EPISODE 744

[INTRODUCTION]

[00:00:00] JM: Serverless computing is a technique for deploying applications without an addressable server. A serverless application is running on servers, but the developer does not have access to the server in the traditional sense. The developer is not dealing with IP addresses and configuring instances if there are different services to be able to scale. Just as higher level languages like C abstracted away the necessity of a developer to work with assembly code, serverless computing gives a developer more leverage by letting them focus on business logic while a serverless platform takes care of deployment and uptime and auto scaling and other aspects of cloud computing that are fundamental to every application.

Serverless can mean several different things. It can mean backend as a service products, like Firebase; function as a service products, like AWS Lambda and high-level APIs, such as Twilio. We've done many different shows on serverless, and if it's a hard topic to understand, you can dig into our back catalogue and find some previous episodes about serverless.

Today's episode is about ZEIT, Z-E-I-T. ZEIT is a deployment platform built for serverless development. In ZEIT, users model a GitHub repository in terms of the functions within their application. Zeit deploys the code from those functions on to functions as a service and allows you to run your code across all the major cloud providers.

Guillermo Rauch is the founder of ZEIT and he joins the show to discuss his vision for the company and the platform as it looks today. Guillermo was previously on the show to discuss Socket.io, an open source tool that he created.

We are looking for sponsors for Software Engineering Daily. We reach around 50,000 developers and you listen to the show. So if people like you are your target demographic for a product you're building, then you can check out softwareengineeringdaily.com/sponsor or you can tell your marketing director or somebody in marketing. That would help us grow and stay alive and be healthy and successful.

We're also conducting a listener survey. So if you have anything that you really hate about Software Engineering Daily, now is the perfect opportunity to fill out the survey and direct your frustrations at us in a constructive fashion, or if you like the show, then filling out the survey would be much appreciated as well. We're giving away some free swag to people who fill out the survey on a random basis.

With that, let's get on with today's episode.

[SPONSOR MESSAGE]

[00:03:00] JM: This episode of Software Engineering Daily is sponsored by Datadog. Datadog integrates seamlessly with container technologies like Docker and Kubernetes so you can monitor your entire container cluster in real-time. See across all of your servers, containers, apps and services in one place with powerful visualizations, sophisticated alerting, distributed tracing and APM.

Now Datadog has application performance monitoring for Java. Start monitoring your microservices today with a free trial, and as a bonus, Datadog will send you a free t-shirt. You can get both of those things by going to softwareengineeringdaily.com/datadog. That's softwareengineeringdaily.com/data.

Thank you, Datadog.

[INTERVIEW]

[00:03:54] JM: Guillermo Rauch, you are the founder of ZEIT. Welcome to Software Engineering Daily.

[00:03:58] GR: Thank you. Glad to be here.

[00:04:00] JM: ZEIT is a cloud computing and deployment platform for developers. There are many different kinds of developers these days. There are people at big companies. There're people at small companies. There're people who like to start companies. There're people who have a stable job but want to hack on some side projects. What type of developer is ZEIT for?

[00:04:21] GR: So the most important thing about ZEIT is that it's a serverless deployment platform. I think the beauty of serverless is that it's sort of enabling a developer to only worry about their business logic and not so much about everything that goes on with servers and cloud infrastructure configuration and so on. So I'd say it targets a very generic type of developer that wants to worry only about creating their applications, and deploying code to the cloud, it executes when web traffic comes in.

So we've seen adaption in anywhere from people that are learning how to code all the way down to the developer at the enterprise that wants to iterate more frequently and still deploy with the highest confidence in scale and security.

[00:05:08] JM: Tell me about some of the trends in software development that you're noticing that are shaping your decisions around building ZEIT.

[00:05:16] GR: That's a great questions, and there are a few very important ones. So one of them is that the standard way that we deploy applications is becoming global. So when you make a now deployment, we automatically configure a CDN layer in front of your deployment. So for example when you deploy a static website or when you respond with the HTTP cache headers from your deployment, that response will be globally available instantly to any of your visitors.

So I think this is interesting, because it didn't used to be the case that CDN Was a default expectation. Like you would say, "Oh, I need to optimize this or optimize that. I'm going to deploy and then I'm going to add a CDN layer on top," and you think this is a very healthy thing, because the internet as a medium is extremely global. One of the things that we want to do is that when you deploy to our platform, everything is just fast for every one by default. So that's I think one important trend.

The other one is we're seeing that frontend developers are the ones that are shaping decision making of cloud. So ultimately, the most important thing about any company is the product that customers interact with. Typically when you experience a website or a web app, the first things that you're going to be interacting with are the things that a frontend developer worked on. So you go to a website and they might be built with React or View. You download an app and it might use React Native or Expo or any technology.

I think what's interesting is our platform is uniquely catering to that kind of developer, the frontend developer, that ultimately is the one that is iterating all the time and is shaping the future of your product. So we are doing a lot of things for them, like making build processes instantaneous. Making it to get started with a React or View or Angular application really, really fast. So that's sort of one of the most important things that we've been seeing is the role of the frontend developer is just more important than has ever been, and we're seeing that also the enterprise, there's so much more interest in enabling this kind of developer.

Just to give you a quick example, we see that a lot of companies are taking their design systems very seriously. They are creating style guides and component galleries that is going to allow them to scale their frontend development. So that for example when one team creates a button, another team uses that same button and they don't have to reinvent the button. So this is something that our platform uniquely designs for, this kind of new profile of developer and team within the company.

[00:08:06] JM: Tell me more about the modern interaction between development and design.

[00:08:11] GR: That's a great question. One of the things that I think becomes super interesting and sort of connects to your previous question is the designer, the expectations on a designer and the sort of the leaders in the design industry have become so much more used to also getting involved with code and being a part of the engineering process as well.

I wrote an article a few years ago called Pure UI that made this argument that the role of the designer or developer we're going to converge overtime, because ultimately we've all heard the phrase that design is how it works. When you think about designing an architecture or designing a software engineering system, design is sort of where it all starts and it's the most important task.

Now when you talk about designing or UI, I think a lot of people have mistaken that for like only worrying about aesthetics or pixels. But design is ultimately the most important thing, because design is how it works. So we want to do as much as we can to empower that person to do more, to complete applications, to have an insight and decisions into how it all happens whether something is fast. What the interaction looks like when the network is slow, when the network is failing. All these sort of type of thinking that connects the pixel-oriented thinking and the aesthetic ideas to also the underlying engineering basis.

So that's kind of something that motivates us a lot is let's work on exposing tooling and frameworks such as Next.js where the designer cannot only just think about their application on an art board, but also make it come to life with not so much effort.

[00:10:11] JM: You're mentioning this emphasis on the frontend developers access and ability to iterate quickly with the deployment tool. What's the modern interaction between a frontend developer and the backend developer? Because we still do have backend code.

[00:10:32] GR: Yeah. That's an excellent question as well. So there are many ways that you can sort of architecture a frontend application. The most popular one that has emerged is this idea that your frontend application consumes a series of API endpoints. So the frontend developer can start by working against an existing API if there's one available.

What we see in organizations is that they didn't go yet through the process of untangling their frontend from their API or backend code. So an example of this is you have a PHP script, let's say, that talks directly to MySQL and outputs HTML as a result. So that actually works extremely well, because there are no hops and there's no network latency. There is no machinery between going to the source of truth and outputting what the user wants, which is some piece of HTML with some data that the user is interested in.

Where that breaks down is that you say, "Okay, I'm going to create a mobile application now. So how do I access the source of truth?" Well, you have to create a new series of backend endpoints that query your data and then output it in a way that the mobile application can use. The reason that this breakdown is that now you have two sort of co-evolving ways of querying your backend.

So most organizations in some form or another are thinking about, “Okay, let’s separate the raw data exposure, namely the backend API endpoints from the frontend system that consumes it, and by doing that, we can continue to evolve our frontend independently or even go to more platforms.”

So you’re asking, “Okay. How does a frontend developer then iterate and deploy?” Well, one of the best ways is that this frontend developer will iterate under React, Angular or View application and then they’ll know that there’s some API documentation that they can use to sort of query their data. For a company that’s super important, because by having that API, then they can expose the API documentation endpoints to other consumers or third-parties that might want to build on top of your company’s data and API ability.

So this is where we also see the emergence of GraphQL. There is where we see the ability for the frontend developer to worry about their job in the best possible way. Now what’s unique about out platform is that we give the developer also the ability to define some of the functions that go into the backend and even translate the spectrum of client-only rendering and also server rendering, which has a very important advantage when it comes to minimizing latency and increasing access to search engines and so on. It’s this idea that you can build a really, really modern frontend application without making any concessions in terms of performance and scalability for the modern needs.

[00:13:43] JM: You’re describing a lot of trends here. Are the major cloud providers, like Amazon Web Services or Azure, are they serving these trends well?

[00:13:54] GR: I think the main issue with the current infrastructure providers is that there’s that idea of like, “Okay. There is infrastructure and then there’s a platform that it can deploy to.” The infrastructure primitives tend to be very, very low level. We tend to think about this at our company as a compilation pipeline almost, where the most productivity that it can enable to developers when you give the high-level programming language, and that high-level programming language tends to compile down to primitives that the computer can understand. I think that’s what we’re starting to see in the cloud space where there’s this incredible power of primitives. But without the high-level interface, it’s very hard to become productive with them.

So there's almost this tradeoff between, yeah, you have access to the raw power of the computer. In this case, it's at cloud super computer, but you need an interface that is going to make you very productive as well. Because as I said earlier, I think something that we have to be very mindful in our industry is that we all live and die by the experience that we give to the customer, and the customer, the end user only cares about the pixels on the screen. They don't care that you use the latest infrastructure. It's like 10 layers removed from their life.

I think where we come in is that we are able to elevate all these amazing infrastructure that already exists and we can give you this high-level interface that we know is what's going to enable this great end user interactions. That's why I was saying that we think we've spent a lot of time thinking about frontend developer tooling, because we think, "Okay. What are the patterns that are working really well on the web and on mobile to make really snappy interactions?" What's working well to make users happy?

Then we worked backwards to the technology and we say, "Okay. What are the right primitives to use from this amazing infrastructure that exists, like Google Cloud, and Azure, and AWS, and what is the best experience that we can give to the developer so that they can – Without worrying about it, utilize those primitives and server the end user in the best way possible?"

[00:16:15] JM: Okay. So the idea of a higher level cloud provider with the catering to these trends that you're talking about, I think that's what you're going for with ZEIT. Can you talk more about the products that you're building at ZEIT?

[00:16:34] GR: Yup. So the main product that we build is Now. It's a platform that enables for you to only give us code. In particular, you give us the build steps of your code and these serverless functions that you can define in any programming language, and then we deploy them automatically for you and we give you a URL back. We deploy them to a CDN so that you don't worry about regions and you don't worry about latency to your end customer.

As I said, you don't configure any servers. You just give us code. The way that you typically give us code is you install, for example, the GitHub app that connects Now to the source code storage and automatically deploys it for every push of your code, for example. If you think about the day-to-day of your application creation process, the developer pushes code to GitHub. A deployment is automatically made and that anyone in the team can see in real-time the output of that deployment. They can interact with it, QA contest it. The boss can check if it looks good to them, and there's this collaboration that happens around the result of your deployment.

If you think about what GitHub did for code, we're doing for execution of that code. So we really bring it to live, which is a very exciting idea. So that's the main platform, which we call Now. You might recognize it, because when you share a link, the URL ends in .now.sh.

In addition to this, we create a series of what we call builders to plugin existing frameworks with zero effort. For example, we have a builder called Now/node. So you can define your code in Node.js and we automatically build it and deploy it for you. Another one of these is Next.js, which is a React framework that builds in all the best practices for production in one package without you worrying about setting up webpack and setting up the build pipeline and so on. So you can use Now/next to create a next generation React application.

So this is sort of how the platform comes to life, and the best thing about it is anyone can write these builders to target any modern stack of technology or framework. It's very universal in nature. Any application that you've been able to write until now, you can introduce in our platform. We make it serverless, which means you only pay for as much as your application is using and there's never lock-in or any APIs that have to do specifically with the underlying infrastructure primitives.

So when you deploy to Now, you only really worry about your framework or application code and there's an API call that would say, "Oh, I'm talking to Azure." "Oh, I'm talking to Google Cloud," so that you can continue to evolve your code without having to learn all those specific things that could lock you in over a long-term.

[SPONSOR MESSAGE]

[00:19:47] JM: Kubernetes can be difficult. Container networking, storage, disaster recovery, these are issues that you would rather not have to figure out alone. Mesosphere's Kubernetes-as-a-service provides single click Kubernetes deployment with simple management, security features and high availability to make your Kubernetes deployments easy. You can find out more about Mesosphere's Kubernetes-as-a-service by going to softwareengineeringdaily.com/mesosphere.

Mesosphere's Kubernetes-as-a-service heals itself when it detects a problem with the state of the cluster. So you don't have to worry about your cluster going down, and they make it easy to install monitoring and logging and other tooling alongside your Kubernetes cluster. With one click install, there's additional tooling like Prometheus, Linkerd, Jenkins and any of the services in the service catalog. Mesosphere is built to make multi-cloud, hybrid-cloud and edge computing easier.

To find out how Mesosphere's Kubernetes-as-a-service can help you easily deploy Kubernetes, you can check out softwareengineeringdaily.com/mesosphere, and it would support Software Engineering Daily as well.

One reason I am a big fan of Mesosphere is that one of the founders, Ben Hindman, is one of the first people I interviewed about software engineering back when I was a host on Software Engineering Radio, and he was so good and so generous with his explanations of various distributed systems concepts, and this was back four or five years ago when some of the applied distributed systems material was a little more scant in the marketplace. It was harder to find information about distributed systems in production, and he was one of the people that was evangelizing it and talking about it and obviously building it in Apache Mesos. So I'm really happy to have Mesosphere as a sponsor, and if you want to check out Mesosphere and support Software Engineering Daily, go to softwareengineeringdaily.com/mesosphere.

[INTERVIEW CONTINUED]

[00:22:07] JM: The idea of ZEIT, of this higher level cloud provider, this is something that was first pursued by Heroku, and Heroku has done a great job at it. Although you have just articulated a solution to one of my biggest beefs with Heroku, which is that every month I get a bill for a bunch of Heroku projects that I keep running, because I like my tinkering and my projects that nobody uses, because some day like I'll make them useful. I'll make them popular and whatnot. But until then I pay whatever, 20 bucks a month for a node or something, because it's not "serverless."

One of the features of serverless is that it spins down when you're not using it and you get to just pay for what you actually use. Tell me more about how you accomplish that.

[00:23:07] GR: That's great insight. Heroku indeed did a great job for what today we could call serverfull, I suppose, which is when you deploy to Heroku as the server interface. It's not even just managing the servers as machines, but the interface that you give them and they give you is sort of, "I'm going to run a process and I'm going to put a server inside," and that server can accumulate towering complexity. Like it just gets bigger and bigger and bigger.

What that hurts you over the long-term is with this idea of spinning down and spinning backup really quickly, which is what you just mentioned. The best feature of serverless in my opinion is twofold. One is the scale to zero and scale to infinity aspect. So if your project is not being accessed, it just adds zero copies of it. And then for every request that happens concurrently, more copies of it are issued.

So if you have a traffic spike or your website indeed becomes very popular, that it's handled very sort of by design. One of the things that enables this is that when you go from zero to one, it has to be very, very fast, and that's where the Now platform guides you a lot. I mentioned this idea of the builders and you say, "Hey, I want to use Node.js." We do a lot of work under the hood to make sure that when you're spinning up, it's instant.

So the end user, once again, we're always thinking about this end user, it doesn't have to wait around for your application. The beauty of it is that you end up paying only for those hundred milliseconds intervals of compute that are actually being utilized. Think about this from the development process perspective.

When developers are testing and building and branching off and collaborating on software, they're constantly going to be making deployments. As I said, we make one for every push. So of all of those, we allow you to go back in time and click on any of those links and see what your application was like before. We allow you to do real-time reviews. But then overtime all those deployments just wind down and don't cost you anything. I think that's one of the most remarkable features, because serverless is also helping not just lower your cloud spend. Not just make you more productive, but it's enabling entire categories of collaboration opportunities that really were impossible before.

I think the objection before would have been, "Oh my god! I'm going to have so many versions of my application running that I can literally not even imagine this is possible." So that's kind of the most exciting thing about serverless to me. In addition to that, I think there's this idea that you've probably have heard of the idea of functions as supposed to servers, and you've probably heard of function as a service and so on.

The interesting thing about serverless until Now came along was it had to rethink everything about how you wrote applications. So before you were saying, "Oh, I'm going to spin up a node server." Then you have to go and like learn about this Lambda API, for example. Those two things looked really, really different. It's almost like how to like rethink everything that you knew about software engineering at some point.

However, what we realized, and you mentioned this idea of the high-level, is that we could compile down to this function primitives. So we can give you all the benefits of serverless without having to reengineer to much or relearn everything from scratch. So when we give you support from Node.js or even PHP, which we support, it looks exactly like it does when you go to Node.js.org or PHP.net. So we're really not in the business of inventing new APIs here. We're in the business of abstracting out those really innovative technologies and you really don't have to worry about them too much.

[00:27:18] JM: Let's give a concrete example here. So let's say I'm building a photo sharing app, like an Instagram and I've got a mobile client that's just a thin client for taking pictures and looking at my feed of pictures. Then I've got a backend that accepts those pictures and does stuff like writing it to a database and maintains the user accounts and followers and stuff. Then you of course got a web client that is somewhat similar to the mobile client.

Tell me about the difference between what this deployment would look like in the serverfull world versus in the ZEIT serverless world.

[00:28:03] GR: Love that question. The main and most important thing is that the question that I was asked is like, “Okay, what’s the entry point to my application?” In the serverfull world, it was going to be a file or process. Let’s say I’m using Node.js or Go. It’s going to be server.go or server.js. When it gets built for production, that one unit is going to contain the information about every single aspect of your application, every single API entry point, every single version of every API endpoint. Even API endpoints that don’t get accessed by users anymore end up all in that one unit. We can also call that a container if we wanted to.

So overtime, as you add more API endpoints and more logic to your application, that is going to just sort of grow infinitely and there’s nothing stopping you. As teams and people continue to collaborate on, they just add more files and more dependencies and so on. Node.js developer might be familiar with projects where their package.json file contains a hundred dependencies. Then when they run yarn or npm install, they just wait and wait and wait until this one server sort of comes to life. Then you start having issues with like boot up time or you run Nodeserver.js and you wait a long time, or your compilation times even get very, very long because you just have everything in one. It’s like really put your API endpoint X in one basket the server.

With Now, because we used to write software this way and we learn about all its pitfalls. With Now, we took a dramatically different approach. So what we say first fall is that we embrace this idea of the monorepo. So you create one repository where your, for example, API project is going to live, or even your API and your frontend project are going to live. I would start by creating a folder called API, and then inside that I would create subfolders or sub-files for my API endpoints, like images.go for my images reception API endpoint. I would have sessions, users, and all my API endpoints defined there.

On another side I would create a folder, for example, for my web frontend and I would call it web or www, and inside that I would put all the files related to like, let’s say I picked View to use to create my frontend. So I create my app.view file, my [inaudible 00:30:45] JSON for the dependencies of that and so on. Then in my Now JSON file, which is how you can figure your deployments you say, “For this API when I use GO, for this API when I use Node, and for my frontend when I use View.” When you deploy, this is the interesting thing that happens. When you run Now or when you run Now dev in the future, all these builds are going to happen concurrently. So we're never again putting all these eggs in one basket. We literally use the cloud to sort of parallelize a process of compiling all of these separate things. Because as I said earlier from an abstraction perspective, they are separate and independent. But to you they feel like a cohesive thing.

When you get your Now URL back from the deployment process, you can still access everything as a cohesive interface. We can go to your deployment URL and consume your frontend. You can go to /api/files and you get access to that file's entry point that was written in Go. But as we can see, we sort of give you the experience of a monolithic application. But under the hood, we're outputting all these different artifacts that can scale independently.

For example, files of Go ends up becoming a Go function, a serverless function that can scale to zero, scale to infinity, charge you per hundred millisecond. Your www frontend becomes a set of static asset, for example, that gets served directly from CDN. I can even create serverless functions that do dual rendering. They execute on the backend and they can execute in the front. You were mentioning, for example, I want to create a native application. There's a very exciting new trend that is being pioneered by Airbnb and others of server rendering native applications, where you go to an entry point and it outputs some markup-like structure that a mobile application can rehydrate. By doing so, bypass the slow App Store rollout review process and iterate more frequently and fix bugs more frequently.

So the benefit of this is that now I have my monorepo, I have a global view of the entire evolution of my project. I have my APIs. I have my web frontends. I have my native frontends. They can share assets between them that are shared at compile time, and then all the artifacts that they output are actually independently scalable and have a limited surface for error. If there's something wrong with one endpoint, it doesn't bring down the rest.

Going back to the serverfull world, if something happened to your dyno, everything crumbles within. A bad rollout where you messed up a syntax error, for example. Let's say that you somehow let that sneak in, and it happens very frequently in our world into one endpoint, it would only impact that artifact. So it's not even just that you pay less money or that you're more productive because you have this like single place where all your changes go. The beauty of this is also containing error, because we're not perfect and errors always happen. For a lot of outages that companies experience, we can always trace them back to human error. So this different way of not putting all your eggs in that one server basket sort of solves a lot of problems that we've seen over the past few years in the deployment space.

[00:34:21] JM: You mentioned in there the idea of the monorepo. This might be new to some people, but the monorepo is actually how Facebook manages much of their codebase. It's how Google manages much of their codebase. Give a bit of an overview for what the idea of the monorepo means and how monorepo has been used at successful companies.

[00:34:45] GR: It's interesting, because monorepo in some way is very old. If you think about the creation of Git as a repository, right? As a repository for the Linux Kernel, the entire Linux Kernel project with every single module whether static or dynamic or the Linux Kernel is one repository. What happened was overtime, I think we started as engineers trying to figure out, "Okay, what are the best ways to organize our codebases? What are the best ways to iterate it on our projects?" and so on. With systems like GitHub, we got this infinite power of great repositories and it became very cheap to spin up a repository, like just, "Oh, new project."

There're two paths forward here. Like if you're going to create your API, you could even have one repository for API endpoint. So we're talking about the file upload or image upload endpoint. In one universe I could go ahead and create API/images and have that be one repository with its own issues, with own history and so on.

In another universe I could what I described earlier. I can have just a folder with subfolders and it all goes into one repository where issues are tracked, where there's one cohesive deployment story where when a new developer joins your team, they clone that and then they run dev and then they're developing locally. Between this two universe is, "Okay. What are the pros and cons of each approach?" As you mentioned, the monorepo, the one where you have only one repository and you just use directories to organize your code, that's been a very successful strategy for companies like Facebook.

Similarly, Facebooka also thrived in using what I call the original serverless model, which is PHP, which is people only worried about the invocation lifecycle. What I mean by this is that repos comes in, code is executed, response goes out and they didn't worry about servers. So we think – And after having used both over many, many years, we think that the monorepo has very tremendous productivity benefits.

As I mentioned, one of them is there's a cohesive development story. You just clone the repo and there's one script or one instruction run to make everything work locally. Another one that's very interesting that I've come to realize over the years is like it when my code organization, and even my organization as a company reflect what the end user consumes. This could be an entry point, no pun intended, into understanding why the non-monorepo could be flawed on a technical level. We always have preferences and we have success stories, but I don't think many have tried to argue why one is technically superior to the other.

I could attempt to explain that in the sense that let's say that your Instagram clone that you just describes succeeds and you say, "I'm going to open my API to the public and it's going to be called softwareengineeringdailygram.com.api."

[00:38:03] JM: All right.

[00:38:04] GR: I'm going to have all my documentation and I'm going to tell my consumers that it's api.softwareengineeringdailygram.com. When the end user, whether it's a developer or your customer interfaces and experiences this API, they go to one place. They go to one place and then they add a path name and they're accessing a different endpoint.

So what you're giving your user and what you're giving your developer and what you're giving your customer actually has one cohesive shape. But then you arbitrarily decided that that shape was not going to be mirrored in your backend infrastructure and in your code organization. I think you have to have a good argument as to why you would do that, because when we achieve a symmetry between the output and its underlying backend organization, I think we maintain one cohesive mental model. We maintain one developer workflow model. Like you understand exactly with the same – When you go and edit your API documentation, you're tell them, "Hey, there're all these paths." But then when you go back instead of actually having paths, you have all these code separated into dozens of repositories.

I think monorepo really is the most natural and cohesive way of mirroring what the end user experience is with how you develop it, and in doing so I think you just make your life a lot easier. It's not that the other one doesn't work. I think we have to be very clear in this. I think one of the interesting things about our industry in 2018 is that we have the engineering resources in a lot of cases and we have the cloud power to make a lot of different things work.

You're going to go and meet organizations that made the multi-repo world work really well, but that doesn't mean that it's the optimal way of doing things, and that's why I think it's so important that we really reflect on, "Okay, what are the things that result in the less amount of indirection?" I think the monorepo accomplishes that really well.

[00:40:16] JM: If I'm a developer today, I may not have my app architected in the way that is meant for ZEIT, because I think in the ideal world, if I understand correctly, my different functions would be broken up into different files. Whereas most people today have their apps, I believe, architected in this singular Node.js file or a couple of files and it's just very big and monolithic. Are you encouraging people to re-architect their application for ZEIT or are you going after new greenfield applications and saying, "Hey, try out architecting your app this way and it's going to work well." What's the model for getting people to build their applications in this fashion?

[00:41:07] GR: That's a great question. First of all I want to say the market is already moving in the direction of the multi-entry point world and the cohesion between what the user access is ends up being one discreet entry point in your backend. I think the biggest testament to this – Or Next.js React framework has thrived and continues to grow so fast. If you go to nextjs.org and you look at the sizes of the companies and users that are using it, I think some of the biggest, funny enough, Chinese [inaudible 00:41:45] have recently began adapting it in a very large way, which means a huge percentage of internet users today are experiencing this new way of programming.

As I said, they end up experiencing it because they go to, for example, ebaychina/item, and that I know is going to map all the way down to pages/item.js, which is how Next.js tells you that you start your project. You define page entry points. You don't define a monolithic server.

The reason that we pioneered is with the web frontend, is that for the web it was very, very obvious that we didn't want to ship huge bundles of JS for an entire application. We wanted to break it down, like the famous code splitting and the famous bundling and web packs of the world. The idea here was like, "Hey, let's give to the end user only the code that was needed for that discreet page that they were in."

The example that I always like to use was at one point in time in every organization, people would always add an extra copy of jQuery to their project to ship into prod. It wasn't unusual for us to see that we would go to a website and, "Oh! It's working slow." "Oh! It's loading three copies of jQuery." The way that we saw this is, "Okay, as teams evolve an application, they will always add more and more complexity. How can we solve this for the end user? Okay, let's limit the amount of complexity that the user has to download to the specific section of the application that they're experiencing."

Now, what's interesting about serverless is that we're applying the same exact methodology but for the backend. I don't expect this to be very surprising or challenging for organization to adapt, because most likely they're already doing code splitting for the frontend. So serverless is just code splitting for the backend, and it makes perfect sense.

Funny enough, when I was sort of talking to customers and understanding, "Okay, why do you love this methodology of software so much and why has it made you so successful?" Customers like Nike, for example. The answer that I got was not all – Like it's superfast, super technology advanced, and super this or super that. What I would hear is when a developer opens a project, they can just naturally navigate through it.

Going back to server.js, when you go to server.js, you open up the file and you start having to learn all these abstractions that the developer cited overtime. You have to untangle middleware out of it. You have to learn about all these instantiation custom functions that someone wrote overtime. Maybe you have to scroll through 300 lines until you find your first API endpoint definition, and then maybe that requires another file in the file system that is an arbitrary location.

Whereas in this new world that we're proposing, you open the project, and mostly you'd find that a folder called API that, as I said earlier, it mirrors exactly, or perhaps with a few slight differences, how the end user experiences it. So our APIs, datacode/api/deployments has a folder called API and another folder called deployments inside it. I don't expect organizations to be very surprised by this. I think it's probably something that they've been eagerly waiting for for many years and it's finally becoming very, very easy to do.

As I said, some programming language communities have been enjoying this work, this way of reasoning about their programs for very long. Like people that have been doing PHP or similar technologies where the server abstraction is nginx or Apache and then the developer only worries about the code. They've been thinking this way for a longtime and they've been very successful with it as well. For the companies that have already written their server.js and want to like start becoming serverless in a way that is more incremental.

We also support wrapping their servers into serverless function. Now, there's going to be caveats. You're not going to enjoy all the benefits that I described earlier. Perhaps your cold boot up times are going to be a little slower. But it's a good way of making the first entry point into this world. So the way that we support this is you bring in your server and we at the build step you say, "I want to use now/node/server and point it to server.js," and then you're going to get a singular serverless function built that exposes your server. So it's a good way of even learning about what are the downsides of writing servers in this new world, because it might not scale as well overtime, but you still get a lot of the benefits, like paying for every hundred milliseconds of compute. You start learning about this new way of deploying. New endpoint is going to be created serverlessly while legacy ones are shipped through the server. That actually works extremely well.

Let's say that before we have this conversation today, you already started on softwareengineeringdailygram and you had an expressed server, for example, and it covered a few API endpoints. You can totally deploy that, and then all your new API endpoints can be singular function entry points and then you're going to get all the other benefits and migrate it as sort of incrementally overtime.

So there're a lot of options for that. If you go to zeit.co/docs, we have plenty of documentation how to approach this world of either going serverless and sort of learning about this new ways, or I have an existing codebase. What do I do? Or even from those that are more future-oriented, we have an example that we call the monorepo where we even mix and match several different programming languages for each API endpoints. We have an API endpoint defined in Python, one in Go and one in Node.js. So you can sort of see how regardless of all these different technologies being involved, the deployment strategy is always the same. It's Now. You either push a GitHub or you run the Now command and you get a URL back.

[SPONSOR MESSAGE]

[00:48:13] JM: How do you know what it's like to use your product? You're the creator of your product. So it's very hard to put yourself in the shoes of the average user. You can talk to your users. You can also mine and analyze data, but really understanding that experience is hard. Trying to put yourself in the shoes of your user is hard.

FullStory allows you to record and reproduce real user experiences on your site. You can finally know your user's experience by seeing what they see on their side of the screen. FullStory is instant replay for your website. It's the power to support customers without the back and forth, to troubleshoot bugs in your software without guessing. It allows you drive product engagement by seeing literally what works and what doesn't for actual users on your site.

FullStory is offering a free one month trial at fullstory.com/sedaily for Software Engineering Daily listeners. This free trial doubles the regular 14-day trial available from fullstory.com. Go to fullstory.com/sedaily to get this one month trial and it allows you to test the search and session replay rom FullStory. You can also try out FullStory's mini integrations with Gira, Bugsnag, Trello, Intercom. It's a fully integrated system. FullStory's value will become clear the second that you find a user who failed to convert because of some obscure bug. You'll be able to see precisely what errors occurred as well as the stack traces, the browser configurations, the geo, the IP, other useful details that are necessary not only to fix the bug, but to scope how many other people were impacted by that bug.

Get to know your users with FullStory. Go to fullstory.com/sedaily to activate your free one month trial. Thank you to FullStory.

[INTERVIEW CONTINUED]

[00:50:35] JM: Something worth pointing out that's kind of cool is that in today's world you can build a cloud provider on top of pervious cloud providers and your differentiator can be the design decisions and the developer experience that you're offering to the developers, because that's what differentiates ZEIT. I mean, there are – Obviously, other cloud providers are doing serverless and they have a wide range of options and you could couple together something that would feel like ZEIT on other cloud providers, but you are giving an opinionated view.

So let's go into that ZEIT architecture. Take me inside what ZEIT is built on and how that architecture looks.

[00:51:24] GR: That's one of my favorite things to talk about really, because I've been watching the virtualization in cloud space for many years. We started with, "I'm going to have a physical computer and I'm going to have a server there," and then we introduced this idea that actually it's better to model it as a VM. Even if I'm going to use the entirety of the hardware's resources, it's better to model it as a VM, because it gives me all the portability benefits and it allows me to swap the operating system that I'm using. I'm not incurred to the hardware decisions and it allows me to operate the hardware without changing my VM and my software configuration.

I think what we're seeing now is that the same is happening to cloud providers in a way where what the VM did for hardware, we're doing to cloud infrastructure, because even today ZEIT is able to compile down to the primitives of any cloud provider. We use Google Cloud. We use AWS. We use Azure, and the way you experience it is, as I said, like you were operating on a software level and the cloud providers are the hardware, and you don't want to mess with hardware. You want to focus on delivering and creating value for your customers. So that's one of the most important principles here, because the way that Now is architected allows you to startle across cloud providers. We make the decisions on your behalf what's the best primitive to use for your needs from all these wealth of options. We've already even migrated on behalf of our users from certain initial decisions of one cloud provider to another one with zero downtime and saving our customers' money, saving them time, saving them energy and getting the best performance.

Once again, it's like you were writing your cloud VM, like what EC2 did to hardware, and then we upgraded the machine to be a more powerful Intel processor. But it's even more powerful than that. It's not even just about performance. In a lot of cases, it's about the regulatory landscape. Now can go to places where no one has been, and our plan is to go and host the Now platform with Alibaba Cloud. Go to regions in India and China that are just not available uniformly across cloud providers.

If you choose Google, you're at Google peril to wait and go meet customers in a certain region that they are not in yet. So this is really the beauty of this higher level cloud provider as you mentioned, and there are benefits also in terms of lock-in and APIs, because in the build phase is when we create the artifacts that end up going to specific cloud providers. But what we give you, and going back to the analogies to the compiler world, when a compiler takes source code, a modern compiler will create an intermediate representation of your source code, and then that intermediate representation of your source code is what's used to generate the machine code and run your software. Now is acting as an intermediate representation.

So when you were orchestrating that monorepo with your APIs in Go and Node and Next.js and View and so on, the code is written as the APIs mandate for Node, like the View codebase and the Go documentation. At build time, we have to convert that into whatever suits each cloud provider the best to make it serverless and to make it the most optimal. That's even where you're not even locked in, because you just worry about, "Okay, I'm going to write a builder myself if I want to that changes how the code is written for the end developer." So there's never a situation where you say, "Oh, should I lock myself in?" and there's no escape. You can run the code locally. We're going to announce this soon. We're going to even allow you to package it and deploy it on-premise or on legacy containers if you wanted to.

So what we want to make clear here is that there's a new opportunity that has emerged to really take your code everywhere in the world despite the boundaries of the underlying infrastructure providers, despite what regions they exist in in a way where your code is absolutely controlled by you and only abides by these standard compliant APIs in standard compliant protocols of the web.

[00:56:11] JM: So take me through what happens when I deploy my app to ZEIT.

[00:56:16] GR: Great question. So let's talk about the most simple example that you could ever think of, which is an entry point into your app that renders Hello World. The way that I started is I write index.js, and inside I export a function. In Node.js you do like module.exports=function, and then it takes as parameters the request and response objects.

So for those that are familiar with Node.js, that's the exact same thing as when you would do HTTP create server and you would [inaudible 00:56:51] that function inside, it takes a request object, then a response object and then you can respond. Inside I would write rest. [inaudible 00:57:00] "Hello World" and respond with my Hello World.

So the difference here is that instead of creating a server abstraction, I said module.export. I'm going to export my handler, my request handler function. This is when I think serverless will start making a lot more sense for a lot of people, because we all hear about functions and the question is like, "Okay, what do these functions do?"

Arguably, these functions have already existed. Before we would pass that function as a parameter to create server. Nowadays, we can just say export and expose it. That's what allows your code to interoperate it with Now. The next step is in Now.JSON you say, "Built," and you say, "source," in [inaudible 00:57:46] js use Now/node. Now/node is actually just an npm module that is open source and published to the npm registry that will do all the heavy lifting of saying, "Oh! Have a node.js function. I want an actual serverless function output," which we call a Lambda, "to be generated from this."

For example, it will do the heavy lifting of packaging your code. For example, in addition to index.js in the future you can have a [inaudible 00:58:16] and you can have yarn.log and all that, and that's what this now/node builder is in charge of doing. The beauty of this is that we've created builders for all the most popular technologies and frameworks, but it's fully open source.

Within a week of the release of Now 2.0, which introduces concept of builders, someone had created one for OCaml, for example. Someone has extended as of yesterday our PHP one. There's a WordPress one being created, funny enough. So once you defined your now.JSON and your index.js file, you run Now. That's all you have to do. When you go to your deployment URL, the function will get executed. Notice that it called it index.js and then that was so that when I go to slash, it executes my function.

What's neat about this is that it allows for this like file system based reasoning. Now say that I want to create a more sophisticated API and I don't want to have a mess of files in my root directory. All I have to do is I create an API folder. I create a users folder inside and I put my index.js file inside it once again. I say, "I want that to use now/node in my now.JSON file," and that's it. When you deploy, you're going to see that – I don't know if you remember from the old days of Apache that what would happen in that deployment is that you get a directory listing output. It's going to say, "Index off," because there's no index to render and you're going to see an API folder. You're going to click on that API folder and you're going to see your users folder, and that folder is going to have a special little icon, which is the Lambda icon, which represents a function in this branch of mathematics of Lambda calculus. That's Lambda function, instead of just being a static file, it's almost like this special executable file. When you go to it, it's going to execute code.

The beauty about serverless in my mind is that it really allows for this like file-based reasoning. So it's almost like when you put a photo in a CDN, photo.jpeg, and the user consumes it and the browser is in charge or decoding the photo and rendering it as a photo. Now that's happening is that you can output this special Lambda files that we call them, and then when you go to them, they execute code in the cloud nearest to where you're located. It's a really, really powerful concept. The idea that we can mix and match static files and these files that execute code on demand and that it's extremely cheap to actually execute them and that they scale infinitely, just like photo.jpeg scales infinitely in a CDN and that you never worry about – One of the interesting things here is that I always mention to the people that I talk to, my friends, when I explain this is when you would put a file into a CDN, you would never set up monitoring or paging. You never be awoken in the middle of the night that your photo is down. That's just a concept that is completely alien, "Oh! My asset in a CDN is down, because the server is down, or the cluster is down, or Kubernetes is having an outage, or Kubernetes was hacked, or whatever." That's just not a concept that even goes through your mind. We're doing the same for code execution, because that little Lambda file that can be placed in the CDN doesn't have the room for failure. Even if a server goes down, that little file can be obtained by another server and be executed on demand. Even if the file crashes halfway through execution, it can trivially retried. It can be redeployed in other regions if an entire region goes down.

So just imagine a world five years from now when everyone in the world is deploying code with the confidence of that CDN serving a file. I just think that we cannot even measure yet what this is going to do for the world of deploying code throughout the world and really what it's going to do for like us as developers really not worrying as much.

For some of us that have been so minded about performance and stability and response times overtime to not have to worry about this, to not have to be on PagerDuty to sort of respond to events that happens as a result of like our servers crashing and so on. It's just really, really transformational. The scalability to benefits don't even have to do just with like the computational scalability. This is something that I always like to emphasize. Computational scalability is by and large a solved problem. Even with server.js, I can put it inside a VM and it can spawn a thousand VMs and it's going to scale. There's no discussion that it's going to scale.

But the question is, is it going to scale at the team level? Is it going to allow people to continue to contribute to a project overtime without stepping on top of each other, without test taking forever to run, without boot up times getting out of hand, without build times being super, super long, without trading off the inability for every single push to be deployed instantly?

All these things just compound to economic effects that I'm just personally excited regardless of us having such a bet in this space as a company. As an individual, I'm just excited about seeing what this is going to do to the world five years from now because of this compounding effect. That might seem little. I think sometimes it is easy to underestimated them, because ultimately the difference between server.js and index.js is just a slight abstraction difference, a slight even naming convention, a slight directory structure reorganization. But the compounding effects are what gets me excited.

[01:04:32] JM: I share your excitement over the vision. There are a number of different companies who are pursuing this kind of vision. So you've got Netlify, Firebase to some extent. I think there's one or two others, and I think the wind that's at your back regardless of competitors is that there's just a growing number of developers. I think the developers who are coming online, so to speak today, are also very open-minded. They're flexible. They're willing to move from thing-to-thing. I feel like developers today are less prone to getting trapped in, "Oh, I'm a Java developer. That's what I'm going to do the rest of my life." I think there's a much more higher sense of mobility between roles.

Now that said, it's hard to develop mindshare around an idea, even a good idea in software engineering. There's some extent to which you have to do marketing. You have to reach developers in the right way. You have to evangelize. Now, some products do grow virally natural, and I think I saw I think it was earlier this week or last week, Spectrum, which was built with ZEIT got acquired, which I think was a really good show of support for your products and somebody has now built a fully-fledged business that got an exit on top of ZEIT. So congratulations there.

What's your plan for building a following, or is word of mouth working just fine for you?

[01:06:09] GR: Funny enough, the spectrum example is just an example of how well word of mouth works, because we met their founders originally through conferences where we're just talking about this new way of deploying and they were attracted to our platform and they set everything up on our platform without really any interaction or special sort of setup. They were just another success story of what can you do when you give people the right tools to build a customer-focused product?

So the business of spectrum is to connect the world through real-time chat communities and to basically capture the entire spectrum of communication online. So their business is not in configuring servers. That's what we do. That's what we expose for you. That's what we give you as a service. So really that's the focus of our business right now. That's the focus of our marketing. It's all about providing the customer-focus. When I mean customer, I mean the end user. The end user has to have an amazing experience on our platform, and a result you sort of work your way backwards and say, "Okay. How was this built? Why was this so successful?

Why is it working so well? Why is it so fast?" that's what we want to project as a product, as a service. That's what we day-in and day-out obsess about. We don't obsess about catching the latest trend. We don't obsess about measuring the number of stars in GitHub, even though our open source products are some of those – They're capturing the mindshare of pretty much every single developer in the world. But we live and die by that sort of customer-driven focus and being the host of excellent applications.

What you're going to see from us in the future is continuing to answer all these needs for production quality applications. So everything from measurement, to testing, to sort of giving you the full package so that the next WhatsApp is built on now. That's what we're going after. We're starting to see early signs of success in this strategy and we want to double down in this. Excellent application hosting is really what we stand for.

[01:08:42] JM: Guillermo Rauch, thanks for coming on the show. I really enjoyed talking to you and I appreciate the vision for ZEIT, and I think you're on point.

[01:08:49] GR: Thank you.

[END OF INTERVIEW]

[01:08:53] JM: This podcast is brought to you by wix.com. Build your website quickly with Wix. Wix code unites design features with advanced code capabilities, so you can build data-driven websites and professional web apps very quickly. You can store and manage unlimited data, you can create hundreds of dynamic pages, you can add repeating layouts, make custom forms, call external APIs and take full control of your sites functionality using Wix Code APIs and your own JavaScript. You don't need HTML or CSS.

With Wix codes, built-in database and IDE, you've got one click deployment that instantly updates all the content on your site and everything is SEO friendly. What about security and hosting and maintenance? Wix has you covered, so you can spend more time focusing on yourself and your clients.

If you're not a developer, it's not a problem. There's plenty that you can do without writing a lot of code, although of course if you are a developer, then you can do much more. You can explore all the resources on the Wix Code’s site to learn more about web development wherever you are in your developer career. You can discover video tutorials, articles, code snippets, API references and a lively forum where you can get advanced tips from Wix Code experts.

Check it out for yourself at wicks.com/sed. That's wix.com/sed. You can get 10% off your premium plan while developing a website quickly for the web. To get that 10% off the premium plan and support Software Engineering Daily, go to wix.com/sed and see what you can do with Wix Code today.

[END]