Progress

What Serverless Means for Enterprise Apps

Michael Salinger

WEBINAR TRANSCRIPT

Laura Lewis:

Welcome everyone. I'm Laura Lewis and I'm on the marketing team here at Progress, supporting our Kinvey product. It's my pleasure to introduce today's webinar, "What Serverless Means for Enterprise Apps." Today we have a highly regarded speaker with us. Michael Salinger is the Senior Director of Engineering for Kinvey at Progress, where he leads the team responsible for the development of the Kinvey serverless cloud. Michael has extensive experience in cloud, serverless technology, web, mobile, and back-end systems, having architected and built multiple SAS and cloud products over the course of his 17-year career. At Kinvey, Michael designed and architected the FlexService Runtime, a Node.js framework, for the development, deployment, and execution of serverless functions and micro services for enterprise applications. Please join me in welcoming Michael.

Michael Salinger:

Thanks Laura, and good morning everyone. Serverless has become quite the buzz-worthy term over the past couple years with the introduction of AWS Lambda and other similar services. Today we're going to try to unpack what serverless means, separate out some of the myths from the reality, and also look at what serverless means for enterprise apps and how it can be impactful on building enterprise applications. So to start off today, we're going to take a couple of polls to get a feel and an assessment for what you all think about serverless based on what you've heard and researched and read. And the first poll question is, what does serverless mean? What do you think it means based on what you know? Is it implementing applications without servers? Implementing applications without any operations or "no-ops"? Implementing applications without thinking about resources or implementing server logic with individual functions -- such as functions as a service? Or are you not sure? So the results are interesting. A lot of people unsure, we've got some -- looks like implementing server logic with individual functions is high up there, and implementing without resources. Good.

Take just one more quick poll question before we get started. How does a serverless architecture impact enterprise applications? They don't at all, they're a means of keeping costs down, they help accelerate development, they help developers focus on the important parts of the app, or I'm not sure. All right, most of you thought that it helps developers focus on the important parts of the app, then accelerate development -- and means of keeping cost down also was high up there. All right, so now we're going to start just by looking at what we're going to cover today.

So first we're going to take a look at what is serverless, and separate some of the myths and misconceptions out there from reality of what serverless is. We're going to follow that up with looking at the building blocks of serverless and what are the different components that make a serverless architecture. Then we're going to turn to what does serverless mean for enterprise applications? How can it impact me as a developer building an application for my enterprise? And then we're going to take a look at how Progress Kinvey enables serverless and helps build enterprise applications. And finally we're going to take a look at some general design considerations for building a serverless app. So we'll start off with what is serverless?

Some common misconceptions -- and the first one is the one that received I think the highest poll score -- is not just functions as a service. It's not just FAS. Now, functions as a service is a critical component to a serverless architecture, and it is a common usage of serverless, but it is not the only aspect of what a serverless architecture entails. It doesn't mean no servers. A lot of times when I'm introducing serverless architecture to two different people, one of the common questions I get is well, there has to be a server somewhere. You can't have something without servers. Well it doesn't mean no servers, and we'll see a little bit later what it means in relation to servers. It's not one specific technology or specific technologies. It's not functions as a service, as I said before. It's not Lambda, it's not Kubernetes or Docker -- although all of those pieces do contribute to a serverless architecture, and they're examples of technologies you can use to implement a serverless architecture. In and of themselves, they are enablers of serverless -- they're not what serverless is itself. And it's not no-ops. Operations is always a key component of any application. While the role of operations will change in a serverless architecture, it doesn't alleviate the need of operations altogether.

So then what is serverless? We kind of looked at what it isn't, now we're going to take a look at what it is. So serverless refers to any cloud native service for implementing application logic that allows the developer to focus on the app and not think about servers. Let's unpack this a little. Cloud-Native. Serverless is not client side -- although there can be client-side components that interact with serverless as part of the architecture. In and of itself, serverless is definitely not something that you put on a client. It's not installed software. If you're installing it on your own network, on your own servers, and the claim is that it's serverless -- it's missing the point a little bit of Cloud-Native and serverless. And one of the reasons for that is -- the next couple bullet points -- it's provisioned on-demand, it's fully redundant, load balance, and auto scales. So all of those things -- the cloud enables serverless to be possible. So the fact that you have resources that are managed in the cloud enables what we'll talk about in the next couple slides as serverless. So it's used for developing application logic. And some examples of that type of application logic are integrations with various back-end systems, various functions to respond to events, messaging, different services -- such as data store, CDN, file store -- really anything that would require high-level performance server processing -- that type of application logic is what serverless enables.

And finally, and this is probably the most important piece, it allows you to focus on the app and not the servers. So no provisioning, no managing, no scaling concerns, no worrying about how your app will communicate with the servers. The back-end of your application is an extension of your application. It's not something that is separate that you have to worry about. You're focusing on your app, you're focusing on what your app needs to do. Not the servers, not the resources. So again, serverless refers to any Cloud-Native service for implementing application logic that allows the developer to focus on the app and not think about servers. Or to put it another way, serverless allows application developers to focus on the application logic rather than the kinds of resources that are needed to run the application.

So we're going to talk now about some of the different building blocks of serverless. What are the components that make up a typical serverless architecture? Well the first one is one that most people have heard about, and most people think about when they think about serverless. And that's cloud functions -- also known as function as a service. The second is micro services. Third, cloud services. And then the fourth is events. So cloud functions -- also known as function as a service -- are single, stateless, atomic functions that scale. So what does that mean? You write a single function to implement a piece of business logic. And that function is invoked by the client application or some event. Each function scales independently based on its needs. So let's take an example of where you might have two pieces of your application, and one of them is a data integration function that makes a request out to a REST API, and returns the data. And the second is a password hashing algorithm for your user authentication. Now each one of those has very different performance needs. Password hashing is a very expensive operation, and requires a lot of CPU, and so its scaling needs are going to be a lot different from a data operation where you're making a request out to an external system, and basically waiting for the result to come back. And so if there's a spike in one use or another, the benefit of cloud functions is that these are separately and independently scaled. So each one of these is scaled up or scaled down based on the current needs of the application. So as a certain aspect of your application is hit harder, that particular piece will scale up to its own individual resource needs.

In a typical application, you may have tens or hundreds of cloud functions. Really anything that you would need typically to off-load to a more highly performant back-end server, you would consider using a cloud function for.

So the next component is micro services. Now micro services have been around for a lot longer than serverless architecture has. So while micro services themselves are not unique to serverless architecture, they are a critical building block of serverless architecture. And serverless micro services have very specific characteristics. But looking at just in general what micro services offer, the emphasis is on micro. They need to be very small, very lightweight, single-purpose services. In fact, in a serverless context, because of what we discussed in regards to cloud functions, you want to keep your micro services as small as possible. Because you want to take advantage of the individual scaling characteristics that serverless offers, you would want to keep the micro services focused on a single task. The reason you would use a micro service as opposed to just a cloud function is if you had some code or some functionality that is shared and could use the benefit of having a slightly larger unit of code than just an individual function. A typical application may have several different micro services, and one of the things that makes micro services serverless as opposed to

just generic micro services is that they're written in a way that the server infrastructure transports scaling is hidden from the developer. So the developer never has to take into account how am I going to access that micro service from my app. The developer doesn't have to worry about how that's going to scale, what the characteristics of it are. The developer, again, just writes the logic and focuses on the code.

So cloud services are another component of a serverless architecture. Cloud services are no-code services that implement some back-end application need. Some types of cloud services would be cloud data store, user management, integration with data sources, SSL providers, messaging, notifications, email, push, and so on. Really any service in the cloud that helps developer implement some logic in their application without thinking about servers. And finally we get to the glue of serverless architecture, and that's events.

So the logic in serverless responds to events. And those events can occur inside application, they can be external events, invoked events. And some examples -- you know, you could have a serverless function that executes before creating a new entity in some data store. You can invoke in response to an HTTP end point. A serverless function can be invoked in response to some client-side event, such as geofencing. An example that I've used before is, let's say you have a retail store application that is on a mobile app that wants to let the store manager know when a customer with the mobile app enters the store. So a geofencing event could trigger a cloud function that sends that store manager a Slack notification for someone who has the app and is part of the preferred program for the store.

Events are still in the early stages. They are common to pretty much all serverless providers, and there is a lot of effort right now to really flesh out what they can do and how powerful they can be, and I think there's going to be a lot more in the next three to five years that we're going to see from the events space. There is actually a consortium right now that is working to create a standardization around event message passing. So there's really a lot of movement in this area, and it's worth watching and keeping an eye out for.

So now, what does serverless mean for enterprise applications? So we know what serverless is, we know what its basic building blocks are. But the important part of this is how does it help me as an enterprise developer build applications? So it's all about delivering value. As a developer, not having to think about the resources, not having to think about how my application will scale, how I'm going to communicate between my application and my back-end systems -- it allows me to focus on my value -- the value that my app provides that's different than what anyone else is providing. It allows me to spend the limited development hours that I have on that value, and less time on common technical hurdles that most applications have to deal with anyway. It allows me to focus on the experience -- build the UX first, think about the client experience and how the client will interact with the app, offload those complex app functions to cloud services, cloud functions, and microservices. Take away any of the heavy processing from the client apps so the UX can be fluid and delightful for the user to use. It allows me to not worry at all about resourcing, high availability, scaling needs. Don't have to think about how much memory my application's going to use, how much CPU I need, what the disk space requirements are. I focus on my application and let the serverless infrastructure take care of the rest.

Just one word of caution. It's not an excuse to write bad code. I've had conversations before with some people where they would write something like a triple four loop that you see here that is going to be grossly inefficient, and expect the serverless architecture to kind of correct their mistakes. And a serverless architecture will do well at scaling and managing resources for well-performing applications. But if you have something that is really a bottleneck -- like something that you see here -- no amount of scaling is going to take care of that, because the performance needs of something like this will outstrip the ability of any system to scale in a reasonable time. So while you don't have to take into account memory, and while you don't have to take into account CPU, and worry about how it's going to scale, you still need to write good, efficient code.

So besides value, it also enables high productivity for developers, because there's no provisioning, no waiting on IT or DevOps to stand up on a dev server for me to be able to start testing out my app, and countless hours spent getting the production system and production servers ready. Because you're creating small atomic units of code in configuration-based services, you're able to engage in frequent low-risk iterations and make small changes to your code, which reduces the risk of errors popping up and bugs getting into the system. And so it really is conducive to agile and lean methodologies.

So I want to spend a little time just talking about Progress Kinvey, which was built from the start as a serverless platform. Again, all of the serverless components that I've talked about are representative in Progress Kinvey, and I'm just going to spend a little time illustrating some of those features and how they relate to the serverless architecture.

So first we'll start with cloud functions. Kinvey's cloud functions were launched in 2012. It was essentially function as a service before function as a service was a thing. And as we described earlier in cloud functions, there are small atomic stateless functions that respond to events such as onPreSave of some entity into a data store, onPostFetch, et cetera. They can be invoked via an SDK or HTTP, and they scale as needed and they're load balanced and redundant.

There's an example here of just a very simple cloud function on the Kinvey platform. In this case it is a function that will respond to a onPreSave event for an entity on a specific data collection. And here I'm just doing a very simple validation to make sure that the owner property of the entity contains the progress.com domain name. And if it doesn't, I automatically append it before continuing to save it to the data store. And so that is just one very small example that illustrates some of the power of just doing these simple validations in a very easy, low code manner. Because you're hooking into the events, you're taking advantage of the Kinvey data request pipeline, and you're just inserting yourself right into the middle of that.

Another example similar to what I spoke earlier about the person entering into the store -- this function can respond to that geofencing event, and send a Slack notification to the store manager when someone enters the store.

So now we're going to talk about flex services. And flex services are Kinvey's implementation of stateless, serverless, micro services. They can be used for data integration needs, custom authentication, and business logic. And using an SDK that Kinvey provides, called the Flex SDK, all of the server components

are hidden -- HTTP, ports, routing, middleware, et cetera. The developer just implements specific functions in the micro service in response to different events. And it also integrates with the common CI/CD systems -- so you can introduce continuous integration and continuous deployment into your microservices. And, like everything else, they're automatically load balanced, redundant and scalable.

So in a flex service, you register different functions in your flex service to respond to different events. Now flex services are all Node.js projects, and they allow you to -- because they're a Node.js project, you can make use of all of NPM and the ecosystem around NPM -- and wire up, you know, different functions and have different common code and common files shared among these functions. But each function is wired up to a different event. So here for example I'm registering this transformations.order function to the transform order event and the support case function to the transform case event. So again, here's the function that implements the support case event, and here I'm doing something similar that I did in the cloud function where I'm getting a different ID from the Kinvey user store and attaching it to the body before I save it. And in this case I'm saving it to a remote ticket system in order to integrate with that ticket system and match the IDs up appropriately.

And finally, Kinvey offers a host of cloud services that are configurable. Some of these are listed here -- data store, user lifecycle management with role-based access control, no code configuration based data integration services for common data providers, cloud caching, notifications, locations, and many more. This is just a sampling. And Kinvey allows you to configure these through a web-based application, and then easily wire them up into your own client app.

And Kinvey provides numerous ways to invoke your cloud functions via collection hooks, end points, and the SDKs. Now collection hooks are data events, so we saw before onPreSave was an example of a collection hook. So there are hooks right around different data operations, such as fetch and delete and save. End points are just cloud functions or micro service functions that you can invoke on demand or in response to some client-side event. And then I'm going to talk a bit about the SDKs. And SDKs are not traditionally thought of as part of a serverless architecture, but I think they add a lot to the spirit of a serverless architecture, because they treat the serverless cloud as an extension of the app. So when you're invoking your serverless code or invoking an event on your application -- whether that be a iOS native app or a Progressive web app or a chat bot -- using an SDK allows you to invoke that serverless code as if it was part of the app and the app language. You don't have to worry about HTTP or the transporter dealing with different status codes, or what the proper REST implementation is. And so the SDKs enable app-first development native to your language or framework of choice. It enables high productivity by removing the transport and protocol concerns, and ties everything together.

So here, just a couple quick examples of using the SDK. This is using our NativeScript SDK, where in the first example, I'm doing a simple fetch from a remote data source. Now that remote data source could be Kinvey's data store, it could be an integration with sales force, it could be an integration with some REST API. The point to note here is that this way of retrieving data is the same. The interface is the same regardless of what data source I am using. So the power of the SDK in the Kinvey serverless platform is that, again, it allows me to focus on my app's value without worrying about how do I query Salesforce, how do I query a REST API? I don't have to do it differently for each type of data source. I have one way of retrieving data, and that way is

the same no matter what my data source is. And then the second example just shows an example of how I could just easily invoke a serverless function from my app.

So finally we're just going to talk about some key design considerations for serverless apps. And things to keep in mind as you start building your first serverless app, or as you're expanding your knowledge and trying to refine. So given that what we stated before as the benefits of serverless -- one of the first best practice here is to focus on the UX and the value first of your app. Design your user experience, build the logic that gives your app value, and then as needed extend the app serverless components. So start from the point of view of the app, and prototype it, build your UI. Utilize -- even if you have a lot of integration points, utilize something like Kinvey's data store to prototype your data. As you're building it out, and as you see certain functions -- while this could potentially benefit from the scalability of the cloud -- extend the app with those serverless components.

So second key point here is to federate remote sources. And as I showed with the SDK example, you can use serverless functions data integration services, or micro services, for external data access or remote API access. And then the serverless cloud becomes your single interface for all your external systems. So as I showed in the SDK example, one interface -- regardless of what the external system is. And it simplifies the communication layer of the client app. I only have one SDK for accessing external data or remote sources. I only have one interface. And I don't have to implement a REST communication layer, a Microsoft SQL communication layer, a salesforce.com communication layer. I have one and the serverless cloud takes care of the actual communication with the remote systems.

Think small, think modular. So I gave the example earlier of the authentication function versus the data access function, and how they would have different scaling needs, and how the serverless cloud could optimize and make those scaling needs unique to each individual function. Overall in your app you should do the same. Each unit function micro service cloud service should do one job. And that helps to enable those scaling benefits I mentioned, but it also makes your code easier to maintain and makes any changes that you have to make to your code lower risk.

Avoid monoliths, avoid large complex micro services that try to do many things -- unless they no longer become micro services. Group common tasks together when necessary but, again, think small. Think stateless. Now serverless by its nature is stateless, because the functions are being scaled independently, because the micro services are being scaled independently. They're ephemeral. They are scaled up and scaled down just based on the needs of application. So in and of themselves, they can't contain any kind of state. There's no guarantees that a single instance will be running at any given time. Any service-side state that you need should be managed using some kind of cloud data store or some key value store. But the functions and the micro services themselves should be designed in a stateless manner.

So in conclusion, think about your app, not your infrastructure. Think about what value it has, think about what you want your user experience to be. Enable high productivity by removing infrastructure transport protocol concerns, scaling concerns, high availability, load balancing from your vocabulary. Just worry about the value and the UX.

And then have a serverless mindset. Back-end services are an extension of your app. They're not something separate that you call. Just think of them as part of your app that you invoke when needed. Make everything small and modular. Make everything stateless. And finally, build what really matters. Don't worry about the rest.

Laura Lewis:

All right. Thank you Michael. We're going to open up this discussion now and answer any questions that have come in. Don't forget to type anything you may be wondering into the chat window. So to kick it off, what's the difference between Kinvey Flex Services and AWS Lambda?

Michael Salinger:

So they're both serverless implementations of function as a service. Flex services also allow the ability to do micro services in a serverless manner as well. But I think the biggest difference is that Lambda is a general purpose function as a service system. So it's there for you to write serverless functions, have them scaled, and then do what you want with them separately. Whereas Kinvey Flex Services are really part of an app development platform. So they are a component of a whole toolset that's used for building your app that is all integrated together. And so I think the mindset is a little bit different in how both Kinvey and AWS approach the problem.

Laura Lewis:

Okay, great. What about is serverless only for cloud apps? Can it be implemented for on-premise micro service apps?

Michael Salinger:

Kinvey -- I mean, sorry, serverless -- is not only for cloud apps. On-premise apps can certainly make use of a serverless infrastructure, but the serverless component of it should be in the cloud. But that doesn't preclude you from writing local mobile client apps, desktop apps even, or all different kinds of experiences that make use of that cloud infrastructure.

Laura Lewis:

On a similar topic there, how do you optimize performance of your serverless functions?

Michael Salinger:

Really you just use general software development performance practices. You don't have to do anything special to optimize the performance of a serverless function, other than, like I said, keep it small and try to keep the execution time of each function down to a minimum. But using just general purpose performance considerations that you would use in any general purpose programming language, I think, is all you really need to do. Because that's what the serverless clouds are build to handle.

Laura Lewis:

What cloud service provider do you use or do you have your own?

Michael Salinger:

Well Progress Kinvey uses multiple cloud service providers. So we use AWS Lambda, we use Azure, we use Google Cloud. And I've used all three personally as well.

Laura Lewis:

All right. So I have another question in here. This one has a little bit of detail. So my experience so far with Azure serverless has not been great. Containers as a service are much more expensive built by the [second?] than your own Docker VM. And in particular, certain essential components like file storage are HDD only, not SSD. Does this just reflect that Azure is not ready for prime time?

Michael Salinger:

Yeah. So Azure specifically -- their container as a service offering was really just launched within the last month, I think. And I think that they were doing it kind of probably earlier than they probably should have, due to competitive pressure.

Laura Lewis:

All right. What's the difference between Pivotal Cloud Foundry and Progress Kinvey?

Michael Salinger:

So Pivotal Cloud Foundry is -- although they're moving in a serverless direction -- is really more of a platform as a service than a serverless architecture. And because with Pivotal, you still have to provision resources. So because Pivotal is more of an on-prem type solution, you still have to determine, you know, how many nodes you're going to have. When you're deploying a service, you have to determine what its resource needs are going to be. It's much more abstracted than doing something as basic as an EC2 instance on Amazon. But it's still there. With Progress Kinvey and other more pure serverless providers, none of those constraints come in. You don't think about, you know, any of those resource needs at all. Again, you just focus on developing your application. So Pivotal gets you part-way there by taking away a lot of the thought about it, but it doesn't get you all the way there in not thinking about it at all.

Laura Lewis:

All right. Great. Why would I use a cloud function instead of my app's own web API?

Michael Salinger:

Again, so you would use a cloud function rather than a web API because as you're developing your application it would increase your productivity. Rather than having to build the entire web API and stand up Express.js or something, and then worry about the routing and all the middleware, implementing the cloud function just allows you, again, to focus on the value that your app provides, rather than all of the infrastructure that your web API needs.

Laura Lewis:

What does Kinvey do to handle latency between different function invocation?

Michael Salinger:

Yeah. There's -- I'm trying to think of the right way to answer this question. So the function invocations are -- yeah, I think I'd need to have more of a conversation on what that's talking about, because I'm not quite sure what kind of latency he's talking about or seeing or he's seen in other providers.

Laura Lewis:

Okay. No problem. If you want to put more detail in the chat, we can do that. Or we can absolutely follow up with some people afterwards too. We're not going to be able to get to all of the questions that we see here

today. So just to, I think, close it out with one more here. Can we take advantage of serverless architectures through on-premise deployments?

Michael Salinger:

So you can, but not as full-featured as you would from a pure serverless. Because one of the chief benefits of serverless in application development is not thinking about the resources. And in an on-premise deployment, the developer has to think less about the resources, but there comes a point where in on-prem, since you're installing whatever platform you have on [bare metal?], at some point you're going to hit an allocation where you need more bare metal. And with the cloud, you know, cloud providers do this as a business. So they're making sure that everything is provisioned and ready before you need it. And so I have seen examples and talked to people where they've done on-premise deployments of private clouds and they run into a lot of the same problems that they ran into pre-cloud where it works for a while, but then they get to a point where they're resource-strapped and IT has to provision more hardware in order to scale out the private cloud. So while private cloud is certainly a viable middle ground or stepping stone, it doesn't give you the full productivity benefits or the full scalability benefits that you get from a public cloud provider.

Laura Lewis:

All right. Thank you so much Michael. So thank you for submitting the questions. They were all great. And I'd also like to thank Michael for joining us today to share his knowledge. We hope you enjoyed the event, and look forward to having you on future Progress webinars. Be sure to check out our previous presentations on our website, and follow us on Twitter, LinkedIn, and Facebook. Both notices will go out before the end of the week. Thank you again, and have a great rest of your day.