EPISODE 740

[INTRODUCTION]

[00:00:00] JM: When a user makes a request to a product like The New York Times, that request hits an API gateway. An API gateway is the entry point for an external request. An API gateway serves several purposes; authentication, security, routing, load balancing and logging. API gateways have grown in popularity as applications have become more distributed and companies offer a wider variety of services.

If an API is public and anyone can access it, you might need to apply rate limiting so that users cannot spam the API. If the API is private, the user needs to be authenticated before the request is fulfilled. One example where an API gateway might be useful is that New York Times examples. Maybe a request comes in from somebody who is a subscriber to The New York Times, so you don't want to show them the paywalled version of the site. You want to authenticate them. But if they are not a subscriber, then you want to show them perhaps a version of the site that tries to get them to sign up or restricts access to certain content.

Kong is a company that builds infrastructure for API management. The Kong API gateway is a widely used open source project, and Kong is a company built around supporting and building on top of the API gateway.

Marco Palladino is the cofounder and CTO of Kong. He joins the show to tell the story of starting Kong 8 years ago and how the API gateway product evolved out of an API marketplace. Marco also discusses the architecture of Kong and his vision for how the product will develop in the future including the Kong service mesh. This was a great episode about APIs, cloud infrastructure and, of course, API gaetway and service mesh, which has been a popular subject in recent episodes of Software Engineering Daily.

Before we get to this episode, I want to mention we're looking for sponsors for Q1. If you are interested in advertising on Software Engineering Daily, you can go to softwareengineeringdaily.com/sponsor and find information about sponsorship and you can also send me an email, jeff@softwareengineeringdaily.com. We'd love to hear from you. We're also looking to get some feedback from our listeners. If you go to softwareengineeringdaily.com/

survey, you can give us feedback on the podcast. We'd love to know what we're doing wrong and what we're doing right and you can help us with that information.

Let's get on with the show.

[SPONSOR MESSAGE]

[00:03:00] JM: I have been going to the O'Reilly Software Conferences for the last four years ever since I started Software Engineering Daily. O'Reilly conferences are always a great way to learn about new technologies, to network with people, to eat some great food, and the 2019 O'Reilly Software Architecture Conference is for anyone interested in designing and engineering software more intelligently. There are actually three software architecture conferences. One is in New York, February 3rd through 6th; in San Jose, June 10th through 13th; and in Berlin, November 4th through 7th. So if you're in New York, or San Jose, or Berlin, or you're interested in travelling to one of those places, take a look at software architecture. You can get a 20% discount on your ticket by going to softwarearchitecturecon.com/sedaily and entering discount code SE20.

Software architecture is a great place to learn about microservices, domain-driven design, software frameworks and management. There are lots of great networking opportunities to get better at your current job or to meet people or to find a new job altogether. I have met great people at every O'Reilly software conference I have gone to, because people who love software flock to the O'Reilly conferences. It's just a great way to have conversations about software.

You can go to softwarearchitecturecon.com/sedaily and find out more about the software architecture conference. That's a long URL, so you can also go to the website for this podcast and find that URL. The software architecture conference is highly educational and your company that you work for will probably pay for it, but if they don't, you can get 20% off by going to softwarearchitecturecon.com/sedaily and use promo code SE20.

Thanks to O'Reilly for supporting us since the beginning of Software Engineering Daily with passes to your conferences, with media exposure to different guests that we had early on.

Thanks for producing so much great material about software engineering and for being a great partner with us as we have grown. Thanks to O'Reilly.

[INTERVIEW]

[00:05:43] JM: Marco Palladino, you are the cofounder and CTO of Kong. Welcome to Software Engineering Daily.

[00:05:48] MP: Thank you, Jeff. Glad to be here.

[00:05:50] JM: Kong is an API platform, and to explain what it means and what your business is, we should start with the concept of an API gateway and then we'll get into some elements of the story behind Kong and how you got to where you are today. But let's just start with the API gateway. What is an API gateway?

[00:06:12] MP: An API gateway is a middleware that usually sits in between the API the developers are building and clients that are consuming those APIs. The idea of an API gateway or API management, when it was born in 2007, 2008 when mobile applications became a thing. So we had all of these APIs in the monolith in our backend that these mobile applications will have to consume. In order to enable that consumption, we usually put a middleware. It's effectively a reverse proxy that that security authentication load balancing on top of our APIs to enable those clients to consume the API.

[00:06:51] JM: What are some of the problems that an API gateway solves in a modern software architecture?

[00:06:57] MP: So the concept of API management was born when mobile was popular back in 2008 when mobile first came out in 2007, 2008, 2009, but then API management has been evolving from there. So the traditional use case of having an external either community developer base or mobile clients, it's still there, that use case is still there, but that's not just the only picture we're having now.

As organizations are moving and transitioning from monolithic applications to microservices, now we find out we have lots of those APIs internally as well. So the role of an API gateway, the role of API management has been evolving to take care of these additional internal use case, and as we transition to microservices of course, that use case becomes more and more intense to a point where the concept of API management will evolve so much that it's not the same anymore.

[00:07:52] JM: I want to roll back to 2010 when Kong was started, and then we'll come back a little bit later to the state of things today. But the company has a really interesting story and it's gone through a series of product evolutions that I think is really interesting. So Kong was started eight years ago in 2010. How was software architecture changing around that time?

[00:08:19] MP: Well, 2010 was the beginning of APIs the way we traditionally know it. So before 2009, 2010, the industry was working towards the concept of SRA, service rented architectures. Then from there with mobile, now we have to make some of those services available to the external world. So when API management comes out in 2010, those are the years where, for example, Jeff, we're seeing the introduction and the popularization of restful APIs, JSON restful APIs that are sitting on top of our systems and now developers can easily consume them.

Now when we talk about Kong, Kong itself as a technology, was released in 2015, but in 2010 we built the API marketplace, my previous company, Mashape, and Mashape was an API marketplace where developers could consume and publish APIs. Then from Mashape, our underlying technology of the marketplace, then that became Kong in 2015. So it's a very similar story as, for example, .cloud and Docker.

[00:09:26] JM: Right, exactly. I want to go through some of that product evolution, because I think it's probably instructive for anybody who's building a software company. In 2010, was that a little bit early to be building an aggregator of APIs? Why did you end up with an API marketplace and then also an API gateway product?

[00:09:48] MP: Yes. I'm a developer first and foremost, and in 2010, you're right, APIs were not as mainstream, but we had a few of them around. More and more tooling was emerging to help developers create more and more of those APIs. Me, as a developer first and foremost, I was looking for a place where I could find APIs. Back then, the only place for that used to be Google. You would Google a product or a service or something you wanted to do and then add the API word at the end of your search, and that perhaps would show up some APIs. It was a very inefficient way of looking for services.

So when we created Mashape, we wanted to have a place where developers could publish the APIs and then others developers could very easily search for them, consume them, and if the API was a paid API, then a place to manage all of those subscriptions across not one API, but hundreds of APIs that the developer would be consuming.

[00:10:43] JM: And when did you get to a point where you were kind of thinking, “Okay, we need to do something difference.” Because you got very close to running out of money at a certain point, and this was like in 2009. What was it like trying to raise money in 2009 right after the market had tanked?

[00:11:05] MP: Yeah, it was very hard. Plus on top of that, you have to consider that I come from Italy. So I’m an immigrant, basically, me and my cofounder. We moved to Silicon Valley to raise money. So on top of all the challenges of building a company, creating a great product, on top of that we also had a challenge of actually being able to legally stay in the country. So that kind of slow us down as we were fundraising. We had to deal with all deal of bureaucracy and visas and green cards. That was fundamentally a distraction to the actual creation of our company and our product and the fundraising as well.

We were able to find a few angel investors who believed in us in 2010, and that effectively allowed us to keep going from there. So we raised a seed round in 2011. Then we raised a series A in 2014. Then we raised a series B and so on. But the first 100k – Let me tell you something. That was the hardest time of my life.

[00:12:06] JM: Okay. Tell me more about that. What do you remember from those difficult periods in those early days when you’re trying to raise that first amount of money?

[00:12:13] MP: We came here with a prototype in 2010, me and my cofounders, and we really didn’t have money to stay here. So back when we came here with the idea of staying here for a few months, try to raise some money, but we only had money to survive for one week. I was in my 20s. So we landed in San Francisco. Money runs out after one week. We don't know what to do.

The thing is when you are an immigrant in any country, we believe that you really have that one thing, that extra gear that other people rarely have, and that extra gear, it's called desperation. So we were really trying to email people to host us, trying to find anybody who could help us. Back then there was this person who was doing philanthropy. He was helping other entrepreneurs. He answered to us and he was like, "Oh yeah, guys. I can host you."

So this person was Travis Kalanick, but that was before Uber. That was immediately after Red Swoosh. He sold that for few millions and then he was doing philanthropy and he was looking at what to do next. So that was before he became the CEO of Uber. So he was like, "Hey, guys. Yeah, sure, I can host you." We're so thankful to him, because actually without him, we probably would have closed our angel round. So our angel round was signed and negotiated on his kitchen counter in his place here in San Francisco.

[00:13:41] JM: So you go the angel funding. You eventually got a series A and you were building the API marketplace business. At what point did you decide you wanted to build an API gateway product also?

[00:13:54] MP: The API gateway was the underlying technology in the API marketplace. So in order to access the technology, developers would have to use the marketplace in the first place. The marketplace actually grew quite a bit. We ended up being the largest API marketplace in the world. We had 300,000, 350,000 developers consuming those APIs. The problem was we were the middleman of a long tail of developers that were selling APIs and consuming APIs and it didn't scale from a business standpoint. A developer would create an API and then a week later the API would go down. But in the meanwhile he would have attracted some customers, and those customers would ask for chargebacks. So we were dealing in the middle of all of these little transactions with fundamental and reliable APIs.

So Kong was the underlying technology powering all of these requests. We were handling billions of requests per month. In 2015 we decided that the marketplace from a business standpoint, it didn't really make lots of sense, but that technology was very powerful. So in 2015 we decided to extract our technology, and that became Kong overtime.

[00:15:08] JM: Oh, you're saying there are a bunch of developers early on who are building API business, but I guess it was just too early and some of these API business that people were building were not well-architected, or they didn't make enough sense. The economics didn't make sense. So they would end up shutting them down after a while, or the cloud infrastructure wasn't as hardened at that point. So maybe that was flaky infrastructure. Then you're the middleman. So if you're reselling their APIs and their APIs are unreliable, you're responsible for this chargeback stuff. That's a problematic situation to be in.

[00:15:47] MP: That is correct.

[00:15:48] JM: Well, we had Rapid API on the show recently, and at a certain point you decided to sell the API marketplace business to Rapid API so that you could focus on the API gateway, because I think you got to a point where APIs were getting a little more hardened. But then you were kind of in a situation where, okay, you've got two appealing businesses that are both operationally intensive. Walk me through the decision to sell off the API marketplace business and focus on the API gateway.

[00:16:19] MP: Yes. When we released Kong in 2015, Kong immediately found a product market fit, and Kong is an open core product. That means we have an open source version of Kong that anybody can download for free and use and deploy in production. Then there's an enterprise platform that's available in an enterprise package that large organizations can purchase from us. That's the business model. It's very similar to Elastic. It's very similar to Confluent, Kafka and so on.

So in 2015 we released Kong, and Kong immediately takes off. So we have lots of developer adoption. We are building a business on top of Kong and it turns out that the marketplace for us became more of a distraction rather than a main core business we were doing. So we decided to divest the marketplace to great theme, the Rapid API team, that's been doing great things with the marketplace and then realign the company exclusively around Kong. Therefore, we even changed the name of our company from Mashape Inc. to Kong Inc.

[00:17:24] JM: Focus can be so important at a startup, and if you are going after two business that are viable, you may end up just destroying both of them. When you were in the position of running both the marketplace and the API gateway business, was there a point where you realized that these were competing for your attention in a way that was making the whole company suffer?

[00:17:51] MP: Definitely. Not just attention, but resources as well. I mean, the company at that point was still a very small company. We were less than 20 people, which means that we couldn't really focus 100% on both of them. So we had to make a choice. You know, Jeff, the marketplace at that point had been running for five years. For five years we've been trying to make that work. For five years we've been trying to work on it, fix it and keep iterating on it. There is something to say. After five years of basically struggling to marketplace successful, even the energy, the mindset, the mindshare energy that that product had in our minds at that point was very low.

As soon as Kong proven itself to be a greater opportunity for us, we made the decision to get rid of our work, the result of five years of work and focus on Kong. I mean, it was a hard decision. It wasn't easy for us to make, because it was our baby at the end of the day. The marketplace was basically what we spent our lives on for the previous five years, and at one point you have to make a decision, "What are we going to do here? Are we going to keep them both and then risk failing on both, or are we going to focus 100% of our energies on something that quite frankly made us more excited at that point?" That thing was Kong. So we decided to focus 100% on Kong not just money, not just company resources, but mind and spirit on Kong.

[00:19:25] JM: That decision is paying off. Let's have a conversation about Kong. So there have been API gateways before Kong. There've been load balancers. How was Kong different?

[00:19:40] MP: Like many products out there, an important factor for success is timing. Timing was very important for Kong, and it turns out that 2015 was a very good timing to release a product like Kong. In 2013, in 2014, there were some massive industry changes happening. One of them being Docker, which was released in 2013 and the other one being a Kubernetes, which was released in 2014.

Now Docker and Kubernetes fundamentally changed how we build our software, and that change led to a greater importance of APIs in any organization in any architecture. The reason for that, it’s simple. Where Docker and Kubernetes gave to everybody else the tooling to finally think disrupting their existing monoliths, refactoring those monoliths into microservices, and it’s much easier to manage a microservice rented architecture when you already have solutions like containers in place and Kubernetes in place and the entire ecosystem that’s around those tooling.

So when we released Kong in 2015, this change was already happening and APIs became of a greater importance not just for that external communication, the mobile communication, external clients like it used to be back then, but also for internal communication. Because as soon as we decouple our software in different services, the way these different components are talking to each other, it’s going to be APIs. The market of APIs just became so much more bigger because of containers, of Kubernetes, of microservices. So when we released Kong in 2015, we built a product that could work for external APIs, but also for internal APIs, and that made Kong very successful moving forward.

[SPONSOR MESSAGE]

[00:21:45] JM: This podcast is brought to you by wix.com. Build your website quickly with Wix. Wix code unites design features with advanced code capabilities, so you can build data-driven websites and professional web apps very quickly. You can store and manage unlimited data, you can create hundreds of dynamic pages, you can add repeating layouts, make custom forms, call external APIs and take full control of your sites functionality using Wix Code APIs and your own JavaScript. You don't need HTML or CSS.

With Wix codes, built-in database and IDE, you've got one click deployment that instantly updates all the content on your site and everything is SEO friendly. What about security and hosting and maintenance? Wix has you covered, so you can spend more time focusing on yourself and your clients.

If you're not a developer, it's not a problem. There's plenty that you can do without writing a lot of code, although of course if you are a developer, then you can do much more. You can explore all the resources on the Wix Code's site to learn more about web development wherever you are in your developer career. You can discover video tutorials, articles, code snippets, API references and a lively forum where you can get advanced tips from Wix Code experts.

Check it out for yourself at wicks.com/sed. That's wix.com/sed. You can get 10% off your premium plan while developing a website quickly for the web. To get that 10% off the premium plan and support Software Engineering Daily, go to wix.com/sed and see what you can do with Wix Code today.

[INTERVIEW CONTINUED]

[00:23:43] JM: Let's go through a couple examples. A product like GIPHY, GIPHY is an API for GIF search. If a user makes a request to GIPHY, they're looking for a GIF about basketball, or ice cream. What is involved in that request from an external user and where does Kong fit into the lifecycle of that request?

[00:24:08] MP: Yeah. Every time we're making a search on GIPHY, every time we're trying to pull content from GIPHY, all of that happens via an API. That's an API that the client could be a browser, could be a mobile app is making to GIPHY to retrieve that content.

Now of course these clients have to be authenticated, have to secured, have to be authorized, have to be rate limited and so on and so forth. So Kong sits in between that connection. So when a client consumes GIPHY, in reality what the client does, it's consuming Kong and then Kong redirects that request to the GIPHY API server. Basically Kong, it's the underlying technology that's powering these API requests.

[00:24:52] JM: How does that contrast with an internal API? Like if I'm working at a bank and I make some request to a microservice within the bank, how does Kong fit into the private internal APIs?

[00:25:08] MP: Yeah. With Kong, we are working with very large customers. So when you think of top global 5,000, top Fortune 500 organizations, each one of them needs to modernize their architecture. What they're doing is decoupling their large legacy systems into smaller microservices that now talk to each other. The technology that powers these new next generation architectures are APIs still, because one service communicates with another service via an API that happens to be internal and not external anymore. Guess what? It's still an API. So likewise, an external client needs to be secured, authorized, authenticated, rate limited and so on. Well, even an internal API needs the same facilities. So the organization at that point has a choice. Are they building that by themselves or are they using something like Kong to do that for them?

[00:26:06] JM: The architecture of Kong is that it's often fronted by a load balancer. So there's a load balancer in front of Kong and then there are multiple instances of Kong. But then Kong is also doing some routing itself. It's arguably a load balancer itself. Why not have Kong as the frontier? Why do you need a load balancer in front of Kong?

[00:26:32] MP: It can actually work both ways. The most popular solution would be to have a load balancer in front of Kong, because the assumption is that we're going to have multiple Kong nodes to process the incoming traffic to our APIs. So we need something that redirects the original traffic to these Kong nodes. Especially in the context of cloud, usually users decide to put let's say AWS, elastic load balancer in front of Kong, and that effectively is a load balancer that runs in front of Kong. But you're right. I mean, if the user wants to run Kong as that initial load balancer, that's still possible. So it can work both ways.

[00:27:10] JM: What role does Kong play in authenticating requests?

[00:27:15] MP: Yeah. That's one of the most common functions for an API gateway in general and for Kong as well. Kong has more than 10 different authentication or authorization schemes that the developer can apply on top of any API that's powered by Kong. So for example, let's say that the user wants to protect the API with an API key with OAuth 2.0. Instead of building that system, the user can leverage Kong to do that for him.

For example, even large organizations have this big problem. They have lots of services. They have lots of APIs, the risk is that each developer, each team that's building those APIs are going to reinvent security over and over again, which creates fragmentation in the long term. These organizations are using Kong as a way to consolidate security across multiple APIs and multiple services by using either an integration with LDAP, Active Directory, OpenID Connect and so on.

[00:28:17] JM: Let's take a step back. I should have asked this earlier, but can you just describe the architecture for Kong as it stands today?

[00:28:24] MP: Yeah. So when we built Kong, we really built Kong to be the fastest gateway. So we made some very specific technological decisions to achieve that, and then we built Kong to be extensible. So when we're thinking of Kong, we're thinking of a very fast runtime that's built on top of a framework called open OpenResty.

OpenResty is a framework that extends nginx with Lua scripting. Now, Lua, it's a very interesting programming language. It's very popular within mobile applications, within gaming, within embedded devices, and the reason for that is that Lua is a very good language to work in constrained environments. Very good performance, very low memory consumption. So by scripting nginx, which is the underlying core, with Lua and building our gateway with Lua, nginx and a virtual machine called LuaJIT, we're able to make the performance of Kong extremely low latency. Being low latency, it's not just nice to have. It's kind of a requirement as soon as we are using Kong in a microservice rented architectures. The reason being, we have multiple service communications going from A, to B, to C and then all the way back, and the gateway has to be fast or otherwise that latency will compound overtime.

For example, our stack, which is nginx, Lua and LuaJIT, if you're familiar with a company called Cloudflare, the global CDN. They're competing with Akamai. Cloudflare, it's processing, if I'm not wrong, 8% of the world global internet traffic on the same technology; nginx, Lua and LuaJIT. With Kong, we took this stack and we built a very high-performant gateway with it that not every user, every developer, every organization can run within their systems.

The second thing with that was to make Kong extensible. So we do have this concept of plugins, and plugins are functions, middleware functions that developers can build on top of

Kong to pretty much allow Kong to do anything. So out of the box, we of course have a few plugins that developers can use immediately. Think of security plugins. Think of rate limiting plugins, or authentication plugins, logging, monitoring plugins.

But then the developer can also have full control of Kong by creating additional plugins on top of Kong. So if there is any specific requirement or a legacy integration that has to be built, the developers can do that via Kong. As a matter of fact, Kong also has an ecosystem of plugins. We have more than 500 plugins on GitHub that developers have built to extend what Kong does out of the box.

[00:31:17] JM: The core functionality of Kong's proxy layer is nginx. So for those who don't know, explain what nginx does.

[00:31:29] MP: Nginx is a very fast web server that has a very powerful asynchronous run loop to process incoming requests. So in the world, usually there are two different popular web servers. There's like plenty of them, but the most important ones are Apache and nginx. Now Apache traditionally used to start a new tread for every incoming connection, which would have a higher memory and CPU consumption on the server.

The innovation that nginx introduced would be to get rid of these one tread per connection kind of system and introducing in asynchronous run loop that would run multiple requests on the same tread. So nginx is one of the most popular web service because of that, and basically the market is Apache and nginx as we speak. So nginx has been running in production in front of websites, in front of the entire internet, 60% of the internet for the past 10 years, and we took nginx, which has a very solid foundation for our gateway. We took that and we extended it with Lua to be able to support the API gateway features.

[00:32:44] JM: Just to clarify for people. A lot of people listening might be saying, "What are you talking about? I write all my server code in Node.js, or I write it in a Django app." What's the difference between the notion of a server for Node.js and the notion of a server for nginx?

[00:33:02] MP: Correct. There are many developers that are accepting requests directly, for example, in Node.js. Perhaps you can even do that. Now many other developers though prefer to put something like nginx in front of their Node.js servers, because nginx has been hardened to support a whole variety of use cases that effectively protect that Node.js server in the first place, unless the developer decides to implement all of those features within the Node.js server as well.

For example, nginx can implement some very good DDoS protection, or web application firewall features that ultimately will help that Node.js server work better by securing it from malicious requests, for example. Nginx in the first place is also very good product to be able to handle lots of incoming requests and then keep them alive while the Node.js server in the backend is slowly processing them overtime. It's just a good way to make the Infrastructure that we're building more solid.

[00:34:09] JM: Nginx can be configured in different ways to do different things. What does that mean? How would an nginx configuration differ depending on the application that nginx is fronting?

[00:34:23] MP: Well, because nginx fundamentally can be put in front of any TCP traffic. It can be HTTP. It can be TCP. It can be streaming. It can even be UDP. It can be pretty much anything. Depending on what services and what servers we're putting behind nginx, then we'll have to configure nginx to be dealing with these different sort of use cases. For each one of them, we can also use nginx specific modules that are part of the nginx ecosystem to protect and extend what that specific traffic is doing.

[00:34:56] JM: Kong uses OpenResty to configure nginx. So you don't configure nginx by just configuring nginx itself. Can you help me understand that? Why not just configure nginx directly? Why do you need this OpenResty layer? I guess maybe we should discuss what OpenResty is.

[00:35:18] MP: Yes. OpenResty is a framework that's built on top of nginx and allows developers to extend nginx with customer strips and custom logic, business logic. There's another way of doing that, and that would be to create an nginx module, but that's extremely complicated and it's very easy to shoot ourselves in the foot with an nginx module.

With Lua instead, we have a higher level language that allows us to script nginx without making all of those considerations, but still keeping the same performance that we would experience with a module basically.

Now, OpenResty basically allows us to build on top of nginx. So with Kong, we use OpenResty to build the entire gateway features on top of nginx. Nginx per se is a web server. It's not an API gateway. Now API gateway doesn't mean the processing and securing of requests. It also means a whole set of different tooling and platforms that allow the developer in the first place to be successful with the API Infrastructure they're building.

Usually that entails having a developer portal. That entails having hooks to be able to integrate whatever CI/CD workflow they're using within their API development process. So what Kong does, it's extending the nginx web server to deliver gateway capabilities on top of that server by leveraging the OpenResty framework.

[00:36:38] JM: So as you've said, OpenResty uses the Lua language, which you don't see very often. So you said that that's because I think LuaJIT has a just in time compiler and it's really resource efficient. Tell me more about why OpenResty uses Lua and how Lua fits into the whole architecture of Kong.

[00:37:04] MP: Yes. Lua happens to be a very lightweight language that can be integrated with any C codebase. So what nginx has built in C. So what OpenResty does, it's implementing, integrating the Lua language within the nginx runtime so that effectively the request and response lifecycle on nginx can be escalated to any custom Lua scripting code that can now interact with that lifecycle. Effectively, we're hooking into the request response lifecycle of the Vanilla nginx and we're extending that with customer scripting we can build in Lua.

Now LuaJIT is a very fast Lua virtual machine and arguably one of the fastest virtual machines in the world for any language. Now the cool thing about LuaJIT is that it's so fast and it's so quick and it's so lightweight, it can really make the difference as we're doing thousands, millions, billions of customization on that request response life cycle on top of nginx. So LuaJIT is a very good piece of technology that makes our descripting in Lua very performant. So by integrating LuaJIT on top of nginx and running Lua code on top of that, now we have a very fast customizable environment that runs on top of a very fast and solid foundation, which is nginx itself.

[00:38:34] JM: The Kong API gateway can intercept a request response lifecycle and execute hooks. What is an example of a hook that I might want to execute as a request is coming through my API gateway layer?

[00:38:50] MP: Yeah. For example, basically a request comes in and nginx is receiving that connection. Now nginx, thanks to OpenResty, will ask custom Lua code, "Hey, do you want to do anything with this request and do you want to do anything with this TLS handshake, or do you want to do anything with a response getting back, or do you want to do anything with that client after the response is being sent back?" Basically we have different phases that we can hook into to tell nginx to do something else than what nginx would normally do, and that something else, it's effectively our Lua code running on top of it.

[00:39:29] JM: Okay. So there are these different phases through which a request goes as it's entering the API gateway and being processed by the API gateway. Take me through those phases.

[00:39:41] MP: For example, some of these phases can be the access phase when we first receive a request. Then another phase would be when we receive the headers, the response headers from the upstream server into nginx. Then another phase would be the body. After we receive the headers, we then receive the actual body of the response and we can act upon that. Then another phase would be the log by Lua phase, which basically it's being triggered every time the request response lifecycle has been concluded. Every time the client is receiving the last byte, now we can do something else in the log by Lua phase. Usually, for example, what Kong does in this phase would be to support plugins that allow the user to now log the entire transaction in any third-party system. That can be Kibana, the ELK Stack, or Splunk, or StatsD, or Prometheus, OpenTracing and so on.

[00:40:39] JM: Kong is a statefull application. There's state that needs to be maintained because maybe authentication rules change overtime, or other kinds of policies change overtime. Maybe you're making some different decisions about routing. What kinds of state need to be stored for an API gateway and what are the requirements for a state management system?

[00:41:02] MP: Yes, you're on point. So being able to store credentials, it's going to be one of them. Being able to store rate limiting counters is another use case for Kong. With that said, Kong per se is a stateless server, which means that we can get rid of Kong nodes, start new Kong nodes, horizontally scale the architecture without worrying about any state that's being stored within Kong.

The reason for that is that Kong, it's relying on a third-party dependency to store the state, and that can either be Cassandra or PostgreS. Usually where, for example, use cases where developers or organizations are trying to implement an API gateway layer that goes across multi-cloud, goes across multi-regions and they're trying to enforce a global rate limiting state across each one of these different deployments.

To do that, Kong, for example, stores the rate limiting counters in Cassandra, and Cassandra will then propagate those rate limiting counters across each region, each data center to keep them up-to-date. But Kong per se, the actual server built on top of nginx and LuaJIT, that server is stateless. So we can add Kong nodes if we have more traffic. We can remove Kong nodes if we have less traffic. As a matter of fact, we are processing use cases up to a million transactions per second in one of our enterprise deployments.

[00:42:30] JM: Tell me about why Cassandra is an appealing database for this purpose.

[00:42:37] MP: Yeah. Cassandra is an eventually consistent masterless database, which means that we can start a Cassandra cluster made of more than, three, four, five Cassandra nodes and we can write and read on any Cassandra node and then make the request successful. That's very different to systems like PostgreS where we have a master process that's going to handle the writes, and then we have perhaps other PostgreS servers that are able to handle those reads.

With Cassandra, we don't have to write on one server. We can write on any server and then Cassandra itself will take care of eventually replicating that information across all the other nodes. So it's a great database to use if you're trying to eventually consistently replicate information across multiple nodes and perhaps even multiple regions or multiple data centers. Cassandra was born from the DynamoDB paper that Amazon released a while back.

So DynamoDB, if you're familiar with AWS offerings. So DynamoDB effectively runs on top of the same paper. It's a highly-distributed, highly-scalable system, and Cassandra is built on top of the same specification. So with Cassandra basically, we can run the same highly-scalable, highly-decoupled and distributed database on top of our own Infrastructure.

[00:44:03] JM: Do people use PostgreS when they need stronger consistency?

[00:44:07] MP: Correct. They can use PostgreS when they either want more simplicity or they can use PostgreS when they do not have these large distributed requirements for Kong. For example, if they're running Kong in one or two different data centers or regions, PostgreS is great for that.

Now with that said, Kong supports either Cassandra for larger use cases or PostgreS for smaller use cases, but are also working into getting rid of the database dependency in the first place whenever Kong is not being used as a credential store. So let's assume that in the future you want to use Kong with OpenID Connect, which is possible today. OpenID Connect doesn't store any credential in Kong. It stores credentials within the OpenID connect provider's database. So Kong effectively wouldn't need a database by itself. We can just leverage that third-party service for doing our credential validation. So we're going to work into enabling a DB-less Kong deployment very soon in the future.

[SPONSOR MESSAGE]

[00:45:20] JM: This episode of Software Engineering Daily is sponsored by Datadog. Datadog integrates seamlessly with container technologies like Docker and Kubernetes so you can monitor your entire container cluster in real-time. See across all of your servers, containers, apps and services in one place with powerful visualizations, sophisticated alerting, distributed tracing and APM.

Now Datadog has application performance monitoring for Java. Start monitoring your microservices today with a free trial, and as a bonus, Datadog will send you a free t-shirt. You can get both of those things by going to softwareengineeringdaily.com/datadog. That's softwareengineeringdaily.com/data.

Thank you, Datadog.

[INTERVIEW CONTINUED]

[00:46:16] JM: Let's go a little bit deeper into that credential question, because I think the credentials and the policy management is something that can potentially slow down an API gateway if it's not architected correctly. So if my API gateway gets a request coming in and I need to go out to some remote database that's not adjacent to my Kong or my API gateway instance, that could really slow down the authentication experience. How do you keep the authentication step performant?

[00:46:48] MP: You are correct. Authentication can be very tricky, and especially when relying on a third-party service, it can add some latency. So that's why it's important as the developer configures authentication on top of Kong, he's also being provided with options to perhaps cache that credential in-memory for a few seconds or for a few minutes so that Kong doesn't have to rely on a third-party credential store on every request, but it does that the first time and then it caches that result in-memory for a configurable amount of time, which means that we are reducing the latency between Kong and the third-party credential store and we are increasing the overall speed of the system.

That of course comes at the cost of consistency, because if you're caching in-memory, the credential, and we're validating that credential, let's say just once every 60 seconds. If we do revoke that credential from the third-party credential store, then we may have to wait up to 60 seconds in this example before that credential is being invalidated by Kong, by the in-memory cache of Kong.

[00:47:58] JM: So on the initial authentication, is a typical pattern that Kong looks at Cassandra for authentication information and then it might cache the data in some kind of in-memory data storage system? Where is it being stored in-memory? I guess take me a little bit through that data management issue.

[00:48:19] MP: Yeah. A request comes into Kong, and Kong needs to understand, "Okay, what's the API we're trying to consume? What's the credential, and is the credential valid?" Kong will make a request to either Cassandra or to any third-party credential store we might decide to use, like any OpenID Connect provider.

Once we retrieve this information, Kong can optionally cache this information in-memory so that when a new request comes in, a second request comes in into Kong, we do not have to look up the database every time, but Kong will just look up, it's in-memory cache, to understand if the client is valid or not. This is a technique that allows Kong to reduce the number of requests it makes through the database or to an OpenID Connect provider. By reducing the number of those requests, we make the system faster at the cost of having it a little bit less consistent whenever that credential is being revoked, because now it might take up to 60 seconds if the cache if 60 seconds to invalidate that credential in Kong.

[00:49:23] JM: If the API traffic gets really heavy, what happens during a scale up? Do you scale up Kong itself? Do you add multiple instances of Kong, or do you scale – What do you have to do for scalability?

[00:49:38] MP: You can just add more Kong nodes, and that would take care of that.

[00:49:42] JM: Okay. All right. Simple enough. Do you have to add more database nodes in that case also?

[00:49:49] MP: You may or may not depending – Cassandra, it's a very high-performing database. We do not need many Cassandra nodes to handle an extremely high amount of requests. With that said, Kong is really trying not to rely on the database as much, but only on the first request like I just explained. So usually keeping the same database nodes should be sufficient.

Of course, this varies on the number of traffic, on the amount of traffic we're trying to handle. If we're going from handling 1,000 requests per second to handling out a million request per second, then yes, most likely we'll have to scale the database as well.

[00:50:27] JM: Okay. Well, we've gotten a pretty overview of the architecture at this point. Let's talk about more of a business point of view. So you have a product, Kong Enterprise. People can just deploy Kong by themselves, because it's open source. What are people buying from you when they go for the enterprise suite?

[00:50:47] MP: Yeah. Kong, it's an open core product. The open source suite provides all the open source plugins. Provides the open source gateway functionality, then the enterprise package, it's built on top of the open source core and extends the core with an enterprise platform. The enterprise platform delivers visibility, delivers management features. It's basically a controlled plane that allows the organization to manage all of their APIs, all of their teams across any team, any business unit within that organization.

[00:51:23] JM: So this begins to sound something like a service mesh. In fact I just did an interview yesterday with Linkerd, William from Linkerd, and there's a lot of overlap between what you're doing and what a service mesh does, because a service mesh is this thing where you have a sidecar with all your different services and you can use this central control plane to do things, like authentication management and policy management and it seems like Kong could also be used for this. How does the Kong architecture compare to these Kubernetes service meshes?

[00:52:06] MP: Well, Kong supports Kubernetes very well, and the latest release, Kong 1.0 also supports service mesh. So in the industry, we're seeing that there are four main architectures that usually are being adapted by our developers or customers. One of them is the traditional monolithic architecture. Then the next one is the mini-services architecture, which basically allows different teams and different products to communicate with each other internally.

Then after that, we're seeing microservices, where one of those applications can be either further decoupled into separate services communicating with each other. Then we are also seeing serverless with AWS Lambda, Azure functions and so on in certain use cases.

Now with Kong, you can use Kong on top of each one of these different architectural patterns. You can use Kong for traditional API gateways. You can use to connect teams internally and their products. You can also use Kong for service mesh if you're decoupling your monolith into separate components. Now with Kong, you effectively have one runtime that can handle both the traditional north-south traffic and service mesh east-west traffic within the organization.

On top of that, the service mesh that Kong delivers, it's still relying on the same runtime, which means that it's still extremely fast, extremely performant and via plugins can also be extended to do things that perhaps Kong doesn't do out of the box in a service mesh deployment. But you're right, the feature set of north-south and the feature set of east-west is very similar and all the things we usually do in east-west or all the things we usually do in north-south will have to adopt in more and different patterns.

So being able to do rate limiting, circuit break [inaudible 00:53:52]. Being able to keep that latency down. Being able to observe what's happening within our systems. It's not a concern that's specific to one or another. It's a concern that both north-south and east-west will have to fix. Therefore, Kong is being used for that today.

[00:54:08] JM: So is the service mesh model for Kong, is it the model of deploying a sidecar to each of the services and then having a central control plane that the sidecars are communicating with?

[00:54:21] MP: You got it. So you can deploy Kong as a service mesh by injecting Kong as a sidecar proxy to a microservice, but Kong is platform agnostic. You can do that with Kubernetes. You can do that with a sidecar container, but you can also do that on any other platform that Kong supports. Kong supports 15 different deployment methods. We support containers, Mesosphere, COS, we support Vanilla Docker of course. We support any cloud; Azure, Amazon, Google, and we also support bare metal platforms.

Basically, with Kong, you could start a service mesh that runs across Kubernetes, but not limited to Kubernetes. Also on-premise and on virtual machines and any other cloud. So effectively it's a platform agnostic hybrid mesh that you can deploy with Kong.

[00:55:13] JM: When I talk to people about service mesh, performance is a really big concern, because if you have, for example, a service proxy that's deployed as a sidecar to all your containers and it's sitting next to your service and every service request is going through that sidecar, for example, if your service proxy is written in Java, then even just a GC pause can really give you a serious performance penalty, and that can be problematic if this thing is a bottleneck to every single service request. Tell me about architecting the sidecar, what you have built in the sidecar.

[00:55:58] MP: It turns out that when we built our runtime, so like we said, nginx, Lua and LuaJIT, we were able to achieve a great performance in most uses, a sub-millisecond processing performance. So that same runtime we've been running for traditional API gateway is also so lightweight and so fast that out of the box it support service mesh. So that's why we were able to support one runtime for both east-west and north-south.

[00:56:24] JM: That's pretty cool. That's sounds like a – But you were very happy to find that out.

[00:56:28] MP: Well, it happened for a reason.

[00:56:32] JM: I guess so. I mean, is there any material – So what you're saying is basically the same software that is in the API gateway that has been accepting requests from the external internet and doing routing through the API gateway in a performant fashion. You can just repurpose that in a sidecar to be able to have services communicate with each other.

[00:56:58] MP: Yeah. Not only that. We can do that with a better performance than Envoy, for example.

[00:57:02] JM: Really? So tell me more about like how you think this market is going to unfold. I go to these KubeCons, the Kubernetes Conferences, and if there's any water cooler or like hot topic right now, it's sort of like who's going to win the service mesh world, or do you need a service mesh? How do you think this world is going to unfold? How big of a market do you think there is for the service mesh usage?

[00:57:32] MP: Microservices, they're a very hot topic right now. Not everybody needs microservices. I mean, microservices, they're harder to deploy, harder to manage. There is a non-insignificant premium that developers will have to deal with once they move to microservices. So it's a great architectural pattern if we're talking about extremely large use cases that have to be hyper-scaled and hyper-optimized with the microservices architectures. Now when we talk about microservices, the way we make that microservice architecture work is with service mesh. Service mesh is not a technology. Service mesh is a pattern. So you can implement service mesh by using a wide variety of technologies that are available to developers today, one of them being Kong.

Now pragmatically, the enterprise and these large enterprise organizations that are going to adapt microservices are not going to exclusively have microservices running in their systems. We have to be pragmatic here. Some use cases don't make sense in a microservice rented architectures. Some others make lots of sense.

So the organization and developers will have to deal with a hybrid architectural pattern within the organization. It's not going to be exclusively microservices. It's going to be also monolith for some use cases. It's going to be serverless for some others.

So with Kong, we're looking at what our customers are doing and we're seeing that they're running monoliths and serverless and microservices alongside with each other, but they have a big problem. They cannot link them together and they cannot have visibility across each one of these different architectures from one place. So with Kong, we're giving a platform that can hook into each one of these different architectures, including service mesh, but also monolith and serverless. Then on top of that, have a consolidated way and runtime to do things like authentication, security, observability, mutual TLS, but also things like developer portal.

The idea of an API catalog, which was the whole point of Mashape back in the days which is being repurposed in Kong by having an internal catalog of APIs that runs internally for the enterprise, because as soon as we have a thousand different services running, we need to have documentation and a place where we can find and explore these different services. So that idea is also being repurposed, but it's not specific to one architecture. It's going to run across each one of these architectures.

[01:00:10] JM: That's a really bright future. Give me a little bit more context for where you're at today. I know Kong 1.0 just launched. Tell me more about where you are in terms of your plans and where you're going.

[01:00:23] MP: Yeah. Kong, it's one of the most popular service platforms out there. We have about a hundred employees in our San Francisco HQ. We do have more than a hundred enterprise customers, but Kong has an open source global adaption as well. So we're running more than 70,000 active Kong nodes across the world in either gateway or mesh deployments.

We do have a global community. We do have 38,000 community members that are contributing to Kong in some form or another either by creating plugins by augmenting the ecosystem. So we're going as a company to keep nurturing this community adaption by making Kong open source even more powerful moving forward, but then we're also going to be working with our enterprise customers to create what we call a service control platform, which is the evolution of API management, a service control platform that allows these customers to manage their APIs across a different variety of architectures within their systems across a different variety of platforms, Kubernetes, but also bare metal, but also multi-cloud and so on.

[01:01:34] JM: Okay. Marco Palladino, thanks for coming on the show. It was really great talking to you.

[01:01:37] MP: Thank you, Jeff.

[END OF INTERVIEW]

[01:01:41] JM: GoCD is a continuous delivery tool created by ThoughtWorks. It's open source and free to use, and GoCD has all the features you need for continuous delivery. Model your deployment pipelines without installing any plug-ins. Use the value stream map to visualize your end-to-end workflow, and if you use Kubernetes, GoCD is a natural fit to add continuous delivery to your project.

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

Thank you so much to ThoughtWorks for being a longtime sponsor of Software Engineering Daily. We are proud to have ThoughtWorks and GoCD as sponsors of the show.

[END]