Microservices in action, Part 1: Introduction to microservices

Smaller, faster, stronger: Building better cloud applications from the ground up

Rick E. Osowski (osowski@us.ibm.com)
Technical Product Manager
IBM

26 June 2015

In this multipart series, learn how microservices make cloud applications more manageable, scalable, and reliable. This first installment gives you a high-level view of microservices' role in cloud architectures and contrasts microservices-based systems with older, monolithic models.

View more content in this series

Throughout 2014 and into 2015, *microservice* became the hot new buzzword, quickly supplanting *cloud*. This article is the first in a multipart series about implementing microservices. In this installment, I'll walk you through the history of microservices and what it means to build on a microservice architecture, providing a firm foundation for building and reviewing actual microservice applications in subsequent installments. I'll close with some topics that you'll read about later in this series, including microservice capabilities coming to IBM Bluemix throughout the rest of the year and into next year.

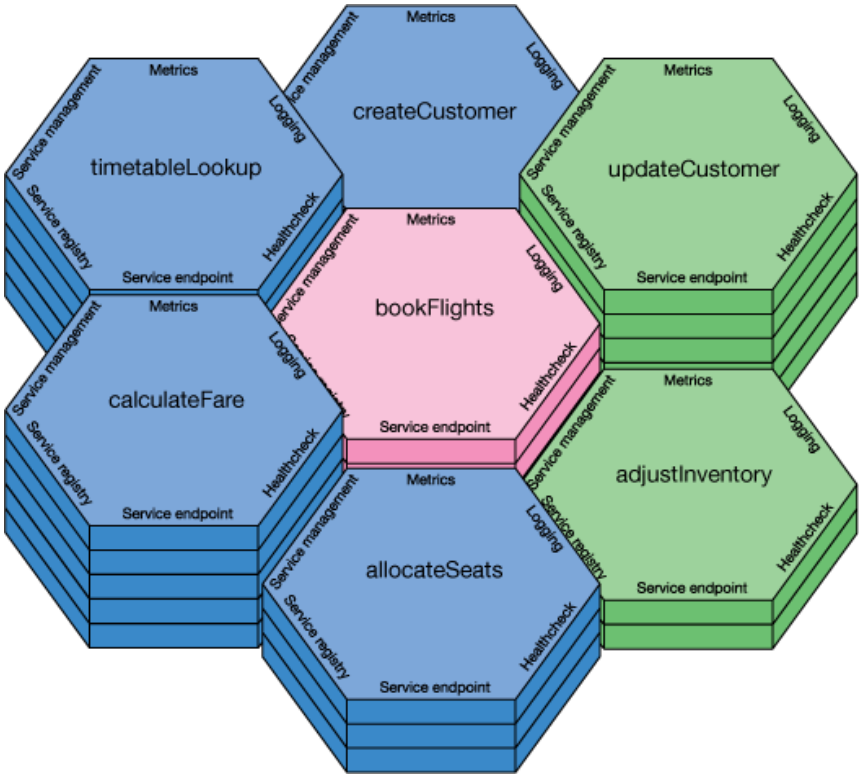
Netflix streaming: The birth of popularized microservices

Whether or not you've heard about microservices, I'm sure you've heard of Netflix. I'm even willing to bet that you've heard of Netflix Open Source Software (NOSS), thanks to Netflix's success in creating and releasing software for managing cloud infrastructure — the software that powers the Netflix digital-entertainment-streaming empire.

Starting around 2009 — driven completely by APIs and riding the initial wave of what we would come to know as microservices — Netflix completely redefined its application development and operations models. At that time, the company was derided by industry onlookers with "You guys are crazy!" or "This may work with Netflix, but no one else can possibly do this." Fast-forward to 2013, when most of those sentiments changed to "We're on our way to using microservices." More than 525,000 Google search results for *microservices* suggests that the concept is definitely both valid and powerful.

But what is a *microservice*? What is a *microservice-based architecture*? Figure 1 shows a conceptual view of microservices for a travel-booking service. Each of the seven tiles in the figure represents an individual microservice. They are arranged to show which microservices can interact with other microservices, providing necessary capabilities to both internal and external-facing applications. The services' different vertical heights represent how they are used in different quantities in relation to one another. Throughout this article, I'll cover the foundations of microservices so you can gain an understanding of how to represent your own microservice-based architecture.

Figure 1. Microservices conceptualized



For an in-depth explanation of what microservices started out as, read Martin Fowler's excellent blog post. What I'll try to capture here is the essence of that post, its application to your environments today, and how to get there.

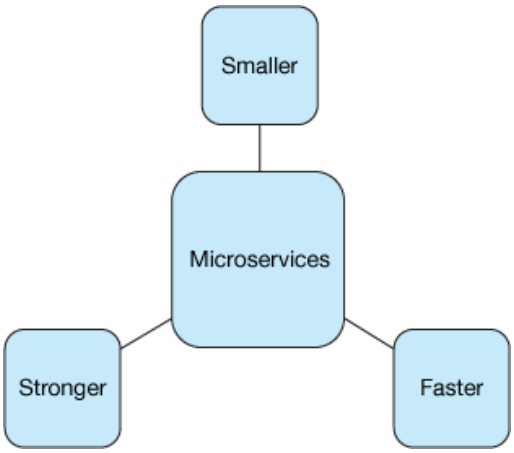
Follow-up reading

If you are not familiar with SOA but would like to be, you can read a great explanation of SOA's origins and how it technically differs from microservices at an implementation level.

In a conference talk, Adrian Cockcroft, formerly of Netflix, defined microservices as "fine-grained SOA (service-oriented architecture)". You don't need to be intimately familiar with SOA (an architecture style coined more than a decade ago), just the terms used in the acronym. Ideally, you are building an entire architecture out of services from day one. In microservices, each of these services has a single solitary purpose with no side effects, enabling you to cover greater scale with fewer overall dedicated engineers.

To define microservices and the associated architectures, I'm adapting and modifying the "bigger, faster, stronger" phrase used to describe modern athletes: **smaller, faster, stronger** (see Figure 2). In essence, microservices are many smaller architectural components, built and delivered with speed, becoming stronger, both independently and as a whole.

Figure 2. Microservices: Smaller, faster, stronger



Smaller

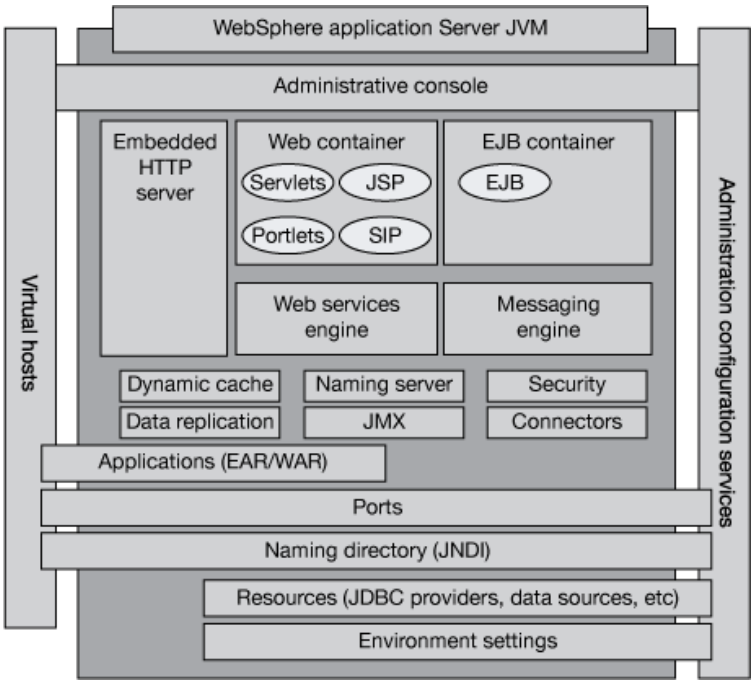
Microservices means no more monoliths. Monoliths are big, clunky, slow, and inefficient, like the Grim Monolith in Figure 3. We are moving away from a world with 2GB WAR files (yes, just the WAR file. Not the application server or operating system components. This is a true story!) to a world populated by many services of 500MB each, containing entire applications, servers, and necessary operating system components.

Figure 3. A Grim Monolith

The migration from mainframes to client/server architectures was a large step and one that many companies and developers alike struggled with. The more recent migration from core web-based application servers to SOA adoption was a similar struggle. Many components included in application servers of years past lend themselves to microservices; however, they are still packed inside multigigabyte installation binaries. Figure 4, for example, shows the architecture of

a traditional web application architecture, deployed using WebSphere® Application Server and Java™ Enterprise Edition components.

Figure 4. Standard WebSphere Application Server Application architecture

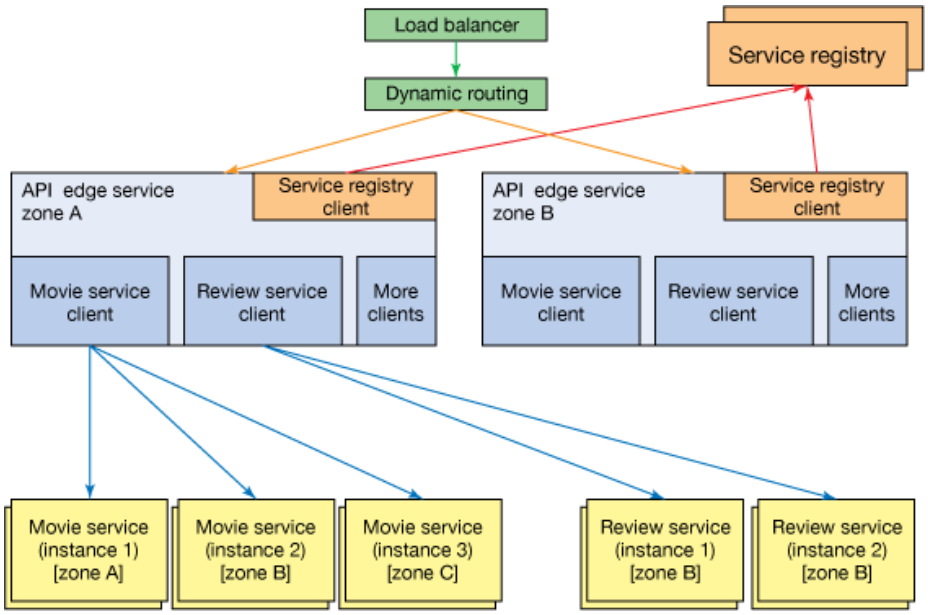


Microservices are an exercise in integration with all interacting components being much more loosely coupled. The entire idea of microservices becomes *plug and play*. I will touch on this more in the Stronger section, but essentially a microservices-based system employs the *shotgun method* at scale, to maintain and secure more small components instead of fewer large components. You remove single points of failure and distribute those points of failure everywhere.

Building for failure can only be done with smaller pieces. If you build a monolith for failure, you spend too much time focusing on the inefficiencies of every edge case. If you build a single service instance for failure, other service instances take over when consumers make requests.

The diagram in Figure 5 is one example of an implementation that uses microservices.

Figure 5. Conceptual routing example of a video-streaming application



In Figure 5, each individual box is maintained on its own, scales on its own, knows where it sits, and knows where to get the information it needs. Not every microservice architecture requires every component in this diagram, but they do help!

Comparing Figures 4 and 5, you can see the difference in how a similar application would be deployed. In Figure 4, everything is deployed to a single process on a vertically scaled system. When more throughput is required, the entire server stack is stamped out again and again. Each server runs inside its own process. The only way to get more throughput in Figure 4's *Web Services Engine* or *EJB Container* would be to scale the entire *Server JVM* to a new instance inside of a clustered environment. However, this would also create another *Web Container*, another *Embedded HTTP Server*, another *Messaging Engine*, and so on, whether or not those components need to be scaled.

In contrast to Figure 4's type of scale, Figure 5 has components that scale independently. I'll touch on the individual components and how they scale in the Faster section, but for now focus on the distributed nature of each component and service. Unlike in the example application in Figure 4 — which requires the full highly available web application server stack to provide availability — these components are distributed by nature and only provide a single, focused capability, often using different technology from other components. This structure enables the application architecture to evolve much **faster** and include newer technologies, as well as newer releases, independently of the other components.

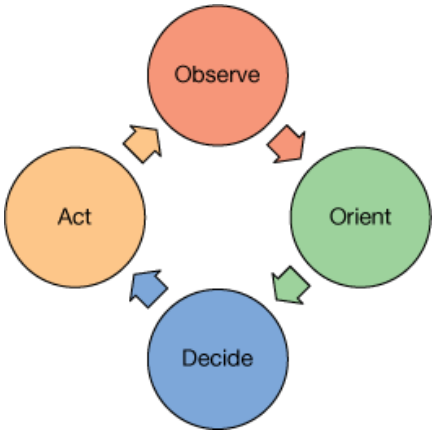
To summarize, smaller is better to develop, operate, maintain, and interact with.

Faster

Another buzzword that surfaced along with *cloud* is *DevOps*. The DevOps movement empowers developers to control more of their code along the delivery pipeline, integrate continuously, and

achieve more visibility. The main principles of DevOps, as shown in Figure 6 in clockwise order, are *observe*, *orient*, *decide*, and *act*.

Figure 6. Principles of DevOps



What could be better than delivering smaller pieces faster, right? There's no way you can deliver updates to a monolithic application server instance every two weeks, but in that same time frame you can definitely deliver updates to a single service consumed by many other services. Less overhead is incurred by smaller components when you build new staging environments, move items through your pipelines, and deliver to production environments.

Don't get me wrong. Continuous delivery and integration can still be done with monoliths. I've seen it happen. However, then you're juggling boulders, not marbles. It's much easier to recover from dropping a marble than from dropping a boulder.

The development cycles associated with DevOps lend themselves well to microservices. You are aiming for shorter development cycles that continuously add functionality, instead of longer development cycles that build a complete holistic vision at once. This development methodology, known as Agile, is a fundamental practice responsible for the success of DevOps. Whether you choose iterative or incremental development, the combination of microservices, DevOps culture, and Agile planning enables you to quickly build out an entire infrastructure in the time it would have taken you to plan your first waterfall cycle in years past.

The other aspect of *faster* relates to execution. Microservices are built on the notion that if you need to go faster, just throw more resources at it. *A manager's dream!* By building every service to be independently scalable, you allow for interaction among components to take advantage of a pool of resources instead of single component interfaces.

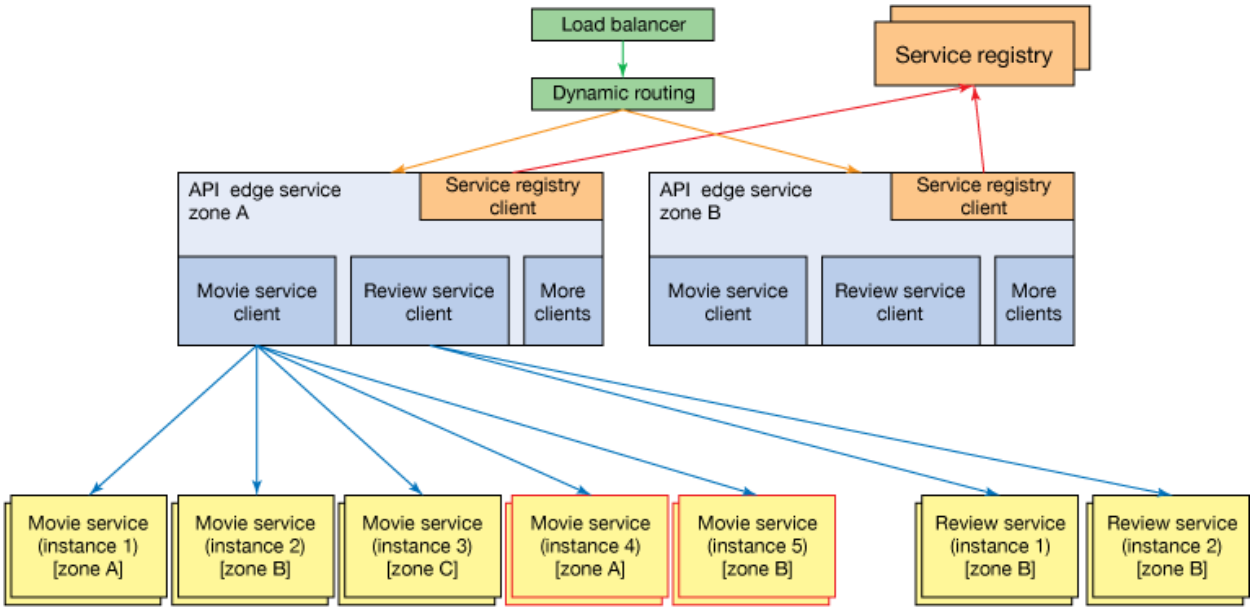
Returning to the previous example, in Figure 5 you see Service Registry servers and clients. This capability is critical in a microservices-based application. In this example, the edge services contain Movie Service and Review Service references. Based on load, these services scale at different rates; therefore you can no longer manage them all the same way at the same scale.

As the Movie Services scale, the Service Registry automatically knows about the new service instances that get created. When an Edge Service tries to handle a request, it makes a call to the

Service Registry and gets client references for all the services it depends on. The Movie Service client reference is more likely a new one that has been created fairly recently, whereas an old instance of the Review Service previously used could be returned. This Service Registry capability allows your microservices to truly function as "one of many," with a loosely coupled dependency between your components, but a highly reliable capability to get more copies of a component when needed.

Figure 7 shows the same conceptual architecture for the video-streaming application in Figure 5, with the addition of scaled out microservices for the Movie Service.

Figure 7. Conceptual scaling example of a video-streaming application



Efficiently scaling as needed, with systems that are self-aware of new instances. No, this isn't Skynet. It's what makes applications built on microservice architectures that much stronger.

Stronger

Not all systems are meant to be long-lived. They are created when needed and removed when they no longer serve a purpose. As I mentioned before, this removes single points of failure by distributing those points throughout the system, knowing that you need mechanisms to account for services and instances that are unavailable or performing poorly.

Cattle, not pets

With microservices, the notion of deployed systems becomes *cattle, not pets*:

- Pets are given names; cattle are given numbers.
- Pets are unique; cattle are often identical.
- Pets are nursed back to health; cattle are simply replaced when ill.

—*Adapted from Gavin McCance's CERN Data Centre Evolution presentation.*

This notion leads to creating services that are many of one, as well as one of many. No longer do we count service instances on one hand or manage long-term instances and worry about maintaining state, storage, system modifications, and so on. Don't get me wrong — we are still very much interested in performance tuning and configuration, but these things now happen much earlier in the development cycle instead of in staging or production. This approach gives us many services, and many instances of each service.

To validate this pillar of strength, in its journey, Netflix began harnessing chaos — a Chaos Monkey to be exact (see Figure 8). The Chaos Monkey is a cloud application component that Netflix uses to introduce systematic chaos into application operations.

Figure 8. The Chaos Monkey

This capability would go through the infrastructure and purposefully turn off services and service instances that were critical to production. Why would Netflix do this? How could it do this and survive?

First, the why. It's a way of making it easy to identify where you need to fail fast. Are new services too slow? Do they need to scale more efficiently? What happens when an external service provider goes down, not just internal services? All of these things need to be accounted for in a microservices architecture.

Second, the how. Netflix can survive thanks to the *shotgun method* I mentioned earlier. The idea is simple: Provision enough service instances that 99.9999 percent of requests will complete successfully. Any failed requests will work on a retry. Pull the plug on a service instance, and another local one will take its place. Pull the plug on an entire service, and your system should compensate or reroute users to other availability zones or regions with that specific service. If nothing else is available, the user or request should fail fast and not wait to time out.

Netflix expanded the Chaos Monkey concept and released the capability as Simian Army (see Figure 9), to include Chaos Monkeys, Janitor Monkeys, Conformity Monkeys, and Latency Monkeys — cloud application components that introduce specific chaos into operations, including latency and compliance issues

Figure 9. A Simian Army

As you can see, Netflix (and other companies adopting microservices) subscribes to the idea that what kills your application makes it stronger.

Conclusion to Part 1

The main benefits I touched on for microservices are:

- Their small size enables developers to be most productive.
- It's easy to comprehend and test each service.
- You can correctly handle failure of any dependent service.
- They reduce impact of correlated failures.

However, there are issues with adopting microservices that I haven't touched on yet. A future installment in this series will cover in depth how to manage and monitor the complexity at scale, stressing the importance of fine-grained reporting and timeliness of metrics.

The notion of more services means more contracts between teams, as the need to maintain interfaces and API versions becomes more commonplace. This is often handled well enough inside a DevOps model, whether you are delivering on a microservices-based architecture or not.

You might have noticed that I didn't touch on specific technologies used to implement microservice applications. This was intentional, because upcoming installments will cover how IBM develops and deploys its own services on microservice architectures, powering Watson (see Figure 10) and IBM Containers on Bluemix today. This includes everything from Docker to Node.js to Netflix OSS to IBM's homegrown organic capabilities.

Figure 10. IBM Watson cloud services, powered by microservices

Look for the coming articles in this series for more in-depth coverage and case studies on microservices inside and outside the IBM Cloud. Feel free to reach out directly on Twitter @rosowski or comment below to continue the discussion.

Acknowledgments and images credits

I'd like to thank Jonathan Bond for his presentation, which allowed me to capture and orient my thoughts and also borrow a few images (Figures 1, 5, 6, and 7).

External image credits: Figure 3 , Figure 4, Figure 8, Figure 9.

Resources

- Microservices: Read an in-depth overview of microservices by Martin Fowler.
- State of the Art in the Art of in Microservices: Check out Adrian Cockcroft's DockerCon 2014 keynote address.
- On SOA, microservices, and neologisms (Tony Pujals): This blog post contains excellent background on SOA, and it explains how it does and doesn't compare to microservices.
- "*Agile DevOps*: Unleash the Chaos Monkey" (Paul Duvall, developerWorks, October 2012): Read more about ensuring that your production infrastructure in a cloud environment can recover from inevitable system failures.

About the author

Rick E. Osowski

Rick Osowski is a technical product manager for IBM Cloud. With over 10 years of deep technical experience in IBM's middleware and business process management capabilities, Rick now focuses on next-generation cloud platforms, delivering hybrid cloud solutions to the company's expanding client base. He has a bachelor's degree in computer science from Pennsylvania State University and considers himself hopelessly addicted to architecting, scripting, and automating in work and everyday life. When not strategically placed in front of a computer monitor, Rick enjoys playing hockey, listening to music, and traveling.