EPISODE 953

[INTRODUCTION]

[00:00:00] JM: Java programs run in a very different environment than they did 10 years ago. Modern infrastructure often runs on containers sitting in a Kubernetes cluster. The optimal configuration for a Java program in that context is different than it was for an environment dominated by virtual machines and bare metal. When you are co-scheduling your services with each other, those services could be fighting for resources. You may want to optimize them with more ahead of time compilation.

Quarkus is a system for accelerating Java performance through the use of GraalVM. In a previous show, we explored the basics of GraalVM. In today's show, Guillaume Smet and Emmanuel Bernard join the show to describe an application of GraalVM, which is the acceleration of Java. Guillaume and Emmanuel are engineers at Red Hat and they're working on changes to the Java ecosystem that are informed by the cloud and the rise of Kubernetes.

GraalVM and Quarkus are fairly complex topics, but they seem very futuristic and they seem relevant. So I hope you get something out of this episode even if it's a bit hard to understand on a technical level. If you are deeply familiar with Java, I think you will get a lot out of it.

If you're building a software project, post it on Find Collabs. Find Collabs is the company I'm working on. It's a place to find collaborators for your software projects. We integrate with GitHub and make it easy for you to collaborate with others on your open source projects and find people to work with who have shared interests so that you can actually build software with other people rather than building your software by yourself.

Find Collabs is not only for open source software. It's also a great place to collaborate with other people on low code or no code projects, or find a side project if you're a product manager or somebody who doesn't like to write code. Check it out at findcollabs.com.

[SPONSOR MESSAGE]

[00:02:17] JM: This podcast is brought to you by PagerDuty. You've probably heard of PagerDuty. Teams trust PagerDuty to help them deliver high-quality digital experiences to their customers. With PagerDuty, teams spend less time reacting to incidents and more time building software. Over 12,000 businesses rely on PagerDuty to identify issues and opportunities in real-time and bring together the right people to fix problems faster and prevent those problems from happening again.

PagerDuty helps your company's digital operations are run more smoothly. PagerDuty helps you intelligently pinpoint issues like outages as well as capitalize on opportunities empowering teams to take the right real-time action. To see how companies like GE, Vodafone, Box and American Eagle rely on PagerDuty to continuously improve their digital operations, visit pagerduty.com.

I'm really happy to have Pager Duty as a sponsor. I first heard about them on a podcast probably more than five years ago. So it's quite satisfying to have them on Software Engineering Daily as a sponsor. I've been hearing about their product for many years, and I hope you check it out pagerduty.com.

[INTERVIEW]

[00:03:44] JM: Guillaume and Emmanuel, welcome to Software Engineering Daily.

[00:03:47] EB: Hello.

[00:03:49] GS: Hello. [inaudible 00:03:49].

[00:03:49] JM: We have done a show in the past about GraalVM and I want to spend most of the time assuming that people have some sense of what GraalVM is, because if they don't, they can listen back to that last episode. But in case they don't want to do that, give people a brief overview. What is GraalVM?

[00:04:10] GS: Basically, GraalVM, simplified a bit, it allows you to create [inaudible 00:04:16] executable from your Java application and also support [inaudible 00:04:25] such as Ruby or

JavaScript. But if we focus on Java, the idea is to take your Java application transform it into an executable. The good thing with GraalVM is that compiler is written in Java, so it's quite easy to debug, maybe not the exact term but at least it's easier than with JDK.

Yeah, so you end up with very minimal executable and it also has the ability to start faster than the traditional JDK. Yeah, it's quite impressive when it's used Quarkus, because you can start your Java application in a few minutes and that's a game changer.

[00:05:20] EB: Yeah. If you take a step back, if you look at how Go applications are actually compiled and how they work, Go is a garbage collected language and they compile the application into a native executable that you run on the target platform, and that's essentially what GraalVM provides to the Java universe, with caveats, but that's what it provides to the Java universe. I think – I forgot the actual details of the interview you had with Thomas, but I'm pretty sure he probably mentioned that as some sort of a target.

[00:05:54] JM: Most of that interview was talking about GraalVM as a polyglot language platform. Meaning that many different languages could execute on top of this single virtual machine. Today's show is going to be mostly about how Java executes on top of that virtual machine. Is Java some kind of special case in terms of how it runs on GraalVM?

[00:06:22] EB: It is a good question and I'm not 100% sure. I know in their mission, they definitely want Java to be kind of yet another language in the sense that the way they would support Java 8 and Java 9 and the futures of Java would just be just like as if it was somewhat of a different language. That's what I understood from them. Yeah, that's a good question whether Java is in some sort of a really first-class citizen at the implementation details.

[00:06:51] JM: Who is using GraalVM today and how are they using it?

[00:06:57] EB: Another question I'd have to say I'm not sure. But as you said, for why GraalVM really push this notion of polyglot and people looking after pretty high-performance for JavaScript language is another key language that they have. They were looking at that, because they were benefiting from a lot of the Java future machine technology to really run their language in a much more efficient way.

When we saw GraalVM, we were really interested in the pure Java aspect. To some extent, the polyglot aspect is not really core to the fact that it's really useful for us. What we saw about it was the capacity – Because you precompile the application, as Guillaume was saying, the capacity to really start the application like in a few milliseconds, but also to minimize the memory usage. If people remember Java, it's birth is from the 90s. At the time, the idea was to really optimize take one big machine, one big process that would actually run as many request per second as possible. Okay?

Then it's some tradeoff. They really got very good throughput, but they made some tradeoff as far as consuming a bit more CPU and consuming more memory for that throughput. It wasn't a problem at the time. Memory was cheap. Why it's so relative? It's funny how you really change the optimization of a time. We can come back to that later.

But when we look at it today and in the universe of microservices, the way you scale is not so much by saying I'm going to get a beefier process or a bigger machine. It's more I'm going to deploy my application a second time or a third time or a tenth time. Your unit of deployments has to be very compact to consume as less resource as possible. Some sort of orchestrator, of course nowadays, a lot of people are replatforming on Kubernetes. In this case, a container would be your unit of work. Your application would be deployed as several instance into different containers.

Because of that, the fact that Java was a bit heavy on the memory usage, led to Java being a bit as, well, quite at a disadvantage in this replatforming. What GraalVM provides by changing the way – In a normal virtual machine, a Java virtual machine, the Java is interpreted and then we see the hard path and that hard path is actually compiled dynamically.

To do that, the JVM has to keep a lot of information around classmate data. Who is the super class of who? What method has been overloaded? Which part of the method is used a lot versus not a lot? All of that information is necessary for the just-in-time compiler to do its job and do the best compilation possible at the time and change overtime. But that cost is pretty heavy in a microservice where you tend to have less code, more focused and sometimes that microservice is not too focused on having data. Maybe it's resting point doing things.

In that model, the size of the data you dedicate to your application that contains your object is very small compared to the metadata, we call it the meta space in Java. That is containing all of the information about the classes themselves. What GraalVM provides is essentially getting rid of the whole just-in-time compiler and all of the metadata associated with it by precompiling everything at build time, just like a Go application.

[00:10:40] JM: If I understand correctly, what you're doing with Quarkus is you're looking at people deploying Java applications to Kubernetes and you're saying the memory footprint of these Java applications needs to be lower. The way that we're going to get it down is by being a little bit more aggressive with our pre-compilation and lowering the memory footprint of essentially the runtime execution, like hot path detection and memory retention stuff. Am I understanding it correct?

[00:11:18] EB: Yeah, that's a good explanation. We're from Red Hat. I don't know if we said that. Red Hat is on a journey to really accelerate the deployment of Kubernetes via our distribution open shift. We do believe in that model. Then we have also a massive investment in our Java middleware. We've been trying to go into a cloud native journey of our Java middleware and of course make it run really as good as possible on Kubernetes. That memory usage was really a bid blocker and we've been walking with our open JDK. So the normal JVM side of things that Red Hat has quite a few engineers on that to try and find the big reason for Java being heavier than its competition in that kind of space.

What we found is that by the nature of Java being very dynamic, being able to load classes at one time and a lot of interesting things like that, which gives a lot of power to Java. They had to keep a lot of metadata as you were describing. Also the Java middleware itself sort of abused the fact that Java was extremely dynamic. Every time you start your application, a framework will look at all of your classes. Try and find the one it's interested in and so on and so on. That leads to a lot of operation that happen at runtime and a lot of memory consumed to do that job.

When we saw this, GraalVM, essentially it's called ahead of time compilation. When we saw that model we realized, "Okay. That could be the game changer we've been looking for." Instead

of going through incremental improvements on the memory usage, really go to the – How do you call that? Chasm I guess? At least for the Java ecosystem. Making sense?

[00:13:03] JM: Yeah. I guess I didn't remember this about GraalVM. It does more ahead of time compilation than just-in-time compilation?

[00:13:15] EB: Yeah, they do – So that's the funny part, right? When you think about GraalVM being able to run JavaScript, you're thinking, "Well, they do just-in-time compilation and things like that," which is true. But in the case of Java, there is no just-in-time compiler. They literally compile everything at build time. That's why they can get rid of the whole just-in-time compiler. That's much less got to bundle. Also, it lets them limit the metadata that is necessary to follow just-in-time compiler to do its job. They are assuming something about the Java application that is somewhat uncommon to a Java developer. They assume a close-world assumption.

They assume that at build time, all of the cloud pass that will ever be used is known. They do static code analysis and do data code elimination to get rid of all of the unnecessary coding methods. That's why you also gain in memory.

It has a drawback, because as I was saying, a lot of framework in the Java ecosystem are built with this notion of reflection. Being able to load classes dynamically and so on. When we saw GraalVM, we said, "Okay. That's great." But the JVM itself cannot really solve the problem. The ecosystem also has to join the game, and Quarkus is really about making frameworks move to these build time closed-world assumption universe.

[00:14:41] JM: Framework. Explain in more detail what the problems with frameworks are. I'm thinking Spring framework, for example. Spring framework, if I want to run a Spring framework application on top of Kubernetes, are there some kind of issues with doing that today?

[00:15:01] EB: Yes. Well, it depends how much memory you're ready to pay for. What's the density of the application you want to have on your, say, cloud provider or Kubernetes deployment or whatever. Because what a framework does – By the way, that's not unique to Spring. App servers have the same models. Hibernate, which is the object relational mapper also use the same model

When you think about it, what a framework does, it will pass some configuration file. It has to read on XML file or some other formats to read what you ask it to do. Then it does what I described, this notion of class path scanning to the metadata, the annotations. That's what they call in Java around your code. This class is actually a class to be persisted. I need to be aware of that.

Then for those classes, these frameworks will want to do reflection to be able to dynamically invoke those methods. For example, you will want to inject an object into another object and this will be done dynamically. You do that. Then you build your internal model, which is the runtime model. In the case of object relational mapper, they would be generating the SQL queries necessary. Finally, the framework starts.

All of that work, except the very last part of finally the framework is ready to answer your request. When you think about it in a closed-world assumption, you can move them from startup time, which is what the entire Java ecosystem had been focused on and to now and you can shift that to build time and not have to do that at all at runtime. You're faster and you consume way less memory because the amount of code that is actually reading the configuration file and doing the reflection and preparing the framework to be ready and optimized is actually pretty massive. We're talking about a lot of classes that don't have to be loaded at runtime, which means smaller memory usage.

[SPONSOR MESSAGE]

[00:17:11] JM: This episode of Software Engineering Daily is sponsored by Datadog. Datadog integrates seamlessly with more than 200 technologies, including Kubernetes and Docker, so you can monitor your entire container cluster in one place. Datadog's new live container view provides insights into your container's health, resource consumption and deployment in real-time. Filter to a specific Docker image or drill down by Kubernetes service to get fine-grained visibility into your container infrastructure. Start monitoring your container workload today with a 14-day free trial and Datadog will send you a free t-shirt. Go to softwareengineeringdaily.com/datadog to try it out. That's softwareengineeringdaily.com/datadog to try it out and get a free t-shirt.

Thank you, Datadog.

[INTERVIEW CONTINUED]

[00:18:04] JM: We covered this in the previous GraalVM episode I believe, but just in case people have forgotten that, explain in a little more detail why is the memory footprint going to be lower? Use the Spring framework example. Talk about why this situation that you're describing with all these configuration and reflection stuff being in-memory, why is this so problematic and how does GraalVM reduce that memory footprint?

[00:18:34] GS: Maybe let's take an example. We made a few modifications to hibernate validator to fit into Quarkus and how we wanted to build the Quarkus framework.

[00:18:50] JM: Hibernate validator, that's like an ORM validation system?

[00:18:55] GS: Yes. It's more of an object validation system. The idea is you put annotations on your object, on your beans and you say, "Okay. This property must not be [inaudible 00:19:08]. This property must be a valid SKU or whatever." You add your annotations on your classes and the other task is to check that the instance validates the constraints you put on your object.

When we initialize hibernate validator, we have to gather all these metadata. We have to check for the annotations. We have to check for the classes. We have to initialize the constant validator and so on. This takes a lot of time. The idea was to move all these to build time. When you compile your application, we are gathering this information and we are initializing a number of object.

The nice thing about that is that a large part of the hibernate validator code is designed to gather this information and build the meta model. If you have built the meta model before, if you have built it at build time, then all these classes can just be removed from the image, because you won't need to use them.

In the end, you end up with far less classes in your image and you end up with less memory consumption in the meta space, because you don't have to have all these classes initialized. This is for one library, but imaging that we do that for all the libraries of the framework, you end up with all these typically – Emmanuel gave the example of the XML file. So if you take [inaudible 00:21:11] or our object [inaudible 00:21:14] mapper, we have it passing an XML file. For this you need [inaudible 00:21:21]. You need [inaudible 00:21:23]. You need [inaudible 00:21:25] and this is a massive amount of potential metadata that you will keep during all the life of your application.

When you move that parsing at [inaudible 00:21:36] time, and you can do that because you are in a closed-world assumption. You end up with all these classes gone from the native image. So you have far less memory consumed by all these classes. This is one part of the gain we have with GraalVM, and it's one of the interesting part of GraalVM.

But keep in mind that some of the walk we do, we also do it in a JDK environment. We support GraalVM. I mean, we take the better of it, but we also have some gain a JDK environment. Even using Quarkus with standard JDK, we will see that we have far less memory consume because we moved quite a lot of work at build time.

[00:22:29] EB: To us, the metrics that does matter – We're assuming people are moving to many more deployment units. Let's call it the microservice pattern just for simplification, but it could be functions. It could be not quite microservice universe. What's important is the number of request per second, per megabyte you're consuming, because the modern way to scale up your application is really to say, "I want to have an auto-scaler that will listen to some metrics and decide to deploy your second, or third, or fourth instance of my application dynamically and also scale down as necessary."

The fact that you can shave memory, especially the initially memory, for initialization of the frameworks that won't be used anymore after is going to be very useful to really get a massive advantage in numbers of requests per second per megabyte of your application on a given platform.

[00:23:31] JM: What do you need to build to make this a reality? This accelerated – Or I guess I should say lower memory footprint Java implementation?

[00:23:46] EB: Okay. Let's do it in two parts. Let me give you a little bit of the limitations of running Java on GraalVM and probably I'll let Guillaume explain what we call an extension, which is how Quarkus make a framework that we use, that we know, and shift its work as much as possible at build time.

GraalVM, as we said, is compiling everything at build time. It's assuming all of the code of the application is available. If you do that, you cannot do arbitrary reflection or arbitrary scanning, because you will have eliminated a lot of code that the compiler believes you're not using. If you can do reflection anywhere, it means any piece of code is potentially accessible. Therefore, you cannot do the [inaudible 00:24:29] elimination and all of the good and positive aspects of GraalVM are out the window.

You can still do reflection, and that's a core aspect of the Java platform, but you have to list manually the classes you need to do reflection on. They have toolings to try and help them, but I won't go too much into that detail, that amount of detail. It can be very tedious though to make sure you've exercised all of the areas where you need to do reflection, list those classes, all the fields and make that happen.

Same for Java is – The Java ecosystem loves this notion of proxy, which are classes that are dynamically generated. Again, you need to list them manually to provide the information to GraalVM. With the code that you have plus the list of reflections you want to do on specific classes, then GraalVM will go and compile your whole application.

That's all good when you do a Hello World or a very pure JDK-centric application. But when you start using frameworks like Spring or the whole Java ecosystem, every framework using annotations essentially, you will have to give to that framework – That framework will need to a reflection. You as a user will have to list, give that information to the framework, and that's a lot of work. That's somewhat impractical really. Then comes Quarkus.

[00:25:58] GS: The idea with our extension system is that for each library for which we need some GraalVM configuration, something like that, the extension will take care of that for you. For instance, in the case of – I don't know, hibernate validator, let's take this example again. Each bin you will validate, you will need reflection on that. You don't want to write GraalVM configuration file for that. So we do that for you. That one part of the extension system –

[00:26:33] EB: The reason we can do that is that each framework semantically knows which class it needs to apply reflection on, because you as a user have naturally put some sort of metadata. In the case of hibernate validator, it's the constraints. You say, "Hey! I want that string to always be an email." We detect the ad email annotation. Therefore we know we will do reflection on that.

The frameworks have the semantic knowledge of which class you as a user want to do a reflection on. You as a user don't have to provide that. It's just a framework that will interact with the extension and then provide that information to GraalVM.

[00:27:11] GS: Yeah. Another part of our extension framework is that at build time we will scan the annotations used by the framework. For instance, if you have your [inaudible 00:27:24], so your Rest services, you will have annotations on these methods. So by scanning the annotations at build time, we can get the list of all the method we will use as [inaudible 00:27:40]. We can generate code to initialize things eagerly at build time. That's really what we do in extensions.

One big part is let's do whatever we can do at build time in the extension framework and then we will generate code that will be executed later. In some cases, with UI extension, what you do is I scan my annotations. I generate some byte code. When I would start my application, I won't have to scan the annotations again. I will just execute the byte code and it's far faster and you don't need to scan all your class paths at runtime.

The second part is without this knowledge, we can simplify the configuration of GraalVM because, yeah, we know that these [inaudible 00:28:37] will serialize this object to [inaudible 00:28:41]. So we know we will use reflection and we will declare it to GraalVM and say, "Oh! We will [inaudible 00:28:50] this object for reflection, because it will be serialized [inaudible

00:28:57] at some point." I think there are two components really in our extension system. Move whatever we can at build time and simplify the configuration with GraalVM if you are using GraalVM.

[00:29:12] EB: Yeah. If you look at it from a user, the goal of Quarkus is to take the programming model they know, whether it'd be the Java ecosystem – I'm sorry. We're pushing a lot of acronyms here unfamiliar with Java. [inaudible 00:29:33] way of doing things. Whether it's be the Spring annotations. Whether it's be the way you persist entities. We make that run as they're used to in a "normal" Java ecosystem, but we do the hard work of making sure GraalVM has the right set of information to properly compile the application.

[00:29:56] JM: Quarkus makes my application run faster not just because of GraalVM but because of the – Is it called the hotspot compiler? Is that what it is?

[00:30:04] EB: The hot spot compiler is what is inside the normal JVM. Hot spot is really the just-in-time compiler I was describing. Yeah, that's a bit of a simplification, but that's it. GraalVM comes with essentially an alternative version of that, except instead of using a just-in-time compiler also compiling your code while you run it, it actually does it at real-time. That's some more clarification.

The other clarification is a code that is running on GraalVM might be actually a bit slower than a code running on hot spot, because it doesn't necessarily have the just-in-time capability to adjust itself as it sees how the code is actually used. Technically, your request per second, the pure request per second will be lower on GraalVM than it is on the Java hot spot. But you can definitely compensate that by the fact that you are consuming way less memory. Assuming your application is stateless, then deploying a second or a third instance will more than compensate on that.

What Quarkus offers is really a way for you to decide whether you want to run in the normal JVM universe or in the ahead of time GraalVM universe, and there are pros and cons for those. If you're very focused about application density, then the GraalVM aspect will be interesting. On the other hand, if your application is very memory heavy or you're very focused on the

maximum throughput for one instance, then hot spot will definitely be a better case for you. Quarkus abstracts that.

The other advantage that Quarkus has is essentially for the same amount of memory that in other cloud native Java-based platform would use. We run more request per second, because having less of these initial memory overhead, we literally have more memory for the application itself and for each of the requests and each of the request per second. Our middleware has been optimized to not have any bottleneck or as less bottleneck as possible for many years now. We're very confident about that kind of limitation.

[00:32:24] JM: I understand at this point that Quarkus is going to reduce my memory footprint. Is it also going to make my programs run faster?

[00:32:34] GS: The idea is really about improving the startup time. This is the first component of it.

[00:32:42] JM: That's true.

[00:32:42] GS: We want your application to start fast. In a microservice world, you will probably want to start multiple instances of your applications and maybe you will have autoscaling. If you have autoscaling, you'll want new instances to start almost instantly. Even more important when you are considering serverless and functions where you potentially scale to zero. You might not have an instance of your application and then you want to start on ten of them at the same time for a certain load. You will need your application to start instantly.

The idea of moving everything to build time is to really to improve startup time. If you take the exact same component configured in the exact same way, they won't be faster on Quarkus, because they have the same components configured in the exact same way. What will be different is that it will start far faster. Of course, we try to optimize Quarkus to run faster, but it's not magic. What is a big magic is how we start faster because of the Quarkus infrastructure and how it is designed.

[00:34:11] EB: But I would argue you will develop faster, which is an interesting aspect. I'm sure CTOs look at their bills and the headcounts is a factor. The reason you will develop faster is that be c Quarkus starts much faster, as we've been describing earlier in this podcast, we've been able to implement something that we call live coding and precisely live reload where you start Quarkus in dev mode and then you start typing code in your ID, whatever that is, VS Code, or IntelliJ, or Eclipse. Then you go back to your browser, you refresh to see the updated page of the updated resting point and you see it right away.

It looks like, okay, no big deal for somebody doing a PHP, for example, because that's all they've been running their program and their development forever. For a Java developer, the usual lifecycle is I'm coding. I'm deploying the application. So I'm packaging the application. I'm deploying the application, including – Then I'm including starting it, really. Then I can go and test. There is a bit of not enough time for a coffee probably, but quite a bit of a lag time.

With these live reloads, you really give some extremely short feedback loop between the change you're making and the results you're having. It sounds tiny, but it makes so much of a difference – We call it developer joy in marketing parley for us, but it is really very good to see the mistakes you make. Go fix it and come back.

The other aspect is the test suite. We've seen more classic Java applications, backend application. I'm speaking having pretty long test suite because of the slowness of the framework to start and preparing all of the metadata we've been describing. With Quarkus, this is massively optimized so you will see drastic benefit in the time it takes for test suites, for your test suite to run.

These are I think also key important aspect, not just about the pure cloud bill that you will get, but also how many people will be, or rather let's say the same team. How many more microservices they will be able to handle just by having a faster feedback loop and get the job done faster.

[00:36:41] JM: How do I begin to adapt Quarkus?

[00:36:48] EB: You can come to Quarkus.io, and then we have a way to generate your first project. It's called code.quarkus.io where you see all of the technologies we already support. What is important is you – I always joke and say that you already have 5 years of experience in Quarkus if you are in the Java ecosystem, because when you think about it, the application that runs, the piece of code that is run at runtime is not Quarkus code. This is the Rest server, in our case, [inaudible 00:37:22] relational map, object relational mapper, in our case, hibernate ORM. These match your technologies and also APIs you already know.

To get started with Quarkus, just the fact that you already know the Java backend ecosystem makes you – You know already 90% of what you really need. For people that are a bit less familiar with the Java ecosystem and more familiar with the pure Spring ecosystem, we have what we call a Spring compatibility API layer which lets you have your Spring annotations and they will be transformed at build time and run on Quarkus at runtime.

[SPONSOR MESSAGE]

[00:38:14] JM: If you are a SaaS or software vendor looking to modernize your application distribution to gain more enterprise adoption, checkout replicated.com. Replicated provides tools to deliver your Kubernetes-based application to enterprise customers as a modern on-prem private instance. That means your customers will be able to install and update your application just about anywhere.

Bare metal servers in a cloud VPC, GovCloud and their own Kubernetes cluster, vSphere. This is a secure way the your customers can use your application without ever having to send data outside of their control. Instead of your customer sending their data to you, you send your application to your customer.

Now, this might sound difficult and maybe you're not used to it because you're a SaaS vendor. You're a software vendor, but Replicated promises that recent advancements from tools like Kubernetes make it far easier than before, and the Replicated tools can help vendors operationalize and scale this process.

The Replicated tools are already trusted by noteworthy customers like HashiCorp, CircleCI, Sneak and many others. As a result, over 45 of the Fortune 100 already have an application deployed via Replicated in their infrastructure. That's a strong sign of adaption.

Go to replicated.com for a 30-day trial of the full Replicated platform. You can also listen to an interview with Grant Miller, the CEO of Replicated, that we did a while ago.

Thank you to Replicated for being a sponsor of Software Engineering Daily, and you can check it out for yourself at replicated.com and get a free 30-day trial.

[INTERVIEW CONTINUED]

[00:40:21] JM: Let's take a step back. Your perspective is that Quarkus is designed for a world of new application structures. Could we talk a little bit more about how you see application development changing and why Quarkus is a worthwhile project in that environment?

[00:40:39] EB: I got two theories. The first one is people have les to work on a given application that the pressure to get the business innovation out as fast as possible is higher than ever and the competitiveness here has just increased. The time you take to write your application, and it can be as long as knowing this new framework to how long it takes to deploy and run your test suite and so on is definitely shrinking.

You need to get down. That's the key important aspect. At anything you make somewhat simpler for the developer is going to be a huge gain down the road. People don't have time to explore the technology and look at it for days and days. They probably do a sneak peek of one hour if they like it. Another half a day if they like it. They'll start to give it a try on some of your real projects. That's one aspect.

The other aspect to me is because we're in a world of automation and the notion of deployment, that used to be a fairly complex process where everyone was sweating and making sure things were tripled checked before you were deploying in somewhat fading away. People are embracing those, what I call dynamic orchestration platforms. I do have Kubernetes in mind, but you could think of the cloud provider as another one of those.

My theory that because you have that, you can do microservices, meaning you can split your application into smaller bits because you can deploy them without too much cost, because all of the maintenance and making sure the app stays up is somewhat handled by the platform. Instead of neutralizing this cost into one monolith, you can split your stuff into microservices.

The reason you do microservices, to me, because it's still costly. Each microservice is simpler, but then the communication between microservices makes the entire system more complex. Why do we pay that cost? To me, we do pay that cost to be more agile and really address the business needs. You will probably do less features from a throughput point of view, but you'll be able to deliver the right feature much faster.

If at some point in one area of your application you decide that the technologies you've chosen is definitely not the right one, you'll be able to scrap entirely the code and even potentially the database system underneath and, for example, decide to go from a relational database to an in-memory database and rewrite entirely the code. That is manageable, because it's very well-known piece of your application and it's not like a big, gigantic things with lots of interaction with other areas.

[00:43:28] JM: How did you guys get involved in this project? What were you doing beforehand and what caused you to start working on Quarkus?

[00:43:37] EB: I'll start because I was a tiny bit earlier than Guillaume into that. I'm the cofounder of Quarkus and we decided to go for Quarkus from – In some ways, we've started Quarkus many, many years ago, as I was describing the need to optimize our middleware into much more memory-constrained environment that's a dynamic orchestration platform we're essentially offering to the world.

We've been optimizing our frameworks. As I was saying, interacting with our JDK team to try and find incremental improvements in that area. When we saw GraalVM, we saw the potential and we saw that because we were somewhat knowledgeable with enough piece of the middleware ecosystem, we could try and move that ecosystem from startup time to build time.

We did something a big weird for Red Hat. Red Hat usually is open from the get go and we just put it out there on GitHub or whatever and then people try.

In this case, we started with in somewhat of a secret, like a stealth mode even within the company and we got a lot of pushback for that, because that was definitely not the usual way of doing things. But we started with a very small team and said, “Okay. Let’s take those two or three or four technologies and let’s try and make sure these build time stuff works and that we can really shift the Java ecosystem to these new universe.” We did that and we iterated three months after three months by increasing slowly the team size and then bringing all the Red Hat middleware teams onboard.

[00:45:17] GS: Yeah. That’s when I joined. I’m the Hibernate validator project lead and I joined the Quarkus team to improve the integration of Hibernate validator and also make some changes in Hibernate validator to be more integrated into the Quarkus way. That’s how I joined, and I started working on a lot of [inaudible 00:45:44] fixing bugs, writing documentation and whatever. That’s how I got involved in the project.

[00:45:54] EB: Just to take a step back. On Red Hat middleware, I’m the chief architect for data. I’m usually around the data related projects. So the persistent framework, that data grid, which is a distributed key value store. Change data capture. There is a project called Debezium and so on.

But when saw that potential into shifting the Java ecosystem, it was sort of all hands on deck as far as let’s give it a try and let’s make sure everybody in the organization see it as with the potential it has, and it required to break the silos we tend to have within organization and say, “Okay. Let’s take the key people across areas and do a concerted effort.” Because not only does it make user Java application much smaller in memory usage and so on, but it’s also going to be a very key aspect of Red Hat middleware, which is primarily written in Java. It’s a big win for us as Red Hat offering those middleware as services, just like cloud providers, but also it’s kind of for free that actual end user can also write the application with the same technology.

[00:47:06] JM: What’s the hardest engineering problem that you’re working on within the Quarkus project right now?

[00:47:12] EB: By the way, the hardest problem is not engineering. It's making the right choice of where we do invest. But back to your question – Guillaume, do you want to discuss the whole reactive rework that we're doing maybe?

[00:47:25] GS: Yeah. Right now we are working on totally new HTTP layer based fully on vertex and [inaudible 00:47:35]. At the beginning of Quarkus, we integrated a component called [inaudible 00:47:41], which is an implementation of [inaudible 00:47:43] and it was basis for all our HTTP layer. The issue with that is that we have – In Quarkus, we have integrated imperative way of coding and also the reactive way of coding. Having both in parallel and having two different implementation of things, you couldn't share resources, and that's something we are working on in Quarkus. Trying to use as less resources as possible.

The idea of this HTTP layer rewrite is to put everything on vertex and base everything on vertex and have only one [inaudible 00:48:34] dealing with everything. It's also an improvement in CPU usage. It lowers the number of context switches we have. It's really something that we wanted to do for quite a longtime. The thing is that it's really – While it's a very low-level layer. So when you change that entirely, it comes with a lot of challenges. You do that while a lot of other people are working on those part of the code. It's not an easy thing and you also have to co-design some key part of the code. While working on this new HTTP layer, we also have to rework all our security layer. It's really a big engineering challenge and also a big organizational challenge because, yeah, we have to make progress while people are working on very low-level layer.

Yeah, that's what keeps us busy right now. We have released the first step of this journey. We have a few [inaudible 00:49:44] coming with improvement on top of that. Yeah, the idea is really to be able to unify everything by using a vertex and having reactive at the core of Quarkus.

[00:50:06] EB: The reason we want just essentially one pipeline to process all requests whether you end up doing the blocking way from a problematic point of view or the non-blocking way is because, again, it's about resources. It's about memory. Every extra class we bring to the Quarkus platform to do something means a bit more memory use that we could avoid and that we could be better used to serve the request per second the user is looking after.

Remember, microservices. When splitting your application into many units of deployments means every extra bit you save counts. That's an important aspect for us to try and say, "Let's not try and have two framework to do the same thing in slightly different way, but try to unify that." It's not for the engineering beauty of things.

The reactive aspect, we wanted reactive to be at the core, and it's again a resource conception aspect. With a non-blocking model, you can definitely save from a memory usage point of view and in those instances, you do require to serve your request. It's heavily used in heavy data ingestion needs. Any IoT related platform would definitely want to have a very low-level, a very low consuming end point. That's an important aspect. But reactive, it comes with its challenges from a programmatic point of view. A user will be able to stay blocking if it's just easier for him or her and if they just are okay to pay the extra resource cost.

[00:51:41] JM: Have you all seen any other cool projects within the GraalVM ecosystem that stand out?

[00:51:47] EB: For us, fundamentally. This is a new era for Java. There were some pretty heavy eras, but not a lot of them in the Java ecosystem, because it's fundamentally a fairly stable ecosystem. But we've seen quite a few turn at some point that really changed the landscape into how you write an application. Annotations was one.

We see these shifting of the frameworks at real-time. Probably the biggest new era that at least the Java backend as ever seen. Whether it'd be Quarkus in the end or something else, of course we do believe Quarkus will be the one, but this will be a change forever. There is Quarkus. Another one is Micronaut, which came to the problem slightly before, but also before GraalVM was something. They also try and shift as much as possible at real-time.

What they didn't get from the get go and don't have is really literally a bit of a dialogue between a framework. Well, the extension model that we described is really a dialogue between the framework that people use, whether it'd be the dependency injection framework or the rest end point framework or whatever, and GraalVM to really provide it the right information. Also, try and have – We didn't go too much into that details, but the notion of dead code elimination is very

sensitive. If at some point the compiler is not quite sure whether your code is used or not, or course you will be conservative and keep all of the code.

But because you're at the application level, you know for example that this application will never use, say, a second level cache. You can help the dead code elimination by simplifying a little bit the code and making sure second level cache is not only disabled, but entirely removed from the codebase and then you save, again, a little bit more memory.

Yeah, Micronaut is also a cool framework in that area. There is one whose names escapes me, which is around writing common line interface, tools essentially, but in Java, which was very prohibitive in the past because starting the JVM was – Well, first of all, you had to install the JVM, plus your tool, and then starting the JVM is like a two-second proposition, let's say, plus the memory usage. If you want to write a very simple and very reactive common line tool, that was just prohibitive. There are some frameworks that are built around GraalVM to improve that ecosystem.

Is it [inaudible 00:54:27] or something like that, Guillaume? Do you remember?

[00:54:30] GS: [inaudible 00:54:30].

[00:54:31] EB: [inaudible 00:54:31]. There you go. I was close. Do you see any other that I might have missed?

[00:54:36] GS: I think these are the two most interesting projects apart from Quarkus.

[00:54:43] EB: By the way, I think on the GraalVM – We're less verse in the GraalVM polyglot side, but I think people see a lot of interesting things with our support for GraalVM. It's not my thing really, but I believe people really appreciate the way GraalVM runs are and they can even mix and match a bit of Ark doing some work, extracting information and then using it in another part of their application, which is in plain Java. That's one aspect.

When we say polyglot in GraalVM, it's not only that it runs more than one type of language, but it runs all of them in the same piece of code and you can literally interact between one language piece and another language piece without interrupt cost.

[00:55:32] JM: All right, last question. Imagine it is 5 years into the future. How will Quarkus have changed my life?

[00:55:40] EB: That's a tough one.

[00:55:43] JM: Even if it's in imperceptible ways.

[00:55:46] EB: I'm trying to think five years in the future. Overwhelmingly, we've seen like awesome feedback on the Kubernetes and people are absolutely interested into learning about that. We knew we were on to something, but you never know how much that is. The memory consumption was absolutely massive. Much bigger than we even anticipated, at least at my level. I even had [inaudible 00:56:11] come to me and say, "You know, I love your stuff. I think I'll talk to my dev team too so that they go and adapt it because I'm sick of those Java processes taking so much memory and so much time to start."

If anything, Quarkus would have – Some people have predicted the death of Java for quite for a while, but I think it will definitely keep the next prediction for quite a few years with this new evolution. To me, the other aspect is if we keep building the ecosystem properly and keep having these live reload model that will change the way people, new generation come at Java and how they write application will probably vastly change instead of being very, for example, test-first centric, because that was the smallest unit of work to really deploy part of your application and make it run. It would probably be much more live reload centric where you code, you see the result right away so you can even adjust the UI while you also add a new field in your database and your object. That is to me a game changer.

There was actually a framework called play framework, the V1. That had really a very good user experience and we've been shamelessly stealing all of the good ideas they had at the time, but also improve on the memory usage. What we knew is that you can give people a massive memory improvement, but if development experience is really odd or crap, it would just not

adapt it. They don't have the time. We also needed these awesome user experience as well as the memory advantage.

I don't know. Let me do a very stupid prediction. Because we save so much money on the cloud bill, maybe there'll be more people in your guys' team and you can achieve more. I don't think that one will be true, because the money will be reused elsewhere, I'm sure. But that would be a good one.

[00:58:18] JM: No! I hear these cloud providers pass on their savings to the customer.

[00:58:22] EB: Oh, yeah. That must be right.

[00:58:25] JM: All right, guys. Well, Guillaume, anything to add?

[00:58:29] GS: Well, yeah. I want to work something about our community. We have a very open community at Quarkus, a very inclusive one. If you are interested in starting open source, working on a new project, be creative with new ideas. You will be very welcome. We have more than 150 contributors now and it's growing very fast. If you feel like learning, you'll be welcome.

[00:59:04] JM: All right, guys. Well, thanks for coming on Software Engineering Daily. It's been fun talking to you.

[00:59:07] EB: Thank you very much. Yeah, it was a good conversation.

[END OF INTERVIEW]

[00:59:20] JM: If you want to extract value from your data, it can be difficult especially for nontechnical, non-analyst users. As software builders, you have this unique opportunity to unlock the value of your data to users through your product or your service.

Jaspersoft offers embeddable reports, dashboards and data visualizations that developers love. Give your users intuitive access to data in the ideal place for them to take action within your application. To check out a sample application with embedded analytics, go to

softwareengineeringdaily.com/jaspersoft. You can find out how easy it is to embed reporting and analytics into your application. Jaspersoft is great for admin dashboards or for helping your customers make data-driven decisions within your product, because it's not just your company that wants analytics. It's also your customers.

In an upcoming episode of Software Engineering Daily, we will talk to TIBCO about visualizing data inside apps based on modern frontend libraries like React, Angular, and VueJS. In the meantime, check out Jaspersoft for yourself at softwareengineering.com/jaspersoft.

Thanks to TIBCO for being a sponsor of Software Engineering Daily.

[END]