EPISODE 11

[INTRODUCTION]

[0:00:00.4] JM: Swift is a language that is most commonly used to write apps for Apple client devices, such as iPhones. Since being released in 2014, Swift has become one of the most popular languages due to its high performance and developer ergonomics. In 2015, Swift was open-sourced, creating the opportunity for Swift to be used outside of the Apple ecosystem.

If you write an iPhone app today, your front-end is in Swift and your backend is probably in Node.js, Java, or Ruby. Engineers are working to port Swift to the server so that the Swift developer experience is isomorphic, the same language on the backend and the front-end. Chris Bailey is an engineer at IBM working on Kitura, a Swift web framework. In this episode we discuss the history of Swift, why it is so appealing to developers, and why Swift could become a server-side language with as much popularity as Java.

Software Engineering Daily is having our third meet up on Wednesday, May 3rd at Galvanize in San Francisco. The theme of this meet up is fraud and risk in software. We'll talk about ad fraud and fraud that Coinbase faces. We will have great food, engaging speakers, in a friendly, intellectual atmosphere. To find our more, go to softwareengineeringdaily.com/meetup. Now, let's get on with this episode.

[SPONSOR MESSAGE]

[0:01:34.4] JM: At Software Engineering Daily we need to keep our metrics reliable. If a botnet started listening to all of our episodes and we had nothing to stop it, our statistics would be corrupted. We would have no way to know whether a listen came from a bot or from a real user. That's why we use Encapsula to stop attackers and improve performance.

When a listener makes a request to play an episode of Software Engineering Daily, Encapsula checks that request before it reaches our servers and filters bot traffic preventing it from ever reaching us. Botnets and DDoS attacks are not just a threat to podcasts. They can impact your application too. Encapsula can protect your API servers and your microservices from

responding to unwanted requests. To try Encapsula for yourself, go to encapsula.com/sedaily and get a month of Encapsula for free.

Encapsula's API gives you control over the security and performance of your application. Whether you have a complex microservices architecture, or a WordPress site, like Software Engineering Daily. Encapsula has a global network of over 30 datacenters that optimize routing and cacher content, the same network of datacenters that is filtering your content for attackers is operating as a CDN and speeding up your application.

To try Encapsula today, go to encapsula.com/sedaily and check it out. Thanks again, Encapsula.

[INTERVIEW]

[0:03:15.4] JM: Chris Bailey works on Swift at IBM. Chris, welcome to Software Engineering Daily.

[0:03:20.7] CB: Thank you for having me, Jeff.

[0:03:22.2] JM: Yeah, it's great to have you. Today, we're talking about Swift on the server, but I'd like to start with a discussion of Swift itself. Swift is a programming language that is most commonly used to write applications for the Apple ecosystem. Why was the Swift language created?

[0:03:40.6] CB: Swift was created by Chris Lattner who had previously worked on LLVM, which is kind of like a platform and a toolkit for creating and running programming languages. It has an implementation of C [inaudible 0:03:55.2]. I think when he joined Apple, one of the things that was kind of clear is that the program for the Apple ecosystem, you have to use Objective-C, which has been around a longtime, and I guess it's had to say hasn't had a lot of modernization done to it.

Swift was this opportunity to revamp the developer ecosystem for the Apple ecosystem and to create a new language which those developers would find easier to use and could actually provide their input to. The goals of Swift are declared that it should be fast, expressive, and

safe. Basically, those are the characteristics that developers really want to build their applications.

[0:04:51.3] JM: Swift came out in 2014, but the iPhone has been around for much longer than that. Before Swift, as we mentioned, iOS apps were written in Objective-C and both Swift and Objective-C compile to the Objective-C runtime. Give an overview of the compilations paths for Swift and Objective-C.

[0:05:14.4] CB: That's actually not completely true.

[0:05:19.5] JM: My Wikipedia has failed me.

[0:05:22.7] CB: If you're running on Linux, there is no Objective-C involved at all. When you compile Swift code, what you're actually creating is a native binary, so an executable or a library for that platform. Now, what it does do is it does seamless bridging with Objective-C. If you have existing Objective-C code that you're using somewhere in the Apple ecosystem, whether it's desktop Mac, or it's iOS, or it's watchOS, then you can reuse that code, because you can call Objective-C coding, use it from your Swift code.

If you're running inside the Apple ecosystem, there's a good chance that you've got some Objective-C code running something inside your application. On Linux, there is no Objective-C runtime that's provided, so everything is native Swift code that's compiling down to native modules for that platform.

[0:06:25.3] JM: When Swift was introduced, it was described as Objective-C without the C. I think you could also describe it as having these collections of features that allow it to reach for those — What is that? The ideal — You said security speed. What were the pillars? The pillars of Swift that you mentioned?

[0:06:49.0] CB: Safe, expressive, and fast.

[0:06:51.6] JM: Safe, expressive, fast.

[0:06:53.8] CB: Yeah. The aim is that it's safe and that the language itself makes it harder to program with bugs. One of those concepts is something called optionals. One of the classic problems that you get in programming is null pointers. In Java, if you have a value which you've initialized which you created, but you haven't initialized it, it's going to have null as the value. If you try and use null, so you think it's like a string, and then you try to access something from that string, you'll get a null pointer exception.

If you do that in C code, so you have uninitialized values that you try to access, then that could be somewhere random in memory and you generally end up with a crash. Swift tries to avoid that problem. Yes, crash or a [inaudible 0:07:47.4]. Yeah.

Swift tries to avoid that problem by using optionals. The idea is when you have a value, you can mark it as being optional and that means it has scope to be nil, or you could have something which is not an optional which cannot possibly be nil. Most values are non-optional, and that means they must be initialized to something. You know you're never going to get this null pointer exception of this [inaudible 0:08:17.8] trying to access the value, because you know it cannot possibly be nil.

If you want something that could be nil, then you have to mark it as an optional. Before you can convert something as optional to something that isn't optional, then you have to do a check around it to make sure whether or not it's nil. That means you've actually got this sense of types which you can safely access without worrying that it may or may not be nil and having to wrap everything you do in these nil checks. That's just one of the things that they've implemented to make it easier for developers to write code which have left bugs.

[0:08:55.5] JM: We had a show recently about LLVM. As you mentioned, the creator of Swift is Chris Lattner who also created the LLVM. The LLVM is this ecosystem of compiler tools. How does Swift leverage LLVM?

[0:09:13.1] CB: LLVM lets you effectively build multiple languages, and Swift is one of those. When you're writing Swift and you compile it, it's actually using the compiler tools from LLVM to take your Swift code and convert it into the machine assembler effectively which is going to be executed on the platform.

[0:09:36.3] JM: in the LLVM show, much of the discussion focused around this intermediate representation that there's a lot of work that goes into being optimized. Any code that compiles down to the intermediate representation language gets to take advantage of the intermediate representation optimizations. It's almost like the Java bytecode of — It's a more flexible platform than Java bytecode.

I guess you spent a lot of time in the Java ecosystem. I was looking at your background, you spent 15 years in the Java ecosystem. How does the LLVM platform compare to the JVM?

[0:10:27.0] CB: In a lot of ways, the concepts are the same. When you're doing compilation, whether it's ahead of time, which is what you have with Swift. When you write your Swift code, you compile it, you get these binaries, and that's what you execute as a single upfront compile stage. When you do that, the first thing that happen is the code that you write gets parsed and that creates, yes, this intermediate representation language. At that point, you can start to do the optimizations.

You can look at things like loops and you can say, "Well, I know that people typically write loops such that I go around it a whole number of times and break out only once, say, a hundred iterations has being completed. So usually, you go around that loop several times before you exit. You can use that to optimize to fast path to be to iterate around and the break out to be the exceptions.

The way LLVM works, because you have this intermediate language, then it doesn't really matter what the front-end programming looks like. Once that's parsed and comes down to the intermediate language, then those compiler optimizations can be applied regardless of what the front-end syntax looks like. That's one of the ways in which Swift, even though it's a relatively new language, has quite a lot of compiler optimization in it, because it gets to reuse what was already there in LLVM.

Those concepts are actually very much the same in Java. The difference is java kind of has two compilation steps. The first; you write your Java code and you run the Java-C, the Java compiler. That creates bytecodes. Bytecodes are just portable. You can pick them up and you

can run them on any platform. Unlike Swift, where you compile on platform that generates a binary smart platform, the input of compiling your Java code is these bytecodes in a class file and you can pick up that class file and run it on any platform at the JVM.

Now, when you run it on the JVM, that JVM still has to get to the point that it's running assembler instructions for the machine that it happens to be on. The way Java does that is to do that compilation down to machine code at runtime. So it uses something called a JIT, just-in-time compiler, to do that. The concepts of just-in-time compilers are actually very much the same as they are in LLVM. It takes the bytecode and it converts that into an intermediate representation language, an IRL. From there, it applies the optimizations before compiling it down to the machine code.

Now, one of the slight advantages that you get from the JVM approach of doing this at runtime is back to that loop example that I gave you. A static compiler, one that compiles ahead of time, can only guess how a loop is being used. It's going to assume you'll go around the loops several times and then break out only once you've gone through these multiple loops around the cycle.

A just-in-time compiler has the advantage that it can actually profile the way the code is being executed, because it compiles at runtime. It can see how your program is actually running and it can say, "Actually, this loop is different than normal." Typically, it will not go around the loop. It will do a check and it will break out immediately, but only in the exception case does it go around the loops several times.

It can actually optimize the reverse way to what you would typically have because it can actually watch the code and see the way it works. That means that there's actually slight advantage to just-in-time compilation. That means that it can achieve higher performance than you can get from a static compiler like you get with Swift and LLVM.

Now, better performance. That sounds good. It does have a tradeoff like everything in this world, which is that it needs more memory because it has to run this compilation in memory at runtime. Secondly, it has the disadvantage that because it has to compile it at runtime, your initial performance isn't as good.

Overtime, its performance gets better, so it's out at the boxed performance, as it's called. It's not as good as you get from ahead of time compilation from something like Swift. For very long running applications, just-in-time compilations tends to be good. Which is why, interestingly, there are a couple of people looking at adding just-in-time compilation to Swift, to allow that intermediate representation to be converted to machine code at runtime for Swift rather than ahead of time.

[SPONSOR MESSAGE]

[0:15:30.3] JM: Dice.com will help you accelerate your tech career. Whether you're actively looking for a job or need insights to grow in your current role, Dice has the resources that you need. Dice's mobile app is the fastest and easiest way to get ahead. Search thousands of jobs from top companies. Discover your market value based on your unique skillset. Uncover new opportunities with Dice's new career-pathing tool, which can give you insights about the best types of roles to transition to.

Dice will even suggest the new skills that you'll need to make the move. Manage your tech career and download the Dice Careers App on Android or iOS today. To check out the Dice website and support Software Engineering Daily, go to dice.com/sedaily. You can find information about the Dice Careers App on dice.com/sedaily and you'll support Software Engineering Daily.

Thanks to Dice for being a loyal sponsor of Software Engineering Daily. If you want to find out more about Dice Careers, go to dice.com/sedaily.

[INTERVIEW CONTINUED]

[0:16:50.0] JM: You're talking some performance benefits of Swift. Certainly, those performance benefits are relevant to the popularity of Swift. Swift has quickly become one of the most popular languages around. In 2015, it won first place in Stack Overflow's top languages survey. In 2016, it won second place.

Performance is not the only reason that people love Swift. There are a number of developer ergonomic aspects of Swift that people love. What are some of those features? I know you mentioned one earlier in terms of the avoidance of null pointer exceptions. What are so the other ones?

[0:17:38.8] CB: Optionals is a good example. Another one that you can do is — Let's say you're writing a function, and that function returns a value. You're just doing some kind of simple check to see whether something is on or off. You expect it to return true or false.

Now, in Java, if you called that function but didn't actually do anything to pick up the return value, Java will compile it. That's fine as far as it's concerned. You called a function, ignored its return, and that's okay in Java terms.

In Swift, the complier actually gives you a warning. It says, "You called a function, which returns a value and you didn't actually pick up that value." That's a compiler warning, which helps you understand, you probably should have done something with that return value. It's another way which is trying to help the developer not make mistakes and insert bugs into that code.

Now obviously, maybe you intentionally only have a return value at something that you may want to know but you don't have to know. They also allow you to mark that function as having a discardable result. So you just add an annotation to the function as the function as the function writer as apps discardable result. Then, people using that function don't have to do anything with the return. If you don't mark it as discardable, and they don't do anything in return, then they get a compiler warning to say you probably just inserted a bug into your code. There's many, many more like that.

One of the really cool things about Swift is there's something called Swift evolution, which is both a mailing list and a mechanism so that anybody can actually raise a suggested change to the language. This is enabling those community of developers to actually say, "Have you thought about doing X as a way of making a programmer's life easier?" That tends to be fairly unique. Most of the languages I've worked with in the past, there's a core team of people that decides what the language should look like. They implement features, and other developer just get it.

Whereas with Swift, they're taking these approaches saying, "Well, let's ask the people who have to write Swift code what they think would be useful to have in the language and how things should change." Even stuff that comes from the Apple core team working on the language goes through Swift evolutions. They propose the code change to a programming style and the community, so anybody that wants to sign up for the mailing list or go and look at the GitHub repo can comment on whether they think it's a good or a bad or how it would affect them.

So I think that's unique and it's actually a really, really neat improvement to help build a language that developers actually want to use by having the developers provide their input on what they want it to be.

[0:20:52.7] JM: Swift has been used on client devices for several years now. Many of the iPhone apps that I use are written in Swift. Why should developers build server-side apps with Swift?

[0:21:05.8] CB: There're a number of reasons, but the first of them, let's say you are one of those developers building an iOS or a watchOS or a macOS app. You're building an application for an iPhone. There's a good chance that that application needs to have some data store or functionality which you need to have off the device.

There are many examples of this. Let's say you're building a to-do list and you want to be able to save that so that you can access it from your laptop as well as from your phone, or — I don't know? You got a game and you're trying to store high scores, so you can compare them to other people. To have that, you need something off the device. You need some backend server, and that server will work with the database or could work with many other things.

The advantage of having Swift on the server is that I, as the developer for the iPhone app, I can also write to that server piece, which is going to take the data and store it in a database. That means I don't have to learn two programming languages or I don't have to work with another team. I can do both parts of that myself. That's really the first reason why it's interesting, purely, to developers.

Some of the other reasons why Swift on the server is interesting is back to the characteristics of being fast. Actually, not requiring a lot of memory is quite of an advantage. That comes from being a language designed to running on a device. It's very, very low memory footprint. It doesn't take a lot of RAM in which to run that server application and the same features as being expressive and safe means that it's an interesting language to be used on the server as well.

[0:23:08.5] JM: If we're talking about the other languages the people might be using the server, those are Java, Node.js, Ruby, primarily maybe Python. All of these languages have some of the developer ergonomics of Swift, though Swift was created later.

It's safe to assume that with Swift, these were built in from a more fundamental point of view. I don't want to call them bolt-ons, but it was not until Java 8 that this emphasis on functional programming entered the language.

We could talk about those, but I guess let's focus on performance. I know that Java is the performant of the three. Between Java and Node.js and Ruby, Java is the most performant. I know a lot of Ruby shops end up making their application in JRuby. They start with their Rails app and then when they get to a point where they need the performance of Java, they build their Ruby app to compile to Java or to compile it to Java bytecode. I don't remember exactly how it works. How do these other languages compare to Swift on performance?

[0:24:38.0] CB: That's one of the areas where Swift comes in pretty well. Some of it is down fundamental characteristics. Swift, like Java, is a typed language. What that means is when I declare a variable or a constant, so I declare a thing, I declare what type it is. I say, "This thing is an int, or it's a long, or it's a string, or it's a hash map." I have to state exactly what it is.

That's true of languages like Java. It's true of languages like Swift but also for things like C. The other end of the scale is something like JavaScript, whether that's in the browser on the server in Node.js. When you write JavaScript code, you just say it's a variable. You don't actually give any information about what type that is.

The side effect of that is when I create a var in JavaScript, it can store an int or a long, it could store a string. In fact, it can actually be a function. That makes it somewhat easier to program. I

just keep declaring var. They don't have to think ahead of time about what's going to be stored in it. It's actually makes it slightly quicker to develop. The downside is when we compile that code, once we actually have to compile that down to machine code, in an assembler instructions, the challenge is knowing what the things are.

In Swift and in Java, because I declared, say, two values as an int and I put a plus sign between them, so I want to do the plus operator. What will happen is Java and Swift will both go, "Okay, those are two ints, and you want me to app them together? That's a single assembler instruction. That's an [inaudible 0:26:34.0]. I can app the two together and to return the value in a single instruction.

To do the same thing in an untyped language like JavaScript where I got to vars, the first thing the compiler has to do is actually interrogate what's inside those vars to work out what it is, because if it's two ints, it's still going to use the [inaudible 0:26:56.4] instruction, but it has to put [inaudible 0:26:58.8] in place to understand that var A happened to be an int and not a function or a string, and var B happened to be an int and not a function or a string.

Having languages which are typed means that they execute much faster at runtime, because it's easier for the compiler to know what it does and to create the minimum number of assembler instructions to execute it.

That means typed language like Java and Swift are much, much faster than untyped languages like JavaScript and Ruby and so.

[0:27:31.9] JM: I saw you give a presentation where you were discussing the importance of not just Swift high performance but also the fact that it has a small memory footprint. This is particularly important as you mentioned in this talk that I saw in the time of cloud computing, especially today where we are getting closer and closer fit between the size of our programs and the size of the virtual machines, or containers that we're renting out from cloud service providers. How important is the small size of Swift's memory footprint to somebody who is developing a server-side application that they're going to be deploying to the cloud?

[0:28:20.7] CB: The interesting thing there is anybody that wants to deploy an application to the cloud will soon start to understand that you have to deploy into some kind of container, so you're kind of renting a virtual machine, or a virtual bit of hardware that your application is going to run in.

It pretty much doesn't matter which cloud provider you go to. They charge you an amount of money according how much memory that container needs. For example, in Amazon, if I wanted a nano instance, I get charged something like $5 a month for 512 megs-worth of memory. If I want 1 gigs-worth of memory, then I get charged twice that. If I want 2 gigs-worth of memory, then it doubles again. The amount that they charge you is actually based on the amount of memory that you want to use.

Usually, for clouds, you actually have to use [inaudible 0:29:26.8] quite a large container before it starts giving you more CPU. Basically, the charge rate is done by memory, not by processing power or such. The smaller your application is, the less memory its needed, the cheaper it is for you to run in the cloud.

Running something like Swift actually uses less than half the memory of running something like Java, and that means that if I have a large server application which has to run many, many instances in order to deal with tens of thousands of users, if I've written in Swift, it would cost me half as much in terms of hosting costs in the cloud as if would if I've done it in Java. This is kind of new breed of economics of running an application based on the cloud, which is gone from how many CPUs does my machine have, because it's the CPU that determines how much sprocessing I can do.

In the cloud, you get charged by memory and the amount of CPU you get kind of comes for free with that. If I can build a server application that uses half memory, it cost me half as much to run it.

[0:30:38.3] JM: This is slightly off topic, but how much of an application — Of a typical application's cost is network IO? Because I think they also charge you based on network IO, right? It's not just the memory footprint.

[0:30:52.1] CB: They do. Yes. If you consider that — The amount of network IO you've got basically depends how many users you have, not what language you happen to be receiving the requests or responding to the requests in. If you're building server applications which are using a REST API, then that traffic is based on the number or users. That's kind of a fixed cost regardless which language you implement the server in.

But, yes, that is a factor for running the application. The hosting cost itself is based almost entirely on the amount of memory that you're using.

[0:31:31.7] JM: Okay. Sure. That's the highest order bit basically, is the size of the — The memory size, memory footprint.

[0:31:41.0] CB: Yeah.

[0:31:41.4] JM: Okay. Great. I need to do a show about cloud economics.

Isomorphic development is this term that's used to describe development where the language is the same on both the client and the server. I think it goes without saying that isomorphic development is desirable. We would rather have all things being equal. The same language on the client and the server so people can move fluidly between client and server. We don't have silos in terms of language understanding.

One valuable aspect of isomorphic development is that you can actually choose to execute code on either the client or the server depending on how much available resource the client has. This is not going to be true for every aspect of an application, but there are certain portions of the code that could execute on the client or the server.

I've seen you talk about this when you're discussing isomorphic development in Swift and it got me thinking, because I don't know much about this isomorphic development pattern. Are there libraries for intelligently managing how much code executes on either the client or the server, or do you just deploy the code to both the client and the server and you have some kind of flag that you turn on or off depending on how much battery life is available on the phone, or how much — What version of the phone it is. How do you manage that type of isomorphic development where

you have code that can execute either on a client or the server? How do you manage that intelligently?

[0:33:21.0] CB: I'm not aware of any framework that does it intelligently today, and it would probably be an interesting thing for someone or a startup to actually produce.

[0:33:33.9] JM: It'd be useful.

[0:33:34.0] CB: There's some kind of classic use cases for it and certainly in iOS development in Swift. The first one is going to be your obvious — You've got users who are complaining about how quickly the application drains memory and it's because you're doing these huge computational tasks inside the device which could be uploaded to the server. In that case, one of the easiest things you can do is just say, "Okay. I'm going to release an update of the app that actually does the processing of this thing in the server rather than the client's name."

It makes a request at the server and gets the results back and the server is actually going to do that big computational task, which would use a lot of battery if it was on the device. You got that idea of being able to drag code around based on testing, whether that's testing you did ahead of release or based on user feedback.

Another way of looking at it is offline mode. When you're developing stuff for a phone, there's a good chance that you've got function that you want to be able to use offline as well as online. You could run the app such that if you're offline, there's a certain set of capabilities that we can execute on the device. If you do have a network connection, then we can have a server do that instead and you're using exactly the same code in both places.

We're just offloading it to the server because network is available and it's best done on the server because of maybe batter reasons, maybe data it can access, maybe can get updates from another service. For offline mode, you have to have some of that function available locally.

If your backend was written in a different language, then you'd have to implement the same function twice. If you've got the same language in both places, not only are you being able to do both parts with one development team that you're reusing code between both parse.

[0:35:43.6] JM: For engineers that are building in the Apple ecosystem, Swift on the server gives them this integrated developer experience. When I was researching for the show, I found plenty of people who were like, “Oh, Swift on the server is great. I don’t have to write my server-side code in Noje.js anymore.”

Although, this is the same reason that people choose React Native together with Node.js, or if we’re not talking about the Apple ecosystem, people choose Node.js together with some frontend JavaScript framework. I think this was the thrust of what made Node so popular in the first place was people were saying, “Oh my gosh! I have to write a JavaScript frontend app and a Java backend, or a Rails backend, or a Python backend,” and people didn’t like doing that as much as they did doing everything in JavaScript.

For people who take seriously the premise of React Native — By the way, React Native, for people who don’t know, is essentially a way of reusing web components that you — Not web components, but they are components, react components that you build on the web. You can reuse them in your iOS app so that you don’t have to rewrite every piece of code, because a lot of mobile apps, their mobile app looks like the desktop app, or it has a similar experience, similar components. This is a pretty common pattern.

People can choose to do react native where they have these components that are in a JavaScript-like thing in — Or I think they are in. They’re in react. They’re in something that looks like react — Gosh! I don’t remember exactly what it is. It interweaves with their iOS code and there are some compilation step where the React Native component gets turned into Swift code or — I think. Maybe it just gets rendered — I should refresh my memory on this. The thrust of my question is; for people who can choose between these two full-stack paradigms, and I think this is probably kind of a goofy question to ask because I think people basically — They either evolve in the Swift ecosystem and full stack Swift makes sense to them, or they evolve in the JavaScript ecosystem and full stack JavaScript makes sense to them. Let’s just say a hypothetical person who could do either one. What kinds of applications would make sense for full stack Swift and which ones would make sense for full stack JavaScript?

[0:38:24.6] CB: The first part there is this whole concept of isomorphic development. We're using code front-end and backend. It's not new and it's one of the reasons why Node.js is so popular working with the browser.

JavaScripts and Node.js is kind of one of the big areas where isomorphic has taken off and it's been of huge interest. Now, that does leave to an argument that you have some things at React Native. You could just use JavaScripts to create great React Native apps on all of your front-ends. If we consider our front-ends currently to the web. iOS and android is the major players.

Now, once that would work, there's some thought leadership done by a group called ThoughtWorks and they kind of first implemented this at same cloud. They started talking about the concept is something called the BFF, the Backend for front-ends.

[0:39:27.9] JM: I did a show on this.

[0:39:29.4] CB: Okay.

[0:39:29.8] JM: Yeah, I did a show on BFF with Lukasz Plotnicki who worked with Soundcloud on that ThoughtWorks project.

[0:39:38.2] CB: Yeah. The key conclusion that they came to is mobile and web frontends have different requirements and they therefore have different requirements of the backend. Your backend for web needs to do different things to your backend for mobile.

Under that paradigm, it makes sense to have JavaScripts and Node.js as the backend of web. If you got to have a separate backend for mobile already, then what would that be depending on what your frontend mobile implementation is.

You got this sense of language pairing, but the other thing that came out of the Backend for Frontend model was around having self-contained agile teams. It's the fact that not only do you have different requirements for the backend for web than you do for mobile. You probably have different timescales on which you want to be able to deliver function.

Let's say there's the new release of iOS that's just come out. There's a new function that you want to be able to exploit. That means you need to deliver a new version of your mobile app and you may well need to provide a new version of the backend for it because it needs to access something new from a service. You want to be able to build that and deliver it as quickly as possible.

That means that if you have a single backend, you're actually tied to the backend delivery schedule, because they have to worry about other frontends and potentially de-stabling those other frontends by changing the backend. The reverse is true for mobile clients. If you're using something like React Native and you're using, basically, the same codebase across all of your mobile frontends and a new version of iOS comes out and you want to exploit that new function. You can't just app the code and deliver it for iOS, because you're potentially affecting your android deliverable as well.

You could release the iOS code without releasing the android version, but you have changed that same codebase. You've added a level risk and a level or retest that needs to be done to make sure you don't affect the android implementation.

Under this Backend for Frontend pattern, you actually want to get to the point where you have a separate android team, a separate iOS team, and a separate web team, and that they own both their frontend and their backend, so that they can deliver on their own schedule for their own customers, their own users' needs and for the feature and function that's relatable to them. Then, that becomes best to do in the same language frontend and backend.

Now, with things like React Native, yes, you can do the same stuff across both platforms and some of the integration with the device is actually pretty good, but you still end up with this — The look and feel for an iOS app is generally different to what it is for an android app. If you're doing things consistently because you're using React Native, then you either start to have to toggles in there to do different things differently on android and iOS or you actually end up with two different apps. At which point, you might as well be doing them in Swift for iOS and in Java for android.

Generally, this concept of wanting to develop features for your users, deliver them on your own

schedule and wanting to be self-contained in agile and not have a knock in impact to teams delivering for android and for web. It does make sense to separate things out into separate projects. Then, it becomes more obvious that doing it in the native language for each of those three makes sense, so you end up with a Swift application on iOS. Where the Swift backend, you end up with a Java implemented android app with a Java backend, or maybe [inaudible 0:43:56.4] for those two. For web, obviously, JavaScripts and Node.js on the backend.

[0:44:01.9] JM: Okay. I could also just see the more simplistic world, let's say, of a startup and you say, "You know what? Our mobile customers are the people we need to focus on the most and we've got limited developer resources. We're going to focus on the Swift iOS app, and because that is our main focus, we're going to put Swift on the server. The web platform is going to be an afterthought and maybe we have one guy, or one girl that works on the web platform, and we have non-isomorphic development there."

Maybe you get Swift as a — Maybe you get some Swift-like thing that compiles to JavaScript later on. I would be surprised if that happened. Then, you could have a more simple isomorphic development world.

Anyway, you've convinced us that Swift on the server makes sense in some contexts, maybe not every context, but at least in some contexts. You've also said that when Apple announces Swift, it was just for client devices. The other thing is that it was closed-source. In 2015, Apple open-sourced it. Why was it important that Apple open-sourced it? Why was this an important point in the timeline towards getting Swift on the server to work?

[0:45:28.8] CB: Yeah. Swift is an open-source project was announced at the start of December. Actually, the day after that was the day I started working on Swift. Part of that was — Whilst part of the open-source announcement said Swift is now available on Linux, and that was certainly true. It was just very, very limited. You had the ability to write and execute Swift code on Linux, but there was no, shall we say, API ecosystem.

Some of the core libraries for Swift just did not exist on Linux. The concurrency library, the thing that lets you run your code or even multiple CPUs which is called grand central dispatch, that didn't even compile on Linux on the day Swift was open-sourced. The main library that people

are used to using called Foundation. It compiled but I think only about a third of the APIs were actually implemented.

You could write Swift code and you could run it, but the APIs that actually let you do meaningful tasks really weren't there. By open-sourcing it, it's actually allowed the community and people who are interested in making Swift successful on non-Apple ecosystem platforms, they could actually help to make it ready.

One of the first things I started working on with grand central dispatch; getting this concurrency library working, because servers are multi-CPU, and if you want to deal with large numbers of concurrent connections, you need multi-CPU exploitation. I think it's fair to say that if Swift wasn't open-sourced, then dispatch would not be available on Linux today and I don't think a lot of this work has been done and Foundation would have happened.

That's not to say that Apple doesn't care about Linux. It's a more a case of there's a limited number of employees which can work on this stuff. By opening up the community, it enables a huge number of people who think Swift on the server on Swift on Linux is going to be valuable to take part and to actually put something in to get something back from this.

I think that's one of the reasons why in the first six months of being open-sourced, there was something like 12 different server-side frameworks that sprung up.

[SPONSOR MESSAGE]

[0:48:26.3] JM: For more than 30 years, DNS has been one of the fundamental protocols of the internet. Yet, despite its accepted importance, it has never quite gotten the due that it deserves. Today's dynamic applications, hybrid clouds and volatile internet, demand that you rethink the strategic value and importance of your DNS choices.

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

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

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

[INTERVIEW CONTINUED]

[0:50:20.3] JM: When you say that Swift was open-sourced, does that mean that the compiler was open-sourced?

[0:50:25.4] CB: Every single part of it has been open-sourced. It's built on LLVM which was always an open-source project. Swift, the language and the complier itself, that's been open-sourced.

[0:50:40.0] JM: When you say that language is open-sourced — I guess I don't understand what that means, because people could — When Apple first announced it, when it was "closed-source", people could still write applications with Swift, right?

[0:50:51.5] CB: What this means is, if I wanted to actually modify the programming language itself, then there's a process to that for open-source. I can download its source code, I can modify it, and I can send that back with a request to get it integrated.

[0:51:07.9] CB: The source code for a language, that's essentially the compiler, right?

[0:51:12.8] CB: Yew. For Swift, you are basically talking the compiler, which parses the Swift code that we've designed down to the intermediate language and then compiled that down to machine assembly.

[0:51:29.5] JM: Right. I just wanted to clarify that, because I was like, "Okay. When Swift announced it, it was closed-source, but people could still write code with it, so the documentation was there." It's just that you didn't know what was going on between the Swift code you wrote and the machine code that executed on your box.

[0:51:50.2] CB: That's correct. As part of the open-sourcing, they also open-sourced a lot of the core libraries. Foundation became open-source dispatch, or ground central dispatch, the concurrency library. That was open-sourced. Swift package manager; their way of doing build and dependency management, that's open-sourced. The test framework is open-sourced. Really, all of the core capabilities are all there for people to actively contribute to.

[0:52:24.9] JM: As you mentioned, you've been working on Kitura, which is Swift web framework that IBM is building and there's a bunch of other Swift web frameworks, or web framework allows people to build web applications to run Swift on the server. As you're talking about building this concurrency library, it sound so interesting. It's so exciting, because I've done a bunch of shows recently about scheduling, and scheduling in terms of; okay, you've got 15 map-reduced jobs and you've got a bunch of different machines sitting on your cloud provider that are assigned to you and you can allocate those jobs to those different servers however you want to. That's a hard problem. It's not a solved problem. It's — Everybody writes custom schedulers in order to do it.

You're talking about writing a scheduler for threads at a low level, and I imagine that the problems and the subjective challenges you have are just as interesting, just as debatable as these things that occur at the higher level. Is that accurate?

[0:53:42.8] CB: Yes. Implementing good concurrency libraries is not an easy task. For Swift in dispatch — We've got the advantage that it was implemented by the team and Apple have been working on this dispatch for a number of years. The problem is that it didn't work on Linux. Our first problem is really more porting something which already existed to a different platform.

The other advantage is that a couple of my colleagues; [inaudible 0:54:15.2] and Dave Grove, they have a very long history of working inside the Linux Kernel on Runtimes, and they've been working on this for the last year or so to really, not just make it work on Linux, but to do some of

that really low-level scheduling kind of stuff to make sure that it will take work and dispatch it to all available CPUs and actually scaled to almost 100% CPU usage on the high load. That's the goal of concurrency library, is to make sure that any work that you want to do can be distributed over all of the available CPU results and, therefore, executes as quickly as possible.

[0:55:00.3] JM: You have spent 15 years of your career building Java applications as we mentioned. That is most of the time that Java has been around. From what I hear you say and in the presentation I watched of you, you seem to believe that Swift has a real change at becoming the enterprise development language that supplants Java, or at least is another one that can rival Java in terms of its size, in terms of its penetration, in terms of its usefulness and performance. What's your prediction for how that's going to proceed, how that roll out is going to proceed? Yeah — I don't know. Do you have any interesting predictions?

[0:55:50.5] CB: Java has been around for something like 23 years now, and I don't see it's going away. In the same way that some of the older language which people considered to be, now, redundant COBOL, a huge amount of infrastructure like ATMs, the backing infrastructure for that is still running on COBOL, and COBOL is still going strong. I don't think Java is ever going to go away.

There's a whole load of workloads which you do in Java today that you'll potentially be able to do on Swift in the future if you want to. I think the fact that it's a typed language like Java, means that it's got the potential performance. The fact that it's typed also means that you're less likely to have problems and bugs that just appear at runtime where you have calculation problems, and that means it's got potential for doing transactions and that sort of thing. It's got potential to cover a lot of the use cases that Java has today.

I think we'll see Swift growing overtime, and I think you'll see it being applied to many, many more use cases in the same way that Java can be used in almost any scenario, but I don't think Java is going to go away. I think it's going to be there for a very long time, and Java will continue to evolve.

As you said, Java 8 brought in some new stuff. It brought in Lambda, so the ability to pass around functions and it had more functional programming constructs added. I think Java is

going to continue to evolve and it will continue to be a huge program language. I believe there's a lot of space for Swift to grow, particularly for cloud deployment.

I think another interesting place is going to be around Internet of Things, because it's small and it's already being designed to run on an embedded device. There are many, many more embedded devices that it's a good language for.

[0:58:15.0] JM: Fascinating. As far Java going away, I think about all of the Hadoop technologies and how closely tied with the JVM ecosystem they are right now. There are some work on interoperability with other languages going on, but just taking the Hadoop ecosystem. If you looked at, "Okay. If I spin up an enterprise today, okay, maybe I do a lot of the business logic in Swift, like the transaction logic, or just server-side stuff, but if I get enough data to build some big data stuff, then I'm going to need to work with Java."

Yeah, I could even imagine a polyglot world where you want Swift as part of your server-side business logic and you want JVM stuff as the other server-side logic. Anyway, interesting.

Well, I think we've covered a lot of ground. Chris, it's been really interesting talking to you.

[0:59:18.0] CB: Yeah, no worries. Thanks for inviting me and giving me the opportunity to chat.

[0:59:24.8] JM: Yeah, for sure. Maybe if there are developments in the future that you want to shed more light on, I'd be happy to have you back on.

[0:59:33.1] CB: Yeah, I'd love to. I think there's going to be many more things in the figure to talk about as Swift evolves and starts getting used in many, many more different ways.

[0:59:43.4] JM: Wonderful.

[END OF INTERVIEW]

[0:59:45.3] JM: Thanks to Symphono for sponsoring Software Engineering Daily. Symphono is a custom engineering shop where senior engineers tackle big tech challenges while learning

from each other. Check it out at symphono.com/sedaily. That's symphono.com/sedaily. Thanks again Symphono.

Thanks again Symphono for being a sponsor of Software Engineering Daily for almost a year now. Your continued support allows us to deliver this content to the listeners on a regular basis.

[END]