EPISODE 1217

[INTRODUCTION]

[00:00:00] JM: WordPress is a free and open source content management system or CMS written in PHP. Since its release in 2003, WordPress has become ubiquitous on the web and it's estimated that roughly 60 million websites use WordPress as a CMS. However, despite its popularity, WordPress has limitations in its design. WordPress sites are dynamic and the front and backend are tightly coupled. A dynamic full stack application can be used when handling complex functionality, but also slows down the site and opens up security vulnerabilities.

Zeev Suraski is an expert in PHP. He's also the CTO of Strattic, which is a static site generator and hosting platform that specializes in converting WordPress sites into static architectures. Today's show focuses on PHP, but also has some discussion of WordPress architecture. Zeev joins the show to talk about the place of PHP and modern web development and how his company Strattic helps WordPress developers build modern, fast and secure websites.

[INTERVIEW]

[00:01:00] JM: Zeev, welcome to the show.

[00:01:02] ZS: Hi. Good being on board.

[00:01:05] JM: You were a very early PHP programmer, and I'd love for you to take me back to the early days of PHP. What were the goals that the language set out to accomplish?

[00:01:19] ZS: So, yeah, we're talking the late 90s, and basically then there were not too many solutions for creating dynamic web applications. The goal back then was to create a language that is really simple and allows you to create dynamic web applications in a pretty straightforward way without too much complexity and to really take away not just the complexity in the language syntax, but also in setting up a backend solution.

When I’m talking by the way about dynamic, we're in 2020, a long time after it. So I’m not talking about the more modern meaning of the word dynamic where we typically talk about which client side. I’m talking about something which is simply more advanced than just plain static HTML where you have some sort of logic running in the app doing some calculations or communicating with a database and so on. And the goal of PHP was to essentially enable these kinds of apps and do it without the complexity that was typically associated with it.

[00:02:24] JM: What were some of the early design decisions around PHP that really influenced how the language has ended up?

[00:02:32] ZS: That's an interesting question. So I have to admit, PHP was a language that evolved a lot more so than was designed. I got involved with PHP when actually PHP 2.0 already existed. And very little design has gone into PHP 2, though a lot of kind of mistakes that were made or things that were just kind of made sense at a time when it was one person creating a solution that originally was all about suiting his own needs. And when we created PHP 3, when Andi Gutmans, my colleague back then and myself created the foundation for PHP 3, we actually didn't question almost any of the fundamentals of PHP 2 and we kind of copied most of them verbatim and we ended up kind of replicating the same mistakes and also adding some of our own.

And since then, for a long time, for several years that these least PHP simply evolved. The goals, like I mentioned before, was really about simplicity. We wanted to create a language that would be simple to use, would be a kind of similar to a scripting language version of C. All of us essentially, the development team had C background, and we on one hand loved C, but on the other hand it was quite obvious that it was the wrong language for the job as far as the web is concerned. I mean, even though I do have some background in developing C and even C++ CGI’s, it's not something that I think anyone should do or should have done for that matter. And we wanted to bring on one hand the kind of the power of the language, but at the same time make it available to end users that don't need to learn the complexities of C memory management and typing and all that. So maybe when we talk about design decisions, the ones that would made early on, it was – Maybe the main design decision was to create a loosely typed language where scalars are essentially interchangeable and strings convert to numbers and back pretty much seamlessly depending on the context. And we wanted it to plug in seamlessly into web servers. We wanted to make it web first. So we're not after creating a general purpose language whenever there was any sort of conflict between creating something, which is more specialized for the web workload or the web environment we would – Versus getting it to be a more general purpose, we would opt for making it more web specific. And I think those are really the two early design decisions that we have stuck by for a very long time.

Another decision that we adopted relatively early on when PHP 3 really started to gain popularity, and it really gained popularity beyond our wildest dreams. We didn't even know to dim it back then. So when it did, we wanted to create a language that was downwards compatible so that the huge workloads, the growing number of workloads that were using it will not have to radically change. So that did make it more difficult to fix some of the mistakes from the early days. But on the other hand, we have gigantic applications in application ecosystems that have grown on top of PHP, and I believe it has a lot to do with that.

[00:06:10] JM: Why did PHP get this reputation as being insecure?

[00:06:19] ZS: It's an interesting question. I think that I actually wrote to blog post about this about probably 15 years ago. It has to do with several things. First of all, PHP did have its share of issues. Not something which is kind of disproportional to the amount of code that is in it or the amount of popularity and eyes that are scrutinizing it, but of course like any piece of software, especially open source software that is relatively easy to scrutinize, it had its share of real security issues. So there's that. I'm not going to deny it. I don't think there's any point in denying it. But actually that list is not that long.

Then though I think at least two other things that played a role in this reputation, one is that there was some features which I would call missed features that made it all too easy for end users to create insecure code. And we could play the game where we would say it's the end user or the code author's fault because they did not have to use this feature insecurely or it's not meant for this use case and so on and so forth. But I think that that it's fair to take responsibility for that and say that things like a feature like register globals or magic quotes and all sorts of other things that are today kind of baked in PHP's history and rightfully so were just missed features that made it all too easy to get insecure applications.

And then there was another thing, which may sound a bit like an excuse, but I actually think it had the biggest role in creating the perception of insecurity for PHP, and that is the fact that in the early days most of the popular PHP applications included the name PHP in their name. So things like PHP Nuke or PHP this or PHP that. And whenever these apps had security issues and because they were becoming very popular, security issues were found. Just in people's minds when you had PHPX or PHPY creating published vulnerabilities, in their mind it was linked with the language itself.

Now, like I mentioned, partially it's probably justified because in some cases which we made it all too easy to create insecure code, but a lot of it was actually not justified. And like in the same manner, that not any security vulnerability that is found in, say, OpenSSH, reflects badly on C in the same way not any vulnerability that was found in some popular PHP app had to necessarily reflect badly on PHP.

Kind of another thing that I think played a role is that PHP was one of the first languages, maybe the first language, where by definition almost any applications you created with it was remotely accessible. Because, again, we focused on the web workload and we designed it to be for the web and almost nobody used it outside of the context of the web. And when you're creating a multi-accessible application, especially in the early days when the knowledge about how to create secure applications was really scarce, if there is a vulnerability, the likelihood that it would be found and exploited was a lot higher than a vulnerability in some app written in C or virtually any other language, because in other languages, they're not remotely accessible. They’re not by definition web applications that are both accessible. A lot of client code. A lot of administration code and so on. So I think that also played a role. The fact that just in terms of the environment, the web environment is simply hostile by definition. Anyone from anywhere on the planet can attempt to attack you and exploit the vulnerabilities. So for sure, that was also the field in which we played had a lot to do with it as well.

[00:10:15] JM: There are a number of really large PHP applications that have scaled over time. Principally, I think about Facebook, because I've done a lot of coverage of Facebook. Tell me your thoughts on the scalability techniques of some of these large Facebook or PHP apps. I guess there's also Slack. But what are your thoughts on the architectural choices in the languages that have been built on top of PHP?

[00:10:45] ZS: So I can – Definitely, I mean, there's also Wikipedia and there's quite a lot of like gigantic and WordPress which virtually powers the content side of the web. Yeah. As far as PHP is concerned, our approach from the get-go on the low level was to create a share nothing architecture. So essentially this was definitely – if we go back 25 years into the past in terms of looking at the technologies back then, the queen of the web back then was Perl. And there was ModPerl for those who remember, which took pearl and made it a bit faster than using in a web environment. But it was a bit messy. Essentially, when you were developing a Perl-based web application and running it on ModPerl, you needed to kind of account for things that generally I don't think that ideally a web developer should account for. So if you needed to, for instance, clean resources at the end of handing a request. And if you didn't, then you could create all sorts of memory leaks or resource leaks, which in, again, a hostile web environment, it's hostile not just in terms of the ability to exploit vulnerabilities, but also in terms of potentially the load that you need to be able to code against. And Perl was not really all that good in it and it really left the burden for handing this to a large degree on the shoulders of the developers.

We wanted to essentially make it possible for the developer to only focus on serving a single request at the end of serving the request at the beginning. And at the end of serving this request they don't have to worry about resources at all. Whatever they used is going to get feed. That includes both memory and sources, database connections, file handles and so on, and network connections and so on. In the same manner, when they begin the request, they don't have to worry about do I have some leftovers from a previous request? Everything is clean. And for the most part, when I say the most what, maybe sounds a bit of an understatement, but it really worked well. You really did not have to worry about it. And for a very long time I think it scaled very nicely and it allowed the kinds of applications that you mentioned like Facebook and Wikipedia and others and WordPress and others to really work in scale very nicely. The share nothing architecture made PHP a very good choice for creating scalable apps, because the scale nothing part didn't work just at the request level. It also scaled nicely beyond the single server. So you could have a single server that had, I don't know, a hundred PHP processes that could handle a hundred concurrent requests. But when that was not sufficient you could throw in another server, another server to scale pretty much linearly.

Now, of course, you also had to potentially scale shared resources like the database. But as long as you had a solution, as long as your database was scalable, the PHP side of things really posed an issue and made scaling the application relatively straightforward. I can't attest to how it is to deal with Facebook scale. I'll admit that, or for Wikipedia scale for that matter. There're a lot of things that I'm sure they've done in terms of the management and so on. But at the fundamental level, PHP applications, because they encourage you to share nothing other than through third-party systems like databases or web services, they scale very, very nicely and very linearly. And we see it to this date. I mentioned that this worked really well for a very long time, but I think maybe about five years ago, slightly more than that, the model of microservices became more and more pervasive. And PHP was admittedly less geared toward that, less tuned towards that because we always assumed that this fairly negligible overhead that is associated with starting up the process and then cleaning it up in the end, it's negligible compared to the runtime of the request because the request typically did some meaningful processing and crunching and so on. Even if it took 300 milliseconds, compared to two, three millisecond digitization and maybe 10 millisecond cleanup, at the end, those 10, 15 milliseconds of initialization and cleanup were negligible.

When you're dealing with a microservice that could – Its own runtime could only be ten or five or fifteen milliseconds, then suddenly this initialization and cleanup, which take off the same amount of time, they're no longer negligible. And then I think models like the modeling of node.js became a lot more popular. I still think that they are more complicated to develop for and the less robust in terms of reliability. If a PHP process caches, there will be zero influence for the overall stability of the server. If a node.js process caches, the effect is a lot more substantial. But on the flip side, the node.js model gives you the ability to handle a much larger number of connections and handle microservices a lot more efficiently.

Thankfully, there's a model that someone or a group of people has bought this model also to PHP. So this is also why I kind of feel comfortable mentioning that without kind of bashing PHP 2 hard, because today there's also the ability to choose this model for PHP using a component called Swoole, and you can essentially do away with this complete isolation, the varicose version that has served us really well since the late 90s until recent years. You can do away with it and instead have a non-blocking IO approach where you are handling requests a lot more efficiently and if that is necessary. Both models are available today. Still, the traditional model is a lot more common and popular, but the alternative model, the node.js like model is available and is growing in popularity.

[00:17:13] JM: So you now run or you're the CTO of a company called Strattic. What programming languages choices are you making at Strattic? Are you are you all in on PHP?

[00:17:26] ZS: Actually we're not. We obviously use PHP for the actually learning WordPress, but WordPress is something that we only run at the backend and in kind of behind the wall. The folks who end up viewing websites that are hosted on static don't actually ever interact with WordPress directly and consequently they never actually interact directly with PHP. For the most part, like I mentioned, we run WordPress at the backend. Our customers, the ones who are running those websites, they're using WordPress directly in PHP and they use it pretty much traditionally in the same way that they would use it anywhere else to create content, create new update and so on.

But at the end, when they want to publish this content for the world to see, they hit a publish button. We actually create a static version of the website, which at the end is 99 just HTML and CSS and JavaScript and whatever static assets are needed to serve the site. In terms of what we use, this is really more of a cloud DevOps kind of system and it's mostly built with a node.js and with a tiny bit of Python. But really for the most part, node.js. It's not a decision that I was actually involved with, because I actually joined the company after much of this was already in place. But I think it's a valid choice when you're creating an application like this, which is very DevOpsy. Not a lot of web content and make extensive users of lamdas, which unfortunately still don't have native support for PHP on Amazon, on AWS. I think that the choice is very valid to go with it.

[00:19:18] JM: So I'd love to know a little bit more about your journey in the PHP world. You were the co-founder of Zend, which built a lot of technologies around PHP. What was the business model for Zend?

[00:19:36] ZS: So I started slightly earlier we got there, but basically when Andi Gutmans myself got involved in writing PHP, I mentioned this before, we didn't really have any plans or even we didn't really imagine what's going to happen with it. We certainly did not imagine that it would become one of the most popular platforms for creating web applications in the world. Back then PHP 2 was very small. I mean, it had probably something like 10,000 people using it all around the world. And fairly shortly after PHP 3 was released, we were talking about softly a million websites that they were using it. It grew really rapidly and to the point that we really didn't believe the numbers for a very long time until we realized that those numbers – We had started getting numbers from all sorts of different directions. In different directions, we realized those numbers actually too.

In parallel to those numbers growing we started getting more and more requests from companies that were using PHP. We talked about again the late 90s or early 2000s, and they wanted things that the open source version did not offer. I mean, namely the ability to sell PHP-based applications without disclosing the source code. The ability to manage applications for business critical applications. Again, because we didn't really have any special plans or imagination about what feature is going to be used for, we really thought about it more of as a solution to create applications for personal purposes. But suddenly more and more companies were using it. And the ability to use it in a business critical environment was somewhat limited.

I mean, not in terms of licensing or anything, but just in terms of uh support, a lot of companies would not really be able to use it and developers will not be able to convince their managers to use it unless there was a company that provided commercial backing.

So that's why towards the end of '99 we founded Zend Technologies with a fairly wide mandate to provide commercial backing for PHP and help proliferate it. We created several things along this journey both software solutions to help manage PHP in the business critical environment, manage clusters of PHP, deployments of applications, performance management, performance boosting and all sorts of things like that. These solutions are available to this date and are quite commonly used in enterprise deployments of PHP.

The fact I think that we provided commercial backing for PHP both in terms of providing support, but also we really talked with a lot of the largest companies in the industry including IBM and Microsoft and Red Hat and Oracle and virtually all of the major players in this field to make PHP more acceptable. Because historically PHP, some of it is related to what you mentioned in the security questions. It enjoyed “the perception” of a non-serious hacky solution for a long time. Some of it was justified, but a lot of it was not. And we worked really hard to make PHP acceptable and mainstream. And today, I mean, the fact that you can find PHP, for instance, on Google Cloud, it's a deployment option. It's available. It supported – The divers of Oracle are supported by Oracle and not by the community. And Microsoft also supports PHP and has support for PHP on its cloud and so on. These things did not just happen. Those had a lot to do with things that we did in Zend. Proactively touching base with those companies and working with them to both improve the technology on the platform, but also make it acceptable and mainstream and get the kind of seal of approval so that users that are using oracle, users that were using Windows server and wanted to use PHP, they would see that the kind of main vendor approves of that and does not consider to be some sort of an unsupported environment.

Other things that we did over the years, I mean, we did a lot of things, but some of the things that are worth mentioning, we also worked on creating a framework that would take advantage of some of the then most – The newest and the most advanced features of PHP 5. PHP 5 was the first version that really td introduced object-oriented programming. And there was a bit of a vacuum in terms of the framework that people could use to develop apps on top of PHP instead of reinventing the wheel every time. So we also led the development of the open source Zend framework, which is still today it's called Namina. It was renamed a couple of years ago. But it's still very popular. Back then it was perhaps the most popular enterprise frameworks for PHP. Today it's one of the popular frameworks for PHP. And I think that too helped help PHP proliferate, because it was one of the – It probably was the number one complaint that we heard from users that in terms of interoperability and in terms of avoiding – Having to invent the wheel every time, it was lacking. We were talking about already about a decade in and still there wasn't a super popular framework for using PHP. So that too I think was important. In general, like I mentioned, the mandate was pretty wide. Anything that we could do to help proliferate PHP is something that we considered and we acted on a lot of it as well.

[00:25:23] JM: So in modern software engineering, when do you think PHP is a good choice? Do you have any framework in your head for what kinds of applications should or could choose PHP?

[00:25:39] ZS: That is excellent question. I think, in reality, the most honest answer for you would be that I don't. But it doesn't mean that there is no use case for PHP. I still think that PHP is a good choice. The millions and millions of developers that are using PHP and are familiar with PHP, and today with the performance gains that we've realized in recent years and with the robustness of the solution, and the fact that we've introduced the ability to use it in a synchronous IO, not blocking IO kind of way. So in terms of what you can do with it, you can really do anything.

In general, from the get go, even when we were kind of working hard to proliferate PHP, we're never kind of zealots around it and we were very open to people choosing whatever they prefer to use. We thought that PHP had some advantages over other languages. But in the end today I think it's very much a personal preference. I still think that developing a web application that doesn't need to scale through the roof. And let's face it, in many cases that's the case. And even if you think two or five years in, you don't necessarily expect to have to deal with millions of concurrent users at a given time. Most apps don't reach Facebook scale.

Then the traditional model that PHP offers that I mentioned earlier, this sure nothing approach, is by far the simplest and fastest to develop with compared to the node.js or the Swoole approach that exists in PHP, which is equivalent. So for those use cases, I would recommend PHP hands down. I think it's the best solution and there are really few downsides to it.

Another use case is that if you have a team or you yourself depending on the nature of the project and you have a lot of experience with it, I still think that it's a very productive language. It's very fast. It can really scale well and there's no real need to switch to something else. Where I think I would consider PHP as one of the options, but not necessarily the only option, is really the super scalable applications where you need the asynchronous IO approach. And then I think some of the gains of PHP are diminished because, I mean, the language syntax remains the same. But in terms of what it buys you, the safety and simplicity that he buys you from an architectural perspective kind of goes away and it's more of an apples to apples comparison with Node.

And then with Node, Node is more popular than PHP on this font, hands down. I mean, for asynchronous IO, there's a lot more support in Node than there is in PHP. Not so much to some degree in the code itself in what they can do, but more so in terms of people who are using it and have experience using it in such an environment. So you can definitely find a lot more node.js developers than you can find PHP Swoole developers. So that can also play all in the decision.

Like I mentioned, in Strattic, we do use node.js for that purpose. And I’m not necessarily certain that if it was up to me to make the decision, as I mentioned, it happened before my time. But I’m not sure that I would have made a different decision. In the end of the day I think that choosing a language is a very much individual decision or matter of preference. I think that modern languages, if you go back in time, then those substantial differences in what you could or couldn't do with different languages. Today, those differences, those gaps are really narrowed down and it boils down to logic, to a matter of preference. So it's really up to individual choices or team choices what they prefer to use.

[00:29:44] JM: I'd like to go into some detail about the company that you're working on now, Strattic. Your company is built to improve the uptime and performance of WordPress sites and to make WordPress sites static websites. Can you describe the thesis of Strattic?

[00:30:07] ZS: Yeah. So WordPress is by far the most popular web application on the planet essentially. It accounts I think uh for something like 40% of the web, the whole web workload, just one app. And everyone, maybe with a small asterisk, but almost everyone is using it or including anyone from someone creating their own website and all the way to the largest enterprises in the world. So that is in terms of like 20 seconds on what WordPress is.

The problem with WordPress is that in terms of this application that was created something like 20 years ago, the architecture is such so that performance is a major issue. And even though playing Vanilla WordPress, this thing that you just download and install, it's pretty fast, but nobody uses plain Vanilla WordPress. You start installing plugins and you start creating complex custom code and so on. And most real WordPress deployments are pretty slow. And there are all sorts of ways to tackle that, but it's considered a major pain point in the WordPress world. So that's one issue.

The second issue is security. the LAMP stack, we touched that earlier a bit. The LAMP stacking for that matter, and not just the LAMP stack, but essentially any stack that has programs running in response to end user requests that essentially uncode and process the request and generate a response. One way or another, they're potentially vulnerable to attacks, to attacks, to pretty bad attacks. Anything is vulnerable to some kind of attacks, but such systems that generate responses on the fly, they're a lot more vulnerable to attacks and those attacks can be pretty bad ones including potentially hacking in and grabbing the data or changing the code, altering the code and doing all sorts of bad things to people who visit the website and so on and so forth. So that's the second issue.

And the third issue that exists with WordPress is scalability, which is kind of the cousin of performance, but not quite the same. If your website is really popular, even if it's fast when just a few, a handful of people visits it, if you're successful and suddenly a lot of people visit it, it's not that easy to scale it. Even though WordPress itself scales relatively nicely, there are other things including certain plug-in considerations that you need to take into effect, scaling the database and so on. It's not something inherent or unique to WordPress, but it's something that is complicated with scaling any dynamic application. And WordPress has its own share of complexities.

The premise behind Strattic is to take all of these three complexities away so that performance would be superfast regardless of what plugins you're using. Essentially, take this whole consideration away from you. Similarly, with security, factor out this this whole issue of a dynamic language that is dynamically processing the request and generating response on the fly. Factor it out and take it out of the equation. And thirdly, make it super scalable and essentially pretty much agnostic to the level of load that it's being subjected to.

And the way we do it is simply by turning – I mean, of course it's easier said than done. But simply “turning’ the WordPress from a dynamic LAMP-based application into a set of static assets into a set of static files, we have a technology that plugs into WordPress to determine which essentially map all the pages that need to be publicly accessible and we then create replicas, static replicas of them. For the few things that require true dynamic handling like sending emails. If there's a something on the website that uh needs to send an email to process a form or really do something that by definition cannot be done statically, cannot be placed with a static alternative, then we implement this technology, these procedures using Lambdas, using Amazon Lambdas. And the result is essentially a set of static files that end users can communicate with. End users in this case, I’m talking about the people who are accessing the website is hosted on static. When they would be typing in the address of the website, the browser will not be communicating with WordPress on the backend, but rather just downloading static assets from a CDN.

Of course, that means if I visit those three issues that I mentioned earlier, that would be really fast. A lot faster than having to wait for the page to be generated by PHP or for that matter any other dynamic platform. It's a lot more secure because the attack surface is almost completely gone. I mean, usually WordPress and its plugins have a lot of vulnerabilities found virtually almost every week. And when you turn it into static, virtually all of them are no longer relevant. There's no word for signing. There's nothing to communicate with. No code to punch a vulnerability through. And like I said, the attack surface is simply almost exclusively gone. There's still a set of certain client-side attacks that could be somehow waged against even a static website. But for the most part, the vast majority, north of 99% of the attacks was simply rendered useless and inoperable and impractical to perform.

And thirdly, because this is a CDN, instead of static files, then it's just as fast if one person visits the site or if thousands of people visit their website. Actually, in some cases even faster, because it has a geographical distribution. And the more people using, say, I don't know, a website is popular in Brazil, then only the first users that are visiting the website in Brazil are going to have to wait for it to come over to the Brazilian POP. And after that it's actually going to get faster for the rest of the users. So it actually gets faster the more it's getting used, which is, well, exactly the kind of scalability they would want to have. So in a nutshell, that's what static is about.

[00:37:09] JM: So how often do you convert the website in its dynamic form to a static form? is it every time the user issues an update to the website you pull a complete static copy of it?

[00:37:29] ZS: Yes and no. Generally, we'd need to update the copy of the website, the static copy of the website every time there's a change. And until not too long ago, until a few months ago, that was the only way to do it. Today we made the plug-in technology, that the technology that plugs into WordPress more intelligent and we actually know which parts of the website changed. So we have the ability to quick publish and only so we publish those parts of the website that have changed. You still have the ability to force a full publish if you want to to be extra safe that we didn't miss anything.

And another thing that we're introducing is the ability to simply ask for a very specific page to be updated. If you don't want to even trust or rely on our technology that detects what changed and you want to essentially tell the system only this being changed, because, I don't know, you fixed – You fixed the typo or something and you really don't want anything else to have to change, then you have these abilities as well. And of course this greatly reduces the amount of time you need to wait before your changes are publicly available.

[00:38:43] JM: What have been the biggest engineering challenges of building the serverless static site generation tool that is Strattic?

[00:38:53] ZS: So, first of all, I would say that I think it required a leap of faith to actually think that it's feasible to do, especially for me coming from my background, we're all about dynamic applications for the last 20 years, to really think that you can take a dynamic application like WordPress that literally generates everything from scratch on every quest and you can actually transform it into a server, an offering that doesn't even have WordPress signing. I didn't mention it. But when you're not actually editing, offering content, the WordPress container is actually not running at all. It's completely disabled and the only thing that's alive are those set of static assets and lambdas.

So to make the sleep of faith and imagine that idea is possible, that's a pretty big step. I know it was for me when Miriam the founder and CEO contacted me a few years back and kind of vent this idea by me. My initial reaction was that it's probably not feasible. That it's not – I mean, you could probably do 60%, but the rest is going to be an uphill battle. So I had to make this leap of faith myself as well. But fortunately she convinced me that she believed that it was possible, and it is. And it's not that you can do anything. There are some things that are super dynamic and inherently dynamic that don't translate well into this model. But since we're dealing with a huge market and a huge user base, even if we can cater to 80%, 80% or 85% of the deployments, which I think is very reasonable, it can still be a very successful solution that will bring a lot of good for this community, for this ecosystem.

In terms of the technology assets, one thing was to create everything in a completely scalable way. Strattic was created from the ground up as a service, as a cloud service that is completely scalable and really everything – this is actually something that I personally did not have that much experience with because I always kind of worked on the infrastructure for creating web apps, but I didn't actually create a very large scale application that has a lot of different moving parts and all of them need to scale. It was interesting for me to be involved in that and see how it can be done. Of course, use those uh assets that today are kind of household items at the disposal of every modern developer. All of the AWS or, for that method, cloud services that you can use. So carrying something which is super scalable was one of the challenges.

I think another challenge that we have is ongoing. Basically, WordPress, is not one thing. It's not that we can say that we support WordPress, and we're done. It's an ecosystem and it has tens of thousands of plugins and a lot of plugins that are actually being used. So it's not that like out of 50,000 plugins and 49,900 of them are not being used. A lot of them are being used. And we need to be able to ensure that we work with all of them or most popular ones. So that is something which is an ongoing challenge to make our system more and more compatible with more and more plugins. But I'm happy to say that we've already reached the point where we've had a lot of users that onboarded themselves so that they basically opened in a tile account, migrated to Strattic. It worked and they put in the credit card and started using it. Something that honestly took us a lot less time than I envisioned it would. It means that already our solution is sufficiently generic and supports a sufficient amount of plugins that a lot of people will just be able to on board themselves. But still when we have new enterprises that are joining in, it's not uncommon for us to have to tweak the solution and add support for new plugins that are not currently supported.

I think there are a lot of different actions we can go into the future with all sorts of potentially using artificial intelligence and potential machine learning to do some of that automatically. It's still something that is kind of a long-term vision and not necessarily something that we are working on and going to do tomorrow morning. But I think there a lot of interesting challenges as we would want to make this solution applicable to more and more different kinds of deployments, then we would probably also need to involve some more advanced technologies as we go along.

[00:43:43] JM: Cool. Well, Zeev, do you want to add anything about the future of the PHP ecosystem or any thoughts about the future of software in general?

[00:43:54] ZS: On the PHP side, I can say that with a bit of a – I'm a bit conflicted about it, because I think PHP in recent years has taken a bit of a change in direction. And if we look at its roots, it was really about making it possible to create fairly complex web applications in simple ways and providing developers with simple ways to create it. In recent years, the action changed and less emphasis on simplicity, more emphasis on stickness and advanced language structures that are applicable, in my opinion, to a relatively small subset of the user base. At the same time, also, things like downwards compatibility have started playing a much less big of a role in the decision-making process of the folks who are stealing the language nowadays. I have to mention that it's a very open environment, and even though I'm one of the main folks who created in the first place, today essentially anyone who wants to can relatively easily get involved with the decision-making process of PHP. And most of the people that do get involved are the people that are probably unhappy with the way it is now, because otherwise they wouldn't have necessarily gotten involved. And it shows. It's being transformed to a more sticked and less downwards compatible language. PHP 8, it was released a couple of months ago. It is even described by legacy users, and by legacy users I mean mostly everyone, people who use WordPress, people who use all sorts of pieces of software that are using – That are dependent are based on PHP. It's described by some of them as a nightmare, because essentially downwards compatibility played a very secondary role in the decision making process and there's a lot of work that needs to be done, for instance, on WordPress before it can learn on the latest version of PHP, which is unprecedented. We've had downwards compatibility packages before, but never to that point and never without a very strong justification that went beyond kind of a change of taste.

Hoping I'm no longer directly involved with the development of PHP itself. I've been involved until not too long ago. So the Jet engine that went into PHP 8 is something that I was involved with. Dmitry Stogov was the lead the developer that created this pretty magnificent piece of code that makes PHP 8 tick a lot faster than previous versions was in my team at Zend before I joined Strattic. But I'm no longer involved in the day-to-day development of PHP. I'm hopeful that the feedback that is being generated by the user base of PHP will resonate with the folks who are leading the developer of PHP nowadays and so that they will continue on one hand making PHP more and more advanced, but not at the expense of essentially make it more complex and bake the stability that we've become kind of known for for the last 20, 25 years.

So like I said, I'm a bit conflicted about it. I'm hopeful. I think that the feedback that is being generated from users, people are listening to it, but maybe not enough. And I'm encouraging users, the end users who, in my opinion, the real owners of PHP, if they think that uh PHP needs to tweak its direction make their voices heard because internals, this mailing list were the decisions and the action for the future versions of PHP is being taken. It doesn't have a sufficient amount of representation from kind of mere mortal "users", and I think that this is something that ideally we need to change. So I'm really encouraging users, even if they are not contributors to PHP, in my opinion they own PHP just as much as the contributors do and it's theirs just as much as it is the contributors and they should make their voices heard.

[00:48:04] JM: Okay. Well, thank you so much for coming on the show. It's been a real pleasure talking to you.

[00:48:09] ZS: Thank you. Thanks for the time.

[END]