EPISODE 333

[INTRODUCTION]

[0:00:00.2] JM: Spring Framework is an application framework for Java and JVM languages. Spring was originally built around dependency injection, but it grew to be an entire ecosystem of tools and plugins for Java developers. Spring was originally released 15 years ago and since then a lot has changed around application development.

For example, many engineers today deploy their applications to the cloud in microservices architectures. The expectation around frameworks has also changed with the rise of Django, Ruby on Rails, and Node.js since Spring's rise.

Spring Boot takes an opinionated view of building production-ready Spring applications. By taking this opinionated view, Spring Boot gets engineers up and running faster than the traditional Spring. Josh Long is a Spring Developer Advocate at Pivotal and he joins the show to discuss Spring Boot and the history of the Spring Framework.

Software Engineering Daily is having our third meet up; Wednesday, May 3rd at Galvanize in San Francisco. The theme of this meet up is fraud and risk in software. We're going to have some great food, some engaging speakers, and a friendly, intellectual atmosphere. To find out more, you can go to softwareengineeringdaily.com/meetup. We would love to see you there. We'd also love to get your feedback on Software Engineering Daily. You can fill out the listener's survey available on softwareengineeringdaily.com/survey. It would be really helpful to us, and let's get on with this episode.

[SPONSOR MESSAGE]

[0:01:46.2] JM: Spring is a season of growth and change. Have you been thinking you'd be happier at a new job? If you're dreaming about a new job and have been waiting for the right time to make a move, go to hire.com/sedaily today.

Hired makes finding work enjoyable. Hired uses an algorithmic job-matching tool in combination with a talent advocate who will walk you through the process of finding a better job. Maybe you want more flexible hours, or more money, or remote work. Maybe you work at Zillow, or Squarespace, or Postmates, or some of the other top technology companies that are desperately looking for engineers on Hired. You and your skills are in high demand. You listen to a software engineering podcast in your spare time, so you're clearly passionate about technology.

Check out hired.com/sedaily to get a special offer for Software Engineering Daily listeners. A $600 signing bonus from Hired when you find that great job that gives you the respect and the salary that you deserve as a talented engineer. I love Hired because it puts you in charge.

Go to hired.com/sedaily, and thanks to Hired for being a continued long-running sponsor of Software Engineering Daily.

[INTERVIEW]

[0:03:16.3] JM: John Long is a Spring Developer Advocate at Pivotal. Josh, welcome to Software Engineering Daily.

[0:03:21.0] JL: Thank you. Thank you so much for having me.

[0:03:23.2] JM: Let's start off by talking about the Spring Framework. For people who have no idea what Spring is, what is the Spring Framework?

[0:03:29.5] JL: The Spring Framework originated in 2002, I guess, is kind of my earliest incarnations people know about it. It serves the goal of helping people write code without worrying too much about the middleware with which that code is interacting. It did that by letting people use dependency injection, which it's a fancy way of saying that you write code to interfaces as supposed to — Because you have interfaces, you can plug in implementations. You can swap out implementations. You write code against the data source as supposed to writing code that looks up at data source inside of JNDI, that kind of thing.

That indirection makes it possible for Spring to do all sorts of cool stuff for you, because now that you've got this data source or any other sort of collaborating object, you can dynamically subclass the type that you've provided or that you're collaborating with and you can do aspect oriented programming, you can provide references to transaction management. Rather, you can interpose transaction management, that kind of stuff. All sort of transparent to the component that's relying upon this dependency. We call that dependency injection.

That's what Spring started off as, right? But it turns out when you encourage people to write code that's real clean and sort of object-oriented and it doesn't have all these tight couplings to dependencies, then you promote unit testing. You promote ease of evolution. You make it easy to move a code from one environment to another, so you get portability. That's sort of the original, I think, the motivations for people moving to Spring more than 15 years ago.

Since then, it's become a whole ecosystem, and we can talk about that if you want, but I'm not sure if that's your question.

[0:05:04.0] JM: We'll get there. This notion of dependency injection where you can specify a data source and you have a looser coupling to what the actual end data source is, whether it's MySQL or Postgres or Cassandra, you have some intermediate middleware that is taking care of translating the format of the database to format of the framework, the end memory framework, the Java program that's running. Is that accurate?

[0:05:39.5] JL: Not so much.

[0:05:40.1] JM: Not so much.

[0:05:40.5] JL: What you're talking about is more of a framework I guess. No. I supposed you're talking about an adopter. What we are talking about, when I'm talking about a data source, I mean suppose you have specifically a pointer to javax.sql.datasource and what Spring makes it easy to do is to work with a, let's say, in your prototype case, you want to work with something like a H2, right? In-memory embedded, but still SQL database that complies with the javax.sql.datasource, the contract.

Tomorrow — Again, when I say tomorrow, let's imagine more in the hypothetical situation, 15 years ago. Tomorrow, I may decide I want to move that application that is using my little in-memory database and I want to point to Oracle which is bound into a JNDI context. They're both a javax.sql.datasource. This not about isolating — This is not about abstracting away unnecessarily the fact that using a SQL database versus a Cassandra versus a MongoDB. This is about locational decoupling.

The reference to the thing I'm using, it still a SQL data source. It's just that I, in my code, don't have an explicit requirement that that data source be the production instance that we've got in production. I can actually swap out a mock in my test. I can use in-memory one in my local development environment, et cetera, et cetera, et cetera.

At a higher level there's other things that allow you to talk to other data sources like Mongo and Cassandra. Again, that's an ecosystem discussion.

[0:07:02.5] JM: Back when I was a programmer, all of the jobs that I had were Java programs that involved Spring and I never understood Spring. I could never understand it. Literally, all of my software engineering work was in this environment. What are some of the aspects of Spring that people have historically disliked or not understood?

[0:07:29.3] JL: Good question. The first thing is to understand that what I just described, this idea of dependency injection, is the least significant or least relevant part of it. If you're using Spring just for dependency injection, you're missing out. What has happened is people have built frameworks on top of Spring, on top of that basic component model. This idea that you've got a single place where your objects are wired together and they are free to work and to function without any coupling to the other components; these collaborating objects.

We have technology that serve all sorts of use cases, verticals if you will. If you want to build a web app as MVC, there's Spring MVC. If you want to do batch processing, there's Spring Batch, or integration. For new microservices, there's Spring Cloud, et cetera, et cetera, et cetera. The thing is that there's this whole ecosystem of technologies. The dependency injection is just sort of how you get there. I think, today, in 2017, we just take for granted that, of course, that code is going to be written in such a way as to not be coupled to the sort of specific reference. I'm not going to write code that is today coupled to the IP address for my production database. That makes no sense. I would externalize that configuration. I'd have a little bit of indirection there.

As to things that people don't like, I think perhaps one thing that a lot of people soured on eventually was XML. XML was the way that we helped you wire up the objects in your application. Remember, Spring originated — It came from many years before Java 5. What we needed was some way to tell Spring, "Here are the objects in my application. Here is how they're wired together. Here are their relationships."

If Spring knows about that wiring, then it can do things. As I say, it can interpose itself in between you and your collaborator and provide services for those collaborating objects.

The only way could sort of describe that relationship in the beginning was to use XML. We didn't have the ability to annotate your code with Java annotations in Java 5. A lot of people, when they think of Spring, circa 2005 or circa 2004, they imagine that it's only XML, which it hasn't been true since 2006, but nonetheless, there are still some people who are, I think, suffering from that idea.

The other thing that I think a lot of people struggle with, not a lot, but certainly enough that I'm going to mention it here, is that there is, as I say, a large ecosystem. There's a lot of different things you could choose to use in the Spring ecosystem. Maybe you've got a specific use case, you want to build a web application. Then, by golly, use Spring MVC. If you're trying to do security, you can use Spring Security and you can layer that into other applications. If you want to do batch processing, or integration, or that kind of stuff, there's Spring Batch and Spring Innovation respectably. If you want to do data access with all these different sort of specialized data source, No SQL data source, there's Spring Data. There's just a lot of different options. If you're a new comer coming into the world of programming from university, for example, and you start to look at all these verticals and you think, "Okay. I've got to learn Spring, and they all start with Spring. Where do I even begin?" That can be, I have no doubt, a bit daunting.

Sort of the main complaint that we used to get is, "How do I get started?" For that, we have Spring Boot. Spring Boot hopes to at least alleviate anything that — Any kind of friction risk associated with the actual getting an application to the point where you can write code that is business differentiating.

That still doesn't alleviate the requirement that you understand what you're trying to do. You're trying to do batch processing. You're trying to do integration, et cetera. What it does do is get sort of all the configuration, all those sort of boilerplate. I think those are the two big things that people have historically had trouble with.

[0:11:00.0] JM: Right. There have been a lot of changes to application developments since Spring was the framework to rule them all. There have been new frameworks that have come up. There have been, obviously, the move to the cloud has occurred while Spring has been the predominant framework. As those things have changed, Spring has made changes as well and those changes are probably best epitomized by Spring Boot, which is the project that you mentioned. What is Spring Boot?

[0:11:34.0] JL: Spring Boot at the simplest level, it is still Spring. It's just an opinionated approach, an opinionated start on top of the Spring Ecosystem and the — By Spring ecosystem, I really do mean all the things that people use with Spring, not just Spring proper. Not just the Spring projects, right?

The way we provide that opinion is by providing something called auto configuration, which is this component model that works very much like what you would have done before. Remember, 15 years ago, you would have told Spring about your objects and they would have all come together at one time and they formed some sort of useful thing. There would have been a web framework, or whatever.

There's still that. There's still this idea of objects being contributed to the runtime, sort of final state of the application. What's slightly different is that now we can make it so that those objects are dynamic. They get contributed based on conditions that can be evaluated. You can say, "Okay. If this class is on the class path, then activate this component. Create this object and contribute it to the object graph. Wire it into the collective thing."

We can say if this library is here, or if an object of this type already exists, then don't bother. Otherwise, do bother. For example, if there's a library in the class path but nobody is already explicitly defined bin, then Spring Boot will create something for you. That dynamic programming model means that we can provide what we call starters, and these starters are just integrations with all these different ecosystem projects.

If you wanted to do metrics, or if you wanted to security, or innovation, or batch, or NoSQL, if you want to messaging, if you want to do thin grid, if you want to use Facebook, or Twitter, or LinkedIn, or designing with, or just act as a client to some sort of arbitrary OAuth endpoint, or if you want to talk to a statsD sort of a time series database, whatever your use case. It's as simple as adding what we call a starter, which in turn just brings in libraries, which in turn light up or activate these tests that I was telling you about. These aforementioned test that say, "Okay. This library is here. If there's, then go ahead and wire up the default useful sort of set of objects that we need to build to talk to this thing."

If you have Spring Boot starter RabbitMQ on the class path, that's a dependency, it's a maven dependency. You can use gringo. It doesn't have to be maven, but it's a dependency if you add that on to the class path. That activates the — It contributes the auto configuration, the defaults configuration for RabbitMQ which in turn points to the local RabbitMQ instance and so on. It gives you a transaction. It gives you a RabbitMQ template that you can use to talk to it. It gives you a default connection factory pointing to RabbitMQ. All that work that you would have done to use RabbitMQ is just done for you. If you, at any point, want to override the defaults, it's just simple as defining your own bean or the right type and that will be preferred.

[SPONSOR MESSAGE]

[0:14:26.2] JM: For more than 30 years, DNS has been one of the fundamental protocols of the internet. Yet, despite its accepted importance, it has never quite gotten the due that it deserves. Today's dynamic applications, hybrid clouds and volatile internet, demand that you rethink the strategic value and importance of your DNS choices.

Oracle Dyn provides DNS that is as dynamic and intelligent as your applications. Dyn DNS gets your users to the right cloud service, the right CDN, or the right datacenter using intelligent response to steer traffic based on business policies as well as real time internet conditions, like the security and the performance of the network path.

Dyn maps all internet pathways every 24 seconds via more than 500 million traceroutes. This is the equivalent of seven light years of distance, or 1.7 billion times around the circumference of the earth. With over 10 years of experience supporting the likes of Netflix, Twitter, Zappos, Etsy, and Salesforce, Dyn can scale to meet the demand of the largest web applications.

Get started with a free 30-day trial for your application by going to dyn.com/sedaily. After the free trial, Dyn's developer plans start at just $7 a month for world-class DNS. Rethink DNS, go to dyn.com/sedaily to learn more and get your free trial of Dyn DNS.

[INTERVIEW CONTINUED]

[0:16:24.8] JM: It makes sense to have an opinionated framework in a world where Spring has become a sprawling landscape and marketplace of different options you can choose from, and you covered some of those opinionated views. I think this is similar to — It's almost like a Ruby on Rails on top of the Spring Ecosystem. Ruby on Rails is opinionated and they don't apologize for the opinionated position.

[0:16:54.1] JL: If I may?

[0:16:54.8] JM: Please.

[0:16:55.5] JL: Yes, absolutely. We absolutely — The Java ecosystem, and Spring especially, we all owe a great debt to technologies like Ruby on Rails, which contributes to this convention over configuration idea. We actually had a previous convention over configuration technology called Spring Roo, which is I think more closely related to Ruby on Rails. It was code generation. The result was that you had code that was generated for you. The result, I think, for the use cases that we had when Ruby on Rails came out, I think that approach was certainly valid.

You mentioned that appropriate to have an opinionated approach on top of Spring. It's not just spring, it's just the number of things that a developer in Java is supposed to do these days has just grown just so much more dramatically. It is asking a lot to have a Java developer be good at all these different things, to remember all the nuances of all these different things. Whereas before, there's only a couple of things, only a few things you were likely going to do. You're going to build a web app, maybe babysit it, talk to a database or whatever, "What kind of database?" "Oh, some sort of SQL database?" That set of things that your average developer had to deal with has just grown. I don't know that there is any kind of common case application today.

The way we do this, is this is at runtime. The object graph is not code generated. Unlike Ruby on Rails where people have historically sort of struggled to undo the assumptions and the opinionated framework, you can override or undo the assumptions with Spring Boot. As I said before, it's as easy as providing — Configuring your own object of a certain type. We call that a bean in Spring. It's just an object that you tell Spring about of a certain type, and if that type matches one of the default objects that Spring Boot tried to provide for you, then your gets preferred and the other one that Spring Boot would have provided for you, it's dropped. Its creation is short-circuited.

The result is that you can now undo, because it's a framework. It implements the open-close principle. It's open for extension, but closed for modification. Now, obviously, it's not closed for modification, it's open-source, so you can do whatever you want to it. The idea is that you don't have to fork or recompile Spring itself to change its behavior. You just plug in objects of a certain type and they get plugged in to the machine, cognitive machine.

[0:19:04.7] JM: Let's imagine I am working at, let's say, a giant enterprise. Typical giant enterprise that has a big — Just Spring applications internally. I think of —

[0:19:16.6] JL: Like Netflix, or Alibaba?

[0:19:18.0] JM: Sure. Netflix, Alibaba, eBay, Amazon, all the companies I've worked at and many of the companies that I've interviewed on this show. Let's say there's somebody at this organization that wants to stand up a new service, or they just want to do some sort of experiment and they use Spring Boot, and they stand up their brand new service, their brand new app, and it gets to a scale where they want to integrate it with the preexisting, some preexisting Spring monolith. What's the pattern for integrating a Spring Boot app with a preexisting spring enterprise monolith?

[0:19:54.0] JL: It depends. You mean in the same running process, or do you mean as sort of a satellite service to which they communicate over a network process?

[0:20:01.6] JM: Look, whatever is the typical — What is the typical integration pattern you see.

[0:20:05.2] JL: What we see is that we have a lot of people — We know a lot of organizations are coming at this from the world of the monolith. They have an existing application. For them, if they're happy in that world, that's fine. They should absolutely — There's no reason — For example, if you have a small team and you're four people and you've just got this application that's served you well for 10 years, or whatever, and it's sort of just chugging along. It's not going to ever need more than a few developers. Maybe it's even on maintenance mode, or whatever. Then, maybe it's simple enough to just move your existing application and move it to the world of Spring Boot and keep it a monolith.

What we care about at Pivotal is agility, and I don't mean sort of — I'm not trying to speak specifically to —

[0:20:48.7] JM: Little A-agile.

[0:20:50.0] JL: Yeah, little A-agile. Thank you. Right. Exactly. What we care about is that we care about getting results to production. If you've got an application and you're able to update it as fast as the business needs, then this newfangled microservices architecture isn't for you and maybe it's simple enough to move your old application to just move it to Spring Boot.

In that case, it's simple enough, because your old application is a Spring application and your Spring Boot application is a Spring application. What I like to do, and this is what I tell the people I talk to; add the Spring Boot application annotation to your codebase. Bring in the right library set up main build accordingly, or if you want to shortcut that step, you can go to start.spring.ao and code generate just the preliminary maven build and the directory structure, and then you can use that as a queue.

Anyway, however you get that build setup, add Spring Boot application, the annotation, to your existing Spring application somewhere and then start up and start — Once it's working, start removing code. Find out the components that Spring Boot as defaults for, and then remove the code that you've already written. Often as not, it gets to the point where you have just your business logic, your particular components, your servlet components, maybe your MVC application controllers, or your services, or your repositories, or whatever, and nothing else. Spring Boot should, hopefully, if we've done our job right, provide default auto-configurations for the very large majority of the infrastructure code that you may have had to configure before.

I like to do this when I have some music, maybe get some alcohol, take the night off. It's a good night. The next thing you know, we've got a Spring Boot application half the code, which is — Well, I don't know how small your application would have to be to move half the code. You can have very little sort of cognitive overhead. What is left should be just your business logic, the things that you really care.

[0:22:37.3] JM: What's appealing about Spring Boot, or one of the things that's appealing is it has these developer ergonomics built into it that you don't have with the core Spring, or the old world Spring, the things that send shudders down my spine, like the XML configuration. It looks at Groovy as the desirable language that we should be building our apps around. Could you talk about some of the developer ergonomics that are built into Spring Boot and why you've made those decisions around them?

[0:23:10.6] JL: Sure. By the way, when you say you, I expect you mean the Spring team of which I'm just one tiny — Almost insignificant member.

[0:23:15.4] JM: Yes, that's right. That's right.

[0:23:16.9] JL: Yeah. That team, by the way, is just — They're at spring.io/team. They're enumerable and they're awesome. I am just the loudest clown on the team. Yeah, some of the things that motivated us in building Spring Boot — Developer ergonomics. I like that. I'm going to — Do you mind if I borrow that?

[0:23:32.6] JM: Please do.

[0:23:34.4] JL: The developer ergonomics were motivated by, as you know, this sort of paradigm shift, this move to the cloud, and this need to maximize productivity and to reduce the cognitive load. There's a lot of things that we support in the framework itself. We've seen — By the way, we're not the first. Let's be very clear. We're not the first in a lot of cases to support these things. It's just that we integrated them, because they are good ideas and we're very — We try to come by it very honestly.

For example, one use case that has seem, or seemed rather, sort of prolific, was that people wanted to remove the possibility of drift. They wanted to certify that their code that worked on their machines can be the same thing that works in production, and so they wanted to remove the drift. A lot of organizations, especially the high performing organizations, would take their applications and they deploy it to something like a Tomcat, or Apache Tomcat, or a GEDI, or whatever. They keep that application server binary, the distribution itself in the codebase.

The application server configuration lives with the code. It evolves with the code. It moves through development Q&A staging and it went through the continuous integration environment, et cetera, all in one sort of part and parcel thing. That tells us that a lot of developers want to guarantee that if it works in CI, then it will work in production. They don't want drift. They don't want to have to take their binary and to deploy to some other thing that is potentially running other things on it that hasn't been updated to the same releases and patches and so on. They've removed that, and that tells us that a lot of people treat these things as the same thing even though the traditional deployment models would have you keep them as two separate things; one dealt by developers, and one managed by operations. Now, of course, we've got DevOps, there's that unification. This ideas that both sides of the team are working on the same thing.

We see that manifest in terms of embedded application servers, embedded containers. Embedded Apache Tomcat, embedded GEDI, embedded Undertow, which for those of you who haven't heard about it, it's an amazing embedded servlet container from the folks at Red Hat working on the JBoss application server, or Wildfly, now it is. You can support all — Those are all supported by default out of the box. Of course, you can still deploy it to traditional servlet container environment if you like, but by default you get a public static void main application.

We also sought to really bake in sort of the best of breed patterns, patterns around building applications that are destined for the cloud. I think the folks at Heroku did an amazing job with the two-factor manifesto from 2008, 2009, a lot of great ideas in there. One that features prominently is this idea that configuration that changes from one environment to another thing is that should be different in each environment, like pass roads, and locators, and that kind of stuff. This configuration should live external to the application itself, outside of the binary, so you don't have to recompile. Spring Boot supports that kind of externalized configuration. It's a first-class thing.

What else? We've seen that a lot of people are doing this idea of having an application with a server-side generated web application all on the same binary. That's the business logic itself. That isn't really as common today. I think you'll agree, there's probably — It's far more likely you're going to be talking to a risk service. We've prioritized making REST and building services as supposed to building, let's say, JSP, or JSF applications, first-class that is, right? You can. You can do those things still, but it's not what we see a lot of developers doing. They're building client-side JavaScript. They're building android or iPhone clients and so on and talking to services in the backend. I think that underpins a lot of what we're doing in the ecosystem at large, and that's just sort of come together in Spring Boot.

[0:27:16.8] JM: Can you talk more about how Spring Boot interacts with Pivotal? You work at Pivotal. Pivotal owns Spring, or they are responsible for Spring — Maybe just talk about the relationship between Pivotal and Spring.

[0:27:31.2] JL: Let's see. Spring, it was an open-sourced, I supposed you'd say, when Rod Johnson, the creator of the technology released his first book, he wrote his first book — Whoa! I beg your pardon. That is awkward. Rob Johnson released his first book and that source code became pretty popular, and so it took on a life of its own as repository and source code, genetic code, then became very ubiquitous, and they formed a company around it, that was called

SpringSource. That technology, that company grew to the point where in a recession economy, even in 2009, they were bought by VMware for $420 million, something like that.

Then, VMware managed the projects and the teams that work on it. By the way, during that time, SpringSource had — It became a hub for all sorts of really great technology. The lead developers that work on a lot of the big Apache projects, the least of which of course is Apache Tomcat, were employed by a company called Covalent, which SpringSource bought. We, to this day, have done just and still do just a large majority of the commits to things like Apache Tomcat, and for a while there. Even HTTPD, Apache HTTPD. That was the technology that we worked on.

We eventually acquired RabbitMQ. We acquired GemStone, which is the company that makes the GemFire data grid, which is now called Apache Geode. All these sort of great open-source bits under one roof. Then, in 2013, EMC, which owns VMware and VMware which manages the Spring projects, along with VMware, they all sort of saw fit to say , “Okay. There’s this new direction. It’s a different direction. It’s worth walking that path,” and so they created a company called Pivotal.

Pivotal is sort of the integration of building applications destined for the cloud. We have a platform called Cloud Foundry, which is also open-source. It’s managed by the Linux Foundation. These technologies are sort of all sort at the intersection of building applications destined for this new environment, this new world. Pivotal sponsors a lot of the developers that work on Spring, but by no means the only developers. It is a vibrant open-source on GitHub, so we have just a crazy amount of activity on the projects. I think that’s about it. I think that what you’re asking, right?

[SPONSOR MESSAGE]

[0:29:57.2] JM: Don’t let your database be a black box. Drill down into the metrics of your database with one second granularity. VividCortex provides database monitoring for MySQL, Postgres, Redis, MongoDB, and Amazon Aurora. Database uptime, efficiency, and performance can all be measured using VividCortex.

VividCortex uses patented algorithms to analyze and surface relevant insights so users can be proactive and fix performance problems before customers are impacted.

If you have a database that you would like to monitor more closely, check out vividcortex.com/sedaily. GitHub, DigitalOcean, and Yelp all use VividCortex to understand database performance. Learn more at vividcortex.com/sedaily and request a demo.

Thank you to VividCortex for being a repeat sponsor of Software Engineering Daily. It means so much to us.

[INTERVIEW CONTINUED]

[0:31:16.7] JM: When you talk about these different open-source projects, for example, the GemFire database, or GemFire system — I did a show about Geodge, Apache Geode, very interesting in-memory grid system for managing distributed systems or distributed objects. What's the general pattern for what projects Pivotal wants to take on?

[0:31:44.2] JL: That's interesting. When you say projects that we want to take on, do you mean —

[0:31:48.5] JM: Projects that Pivotal wants to contribute to, or support, or build service contractors on top of, or consultants, I guess. What are the battles that you pick?

[0:32:02.2] JL: Very good. Again, what we're doing is this idea of agility, little A-agility, and we see that as a continuum. We see that if you're just doing agile software development but the rest of the organization isn't doing that, you're not going to get anywhere. You're going to be bottlenecked downstream at some other point in the workflow. You really need to think about the sort of holistic picture.

We have a consultancy called Pivotal Labs that they were bought by EMC and that became part of the Pivotal adventure. They are a company that's been around better part of two decades before Pivotal itself. They're now part of the family. We have the SpringSource Technologies, we have Cloud Foundry which came from VMware. We have all these technologies, and the unifying thing that we care about is helping organizations deliver software faster. If you think about what that implies, it means you've got to be able to do — Think about how to build software efficiently. We help people learn how to build better software. They come into our offices. We do pair programming. We do continuous delivery. We teach them how to do continuous delivery.

We care about sort of the knock-on effects of moving to that approach. When you move to continuous delivery and you have a lot of different teams that move into continuous delivery, what you want to do is make it easy for these teams to deliver independently of each other, at their own cadence. Of course, I think, at least I think, and I think a lot of people would agree, the logical sort of conclusion of that is microservices. You decouple, otherwise, sort of independent things into separate processes.

Of course, when you do that, now you have things that you need to manage, and run, and scale, and operation-wise. If the cost of operationalizing your services is prohibitive, then you won't do it, and so you'll just be stuck in this monolith which has even more gravity. We need a platform. We need something to operationalize at and to manage it and so on, that's what Cloud Foundry does.

When you move to microservices, some of your data workloads become different, your use cases become different. There's communication across processes that has to happen, so you have microservices themselves. They have reliability patterns that they need to care for, so we have Spring Cloud and Spring Boot for that. When you have services that need to talk to each other as well, there's messaging, so that's RabbitMQ.

Of course, data. Again, we talked about GemFire, which is now Apache Geode. The ability to process large amounts of data, especially the kind of data that's coming off from these different services and to analyze them in a single place and then turn that back into insight is important as well. There's a continuum, and we just see it moving — These things kind of — They sell themselves once you get to that step in the path. Once you understand that you need microservices, then the next logical thing is you need something to manage the complexity and plan by moving to a distributed system, which is what we think Spring and Spring Boot can be very useful. Of course, when you get to lots of different services, you need to operationalize them, you need Cloud Foundry, and the communication between these services and the integration of data. That's something you have to care about. Perhaps using something like Apache Geode and so on — RabbitMQ.

[0:34:57.6] JM: Make sense. Okay. The big picture is coming together. Is there a standard model that you're seeing around how people are building those distributed systems with different Spring microservices? Can you talk more about the general picture for a microservices architecture that uses Spring as the main template for services?

[0:35:20.5] JL: Sure. This isn't our picture. Well, slowly, it's starting to become our picture, but it's certainly — We're not the first to originate these ideas. Let's be very clear. We're very lucky right now, in 2017, to be able to look back over the last 10 years and we can see that countless organizations have attempted this migration, this journey. They've attempted this journey. Some, more successfully than others. Two; this idea of cloud native, where you talk about that all the time, cloud native. My book is called Cloud Native Java.

I think the most famous of these has to be companies like Amazon, and Netflix. These patterns, these best practices, these guide posts on the way, they've been out there. A lot of people have seen them and a lot of the good ideas that people have discovered or sort of realized would help them on the way, or already well-documented.

We sought to codify a lot of these patterns to integrate best of breed technologies from an ecosystem wherever possible, and to make it easy to build applications that take advantage all these things. For example, a common use case in building microservices is service registration and discovery. We have Spring Cloud. Sprint Cloud has support for service registration and discovery. You can talk to Apache ZooKeeper, or Apache Corpus Console, or you can talk to Netflix Eureka. There's that for example.

Another pattern that becomes very useful when you move to this architecture is centralized configuration, reloadable, encryptable, secure, centralized configuration, key and values, things like password locators, that kind of stuff. Auditable. We have a config server, but we also support drawing configuration from things like ZooKeeper and Console.

Another thing it becomes useful is security. How do secure a bunch of microservices and support things like single sign on? We have Spring Cloud Security, which brings in support for OAuth and acting as an OAuth client, and we have an OAuth authorization server that you can use as well, all of these is open-source of course.

Where there are use cases for which we couldn't find a clear and obvious standout solution from the ecosystem, then, and only then, have we plug the gap with our own club. What we're trying to do is to take these brilliant technologies out there and make them as accessible as possible and where appropriate provide an abstraction, a common interface. The service registration and discovery use case is a good one.

We have an interface called the Discovery Client. If you can adapt to your service registration and discovery vehicle, your mechanism, into that interface and you can use it in Spring Cloud consistently to do client-side load balancing, to do declarative REST clients to do whatever just by swapping out the implementation. There are a lot of these sort of patterns, and we baked into Spring Cloud, and that's all building on top of Spring Boot.

[0:37:59.8] JM: What's being worked on today in the Spring Boot project?

[0:38:03.2] JL: Spring Boot right now as we speak, our engineers are hard at work on Spring Boot 2.0, and Spring Boot 2.0, we expect full drop at the end of the year. Hopefully in time for big Spring One Platform Conference here in San Francisco at Moscone. What is sort of interesting about that release is that it will be the first to build upon Spring 5, which in turn features a reactive runtime based on our Reactor Project.

The sort of work that a lot of the industry players have been doing around reactive programming in the last five years, six years, has become — It's kind of reached a critical mass and that work a lot of the really — They key idea behind that is extracted out into — What is it? Four interfaces that are now part of the reactive stream's initiative to which Pivotal, RedHat, Lightbend now TypeSafe, and — What was the other one? Netflix, are all party. That reactive stream's initiative is something that underpins our particular Reactive framework, called a Reactor, and then that in turn is what we're using to build a fully reactive web runtime that isn't necessarily based on servlet engine, servlet API. It's built on Netty, it's a reactive runtime end-to-end. You can build REST services, build web applications, that kind of stuff.

For Reactive programming to be useful, it really has to be end-to-end. What good is having a reactive REST API if you're still bottlenecked at some sort of code in Spring security that's using a thread local? You're defeating the games but moving to that architectural if at some point you're doing blocking data access or doing blocking security and so on.

A lot of what we're doing on the Spring team this year is to make the rest of the Spring projects including Spring Boot play well with whole reactive initiative. Spring Boot 2.0 will be the sort of integration of all that. It will have all these things that it used to, but now there's the option to use the fully reactive sort of experience to build upon that.

[0:40:07.9] JM: Cool. All right Josh, I want to thank you for coming on Software Engineering Daily. It's been great talking to you about Spring and Spring Boot.

[0:40:13.8] JL: Hey, it was my pleasure. Thanks very much for having me. I appreciate it. If you have — If people out there are who want to get started start.spring.io, I encourage you to give it a go.

[0:40:22.1] JM: Great. All right, thank you.

[END OF INTERVIEW]

[0:40:26.2] JM: Deep learning is at the forefront of evolving computing and promises to dramatically improve how our world works. In order to get us to that bright future, we need new kinds of hardware and new interfaces between this AI hardware and the higher level software. That's why Intel acquired Nervana Systems, a platform for deep learning.

Intel Nervana is hiring engineers to help develop this full stack for AI, from chip design to software frameworks. Go to softwareengineeringdaily.com/intel to apply for a job at Intel Nervana. If you don't know much about Intel Nervana, you can check out the interviews that I've conducted with engineers from Intel Nervana, and those are available at softwareengineering.com/intel as well.

Come build the future of AI and deep learning at Intel Nervana. Go to softwareengineering.com/intel and apply to work at Intel Nervana. Thanks to Intel Nervana for being a sponsor of Software Engineering Daily, and I really enjoyed the interviews I have done with the Intel Nervana stuff. I think you'll enjoy them too.

[END]