SACRA

EXPERT INTERVIEW

UPDATED
03/03/2022

Jamstack agency founder on the rise of Next.js and Vercel

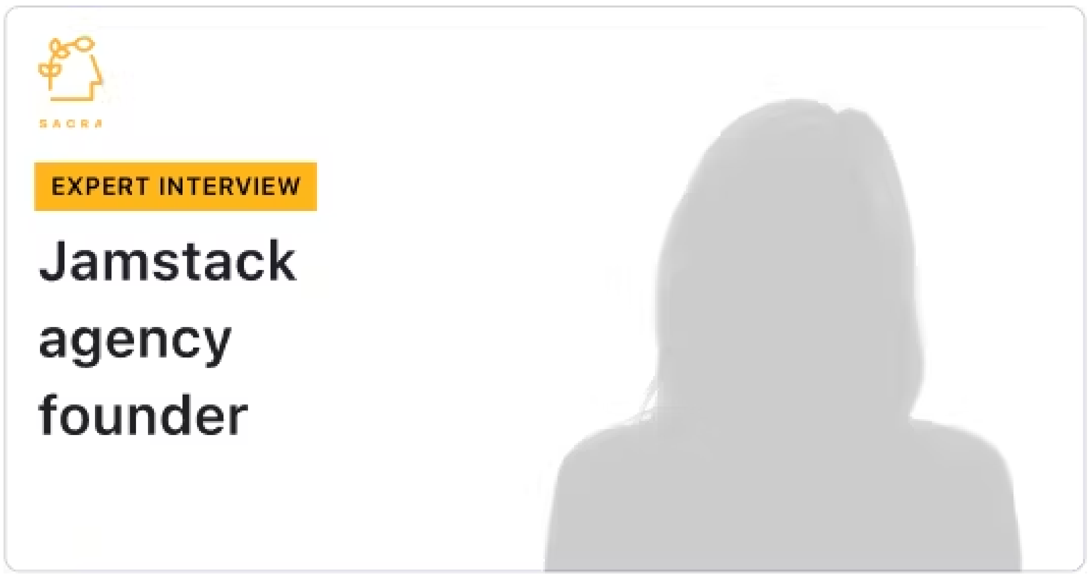
TEAM

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

Published on **Mar 03rd, 2022**

Jamstack agency founder on the rise of Next.js and Vercel

By **Jan-Erik Asplund**



Background

We interviewed the founder of a Jamstack agency to learn more about how writing Javascript has changed over the years, why he won't start projects in anything but Vercel, and why Next.js has changed the face of front-end development for the foreseeable future.

Interview

Let's start with some personal context on you as a developer and how you use Jamstack.

For me, Jamstack really comes from a focus on working with Next.js and Vercel. I didn't necessarily pursue serverless or certainly Jamstack independently. I think Next.js and Vercel both morphed into that sort of architecture, and I was just along for the ride. I wasn't necessarily designing my apps Jamstack when I was deploying to ZEIT before they became Vercel. They were more of a Docker deployment service, but then they pivoted and went all in on serverless. Sometimes they might mention Jamstack, but Vercel doesn't really name it

that -- it's just that's what it is. So I made that transition and here I am, and now pretty much if I'm going to start a new project, it's going to be with Next.js virtually always.
I'd love to get your take on Next.js and how it's different from vanilla React. Why would you always start with Next.js, and how has that led into using ZEIT/Vercel?

Actually, I started deploying to Vercel before I started using Next.js, now I think about it. Next.js previously was interesting, but it was not really what I needed at the time, because they were server-side rendering focused.

Then -- I want to say two years ago or so, maybe it was three -- they kind of shifted their focus and moved towards really a different programming model altogether. Their primary focus from my perspective seems like it's static generation. They started adding these APIs to support the set of different deployment strategies on a per page basis. When I saw that I would have all the options covered at my fingertips, then at that point, it became a no brainer. If I'm building with Create React App, I really have just static deployment options. With Next, as soon as they made that shift, I could more or less -- once I got my bearings -- have an almost identical programming model as I was working with before, but anytime I wanted to do some things that involved different types of deployment, it was right there, just a quick little integration.

Talking about today, you said you're on Vercel. What makes that super sticky so that you keep coming back versus going to Netlify or even deploying on AWS or something else?

I think the distinction is probably most pronounced between AWS versus Vercel. Netlify versus Vercel is more of a similar comparison. Between those two, I think it really just boils down to taste, slightly different workflows, slightly different APIs.

For me, I guess it makes sense. My sense was that Netlify was initially focused almost exclusively on “drop your folder to deploy a static site.” Their functions offerings really took focus only much later in their arc, whereas I think Vercel was more a full stack way of approaching projects from the very beginning. Even when they were doing it in a way that wasn't Jamstack, but they were thinking about servers, right? They were also thinking about static sites. It always felt like it was a little bit

more first class, and the way that they designed everything in their system just fit my mental model.
Compared to AWS, the way I think about it is that Vercel -- now that I understand how it works under the hood -- is I'd like to think more or less what I would build myself if I was going to deploy to AWS. It's basically just packaging up API routes and deploying them as lambdas. There are other tools that you could use to help with that, or you could just roll your own thing, but it's a lot of work. Obviously, they have a good size team of experts spending all their time working on that and continuing to improve it. So that trade-off is a huge win in terms of just being able to outsource all of that stuff that I'm less interested in. I think it's great, and I'd love to work on it, but I just don't have time because I want to be focused on what actually differentiates my products.

Have you seen or can you intuit a point at which there might be performance reasons to switch away from Vercel? Could that plausibly happen, or would it be at such a scale that it's just not worth thinking about for you?

It's not something that I'm finding myself hitting on the projects I'm working on, which are typically greenfield kind of prototypes, early-stage startup types of projects. I think you have to get quite a ways into really hard types of problems -- massive usage I guess -- to where you would hit some of the points where you would say, “We need to have more control.” There are some performance questions, but I think it's more about control. Vercel and Next.js aim to provide this very minimal API surface, and that's great except when you need something that's not easily available for you to tweak. At that point, you might be with a big team looking to implement something like Vercel, but with all of the little configurations and slightly different tweaks that you need specifically for your business.

Performance-wise there are certainly questions around, if you're taking a static generation approach, you can hit limits around how long it takes to build a project. Next.js and Vercel have options that they've exposed that help give you tools to work with that. But I saw an article that was just written this week by a team -- I think at Plex -- who wrote quite a bit about how they've transitioned into this deployment model, and they're operating with hundreds of thousands of statically

generated pages, and they're starting to hit some of the limits. So you definitely can.
Then folks will also look at potentially some latency concerns around, where is your API actually hosted? How close is that to where your database is? You're starting to see a lot of interesting work being done even from Vercel around moving different pieces to the edge so that you can address some of those concerns. So if you have extremely sensitive latency concerns, it's possible that this architecture won't meet your goals. Maybe it will in a year or two from now, if you can do something around edge functions. But for your straightforward needs, Vercel's dashboard is the classic example of, when folks initially thought of this model as just for static sites, Vercel wanted to make the point that, "No, look, we have an interactive dashboard that's personalized for each user, that's running with this architecture." So the latency requirements aren't super demanding. It's not anything like waiting seconds and seconds for your dashboard page to show up. You're talking about milliseconds of difference where you would hit those types of performance concerns potentially.

You mentioned the fact that Vercel has differentiated by not being dedicated solely to frontend and by having this backend concept. We've talked to developers who have worked on projects where they might use Vercel for just the frontend but Heroku on the backend. Is that something you have done or would do, or do you find you can typically do it all in Vercel?

From my perspective, this is something where maybe a lot of folks just don't actually realize what the options are with Vercel and with Next.js. Now, there certainly are scenarios where a team has expertise with another stack on the backend, and maybe they want to stand up a Rails API, or whatever technology they're working with that can't be deployed reasonably to Vercel. That's where it might make sense to leverage the investment you have with your team and with your expertise on the backend to deploy somewhere else.

But for me, it's one of the big advantages to working in this ecosystem that my API can be right there in the same project, sitting right next to my frontend components, and the context switch now, going back and forth, is almost nonexistent. The same sort of transpiling and syntax is available in both places, so it's seamless.

Unless you have some particular reason why you would want to move your backend outside of that paradigm, then I think you get a lot of benefits in terms of just eliminating all of the infrastructure that's involved with running a service like that, where your backend now becomes just static files that are automatically packaged up and deployed to lambdas by Vercel. From my point of view, I'm certainly always using API routes as my primary backend APIs.

You mentioned a situation where deploying the backend to Vercel might not be a good fit, which goes to another question I've been thinking about. To what extent does Next.js unlock a lot of the developer experience upside, and to what extent do you also get that using other languages? I'm curious for your perspective on how agnostic it is to language, as far as delivering on those developer experience features.

I'm not sure if I've looked closely enough at that lately to see what that looks like. Next.js in my view is clearly an outstanding tool to work with. In the visibility that I have into the industry, Next.js just seems to be continuing to grow, with more and more developers starting to understand how nice it is to work with.

Certainly if you're using Next.js, it's a natural fit to deploy to Vercel. If you're working with other technologies and you're thinking about deploying to Vercel completely exclusive of Next.js, I've experimented with that but never done anything serious. It's been a couple years since I've done that, so I'm not really sure what that looks like right now.

One thing we've been thinking about is this comparison with Heroku and how Heroku arose as a specifically Ruby on Rails platform, though obviously it supported other languages. Do you have a take on whether there's something that Vercel is doing differently from Heroku, and if there are lessons learned from Heroku's mistakes?

I'm not really speaking from expertise here, but there's certainly this major paradigm shift with Vercel, whether you're working with Next.js or any other technology, which is that it's going to be deployed serverlessly. That changes your programming model, and everything extends from there in terms of how you're going to design your system.

That's a big shift from the Heroku model, for sure. Whether it's with Rails or JavaScript or any technology that you're deploying to Vercel, you're going to be thinking about how you're interacting with your so-called backend differently. You're going to be thinking about short executions, stateless as opposed to long running processes.

Do you see serverless as a trend that could potentially be replaced with some other way of working, or is it something more durable?

I've noticed more and more teams are moving towards serverless, but at the same time you see a lot of skepticism out there still. You'll definitely see folks trying to make a case that it's a terrible idea and you are going to inevitably hit the wall where you have to move away from it as your product and your system grows. So I guess it's up for debate still in the industry.

I'm biased. Certainly my perspective is that where those limits start to come into play is at a point where you would expect to have the resources to move in whichever direction you need to move, if you need to do something different than what's supported out of the box with Next.js and Vercel. So it buys two, three, four or five years at the start of a project and just completely eliminates all of this friction -- all of this cost of teams to stand up this type of infrastructure and maintain it.

I guess there's also some debate in the industry around whether you can design a system similarly, run it on your own servers and get your costs down dramatically, as you start to pick up a little bit of scale. There will be some questions there, as more and more teams start actually growing on this type of a platform. How soon are they hitting the point where the bills are starting to be like, “Whoa, this isn't going to work”? To be honest, I don't have too much experience actually getting projects to that point. At least I can say that there's plenty of room for a good solid year or two as you're starting new projects to not be worried about costs getting carried away.

You’ve talked a bit about the benefits of building with Jamstack, for lack of a better word. If you were giving a pitch on why a startup should build the first one- or two-year iteration of their product in a Jamstack kind of methodology, I would love to hear that pitch, what that would save them and what the benefits are.

It's important to mention just the workflow that's sprouted up around and been driven by Vercel for small teams or even larger teams. This kind of Git-centric workflow where, by pushing your commit to a branch, the Vercel GitHub app automatically picks that up and deploys it in a preview deployment. That's discoverable as a URL in the pull request itself. Obviously, it can be published into other channels, so that when you're working on a feature and you have designers, product managers and founders who want to take a look at the latest iteration of what you've been working on, your default workflow of just pushing to Git unlocks all of this discoverability and production-like deployment environments for each branch with no effort. There's not even any configuration or setup -- it's just out of the box free.

That's a really powerful thing, when you start to have that friction reduced when you're iterating. That's how you start to be able to move fast in these early phases -- as a team, not just as developers, but even extending out into the wider team and collaborating. The workflow and the development connection to the rest of the team are a big part of the pitch for early startups.

The initial value of a system like this is how easy and fast it is to stand up and get started and prototype, but you often think of that as a trade-off. Firebase is a good example of that. Firebase has its reputation for being really, really easy to get started with, but some people will say that if you choose Firebase initially, you basically are doing so knowing that at some point you will have to migrate to something else, if for no other reason than just for the cost that exponentially tends to scale. The difference with this architecture and this offering is that it's amazing for prototyping, but I think it's also very suitable for transitioning into scaling as well. I mean, you're starting to see big established teams with massive properties moving from other systems onto Next.js and Vercel. So there's no sense that, “Oh, for sure, we're going to have to move away from this someday.” It's, “You can have your cake and have a really good chance that you're going to be able to eat it, too, down the road.”

We looked at Firebase as another technology that has caught on as an easy way to build and deploy real time apps, but from our vantage point it hasn’t exactly become

core infrastructure for the web. Besides cost, are there other factors you’d point to as constraining that?
Well, the really nice APIs that their SDKs have are nice, but ultimately, fundamentally what's below that is a NoSQL document database, so you're limited in terms of the types of queries you're able to execute. Whether the API is nice to work with or not, it's not going to be able to change fundamentally what you're working with.

These types of data stores are designed for extracting very specific workloads that have different scaling requirements of properties than the rest of your app. You can put something in a DynamoDB and just scale it to infinity independently as one piece that you've extracted out of your app, but to build your whole app with this kind of a database is really limiting, or it can be.

That's where I'd throw in a plug for Supabase, which is kind of taking the developer community by storm the last year or so. Supabase markets itself as the open source alternative to Firebase, but it's actually fundamentally just a different technology. It's Postgres over the hood; it's a relational database. Then they're trying to push that ecosystem forward with a more Firebase-like developer experience, with really nice SDKs to work with, really nice APIs, really nice tooling and dashboards and all that stuff. It’s really easy to get set up and get started. So you're seeing some of these things come together to have a real set of options to build real, scalable and sophisticated apps with a Jamstack model.

We've been hearing a lot about Supabase, and other ones that are coming up are Blitz, RedwoodJS, Remix. These might not all fit in quite the same category, but I'm lumping them together in my head as what someone called “post-Jamstack” or “hybrid Jamstack.”

Yeah. Remix is interesting because they're kind of making the case for moving back away from Jamstack. That'll be something. There's no dogma. This is just about what's the best in the moment that's available. It'll be interesting to see where they're able to take that. I haven't followed Blitz too closely, but it seems like they're moving back towards a Next.js focus recently. Everybody's continuing to explore the space for sure and even extending in all directions, both forwards and in some ways backwards -- full circle types of things.

Netlify obviously pioneered Jamstack in terms of terminology, so it's potentially concerning if Jamstack turns out to be a transient thing, while all these practices we've talked about evolve and stay relevant but change in ways that depart from Jamstack. Maybe Vercel has better staying power because it's not stuck dogmatically in this previous idea of Jamstack, though I think Netlify itself is coming to feature parity to some degree. Or, as you mentioned, maybe it's a matter of taste, and there's not so much a massive gulf between what they do.

I think that's more or less been my experience. The last time I worked with Netlify was a couple years ago, so I'm sure that they've been continuing to improve their offering. Not only matching some of the offerings from Vercel, but I'm sure that they have their sort of stronger offerings where their focus differentiates their offering from Vercel a little bit.

Jamstack as a staying terminology and as a staying way of describing how we're all going to be building apps -- I'm not sure. I mean, I think that it started to take on a meaning or an understanding that's more than just how it's been defined, in my mind at least. “JavaScript, APIs, markup,” but for me it's more about eliminating infrastructure wherever you can.

For example, the paradigm shift of connections to your database, which for me -- until Supabase -- has always been a point in the stack where there's friction and where you're concerned like, “Wow, this seems like a place where things go could really go badly.” It's this weird thing, where it's like everything we do is all HTTP, except right here we've got database connections that use these really low-level libraries, and if something goes wrong, you're trying to understand what is even happening here. Now with Supabase, your database becomes just another HTTP API.

For me, that's what Jamstack is all about: eliminating friction, whether it's clunky technologies, servers or infrastructure of any kind, even if that means that it starts to evolve back into more of a server full model in some cases. With Supabase, the core of their offering actually is this technology PostgREST, which is a REST API HTTP server that automatically generates a REST API from your database schema. I've been tracking PostgREST for years -- it's an amazing technology. It's written in Haskell and is rock solid engineering. But if I wanted to use

it, I had to stand up the server, run it, maintain it, keep it running and monitor it and all that stuff, and I just didn't want to do it. I want to pay other teams to manage all my infrastructure for me. So I stayed away from it, and now Supabase comes along and just unlocks that for me by running it for me.
Switching gears to APIs, I'd love to hear if there are APIs where you're dedicated to one -- like Supabase, or maybe Contentful for CMSs or Algolia for search -- and are there other categories where you're more agnostic about what you'll use?

Supabase is definitely getting there for me. I guess a lot of people would view it as pretty early for their offering, but at the same time their underlying technologies are not new at all. So when Supabase talks about being in beta, they're talking about things that don't really matter that much in your system -- it's a dashboard or tooling and nice things around your core technologies. But the core technologies are open source and have been around for decades or years at least, and are known to be the best available in the open source community. So that's definitely one that for me is already there as far as, I'm bummed if I have to work on a project where I can't work with Supabase, because I know it's going to bring in all this headache when you see, "Oh, but the path is right there, we could just get rid of all this nonsense." It definitely puts it in that category.

I'm not sure if there too many other APIs that just every single project has. Stripe -- it's hard for me to think that I would ever build a payment solution with another offering. I guess maybe there are some that you might look at, but that's a pretty trusted go-to API that you're going to work with if you're working with payments. For me, magic.link for auth is one that I'll always reach for. It's passwordless authentication as a service, but I think that's mostly just me. I don't think it's quite as popular as Stripe or even Supabase, and there's certainly plenty of well-established competition with auth offerings.

Trying to think about services that are just like "use whatever works." There are two or three players I think in a bunch of different areas, right? Cloudinary for an image hosting and processing API. And then storage is one where Supabase is also starting to be something to look at there, but that might be one where typically folks would just reach for S3.

In talking about Next.js with folks, it seems like the line between a JavaScript framework and a static site generator has gotten somewhat blurred. I'd love to hear your take on how exactly we should understand the differences between a Gatsby and a React in terms of how they function here.

This is not something that I have a whole lot of experience getting really deep into. Gatsby seems to have really extensive tooling around that particular use case. When I've looked at it, it just seems like that's their focus: building hundreds or thousands of pages statically, even to the point where a page is implemented with techniques that are really centered around that use case. There's a GraphQL layer there that is intended to be used throughout your app. Stuff that I wouldn't think to do if I was just building an app without any of these technologies or services. It seems very much like, “This is the Gatsby way.” And then there's an overwhelming number of plugins and all kinds of things that have grown out of that whole focus, that for me is kind of an afterthought.

Like I mentioned, I just want to make my app the same way that I would if I was working with Create React App, just simple components. They're in a file, and it doesn't look any different than Create React App would. It's only by the defaults of Next.js that it happens to be deployed a certain way. Then if I want it to be deployed a different way, it's this tiny little API surface where I just have a couple of options to choose from. It's not overwhelming. It's just one little tweak here, and boom, now my app is rendered and deployed in a totally different paradigm.

I guess if your site is one that has that model of thousands of pages, and you want to build all of them statically -- maybe I don't know enough to say whether or not Gatsby has meaningful differentiation there. Maybe it does. But for me, I want all my pages to be rendered statically. Vercel refers to it as automatic static optimization. That’s the common use case for me. I guess it must just boil down to the priorities of each app.

Feel free to speculate wildly, but I'm curious what you see the next five years looking like for Jamstack, Next.js or both more broadly. How do you feel about the future?

It's really hard to predict. I would've never predicted when Vercel pivoted from Docker-centric to serverless. Everyone loved their offering at that time. The developers that were using ZEIT were fans of ZEIT. Even just the branding and everything, it was just like chef's kiss. And they just completely changed course overnight. I mean, they took care to make sure that everyone's apps could keep running through the transition, but all of a sudden one day it was, “Boom, we're doing something totally different.” Now, is that something that happens once every five years or ten years or never again? it's impossible to say.

Definitely most of the energy that I can see seems to be pointed at the edge. For me, I guess I'm not so much of an early adopter on that stuff. I try to benefit from whatever the happy path is in terms of the Next.js/Vercel ecosystem and where their documentation is guiding you towards. Whatever they're suggesting seems like the easiest way to go is what I'll generally try to follow. Some of the edge stuff right now still seems a little bit like you have to go looking for it and plug in pieces from different providers, potentially Cloudflare or whatever case may be. So I guess I've been standing by, waiting to see when the time is going to come where I want to really invest heavily in moving in that direction.

But there's really interesting stuff. You can see I don't know that much about it, but I can see that Cloudflare's doing all kinds of crazy stuff, and Fly.io is doing all kinds of things, and now Vercel is making it clear that they're going to be putting a lot of investment into that space as well. If I had to guess, five years from now, we're probably solving some of these problems around latency and global distribution in this same programming model but taking more advantage of those kinds of networks.

Disclaimers