My Journey Away from the JAMstack

The name is all but dead, nerfed by the company who invented it. Here's why Netlify was ahead of its time and where everything went wrong.

By Jared White

Before I give you my side of the story, I'd like to point you to Brian Rinaldi's comprehensive take on the demise of Jamstack (or as I still prefer to call it, JAMstack) for some much-needed context on what's been going down. He asks "is Jamstack officially finished?" and this article is essentially my reply.

TL;DR: the answer is ***yes***.

As for the reason why, we must point our finger straight at Netlify.

Listen, I get it. Running a successful and hopefully profitable hosting company with investors breathing down your neck is hard. I don't begrudge them for having to pivot to enterprise cloud mumbo-jumbo in order to reel in the big bucks and justify their valuation.

But I can't help but feel duped…like so much of the other "enshittification" we've been dealing with in tech over the last few years. **The cycle repeats itself**: we invest our hard-earned time and sometimes money to build on top of friendly, seemingly benign platforms—only to see those platforms wriggle out from under us and morph into something entirely different (and for our purposes, much worse).

Gather around folks, and listen to my story of my first experience with the JAMstack. I'll also explain why prior to this news I'd already moved on from the it and from Netlify, and what instead I believe the web dev industry should be heading towards as a "default" stack.

The Year Was 2015

I had just come off a lengthy stint trying to build and promote a paid, tablet-first CMS. With a failed startup behind me, as well as a number of WordPress sites I simply ***hated*** to administer because they were so buggy and insecure and expensive, I was getting desperate. Due to my experience as a Ruby on Rails developer, I even tried reaching for some Rails-based CMSes, but finding a slam-dunk improvement over WordPress was far from straightforward.

And then I stumbled upon Jekyll. 😍

To this day, I have no clue why it had initially taken me so long to discover Jekyll. Jekyll was integrated into GitHub (powering their Pages product) and was built with Ruby! I do remember hearing more and more about "static site generators" (aka SSGs) as I was winding down production of my own CMS, and I filed that thought away as a possible way to salvage some of the work I'd done.

Eventually I ***finally*** gave Jekyll a real try, and I was floored. Here was an amazing developer-friendly tool where I could just to take a bunch of simple HTML / Markdown, CSS, JavaScript, and image files, run a single command, and BOOM: get a website trivially easy to deploy anywhere. I even grokked the Liquid template syntax without issue, because I was already familiar with both Shopify and my own CMS which had used Liquid.

The only real head-scratcher was the content authoring side of the equation—I couldn't expect my clients to learn how to input Markdown into GitHub—but with my experience having already built authoring interfaces, I figured it wouldn't be hard to put together a simple Rails editor app that could work with Markdown and GitHub under the hood.

I dogfooded Jekyll first for my own personal website at jaredwhite.com, relaunching it in February 2016. (It's since been resigned many times and is now built with Bridgetown instead…but it remains a static site!) From there, I worked on a variety of projects for myself and for clients. To this very day, some of those sites are still on the web humming along without issue (here's one of my favorites) because, hey, ***static sites are awesome!***

Along Came ~~Aerobatic~~ Netlify

When I first got into this brand-new world of modern SSGs, the gold standard for hosting Ruby-powered web applications was Heroku. I was quite familiar with Heroku and had used it on a number of projects. But Heroku had nothing to offer me when it came to SSGs. Heroku was engineered around a model of dynamic web servers and databases, not build-once-and-cache-on-a-CDN-forever deployments.

I suppose I could have just used GitHub Pages, but at that time I was primarily using Bitbucket for hosting my projects and those of my clients. So it was perfect timing that, right when I needed to figure this all out, along came Aerobatic.

Aerobatic was basically Heroku but for static sites hosted on Bitbucket (it was literally an add-on for that platform). **Perfect!** I could easily get that "push via Git and automatically deploy" workflow going with no other setup required. And the deployed sites were fast, secure, and cheap as hell to operate indefinitely.

For my client's content editors, I usually just spun up a cheap VPS on Digital Ocean. They didn't need to be high-powered at all, because those servers weren't promoted to the public or accessed by more than one person really. And because all the content was stored in a Git repo, I didn't even need to wrestle with a database!

However, my love affair with Aerobatic didn't last long. Because another shiny offering soon emerged: Netlify.

The best way I can describe Netlify when I first started using it is "like Aerobatic…except better". I'd encountered a few technical difficulties getting Aerobatic sites up and running, and Netlify was a noticeable improvement. Builds were fast, deploys were rock-solid, and ***It. Just. Worked.*** Plus it supported both GitHub & Bitbucket, so either way I was golden. I forget when they added the Forms feature, but it was pretty early on and that also proved a huge advantage.

One issue many of us encountered when trying to market these amazing new "static site" to potential customers was the term ***static***. Calling a website "static" for many people implied a site which rarely changed and couldn't accommodate any dynamic, interactive functionality. ***Boring.*** For instance, surely you couldn't run your e-commerce site as a "static" site because e-commerce is anything but static!

Enter **JAMstack**.

The JAMstack is Here to Solve All Our Problems (Right?)

Netlify's marketing sleight of hand in inventing and promoting the JAMstack was sheer brilliance. Instead of calling these builds "static sites" and these tools "static site generators", we could say we're building JAMstack sites (to compete with LAMPstack sites I suppose) and using some hot new JAMstack frameworks. The JAM stood for:

- JavaScript
- APIs
- Markup

And lest anyone get confused (because as we'll soon discover EVERYONE eventually got very confused), the JavaScript of the J in JAM referred to ***client-side JS***, not server-side. The whole point of JAMstack was that the tool building out the markup etc. could be written in anything, and just as importantly the ***APIs used by the client-side JS could be written in anything***. After all, both Jekyll and Rails are Ruby-based tools, and I happily used both as part of my JAMstack deployments.

As time went on, a major appeal of the JAMstack was that it allowed a decoupling of the frontend from the backend, which is why Netlify and other hosts like it later on proved extremely popular with frontend developers. Ironically, in today's world where SSR (server-side rendering) and progressive enhancement is now top of mind for many web developers, it's positively ***wild*** to turn back the clock and realize that JAMstack architecture arose during the height of the **SPA** movement. You can even see it in all the marketing materials —your app "shell" could be statically deployed, and then your fancy-pants client-side app could take over and call APIs from all over the web. And a number of JAMstack sites were literally that. Disable JavaScript and what do you get? Maybe a simple header and footer if you're lucky. ***Everything else is blank.*** **Whoops!**

However, that was never ***my*** JAMstack. I had intuitively understood that the exciting promise of JAM was in a sense the reverse acronym of MAJ: Markup, ***then*** APIs, ***then*** JavaScript. In other words, build as much as you can with static HTML (via templates, Markdown, etc.), ***then*** identify what you might need for some dynamic server interactions—which maybe you'd just write yourself as a Rails app or whatever—***then*** write only the JavaScript you absolutely need to access those APIs (understanding that maybe certain dynamic pages would just get fetched directly from the server if need be).

In other words, **progressive enhancement**. 😂

But somehow that story got lost in the fray, and JAMstack eventually gave rise to a rebranded "Jamstack" with the major value prop being something rather entirely different: you could now build entire websites out of ***JavaScript libraries*** (aka React, or maybe Vue or Angular or Svelte) and ***JavaScript frameworks*** (aka Next.js, Gatsby, Nuxt, SvelteKit, etc.). And, whoa, look at this! You don't need servers ever again! You can just write ***serverless functions*** to go along with your frontends! Fullstack, server-first web development is dead, long live frontend + serverless!

(Coinciding with this sea change, Jekyll began a long, slow, painful decline into irrelevance, due to the inexplicable failure of GitHub's leadership to support its proper development and promotion as well as an unforgivable neglect of the

Pages platform. It remains one of my greatest frustrations in 25+ years of web development…so much so that I forked Jekyll in 2020 and created Bridgetown. But I digress…)

Along with this "second generation" Jamstack mindset shift came ***an order of magnitude*** more build complexity. Instead of a straightforward CLI kicking off simple transformations to go from Markdown -> HTML plus concatenate some (S)CSS files together or whatever, you'd get multi-minute long builds and GBs of `node_modules` and poorly-written tutorials on DEV.to about how to send emails from Gatsby functions and which distributed "web-scale" databases of the day are the coolest and crazy CLI tool churn and all sorts of other headaches. Things which used to take hours or days to accomplish in standard Rails or Laravel or Django apps—most of this stuff isn't rocket science, folks—now took weeks or months! ***Progress!*** 🤪

This quick march of slapdash B.S. web technology under the guise of making our lives easier was ***one hell of a whiplash***, and I really hadn't seen it coming when I first entered this space. Instead of JAMstack saving us all from the horrors of WordPress, we were crushed under the weight of Jamstack! Janky SPAs, countless immature buggy frameworks, and NPM ecosystem insanity—along with the multi-headed hydra that is modern React.

It got so bad I wrote about it. (Warning: if you think ***this*** article has taken on a dour tone, avoid reading that spicy take! 🌶️)

Netlify isn't to blame…except Netlify is to blame 😬

We can't put ***all*** the fault squarely on Netlify, in the sense that people mistakenly thought they needed to build their blogs with Gatsby and their business dashboards with Next.js because that's what all the "techfluencers" and VC-backed tool vendors told them.

Yet I **do** blame Netlify, because ***they're the people who invented the term JAMstack!*** Netlify proved more than happy to come along for the ride and oblige as one of the top hosting platforms of choice for this new ecosystem. They ***could*** have come out in favor of saner architectures and better support for languages other than JavaScript (believe me, I went around and around with them about their lack of interest in supporting Ruby-based server applications **even as their own platform used Ruby under the hood!**). They ***could*** have warned us of the dangers of complicated API spaghetti code and microservices. After all, why should Netlify care if your static site calls out to 20 different APIs or your own monolithic API you wrote in a battle-hardened, "boring" server framework? This bizarro-world focus on "serverless functions" and later "edge functions" never made a lick of sense to me (unless Netlify really thought they could somehow significantly profit off of function usage…again, a prospect which makes little sense to me).

Ultimately the failure of Jamstack to live up to the promise of its original JAMstack incarnation, and the industry's manic pendulum swing into unmaintainable architectures and vendor lock-in, led me to abandon Netlify as a hosting platform of choice and look instead to more reasonable options. At this moment in time, that choice for me is Render. Render gives you the best of all worlds: deploy a static site to a CDN, deploy a server API written in any framework you want—even Rails!—deploy a Docker container…anything you need, **BOOM**, done. Want to use PostgreSQL? ***Check.*** Need Redis? ***Check.*** Would you like simultaneous deploys of all these services at once? ***Check.***

I have no business arrangement with Render, and I assure you this isn't a sponsored post. I just really like their service. And if Render does become ***enshittified*** down the road, I'll be super bummed and look for yet another alternative. (I sure hope that won't become necessary!)

What saddens me is Netlify ***could*** have grown into a Render and meaningfully competed with Heroku—except they didn't. They took a different road, and I would argue **they failed**. Hey, hindsight is 20/20...but honestly this was ***so*** easy to predict. 🤷🏻‍♂️

I'm sad to see Jamstack die, but the thing is: ***most*** web applications deployed by most individuals and small teams only need modest server offerings and maybe a static site. Again, it's not rocket science. Most projects never need to become "web scale" and most web architectural complexity is completely unnecessary. You can spin up a Node.js API with Fastify, or a Ruby Roda API, or, heck, some PHP, stick it on a decent cloud server somewhere, and ***that's totally fine***. Boring technology is **great**. And servers are in fact totally awesome, as more and more developers are thankfully coming to discover (again).

The Legacy of Netlify

What Netlify gave us originally was a vision of how to deploy HTML-first websites easily via git commits and pushes, just like Heroku had done for dynamic applications. All we need now is a modern Netlify/Heroku mashup that's cheap, stable, and doesn't need to reinvent the damn wheel every year.

What do we call this, now that Jamstack is dead? I don't know.

I vote for KISSstack. (Keep It Simple, Silly.) 😋

But seriously, I think it's vitally important to remember that simple websites and more complex web applications all sit on a spectrum, and a good web host will be able to identify your individual needs and provision builds and runtimes accordingly—no matter what the particular service offerings might be. (I wrote about this too.)

From my vantage point, the ***only*** goal I care about is to make building & deploying sites dramatically easier for individuals and small teams. (Sorry Big Co.

Enterprises, I don't think about you at all.) And while it's a real shame that Netlify is no longer in a position to usher in this future for us, I'm optimistic we'll see Render, Fly.io, and other companies down the road pick up the slack.

Somebody has to.

Published on July 31, 2023 in #architecture #bestpractices

THE SPICY WEB is a project by **Jared White**.

Send me an **Email Message**

Follow @vanilla on **Mastodon**