SACRA

EXPERT INTERVIEW

UPDATED
03/03/2022

Jason Lengstorf, VP of Developer Experience at Netlify, on Jamstack's anti-monolith approach

TEAM

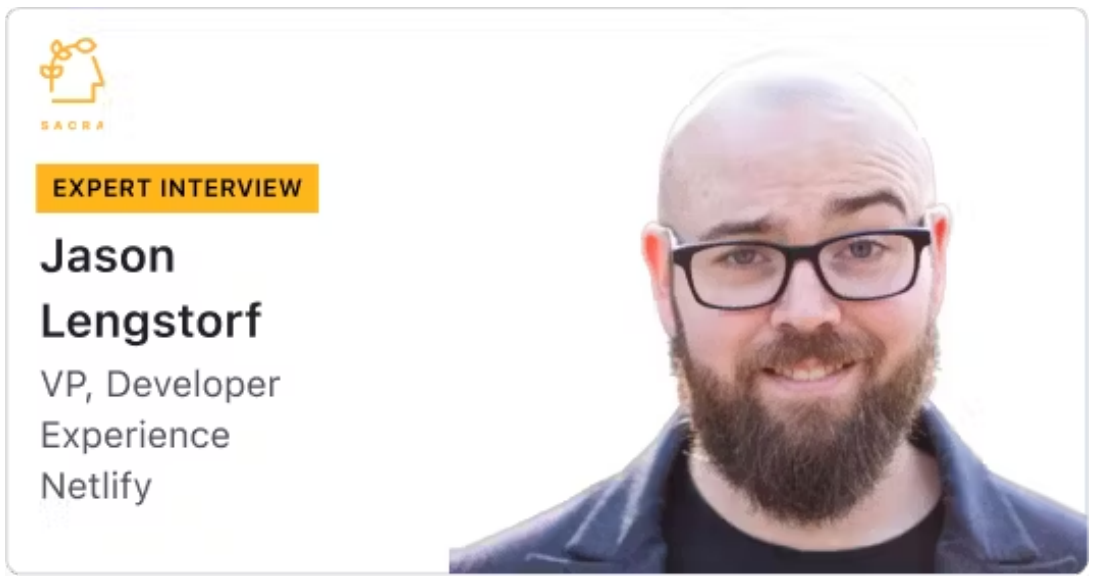
Jan-Erik Asplund
Co-Founder
jan@sacra.com

SACRA

Published on **Mar 03rd, 2022**

Jason Lengstorf, VP of Developer Experience at Netlify, on Jamstack's anti-monolith approach

By **Jan-Erik Asplund**



Background

Jason Lengstorf is the VP of Developer Experience at Netlify. We talked to him to learn more about two trends that are driving Netlify's massive growth: (1) the trend towards serverless and apps and websites that utilize Jamstack-style static content delivery, and (2) the rise of developer tools that abstract away the complexity of cloud services like AWS.

Interview

To start off with, I'd love to hear you define Jamstack and explain your personal path into the world.

The core of what we're trying to do with Jamstack is get away from the really cumbersome process of building for the web. What we saw previously is that web was an outgrowth of the rest of engineering and that meant that you were putting your web development processes through these very heavy back-end review queues of staging environments -- you got to set up

separate containers, you're waiting for a QA team and a DevOps team to do a blue-green deploy. All these things that are really critically important when you're doing a payment system rollout, but when you're making an update to a blog post, it just doesn't make sense.
What the Jamstack is is an answer to that pain that people are feeling. The way that we've bucketed it is that it's a decoupled frontend, so you take the frontend of your website and you develop it completely independently of the other code in your product. It'll only communicate to those other code bases through APIs. You pre-compile as much of that as you can, so you have a build step that'll pull in data that doesn't change between page loads and compile that into static assets instead of trying to load it on every page request. This helps increase your cache ability, lowers your overhead, reduces risk and improves security. There's a lot of good reasons to do that. Then you push that out to the edge on a CDN and using various other methods of delivering that content as close as you can to your end users.

This is the default that we want people to start adopting for the web. It's what we would consider to be the workflow for the modern web. As you extend that with more dynamic use cases, you can still use that Jamstack base of “you're building a decoupled front end that's as pre-compiled as it can be ahead of time, and then deployed to a CDN.” You can still do additional things on top of that, like add serverless funnel to the Jamstack definition. What we're trying to do is get people to change where they're starting from. Instead of starting from “you're on a server, you've got all these extra requirements and pains for how you get something live,” you start from “I can make a change in GitHub, I push it to the repo, everything automatically builds and then it's live.” So we have five minutes from “I identified a bug in production” to “that change is pushed live” because of the default architecture, and then we extend that.

When the Jamstack movement started, a lot of what was created with it was personal blogs, portfolio sites -- simpler kinds of projects -- and over time, and especially today, it's evolved towards more dynamic applications. I'd love to hear your take on where it is today in terms of complexity. What can be built on Jamstack, and how do you see that evolving in the future?

To answer that, I should probably answer the other half of your first question, which is my path to the Jamstack. When I was working at IBM, we were on a more monolithic stack. In order to develop the frontend, your first day as a developer you would get your development machine, and then you would have to configure a Docker container and NGINX and a reverse proxy and all of these pieces so that you could just run the code locally. Then to deploy was a huge set of hoops. We deployed every Thursday, I think. Maybe every two weeks on Thursday, we would get something out to production. It was weeks to get something live, instead of minutes or hours or days, and that was really painful.

On top of that, because of this slower deploy process, we had trouble iterating. That meant that instead of trying something new -- like removing old code and replacing it -- we were doing this layering. We had these geological layers of app that we just bolted more stuff on top of, because we weren't quite sure how to handle regressions. If we made a mistake, the rollbacks were a huge pain, and it took so long to get something live that we were like, “You know what? I'd rather not deal with that. Let's just add something on top that solves our problem.”

We wanted to fix that. We wanted to improve that because, in addition to just the complexity of building, we were having huge slowdowns. You'd go to certain areas of the app, and you would load Django, which would load jQuery, which would load React, which would load another version of jQuery, and then you'd finally get your app. You're looking at tens or maybe forty seconds to load certain dashboards that were really data heavy. We were like, “This is unacceptable. We can't do this.”

So we started looking at better solutions. I didn't know that the Jamstack existed at the time. But we decoupled that frontend, we started building a new React app that laid on top of the old system instead of being a part of it, and we only spoke to the old system through APIs. That allowed us to get rid of all of that frontend cruft, and we rebuilt whole dashboards in our app using this approach.

That was where it really became clear to me, because we saw a 3000% increase in the performance of that dashboard page because we were able to eliminate so much of that legacy code, and we saw that our deploy times -- we got things through our approval queue and into the deployment queue in

an hour. They still had to wait to go through the formal deployment process, because we hadn't changed the way that we deployed the frontend, but suddenly our iteration time went down to borderline zero. By IBM standards, it was zero, statistically speaking. We got really excited about this, and I started understanding the potential for what this could mean for teams. So when I left, I was actually surprised that people only thought the Jamstack was useful for personal blogs, because my first introduction to it was rebuilding an app at IBM.

Fast forward to today and what we're starting to see is that more and more companies are starting to embrace this idea that the Jamstack is a pattern you can use. We're seeing third party tools, like MongoDB has a headless option, WordPress has a headless option, most of the CMSs that are launching today are headless by default. We are seeing tons of API-as-a-service platforms stand up, and those are just completely built for the Jamstack approach. You build a frontend, you speak to the APIs with your frontend, and you deploy the frontend separately. That's a Jamstack site. We've seen everything. People are building ecommerce stores. People are building really impressive things, like the Spring team is building their sites as Jamstack sites -- these are thousands of user sites being deployed from their advanced setup. We've even seen things like the Twilio team replace their console, so the Twilio console now runs on Netlify as a Jamstack site.

You can really take the complexity of this to “anything that you're building for the web, you can build on Jamstack,” because when you extend that default, you've got all the options of what a traditional single page app built with JavaScript would afford you. A lot of these teams were just using a container to deploy a single page app. Put it on Jamstack, and now your single page app deploys in half the time. So I would say the capabilities here are limited to what you can build on the web.

Let’s talk about what you do at Netlify, which is working on developer experience. It's something that people talk about a lot. What’s your understanding of what has happened in the last few years that has made developer experience such a big thing, and why is Jamstack so key to it, besides maybe faster deploys?

The biggest thing that I noticed is that we saw a shift from companies using the web as an augmentation to their business strategy to -- especially when 2020 rolled around -- the web being the driving force of their strategy. With every company, we saw an explosion of online retail in 2020 because everybody had to go online because we couldn't go inside anymore.

I think that shift means that just about every company now is either hiring full-time web developers or working with web development contractors, and in all of those cases, the thing that they're building is a direct result of the experience that those developers have. If you've got developers using tools that they're excited about, that are easy, that provide them with the right level of abstraction where they're focusing on building features instead of on dealing with the overhead and boilerplate and the kind of meta work of keeping the lights on for a project, you are able to ship features faster, and you have lower turnover, because your devs aren't frustrated by process. They're not frustrated by “Oh, if I want to move to a new computer, I got to do three hours of setup work to get this app running” or something like that.

I think that the drive to online really helped the whole world refocus on the fact that developers are driving commerce. They're driving the economy right now at every level. Even hotels, their whole experience now is through apps and using your phone to open your hotel room door. That's all built by developers. So you want your developers to have good tools, because it's your hiring strategy, it's your retention strategy, it's your iterative strategy, it's the way that you lower your go-to-market time. All those things come directly from providing your developers with tools that they're excited about, that they can use easily, and that set up good defaults so that, if your developers take every shortcut in the world, the user experience should still be good. That's what we're trying to set up with the Jamstack. As an ecosystem, how do we make it so that, if a developer shows up and takes every shortcut available to them, they're still going to make a high-performance site that's maintainable into the future and that doesn't take forever to deploy?

One idea we are noodling with is that Netlify and to an extent Vercel are like modern versions of Heroku, being a wrapper around a bunch of other tools and services that

make it super easy to get started no matter who you are as a developer. Does that analogy resonate with you? Do you think that there are things that Netlify is doing better than Heroku did, or if there are any lessons to be learned from Heroku as a phenomenon?
Heroku is absolutely an inspiration for Netlify and Vercel and Fly.io and all these other companies. They're kind of “how do we take a thing that everybody has to do and make it less painful.” What I've found interesting about the Heroku model is they looked at something that was causing somebody pain, and they looked at not the end user, which is I think where a lot of companies will default to. It's like, “Well, what is our customers’ experience? How do we make their pain less?”

They instead looked at the middle. They said, “Well, what's causing that customer pain? Customers are frustrated because we don't roll out updates. Customers are frustrated because the app is slow. Okay, well, why is the app slow? Why does it take us so long to get updates out? Well, it turns out our developers are burning a bunch of time on this configuration and boilerplate and setup, and if we eliminate that, we downstream get these great customer benefits.” Heroku looked at the middle of the stack and said, “We can get rid of container config and wiring up these infrastructure services and auto provisioning things.” I think they did a great job for the era that they came up in. As we've iterated as an industry, we all learned from Heroku, and every company is now focused a little bit more on, “Can we get a Heroku-like experience?”

Netlify came in and said, “Okay, now that all these systems are even easier to spin up and deploy and designed for this idea of being provisioned and configured remotely and being like a base for a layer of abstraction instead of needing to be right up in the developer's face, can we take that a step further? Can we iterate on this idea of making the problems that developers have into a clean set of defaults, so that developers don't have to waste any cognitive energy or time on the things that don't change?” When you have a website, you need that website to go up on the internet. You need a way to roll back. You need a way to do versions. You need a way to share a preview with somebody. All those things need to be true in a high performing team. So let's just make all that stuff a default. It just happens when you use Netlify. Now developers are spending even less time worrying about the configuration and

the boilerplate and the setup and the maintenance. They can just focus on features, and all that rolls downstream to customer benefits. That's where I would say we picked up the baton and ran with it, building on the foundation that companies like Heroku laid.

So everybody moved towards trying to have a Heroku-like experience, trying to abstract away certain things that they figured out developers didn't want to have to manually configure every time.

Exactly. It wasn't really that long ago in the grand scheme of things that everybody had a room full of servers in their office. Those are now commoditized. You don't really need to maintain your own server. The only reason to do it is if you're at such a scale that the trade-off between paying full-time people to maintain your server racks dips below the markup that you're paying to a cloud hosting company, or if you have to be at such a high scale or you have really, really edge case requirements. The vast majority of companies just aren't there. So yeah, use the commoditized thing. Let's spin up GCP or Azure or AWS and let all of that infra go away.

If that's the case, then we can build layers on top of it, because everybody was going to use it anyway and take away your need to have to configure that stuff. Kubernetes is kind of like that. You build companies on top of Kubernetes, and then people don't even need to know that they're using containers or Kubernetes or any of that stuff. They just say, "Well, I need a server." In Netlify's case, we saw that for websites so much of that stack has been completely commoditized. We know we need a CDN. We know we need good DNS. We know we need an efficient way to deliver things. We know we need the ability to recover if something goes wrong. All those things are going to be true. So you can build it yourself or you can pay a little bit of money to Netlify, and in most cases, you'll pay significantly less to Netlify than you would pay to a developer on your staff to solve that same problem.

In the last few years, AWS has put out stuff that's vaguely along the same lines as Vercel and Netlify, like Amplify and App Runner. What's your take on those, and what would Netlify's positioning be vis-à-vis that option?

I think what everybody's trying to do right now is find the right way to onboard people into a new way of working. Depending

on what the company's doing, their incentives are different. In AWS's case, their incentive is to build an abstraction on top of AWS technology so that people who are in the AWS ecosystem have a quick on-ramp into using their tools. Amplify is that. You have a preconfigured bundle of AWS services that, if you go through that on-ramp, you have a really good experience. I mean, it's great. They've done a really good job. What I've found is that there's a little bit of a cliff where you'll exceed what Amplify was designed to do and then find yourself in the pit of despair where you've now provisioned a bunch of AWS services, Amplify can't help you solve the case that you need because you've gone off the beaten path, and you've just been hurled into the sea of services that were configured by a robot that you've never seen before and you have to figure out how to back that out and work through it.
What draws me to Netlify and services in the Netlify space is that we're not trying to say, “We'll push the buttons for you, and then when you're ready, they're your buttons.” We're saying, “Let's not think about those buttons” -- the same way that the vast majority of us aren't thinking about the fact that, when we push a button on our keyboard, an electrical impulse is being sent through a wire to the motherboard and that puts a character on the screen. We don't have to think about that. We just push the keys and we type to our friends and we're searching things on the internet.

We want infrastructure for modern websites to feel the same way. We want it to just disappear as an implementation detail, because you don't need to care that you are using containers or an S3 bucket versus an Azure web storage instance or whatever that is. You should care that you've got files accessible in a performant way. You should care that you've got access to things like serverless functions and not necessarily how those serverless functions are configured or routed or scaled, but just that they work, and we'll scale up if you need more and if you don't want to scale up, you can set the limits in your account -- all those things that you want to do to make sure that your website performs as it should. But we don't want those implementation details to be a thing that you ever really need to worry about.

That's a design choice that we make. The trade-off is that, on AWS, they will happily send you off into the whole AWS ecosystem and say, “Yeah, whatever you can figure out, you can build.” With Netlify, what we say is, “Well, you've exceeded

what Netlify can do built-in. Here's a way that you can go get those other services and wire them into Netlify." But we're not going to take the Netlify wrapper off and say, "Here, go do whatever you want," because in our opinion, that's not a productive way to solve that problem.

It seems to me that the big point is just how big and complex AWS has become, to where it's actually prohibitive to say, "Okay, we'll take off the training wheels. You can build whatever you can figure out." Have we reached a point where that's too big of an ask for the majority of developers?

It's a sign of AWS's success that they've been able to launch so many services, but I also think that it has shown itself as a real challenge for them. You've got companies that make their entire living off of just getting paid to sort out Amazon bills. Corey Quinn from the Duckbill Group -- their whole business is they will come in and look at your Amazon bill and help you understand where you spent money and why. When a service hits that level of complexity, it's clear you can build anything you want on AWS, but it's also clear that you can accidentally shoot yourself in the foot.

You'll see a lot of really heartbreaking stories of students who are trying to learn, and they accidentally run up a $40,000 AWS bill because they turned the service on as part of a tutorial and didn't realize how expensive it was and let it run. Those sorts of things are really hard, because that it's real money when you run that service. It's not like Amazon's just charging you for nothing. They spent all that money, and they're just charging you back. It's a very challenging place to be when it really is like going in with no safety net at all. You're in these services, and if you don't get what you're doing, you can really get yourself into a pickle.

I think that we're looking for the next layer of abstraction. How do we make cloud services safer? How do we let a web dev who has just joined a team push to production with no risk that you're going to accidentally run up a $40,000 bill if you don't pay attention to your AWS account for two weeks?

I've noticed Netlify is a sponsor of various frameworks and other open source projects. I'd love to hear from you about why that is important and what that does for Netlify, for Netlify users and for the ecosystem as a whole.

Our intention with sponsoring projects is we really want to be good citizens in the open source ecosystem. Open source moves everything forward. If we didn't have all the people who volunteer their time to work on these projects, we would be just so far behind where we are now. We want to make sure that Netlify is not taking advantage of that labor, and we want to set a good example by trying to pay back into the systems that we're benefiting from. So for us, it's really important that we reinvest into the open source ecosystem and give back to these projects because Netlify's strength is its ecosystem.

The breadth and depth of expertise in the Jamstack ecosystem -- that modern web ecosystem -- is unbelievable, and what makes Netlify powerful is that you can take any one of those pieces and deploy it to Netlify. What we're trying to do is align incentives. We want to invest in open source frameworks so that they are sustainable, because for us, competition is good. We want more frameworks, more options in the ecosystem, because that's going to push the whole web forward. We want to be close partners with these frameworks so that they are interested in trying to make things work really well on Netlify, because we want to have the best deployment experience. We want every framework out there to work really well on Netlify, so we're trying to align ourselves as closely as we can there.

The other piece of it, too, is we believe that, as teams mature and grow, it's significantly more challenging to standardize on one stack in a big company. The idea of trying to go all in on just Angular or just Next, or just whatever, you name it -- you're going to find yourself with a lot of cases where you're either using way too much machine for the thing you're trying to build, or you're going to have rogue cells inside of your company that are going to say, “Well, we don't want to use that main framework. We're going to use this other one.” If you've built your entire infrastructure around one stack, you are then setting these other teams up to fail. You're setting yourself up to be really fragile when the ecosystem shifts in the future. So Netlify is looking at -- to borrow a phrase from Nassim Nicholas Taleb -- being anti-fragile with the way that they've set up their stack.

For us, that broad experience and broad compatibility with the ecosystem is the key to that. We will evolve with you as you go through whatever iteration. Your head architect quits, the next one who comes in wants to change the whole stack -- fine. You

don't have to rebuild your whole infrastructure as well. We've already got you covered. That's I would say our biggest differentiator and the thing that we're most excited about is: the fact that, as developers continue on the maturity curve and want to use other tools, they can just do that. We're not saying, “Well, you should use this because this is the one that we built and it's the best.” We're instead saying, “Yeah, use whatever you want. We work with all these frameworks, and we're super invested in their success. So please bring all of your projects to us. We want them to work.”

You were saying that at IBM, the Jamstack approach emerged in the shadow of this larger monolithic system. Within huge companies like IBM, where do you see potentially Jamstack-like approaches emerging most readily internally today? Are there still places at these companies where it's going to be hard to make that work?

At a certain level, it all becomes a political discussion, so it depends on how open the people are in the decision making spaces. But there are what I've seen to be really common inroads if a company is trying to adopt this. Typically modern docs teams are developing a kind of docs-as-code approach, where they're writing markdown and that allows for really quick review and a history because they can do it all in GitHub. If you're writing in markdown, the Jamstack is a really natural companion, because all the build tools for markdown are Jamstack-generated. You can deploy your docs site as a really quick way to experiment with this and see how quickly you can get things live and how collaborative and productive people are. Then marketing sites are another good target for experimentation. If you want to stand up a landing page, you need a lot of iteration. You got to switch out advertisements every day. You got to change eyebrow banners or do personalization stuff. There's a lot of options for quickly deploying these things and getting them up and iterating fast.

What I've also noticed is that a lot of teams are using it as an experiment. On their hack day, they'll rebuild a page of their dashboard using this Jamstack approach and they'll be able to get a feature-complete page of their dashboard in one hack day. That to me is mind blowing. That's actually what happened at IBM. We did an internal, like, “Okay, let's spend two days on this project, see what we can build.” And we reimplemented the whole thing and got it shipped to a staging branch, and then we went and showed the rest of the team.

They were like, “How did you do this? When did you have time to do this?” When we explained what happened, they were like, “Okay, we got to figure out how to make this work,” at which point, honestly, the rest of my job at IBM was educating other teams internally on how to make this switch.
I'll be honest. I got a lot of pushback because people know what they know. When you go to a DevOps person who is comfortable with the back-end way of deploying, you have to present it in a way that's saying, “I'm not trying to change everything you're doing. I'm not trying to break your flow. I'm trying to take things off your plate. I'm trying to improve the security and decrease the amount of release engineering that you have to do and the just general busywork that you're doing, so that you can focus on the stuff that's important, like the security of our back-end systems and those sorts of things that are way more critical than whether or not we can update the home page.”

That helps me understand the strategy of focusing on developer experience as a way of encouraging generational change in organizations, so people coming into organizations think about this as a way to build websites.

I think that goes back to what you were saying about why developer experience is so important now, why people are focusing on it. We're seeing more and more changes driven by developers. They'll go to their managers and to their VPs and they'll say, “Look, we want these tools.” It's hard for someone when everybody on your team is saying, “We'll be faster if you just let us have this tool.” It's really hard not to use it. For most companies, it's a no brainer. Swipe the credit card, get them the tool they need, and let them be fast.

If you're a company trying to sell, and you work with companies that do development, getting that developer experience so good that the developers advocate for it within the company -- that’s your best sales motion. You talk about product-led growth or companies that go that way, your best opportunity is to get such loyal fans of the way that you do business that they're going to go fight for you internally. Then you don't need to have cold sales calls. You have warm inbound leads because everybody's going, “Our developers are screaming for this. They want it. How do we get this set up?”

I'd love to talk about APIs as the “a” in Jamstack. Obviously, you and Netlify can’t endorse any specific APIs, but to what degree have you seen the emergence of developer preferences for specific ones in different areas? Some developers we’ve talked to will use whatever the client wants, but a few say they’ll always use this for security authentication or that for search.

I think everybody's got their favorite flavors, so you see loyalism in that sense. “I like what I like because it's the thing I know, and I'm fast with it, and I can get a lot done.”

What I'm seeing is a strong inclination to move toward companies that focus on their APIs as their primary layer of interaction. When they're shipping it like that, you can see that, say, Sanity -- the CMS -- has a huge focus on the developers who use Sanity, and you can see that in the way that people advocate for it. Developers love working with Sanity. Contentful is another one. They have a huge focus on a GraphQL API, and they've got great docs for getting you started, and that is a big winner. You can see why those companies are dominating in the market and why they're so popular. Stripe is another great example. They took something that was easily the most intimidating thing about working on the web, which was taking payments. They focused on making a really developer-friendly API, and suddenly developers are like, “Oh, of course I'm going to use Stripe.” They absolutely cornered that market.

I think it's less about, “I'm going to use this because it's my favorite” or “I must use this,” and it's more that when a company really nails that experience of “we built this to solve your problems and get everything else out of the way,” you can see developers respond to that.

Are there disadvantages to the headless API approach? One thing we've heard is that non-technical marketers don't always love working in, say, Prismic, which can be a little more annoying than WordPress if you are totally new to it.

Let me answer this directly, and then I'll do a more indirect answer as well.

Directly, the biggest trade-off is that, for CMSs that were built as API-first and aren't focused on non-developer use cases,

you lose some things that you might be used to if you're, say, a WordPress developer. You don't get instant previews, you don't get some of these more “table stakes” things for CMSs that a marketer would be looking for. They'll be like, “Where is my ability to look at this before I publish it? Where is my ability to share this with somebody to get feedback?” Those sorts of things do exist, and they can always be built, but with certain tools, you find yourself having to implement your own.
Now the more roundabout answer here is that I think what we saw in a pre-Jamstack world was a constant tension between the marketing team and the development team. Depending on who was higher up in the food chain that made the decisions, you would see a tool get optimized for whichever team they aligned most with. So if the CMO was making the decision and they came from a content marketing background, you're going to see somebody in Sitecore or in Adobe Experience Manager -- one of these really CMS-focused things, but when the developers get into it, they're just sad. They're frustrated all the time. They have a hard time working with it. There's a lot of hoops to jump through. Then vice versa, you would see companies that say, “Well, we're run by developers, so we're going to build a developer experience,” and the marketers get thrown into GitHub repos and into working with APIs directly. You've got marketers trying to write SQL queries to pull things that they need, and it's like, “This is maybe not the best way to solve this problem.”

What I like about the Jamstack approach is that we're making it less of that trade. You can go use WordPress as your headless CMS, so the developers get an API-based “do whatever you want, build the way you want to build,” but the marketers still get that really good WordPress admin situation. You get to pick and choose your tools to best match what you're doing.

You also lose this other trade-off that I think is really frustrating for people, where you've got WordPress, which is a really good CMS, and you've got Shopify, which is really good for ecommerce, but you don't really want to use Shopify's blogging tools and you don't really want to use WordPress's ecommerce tools, because they're both kind of bolted on to the main core feature. In a headless world, you can just combine the two. You have a store page that pulls from Shopify APIs, you have a blog page that pulls from WordPress, and everybody's happy. Everybody gets that really good admin experience on the CMS side for both Shopify and WordPress, and the developers get

APIs with data, and they get to build with whatever frontend in a really consistent and predictable way, despite pulling from two completely different systems. I think that's really beneficial and helpful.

What kinds of things would need to happen for headless WordPress to be the primary way that WordPress is delivered? Or headless Shopify? I assume that for the average mom-and-pop that went online because of COVID, it might be too big of an ask to set up headless Shopify. What's the roadmap -- if there is one -- for that becoming more popular?

I think the thing that we're missing is the companies that commoditize it. With WordPress you had Automattic the company, and then you had ecosystem companies like WooThemes or Envato. They built these huge marketplaces around plug and play WordPress plugins and themes and add-ons and all these different features. I could go shopping and say, "Yeah, I want a forum. I want an E-commerce store. I want moderator controls. I want better SEO." I click all these buttons. My site gets auto-configured, and I've never looked at code.

This model is clearly what we're heading toward. We're not seeing new companies ship monolithic CMSs, or at least not very many. Even WordPress would prefer that we didn't use WordPress as an API, I think, at least from the outward messaging I've seen. They've built a really good API. They've got really good tools around how to use it in headless mode, because I think the writing is on the wall. Developers are making the choices now, and they don't want to work with a lot of stitch-together monolithic CMSs. They want to work with APIs.

My assumption is that, now that we've seen that, the market is aware of that. We're going to see people start to target the commoditization of headless site building. We can see this in a few places. There are some companies that are standing up. There's Vue Storefront, which is like, "Hey, you want a Jamstack-friendly ecommerce store? Click a few buttons and you get that." We've got agencies that are really specialized in this, for the more corporate companies that want to make the switch. We've got more of those plug and play tools starting to show up in companies like Spring, which is very similar to Shopify in that you show up and just say "I want a store," and it

just spins up. That's built as a Jamstack site. They're moving that way.
My suspicion is that we'll see that, as the ROI of using the Jamstack starts to become more apparent -- and we've got that ROI report that just came out with Forrester that shows a huge return on investment for switching to the Jamstack -- and as those results get repeated and it becomes more commonplace, we'll see even the people who don't really want to do it start defaulting to headless, because it'll cost them less to run their services. It increases their margin.

Say you were to build a headless ecommerce that does use Shopify as a backend but is built via Jamstack. The other big advantage is that you're not limited in any way to the Shopify ecosystem for what you want to do with marketing, page experience, card abandonment, whatever. Basically you're opening it up such that you can use any API in the Jamstack universe versus just what's on Shopify. Is that fair to say?

I think what's exciting about that is it's setting up a world where we can build without walled gardens. Something that we're feeling the pain from right now, if you look at recent lawsuits and companies complaining about it, is like the Apple App Store. We've basically forced an entire generation of companies to agree to pay 30% of everything they make, because there's no option. You can't sell things on an iPhone or an Apple device without paying that toll. I think that people are frustrated by that.

The Jamstack prevents a single, large company from completely building a walled garden. They're going to offer great services, and people are going to be willing to pay a markup to get those services. It prevents what I think happened on Apple, where there are things that we actively dislike and we still have to pay a 30% markup to use them, because there's literally no option. I think this is better for competition. It's better for innovation, because companies are incentivized to build great things. We're going to pay the markup to them for using their ecosystem when the quality is there. When they don't deliver on the promises, or when they don't have something that we need, we're not stuck. We can reach somewhere else and use the thing that meets our needs. That's going to push competitors in the space to improve their offerings.

It's the right kind of growth. It's the right kind of competitive pressure that's going to continually improve this marketplace. To me that's probably one of the most exciting things. As developers, we win by using this open approach versus getting into the monolithic approaches where we're stuck with whatever the providers give to us, because there's no pressure on them to improve that. We can't go anywhere else.

Something that comes up in talking about Jamstack is this idea that there's a lot of cyclical things in software development. Like you said, Envato and all these plugin marketplaces and themes recurring, but in a very updated, modern way.

There's a quote that I've been referring back to a lot as I have these conversations, because it definitely does on the surface feel like we're walking in circles a little bit. It's a Herman Hesse quote from the book Siddhartha: "We are not going in circles, we're going upwards. The path is a spiral; we've already climbed many steps."

I think we're going around, but what we did with WordPress fifteen years ago is what was possible fifteen years ago. We hit the limitations, and then we explored a different avenue. We hit the limitations of that, and now we're saying, "Okay, so now we want to do all the things that were good about WordPress, but the industry has advanced since that era. So we can do more, but let's take the good ideas and build them back in here." We loop back around, but with all the good things that we've learned along the way. We move between a set of concepts, but each time that concept that we're revisiting is aided by all of the context that comes from having lived through the last iteration of that concept.

I would agree we're moving back toward marketplaces and all those things, but my hope is that they're more portable. We're not going to see, say, Sanity themes. I don't think that's the way this is going to work. I think we're going to see blog themes. You can swap out your headless CMS. You can swap out your ecommerce provider. All of those things are going to be set up in a way where you, as the consumer, have the ability to choose which tools you want to work with for your dashboard and for your management. Because it's all built on APIs, the theme developers can build adapters: “Here's what a Shopify product looks like; I'm going to convert that into a

generic product data structure. Here's what a big commerce product looks like; we'll put into the same generic data structure. Now my themes can be built with any product from any backend.” All we have to do is this little bit of data munging to get it into the standard product format, which all these tools are going to provide if they're ecommerce.
That's the really exciting innovation that we're headed toward: this idea of not just a marketplace for one tool, but a marketplace for all web development. That’s opened up because of this API-first approach.

Disclaimers