EPISODE 628

[INTRODUCTION]

[0:00:00.3] JM: Modern programming requires lots of integration between APIs. Some of these integrations are trivial, such as using Twilio or Stripe, but there are many more complex integrations. For example, when a large company acquires a smaller company, the acquiring company might want to integrate with that smaller company to leverage the synergies between the two companies. How do you build clean communication patterns between the services of one company and another?

Two teams within a single enterprise can also have integration issues. One team might have a different data model than the other team. One team might be using JSON and the other using XML. In these cases, integrations between APIs can take a considerable time. Ballerina is a programming language that is designed for writing integrations. Ballerina is made for building services that allow two APIs to communicate easily, in contrast to other patterns of API integration such as those involving an enterprise service bus.

Tyler Jewell is the CEO of WSO2, which is a company that specializes in integrations. WSO2 created the Ballerina language and is investing heavily into it; with around 80 people working on Ballerina as of the publication of this episode. In this show, we explored integrations and why this problem required creating a new programming language. Tyler also let me know that the conference for Ballerina, Ballerinacon is on July 18th and it's in San Francisco. Our listeners can attend for free if they use the code BALCON-SEDaily. You can also attend online.

Ballerina is not a sponsor of the show, but I just wanted to mention that because that's pretty sweet that you can go to the conference for free if you're interested in Ballerina. I hope you enjoy this episode.

[SPONSOR MESSAGE]

[0:02:04.5] JM: You listen to this podcast to raise your skills. You're getting exposure to new technologies and becoming a better engineer because of it. Your job should reward you for being a constant learner and Hired helps you find your dream job. Hired makes finding a new job easy. On Hired, companies request interviews from software engineers with upfront offers of salary and equity, so that you don't waste your time with a company that is not going to value your time.

Hired makes finding a job efficient and they work with more than 6,000 companies from startups to large public companies. Go to hired.com/sedaily and get $600 free if you find a job through Hired. Normally, you get $300 for finding a job through Hired, but if you use our link hired.com/sedaily, you get $600, plus you're supporting SE Daily. To get that $600 signing bonus upon finding a job, go to hired.com/sedaily.

Hired saves you time and it helps you find the job of your dreams. It's completely free. Also, if you're not looking for a job but you know someone who is, you can refer them to Hired and get a $1,337 bonus. You can go to hired.com/sedaily and click refer a friend.

Thanks to Hired for sponsoring Software Engineering Daily.

[INTERVIEW]

[0:03:47.0] JM: Tyler Jewell, you are the CEO of WSO2. Welcome to Software Engineering Daily.

[0:03:51.5] TJ: Hey, thank you for having me.

[0:03:53.0] JM: We're going to get into talking about Ballerina, which is a new programming language built at your company WS02, but I think it makes sense to start with actually what WS02 does, because I think that strongly informs the decisions around starting Ballerina and the design decisions within it. WSO2 is an integration vendor. Explain what an integration vendor is.

[0:04:16.4] TJ: An integration vendor provides software to help data move from one place to the next here fundamentally. We do it for a variety of different protocols, different types of applications that need to be connected at different levels of scale and different systems of record like databases and various data stores and other third-party systems that's there. Integration is a pretty big problem. It's to the point where Gartner now says that 50% of all time and cost that goes into digital transformation projects is integration at this point in time. It's pretty much everywhere.

[0:04:54.1] JM: I want to give more color for what kind of integration we're talking about here. I'm just a random rogue developer, maybe I've built stuff within my school, or I've integrated with the stripe API, or I've done some stuff with AWS and those are integrations, but I think we're talking about here a different type of integration. Can you give a little bit more color on what kinds of integrations we're talking about, maybe some prototypical examples of companies that you work with?

[0:05:22.3] TJ: Yeah. In the integration market, it's really broken down into three categories. There's knowledge worker, which is just each of us and we need to integrate Gmail with our home applications, or whatever that is. You tend to do that on a self-service basis. That's not us. The second category is really departmental integration, which is I am a admin at a company and I'm managing our SAP implementation, and someone told me that we need to do a workflow integration with NetSuite. I need a SaaS to SaaS, or an app to app integration to achieve that. You do that integration with an online cloud vendor that's called an iPaaS and they provide templates to facilitate those two points talking to one another over a common workflow.

Or you can buy some off-the-shelf technology like an enterprise service bus, which we do provide that can act as that backbone for connecting those two systems together. Then the third category of integration, which is really the biggest market by far is integration that's done by organizations that considered themselves digitally driven, or they are themselves a software organization.

Most companies now see software as a core competency, even if you're in manufacturing, or telecommunications, or whatever vertical you may be in. As part of that software core competency, they're creating software that helps them compete, or helps them be more operationally efficient. In order to do that, they need to have an integration backbone that all their other software integrates with and that integration backbone connects their data and their applications and their legacy systems together to be able to interoperate and act as one.

[0:07:01.0] JM: I just want to go a little bit deeper on this. That example you gave of an SAP integration, so I think probably people are listening and they're like, “Wait, why do I need an integration vendor to help me with that? Doesn't SAP just have some APIs that I can integrate with? Why isn't this a self-service thing? Why do I need all these process around it?”

[0:07:25.1] TJ: Well, there's a bunch of things. When you are going to create an integration between two systems, oftentimes the data that you're pulling out of one system is not the same format as the data that's going into another system. Structuring the data transformation, how you go about doing that is a discipline. If you're going to do a lot of integrations, you need a repeatable way to do that, that's one thing.

The second thing is that when you're moving data from one system to another, you're probably doing that as part of a transaction that either needs to complete or have rollback. There's distributed transaction management that has to be taken into account, and you have to address a variety of scenarios of error conditions on either one of the different applications, or if there's an error while the data's in motion. That's transaction management and compensation that you have to deal with. Then depending upon the volume and load if you're going to be moving a massive amount of data over a wide distance, you just can't do that with a single computer. You have to have a strategy for how you're going to scalably extract all that data, send it to the right location, sequence it in the right way and do this against a variety of SLAs that you've defined as well.

When you start looking at it at that point of view, it becomes an architectural problem and not just a developer problem, and the architectural problem is about setting up the infrastructure in such a way where you can handle all these policies that you've got. That's where a vendor can really come in and help. In addition to providing core infrastructure software that makes this go a little bit smoother, we help you plan these implementations, we help you actually do the implementation and then figure out how you're going to operate them on going on an evolutionary basis.

[0:09:03.4] JM: To give one other perspective on the idea of integrations and one more example of an integration, if I'm a large software conglomerate, I am going to acquire smaller companies and I'm going to bring those companies into my conglomerate and I want to leverage synergies between those different companies that I acquire in my conglomerate. In order to get those synergies working well, I need to be able to integrate those companies together. Is that another integration we can lump in as we get into talking about these integrations and how to build better tools around them?

[0:09:39.1] TJ: Yeah. I think that people think of M&A as an integration driver, right? It's certainly an event that happens, and so you're really given a choice between consolidating your systems, or getting them to interoperate with one another, which is a form of integration. Other drivers that causing people to have to look at integrations more holistically are artificial intelligence and machine learning, right? If you have the richness of your AI initiatives is based upon how rich your data source is. If you have multiple disparate data sources all over your environment, you need a way to bring these together so that your machine learning algorithms can get the maximum benefit on that. That's a pretty big driver.

Another big driver is just scaling your other applications. If you look at organizations like Netflix or Amazon and the amount of customer demand that they have, they've chosen to be able to scale their underlying systems by disaggregating their architectures into a lot of individually deployable components. When you deploy these things as individual components, you have a very high degree of modularity, you get a very high degree of elasticity for that individual component. Now you have a lot of components, and so guess what? You need integration to make these components work well together.

We've got these major trends around massive data growth, M&A like you described and this disaggregation of architectures, which is just creating a much higher need for things that provide integration along the way.

[0:11:05.4] JM: There are tools for helping with these kinds of integrations. You mentioned the Enterprise Service Bus, or the ESB a little bit earlier. Talk a little bit about the historical tooling that has been built around integrations and the patterns that people have.

[0:11:20.1] TJ: Well, integrations have been around for 50 years, right? The first integrations where I have two mainframes and there's some stuff that we need to move back and forth, and they did it at the OS level network socket-based communications. Those were the earliest forms of integrations that were there. In the 80s and 90s, the predominant form of integration was called a message broker. Message brokers were predominantly just a very generic pipes for sending messages, either synchronously, or asynchronously from one producer to many receivers, or one receiver in some structured fashion on that. That's a good way for sending a large-scale number of events.

ESBs, Enterprise Service Busses came along about 15, 16 years ago through the evolution of an architecture called the service-oriented architecture, which was an evolution from a technology concept called web services. Web services were really the first attempt at allowing services to have an API before even APIs were called that, and then to advertise that API over a network endpoint with some standard protocol and some standard format.

15 years ago, this was the first time that a lot of companies even had a web presence. They were using these web-based protocols for how to connect these different systems together, and the enterprise service bus became an evolution of the message broker with these advanced web-based protocols on top of that to deal with the connection to these web endpoints, the data transformation of the content inside these packages and a lot of specialization around how to deal with HTTP and HTTPS, which is what the predominant protocol was.

That market really took off. The ESB market today is measured in probably more than 10 billion dollars. It's a substantial market. That market itself has been evolving over the past 10 years and it's really now been API-driven marketplace. APIs and the API economy have really come to bear where now all software systems have some API. These APIs are now a universal interface on how you can talk to those systems, and there's even an API economy where – ways that you go about monetizing and measuring the value of these APIs. Everybody's on this rush to API by everything, and so that creates an evolution of the Enterprise Service Bus model, where it's now integrating with these standard interfaces as opposed to just more of the HTTP.

That's been the predominant trend over the past 10 years, but now we're starting to see it change and move towards cloud native architectures. Cloud native architectures are basically on the premise of that all your software is instantly deployable all the time. In order to do that, you have to write your software in highly modularized microservices. Those microservices can be deployable in something like Cloud Foundry or Kubernetes, so that you can get instant scale around them.

Then as a result of that, the way these microservices integrate together is not so much through a dedicated ESB, but through an integration tiers that are melded into your microservices or the Kubernetes substrate, so that the integration capability is just part of the infrastructure that you're making use of as opposed to dedicated middleware.

[0:14:41.0] JM: That is a excellence synopsis of the history of integrations, as well as bring us forward to how modern deployment work flows around endpoints works, including Kubernetes. You and I met at KubeCon, where Ballerina had its large launch. Ballerina is this programming language that was built at your company WSO2. Ballerina was built with these trends in mind, as well as this place that we've landed at where we have this cloud native infrastructure that you've just been talking about.

I think it's probably worth pointing out that we've done a lot of shows about Kubernetes, including one where Brendan Burns talked – who is the creator of Kubernetes, talked a bit about how his vision for where this goes next is the language level primitives that assume that you are on a distributed system and assume that you have these distributed systems problems and potential distributed systems solutions. With that, why did you start a new programming language?

[0:15:47.8] TJ: Let's start with our history. We've been in business for 14 years, and we've done 2,000 integration projects for our customers on that. Our customers use our current technology stack. They do almost 6 trillion transactions a year through it. It's a big volume. We've been able to lay witness to a lot of things.

What we've seen with our customers is we measure with all of our customers what's their release cycle. In over the past 14 years, we've seen their release cycles get better, but not agile. They're more agile, but not agile itself. No matter how advanced they get with the deployment of these integration technologies, fundamentally the integration technology that they use today is a piece of middleware; it turns into a center of excellence. When there's a center of excellence, the development team who needs to write the logic that works through these integration buses has a gate that they have to go through to get to deployment.

This predominant architecture that we've been advocating over the past 20 years provides a mid-scale, immense flexibility, but limits agility. This is a proof point around this in the latest state of agile report that just came out a couple months ago, 59% of organizations around the world report applying and adopting some agile practice, but only 4% of those organizations report getting any benefit and adaptability from agility.

We looked at this and we said, we actually think that middleware is the problem. Middleware is the problem is because it keeps you from releasing it, the cycle that the dev team needs to release at. At Ballerina, so the founders went to the drawing board and they started looking at how do we overcome this problem? They started thinking about rewriting the technology, making it lighter, making it more efficient, but inherently no matter how light or efficient that it got, it was still this fundamental bottleneck.

They realized that the only way to get rid of that bottleneck is that the integration technology that an application needed, needed to be blended into the application itself. The way to get it blended into the application was we need to have a programming language where we have a compiler, and the developer will write syntax that is aware of its environment. Then because the syntax is aware of its environment, the compiler can generate a binary that has exactly what it needs to scale to secure it, to communicate with other services without any additional deployment mechanism for the middleware itself.

That's how the basic idea of Ballerina came up. They go, “Okay, if we're going to do this, most languages that are popular today were built 15 years ago where the network was an afterthought. Now we know there's a lot of best practices about how software systems are built, how they're deployed. Why not design a language where it has, the syntax has distributed systems abstractions built into it, that a lot of other languages force upon the developer that you no longer need, because you can make these assumptions about this is the type of programming that you're going to do.”

They applied these design principles to the type system, to the concurrency and threading model, to the internal memory management system of it all. There are other places that I'm forgetting off the top of my head as well, but by taking that orientation and saying, “Yes, it's a general-purpose programming language, but we intend it to be used to write microservices that talk over the network to others on a safe way that has influenced every design principle they've made with the language and its syntax itself.”

[SPONSOR MESSAGE]
[0:19:35.1] JM: Raygun provides full stack, error, crash and performance monitoring for tech teams. Whether you're a software engineer looking to diagnose and resolve issues with greater speed and accuracy, or you're a product manager drowning in bug reports, or you're just concerned you're losing customers to poor quality online experiences, Raygun can provide you with the answers.

Get full stack, error and performance monitoring in one place. The next time you're struggling to replicate errors and performance issues in your codebase, think of Raygun. Head over to softwareengineering daily.com/raygun. Get up and running within minutes and dramatically improve the online experiences of your users.

Thank you to Raygun for being a sponsor of Software Engineering Daily. If you want to support the show while also checking out Raygun, go to softwareengineeringdaily.com/raygun.

[INTERVIEW CONTINUED]

[0:20:39.1] JM: Let's zoom in on that bottleneck that you pointed out. I have Parts A and B that I want to integrate together. If I want to integrate endpoint A with end point B, you are commenting on some specific bottlenecks that I'm going to create, because of the way that I'm going to do my integration. Help me understand what are those bottlenecks? What are those problems? Where are they manifesting? Am I setting up some service that is between A and B, and then that service is a problematic entity? Why is a language a solution to these bottlenecks?

[0:21:17.6] TJ: Let's take the classic enterprise integration scenario. Today, what an enterprise is going to do is to make two points talk to each other. They're going to first get an enterprise service bus as point C to deploy that. In order to make this integration work, they have to figure out the right service bus, they have to deploy it. Then now that the service bus is deployed and running, they have to make two separate connections, one to point A – the service bus has to make a connection to point A and the service bus has to make a connection to point B.

In order to do that, these service buses have connector approaches that you have to do. These connectors are things that you either need to write or you need to go find one off the shelf and they’re code, but they’re proprietary code. They're not writing normal code like a very sophisticated algorithm. You're writing some scaffolded, structured code against their specific format and there's no standards for these connectors. Every ESB has its own approach.

You've got to go and find those connectors and then you're going to have to maintain them, because the connectors are evolving, right? Then the systems that they're connecting to may be evolving as well. You've got all this change management you need to impose just to keep the connectors going. Then the developer needs to write the logic that goes into the ESB that tells the ESB how to connect to point A, how to connect to point B, and then what to do when the data is flowing in between. Do you want to change the data format? Do you want to call out to another system? There's all sorts of different logic that goes into the middle of that, and that logic is also a different set of code that you have to write and maintain.

A lot of the data transformation is not even code. It's usually done in XPath, or XML. I don't know many developers who really enjoy living their world inside of a query language on that. To get all these set up to run and maintain one integration, you're looking at three to four different types of logic that you got to build that doesn't run standalone. These are things that you got to deploy into the system. You got to maintain this system, you got to deal with all this extra XML and YAML transformation, and then you've got to monitor all this stuff because if something breaks, it can break at any number of different points; it can break at the network, it can break at point A or point B, it can break at the point where it's going into the enterprise service bus, or it can break inside the service bus. Monitoring this becomes a real problem as well.

I have never come across – while integration is everywhere and everybody has to do it, I've never come across a person who wakes up in the morning and says, “Boy, oh boy, oh boy. I am glad I am an integration specialist and that this is what I do all day long.”

[0:23:51.0] JM: It sounds like the root of the issue is the ESB here. If we're talking about Kafka, or TIBCO, or NATS, these are all these service buses, Kinesis or Google pub/sub, these places where you’re publishing messages and then you're having another service consume that message and the connection points between publishing it and that message being consumed, you can do the necessary transformations to get these things to talk to one another. Am I understanding things correctly?

[0:24:22.0] TJ: That's right. Yeah, yeah. When I talk about middleware is the problem and ESB as a form of middleware, because you have to – what middleware does is it tries to scale something for you, tries to make something that you have to do repeatedly easier, but in order to do it you have to first put the middleware down and then the middleware becomes a huge dependency, a massive dependency that takes more care than the application itself.

Your solution to that is basically instead of writing an interface to the pub/sub layer, let's just stand up on either side on both A and B a service that is written in Ballerina. Am I understanding that correctly?

[0:25:02.7] TJ: Yes. In the Ballerina world, if you wanted to integrate A and B, what you would do is you'd write a service C, a microservice that you're going to deploy inside of Kubernetes, and that microservice is written in Ballerina, and the Ballerina syntax understands that point A and point B as endpoints. Endpoint is actually a keyword in the language, and you just define these endpoints, like you define a variable in any other language. Then once they're defined, that takes care of the underlying connection semantics like you had an ESB.

Since you're running a microservice, this micro service itself has an API that can be invoked. When somebody says run this microservice, there's a function call inside. It looks like and feels like a function call, which is really going to be invoked when somebody hits you from the outside and that's where you put the logic. You write that logic and then the logic has access to those variables, those two endpoint variables so that you can talk freely to point A and point B. If you need to do a transaction to make it atomic, there's a transaction block which is similar to what a try-catch block is, right? Which makes the whole thing atomic. If you need to do a data transformation, everything is strongly typed.

What happens is that when you connect to endpoint A, if endpoint A sends you a complicated packet, it gets mapped into a strongly typed record, or an object that is also defined that you've defined in the language, or a string, or guess what? JSON and XML are also primitives in the language. When you get data out of the communication, it's in a variable, it feels natural to the developer and then you can just cast that variable to whatever other format you want to using developer techniques and then just call the other endpoint there.

To a developer, it feels like he's just calling, writing a function, invoking some variables, but what's really going on is that he's enacting these underlying ESB mechanisms without realizing it, there. then the compiler, because it's aware of all these stuff, when it generates that microservice binary inside that microservice is behaving at runtime largely the same way that an ESB would. It turns out that when we deploy it, we can get 30% to 50% more transactions per second on a single VM, because we've just gotten rid of a lot of the Kluft that a lot of middleware systems need, because they're expecting massive volume.

[0:27:28.8] JM: What makes sense to me about this idea is that if you think about the difference between an ESB and very small micro – so to put a cap on what you said, this is a microservice, it’s like a layer of glue that you're putting between A and B that lets A and B stick together and communicate with each other, and you have all the code in one place, which sounds great. We've gotten to a point where our compute abstractions can be shrunk down and we can have economies of scale with very small compute abstractions as opposed to the ESB world, where you have to have these bulky, these bulkier deployments.

Maybe the ESB made sense when if there was more bulky deployments and it wouldn't have made sense to have this huge number of endpoints, because if you've got thousands of integrations across your company the VM era, or the VM era wouldn't make – or I'm not sure, but it sounds like it would be much more costly to stand up.

[0:28:34.9] TJ: I think what you're touching upon here is it's not so much a cost factor, but it's in many ways, it's a manageability factor. When you use an ESB in a service-oriented architecture, it's almost always the case that the number of endpoints that you want to integrate is static and predictable. Organizations know exactly what data sources and applications they want to integrate. They want to do it at scale and they're willing to invest the time to plan that out. In a Kubernetes microservices cloud native architecture, the number of endpoints that you have becomes unpredictable.

[0:29:10.0] JM: Right. You want to make the developer – you want to give the developers more freedom to integrate with each other and you don't want to have to have this ESB in the middle as a single point of synchronization.

[0:29:20.1] TJ: It's a single point of – yeah, we call it a center of excellences or a release gate, right? Because if the middle – if the ESB is there, then the developer can't finish his work until he's coordinated with the ESB team to make sure that it's going to be able to do what he needs it to do on that. That just doesn't move at the developer speed. The whole point of cloud native architectures is that you can keep modularizing as far as you want to go. It’s the developers choice to choose when he wants to publish a new API. Not the center of excellence of the architect's team.

It becomes a feedback loop, because every time you deploy a new microservice with a new API, that's just going to make a bigger need for integration, because as your microservices grow then the number of integration points grows with it.

[0:30:04.7] JM: Okay. At this point, I can see the motivation for wanting to replace the ESB as the integration mechanism for A and B. I still don't quite understand why do you need a new language? Why not set up a JavaScript framework? Why do you need a completely new language to define these integrations?

[0:30:24.8] TJ: There's just a whole number of issues with going with existing languages. I don't even know where to start. First is is that the type systems and existing languages are not type systems that are not – are network aware. JSON and XML are fundamental parts of talking over a network and tables to databases, right? Tables, records, SQL, JSON, XML. In all other languages, these are libraries that you have to explore to be able to work within –

[0:30:55.2] JM: Wat about protobufs?

[0:30:56.3] TJ: Protobuf as well. Yeah. When you're dealing with integrations over the network, data transformation is a massive amount of the work, right? You have to be able to do data in one format, convert it to another format and move it out. At the heart of Ballerina, it's got a strongly type system, all these types are built in to language itself, and so there's a lot of obsession about how you transform from one type to the other. It's in a very natural syntax. It's amazing how natural it is to do that. That's one thing.

The second thing is is that you really do need a union type system in order to work over a network. A union type system is one where a type can be different types at the same – one of a different, set of different types. If you're talking over a network and you're going to get a response back, that response could either be in some positive response that is a record that you expect, or it could be an error, right? You really don't know. Oftentimes, there could be three, four, or five different potential responses that you get. Each one of those responses can be mapped to a different data structure type.

Why put that burden on the end-user to try to get a generic packet at and then you have to inspect the packet to figure out what data type you want to map into it? In Ballerina, we provided a union type system that says, "Look, when I call out to this endpoint, I expect either a string, a JSON, or an error to come back." It's going to be one of those three things. The underlying system will take care of dealing with the mapping of that, and then there is language semantics for the developer to terminate finding out which one of those three it was can match against the types and whatnot. There's all sorts of scaffolding that you historically need that goes away with the union type system.

Other things are the runtime model. The runtime model is in integration scenarios is a worker-based runtime model. The threading pattern that is ideal here is that you want one thread per worker. In a Java environment, for example they give you one thread per class. Nodejs, there's one thread for the entire thing. Go has a threading model that's similar to this. If you want to have everything be non-blocking i/o, you need to have your own specialty thread model, which means you need to implement your own scheduler, which means you need to implement your own VM. A lot of the existing languages just are not optimized for that scenario that's there.

A third thing too is that existing syntaxes do not let you interpret that syntax in a structured way. One of the things that Ballerina’s syntax is both a graphical syntax and a textual syntax, which means that as you're writing the code it’s self-documenting and we can generate interaction and sequence diagrams from any code that you write. If the code compiles, then we can generate a fully qualified sequence diagram from that.

In fact, we do that inside a VS Code, or IntelliJ, or the composer wherever your IDE is, we automatically generate that for you. That's because of the structured approach to that. This just goes on and on and on. There's probably a few dozen design choices that have come along that you just can't do in a language. If you look what's out there, like meta particle, meta particle is in some ways doing similar things to what we're talking about here, but they're adding it on as annotations or as language extensions. That just increases the learning curve for developers, because they have to be a predominant expert in that language and they have to understand this form of annotation or extension.

It's just a lot a lot of things that get layered on that basically increases the learning curve, increases the tool set that a developer needs and we're not sure that it was the right thing to do. That's why we went back to fundamentals and built it as a language.

There are some amazing benefits that come from writing your own compiler. When you have your own compiler, that compiler can make all sorts of intelligent decisions. We're actually able to generate things for Kubernetes and Cloud Foundry. We're able to offer compiler extensions. There's a lot of things that we can generate, because we can put hooks into the code that make it environmentally aware. Most languages don't have that concept built into them, because they didn't intend them to be environmentally aware.

When you bake these things into the compiler, what you're doing is you're removing a lot of unnecessary continuous integration stuff that organizations set up. There's a ton of infrastructure around Jenkins and other CI systems that are really there to generate assets that the developers should be able to generate. As a result, we can get the developer so much closer to deployment without having to go to CI by baking it into the compiler.

[0:35:41.4] JM: I haven't done any shows around the topic of “domain-specific languages.” There's another podcast called Software Engineering Radio, that I believe has done some shows on domain-specific languages. I think that term might apply here. Would you consider this a domain-specific language, where you basically have said integrations are such a big part of writing software, we need a domain-specific language. Or is it something bigger than that that is just being used for integrations today?

[0:36:13.1] TJ: I think that on the one hand, Ballerina is a general-purpose programming language. You can use it to create anything, right? It is definitely that. DSLs have two kinds of meanings to people. If one meaning to you is that it is a language that is attempting to solve a particular set of use cases and doing it really, really well, then yeah, Ballerina is intended to be that. It is a DSL. A lot of people treat DSLs and it's been morphed I think inappropriately to imply a derivative language on top of another, a derivative language of a general-purpose programming language, right? Like Kotlin is optimized for Android, for example. They'd call that a DSL in that sense. I think that's an inappropriate definition, but a large number of people think of it that way incorrectly.

[SPONSOR MESSAGE]

[0:37:13.9] JM: Stop wasting engineering time and cycles on fixing security holes way too late in the software development lifecycle. Start with a secure foundation before coding starts. ActiveState gives your engineers a way to bake security into your languages’ runtime. Ensure security and compliance at runtime. A snapshot of information about your application is sent to the ActiveState platform. Package names, versions and licenses and the snapshot is sent each time the application is run, or a new package is loaded, so that you identify security vulnerabilities and out-of-date packages and restrictive licenses such as the GPL, or the LPGL license, and you identify those things before it becomes a problem.

You can get more information at activestate.com/sedaily. You want to make sure that your application is secure and compliant and ActiveState goes a long way at helping prevent those kinds of troublesome issues from emerging too late in the software development process. Check it out at activestate.com/sedaily if you think you might be having issues with security, or compliance.

Thank you to ActiveState for being a new sponsor of Software Engineering Daily.

[INTERVIEW CONTINUED]

[0:38:44.3] JM: This is obviously the perfect company to create a language that is optimized for integrations, because you have a huge base of users that could potentially try it out, adopt it on mass. I do want to get into language decisions and features and stuff, but I'm really enjoying the conversation around just the thrust of this project. When you are talking to customers who have these integration problems and you come in to them and say, “Hey, we built this new language. It's called Ballerina. We would love it if you tried it out.” What has been their reaction?

[0:39:20.2] TJ: The reaction, very sensitive. I think that in organizations that have been doing a lot of ESB development right and they've been living with it for a long time, they see it as a huge breath of fresh air, because the pain is very acute to them on that. We've seen a huge number of organizations, very top-tier Fortune 500 companies who are actively using it and some of them have put it in production already. That's been really exciting.

The thing about integration though is that a lot of integration is also built by companies who are, I don't want to say laggards, but just very patient in the work that they do on their IT systems. With some of those companies, they have the opposite reaction is, “Well, I've made this 10-year investment into this ESB and you're telling me that I need to – can I support that investment, or do I need to abandon that investment? What are you telling me here?”

Those organizations are looking at hey, yeah, maybe for new greenfield work that we do in the future, this could be better for us, but we really want to just keep maximizing the investments we've already made. There's some category of organizations that are out there. Those how our existing customers react, and I'd say that on the balance most of them are pretty darn excited by what we've done with this.

What's interesting is that we're being very patient in how we bring it to our customers. Our job is to support them and a lot of our customers are with us for up to 12 years. We have these very long-lived relationships with them. We're very cautious in what we've taken to them so far. Most of our outbound work with Ballerina actually is to the cloud native community, that that's why you found us at KubeCon. We're going to continue advertising at those events and doing all of our evangelism around the container community.

The container community largely has no idea who we are. It's really a net new and people are like hadn't heard of you. Tell me more about your company. It's a whole learning experience for them.

[0:41:28.0] JM: Well, I mean, the booth was super popular at KubeCon. You had a lot of people stopping by. What I wonder is when you were talking to the Netflix's of the world, or the Ubers of the world, I think a lot of these companies have centralized around Kafka, or Kinesis, or Google cloud pub/sub and they're using these as the integration points for their highly distributed teams. What are the conversations with those kinds of companies?

[0:41:55.9] TJ: When you get into a company that has such massive scale where they've chosen a Kafka, right? A Kafka has a very relevant and important role in the future cloud native architectures, because there are just certain types of workloads where if you need to do stateful messages at a certain volume, there is no such thing as a better solution than Kafka at that.

Those applications and those workloads that need that SLA are going to depend upon that and it's a great use of that system. As you're building those services, those services guess what churned into microservices, those services have APIs themselves. At some point in time, you're still having a lot of microservices and not all those microservices need to get the high volume messaging of a Kafka and you start worrying about the glue. What's going to be the glue of all the other stuff that I've built here?

The reaction of those organizations are, “Hey, Ballerina may be a fit as the glue in certain areas.” We've also heard a lot from DevOps professionals and sysadmins that hey, there's a large number of scripts and batch processing systems that we write that now need to communicate over a network, Ballerina's a lot more a great way for me to quickly write these sorts of programs that integrate these different systems together.

Then with some of these organizations, they're always looking to squeeze a little bit more performance, so the really high-end, high-scale organizations they tend to take more of a research attitude towards Ballerina. They want to kick the tires, they want to put it on research projects, and they're looking at how they can really get a little bit more performance out of whatever systems they're in. That's been the reactions we get with those sorts of guys.

[0:43:39.6] JM: I know we're in the last quarter of our time here and we haven't even really talked about what the – how the language looks and some of the different language features. Hopefully, we've peaked people's interest enough for them to go check it out. It's obviously difficult to discuss what a programming language does at a deep level over raw audio, so people should definitely go check it out.

Could you describe a little bit about how the language looks, if people were to look at it with the code appear as? What's the development process for writing one of these integrations, or writing one of these microservices, or just describe the ideation and the end the up, getting started process for writing something in Ballerina.

[0:44:21.6] TJ: Well, there's no way we're going to be able to do enough justice on how code looks by doing a talk, but we are – so before I get into my description, we are having a Ballerinacon. We have roughly 500 people coming to it. It's in July 18th in San Francisco, but there's also a virtual test of it, and you can get free tickets to that. If people want to hang out and put it on in the background and listen in and learn more about Ballerina and the language, I think that's going to be a great time for them and we can help your listeners get into that.

The language itself has a syntax that is very C or Java-like at its core. Fundamentally, when you get down to the logic level, you're going to see loops and branching conditions. You've got variables assignment. Data's can be in a record or an object structure. There's also some functional programming elements to it. There's no dogmatism, all right? It's not that the designers have come at it and said, "It's got to be functional programming, or it's got to be object-oriented programming." They've chosen language features that are going to be largely familiar to anybody who's worked on another programming language, but applied to this integration domain.

Where the syntax varies, and you can do main methods, right? If you just want to do a ‘Hello world,’ you can do a main method and that looks largely the same. Where the syntax varies is that most programming languages only have one entry point. It's the main method. You compile it, you get the binary and then you have the main method as the entry point. Ballerina has two kinds of entry points, either a main method, or what it calls a service. A service is something that looks like a method call, it's got a particular signature and a syntax on it, and that service basically defines this microservice is going to run as a server, it defines the protocol that it's going to listen on, it identifies the binding element, and then once you've defined that service, this service can be invoked over different resources and these resources are just method names.

These method names are things that get called when one of those requests that match that come in, and then you just put your logic inside that method name of what you want to do when that request is called. That's the basic format. It has a modular and package management system, so you can import packages that you write, or standard libraries at the top. You can declare global variables, like things that are endpoints, right? You can declare these endpoints as variables, or inner variables that then communicate over things, but then they're just variables.
When you're using these variables, you do dot notation when you want to communicate locally and there's an arrow notation when you need to communicate over the network. It feels very natural. You're using dots, you're using arrows and you're basically making function calls and variable assignments as you go on this.

When you compile it, it's everything goes into a Ballerina file, a .bal file. You just say Ballerina build and it generates a binary, and then you say Ballerina run and if it's a main method, it executes it, and if it's a service entry point, it launches it as a server right then and there, and then you can invoke that server with curl or whatnot on that. That's the entire workflow. If you need to then extend that service, you can go back into it and there's an annotation framework built into it, so you can annotate your service to say, “Hey, I don't want it to just be a service. I want it to be a Kubernetes, it's going to be deployed in a Kubernetes pod.”

There's a Kubernetes annotation and the Docker annotation. There's a bunch of annotations and you can add your own, but then the compiler when it compiles it, it won't just generate the binary but it'll generate the binary plus all the other stuff you need, so you just push it into Kubernetes and it's ready to go.

[0:48:03.0] JM: For the case of service A integrating with service B, what is being written? Let's say there's some data transformation that needs to occur in service A talking to service B. How does service a call the Ballerina middleware and how does the transformation look? What does it look like to specify such a data transformation, and then how does the Ballerina middleware talk to service B?

[0:48:25.7] TJ: Yeah. Let's say the Ballerina microservices has been built and running, right? Let's say you've binded it to HTTP, so it accepts HTTP requests, right? If your endpoint A generates server-side events over HTTP, you just point those events at the microservice, and when that HTTP packet arrives, that will invoke the right resource inside of the Ballerina service there.

That resource method that gets invoked inside, it has two input parameters; one input parameter is an endpoint variable that represents endpoint A, so the call, we call it the caller. The caller is passed in in case Ballerina wants to talk back to the person who called it, and then the second one is the request object, which has the information that's come in to that. When you parse the request object, you can get the payload, and when you get the payload you just cast it to the data type that you need. It's string whatever it is, it's very straightforward in that regard. When it comes time to then talk to in point B, you declare a variable that is to that endpoint.

When you declare that variable, it has all the connection initialization parameters so it knows how to go talk to endpoint B, you can bake in security parameters, a circuit breaker, all sorts of things you can bake in about how you're going to talk to endpoint B, but at the end of the day you get a variable. It's just a variable that is the representation of endpoint B. Then you can call actions against that, just the way you do with any variable, like a method calls. Those method calls have variables that they accept and they return and they're strongly typed.

The data transformation is really just a question of taking a variable that has data of one type and casting it to a variable of another type and then calling a method on that and you're done. When that method finishes, it returns control back to the caller and the service is still running, waiting to accept another invocation from some outside party.

[0:50:28.2] JM: Okay. I know we're basically at a time. I just wanted to ask and this is a big question to end on, but describe the process of creating a programming language that requires so much upfront work. This is a project that required a lot of upfront work. You had to spec out a programming language, you had to figure out how to make a compilation path, all these different things. Could you just talk a little bit about managing a programming language as a project?

[0:50:55.0] TJ: Oh, well. The person who's done all this work is the founder of the company, Sanjiva Weerawaranano. It was his brainchild about three years ago. The evolution of this is it started off with just about six to eight guys who were all existing employees, and basically he put them through compiler training. You start by just saying, here's some rough prototypes of what a language might do and they spent six to nine months just learning how to write parsers and lexers and compilers and just going back to basics on all sorts of stuff there.

They had a North Star direction on what they were trying to achieve, but they went through five, six, maybe even seven iterations before – and they started to settle in on syntax that they could relate to. About a year and a half ago, I'm trying to think, yeah, about a year and a half ago, they realized, "Okay, we're starting to circle around the drain here. Now we really need to see if we can stretch it broader. We're going to need a connector framework. We're going to need to understand how endpoints work." It's got its own micro-transaction capability, you're going to need to have scheduler. You start thinking about scaling it.

Just like any project, they started breaking it down. We broke it down to about 10 different sub teams, spanning language, runtime, build tools, language servers, IDE extensions, you name it. At that point time, the team got up to about 30 people. As an investor, we really like the potential of that. Once we decided that we actually wanted to ship it and make it production grade, we got it up to about 80 people working on the project.

[0:52:28.3] JM: Oh, my God.

[0:52:29.2] TJ: There was a big push on that. Even today, I think one point we had about a hundred 120 people, because the language designers decided to do a rethink on some of the – some pretty fundamental portions of the syntax in January of this year, because they didn't want to put it out until they felt it was just right. Even now we're at like 80 or 90 people full-time working on. I mean, we have as many people working on this language as Pivotal has on Spring, there.

[0:52:55.0] JM: Well, that is a massive investment and I'm excited. I'm excited by the ambition of it. It is an ambitious idea, and I can't even really think of any other, I don't know what the historical analog is to this project. I'm going to be following it closely.

[0:53:09.6] TJ: Oh, thank you so much. That's great to hear.

[0:53:11.4] JM: Okay. Well, Tyler thanks for coming on Software Engineering Daily. It's been great talking to you.

[0:53:14.8] TJ: It's been wonderful. Thank you for having me.

[END OF INTERVIEW]

[0:53:19.8] JM: GoCD is a continuous delivery tool created by ThoughtWorks. It’s open source and free to use and GoCD has all the features you need for continuous delivery. Model your deployment pipelines without installing any plugins. Use the value stream map to visualize your end-to-end workflow. If you use Kubernetes, GoCD is a natural fit to add continuous delivery to your project.

With GoCD running on Kubernetes, you define your build workflow and let GoCD provision and scale your infrastructure on the fly. GoCD agents use Kubernetes to scale as needed. Check out gocd.org/sedaily and learn about how you can get started. GoCD was built with the learnings of the ThoughtWorks engineering team, who have talked about building the product in previous episodes of Software Engineering Daily, and it’s great to see the continued progress on GoCD with the new Kubernetes integrations.

You can check it out for yourself at gocd.org/sedaily. Thank you so much to ThoughtWorks for being a long-time sponsor of Software Engineering Daily. We're proud to have ThoughtWorks and GoCD as sponsors of the show.

[END]