**Microservices & API Gateways, Part 1: Why an API Gateway?**



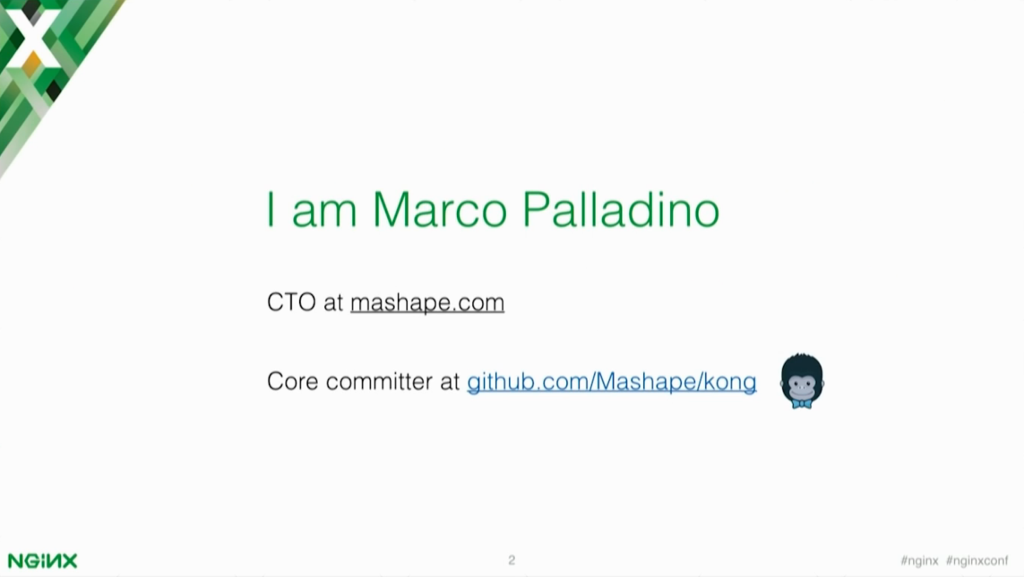
*This post is adapted from a presentation by Marco Palladino of Mashape.com at nginx.conf in September 2016. You can view a recording of the presentation on* [*YouTube*](https://www.youtube.com/watch?v=zs_QyRTIpDM)*.*

*This post is the first of two parts, and focuses on how and why to deploy an API gateway in your microservices application.* [*Part 2*](https://www.nginx.com/blog/microservices-api-gateways-part-2-how-kong-can-help/) *focuses on how Mashape.com’s open source API gateway, Kong, can fit into your microservices architecture.*

**Table of Contents**

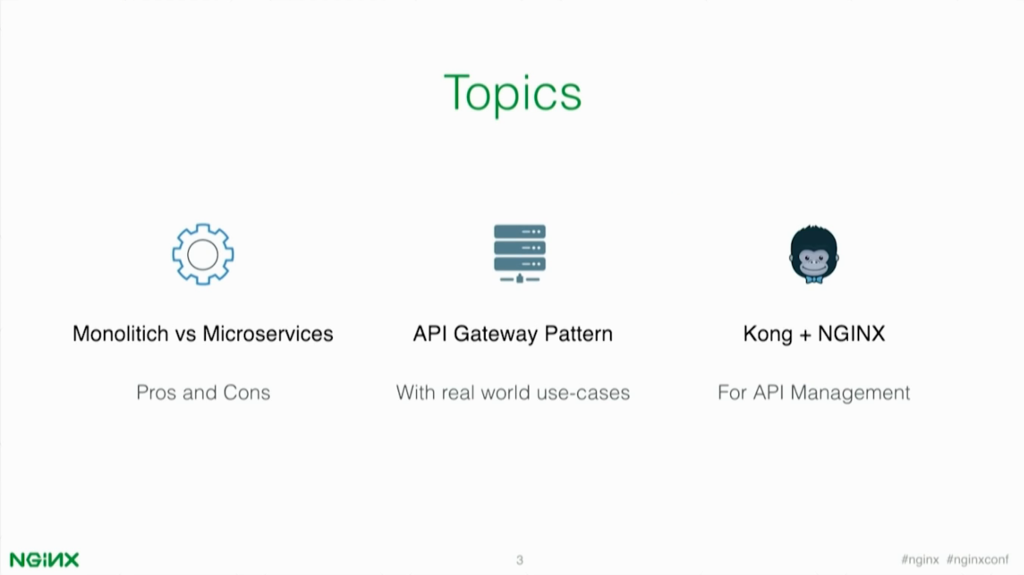
|  |  |
| --- | --- |
| 0:00 | [Microservices & API Gateways](https://www.nginx.com/blog/microservices-api-gateways-part-1-why-an-api-gateway/#MicroservicesAPIGateways) |
| 0:23 | [Topics](https://www.nginx.com/blog/microservices-api-gateways-part-1-why-an-api-gateway/#Topics) |
| 0:47 | [Monolithic Architecture](https://www.nginx.com/blog/microservices-api-gateways-part-1-why-an-api-gateway/#MonolithicArchitecture) |
| 1:45 | [Monolithic Application Pros and Cons](https://www.nginx.com/blog/microservices-api-gateways-part-1-why-an-api-gateway/#MonolithicApplicationProsandCons) |
| 3:55 | [Microservice-Oriented Architecture](https://www.nginx.com/blog/microservices-api-gateways-part-1-why-an-api-gateway/#MicroserviceorientedArchitecture) |
| 5:47 | [Microservice-Oriented Application Pros and Cons](https://www.nginx.com/blog/microservices-api-gateways-part-1-why-an-api-gateway/#MicroserviceorientedApplicationProsandCons) |
| 11:18 | [Why an API Gateway?](https://www.nginx.com/blog/microservices-api-gateways-part-1-why-an-api-gateway/#WhyanAPIGateway) |
| 11:54 | [API Gateway Pattern](https://www.nginx.com/blog/microservices-api-gateways-part-1-why-an-api-gateway/#APIGatewayPattern) |
| 12:53 | [Optimized Endpoints](https://www.nginx.com/blog/microservices-api-gateways-part-1-why-an-api-gateway/#OptimizedEndpoints) |
| 15:28 | [Centralized Middleware Functionality](https://www.nginx.com/blog/microservices-api-gateways-part-1-why-an-api-gateway/#CentralizedMiddlewareFunctionality) |
| 17:24 | [Ops: Blue/Green Deployments](https://www.nginx.com/blog/microservices-api-gateways-part-1-why-an-api-gateway/#OpsBlueGreenDeployments) |
| 18:50 | [Ops: Canary Releases](https://www.nginx.com/blog/microservices-api-gateways-part-1-why-an-api-gateway/#OpsCanaryReleases) |
| 19:34 | [Ops: Load Balancing](https://www.nginx.com/blog/microservices-api-gateways-part-1-why-an-api-gateway/#OpsLoadBalancing) |
| 20:36 | [Ops: Circuit Breakers](https://www.nginx.com/blog/microservices-api-gateways-part-1-why-an-api-gateway/#OpsCircuitBreakers) |
| 21:35 | [Building vs Running a Microservice](https://www.nginx.com/blog/microservices-api-gateways-part-1-why-an-api-gateway/#BuildingvsRunningaMicroservice) |
| 22:01 | [Division of Labor](https://www.nginx.com/blog/microservices-api-gateways-part-1-why-an-api-gateway/#DivisionofLabor) |
|  | [Part 2](https://www.nginx.com/blog/microservices-api-gateways-part-2-how-kong-can-help/) |

**0:00 Microservices & API Gateways**



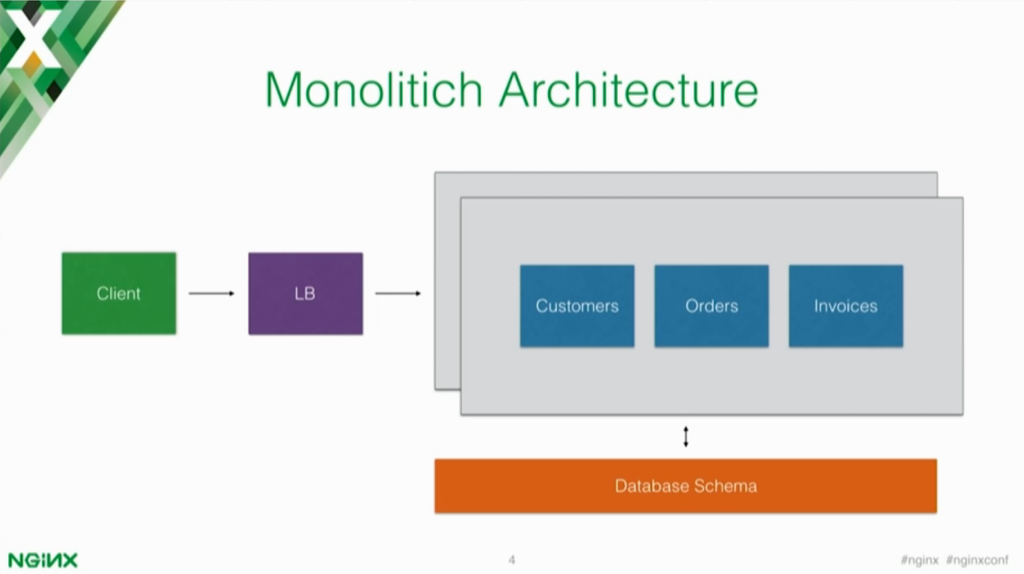
I am Marco Palladino, and today we’ll be talking about microservices and API gateways. I’m the CTO at Mashape.com, and one of the core committers of Kong. Kong is an API gateway, and today we will have the chance to look at how it works and how it can fit your architecture.

**0:23 Topics**



To understand why we need an API gateway, I’ll start by talking about monolithic vs. microservices architectures. What’s the difference? What’s the main shift? Then I will introduce the API gateway pattern and how that can fit a microservice‑oriented architecture. Then we’ll talk about Kong and NGINX.

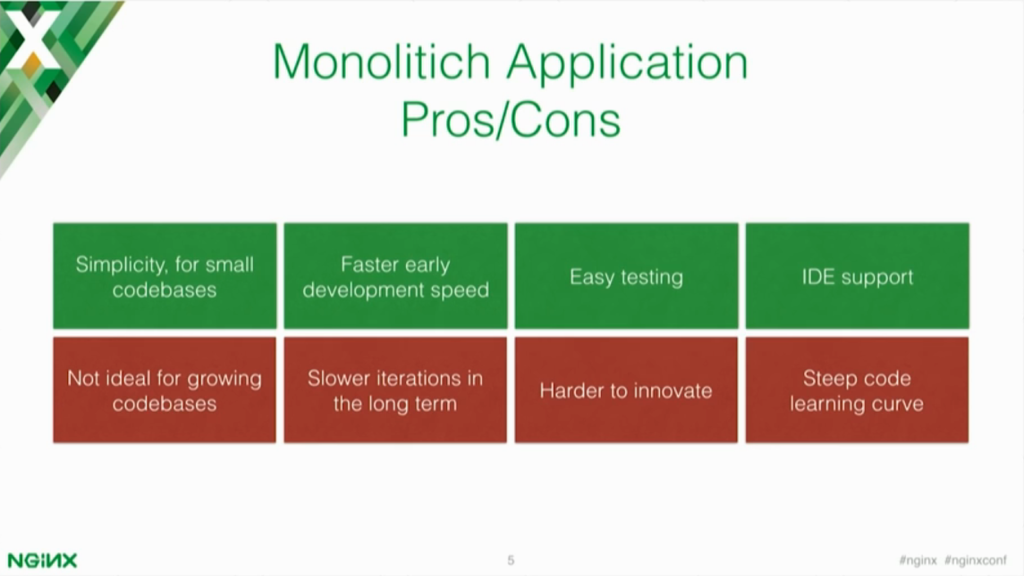
**0:47 Monolithic Architecture**



So, one of the things that we have been witnessing in the past few years is a transition from monolithic applications to microservice‑oriented architecture. We all are familiar with monolithic applications and how they usually work, and this is a simple representation. We have an app that includes all of the business logic and all of the entities that we’ll have to deal with. Usually we’ll have one data store that all of these entities in the app will use for storing and retrieving data.

Monolithic architectures can be horizontally scaled by deploying the same huge chunk of code over and over again on multiple servers. So every time we scale the the app, we’re scaling all of these components all together so it’s one unique piece.

**1:45 Monolithic Application Pros and Cons**



With every approach, there are pros and cons. Monolithic applications can be very easy to build and start with smaller codebases. We can build and bake everything in the same code base, which means we don’t have to worry too much about modularization and how to handle different components together.

It’s easy to test. Usually when we’re testing a monolithic app, we’re starting that one application and then we’re testing our integration unit tests. So it’s one app that we have to start and that’s pretty much it.

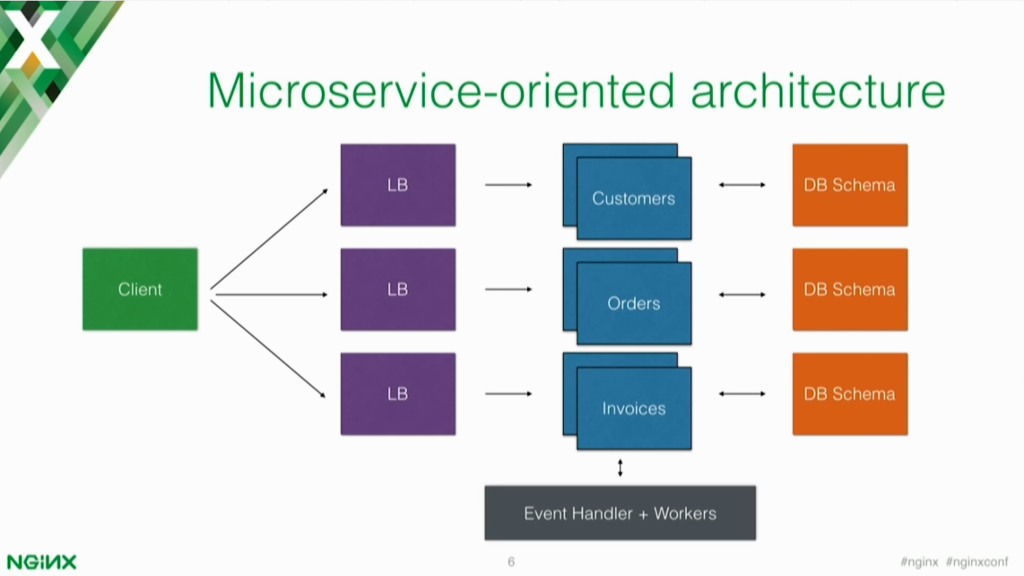
Also, IDEs [[integrated development environments](https://en.wikipedia.org/wiki/Integrated_development_environment)] have very good support for monolithic applications. For example, with Eclipse you can test with lots of tooling around monolithic apps.

But of course, having everything baked together into one codebase is not the ideal if you have a growing codebase. Over the years, the more features you’re building, the more code you add and it’s going to be harder to keep track of code in one place.

Because of that, teams will iterate slowly on top of this huge code base and the bigger it gets, the slower you will be able to iterate on it. Let’s say I want to replace a data store, or want to use a new technology. In a monolithic app, that’s usually painful because you cannot isolate a specific use case.

Because of all of this, you are scaling your organization it’s going to be hard for a team to understand what the codebase does.

**3:55 Microservice-Oriented Architecture**



What we’re witnessing in the past years is a transition to microservices. Containers are also helping in this transition because they create great tooling around microservices, but I’ll talk about that later.

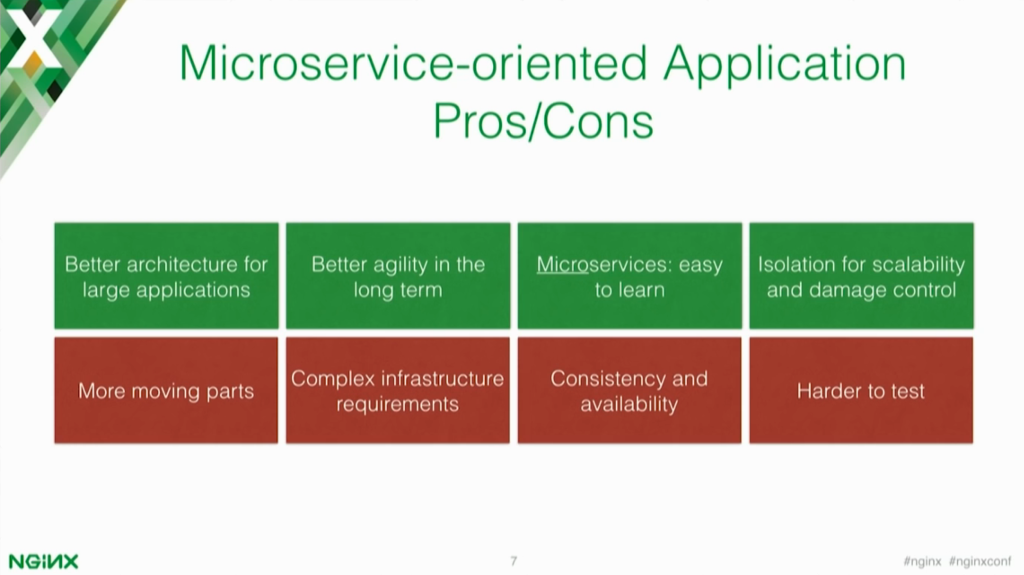
In a microservices architecture, you have the apps split into different components. So instead of having a codebase that handles all of the entities, now you have different services that are being deployed and scale independently from each other. In this example, if I have a customer’s orders and invoices, the entities in these components will be deployed in their own server.

The communication can then happen to multiple formats, usually HTTP or RPC. Sometimes you can also allocate different data‑store schemas to each of these components. So you are isolating how the component works, and you are making it work independently from the other components.

This also means that often you will need an event handler and some workers. In this example, the Client can create a new order and then instead of the client creating a new order and a new invoice, you can create a new order and the Orders component will then push an invoice event, and something else can listen to this event and create an invoice automatically for it.

So you can build an asynchronous application that does not depend on the client and it can work autonomously. This also means that, for example, if the Invoices component isn’t running, you can retry that operation later.

**5:47 Microservice-Oriented Application Pros and Cons**



Again we have pros and cons. Microservices are not a good fit for everybody, and not for every use case. It’s definitely is a good fit for larger applications. If you have a large application, you can split this large application into different components and developers will be able to iterate, maintain, and build those components independently from each other.

The analogy somebody said was that transitioning from a monolithic to a microservice‑oriented application is like having a full chicken and transitioning to Chicken McNuggets. Now you have all of these little components working together, and you can scale them independently.

It means better agility in the long term. Now that each component is doing only one thing and that one thing only, you can easily iterate and innovate over that component without disrupting the other components. Each component will talk to each other using an interface, for example a HTTP RESTful interface. You can change and experiment with implementation and as long as the interface doesn’t change, the app will keep running.

Microservices are micro, which means that if you need to scale the organization or the team, you can start assigning smaller pieces of the app to your team members. This in turn makes it faster for them to understand how the code works and can start iterating on the codebase faster. If they have to use another component, they can use an interface to consume the other component without necessarily having to dig into the code.

If something goes wrong in a monolithic app, usually everything goes wrong. If there is a bug or a problem, usually what happens is you have a high chance of the app going down or not working at all. With microservices, if you have a problem, it’s a problem that’s isolated in a specific component or service of your architecture.

This means that only that part of your app will stop while the other components can keep working. I was explaining how we have an Orders and an Invoices component. If for some reason somebody is pushing a bug on a Friday night on an Invoices component and it stops working, we can still keep track of what we have to do using the Event[‑handler] and Worker. So once the Invoices component is up and running again, we can then create those invoices that have not been created because of a bug.

All of this is possible to implement in a monolithic architecture as well, but what we’re seeing here is that the microservice‑oriented architecture encourages this development style while monolithic apps tend over time to become “spaghetti apps”.

There are significant cons to a microservice‑oriented architecture. First of all, you have lots of moving parts. You no longer have just one app that does everything in one place. You now have multiple components and services that have to work together at the same time in order to have a successful product running in production.

This means that you are suddenly having more complex infrastructure requirements. Now you need to have a way to easily deploy, scale, and monitor those different components independently. That’s also part of the reason why real‑time monitoring and analytics became so popular in the past few years. It’s because once you have a microservice‑oriented architecture, you really have to monitor each one of those pieces and have one central place where you can see your components running and their status.

In a monolithic app, you have one codebase that stores, executes, and links all of the entities together. In a microservice‑oriented architecture, we have different components that are each dedicated to doing just one thing.

This means you now have an availability problem – services can go down and may not be available. Or you can have a consistency problem – you may want to scale these microservices in multiple data centers. So you have to create an app that can take into account these new challenges.

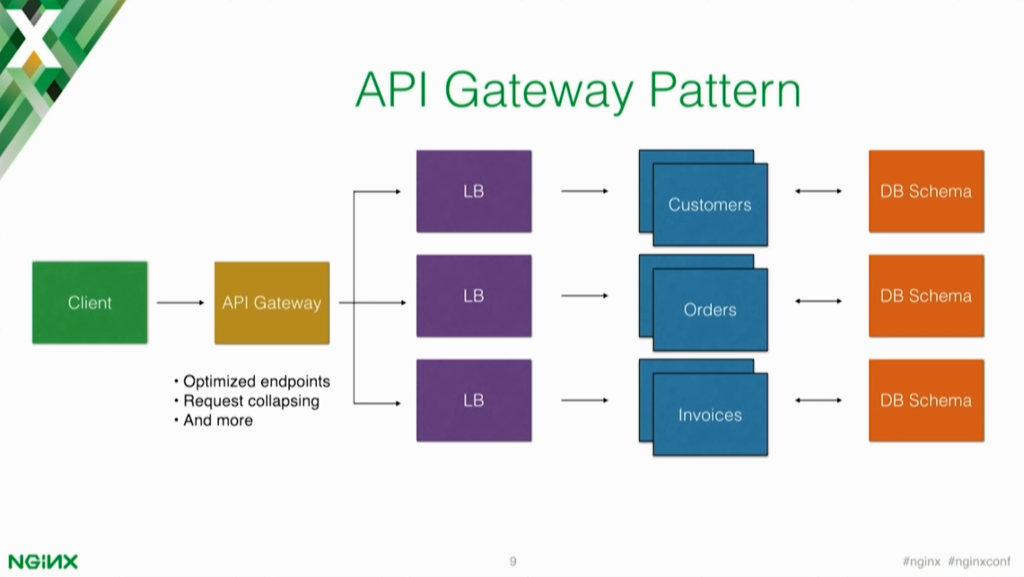
If you want to test a single service, it’s harder and easier to test, depending on what you’re testing. You can easily test an individual component, but it’s harder to test the big picture altogether. Usually if you want to test an app that’s built in a microservice‑oriented architecture, you have to start all of these components at the same time so you can have them talking altogether and can successfully implement an integration test.

**11:18 Why an API Gateway?**



So why do we need an API gateway, and how can an API gateway help in a microservice‑oriented architecture? We always hear the word *orchestration*, so I like this slide – it shows an orchestra with the individuals (who are the microservices) playing their own instrument. Then we have the director (who is the API gateway) who can somehow orchestrate how the requests are being processed by our architecture.

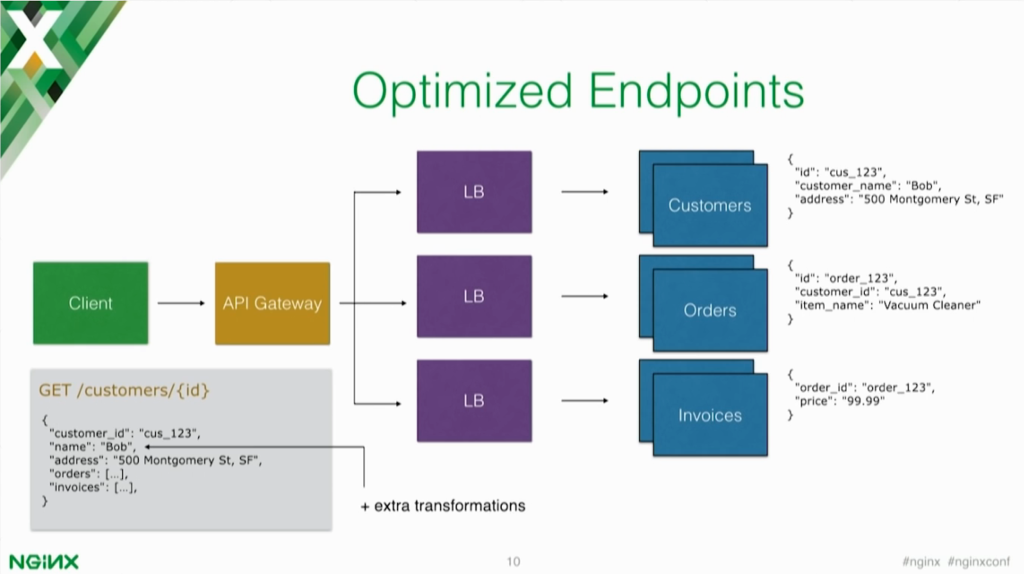
**11:54 API Gateway Pattern**



The API gateway pattern means that you put an API gateway in front of your microservices and make the API gateway become the entry point for every new request that’s being executed by the app. This can simplify significantly both the client implementations and the microservices app.

Before, the Client had to make a request to Customers, then to Orders, then to Invoices. The Client needed to understand how to consume these different services together. With an API gateway, we can abstract all this complexity and create, for example, optimized endpoints that the Client can use and that under the hood make requests to those components.

**12:53 Optimized Endpoints**



For example, optimized endpoints. If we assume that each component returns different JSON responses for Customers, Orders, and Invoices, and let’s say the Client wants to retrieve this information. There are two ways. In the first way, it’s making a GET request to the Customers component to retrieve the customer, then to Orders and then to Invoices for a specific order.

Or we can abstract this client implementation complexity by using an API gateway. The API gateway can then expose a specific endpoint that under the hood will make those requests and then return to the client one unique response after it [the gateway] has consumed the microservices.

This means, for example, we can collapse all of those responses into one response and one request. This helps a lot and is very beneficial especially for mobile clients. You can speed up the mobile implementations and you can also do things like transformations.

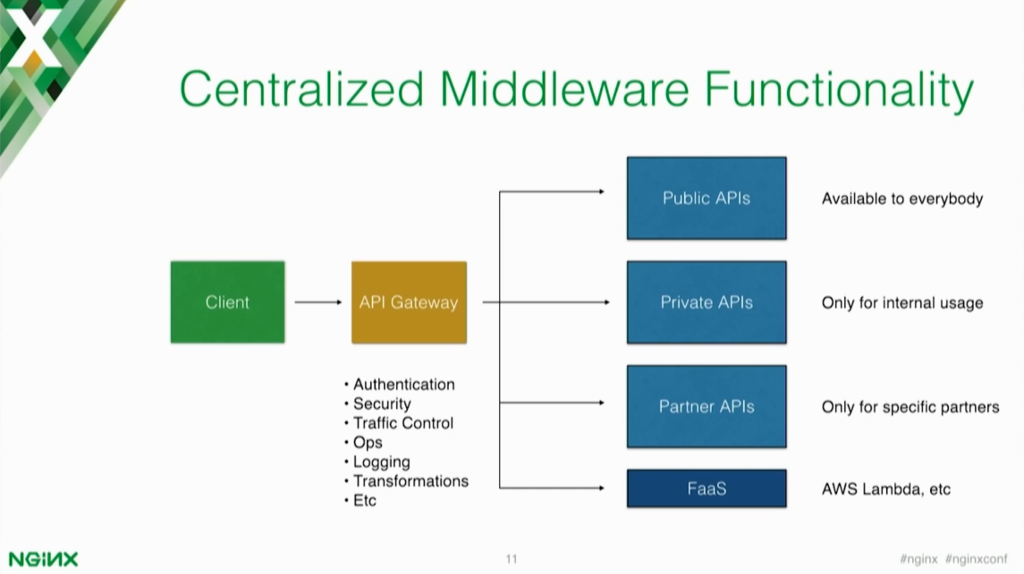
One of the nice things that an API gateway enables is the fact that the client doesn’t have to know a component’s address. The gateway will take care of that. You can change the implementation and you can even change the API interface. Usually that’s problematic because every time you change an API interface, you risk breaking existing clients.

With an API gateway, you can effectively abstract this on a separate layer so you can change the implementation and the interface while still keeping the public interface the same for existing clients. This means you can do things like transformations for example.

API gateways are also a great tool that help with transitioning from a monolithic app to microservice‑oriented app. If you have a monolithic app, you can start [by] putting an API gateway in front of it and then slowly you can start splitting the monolithic app into different microservices and keep the clients working through the API gateway.

The API gate will over time implement and consume the underlying upstream services while keeping the client working. Having an API gateway can help this transition.

**15:28 Centralized Middleware Functionality**



Of course creating optimized endpoints is just one of the benefits of having an API gateway. You can also centralize middleware functionality. As you start creating more and more services, you’ll find yourself in a tough spot – you’ll need authentication and traffic control for the services.

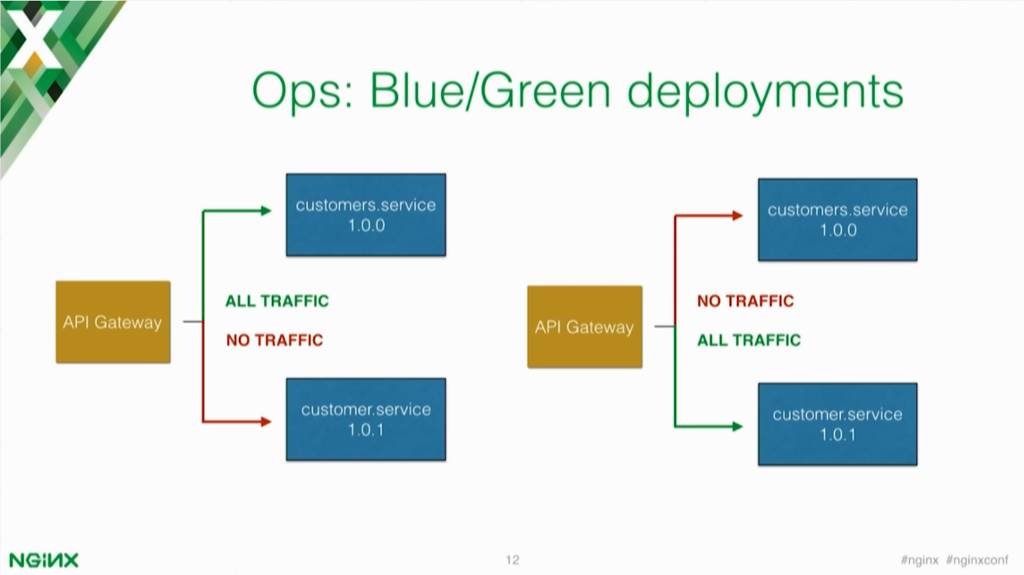
Some of them will be public, some will be private, and some will be partners’ APIs that you want to make available to just some specific users. Sooner or later you will be finding yourself in a place where you’re implementing the same middleware functionality over and over again inside of the microservice.

I believe the microservice should not take care of this complexity. The API gateway should take care of it, which means that the microservice will only be responsible for receiving incoming requests – generating a JSON response for example. The API gateway will then centralize all of the authentication, logging, and traffic control that you need to implement in your app.

Here in this slide, I also introduced at the last minute *FaaS* (function as a service), which is the new cool thing, apparently. An API gateway can work well with that use case as well. In short, FaaS means you’re consuming an API endpoint and that API endpoint does something somewhere.

It’s running in a serverless architecture, or rather on a server that’s not yours. This means you can put an API gateway in front of those functions as well, to run the middleware on top of that as well.

**17:24 Ops: Blue/Green Deployments**

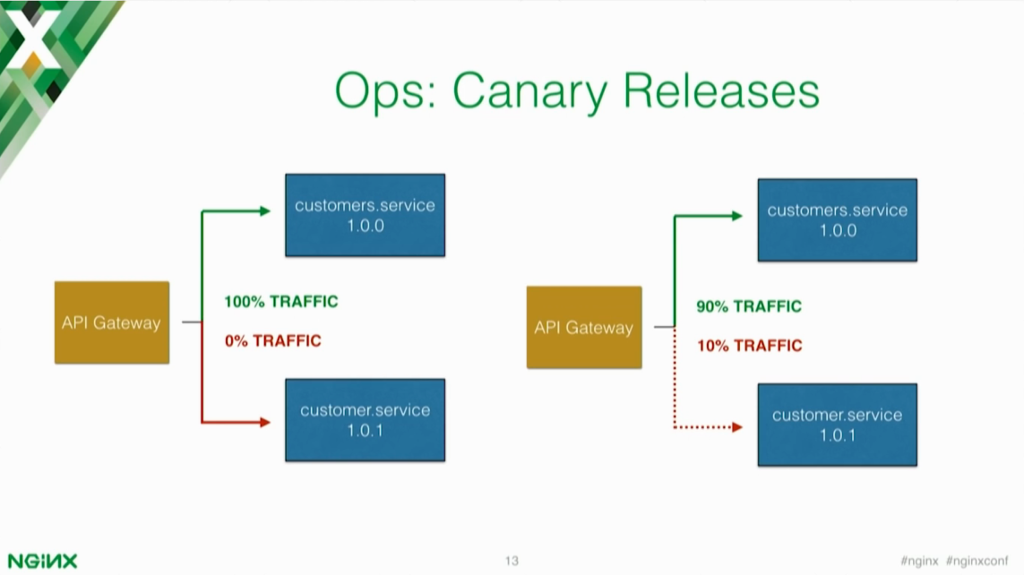


Here I’m just going to show you a few use cases of what an API gateway can do for you and how it can make your life easier if you are, for example, deploying software. In microservice‑oriented architectures, it’s easy to make one specific service deploy multiple times a day; it’s not as hard as having a monolithic application.

In a monolithic app, usually deployments are slower because every time you make a little change, you have to deploy the entire thing again. As the application grows and becomes bigger, that can become slow and problematic. With microservices instead, you can deploy individual components multiple times because it’s usually faster and easier if you’re just deploying a small chunk.

If you need, for example, to implement blue/green deployments, and API gateway can certainly help in making this possible in an easy way. For example, if you have customers running version 1.0.0 and customers running version 1.0.1, an API gateway can know the location of both of these components and then provide an interface to switch traffic from the old version to the new one.

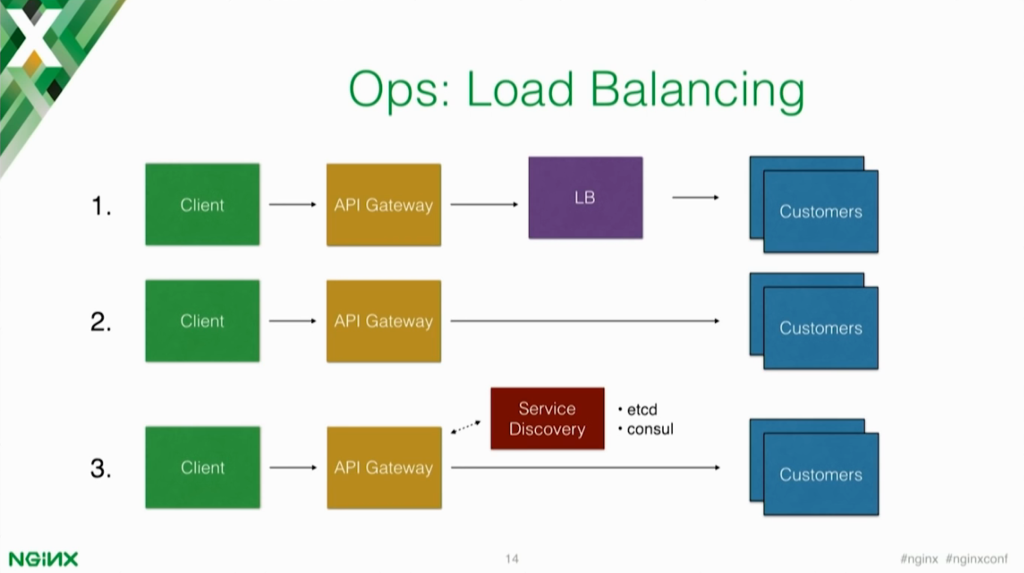
**18:50 Ops: Canary Releases**



Another deployment strategy is canary releases. When you’re creating software, sometimes you don’t want all the traffic to go all at once to the new version of your microservice because the microservice might not have been tested well or there are known problems.

Canary releases allow you to direct only specific amounts of traffic to the new version and an API gateway can take care of this use case. You can decide to proxy only 10% of the requests to a new version of your component and then once you’re sure there are no bugs, you can switch 100% of the traffic to it.

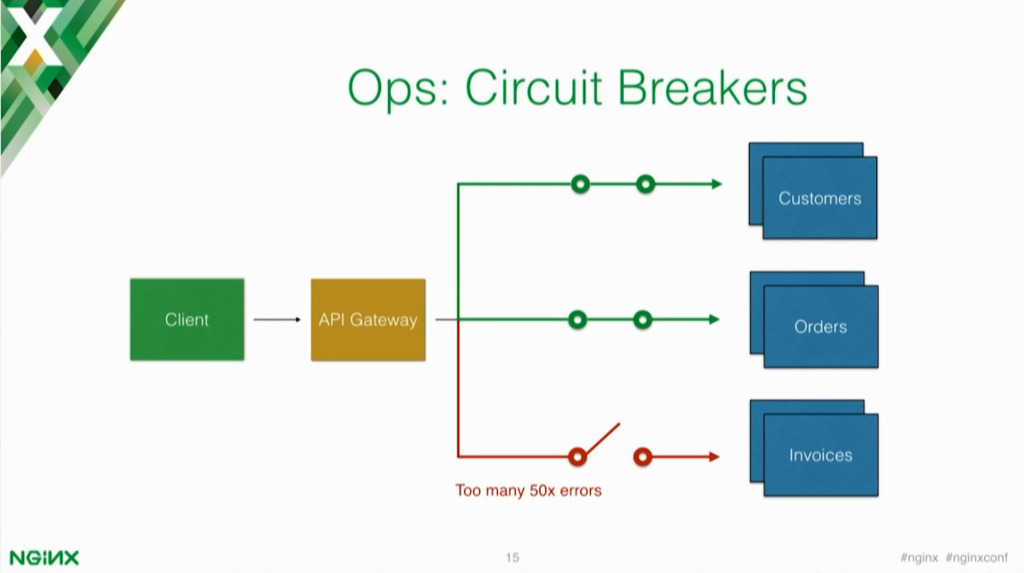
**19:34 Ops: Load Balancing**



Another thing that an API gateway can do is load balancing. In certain cases the load balancer could also be the API gateway. The API gateway knows where all of the services are located and their addresses, so it can either put a load balancer between the API gateway and the upstream service, or it is the balancer. When it is the load balancer, the API gateway leverages a service discovery tool like Consul or etcd to load balance those requests.

The service‑discovery tool will give you a new IP address every time you ask to resolve a DNS address, and effectively you’re doing some sort of a round‑robin load balancing at the DNS level.

**20:36 Ops: Circuit Breakers**

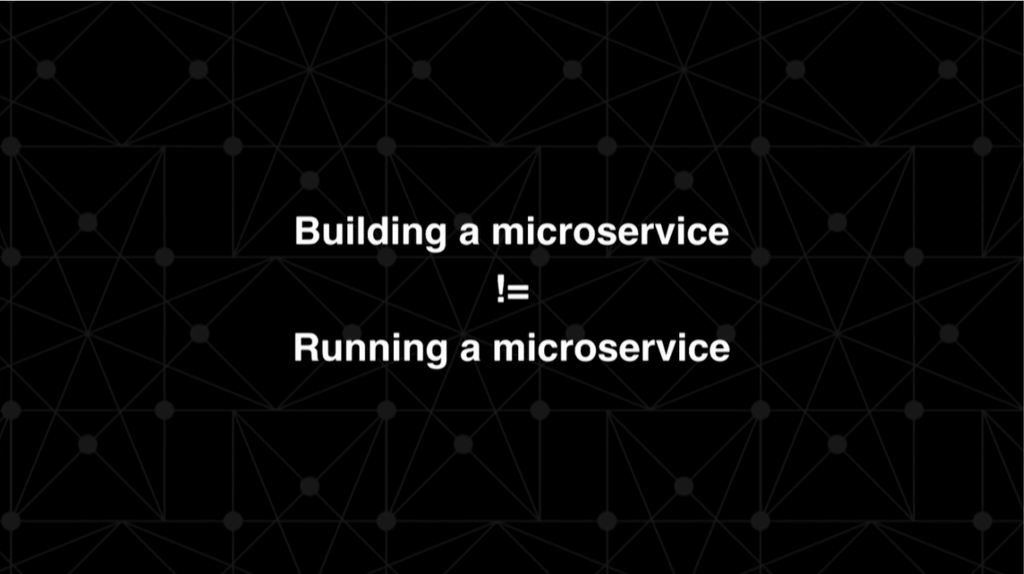


Let’s say that one of your services suddenly stops working and returns a very high number of errors. The API gateway can implement things like circuit breakers, which means that after a specific threshold, the API gateway will stop sending data to the component that’s failing.

This gives you time to analyze the logs, implement a fix, and push an update. Usually when you have a component failing and you have lots of traffic to that component, it’s using other components and you may have a cascade failure in your infrastructure.

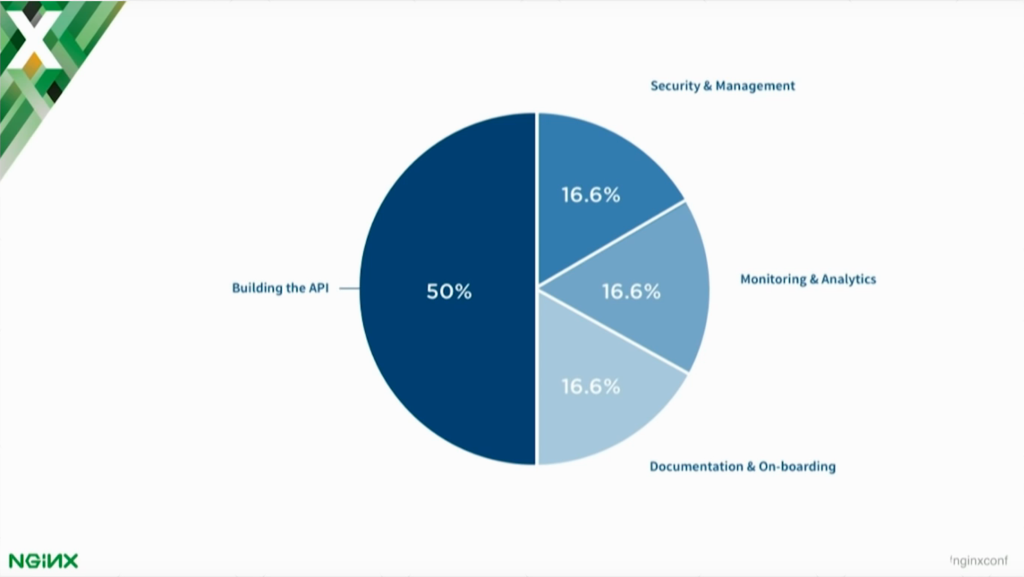
Circuit breakers simply close the circuit so no more requests are allowed to go to that component, and now you have time to fix and implement a solution to your problem.

**21:35 Building vs Running a Microservice**



Sometimes there’s also a misconception that’s very common. Sometimes product managers and software engineers think that building an API is the same as running an API and so building a microservice is the same as running a microservice, but those are two different problems. They have to get addressed in a different way.

**22:01 Division of Labor**



I think that building the API is usually 50% of the workload required when building a microservice. Then there’s this hidden 50% that not everybody takes into account. Now that we have an API running, it can accept requests and return responses, but how do we handle management, credentials, traffic control, and rate limiting?

Then the next step after rate limiting for our users is the issue of whitelisting just a subset of those users to have, for example, a higher limit.

Then monitoring analytics – it’s very important. You need to constantly be able to monitor and track the status of your components and documentation. Microservices can more often than not have an API that you can consume. If you have an API, you also want to have proper documentation. That’s important not just for public developers but for internal private developers in your organization as well.

You may have a voice API and an SMS API, and so if you want the two teams in New York and San Francisco to work together, the teams just have to release a doc for the API interface and then anybody inside of your organization can use that API. It’s very important to keep the documentation up‑to‑date.

This is also part of the 50% of the workload that’s usually not considered when creating a new API in general, but also mmicroservice‑oriented architectures.

*Continue to* [*Part 2*](https://www.nginx.com/blog/microservices-api-gateways-part-2-how-kong-can-help/) *to learn on how Mashape.com’s open source API gateway, Kong, can fit into your microservices architecture.*

Microservices Reference Architecture

Learn how to connect, secure, and scale microservices

[Download now](https://www.nginx.com/resources/library/microservices-reference-architecture/)