following:

- How you'll manage the project/code
- Where you'll store your content
- What software will build your site
- How your build process will be automated
- Where your site will be published
- What services and APIs your live site will use

Let's look at each one in turn.

Setting Up the Project

With no server or database to configure, a JAMstack project is largely a simple folder structure full of text files. The contents of your JAMstack project will usually include the following:

- A source folder to store your content files
- A folder for layouts and templates
- A configuration file containing the settings for the build process
- A static site generator, usually added as a dependency

- A destination folder to save the final output

You can manage your entire project locally, directly on your computer, but it's always best to begin by setting up a hosted Git repository. Even if you are the only site author, starting a repository will give you an online backup, store the history of the project, and eventually allow others to contribute. And in a moment, you'll also see how a hosted repository is key to triggering new builds whenever a change is pushed.

Rarely would you set up your entire project folder from scratch because most site generators have sample projects that you can download from GitHub as a starting point. But before we talk about site generators, let's cover the various ways to manage content on the JAMstack.

Strategies for Managing Content

Every website begins with content. On Content Management System (CMS) systems like Drupal or Wordpress, sites manage content in a database, and editors use an admin interface to write and save the content to the database.

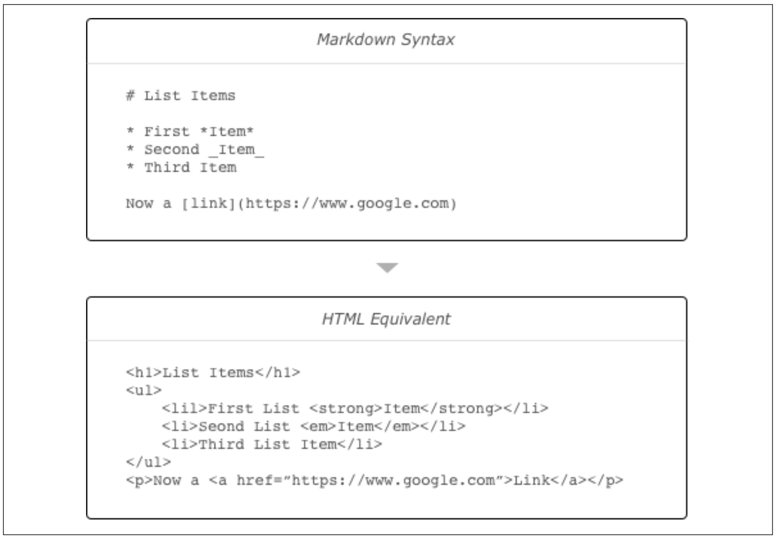
On the JAMstack, there are actually a few content approaches.

Text Files

For many JAMstack sites, content couldn't be more straightforward: it's simply a folder full of files that lives directly within the project with the templates, plug-ins, and other assets. Creating a new file in the content directory will create a new page after the site is published, and changes are made by editing the content files in a text editor and then committing the changes to the Git repository. The name of each file and the folder it is in will determine its URL path after the site is generated.

What does one of these content files look like? It can be straight HTML—though usually paired down to only the unique parts of the page. Common elements like headers and footers then are added by the site generator during the build step.

Authoring site content in pure HTML can still be tedious, however. This is why Markdown, shown in Figure 4-1, a much-simplified syntax, has become the most popular option for writing content. If you’ve used Slack and surrounded a word with asterisks to *bold* it, you’ve used a modified form of markdown.



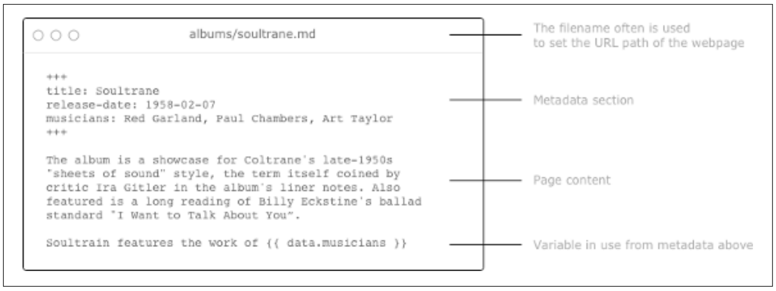
```
# List Items

* First *Item*
* Second _Item_
* Third Item

Now a [link](https://www.google.com)
```

```
<h1>List Items</h1>
<ul>
    <li1>First List <strong>Item</strong></li>
    <li>Seond List <em>Item</em></li>
    <li>Third List Item</li>
</ul>
<p>Now a <a href="https://www.google.com">Link</a></p>
```

Figure 4-1. Markdown and the resulting HTML output

Most common CMSs have additional settings for each page; for example, which layout to use or which category and tags the page should be assigned.

To store details such as these, static site generators have standardized on a top section of each file called the metadata. Often written in YAML, it’s a simple key/value collection of settings and other data that apply to the page, as demonstrated in Figure 4-2.


albums/soultrane.md
The filename often is used to set the URL path of the webpage
Metadata section
Page content
Variable in use from metadata above

```
+++
title: Soultrane
release-date: 1958-02-07
musicians: Red Garland, Paul Chambers, Art Taylor
+++

The album is a showcase for Coltrane's late-1950s
"sheets of sound" style, the term itself coined by
critic Ira Gitler in the album's liner notes. Also
featured is a long reading of Billy Eckstine's ballad
standard "I Want to Talk About You".

Soultrain features the work of {{ data.musicians }}
```

Figure 4-2. A sample markdown file with metadata

Git-Based CMS

It's likely that not all your site contributors will be developers, and using Git and text files may be an unfamiliar experience. This is actually the reason why database-backed CMSs like Wordpress were originally created. However, you probably also have experienced how a complex CMS can create artificial separation between the content editors working in the database and the developers building the code and templates.

Because every web page ultimately ends up as a combination of content *and* markup, it would be nice if you could manage them both by the same process.

There's now a new generation of CMS software does exactly this, managing content as files in Git rather than a database. Usually included in your project as a single-page application (just an HTML file with some JavaScript dependencies), a Git-based CMS allows users to log in to a familiar admin interface to edit content and preview and save changes.

Here's the clever bit: a saved change in the CMS is really made as a Git contribution behind the scenes. That's powerful in that developers and editors are working in the same Git-based workstream, modifying the same files using whichever method is most comfortable to them.

NetlifyCMS.org is an open source CMS that works in this manner. Forestry.io is another.

Headless CMS

This is a category of hosted services that provide a rich online interface for managing and editing content.

"Headless" refers to the idea that these services manage only the content, leaving it up to the developer to determine how and where the site is built and hosted. Larger projects use a headless CMS as a central content store behind their entire collection of sites and mobile apps.

All headless CMS services provide an API that is used to pull the most recent content during the build step (and also dynamically using JavaScript as the site runs).

For example, you might use a headless CMS for your recruiting team to manage job postings. The static site generator would then query the API during the build step, receiving a list of the job postings as JSON. It would then use templates to create a unique HTML page for each posting, plus, of course, an index page listing all the postings together.

Self-Hosted CMS

Running the JAMstack does not preclude you from using something more traditional like Wordpress or Drupal to manage content. In recent versions, both of these popular CMSs have added APIs to access the content they contain. So, much like the aforementioned headless CMS service, you can connect your content to your JAMstack site using the API.

In this way, you can continue to manage content in a familiar CMS but now prebuild the site to dramatically speed up performance by serving rendered pages from a CDN. And becauseWordpress and Drupal would no longer need to be public facing, you have many more options to keep them secure.

Choosing a Site Generator

With content managed using one or more of the just described techniques, it's time to think of how it is transformed into production HTML.

In the beginning, site generators were almost synonymous with Jekyll, the first site generator to gain wider traction and popularity. Jekyll was created by Tom Preston-Werner, GitHub's cofounder, using the Ruby programming language.

Today, site generators are available in almost every language, including Ruby, Python, PHP, JavaScript, and Go. The website staticgen.com is a community-curated directory of static site generators, with more than 30 languages represented. Most all site generators are both free and open source.

Perhaps the most difficult part of getting started with the JAMstack is picking the appropriate site generator from the ever-expanding list of options. The staticgen.com website shows the popularity of each generator, but beyond what others think, there are some real factors that you should consider that will flavor your experience with the JAMstack.

One: The Type of Website

You'll find each generator is created by its authors with a particular type of website or application in mind. Some generators are designed for websites with complex taxonomy and thousands of pages, others specifically for blogs, others for documentation, others for progressive web apps (PWAs), and richly interactive apps for which most all content and navigation is loaded and handled via JavaScript. Our advice is to check out sample projects created with each static site generator to see if they match your own intended results.

Two: The Programming Language Used

It's very possible to use a site generator without knowing the programming language used to create it. For example, even if you know nothing about the Go programming language, Hugo is a great choice because you can download and install it as a binary application with nothing else to set up and no programming knowledge required.

Jekyll also does not require any specific Ruby experience to make good use of it, but you first need to have a working Ruby installation to get it to run on your computer. Gatsby, another popular site generator that uses the React JavaScript framework, requires you to have a recent version of Node.js installed. You get the idea: a little experience with the tooling around the language can help.

It's not a bad idea to consider a site generator written in a language your team finds familiar. As your site grows more complex, you might find the need for your own custom plug-ins and transformations. Familiarity with the generator's language can help you modify and extend its functionality.

Three: The Templating Syntax

A majority of your JAMstack development will be spent creating and modifying HTML templates, CSS, and JavaScript. Take a close look at the templating languages your chosen static site generator supports. All templating languages augment traditional HTML markup with new functionality like reusable code blocks, if statements, and loops. However, the exact syntax used varies and is largely a matter of personal preference. Some developers prefer the minimalist style of a language like Jade or Pug—both of these forgo brackets and closing tags in favor of indentation. Other developers prefer template languages like Handlebars and Liquid that look as close to traditional HTML as possible.

Four: Speed

The speed at which your site generator can build out each page becomes important as your content grows. A site with a handful of pages can take just milliseconds to build, whereas a site with archives of content can take several minutes to complete building. Because each change you publish will run the build process, speed is an important consideration.

Hugo is notoriously fast at building site pages, thanks to Go's multithreaded architecture. Other site generators, by nature of their design goals, build both static HTML and a JavaScript bundle for each page. The JavaScript will be used to reduce load times for a user navigating around the site after it is on a CDN, but the extra work involved increases build time significantly.

Five: Developer Tooling

The core goal of a site generator is to transform content into HTML, but as you know, a site always contains other assets like CSS, images, and JavaScript. Modern frontend web development is starting to look a lot like traditional software development, with tool chains to prepare and compile assets for production.

Before publishing the site, images often need to be compressed, JavaScript transpiled and minified, and CSS combined or converted from formats such as SASS and LESS. Recently, it's also become common to inline CSS and smaller images directly in the HTML for the performance benefit of the browser not needing to make an extra request to fetch these items.

So, the build of a JAMstack site is often *both* of these steps: creating the HTML and compiling these production assets. Some site generators concern themselves with only the HTML, leaving producing the rest of the assets to another utility like Gulp or Webpack. Other generators look to be more holistic and actually include features to handle compiling JavaScript CSS, and images—especially site generators built on JavaScript frameworks like Nuxt (Vuejs) or Gatsby (React). You'll find a lot of generators actually make use of popular tools like Webpack under the hood.

Linting is also common. This is the process of automatically running tests against the HTML, JavaScript, and CSS to check for errors. This is a fantastic way to discover problems before they bite you in production. Linters are often particular about formatting, too, ensuring that everyone contributing adheres to the same standards.

Finally, developers working locally often will preview changes locally in a rapid-fire session of editing files and viewing changes in a browser. To support this, most site generators include a small web server that can allow you to serve the site locally for testing. Better yet is *hot reloading*, which is the ability for the browser to be updated immediately as a change is saved, with no need to click refresh or lose the current state of your application.

Recommending a site generator is a bit like recommending a computing device: it really depends on what you prefer, what you are familiar with, and how you intend to use it. There's really no clear way to be prescriptive, unfortunately. Also, there is very active work being done in this space—by the time any "Best Of" list is published, it's already out of date.

Luckily, trying a first run of any site generator usually takes only a few minutes. We really advise that you to try several and always browse the documentation for any design goals because those will inform you about the problems the creators are working to solve.

Adding Automation

Taking the time to add in build automation is what makes JAMstack development so productive and enjoyable.

All site generators have a build command that can be run from the terminal. The final assets are usually compiled to a folder named *dist* or *build* and then can be uploaded to almost any server or CDN. It's certainly possible to build and then upload assets manually—especially early on. However, as you do more development and involve more team members, you'll find this approach doesn't scale.

We mentioned earlier that you can use a hosted Git repository for your project to provide an online backup and to facilitate collaboration. But now we explore the third reason: triggering automated builds via webhooks.

GitHub, GitLab, and Bitbucket are the three most popular repository hosting services and they all support webhooks. Webhooks are a simple way for any event on a platform (such as a code push) to trigger an HTTP POST to an external URL. When you set up your webhook, you can determine the URL used. The payload of the POST will contain JSON information about the event. The server receiving the webhook uses that information to determine what action to take.

For JAMstack sites, the most common action is to trigger a new build of the website when anything is added or modified via a push to the Git origin. Some more advanced deployment services will even build previews of pull requests at dedicated URLs. This way, teams can browse and test the changes before merging them. It's also possible for each branch of a repository to be built separately at unique URLs. This is a powerful approach to staging environments and working on feature branches.

Ideally, your automated build process will do the following:

1. Listen for notifications of changes to your Git repository (usually via webhooks).
2. Prepare the build environment and fetch any required dependencies.
3. Fetch remote data from APIs (as needed).
4. Build the site and prepare assets.
5. Publish the final site to a CDN.

Using GitHub Pages and GitLab Pages

If you are hosting a project's repository at GitHub or GitLab, you might consider using GitHub Pages or GitLab Pages. Both are lightweight options that connect to your repositories and allow you to automatically build and host a static website. Both support custom domains.

GitHub Pages was originally launched to make developing sites and documentation for the open source projects they host easier. If you just need a simple website to explain your project, it's a great option. Note that GitHub Pages supports only the Jekyll site generator, and only with specific plug-ins.

GitLab Pages supports a wider variety of static site generators, and you can even build your own continuous integration (CI) pipeline to deploy and host your website. GitLab also can be self-hosted if that's a requirement or interest of your organization.

Using Your Own Infrastructure

If you have existing infrastructure, you might decide to run your own hosted build process on a server, virtual machine, or container. There are many tools from continuous integration/continuous development (CI/CD) that you can apply to automating the JAMstack. If you are already running CI/CD infrastructure, you might be able to plug JAMstack publishing into your existing workflows.

Using a Hosted Deployment Service

Because setting up a robust and resilient build system can be difficult, a new category of service has emerged: hosted build services/continuous integration. These services often have apps available for GitHub and GitLab, making the integration with your repository fairly painless. On some, you can select your own deployment target (like Amazon Web Services Simple Storage Service); on others, CDN and Domain Name System (DNS) services are baked in, providing everything you need to both build and host your application.

You want a build service that's secure and customizable. Most build services will commonly create a temporary container, set up your environment, grab any needed dependencies, run the build, deploy the results, and then dispose of the container. It's important that the build service is aware of and supports your site generator, language environment, and other tooling involved in building your website. You'll want to keep dependency settings files like Gemfile, *package.json*, and *composer.php* accurate and up to date.

Selecting a CDN

Even with all the modern advancements in network technologies, performance for websites and applications still succumbs to the two oldest foes of the internet: time and space. Thankfully, the JAMstack approach of publishing directly to CDNs helps solve for both. First, the time required to start serving content is greatly improved by prerendering all the markup, as we discussed earlier. And second, the amount of physical distance (and network hops) that content must travel on the way to users is also reduced, thanks to the geographically distributed nature of CDNs.

A CDN is a global collection of edge nodes located in different data centers, all interconnected with fast, high-quality bandwidth. When a user loads content from a CDN, they are always connected to the closest edge node possible. And, of course, distributing your site across multiple edge nodes also helps availability. If one or more edge nodes, network interconnects, or even datacenters fail, your site remains available.

Here are three considerations for choosing a CDN for your web application:

Edge Locations

Publishing to a CDN usually means your site content becomes available to a minimum of 10 or 15 locations worldwide. You'll want a CDN with edge locations close to your audience, so if you have users in Tokyo, they experience your website served from Tokyo.

Instant Updates

CDNs used to work as fairly unintelligent caching layers in front of web servers. Although they certainly sped up application performance, documents and images on the CDN were always given fairly lengthy Time To Live (TTL) settings. When cache was set to last hours or more, updates were delayed and purging the CDN to make way for new content proved rather difficult. It was often a delicate balance between allowing more dynamic content with frequent updates versus using longer caching to reduce the load on origin servers.

Modern CDNs now allow instant invalidation, replacing files in milliseconds and blurring the lines between static content and dynamic updates. If a near-instant update to CDN cache can be guaranteed each time the site is deployed, TTLs and stale content are no longer a concern. Today, rather than periodically pulling content from an origin server, updates can be pushed to CDNs each time the website changes. This advancement allows us to publish directly to CDNs and bypass servers—a critical component of the JAMstack.

Multiple Providers

Some CDNs locate their edge servers with multiple cloud providers, increasing the durability of your application. Even though a complete datacenter outage is rare, they do happen. It's also not uncommon for a provider to experience a Distributed Denial-of-Service (DDoS) attack toward a particular facility or network. Ensure that your CDN provider is not only distributed across the globe but also across providers. Hosting on such a provider gives you the power of a "cloud of clouds" for the most resilient setup possible.

Providing Functionality: APIs

Even though a JAMstack site can indeed be nothing more than a collection of static assets, connecting your application to the vast world of API-based services is what shows the true power and promise of the JAMstack as an architecture for real-world applications.

It wasn't too long ago that authentication, payment processing, and commerce were all concerns that you needed to host and manage yourself. But today, each of those (and many more features) are now consumable as APIs.

But how expensive will it be? What if the service goes down? How difficult would it be to rip out and replace? These questions are common when evaluating any hosted service, and it can feel daunting to trust critical services to a third party. But consider that all the questions above will also inevitably arise for teams that build or manage technologies in-house.

It all comes down to choice and flexibility, and the move toward decoupled, API-driven architectures (and away from monolithic apps) offers teams the flexibility to decide what to consume and what to build. That's a powerful construct because each layer of the stack—from content, to commerce, to the frontend—can now be vetted and chosen independently.

Let's look at that in action.

Consider the needs of an ecommerce store. First, the site needs to be fast and reliable, even under heavy load. The JAMstack's ethos of static pages published to a global CDN ensures incredibly high marks in both speed and reliability. Without managing a fleet of servers, you can keep pace with the performance offered by the giants of ecommerce.

But equally important, you get full control over the entire frontend, down to the very last `<div>` and pixel. You are free to design and develop any page you can imagine, using the language, tools, and templates of your choosing. Your team can build experiences custom-tailored to your exact specifications. This is a very different approach from fighting the markup output by a traditional server-hosted ecommerce application.

Commerce, though, isn't static. You'll need a way to manage content (in this case, items for sale), power search, manage available inventory, and allow customers to make purchases. On the JAMstack, we use APIs to integrate services tailor-made for each component.

In the example flow shown in Figure 4-3, notice how content is managed in a hosted CMS service called Contentful. Store owners and editors use the Contentful interface to manage store listings and upload photos. Contentful is termed a "headless" CMS because its only concern is storing the content, not the display of it.

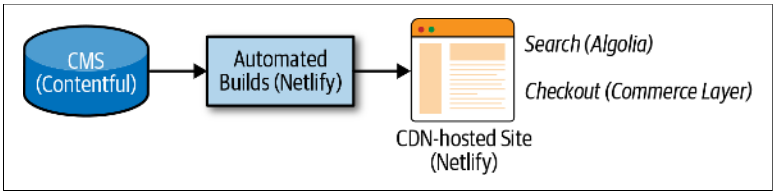


Figure 4-3. Sample JAMstack commerce setup

Adding a new item in Contentful triggers a new build of the website using Netlify's build automation. During the build step, Netlify fetches the most recent store data from Contentful by calling the API. That data is fed into a site generator (like Gatsby) to render out each of the pages.

Also during the build step, APIs are used to send data to Algolia, the search provider, so the search index can be updated as the product catalog changes.

As visitors browse the site, prerendered pages are pulled directly from a CDN, with no running server process anywhere to be found. So how is the site made interactive? By taking advantage of the power of JavaScript to call APIs directly from the browser.

When a shopper types a search term, it's fed to Algolia via Ajax, and Algolia returns JSON with the results. Javascript is used to display those results in whatever way we'd like to integrate them with our application.

After the shopper clicks the buy button, the intent to purchase is sent to a provider called CommerceLayer, whose job it is to power the shopping cart and checkout experience. Again, this happens using JavaScript to call APIs. Like Contentful, CommerceLayer is also headless and makes no demands on our how application looks and behaves.

There you have it: an entire commerce experience built by interconnecting modular components via API calls. Because there is no backend infrastructure to set up or manage, our entire time can be spent creating the frontend application. And because everything we're using is a hosted service, a first prototype can be created in about a day or two.

This is why you might have heard of the JAMstack providing "superpowers" for frontend teams. Today, we can wire together rich, enterprise-grade functionality using API services like the ones here. But again, nothing is prescriptive on the JAMstack and there has been an explosion of new API service offerings designed to power just about any aspect of your project. In fact, you'll find many traditional players like Shopify now offer APIs so they, too, can be consumed in a headless fashion. The explosion of APIs speaks to the momentum of the JAMstack as the architecture of the interconnected web.

CHAPTER 5

Shifting the Mental Model

A Mindset and an Approach

To adopt the JAMstack for your next project, your team will need to have a good mental model of what the JAMstack means. There's one commonly held mental model to watch out for, as it's not quite correct:

The JAMstack delivers static sites.

This is a convenient, bite-sized summary. But it misses the mark.

Sites designed for the JAMstack are sites that are *capable* of being served with static hosting infrastructure, but they can also be enriched with external tools and services.

This small shift in mindset is a powerful thing. A static site is a site that does not often change and does not offer rich interactions. But JAMstack sites go far beyond offering static experiences—whether that be through rapid iterations thanks to build automation and low-friction deployments, or through interactive functionality delivered by microservices and external APIs.

The ability to serve the frontend of a site from CDN infrastructure optimized for static assets should be viewed as a superpower rather than a limitation.

Although traditional, monolithic software platforms need to introduce layers of complexity for the sake of performance, security, and scale, JAMstack sites embody this by default. And because their technical design is based around the premise of being hosted statically, we are able to avoid many of the traps that traditional infrastructure can fall afoul of when being scaled and hardened.

The six best practices presented in the following sections can help in serving your sites from static infrastructure (without frontend web servers) and propel your projects to use all the possibilities of the JAMstack.

One: Get to the CDN

A Content Delivery Network (CDN) is a geographically distributed group of servers that allows you to cache your site's content at the network edge, effectively bringing it much closer to users. Because the markup is prebuilt, JAMstack sites don't rely on server-side code and lend themselves perfectly to being hosted on distributed CDNs without the need for origin infrastructure.

A good CDN will have points of presence around the globe and is therefore able to serve your content from locations geographically close to your users, wherever they might be. This has a number of benefits.

First, this promotes good performance. The shorter the distance between the user making a request for content and the server that responds to that request, the faster bytes can get from one to the other.

Second, by replicating the content being served across many CDN nodes around the world, redundancy is also introduced. Certainly, we'd want to serve content to users from the CDN node closest to them, but should that fail for any reason, content can instead be served from the next nearest location. The *network* aspect of a CDN provides this redundancy.

Third, by becoming the primary source for the content being served to your users, a CDN prevents your own serving infrastructure from becoming a single point of failure (SPoF) while responding to requests. Do you recall the fear of your site quickly becoming popular and attracting massive traffic levels? Oh, the horror! People would talk about the ability to "survive being slash-dotted" or reaching mass adoption.

Designing a web architecture to be resilient to large traffic loads and sudden spikes in traffic has traditionally called for the addition of lots of infrastructure. When your site isn't equipped to offload the serving of requests to a CDN, you must shoulder this responsibility yourself. Often this results in an architecture that involves multiple web servers, load balancers, local caching mechanics, capacity planning, and various layers of redundancy. When the site is being served dynamically and in response to individual requests, the coordination of these resources can become very complex. Specialist teams are generally required to do this properly at scale, and their job is never complete given the need for ongoing monitoring and support of the machinery that will service every request for a site.

Meanwhile, CDNs are in the business of servicing high volumes of traffic on our behalf. They are designed to offer redundancy, invisible and automatic scaling, load balancing, caching, and more. The emergence of specialist CDN products and services brings opportunities to outsource all of this complexity to companies whose core business is providing this type of resilience and scale in ways that do not require you to manage the underlying infrastructure.

The more of our site's assets we can get onto a CDN, the more resilient it will be.

Designing for the Best Use of a CDN

How do we design our applications to get the best results from using a CDN?

One approach is to serve just the containing user interface (UI) statically from the CDN and then enrich it entirely with client-side JavaScript calls to content APIs and other services. This works wonderfully for some applications or sites where the user is taking actions rather than primarily consuming content. However, we can improve performance and resilience if we deliver as much content as possible *in the document* as prerendered HTML. Doing this means that we do not depend on additional requests to other services before core content can be displayed, which is good for performance and fault tolerance.

The more complete the views we can deliver to the CDN and pass on to the browser, the more resilient and performant the result is likely to be. Even though we might be using JavaScript in the browser to enrich our views further, we should not forget the principles of progressive enhancement, and aim for as much as possible to be delivered prerendered in the initial view of each page.

When designing the technical approach to web applications, there are a few commonly used terms that seem to be regularly confused. We've alluded to a couple of them in the previous chapters, but it is worth taking a moment to clarify the differences between some of these common terms:

- Client-side rendering
- Server rendering
- Prerendering

Client-side rendering is performed in the browser with JavaScript. This term is not used to refer to the work that browsers do by default to render HTML documents; instead, it describes the process by which JavaScript is used to manipulate the Document Object Model (DOM) and dynamically manipulate what the browser displays. It has the advantage of allowing user actions to immediately influence what is displayed, and it also can be very efficient in combining data from many different sources into templated views. Web browsers, though, are not as tolerant to errors found in JavaScript as they are with HTML. Where possible, it is prudent to deliver as much content as possible already rendered into HTML on the server, which then can be further enriched and enhanced with JavaScript in the browser.

Server rendering refers to the process whereby a server responds to requests for content by compiling (rendering) the required content into HTML and delivering it on demand for the browser to display. The term is often used as the antithesis of client-side rendering, when data (rather than ready-generated HTML) is delivered into the browser where it must then be interpreted and rendered into HTML or directly manipulate the browser's DOM via JavaScript. (Client-side rendering such as this is common in single-page applications [SPAs].) Server-rendering typically happens *just-in-time* based on what requests are being made.

Prerendering performs the same task of compiling views of content as we see in server rendering, but this is carried out only once and in advance of the request for the content. Rather than awaiting the first request for a given piece of content before we know if the result will be correct, we can determine this at build time.

Decoupling this rendering or generation of views from the timing (and machinery) at request time allows us to expose the risk early and tackle any unknowns of this operation far in advance of the times a request for the content is made by a user. By the time requests for prerendered content arrives, we are in a position to simply return the *prebaked* response without the need to generate it each time.

Two: Design for Immutability and Atomic Deployments

By generating a site comprising assets that can be served statically from a CDN, we have set ourselves on an easier path to optimizing for performance and sidestepping many common security attack vectors. But it is very likely that our site builds will contain a large number of files and assets, all of which will need to be deployed in a timely and confident manner.

The next challenge, therefore, is to establish very low friction and efficient deployment processes to deal with the propagation of so many assets.

When we make the process of deploying a site fast with minimal risk, we unlock greater potential in the development and publishing process, which in turn can yield more efficient and effective development and release cycles.

Confidence from Immutability

It was once very common to deploy sites via FTP. It felt simple and convenient, but this method was the root of many problems.

The action of moving files and folders from one server to another over FTP would change the state of the production server, updating its published web root folder to include the new or updated files and mutating the state of the production environment more and more over time.

Mutating the condition of a production environment might not seem like a problem. After all, the reason we deploy new code is to make changes in our production environment and develop the experience for our users by introducing new content, new designs, or new functionality. But this type of deployment is *destructive*.

After each deployment, the previous state of the environment is lost, meaning that reverting to the last good state of the site could be difficult or impossible to do.

The ability to revert is critical. Without it, we need to be able to guarantee that every deploy will be perfect, every time. That's a lovely goal, but experience tells us that we can't assure that with 100% confidence, and that the ability to quickly address problems resulting from a deployment is essential.

If we combine mutable deployments and managing our own complex hosting infrastructure, the potential issues are compounded further, with deployments potentially resulting in sites that have unknown versions being served to different users, or errant code being propagated to some parts of the infrastructure and not others, with no clear path to identify what has changed and how to address problems.

To minimize the risks associated with hosting environments serving assets jumbled together from a variety of deployment events, the industry began to discard *mutable* deployments, in which the site's web root could change over time, and instead favor *immutable* deployments where a new version of the entire published web root would be compiled and published as a discrete entity, as illustrated in Figure 5-1.

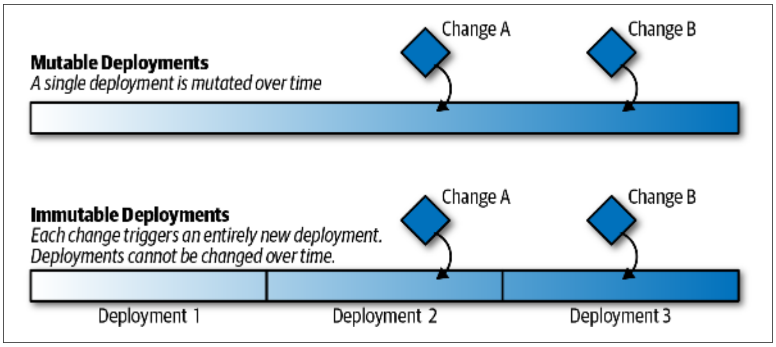

Mutable Deployments
A single deployment is mutated over time
Change A
Change B
Immutable Deployments
Each change triggers an entirely new deployment.
Deployments cannot be changed over time.
Change A
Change B
Deployment 1
Deployment 2
Deployment 3

Figure 5-1. Mutable versus immutable deploys

The Advantages of Immutable Deployments

Whereas mutable deployments modify the code and state of our environments, changing them gradually over time, immutable deployments are different. They are delivered as entirely self-contained sets of assets, with no dependency on what came before, and associated with a moment in time in our version control system. As a result, we gain confidence and agility.

Know what you're deploying

Confidence comes from knowing that what we are deploying is what will be served by our production environment. Without immutable deployments, each deployment changes the state of what lives in the production environment and is being served to the world. The live site ends up being the result of a compounded series of changes over time. The actions taken in each of our environments all contribute to that compounded series of changes, so unless every action in every environment is identical, uncertainty will creep in. (We mention production, but this applies to every environment, such as development, staging, testing, and so on).

The actions we take over time are aggregated into what is ultimately published from each environment.

In contrast, an immutable deployment process removes this uncertainty by ensuring that we create a series of known states of our website. Encapsulated and self-contained, each deployment is its own entity and not affected by any actions taken in the environment previously. Each deployed version persists unadulterated, even after subsequent deployments. Compound effects in environments are avoided so that we can gain the certainty and confidence lacking in mutable deploys.

Switch between versions anytime

It might seem cumbersome to generate and deploy an entirely new build each time we want to make even a small change to our sites, but because this leaves previous deployments intact, we gain the ability to switch between different versions of our site at any time.

And by automating and scripting the build and deployment processes, we can drop the friction associated with building and deploying sites to close to zero, making this process far less cumbersome. Regularly running builds even helps to prove the validity of our build process itself, ensuring that our build and deployment process is robust and battle hardened far in advance of launch day.

The agility to roll back to a previous build or to stage a new build in our environment for testing before designating it the new production version, cultivates an environment in which the risks involved in performing deployments are reduced. By avoiding unrecoverable failures that are the result of a deployment, agility and innovation are given room to thrive on projects. It creates a very constructive development experience rooted in the confidence that our target environments are not fragile, and that they can withstand the demands of active development work. Certainly, we would wish to minimize the risk of any errors being published, but it can happen. With atomic deployments, we dramatically decrease the cost of an error by making it trivial to revert to a previous good state of our site.

Positioning for Immutable Deployments

By choosing the JAMstack for a project, you're already well suited to a deployment process that yields immutable deploys. The build process for the project will generate a version of the site and its assets that is ready to be placed into a hosting environment.

Decoupling that which can be hosted statically from the moving parts of a project that might manage state or contain dynamic content is a critical step. There is a natural decoupling inherent in JAMstack sites, for which data sources and APIs are either consumed at build time, or abstracted to other systems, possibly provided by third parties.

The ability to achieve an immutable deployment process will also depend on the nature of the hosting environment. No matter how we transmit our assets to that environment, it must be configured to generate a new site instance each time changes are deployed to it.

Atomic Deployments

A challenge with creating multiple, immutable versions of our deployments is that the action of deploying a new version of our site, even a one-line change, requires the generation and propagation of a completely fresh instance of the entire site. This could result in the transmission of a lot of files, taking a lot of time and bandwidth. In traditional mutable deployments, this could result in files from many different deployments being served together, with no protection against interdependencies not being synchronous.

To address this, a good practice is to employ *atomic deployment* techniques.

An atomic deployment is an all-or-nothing approach to releasing new deployments into the wild. Figure 5-2 shows that as each file is uploaded to its hosting infrastructure, it is not made available publicly until every single file that is a member of that deployment is uploaded and a new, immutable build instance is ready to be served. An atomic deployment will never result in a partially completed build.

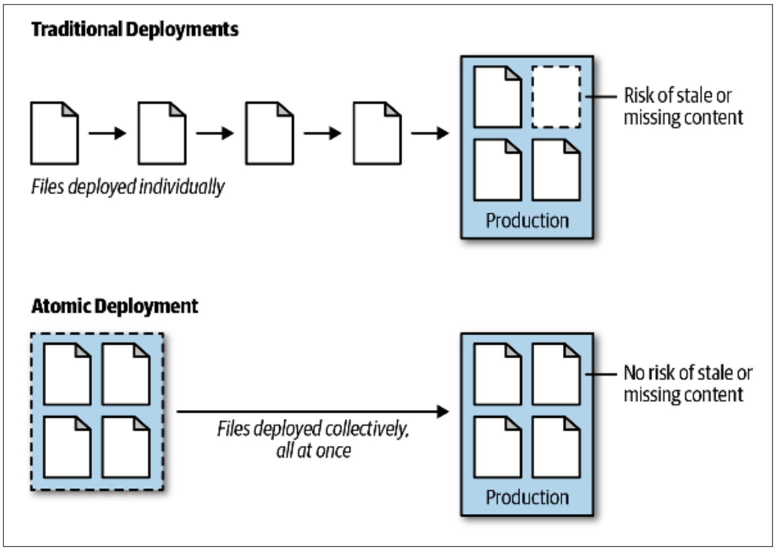

Traditional Deployments
Files deployed individually
Risk of stale or missing content
Production
Atomic Deployment
Files deployed collectively, all at once
No risk of stale or missing content
Production

Figure 5-2. Traditional versus atomic deploys

Most good CDN providers offer this service. Many of them use techniques to minimize the number of individual files being transmitted between their nodes and the origin servers which seed all of the caching. This can become tremendously complex for sites that are not designed as JAMstack sites, and considerable specialist skills are required to do this correctly when the files in question are populated dynamically.

JAMstack sites are well suited to such techniques because they will avoid the considerable complexity of needing to manage the state of databases and other complex moving parts throughout deployments. By default, the files that will be distributed to CDNs are static files rather than dynamic, so common version control tooling can allow CDNs to consume and distribute them with confidence.

Three: Employ End-to-End Version Control

The success of any development project can be compromised when robust version control management, such as Git, is not in place. It is common in web development for projects to employ version control throughout the development process, but then not extend the same version control conventions into the deployment process itself. Heroku, a popular cloud application platform, popularized the use of Git workflows as a means to perform a deployment, raising the confidence in a reliably deployed end product.

Having the version control reach all the way to deployment can help to minimize the opportunities for untracked, unrepeatable, or unknown actions to creep into the processes and environments.

A dependency on databases and complex applications can make the task of this kind of end-to-end version control more difficult. But because JAMstack sites don't rely on databases, it's far easier to bring more aspects of the code, content, and configuration under the umbrella of version control. Further, the use of the version control system itself can become the primary mechanic for doing everything from initiating and bootstrapping a project, right through to staging and deploying the code and the content.

Every aspect of a project that we can bring into version control is one less thing that is likely to get out of control.

Reducing the Cost of Onboarding

Bringing new developers into a project can be a time-consuming and error-prone activity. Not only do developers need to understand the architectural conventions of the project, they must also learn what dependencies exist, and acquire, install, and configure all of these before any work can begin.

Version control tools can help. By tracking all of the project's dependencies in version control and ensuring that any configuration of these dependencies is also captured in code and version controlled, we can apply some popular conventions to simplify the bootstrapping of projects and the onboarding process.

Ideally, a developer arriving onto a new project would be able to get up and running with minimal intervention. A good experience for a developer might look like this:

1. Gain access to the project code repository.
2. Find up-to-date, versioned instructions on bootstrapping the project within the code repository, typically in a *README* file.
3. Clone the repository to a local development environment.
4. Run a single command to install project dependencies.
5. Run a single command to run a local build with optimizations for the designed development workflow.
6. Run a single command to deploy local development to suitable target environments.

Maintaining onboarding documentation in a *README* file within a project's code repository (rather than in another location) helps with its discoverability and maintenance. That's a good first step, but the other items on this list might seem more challenging.

Thankfully, the rise of package managers such as npm and Yarn have popularized the techniques that can help here. The presence of a *package.json* file in a project's root directory (which defines the various dependencies, scripts, and utilities for a project) combined with suitably descriptive documentation as part of the version-controlled code base can allow a developer to run single commands to perform the dependency installation, local build, and even deployments.

This approach not only allows for the automation of these activities, but also documents them in a way that is itself versioned. This is a huge benefit for the health of a project over its lifetime.

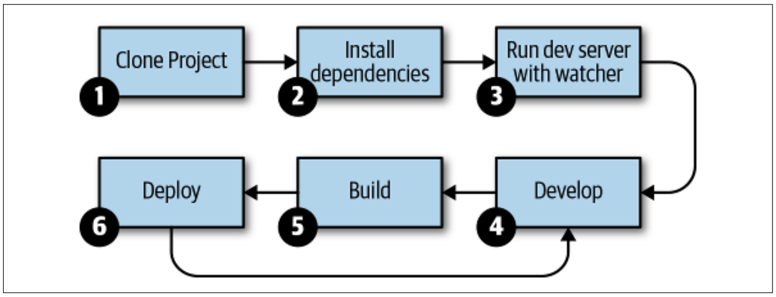


Figure 5-3. Common development workflow

Making Version Control the Backbone of Your Project

Unseasoned approaches to version control can lead to a code repository being used mainly as a central store for the purpose of backing up code. This attitude was once very common and addressed only one narrow aspect of software development. These days, it is more common for development teams to harness the power of version control and use it as the underlying process for orchestrating and coordinating the development, release, and even the deployment of web projects.

Actions and conventions at the core of a tool such as Git—like committing, branching, making pull requests, and merging—already carry significant meaning and are typically being used extensively during the natural development life cycle. And yet, it is common for monolithic applications to invent their own solutions and workflows to solve similar challenges to those already being addressed by Git.

Rather than doing that, and adding another layer of process control that requires teams to learn new (and analogous) conventions, it is far more productive to use the Git activity to trigger events in our deployment systems.

Many continuous integration (CI)/continuous deployment (CD) products can now work this way. Activities like pushing code to a project's master branch can trigger a deployment to the production environment.

Instead of limiting Git to a code repository, we should employ it as the mechanism that drives our releases and powers our CI/CD tools. This anchors the actions taken in our development and deployment processes deeply in version control, and consolidates our actions around a well-defined set of software development principles.

When choosing a platform for CI/CD and deployment management, seek out such platforms that support Git-based workflows to get the best from these conventions.

Four: Automation and Tooling

A key to making JAMstack sites go beyond static is dramatically easing their building and deployment processes through automation. By making the task of deploying a new version of a site trivial, or even mundane, we significantly increase the likelihood that it might be regularly updated with content, code fixes, and design iterations throughout its lifetime.

This principle is true for web development projects on all stacks, but in more traditional, dynamic stacks, the lack of good deployment automation is more often overlooked because content updates can still typically be made without a full code redeployment. This can provide a false sense of security on dynamic sites, which become a problem only when an issue is discovered or a more fundamental change is planned for deployment.

By contrast the JAMstack is well suited to scripting and automation during both the build (or site generation) stage and the deployment stage.

Static site generators exist as tools that we run to compile our templates and content into a deployable site. By their very nature, they are already the seed of an automation workflow. If we extend this flow to include some other common tasks, we begin to automate our site generation process at the very beginning of our development process and iterate it over the life of a project.

With a scripted site generation and deployment process in place, we can then explore ways to trigger these tasks automatically based on external events, adding a further level of dynamism to sites that still have no server-side logic at runtime.

A Powerful Site-Generation Pattern

The tools used for automating a site build can be a matter of personal preference. Popular options include Gulp, Grunt, or Webpack, but simpler tools like Bash or Make can be just as effective.

Let's look at an example of some tasks that might be automated in the build script of a JAMstack site to produce a more dynamic site:

1. Gather external data and content from a variety of APIs.
2. Normalize the structure external content and stash it in a format suitable for consumption by a static site generator.
3. Run the static site generator to output a deployable site.
4. Perform optimization tasks to improve the performance of the generated assets.
5. Run the test suite.

6. Deploy output assets to the hosting infrastructure or CDN.

Figure 5-4 shows this process.

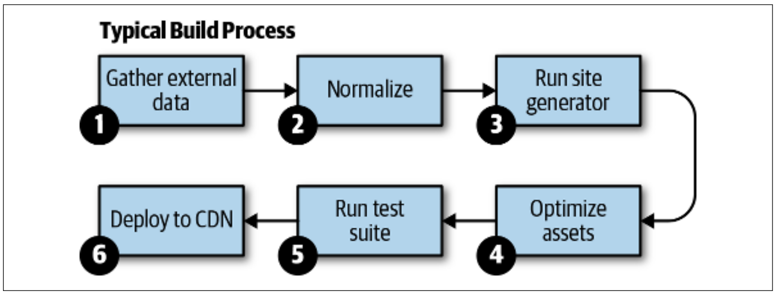


Figure 5-4. Typical build process

This pattern is powerful, especially if it can be triggered by external events. The first step in the pattern we describe here is to gather external content from APIs. The content sources that we are consuming here might be numerous. Perhaps we'll be using content from a headless CMS or the RSS feed of a news site. Perhaps we'll be including recent content from a Twitter feed or other social media feeds. We might include content from an external comments system or a digital asset management system. Or all of these!

If we also *initiate* this automated site build whenever a change occurs to any of the content sources that we will be consuming, the result will be a site that is kept as fresh and as up-to-date as any of its content sources.

An increasing number of tools and services now support publishing and receiving *webhooks*. Webhooks are URL-based methods of triggering an action across the web. They can create powerful integrations of services that otherwise have no knowledge of one another. With a webhook you can do the following:

- Trigger a build in a CI system
- Trigger an interactive message in a messaging platform such as Slack
- Notify a service that content has been updated in a headless CMS such as Contentful

- Execute a function in a Functions as a Service (FaaS) provider such as Amazon Web Services (AWS) Lambda, Google Cloud Functions, or Microsoft Azure Cloud Functions.

By combining these types of tools and building upon an event-driven model to perform automated tasks, things are starting to become a lot less “static” than we might once have expected.

Minimizing the Risk of Human Intervention

Automation is not purely about doing repeatable tasks quickly. It turns out that humans are pretty bad at doing the same task over and over again without making mistakes. Removing ourselves from the build and deployment process is a good way to reduce errors.

Seek out opportunities in your deployment processes where repetitive tasks can be performed by a script.

It’s very common to plan for a time late in a project to do this kind of automation, but we recommend doing it early. This will do the following:

- Create basic documentation of your processes by codifying them
- Introduce version control into your deployment process itself
- Reduce the risk of human error for every time you deploy
- Begin to instill a project culture in which deployments are not heralded as rare, complex, and risky operations

Five: Embrace the Ecosystem

Not only are there tools and services for continuous integration, global deployments, and version control, there are also all sorts of more exotic and ambitious tools and services. The immediacy and affordability of such services is impressive, and they bring incredible breadth and variety in the ways that they can be combined and remixed to create innovative new solutions to once-complex and expensive web development challenges.

One of the key reasons that we need a more descriptive term than “static” to describe JAMstack projects is this rapid growth in the ecosystem rising to support and extend the power of this approach.

Once, when we talked about "static" sites, we would most likely rule them out as candidates for specific use cases on the basis of some common stumbling blocks. Requirements like search, forms, notifications, or payments would rule out a static architecture for many sites. Thankfully, as the ecosystem has expanded, solutions to all of these requirements have emerged, allowing developers and technical architects to go far beyond what they might once have considered possible on the JAMstack.

As a way of exploring the concepts a little, let's talk about some examples of common requirements that used to deter people from pursuing a JAMstack model.

Forms

This is the classic and possibly the most common requirement likely to cause a technical architect to decide that static hosting infrastructure would not be sufficient. It's also perhaps the cause of the most dramatic overengineering.

That may be a surprising statement. After all, form handling is relatively simple and a very common feature on the web. It's not really something that we associate with overengineering.

When the introduction of this feature demands that a site that might otherwise be delivered directly from a CDN (with no need for logic running at a server) shifts to being hosted on a server complete with all the overhead that incurs, simply to be able to handle HTTP post requests and persist some data, the introduction of an entire server stack represents a huge increase in platform complexity.

A multitude of form-handling services have emerged. Ranging from JavaScript drop-ins to more traditional synchronous HTTP requests with API access, the community has noticed how common this requirement is and found a multitude of ways to deliver without the need for a hosted server. Consider products like Wufoo, Formkeep, Formcarry, or even Google Forms.

Search

A search facility on a site can vary dramatically in complexity. Often search can be simple, but in some situations, search features need to be advanced, including a wide variety of ways to determine results and inspect complex relationships within the content of a site.

There are approaches to handle both ends of this spectrum.

Most static site generators have the capability of producing different representations of the content of a site. We can use this to our advantage by generating a search index of the site at compilation time, which then can be queried using JavaScript directly in the browser. The result can be a lightning-fast live search that responds as you type your search queries.

For more sophisticated search facilities, services like Algolia can provide extensive content modeling and be customized to include data insights (with knowledge of your site's taxonomy), provide fuzzy matching, and more. From small sites to large enterprises, Algolia has identified the value in providing powerful search and bringing it to all types of technology stacks, including the JAMstack.

There are also ways of scoping the search of the established search engines such as Google or Duck Duck Go to provide search within a given site or domain. This can provide a functional and efficient fallback for when JavaScript is not available, and is the most direct and simple implementation to provide search capabilities to a site.

Notifications

The need to contact users often introduces complexity to the stack. As with search and forms services, a number of companies have identified this and have established products and services to provide this facility without the need to own the finer details and risks.

Companies like SendGrid and Mailgun can add email messaging and are already trusted by a long list of technology companies that would rather not manage and integrate their own email messaging systems into their sites and services. Going further, companies like Twilio can add sophisticated messaging that goes far beyond just email. SMS, voice, video, and interactive voice response (IVR) are all possible through APIs.

Identity

The ability to authenticate a user's identity and grant different permissions through some form of authentication system underpins a wide variety of features usually associated with a traditional technology stack.

Managing user data, particularly personally identifiable information (PII) requires a great deal of care. There are strict rules on how such information should be retained and secured, with significant consequences for organizations that fail to satisfy suitable compliance and regulatory levels. Outsourcing this capability was not always desirable because this could be seen as giving away precious customer information.

Fortunately, practices have now evolved to allow for the successful decoupling of identity services such that they can be effectively provided by third parties without exposing sensitive customer data. Offloading some of the rigorous compliance and privacy requirements to specialists can bring identity-based features into the reach of projects and companies that otherwise might not be able to deliver and maintain them.

Technologies and identity providers such as OAuth and Auth0 can make this possible and allow for safe client-side authentication and integration with APIs and services. We dig into some examples of this in the next section.

Six: Bridging the Last Gaps with Functions as a Service

As you can see, we have an expanding set of tools and services available to us. We don't need to reinvent or reimplement each of them in order to add their capabilities to our sites. Search, user-generated content, notifications, and much more are available to us without the need to own the code for each. But not every one of these services can be integrated into our sites without some logic living somewhere.

Functions as a Service (FaaS; often called *serverless functions*) can be the ideal solution to this and can be the glue layer that binds many other services together.

By providing us with the means to host functions and invoke them when needed, serverless providers extend the JAMstack even further while liberating us from the need to specify or manage any type of hosting environment.

Being Comfortable with Breaking the Monolith

It is often remarked that being a web developer means that you need to always be learning new things. This pace is not slowing down. Thankfully, however, we don't all need to be experts in all things.

One of the joys of breaking apart the monoliths of old is that we can hand over the deep domain expertise of aspects of our development projects to those who specialize in providing just those targeted services. At first, it can feel uncomfortable to hand over the responsibility, but as the ecosystem of tools and services matures, that discomfort can be replaced by a refreshing feeling of liberation.

By embracing the specialist services available, we can take advantage of incredibly sophisticated and specialized skills that would be impossible to retain on each and every one of our projects were we to attempt to bring them all in-house.

Naysayers might describe concerns about relinquishing control and accountability. This is a reasonable thing to consider, and we'd be well advised to retain control over the services that we can provide more efficiently, securely, comprehensively, and cost-effectively than external providers. However, the areas where this is truly the case are not as numerous as we might imagine, especially when factoring in things like ongoing support, maintenance, data privacy, and legal compliance. Although it might seem counterintuitive at first, by employing the specialized skills of, say, an identity provider, we can avoid the need to intimately understand, build, and maintain some of the more nuanced areas of providing identity, authentication, and authorization.

Instead, by using external solutions providers to deliver this capability to our projects, we can refocus our efforts away from reimplementing a common (yet, complex and numerous) set of features and functionalities, and focus instead on other areas of our projects where we might be able to truly differentiate and add measurable value.

The core intellectual property of your website is unlikely to reside in the code you write to implement how a user changes their password, uploads their profile picture, or signs in to your site. It is far more likely to reside in your content or your own set of services that are unique to you and your business.

As the ecosystem of tools as service available to JAMstack sites grows larger, we see the approach expand into areas traditionally dominated by monolithic, server-intensive architectures.

Selecting which features of a given project are good candidates for being satisfied by third-party services does require some thought. We've mentioned just a few here and there are many more. Considering the total cost of ownership for given services required by your site is important. Do the providers you are considering have suitable terms of service? Can they provide a suitable Service-Level Agreement? How does the cost of that compare to that if you were to build and maintain this for yourself?

More and more regularly, economies of scale and domain expertise mean that providers can be more affordable and dependable than if we were to build comparable solutions ourselves. It is usually easier to imagine the possibilities from embracing and utilizing the expanding ecosystem of tools and services in the context of a real-world example. By exploring how you might design an approach to delivering a real project, with a well-defined set of functional and nonfunctional requirements, you can explore how the JAMstack can be applied to bring significant improvements to an established presence on the web.

In Chapter 6, we do just that. We examine how various third-party services were utilized and combined using serverless functions and various automations. We don't pick a small or simplistic site, either. Instead, we explore a genuine project on a rich, complex, and for many web developers, beloved website.

CHAPTER 6

Applying the JAMstack at Scale

A Case Study: Smashing Magazine

Smashing Magazine has been a steady presence in the frontend development space for more than a decade, publishing high-quality content on themes like frontend development, design, user experience (UX), and web performance.

At the end of 2017, *Smashing Magazine* completely rewrote and redesigned its site to switch from a system based on several traditional, monolithic applications to the JAMstack. This chapter showcases how it solved challenges like ecommerce, content management, commenting, subscriptions, and working with a JAMstack project at scale.

The Challenge

In its 10 years of operation, *Smashingmagazine.com* has evolved from a basic Wordpress blog to a large online estate, spanning multiple development platforms and technologies.

Before the relaunch, the main magazine was run as a Wordpress blog with thousands of articles, around 200,000 comments, and hundreds of authors and categories. Additionally, the Smashing team ran an ecommerce site selling printed books, ebooks, workshops, and events through a Shopify store with a separate implementation of its HTML/CSS/JavaScript theme. The magazine also operated a Ruby on Rails app as its job board (reimplementing the

Wordpress theme to work in Rails, with Embedded Ruby [ERB] templates) and a separate site built with a flat-file Content Management System (CMS) called Kirby (which brought an additional PHP stack, not necessarily in step with the PHP stack required by the main Wordpress site) for handling the online presence of Smashing Conference.

As *Smashing*'s business grew, its platforms could not keep pace with its performance requirements; adding new features became unsustainable. As the team considered a full redesign of *Smashing Magazine* together with the introduction of a membership component that would give members access to gated content and discounts on conferences, books, and workshops, there was a clear opportunity to introduce a new architecture.

Key Considerations

From the project's beginnings, *Smashing Magazine* sought to enhance the site with a better approach to templating; better performance, reliability, and security; and new membership functionality.

Duplicated Templates

All themes had to be duplicated among Wordpress, Rails, and Shopify, each of which adds its own individual requirements to the implementation; some could be built out as more modern and front-end friendly, with a gulp-based, task-running workflow. Some had to be in standard Cascading Style Sheets (CSS) with no proper pipeline around it. Some needed to work within a Rails asset pipeline where making changes to HTML components involved changes in PHP, Liquid template files, and ERB template files with different layout/include structures and lots of restrictions on what HTML the different platforms could generate. To be maintainable, and to promote consistent results across the site, a single, common approach to templates and their build pipeline would need to be established.

Performance, Reliability, and Security

Smashing Magazine went through constant iterations of its caching setup to avoid outages when an article went viral. Even after considerable efforts, the site still struggled to reach acceptable uptime and performance goals. This effort introduced a great many plug-ins, libraries, and dependencies for Wordpress, Rails, and Kirby, resulting in a constant uphill battle to keep numerous Wordpress plugins, Ruby Gems, and other libraries up to date without causing breakage to themes or functionalities. To achieve a suitable level of performance, in a responsible, robust, and manageable way, this architecture had to be simplified and consolidated with *Smashing Magazine* taking ownership of fewer languages and technologies.

Membership

One of the important goals of the redesign was the introduction of the Smashing membership, which would give subscribers benefits across the platform, such as an ad-free experience, discounts on books and conferences, access to webinars and gated sections of the site, and a member indicator when posting comments. However, each of the applications making up *Smashing Magazine* were monolithic systems that applied a different definition of user. For membership to be supported and for the different parts of the *Smashing Magazine* ecosystem to be sustainable, a unified definition of a user would need to be established.

Picking the Right Tools

Taking into account the key considerations, *Smashing Magazine* began exploring an architecture in which the frontend lives in just one repository and one format, all content pages are prebuilt and live on a CDN, individual microservices handle concerns like processing orders or adding comments and identity of users, and members can be delegated to one microservice. The JAMstack.

The first step in the process was to build the right core tools, particularly the core *static site generator*, a *frontend framework* for the dynamic components, and an *asset pipeline* to make it all work well together. After these were established effectively, this set of core tools would enable the use of any number of microservices.

Static Site Generator

To build out all articles, categories, content pages, author pages, event pages with speaker and talk details, as well as RSS feeds and various legacy pages, *Smashing Magazine* needed a static site generator with a very flexible content model. Build performance was also a key constraint because each production build would need to output tens of thousands of pages in total.

Smashing Magazine evaluated several static site generators. It explored tools like Gatsby and React Static that included an asset pipeline and are based on Webpack. Those provide straightforward support for writing pure, client-side components in the same language as the templates for the content-based site. However, the performance when it came to pushing out tens of thousands of HTML pages in every build was not quite there.

Ultimately, the need to achieve a rapid build time combined with the complex content model made Hugo the first choice. It was an ideal combination of performance and maturity.

Hugo can generate thousands of pages in seconds; has a section-based content model; a taxonomy system that allows many forms of categories, tags, related posts, and so on; and a solid pagination engine for all listings. Many of the JavaScript-based generators are working to provide better support for incremental builds and large-site performance, but at the time of this writing, none of them could compete with Hugo in terms of its build speed, and they all fell short of the performance that *Smashing* felt comfortable with for a site of this volume.

There are potential workarounds to the challenge of needing to generate a very large number of pages in the build, like only prebuilding some of the content and fetching the rest client side from a content API. But because *Smashing Magazine* is in the business of publishing content, serving that content in the most robust and discoverable way would be essential. To avoid disrupting the search rankings of older reference articles, it was decided that content should be prebuilt and ready on a CDN without the need for any client-side JavaScript for rendering core content or basic browsing. Hugo was the only tool that was fully geared to meet the challenge with this amount of content.

Asset Pipeline

A modern asset pipeline is what allows us to use transpilers, post-processors, and task runners to automate most of the work around actually building and delivering web assets to the browser in the best possible way.

While many static site generators come with their own asset pipelines, Hugo does not. The *Smashing* team would need to create its own if it wanted support for bundling, code splitting, using ES6 with npm, and SCSS for CSS processing.

Webpack has become more and more of a standard for handling the bundling, code splitting, and transpilation of JavaScript, and it felt like an obvious choice to handle all of the JavaScript processing on this project. However, when it comes to preprocessing SCSS to CSS, Webpack is better suited for use with projects designed as single-page applications (SPAs) than for content-based sites like *Smashing Magazine*. Webpack's tendency to center around processing JavaScript and "extracting" the CSS referenced in the JavaScript modules makes it less intuitive to use on a project such as this and better suited for use with tools like Gatsby where assets are always referenced from JSX files.

The efficiency of the CSS workflow can have a huge impact on developer experience and the success of a project like this. At the beginning of the project, Sara Soueidan, a talented *Smashing Magazine* developer with extensive CSS skills, had built a CSS/HTML pattern library that would form the basis of the site. It was important to ensure that she could work with the same setup as the final site so that she could simply start building out pages using the same partials and as the pattern library.

To make this work, Smashing connected with a team versed in creating open source projects to support the viability and growth of the JAMstack. That team built out a small framework called "Victor Hugo," "A boilerplate for creating truly epic websites!"

Victor Hugo is based on a core of Gulp as a task manager, orchestrating CSS compilation via Sass, and Hugo builds whenever a template changes, and processing JavaScript via Webpack whenever a source file is updated. It also offers us a general task runner for maintenance tasks and similar automated processes along the way.

Frontend Framework

The next step was to introduce a frontend framework. Your first thought at this point might be "why?" Wouldn't just using progressive enhancement with a bit of vanilla JavaScript be good enough for a content-driven site like *Smashing Magazine*?

For the core experience of the magazine—the shop catalog, the job board, event pages, and so on—this was indeed the approach. The only JavaScript involved in this part of the project was built with a progressive enhancement approach and used for things like loading new articles inline or handling details like triggering pull quote animations on scroll and so on.

But *Smashing Magazine* is far more than just a magazine, with features like ecommerce, live search, commenting, and sign-up flows all requiring dynamic user interface (UI) elements. Without a rigorous, well-organized, and logical architecture and development approach, such complexity could quickly result in a nightmare of unmaintainable spaghetti. An effective, modern frontend framework needed to be in place in order to allow the development team to execute as planned. Being unimpeded by the compromises and overheads typically encountered when integrating with traditional monolithic platforms would be a huge boon for the development process, resulting in a far more logical and maintainable codebase.

At the same time, *Smashing Magazine* is not an SPA that can better afford to load a massive JavaScript bundle on first load to rely on in-browser page transitions from then on. A content-driven site needs to optimize for very fast performance on first view for all content-based parts of the site, and that means the framework must be tiny.

Preact fits that bill! It's a 3 KB component framework (gzipped) that's API-compatible with React. In comparison, jQuery weighs in at around 28 KB (gzipped).

This brought a modern component-based framework with the familiar interface of React to the table. Using Preact, the team could build interactive elements like the flows for ecommerce checkout or sign-up/login/recover password processes entirely client side—and then use Ajax to call specific microservices when an order is triggered or a login form is submitted.

We dig into those details shortly, but first it is worth turning our attention to the content that *Smashing Magazine* would be serving through this web experience.

Content Migration

In the previous version of *Smashing Magazine*, the content lived in several different binary databases. Wordpress had its own database in which most content was stored in a `wp_posts` collection with references to metadata and author tables. Shopify had an internal database storing books, ebooks, its categories, and a separate set of authors for those. The job board stored data inside yet a third database with a custom schema.

The numerous and distinct databases were a cause for concern and the root of many obstacles for *Smashing Magazine*.

The first job was to get all this data out of the different databases and into plain-text files in a simple, folder-based structure.

The goal was to get from a bunch of different databases to a folder and file-based structure that Hugo would understand and work with. This looks something like the following:

```
/content
 /articles
   /2016-01-01-some-old-article-about-css.md
   /2016-01-02-another-old-article-about-animation.md
   ...
 /printed-books
   /design-systems.md
   /digital-adaption.md
   ...
 /ebooks
   /content-strategy.md
   /designing-better-ux.md
   ...
 /jobs
   /2018-04-01-front-end-developer.md
   /2018-05-02-architecture-astronaout.md
   ...
 ...
/data
 /authors
   /matt-biilmann.yml
   /phil-hawksworth.yml
   ...
 /categories
   /css.yml
   /ux.yml
```

For existing projects with lots of data, this is always a challenging part of the process, and there's no one-size-fits-all solution.

There are various tools that allow you to export from Wordpress to Hugo, Jekyll, and other static site generators, but Wordpress plugins often add their own metadata and database fields, so depending on the setup, these might not be bulletproof.

Smashing Magazine used a small tool called make-wp-epic that can serve as a starting point for migrations from Wordpress to the kind of content structure Hugo prefers. It does a series of queries on the database and then pipes the results through a series of transforms that end up being written to files. It gave *Smashing* the flexibility of writing custom transforms to reformat the raw HTML from the Wordpress post-bodies into mostly Markdown with some HTML blocks. The project required some custom regex-based transforms for some of the specific shortcodes that were used on the old Wordpress site. Because Hugo also has good shortcode support, it was just a matter of reformatting those.

A similar tool was created using the RSS feeds from the *Smashing* job board, combined with some scraping of the actual site to export that content. Something similar was initially tried for the Shopify content, but the data structures for the shop content ended up changing so much that the *Smashing* team preferred simply reentering the data.

Utilizing Structured Content

An article would have a structure like this:

```
---
title: Why Static Site Generators Are The Next Big Thing
slug: modern-static-website-generators-next-big-thing
image: 'https://cloud.netlifyusercontent.com/assets/344dbf88-
fdf9-42bb-adb4-
46f01eedd629/15bb5653-5a31-4ac0-85fc-c946795f10bd/jekyll-opt.
png'
date: 2015-11-02T23:03:34.000Z
author: mathiasbiilmannchristensen
description: >-
 Influential design-focused companies such as Nest and
 MailChimp now use static website generators for their
 primary websites. Vox Media has built a whole
 publishing system around Middleman...
categories:
 - Coding
 - Tools
 - Static Generators
---

<p> At <a href="https://www.staticgen.com">StaticGen</a>, our
open-source directory of <strong>static website generators
</strong>, we've kept track of more than a hundred generators
for more than a year now, and we've seen both the volume and
```

```
    popularity of these projects take off incredibly on GitHub
    during that time, going from just 50 to more than 100
    generators and a total of more than 100,000 stars for static
    website generator repositories.</p>
    ...
```

This plain-text format with a YAML-based frontmatter before the post body became standard when Tom Preston-Werner launched Jekyll. It's simple to write tooling around, and simple to read and edit by hand with any text editor. It plays well with version control and is fairly intuitive at a glance (beyond a few idiosyncrasies of YAML).

Hugo usesthe `slug` field to decide the permalink on the file, together with the date, so it's important that these combine to create the same URL as the Wordpress version.

Some people intuitively think that something like the content from *Smashing Magazine* with thousands of articles and 10 years of history would be unwieldy to manage in a Git repository. But in terms of the pure number of plain-text files, something like the Linux kernel will make just about any publication seem tiny in comparison.

However, *Smashing Magazine* also had about a terabyte worth of images and assets. It was important to keep these out of the Git repository; otherwise, anyone working with the site would have to download all of them, and each Git clone operation during the continuous deployment cycles would have been painfully slow.

Instead, all the assets were uploaded to a dedicated asset store, a map of the original asset path to the new Content Delivery Network (CDN) asset URL was stored, and during the content migration, all image and asset URLs in the post metadata or the post bodies were rewritten to the new CDN URL. You'll notice this for the `image` meta field in the previous example post.

This is generally the best approach for anything but lightweight sites: store content and metadata in Git, but offload assets that are not part of your "theme" to a dedicated asset store with a CDN integration.

Both authors and categories were set up as taxonomies to make sure Hugo could build out listing pages for both.

Working with Large Sites

One problem that emerged after the content had been imported into the Victor Hugo boilerplate was that building out all the thousands of production articles made editing templates with automatic rebuilds during local development feel sluggish.

To work around that, Gulp was used as a way to add a few handy utilities. First, the full set of articles was added into a *production-articles* folder outside of the main content folder. Then, a Gulp task that could take an extract of 100 articles and move it into the actual *content/articles* folder was added.

This way, everybody could work against a smaller subset of articles when doing local development and enjoy the instant-live reloading environment that JAMstack setups enable.

For production builds, these two Gulp tasks were used to prepare a *prod/* folder with all the production articles before running the actual build:

```
gulp.task('copy-site', (cb) => {
 return gulp.src(['site/**/*', '!site/content/articles/*',
 '!site/production-
articles/*'], { dot: true }).pipe(gulp.dest('prod',
{overwrite: true}));
});
gulp.task('copy-articles', ['copy-site'], (cb) => {
 return gulp.src(['site/production-
articles/*']).pipe(gulp.dest('prod/content/articles',
{overwrite: true}));
});
```

This meant that working on the site locally didn't require Hugo to take thousands of articles into account for each build, and makes the development experience much smoother.

Building Out the Core

After this raw skeleton was ready, the team set up a pattern library section that Sara, the project lead we mentioned earlier, could `git-clone` the repository and start implementing both core patterns, like the grid, typography, and animated pull quotes before moving on to doing page layouts for the home page, article page, category page, and so on.

One of the things that really set the JAMstack approach apart compared to projects that Sara had worked on in the past was that the pattern library setup was the same as the production site setup. This meant that there was never any handover process. The core development team could simply start using the partials Sara defined to build out the actual content-driven pages and work within exactly the same framework.

Going from pattern library partials to a full site mainly involves figuring out how to get the right data from content into the partials and how to handle site listings, pagination, and relations while building out the site. This case study is not meant as a Hugo tutorial, but many of the techniques for this are similar regardless of the site generator used, so it will be useful to look at a few template examples without focusing too much of the specific idiosyncrasies of the Hugo template methods or its content model.

An obvious place to begin is looking at some of the common patterns on the main home page. Just below the fold, you'll find a list of the seven latest articles. Every good static site generator will have ways to output filtered and sorted lists of content. In the case of Hugo, the partial looks something like this:

```
<div class="article--grid__container">
 {{ first 7 (where .Data.Pages.ByDate.Reverse "Section"
 "articles")}}
   ...
   {{ partial "article--grid" . }}
   ...
 {{ end }}
</div>
```

Hugo has a composable query language that lets us filter a sorted list of all pages, select just the ones from the articles section, and then pick the first seven.

Before this list of latest articles, there's a slightly more complex situation, given the top part of *Smashing Magazine* consists of four large hand-curated articles picked by *Smashing*'s editorial staff.

To handle these curated sections, the team created a JSON data file called *curated.json* with a structure as follows:

```
featured_articles:
- articles/2018-02-05-media-queries-in-2018.md
- articles/2018-01-17-understanding-using-rest-api.md
- articles/2018-01-31-comprehensive-guide-product-design.md
- articles/2018-02-08-freebie-hand-drawn-space-icons.md
```

The editorial staff manages this list of curated articles, and in the home-page template uses this pattern to show the highlighted articles:

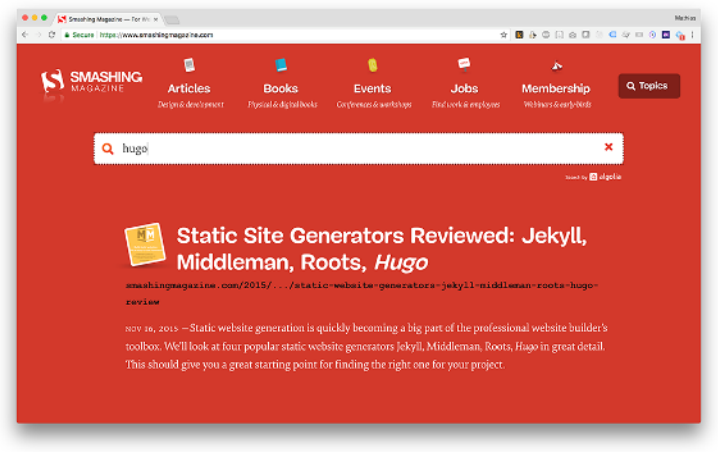
```
{{ range $index, $featured :=
   .Site.Data.curated.featured_articles }}
   {{ range where $.Data.Pages "Section" "articles"}}{{ if eq
   $featured .Path }}{{
partial "featured-article" . }}{{ end }}{{ end }}
{{ end }}
```

This is the kind of loop-based query that you'll run into often when working with static site generators. If *Smashing Magazine* was a dynamically generated site where these queries would be done at runtime, this would be an awful antipattern because looping through all articles once for every single featured article would put a strain on the database, slow down site performance, and potentially cause downtime during peak traffic.

There are places where we also need to watch out for long build times due to inefficient templates, but an example like this adds only milliseconds to the total build time of the site, and because the build is completely decoupled from the CDN delivery of the generated pages, we never need to worry about runtime costs of these kinds of queries.

Search

One thing we can't handle by generating listings up front is site search, as depicted in Figure 6-1, because this inherently needs to be done dynamically.



Static Site Generators Reviewed: Jekyll, Middleman, Roots, *Hugo*

smashingmagazine.com/2015/.../static-website-generators-jekyll-middleman-roots-hugo-review

NOV 16, 2015 – Static website generation is quickly becoming a big part of the professional website builder's toolbox. We'll look at four popular static website generators Jekyll, Middleman, Roots, *Hugo* in great detail. This should give you a great starting point for finding the right one for your project.

Figure 6-1. Smashing Magazine search

For search, *Smashing Magazine* relies on a Software as a Service (SaaS) offering called Algolia that provides lightning-fast real-time search and integrates from JavaScript in the frontend. There are other alternatives like Lunr that don't depend on any external services and instead generate a static index file that can be used on the client side to search for content. For large sites, however, a dedicated search engine like Algolia will perform better.

During each production build, a Gulp task runs that pushes all the content to Algolia's search index via its API.

This is where our asset pipeline and frontend framework become important. Because the actual search implementation happens client side, talking directly to Algolia's distributed search network from the browser, it was important to be able to build a clean component-based implementation of the search UI.

Smashing Magazine ended up using a combination of Preact and Redux for all the dynamic frontend components. We won't go in-depth with the boilerplate code for setting up the store and binding components to elements on the page, but let's take a look at how these kinds of components work.

The actual Algolia search is done within a Redux action. It allows you to separate the concern of talking to Aloglia's API and updating the global state of the app with the current search results from the concerns of how you present these results and from the UI layer that triggers a search.

Here's the relevant action:

```
import algoliasearch from 'algoliasearch'
export const client = algoliasearch(ENV.algolia.appId, ENV.
algolia.apiKey)
export const index  = client.initIndex(ENV.algolia.indexName)
export const search = (query, page = 0) => (dispatch, getState)
 => {
 dispatch({ type: types.SEARCH_START, query })
 index.search(query, { hitsPerPage: (page + 1) * 15, page: 0 },
  (error, results) =>
{
   if (error) return dispatch({
     type: types.SEARCH_FAIL,
     error: formatErrorMessage(error)
   })
   dispatch({ type: types.SEARCH_SUCCESS, results, page })
 })
}
```

With this in place, you can bind an `eventListener` to the search input on the site and dispatch the action for each keystroke with the query the user has typed:

```
props.store.dispatch(search(query))
```

You'll find placeholder `<div>`s in the markup of the site, like:

```
<div data-component="SearchResults" data-lite="true"></div>
```

The main *app.js* will look for tags with `data-component` attributes and then bind the matching Preact component to those.

In the case of the search results with the `data-lite` attribute, you can display a Preact component connected to the Redux store, that will display the first results from the store and link to the actual search result page.

This concept of decorating the static HTML with Preact components is used throughout the *Smashing* project to add dynamic capabilities.

One small extra detail for the site search is a fallback to doing a Google site search if JavaScript is disabled, by setting an action on the form that wraps the search input field:

```
<form data-handler="Search" method="get"
action="https://www.google.com/webhp?q=site:smashingmagazine.
```

```
com">
 <label for="js-search-input"></label>
 <div class="search-input-wrapper">
   <input type="search" name="q" id="js-search-input"
   autocomplete="off"
placeholder="Search Smashing..." aria-label="Search Smashing"
aria-controls="js-
search-results-dropdown" />
 </div>
</form>
```

In general, it's advisable to always have fallbacks where possible when JavaScript is not available, while recognizing that for modern web projects JavaScript is the main runtime, and most dynamic interactions won't work with the runtime disabled.

Content Management

There are a number of different approaches to content management on the JAMstack, but most involve a headless CMS. Traditionally, a CMS like Wordpress is both the tool used by admins to manage content and the tool used for building out the HTML pages from that content. A headless CMS separates those two concerns by focusing on only the content management, and delegating the task of using that content to build out a presentation layer to an entirely separate toolset.

You can find a listing of headless CMS offerings at *https://headlesscms.org/*. They're divided into two main groups:

API-based CMSs
: These are systems in which the content is stored in a database and made available through a web API. Content editors get a visual UI for authoring and managing content, and developers typically get a visual UI for defining content types and relations.

  Contentful, Dato CMS, and GraphCMS are leaders in the space.

Git-based CMSs
: These are tools that typically integrate with a Git provider's (GitHub, GitLab, Bitbucket) API layer, or work on a filesystem that can be synchronized with Git, to edit structured content stored directly in your Git repository.

Netlify CMS is the largest open source, Git-based CMS, while Cloudcannon and Forestry are the most popular proprietary CMS solutions based on this approach.

Both approaches have advantages.

API-based CMSs can be powerful when you're consuming the content from many sources (mobile apps, different landing pages, Kiosk software, etc.), and they can be better geared to handle highly relational content for which constraints on relations need to be enforced by the CMS. The trade-off is that to use them from a static build process, we'll always need a synchronization step to fetch all of the content from the remote API before running the build, which tends to increase the total build time.

Git-based CMSs integrate deeper into the Git-centric workflow and bring full version control to all your content. There are inherent advantages to having all content stored in a Git repository as structured data. As developers, we can use all the tools we normally use to work on text files (scripts, editors, grep, etc.), while the CMS layer gives content editors a web-based, content-focused UI for authoring or management.

Smashing Magazine chose to build the content-editing experience on the open source Netlify CMS to get the deepest integration into Git, the best local developer experience, and the fastest build times. For an authoring team as technical as *Smashing Magazine*, the option of sidestepping the CMS and working directly in a code editor to tweak code examples or embedded HTML is a huge advantage over any traditional, database-driven CMS.

Integrating Netlify CMS

When going from a working design or a pattern library to a CMS-backed site, the traditional approach was to integrate into a CMS (i.e., building out a Wordpress theme or the like). Netlify CMS reverses that process and instead allows you to pull the CMS into a site and configure it to work with the existing content structure.

The most basic integration of Netlify CMS consists of adding two files to a site: an */admin/index.html* file that loads the CMS SPA and a */admin/config.yml* configuration file where the content structure is described so that the CMS can edit it.

The base abstraction of Netlify CMS is a "collection," which can be either a folder with entries all having the same content structure, or a set of different files for which each entry has its own content structure (often useful for configuration files and data files).

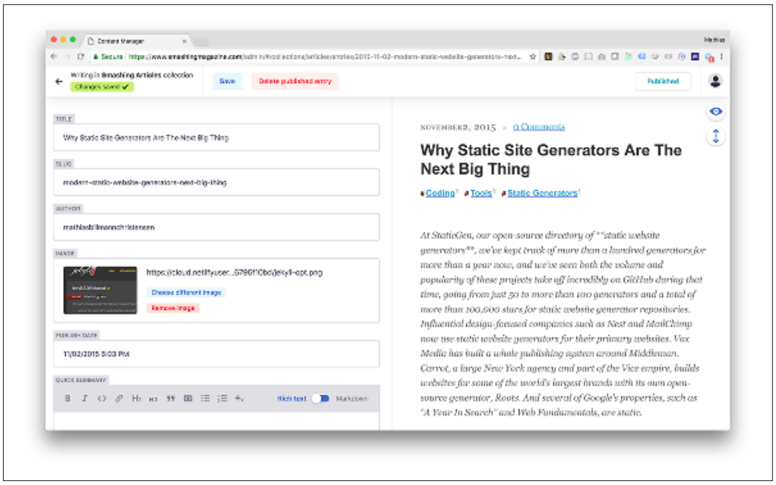
Here's the part of the CMS configuration that describes the full setup for the main `articles` collection:

```
collections:
 - name: articles
   label: "Smashing Articles"
   folder: "site/production-articles"
   sort: "date:desc"
   create: true # Allow users to create new documents in this
   collection
   slug: "{{slug}}"
   fields: # The fields each document in this collection have
     - {label: "Title", name: "title", widget: "string"}
     - {label: "Slug", name: "slug", widget: "string",
     required: false}
     - {label: "Author", name: "author", widget: "relation",
     collection: "authors",
searchFields: ["first_name", "last_name"], valueField:
"title"}
     - {label: "Image", name: "image", widget: "image"}
     - {label: "Publish Date", name: "date", widget:
     "datetime", format: "YYYY-MM-DD hh:mm:ss"}
     - {label: "Quick Summary", name: "summary", widget:
     "markdown", required: false}
     - {label: "Excerpt", name: "description", widget:
     "markdown"}
     - {label: "Store as Draft", name: "draft", widget:
     "boolean",required: false}
     - {label: "Disable Ads", name: "disable_ads", widget:
     "boolean", required: false}
     - {label: "Disable Panels", name: "disable_panels",
     widget: "boolean", required: false}
     - {label: "Disable Comments", name: "disable_comments",
     widget: "boolean", required: false}
     - {label: "Body", name: "body", widget: "markdown"}
     - {label: "Categories", name: "categories", widget:
  "list"}
```

The structure of the entries is described by a list of `fields`, where each field has a `widget` that defines how the value should be entered.

Netlify CMS comes with a set of standard widgets for `strings`, `markdown` fields, `relations`, and so on.

Based on this configuration, the CMS will present a familiar CMS interface (see Figure 6-2) for editing any article stored in the *site/production-articles* folder in the repository with a visual editor.



TITLE
Why Static Site Generators Are The Next Big Thing

SLUG
modern-static-website-generators-next-big-thing

AUTHOR
mathiasbiilmannchristensen

IMAGE
https://cloud.netlifyuser...5796f10bd/jekyll-opt.png

PUBLISH DATE
11/02/2015 5:03 PM

QUICK SUMMARY

NOVEMBER 2, 2015 • 0 Comments

Why Static Site Generators Are The Next Big Thing

Coding Tools Static Generators

*At StaticGen, our open-source directory of **static website generators**, we've kept track of more than a hundred generators for more than a year now, and we've seen both the volume and popularity of these projects take off incredibly on GitHub during that time, going from just 50 to more than 100 generators and a total of more than 100,000 stars for static website generator repositories. Influential design-focused companies such as Nest and MailChimp now use static website generators for their primary websites. Vox Media has built a whole publishing system around Middleman. Carrot, a large New York agency and part of the Vice empire, builds websites for some of the world's largest brands with its own open-source generator, Roots. And several of Google's properties, such as "A Year In Search" and Web Fundamentals, are static.*

Figure 6-2. Netlify CMS

Netlify CMS presents a live preview of the content being edited, but out of the box it was using a generic stylesheet that was not specific to the site.

Because Netlify CMS is a React SPA and very extensible, the best approach for the live preview is to provide custom React components for each collection and pull in the actual stylesheets from the site.

Smashing did this by setting up a *cms.js* entry that the Webpack part of the build would process, and then using that as the base for building out preview templates and custom widgets (a bit more on those when we dive into ecommerce later), and editor plug-ins to better support some of the short codes that content editors would otherwise need to know.

That looks roughly like the following:

```
import React from 'react';
import CMS from "netlify-cms";
import ArticlePreview from './cms-preview-templates/Article';
import AuthorPreview from './cms-preview-templates/Author';
// ... more preview imports
import {PriceControl} from './cms-widgets/price';
import {SignaturePlugin, PullQuotePlugin, FigurePlugin} from
```

```
'./cms-editor-
plugins/plugins';
window.CMS.registerPreviewStyle('/css/main.css');
window.CMS.registerPreviewStyle('/css/cms-preview.css');
window.CMS.registerPreviewTemplate('articles', ArticlePreview);
window.CMS.registerPreviewTemplate('authors', AuthorPreview);
// ... more preview registrations
window.CMS.registerWidget('price', PriceControl);
window.CMS.registerEditorComponent(FigurePlugin);
window.CMS.registerEditorComponent(PullQuotePlugin);
window.CMS.registerEditorComponent(SignaturePlugin);
```

This imports the core CMS module, which initializes the CMS instance and loads the configuration and then registers preview styles (the CSS to be included in the preview pane), preview templates (to customize previews for the different collections or data files), preview widgets (to provide custom input widgets for this project), and editor components (to add custom buttons to the rich-text editor).

The preview components are pure presentational react components that mimic the markup used for the real articles, authors, books, and so on.

A simplified version of the article component looks like this:

```
import React from 'react';
import format from 'date-fns/format';
import AuthorBio from './author-bio';
export default class ArticlePreview extends React.Component {
 render() {
   const {entry, fieldsMetaData, widgetFor, getAsset} = this.
   props;
   const data = entry && entry.get('data').toJS();
   const author = fieldsMetaData.getIn(['authors', data.
   author]);
   return <article className="block article" role="main">
     <div className="container">
       <div className="row">
         <div className="col col-12 col--article-head">
           {author && <AuthorBio author={author.toJS()}
           getAsset={getAsset}/>}
           <header className="article__header">
             <div className="article__meta">
               <time className="article__date">
                 <span className="article__date__month">{
                 format(data.date, 'MMMM')
}</span>
                 { format(data.date, 'D, YYYY') }
               </time>
               <svg aria-hidden ="true" style={{margin: '0
               0.75em'}} viewBox="0 0 7
7" width="7px" height="7px">
                 <title>bullet</title>
                 <rect fill="#ddd" width="7" height="7" rx="2"
                 ry="2"/>
               </svg>
               <span className="article__comments-count">
               <a href="#">
                 <span className="js-comments-count">0</span>
                 Comments
               </a></span>
             </div>
             <h2>{ data.title }</h2>
             <div className="article__tags">
               {data.categories && data.categories.map
                  ((category) => (
                 <span className="article__tag" key={category}>
                   <a href="#">{ category }</a>
                   <sup className="article__tag__count">1</sup>
                 </span>
               ))}
             </div>
           </header>
         </div>
         <div className="col col-4 col--article-summary">
           <p className="article__summary">
             { data.description }
           </p>
         </div>
         <div className="col col-7 article__content">
           { widgetFor('body') }
         </div>
       </div>
     </div>
   </article>;
 }
}
```

Listings, Search, and Assets: Customizing the CMS for Large Projects

Out of the box, Netlify uses GitHub's content API to access all the content, and generate listings and filtering for search queries. The standard setup also assumes that all media assets are stored in the Git repository.

For really large sites like *Smashing Magazine*, this approach begins breaking down. *Smashing Magazine* has more than a terabyte of assets in the form of uploaded images, media files, PDFs, and others that's much more than it would ever want to store in its Git repository. Apart from that, GitHub's API begins breaking down for content listings when there's more than 2,000 files in a folder—obviously the case for *Smashing Magazine*'s article collection.

Search is not just an issue within the CMS, but also for the actual website, and this will typically be the case for any larger content-driven site. So why not use the same search functionality in both places?

We shared earlier how Algolia was used to add site search functionality to *Smashing Magazine*'s core site. Netlify CMS allows integrations to take over functionality like asset management, collection search, or collection listings. It also comes with a plug-in for using Algolia for all search and listings instead of building this on top of the Git repository folder listings. This makes listing snappy even for very large collections.

In a similar way, the media library functionality of Netlify CMS allows for integrations, and *Smashing Magazine* uses this to store assets in an external asset store where they are directly published to a CDN when uploaded. That way only references to the asset URLs are stored in the Git repository.

This extensible architecture allows Netlify CMS to scale to large production sites while sticking to the core principle of bringing content editors into the same Git-based workflow we take for granted as developers.

Every architectural decision we make as developers has some trade-offs. When we work with a Git-based architecture, we might need to supplement our toolchain with asset stores, external search engines, and the like, but few developers would voluntarily agree to the trade-offs that come with storing all their code in an SQL database. So, why do it for your content when that's inherently the most important, core part of any website?

Identity, Users, and Roles

When working with traditional monolithic applications, we typically have a built-in concept of a user that's backed by a database collection and integrated throughout the monolithic application. This was one of the challenges of *Smashing Magazine*'s previous architecture:

each of the monolithic applications it relied on (Wordpress, Shopify, Kirby, Rails) had its own competing concept of a user.

But as the magazine moved from monoliths to composing individual microservices for all dynamic functionality, how could it avoid making this problem even worse?

JWTs and Stateless Authentication

An important part of the answer to this is the concept of stateless authentication. Traditionally, authentication was stateful in the sense that when doing a page request to a server-rendered app, we would set a cookie with a session ID and then the application would check that against the current state in the database and decide the role, display name, and so on of the user based on this lookup.

Stateless authentication takes a different approach. Instead of passing a session ID when we call a microservice, it passes a representation of the user. This could look something like the following:

```
{
 "sub": "1234",
 "email": "matt@netlify.com",
 "user_metadata": {
   "name": "Matt Biilmann"
 },
 "app_metadata": {
   "roles": ["admin", "cms"],
   "subscription": {
     "id": "4321",
     "plan": "smashing"
   }
 }
}
```

Now each microservice can look at this payload and act accordingly. If an action requires an admin role, the service can check whether `app_metadata.roles` in the user payload includes an admin role. If the service needs to send an order confirmation to the user, it can rely on the `email` property.

This means that each microservice can look for the payload information it needs without having any concept of a central database with user information and without caring about which system was used to authenticate the user.

However, we need a way to establish trust and ensure that the payload of the user really represents a real user in the system.

JSON Web Token (JWT) is a standard for passing along this kind of user payload while making it possible to verify that the payload is legitimate.

A JWT is simply a string of the form `header.payload.signature` with each part of bas64 encoded to make it safe to pass around in HTTP headers or URL query parameters. The header and payload are JSON objects, and the signature is a cryptographic signature of the header and the payload.

The header specifies the algorithm used to sign the token (but the individual microservices should be strict about what algorithms they accept—the most common is HS256). Each token will be signed with either a secret or with a private key (depending on the algorithm used). This means that any microservice that knows the secret (or has a corresponding public key) can verify that the token is legitimate and trust the payload.

Normally, a JWT will always have an `exp` property that sets the expiration time for the token. A short expiration time ensures that if a token is leaked in some way, there's at most a short time window to exploit it and it can't simply be used to take over the role of a user. For that reason, JWTs are typically paired with a refresh token that can be used to retrieve a new JWT from the authentication service as long as the user is still logged in.

The Auth0 team that pioneered this standard maintains a useful resource where you can find more in-depth information and debugging tools for inspecting and verifying tokens.

API gateways and token signatures

With a naive approach to stateless authentication, every microservice in the *Smashing Magazine* system would need to know the same secret in order to verify the user payload. This increases the risk of leaking the secret and means that every service would be capable of issuing its own valid JWTs instead of limiting the capability to just the main identity service.

One approach to get around this is to use a signing algorithm that relies on private/public cryptographic keypairs. Then, all the services will need to look up a public key and use it to verify that the signature was generated with a private part of the key pair. This is efficient but can introduce quite a bit of complexity around having to fetch and cache public keys and gracefully handle public key rotations.

Another approach is to have an API gateway between the client and the different microservices. This way, only the gateway needs to share the secret with the identity service and can then re-sign the JWT with a secret specific to each individual microservice. It allows the system to rotate secrets independently for different services without running the risk of sharing secrets across multiple services (where some might even be owned by third parties).

GoTrue and Netlify identity

Smashing Magazine uses an open source microservice, GoTrue, to handle user sign-up and logins. It's a microservice written in Go with a small footprint.

The main endpoints for the microservice are:

- **POST /signup**: Sign up a new user
- **POST /verify**: Verify the email of a user after sign-up
- **POST /token**: Issue a short lived JWT + a refresh token for a user
- **POST /logout**: Revoke the refresh token of the user

All of these end points are consumed client side, and the login and sign-up forms for *Smashing Magazine* are again built as client-side Preact components.

When a user signs up for *Smashing Magazine*, the Preact component triggers a Redux action that uses the `gotrue-js` client library like this:

```
return auth.signup(data.email, data.password, {
 firstname: data.firstname,
 lastname: data.lastname
}).then(user => {
 return auth.login(data.email, data.password, true).then(user
 => {
   persistUserSession(user, noRedirect);
   return user;
 })
})
```

User sessions are saved in `localStorage`, so the UI can access the user payload and use it to show the user's name where it makes sense or pass on the JWT to other microservices where it makes sense.

We see later how the identity of a user can be represented as a JWT and used in tandem with the other services that back the dynamic functionality of *Smashing Magazine*.

The idea of stateless authentication is one of the fundamental architectural principles that makes modern microservice-based architecture viable—and it's something core to the emergence of an ecosystem of microservices that can be combined and plugged together without any foreknowledge of one another, much in the same way that the Unix philosophy lets us use pipes to combine lots of small independent command-line tools.

Where the large monolithic apps each came with their own ecosystem inside (Wordpress plug-ins, RubyGems for Rails, etc.), we're now seeing a new ecosystem emerge at the level of ready-made microservices, both open source and fully managed, that frontend developers can use to build large applications without any backend support.

Ecommerce

Smashing Magazine derives a large part of its income from its ecommerce store that sells printed books, ebooks, job postings, event tickets, and workshops.

Before the redesign, *Smashing Magazine* used Shopify, a large, multitenant platform that implements its own CMS, routing layer, template engine, content model, taxonomy implementation, checkout flow, user model, and asset pipeline.

As part of the redesign, *Smashing Magazine* implemented a small, open source microservice called GoCommerce. It weighs in at about 9,000 lines of code, including comments and whitespace.

GoCommerce doesn't implement a product catalog and it has no user database, routing logic, or template engine. Instead, its essence is two API calls:

- **POST /orders**

Create a new order, with a list of line items, example: `{"line_items": ["/printed-books/design-systems/"]`

GoCommerce fetches each path on the site and extracts metadata with pricing data, product type (for tax calculations), title, and so on and constructs an order in its database. The website itself is the canonical product catalog.

- **POST /orders/:order_id/payments**

  Pay for an order with a payment token coming from either Stripe or Paypal.

GoCommerce will validate that the amount matches the calculated price based on the product metadata and tax calculations (as well as any member discounts, coupons, etc.).

The product catalog is built out by Hugo during the build step, and all products are managed with Netlify CMS. A simplified version of an ebook file stored in the Git repository looks like this:

```
---
title: 'A Career On The Web: On The Road To Success'
sku: a-career-on-the-web-on-the-road-to-success
image: >-
 //cloud.netlifyusercontent.com/assets/344dbf88-fdf9-42bb-adb4-
46f01eedd629/133e54ba-e7b5-40ff-957a-f42a84cf79ff/on-the-road-
to-success-365-
large.png
description: >-
 There comes a time in everyone's career when changing jobs is
 the natural next step. But how can you make the most of this
 situation and find a job you'll love?
price:
 eur: '4.99'
 usd: '4.99'
published_at: March 2015
isbn: 978-3-945749-17-3
files:
 pdf: >-
   //api.netlify.com/api/v1/sites/344dbf88-fdf9-42bb-adb4-
46f01eedd629/assets/58eddda0351deb3f6122b93c/public_signature
---

<p>There comes a time in everyone's career when changing jobs
is the natural next step. Perhaps you're looking for a new
challenge or you feel like you've hit a wall in your current
company? Either way, you're standing at a crossroad, with an
overwhelming amount of possibilities in front of you. But how
```

```
can you make the most of this situation? How can you <strong>
find a job you will truly love</strong>?</p>
```

When Hugo builds out the detail page for an ebook, it outputs a `<script>` tag with metadata used for GoCommerce that looks like this:

```
<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/344dbf88-fdf9-
 42bb-adb4-
46f01eedd629/3eb5e393-937e-46ac-968d-249d37aebcef/assuming-
leadership-365-
large.png",
 "type": "E-Book",
 "prices": [
   {"amount": "4.99", "currency": "USD"},
   {"amount": "4.99", "currency": "EUR"}
 ],
 "downloads": [{"format":"pdf","url":"//api.netlify.com/api
 /v1/sites/344dbf88-fdf9-
42bb-adb4-46f01eedd629/assets/58edd9f7351deb3f6122b911/public_
signature"}]
}
</script>
```

This is the canonical product definition that will be used both when adding the product to the cart client side and when calculating the order price in GoCommerce.

GoCommerce comes with a small JavaScript library that helps with managing a shopping cart, does the same pricing calculations client-side as GoCommerce will do server side when creating an order, and handles the API calls to GoCommerce for creating orders and payments and querying order history.

When a user clicks a “Get the e-book” button, a Redux action is dispatched:

```
gocommerce.addToCart(item).then(cart => {
 dispatch({
   type: types.ADD_TO_CART_SUCCESS,
   cart,
   item
 })
})
```

The item we add to the card has a path, and the GoCommerce library will fetch that path on the site with an Ajax request and look for the `<gocommerce-product>` script tag and then extract the metadata for the product, run the pricing calculations, and store the updated shopping cart in `localStorage`.

Again, we have a Preact component connected to the Redux store that will display the shopping cart in the lower-left corner of the page if there are any items in the store.

The entire checkout process is implemented completely client side in Preact. This proved tremendously powerful for the *Smashing* team because it could customize the checkout flows in so many different ways. Tweak the flow for a ticket to include capturing attendee details, tweak the flow for a job posting to include capturing the job description and settings, and tweak the flow for products requiring shipping versus purely digital products like ebooks.

There's an open source widget available at *https://github.com/netlify/netlify-gocommerce-widget* that has an example of this kind of checkout flow implemented in Preact + MobX, and you can even use it as a plug-and-play solution for a GoCommerce site.

During the checkout flow, Preact forms capture the user's email, billing, and shipping address, and the user can either select PayPal or credit card as a payment option. In either case their respective JavaScript SDKs are used client side to get a payment token for the order. The last part of the checkout flow is a confirmation screen with the order details. When the user selects "Confirm," the following Redux action takes care of the actual sale:

```
export const putOrder = (dispatch, getState) => {
 const state = getState()
 const { order, cart, auth } = state
 dispatch({ type: types.ORDER_CONFIRM_START })
 // Create a new order in Gocommerce
 gocommerce.order({
   email: order.details.email,
   shipping_address: formatAddress(order.details.
   shippingAddress),
   billing_address: formatAddress(order.details.billingAddress)
 }).then(result => {
   // Create a payment for the order with the stripe token or
   paypal payment
   return gocommerce.payment({
       order_id: result.order.id,
       amount: result.cart.total.cents,
       provider: order.paypal ? 'paypal' : 'stripe',
       stripe_token: order.key,
       paypal_payment_id: order.paypal && order.paypal.
       paymentID,
       paypal_user_id: order.paypal && order.paypal.payerID
   }).then(transaction => {
     // All done, clear the cart and confirm the order
     if (auth.user) dispatch(saveUserMetadata(order.details))
     dispatch(emptyCart)
     dispatch(clearOrder)
     dispatch({
       type: types.ORDER_CONFIRM_SUCCESS,
       transaction: Object.assign({}, transaction, { order:
       result.order }),
       cart
     })
   })
 }).catch(error => dispatch({
   type: types.ORDER_CONFIRM_FAIL,
   error: formatErrorMessage(error)
 }))
}
```

There's a bit to digest there, but the core of it comes down to first calling `gocommerce.order` with the email and shipping or billing addresses of the user, and then the GoCommerce library adds the items from the cart to the order.

As long as that goes well, we then call `gocommerce.payment` for the order with the amount we've shown to the user client side and a payment method and token.

GoCommerce looks up the actual line items from the order; does all the relevant pricing calculations, taking into account taxes, discounts, and coupons; and verifies that the amount we've shown client side matches the calculated order total. If all is well, GoCommerce triggers a charge with the relevant payment method. If that works, it sends out a confirmation email to the user and an order notification to the shop administrator.

Identity and Orders

GoCommerce has no user database and no knowledge about the identity service in use. At the same time, the setup needs to ensure that logged-in users are able to access their order history and previous shipping or billing addresses.

Stateless authentication to the rescue: you can configure GoCommerce with a JWT secret, and then any API request signed with a JWT that can be verified with that secret will be associated with a user, based on the `sub` attribute of the JWT payload. Note that the attribute is part of the JWT standard and indicates the unique ID of the user identified by the token.

Client side this is handled in the JS library by using `gocommerce.setUser(user)` with a user object that responds to the `user.jwt()` method that returns a token wrapped in a promise. The promise part is important because the user object might need to exchange a refresh token to a valid JWT in the process.

After the user is set, the GoCommerce library signs all API requests with a JWT.

Calling the `GET /orders` endpoint with a JWT returns a listing of all the orders belonging to the user identified by the token. Using that endpoint and the identity service, *Smashing Magazine* could build out the entire order history panel client side with Preact components.

Membership and Subscriptions

The introduction of a membership feature was one of the biggest motivations for *Smashing Magazine*'s decision to relaunch its site and build a new architectural platform from the ground up. It wanted to offer readers a three-tiered subscription plan.

So far, we've seen that the magazine tackled identity and ecommerce with open source microservices. The implementation of subscriptions introduces a new pattern that's becoming an increasingly popular component of JAMstack projects: using serverless functions to glue together existing services and introduce new functionality.

Serverless is often a contested term. Obviously, servers are still involved somewhere, but the option to simply write some glue code without having to worry about where and how that code is executed can be an incredibly powerful part of our toolbox.

At this phase of the migration, *Smashing Magazine* had all of the pieces that needed to be stitched together in place: GoTrue for identifying users, Stripe for accepting payments and handling recurring billing, MailChimp to handle email lists with groups for member and plan-specific emails, and Netlify's CDN-based rewrite rules to selectively show different content to users with different roles.

To build a fully-fledged subscription service, all it needed was a bit of glue code to tie all of these together. AWS Lambda functions proved to be a great way to run this code without having to worry about running a server-side application somewhere.

To become a member, a user needs to fill out the sign-up form shown in Figure 6-3.

Let's go through what happens when this Preact component-based form is filled out and the user clicks the action button.

Assuming that all of the data entered passes the basic client-side validations, a new user is created in GoTrue via the sign-up endpoint and the script gets hold of a JWT representing the new user.

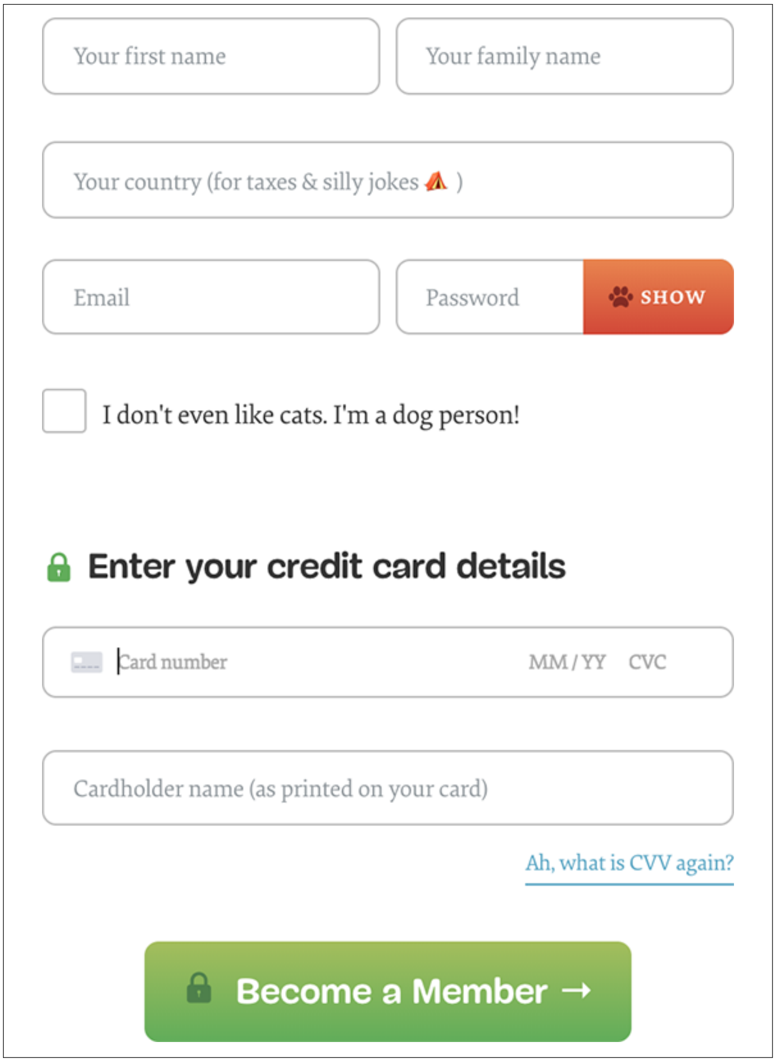

Your first name
Your family name
Your country (for taxes & silly jokes ⛺)
Email
Password
SHOW
I don't even like cats. I'm a dog person!
Enter your credit card details
Card number
MM / YY
CVC
Cardholder name (as printed on your card)
Ah, what is CVV again?
Become a Member →

Figure 6-3. Smashing Magazine sign-up form

Then, Stripe's "Checkout" library is used to exchange the credit card details for a Stripe token.

After these two preconditions are both successful, a Redux action is triggered, calling a Lambda function deployed via Netlify and triggered by an HTTP request:

```
fetch('/.netlify/functions/membership', {
 headers: {Authorization: `Bearer ${token}`},
 method: 'POST',
 body: JSON.stringify({ plan, stripe_token: key })
})
```

The Lambda function handler then triggers a method called `subscribeToPlan` that looks like this:

```
function subscribeToPlan(params, token, identity) {
 // Fetch the user identified by the JWT from the identity
 service
 return fetchUser(token, identity).then(user => {
   console.log("Subcribing to plan for user: ", user);

   // Check if the user is already linked to a Stripe Customer
   // if not, create a new customer in Stripe
   const customer_id = user.app_metadata.customer
     ? user.app_metadata.customer.id
     : createCustomer(user);
   return Promise.resolve(customer_id)
     .then(customer_id => {
       // Check if the user has an existing subscription
       if (user.app_metadata.subscription) {
         // Update the existing Stripe subscription
         return updateSubscription(
           customer_id,
           user.app_metadata.subscription.id,
           params
         ).then(subscription => {
           // Add the user to the MailChimp list and the right
           Group
           addToList(user, params.plan);
           // Notifiy about the change of plan in Slack
           sendToSlack(
             'The user ' + user.email + ' changed from ' +
             user.app_metadata.subscription.plan + ' to ' +
             params.plan
           );
           return subscription;
         });
       }

       // No existing subscription, create a new subscription
       in Stripe
       return createSubscription(customer_id, params).then(
       subscription => {
         // Add the user to the MailChimp list and the right
         Group
         addToList(user, params.plan);
         // Notify about the new subscriber in Slack
         sendToSlack('Smashing! The user ' + user.email + '
         signed up for a ' +
params.plan + ' plan');
         return subscription;
       });
     })
     // In the end, update the user in the identity service.
     // This will update app_metdata.customer and app_
     metadata.subscription
     // for the user
     .then(subscription => updateUser(user, subscription,
     identity));
 });
}
```

There are a few things happening here. For each of the external services, like Stripe, MailChimp, and Slack, the Lambda function has access to the relevant API keys and secrets via environment variables. When working with different external APIs, this is one of the key things that we can't do in client-side JavaScript, because no variable exposed there can be kept secret.

In this case, the behavior of the identity service is a little different. When a Lambda function is deployed via Netlify on a project that has an identity service, the function will have privileged access to the identity service. This is a common pattern, and you can similarly set up Lambda function using Amazon CloudFront and Amazon API Gateway together with AWS's identity service (Cognito) to have privileged access.

Note how the user metadata is fetched from the identity server instead of relying on the data from the JWT. Because JWTs are stateless and have a lifetime, there are cases for which the information could be slightly out of date, so it's important to check against the current state of the user.

The effect of the Lambda function is to use the Stripe token to create a recurring subscription in Stripe and add the Stripe customer and subscription ID to the `app_metedata` attribute of the user. It also subscribes the email of the user to a MailChimp list and triggers a notification in an internal Slack channel about the new subscriber.

After a subscriber has an `app_metadata.plan` attribute, *Smashing Magazine* takes advantage of Netlify's JWT-based rewrite rules to show different versions of the membership pages depending on which user is logged in. The same could be achieved with other CDNs that allow edge logic around JWT, edge Lambda functions, or similar edge functions.

This highlights how something like a full membership engine was built with almost no custom server-side code (the full *membership.js* file is 329 lines long), and with a serverless stack where the developers on the project never had to worry about operational concerns around their code.

Tying It Together: Member Discounts in GoCommerce

A selling point of memberships is discounts on books, ebooks, and conference tickets. But how do we approach this in the JAMstack world of small, independent microservices?

Although GoCommerce has no knowledge of the existence of the GoTrue-based identity service, it understands the concept of JWTs and can relate an order to a user ID by looking at the `sub` property of the token payload.

In the same way, GoCommerce can apply discounts to orders if a predefined discount matches the token payload. But where do we define the discounts?

Again, GoCommerce uses the technique of making the website the single source of truth. Just like GoCommerce uses the website as the authoritative product database by looking up product metadata on a path, it also loads a */gocommerce/settings.json* file from the website and refreshes this regularly.

The *settings.json* holds all the tax settings that both GoCommerce and the `gocommerce-js` client library use for pricing calculation. It can also include a `member_discounts` setting looking something like this:

```
"member_discounts": [
 {
   "claims": {"app_metadata.subscription.plan": "member"},
   "fixed": [
     {"amount": "10.00", "currency": "USD"},
     {"amount": "10.00", "currency": "EUR"}
   ],
   "product_types": ["Book"]
 }
]
```

This tells GoCommerce that any logged-in user who has a JWT payload looking something like what follows should get a discount on any line item with the type “Book” (determined from the product metadata on the product page) of either $10 or 10€ depending on the currency of the order:

```
{
 "email": "joe@example.com",
 "sub": "1234",
 "app_metadata": {"subscription": {"plan": "member"}}
}
```

Note again how there’s no coupling at all between GoCommerce and the identity service or the format in which membership information is represented in the JWT payload. Any pattern in the claim can be used to define a user with a status that should give special discounts. GoCommerce also doesn’t care how the *settings.json* is managed. We could easily set up Netlify CMS to give a visual UI for managing this, generate it from data stored in an external API, fetch the data from a Google Sheet, and generate the JSON or have developers edit it by hand. All these concerns are kept completely separate.

Job Board and Event Tickets: AWS Lambda and Event-Based Webhooks

In the previous section, we saw how GoCommerce could act on a user’s membership plan and calculate discounts, without any coupling between the membership implementation, the identity service, and GoCommerce.

The implementation of *Smashing Magazine*’s job board highlights a similar concern. In this case, some action needs to happen when someone buys a product with the type “Job Posting,” but because GoCommerce is a generic ecommerce microservice, it has no knowledge of what a “Job Posting” is or how to update a job board.

However, GoCommerce has a webhook system that allows it to trigger HTTP POST requests to a URL endpoint when an order has been created. Webhook requests are signed with a JWT-based signature, allowing the service receiving the request to verify that it’s generated from an actual GoCommerce order and not by someone triggering the hook directly.

This again offers us the ability to glue loosely coupled components together by using serverless functions.

Smashing Magazine approached its job board as any other listing/detail part of its site, built from a collection of markdown files with frontmatter by Hugo.

Instead of somehow integrating job board functionality into GoCommerce, the team deployed an AWS Lambda function that serves as the webhook endpoint for GoCommerce payments. This means that each time an order has been completed and paid for, GoCommerce triggers the Lambda function with a signed request. If the "type" of the product in the event payload is a "Job Posting," a `processJob` method is triggered:

```
function processJob(payload, job) {
 console.log("Processing job", job.meta);
 const content = `---\n${yaml.safeDump({
   title: job.meta.title || null,
   order_id: payload.order_id || null,
   date: payload.created_at || null,
   logo: job.meta.logo || null,
   commitment: job.meta.commitment || null,
   company_name: job.meta.company_name || null,
   company_url: job.meta.company_url || null,
   jobtype: job.meta.jobtype || null,
   location: job.meta.location || null,
   remote: job.meta.remote || null,
   application_url: job.meta.application_url || null,
   featured: job.meta.featured || false
 })}\n---\n\n${job.meta.description || ""}`;
 const path = `site/content/jobs/${format(
   payload.created_at,
   "YYYY-MM-DD"
 )}-${payload.id}-${urlize(job.title)}.md`;
 const branch = "master";
 const message = `Create Job Post
This is an automatic commit creating the Job Post:
"${job.meta.title}"`;
 return fetch(
   `https://api.github.com/repos/smashingmagazine/smashing-
magazine/contents/${path}`,
   {
     method: "PUT",
     headers: { Authorization: `Bearer ${process.env.
     GITHUB_TOKEN}`
  },
     body: JSON.stringify({
       message,
       branch,
       content: new Buffer(content).toString("base64")
     })
```

```
        }
      );
    }
```

This extracts the metadata related to the job posting from the order object, formats it into markdown with YAML frontmatter, and then uses GitHub's content API to push the new job posting to the main *Smashing Magazine* repository.

This in turn triggers a new build, where Hugo will build out the updated version of the job board and Netlify will publish the new version.

In a similar way, orders for event tickets trigger a function that adds the attendee information to a Google Sheet for the event that the *Smashing* team uses to print badges and verify attendees at the event check-in. All of this happens without any need for GoCommerce to have any concept of a ticket or an event besides the product definition in the metadata on each event page.

Workflows and API Gateways

This case study has shown a consistent pattern of a core, prebuilt frontend, using loosely coupled microservices, glued together with serverless functions either as mini-REST endpoints or as event-triggered hooks.

This is a powerful approach and a big part of the allure of the JAMstack. Using a CDN-deployed static frontend that talks to dynamic microservices and uses serverless functions as a glue layer, small frontend teams can take on large projects with little or no operations and backend support.

Pulling this off does require a high level of maturity in automation, API routing, secrets management, and orchestration of your serverless code, with a viable workflow and a solid, maintainable stack.

Making the CMS integration work, indexing to an external search engine viable, and powering the Git-based job board required a tightly integrated continuous deployment pipeline for the Gulp-based build and ample support for staging environments and pull request previews.

The integrated, lightweight API gateway layer was necessary to manage routing to the different microservices and ensure one set of services was used from pull request or staging deploys while the production instances were used from master branch deploys.

In this case, Netlify's gateway layer also handled key verification at the edge, so each microservice could have its own JWT secret and only the CDN had the main identity secret. This allows the CDN to verify any JWT from the identity service in a call to a service and swap it with a JWT with the same payload that is signed with the secret specific to the individual service.

It also proved key to having an integrated workflow for deploying the serverless functions together with the frontend and automating the routing and invocation layer, so that testing an order checkout flow with a deploy preview of a pull request would trigger the pull request–specific version of the webhook Lambda function.

We can't imagine that anyone working on large backend or infrastructure system based on a microservice architecture would dream of doing this without a solid service discovery layer. In the same way, we see service discovery for the frontend becoming not only more relevant, but essential as developers move toward decoupled, microservice-based architectures for our web-facing properties.

Deploying and Managing Microservices

The new *Smashing Magazine* is essentially a static frontend served directly from a CDN, talking through a gateway to various microservices. Some of them are managed services like Stripe or Algolia, but the project also had several open source microservices like GoTrue, GoCommerce, Git Gateway, as well as GoTell, and in the beginning the different webhook services were deployed as one service called Smashing Central.

At the start of the project, all of these were deployed to Heroku and used Heroku's managed Postgres database for the persistence layer.

One thing that's essential when working with a microservice-based architecture is that each microservice should own its own data and not share tables with other services. Having multiple services share the same data is an antipattern and makes deploying changes with migrations or making hosting changes to initial services a cumbersome process. Suddenly all the microservices are tightly coupled and you're back to building a monolith but with more orchestration overhead.

After Netlify launched managed versions of GoTrue, GoCommerce, and Git Gateway, we migrated the endpoints there. These services run in a Kubernettes cluster, and this is becoming a more and more standard setup for the microservice layer in the JAMStack. The team could do this as a gradual process, moving these services one by one to the managed stack without interruption given that there was no coupling between them.

The Smashing Central service that was the one microservice written specifically for the *Smashing Magazine* process eventually was completely replaced with Lambda functions deployed through Netlify together with the frontend.

We generally see serverless functions as the preferred option whenever there are small custom microservices that don't need a persistence layer, with Kubernetes (often in managed versions) emerging as the main deployment target for services that have too large of a surface area to be a good fit for systems like AWS Lambda.

Summary

Smashing Magazine moved from a system built from four traditional monolithic applications, with four independent frontend implementations of the same visual theme that had been adapted to the quirks of each platform, to a setup in which the entire implementation was driven by and based on one single static frontend talking to a set of different microservices using a few serverless functions as the glue in the middle.

The team that built this project was very small and entirely frontend focused. All of the backend services were either open source microservices (GoTrue and GoCommerce) or third-party managed services (Stripe, Algolia, and MailChimp). The total amount of custom, server-side code for this entire project consisted of 575 lines of JavaScript across 3 small AWS Lambda functions.

Any interaction a visitor has with the site while browsing magazine content, viewing the product catalog, or perusing the job board for new opportunities happens without any dynamic, server-side code —it's just a static frontend loaded directly from the closest CDN edge node. It is only when an action is taken (confirming an order, posting a comment, signing up for a plan, etc.) that any dynamic code is used, either in the form of microservices or Lambda functions.

This makes the core experience incredibly performant and removes all maintenance concerns from essential parts of *Smashing Magazine*. Before the change, the kind of viral traffic that *Smashing Magazine* frequently received would cause reliability issues regardless of the amount of Wordpress caching plug-ins in use. Plus, the work required to constantly keep Wordpress (and Rails and Kirby) up to date without breaking the site ate up most of the budget for site improvements or updates.

With the new site, it's been straightforward for the team to work on new sections, design tweaks, cute little touches to the different checkout flows, and performance improvements. Anyone can run the full production site locally just by doing a Git clone of the repository and spinning up a development server, with no need to set up development databases. Any pull request to the site is built to a new preview URL where it can be browsed and tested before being merged in—including any new article that the team is preparing to publish.

The Git-based CMS has proved a great approach for allowing both web-based content editing with live previews. It has also enabled developers to dive in through their code editors. Having all files as plain text in a Git repository makes scripting Bash operations (like inserting ad panels or doing content migrations) easy and brings full revision history to all content edits.

Though tools like Netlify CMS or the GoCommerce store admin are not currently as mature as the tools with 15-plus years of history, and there are still some rough edges to be worked out, it's without a doubt that the *Smashing Magazine* team has benefited significantly from this shift. And perhaps more important, so have its readers.

CHAPTER 7
Conclusion

And One More Thing...

When we talk about the JAMstack, it's easy to drill into the advantages that it has for us as developers and the projects we build. But there's something larger at stake that the JAMstack addresses, and we'd be remiss not to discuss it here.

The JAMstack is a response to some of the most urgent threats to the web. But first, a quick history lesson.

When the iPhone was brand new and Steve Jobs was busy killing off Flash, HTML5 applications were meant to be the main delivery mechanism for all third-party applications on the Apple ecosystem. Jobs wrote in his famous open letter to Adobe:

> Adobe's Flash products are 100% proprietary. They are only available from Adobe, and Adobe has sole authority as to their future enhancement, pricing, etc. While Adobe's Flash products are widely available, this does not mean they are open, since they are controlled entirely by Adobe and available only from Adobe. By almost any definition, Flash is a closed system.
>
> Apple has many proprietary products too. Though the operating system for the iPhone, iPod and iPad is proprietary, we strongly believe that all standards pertaining to the web should be open. Rather than use Flash, Apple has adopted HTML5, CSS and JavaScript – all open standards. Apple's mobile devices all ship with high performance, low power implementations of these open standards. HTML5, the new web standard that has been adopted by Apple, Google and many others, lets web developers create

> advanced graphics, typography, animations and transitions without relying on third party browser plug-ins (like Flash). HTML5 is completely open and controlled by a standards committee, of which Apple is a member.

However, web applications at the time lacked the performance and user experience needed to drive the new mobile development system. Ironically, Apple ended up with a proprietary ecosystem for which it was the sole gatekeeper.

Still, the web is the *one* place where anyone can have the freedom and ability to publish to a global audience in an instant—without asking anyone for permission. But if the web fails to deliver on performance, fails to provide suitable security, or fails to deliver a high enough quality user experience, it will eventually lose out to walled gardens with strict corporate gatekeepers.

Large companies are aware of the impact such things have on them, and the benefits that good performance can have on their customers. We are already seeing companies such as Google and Facebook tackle this challenge by introducing their own technologies like Accelerated Mobile Pages (AMP) and Instant Articles to deliver better performance and security, but at the cost of driving publishers to create content within the boundaries of these organizations rather than in the commons of the open web.

Solving the performance, security, and scaling issues of the legacy web while building responsive and engaging user interfaces that can truly delight the people using them is vital to keeping the web competitive, alive, and in good health.

The JAMstack is up for the challenge, providing a set of architectural practices, constraints, and guidelines that aims to make the web faster, safer, and simpler.

About the Authors

Mathias Biilmann is the technical cofounder and CEO of Netlify, a platform for modern web development used by hundreds of thousands of developers worldwide. An active participant in open source, Mathias has contributed to the following well-known projects: Netlify CMS, Ruby on Rails, JRuby, and Mongoid.

Phil Hawksworth is principal developer advocate at Netlify. With a passion for browser technologies, and the empowering properties of the Web, Phil loves seeking out ingenuity and simplicity, especially in places where overengineering is common. Phil's 20 year career in web development includes time as a software engineer at Verisign, an open source evangelist at British Telecom, and technology director at R/GA where he worked with clients around the world such as Nike, Google, Beats By Dre, and Samsung to bring engaging and effective experiences to the widest audience possible.

Acknowledgments

This report would not be possible without the contributions, review, and inspiration from the following people:

Chris Bach is cofounder of Netlify. Previously, he spent 14 years in the agency world, where he started Denmark's first hybrid production agency and won many international awards. He stands behind services like headlesscms.org, testmysite.io, and jamstack.org, and is one of the originators of the JAMstack term.

Todd Morey is creative director at Netlify, a passionate user of JAMstack technologies, and one of the early investors in the company. He cofounded The Rackspace Cloud and was creative director at the OpenStack Foundation.