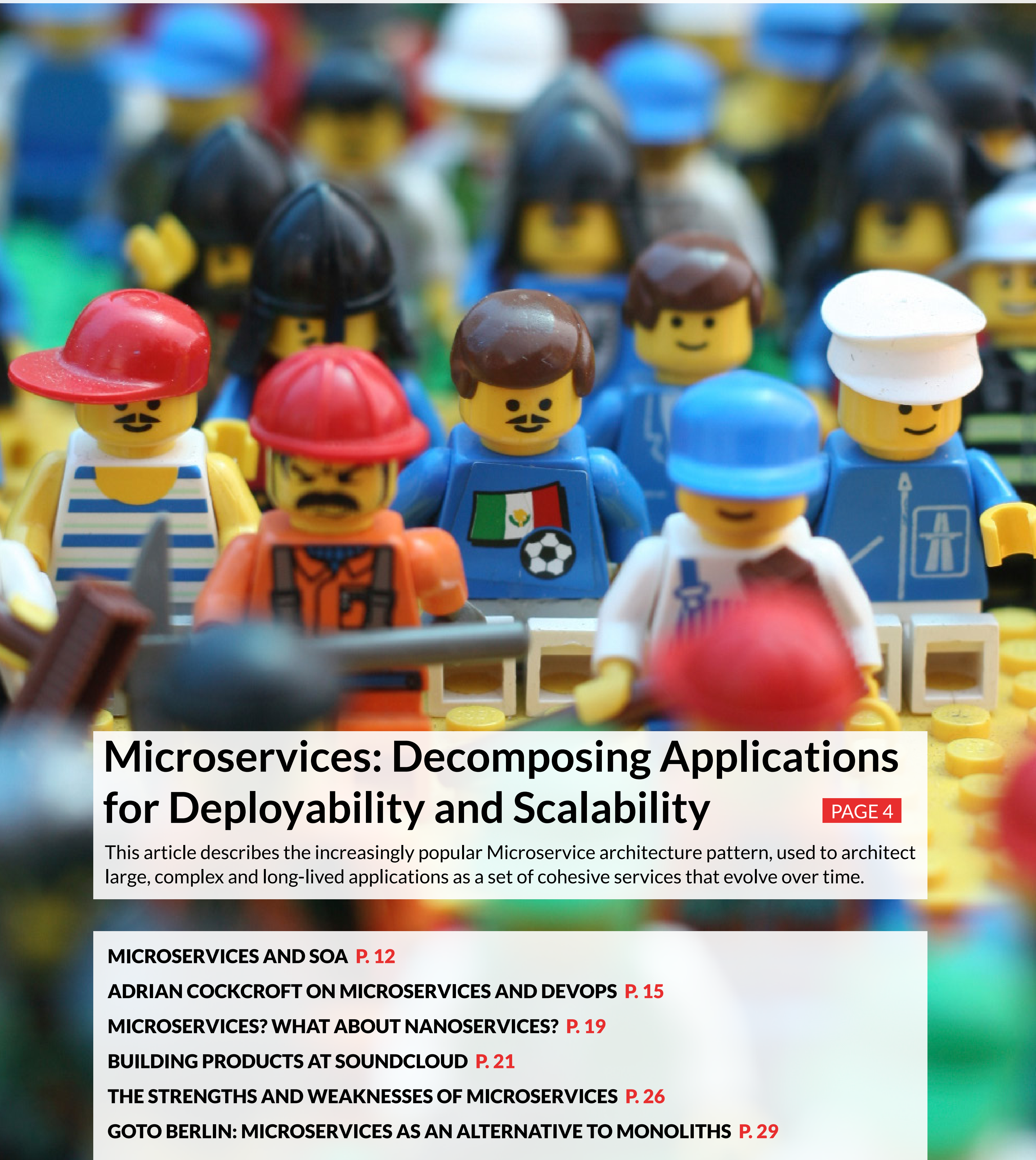
FACILITATING THE SPREAD OF KNOWLEDGE AND INNOVATION IN ENTERPRISE SOFTWARE DEVELOPMENT

Microservices

InfoQueue
eMag Issue 16 - August 2014



Microservices: Decomposing Applications for Deployability and Scalability

This article describes the increasingly popular Microservice architecture pattern, used to architect large, complex and long-lived applications as a set of cohesive services that evolve over time.

Contents

A Letter from the Editor

Harry Brumleve has been practicing software development for over fifteen years and has worked on projects ranging in size and scope from large-scale, international finance/logistics to garage-scale, social media startups. He currently works as a software architect building a high-availability graphics processing SaaS and a few of its business applications.

Microservices are more than just a passing fad. Not just an extension of Service Oriented Architecture and more than just smaller surface areas for service deployments, Microservices combine the best ideas of Agile software, DevOps, and Reactive Systems. Teams which adopt the Microservice philosophy find that they end up deploying their software more frequently and are able to satisfy the needs of their business more consistently.

However this explosion of development and deployment exacerbates the complexity required to manage these services. This complexity is often combatted with DevOps, discrete versioning, and automation, but many companies are not prepared to make this investment and may have to hire specialists to help them overcome these new challenges.

Despite this growing complexity of operational management, the march of business spurs innovation; yielding new tools, practices, and services to support the lifecycle of Microservices and the benefits they promise.

Within this eMag, we have chosen articles which highlight the pros and cons of Microservices, we offer insights from industry leaders, and hopefully we spark a few ideas for our readers so that they may leverage Microservices within their own domain.

In our first article Chris Richardson details several best practices for designing Microservices and helps define their intent: to architect large, complex and long-lived applications as a set of cohesive services that evolve over time.

Mark Little follows up with a guided discussion about the merits of Microservices in light of existing SOA practices. The discussion highlights work from James Lewis and Martin Fowler, with Steve Jones playing the foil as the SOA pragmatist.

Our third article is a summary of an interview of Netflix' Adrian Cockcroft conducted by InfoQ's Head of Editorial, Charles Humble. Adrian relates how Netflix came to embrace Microservices (they have 768 services as of July 2014) and offers some advice on how to design your own set.

After Adrian comes another article by Mark Little that shines light on another side of Microservices: the 'nanoservice anti-pattern'. This article shares some ideas offered by Arnon Rotem-Gal-Oz and offers another strong comparison to SOA and questions the need for a new term to describe that architectural pattern.

SoundCloud tells us about their journey to Microservices in our fifth article. They lay out the needs of their system and why their monolith didn't fulfill all of their needs. They encountered a few issues, but ended up with a system that had much more promise than from where they began.

In our sixth offering, Abel Avram summarizes the difficulties of moving to a Microservice architecture faced by Benjamin Wooton, the CTO of Contino. All of the issues he encountered were able to be mitigated by one means or another and Benjamin ultimately preferred his new system.

Finally James Lewis' GOTO Berlin presentation 'Microservices – adaptive architectures and organisations" rounds out our eMag by a pragmatic look at the finer points of Microservices: from the protocols commonly used, to the architectural patterns needed for their success, to their ability to discretely scale different parts of a larger system independently.

Microservices: Decomposing Applications for Deployability and Scalability

by *Chris Richardson*

This article describes the increasingly popular [microservice architecture pattern](). The big idea behind microservices is to architect large, complex, and long-lived applications as a set of cohesive services that evolve over time. The term "microservices" strongly suggests that the services should be small.

Some in the community even advocate building 10 to 100-LOC services. However, while it's desirable to have small services, that should not be the main goal. Instead, you should aim to decompose your system into services to solve the kinds of development and deployment problems discussed below. Some services might indeed be tiny where as others might be quite large.

The essence of the microservice architecture is not new. The concept of a distributed system is very old. The microservice architecture also resembles SOA.It has even been called lightweight or fine-grained SOA. And indeed, one way to think about microservice architecture is that it's SOA without the commercialization and perceived baggage of WS* and ESB.

What are the motivations for using the microservice architecture and how does it compare with the more traditional, monolithic architecture? What are its benefits and drawbacks? How do we solve some of the key technical challenges of the microservice architecture, including inter-service communication and distributed data management?

Despite not being an entirely novel idea, the microservice architecture is still worthy of discussion since it is different than traditional SOA and, more importantly, it solves many of the problems that many organizations currently suffer from.

The (sometimes evil) monolith

Since the earliest days of developing applications for the Web, the most widely used enterprise application architecture has been one that packages all the application's server-side components into a single unit. Many enterprise Java applications consist of a single WAR or EAR file. The same is true of other applications written in other languages such as Ruby and even C++.

Let's imagine, for example, that you are building an online store that takes orders from customers, verifies inventory and available credit, and ships merchandise. It's quite likely that you would build an application like the one shown in Figure 1.

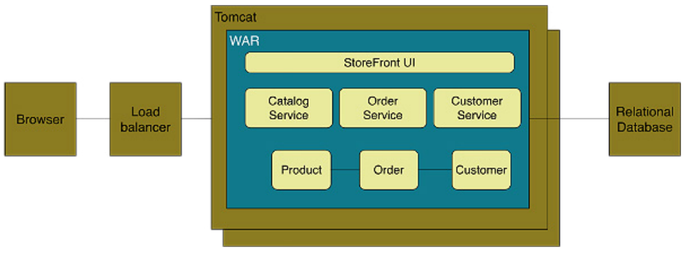


Figure 1 - The monolithic architecture.

The application consists of several components, including the StoreFront UI that implements the user interface and services for managing the product catalog, processing orders, and managing the customer's account. These services share a domain model consisting of entities such as Product, Order, and Customer.

Despite having a logically modular design, the application is deployed as a monolith. For example, if you were using Java then the application would consist of a single WAR file running on a Web container such as Tomcat. The Rails version of the application would consist of a single directory hierarchy deployed using, for example, Phusion Passenger on Apache/Nginx or JRuby on Tomcat.

This so-called monolithic architecture has a number of benefits. Monolithic applications are simple to develop since IDEs and other development tools are oriented around developing a single application. They are easy to test since you just need to launch the one application. Monolithic applications are also simple to deploy since you just have to copy the deployment unit – a file or directory – to a machine running the appropriate kind of server.

This approach works well for relatively small applications. However, the monolithic architecture becomes unwieldy for complex applications. A large monolithic application can be difficult for developers to understand and maintain. It is also an obstacle to frequent deployments. To deploy changes to one application component, you have to build and deploy the entire monolith, which can be complex, require the coordination of many developers, and result in long test cycles.

A monolithic architecture also makes it difficult to test and adopt new technologies. It's difficult, for example, to try out a new infrastructure framework without rewriting the entire application, which is risky and impractical. Consequently, you are often stuck with the technology choices that you made at the start of the project. In other words, the monolithic architecture doesn't scale to support large, long-lived applications.

Decomposing applications into services

Fortunately, there are other architectural styles that do scale. The Art of Scalability describes a useful, three-dimensional scalability model: the scale cube, which is shown in Figure 2.

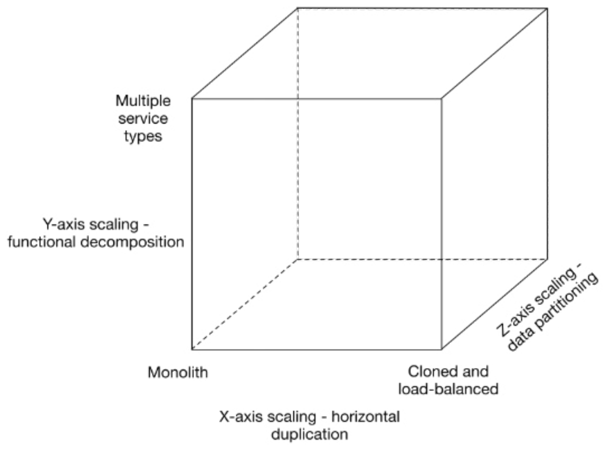

Multiple service types
Y-axis scaling - functional decomposition
Monolith
Cloned and load-balanced
X-axis scaling - horizontal duplication
Z-axis scaling - data partitioning

Figure 2 - The scale cube.

In this model, the commonly used approach of scaling an application by running multiple identical copies of the application behind a load balancer is known as X-axis scaling. That's a great way to improve the capacity and the availability of an application.

When using Z-axis scaling, each server runs an identical copy of the code. In this respect, it's similar to X-axis scaling. The big difference is that each server is responsible for only a subset of the data. Some component of the system is responsible for routing each request to the appropriate server. One commonly used routing criterion is an attribute of the request such as the primary key of the entity being accessed, i.e. sharding. Another common routing criterion is the customer type. For example, an application might provide paying customers with a higher SLA than free customers by routing their requests to a different set of servers with more capacity.

Z-axis scaling, like X-axis scaling, improves the application's capacity and availability. However, neither approach solves the problems of increasing development and application complexity. To solve those problems, we need to apply Y-axis scaling.

The third dimension to scaling is Y-axis scaling or functional decomposition. Whereas Z-axis scaling splits things that are similar, Y-axis scaling splits things that are different. At the application tier, Y-axis scaling splits a monolithic application into a set of services. Each service implements a set of related functionality such as order management, customer management, etc.

Deciding how to partition a system into a set of services is very much an art but there are number of strategies that can help. One approach is to partition services by verb or use case. For example, later on we will see that the partitioned online store has a Checkout UI service, which implements the UI for the checkout use case.

Another partitioning approach is to partition the system by nouns or resources. This kind of service is responsible for all operations that operate on entities/resources of a given type. Later, we will see how it makes sense for the online store to have a Catalog service, which manages the catalog of products.

Ideally, each service should have only a small set of responsibilities. (Uncle) Bob Martin has a PDF about designing classes using the single-responsibility principle (SRP). The SRP defines a responsibility of class as a reason to change, and that a class should only have one reason to change. It make sense to apply the SRP to service design as well.

Another analogy that helps with service design is the design of Unix utilities. Unix provides a large number of utilities such as grep, cat, and find. Each utility does exactly one thing, often exceptionally well, and can be combined with other utilities using a shell script to perform complex tasks. It makes sense to model services on Unix utilities and create single-function services.

It's important to note that the goal of decomposition is not to have tiny (e.g. 10 to 100 LOC as some argue) services simply for the sake of it. Instead, the goal is to address the problems and limitations of the monolithic architecture. Some services could very well be tiny but others will be substantially larger.

If we apply Y-axis decomposition to the example application, we get the architecture shown in Figure 3.

The decomposed application consists of various front-end services that implement different parts of the user interface and multiple back-end services. The front-end services include the Catalog UI, which implements product search and browsing, and Checkout UI, which implements the shopping cart and the checkout process. The back-end services include the same logical services that were described at the start of this article. We have turned each of

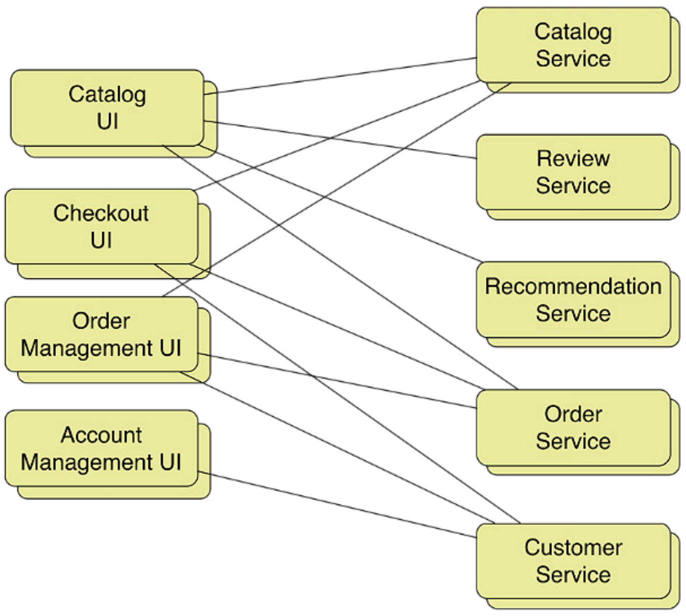


Figure 3 - The microservice architecture.

the application's main logical components into a standalone service. Let's look at the consequences of doing that.

Benefits and drawbacks of a microservice architecture

This architecture has a number of benefits. First, each microservice is relatively small. The code is easier for a developer to understand. The small code base doesn't slow down the IDE, making developers more productive. Also, each service typically starts a lot faster than a large monolith, which again makes developers more productive and speeds up deployments.

Second, each service can be deployed independently of other services. If the developers responsible for a service need to deploy a change that's local to that service, they do not need to coordinate with other developers. They can simply deploy their changes. A microservice architecture makes continuous deployment feasible.

Third, each service can be scaled independently of other services using X-axis cloning and Z-axis partitioning. Moreover, each service can be deployed on hardware that is best suited to its resource requirements. This is quite different than when using a monolithic architecture where components with wildly different resource requirements – e.g. CPU-intensive vs. memory-intensive – must be deployed together.

The microservice architecture makes it easier to scale development. You can organize the development effort around multiple, small (e.g. two-pizza) teams. Each team is solely responsible for the development and deployment of a single service or a collection of related services. Each team can develop, deploy, and scale their service independently of all other teams.

The microservice architecture also improves fault isolation. For example, a memory leak in one service only affects that service. Other services will continue to handle requests normally. In comparison, one misbehaving component of a monolithic architecture will bring down the entire system.

Last but not least, the microservice architecture eliminates any long-term commitment to a technology stack. In principle, when developing a new service, the developers are free to pick whatever language and frameworks are best suited for that service. Of course, in many organizations it makes sense to restrict the choices but the key point is that you aren't constrained by past decisions.

Moreover, because the services are small, it becomes practical to rewrite them using better languages and technologies. It also means that if the trial of a new technology fails, you can throw away that work without risking the entire project. This is quite different than when using a monolithic architecture, where your initial technology choices severely constrain your ability to use different languages and frameworks in the future.

Drawbacks

Of course, no technology is a silver bullet, and the microservice architecture has a number of significant drawbacks and issues. First, developers must deal with the additional complexity of creating a distributed system. Developers must implement an inter-process communication mechanism. Implementing use cases that span multiple services without using distributed transactions is difficult. IDEs and other development tools focus on building monolithic applications and don't provide explicit support for developing distributed applications. Writing automated tests that involve multiple services is challenging. These are all issues that you don't have to deal with in a monolithic architecture.

The microservice architecture also introduces significant operational complexity. There are many more moving parts – multiple instances of different types of service – that must be managed in production. To do this successful you need a high-level of automation, either homegrown code, a PaaS-like technology such as Netflix Asgard and related components, or an off-the-shelf PaaS such as Pivotal Cloud Foundry.

Also, deploying features that span multiple services requires careful coordination between the various development teams. You have to create a rollout plan that orders service deployments based on the dependencies between services. That's quite different than when using a monolithic architecture where you can easily deploy updates to multiple components atomically.

Another challenge with using the microservice architecture is deciding at what point during the lifecycle of the application you should use this architecture. When developing the first version of an application, you often do not have the problems that this architecture solves. Moreover, using an elaborate, distributed architecture will slow down development.

This can be a major dilemma for startups whose biggest challenge is often how to rapidly evolve the business model and accompanying application. Using Y-axis splits might make it much more difficult to iterate rapidly. Later on, however, when the challenge is how to scale and you need to use functional decomposition, tangled dependencies might make it difficult to decompose your monolithic application into a set of services.

Because of these issues, the adoption of a microservice architecture should not be undertaken lightly. However, for applications that need to scale, such as consumer-facing Web applications or SaaS applications, it is usually the right choice. Well-known sites such as eBay (PDF), Amazon.com, Groupon, and Gilt have all evolved from a monolithic architecture to a microservice architecture.

Now that we have looked at the benefits and drawbacks, let's look at a couple of key design issues within a microservice architecture, beginning with communication mechanisms within the application and between the application and its clients.

Communication mechanisms in a microservice architecture

In a microservice architecture, the patterns of communication between clients and the application, as well as between application components, are different than in a monolithic application. Let's first look at the issue of how the application's clients interact with the microservices. After that we will look at communication mechanisms within the application.

API-gateway pattern

In a monolithic architecture, clients of the application, such as Web browsers and native applications, make HTTP requests via a load balancer to one of N identical instances of the application. But in a microservice architecture, the monolith has been replaced by a collection of services. Consequently, a key question we need to answer is what do the clients interact with?

An application client, such as a native mobile application, could make RESTful HTTP requests to the individual services as shown in Figure 4.

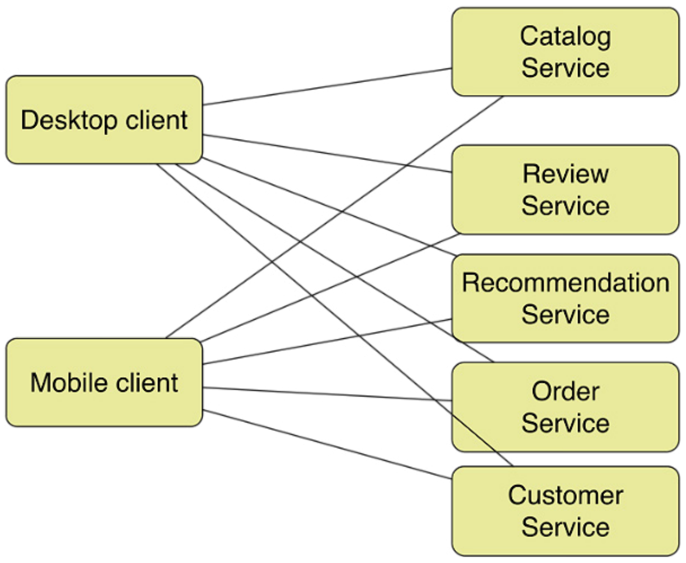


Figure 4 - Calling services directly.

On the surface, this might seem attractive. However, there is likely to be a significant mismatch in granularity between the APIs of the individual services and data required by the clients. For example, displaying one webpage could potentially require calls to large numbers of services. Amazon.com, for example, describes how some pages require calls to more than 100 services. Making that many requests, even over a high-speed Internet connection, let alone a lower-bandwidth, higher-latency mobile network, would be inefficient and result in a poor user experience.

A much better approach is for clients to make a small number of requests per page, perhaps as few as one, over the Internet to a front-end server known as an API gateway, which is shown in Figure 5.

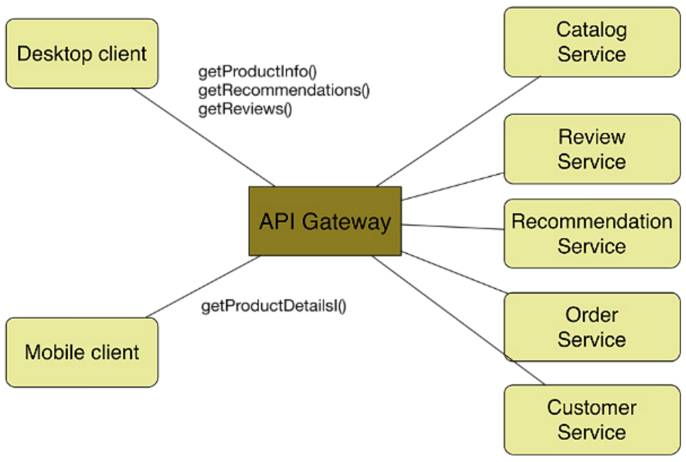


Figure 5 - API gateway.

The API gateway sits between the application's clients and the microservices. It provides APIs that are tailored to the client. The API gateway provides a coarse-grained API to mobile clients and a finer-grained API to desktop clients that use a high-performance network. In this example, the desktop clients makes multiple requests to retrieve information about a product, whereas a mobile client makes a single request.

The API gateway handles incoming requests by making requests to some number of microservices over the high-performance LAN. Netflix, for example, describes how each request fans out to on average six back-end services. In this example, fine-grained requests from a desktop client are simply proxied to the corresponding service, whereas each coarse-grained request from a mobile client is handled by aggregating the results of calling multiple services.

Not only does the API gateway optimize communication between clients and the application, but it also encapsulates the details of the microservices. This enables the microservices to evolve without impacting the clients. For example, two microservices might be merged. Another microservice might be partitioned into two or more services. Only the API gateway needs to be updated to reflect these changes. The clients remain unaffected.

Now that we have looked at how the API gateway mediates between the application and its clients, let's look at how to implement communication between microservices.

Inter-service communication mechanisms

Another major difference with the microservice architecture is how the different components of the application interact. In a monolithic application, components call one another via regular method calls. But in a microservice architecture, different services run in different processes. Consequently, services must use an inter-process communication (IPC) to communicate.

Synchronous HTTP

There are two main approaches to inter-process communication in a microservice architecture. One option is a synchronous HTTP-based mechanism such as REST or SOAP. This is a simple and familiar IPC mechanism. It's firewall-friendly so it works across the Internet, and implementing the request-reply style of communication is easy. The downside of HTTP is that it doesn't support other patterns of communication such as publish-subscribe.

Another limitation is that both the client and the server must be simultaneously available, which is not always the case since distributed systems are prone to partial failures. Also, an HTTP client needs to know the host and the port of the server. While this sounds simple, it's not entirely straightforward, especially in a cloud deployment that uses auto-scaling in which service instances are ephemeral. Applications need to use a service-discovery mechanism. Some applications use a service registry such as Apache ZooKeeper or Netflix Eureka. In other applications, services must register with a load balancer, such as an internal ELB in an Amazon VPC.

Asynchronous messaging

An alternative to synchronous HTTP is an asynchronous message-based mechanism such as an AMQP-based message broker. This approach has a number of benefits. It decouples message producers from message consumers. The message broker will buffer messages until the consumer is able to process them. Producers are completely unaware of the consumers. The producer simply talks to the message broker and does not need to use a service-discovery mechanism. Message-based communication also supports a variety of communication patterns including one-way requests and publish-subscribe. One downside of using messaging is a need for a message broker, which is yet another moving part that adds to the complexity of the system. Another downside is that request-reply style of communication is not a natural fit.

There are pros and cons to both approaches. Applications are likely to use a mixture of the two. For example, in the next section, which discusses how to solve data management problems that arise in a partitioned architecture, you will see how both HTTP and messaging are used.

Decentralized data management

A consequence of decomposing the application into services is that the database is also partitioned. To ensure loose coupling, each service has its own database (schema). Moreover, different services might use different types of database – a so-called polyglot-persistence architecture. For example, a service that needs ACID transactions might use a relational database, whereas a service that manipulates a social network might use a graph database. Partitioning the database is essential, but we now have a new problem to solve: how to handle those requests that access data owned by multiple services. Let's first look at how to handle read requests and then look at update requests.

Handling reads

Consider an online store where each customer has a credit limit. When a customer attempts to place an order, the system must verify that the sum of all open orders would not exceed the customer's credit limit. It would be trivial to implement this business rule in a monolithic application but it's much more difficult to implement this check in a system that manages customers by the CustomerService and orders by the OrderService. Somehow, the OrderService must access the credit limit maintained by the CustomerService.

One solution is for the OrderService to retrieve the credit limit by making an RPC call to the CustomerService. This approach is simple to implement and ensures that the OrderService always has the most current credit limit. The downside is that it reduces availability because the CustomerService must be running in order to place an order. It also increases response time because of the extra RPC call.

Another approach is for the OrderService to store a copy of the credit limit. This eliminates the need to make a request to the CustomerService and so improves availability and reduces response time. It does mean, however, that we must implement a mechanism to update the OrderService's copy of the credit limit whenever it changes in the CustomerService.

Handling update requests

The problem of keeping the credit limit up to date in OrderService is an example of the more general problem of handling requests that update data owned by multiple services.

Distributed transactions

One solution, of course, is to use distributed transactions. For example, when updating a customer's credit limit, the CustomerService could use a distributed transaction to update both its credit limit and the corresponding credit limit maintained by the OrderService. Using distributed transactions would ensure that the data is always consistent. The downside of using them is that it reduces system availability since all participants must be available in order for the transaction to commit. Moreover, distributed transactions have fallen out of favor and are generally not supported by modern software stacks, e.g. REST, NoSQL databases, etc.

Event-driven asynchronous updates

The other approach is to use event-driven asynchronous replication. Services publish events that announce that some data has changed. Other services subscribe to those events and update their data. For example, when the CustomerService updates a customer's credit limit, it publishes a CustomerCreditLimitUpdatedEvent, which contains the customer ID and the new credit limit. The OrderService subscribes to these events and updates its copy of the credit limit. The flow of events is shown in Figure 6.

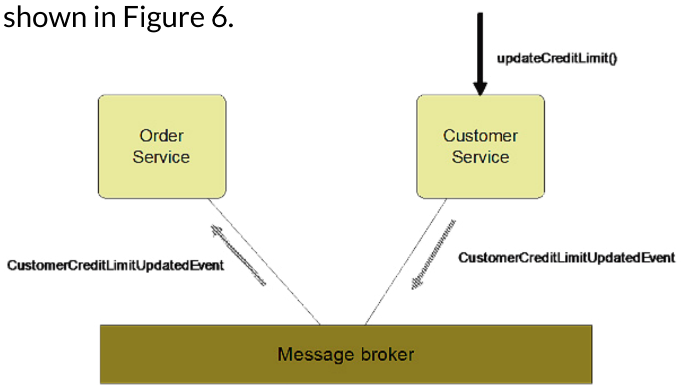

updateCreditLimit()
Order Service
Customer Service
CustomerCreditLimitUpdatedEvent
CustomerCreditLimitUpdatedEvent
Message broker

Figure 6 - Replicating the credit limit using events.

A major benefit of this approach is that it decouples producers and consumers of the events. Not only does this simplify development but compared to distributed transactions it improves availability. If a consumer isn't available to process an event then the message broker will queue the event until it can. A major drawback of this approach is that it trades consistency for availability. The application has to be written in a way that can tolerate eventually consistent data. Developers might also need to implement compensating transactions to perform logical rollbacks. Despite these drawbacks, however, this is the preferred approach for many applications.

Refactoring a monolith

Unfortunately, we don't always have the luxury of working on a brand new, green-field project. There is a pretty good chance that you are on the team that's responsible for a huge, scary monolithic application. And every day, you are dealing with the problems described at the start of this article. The good news is that there are techniques you can use to decompose your monolithic application into a set of services.

First, stop making the problem worse. Don't continue to implement significant new functionality by adding code to the monolith. Instead, you should find a way to implement new functionality as a standalone service as shown in Figure 7. This probably won't be easy. You will have to write messy, complex glue code to integrate the service with the monolith. But it's a good first step in breaking apart the monolith.

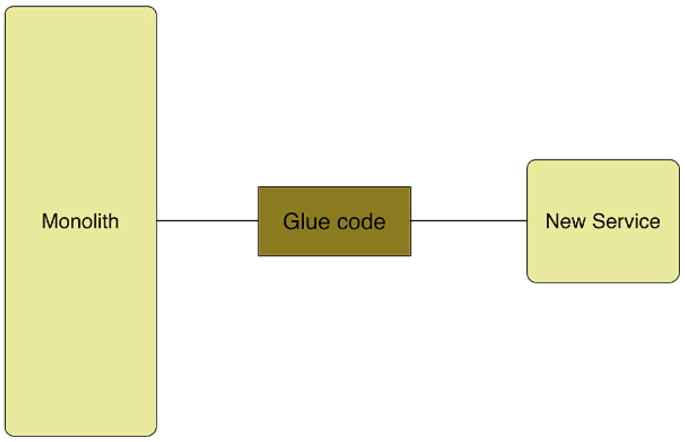


Figure 7 - Extracting a service.

Second, identify a component of the monolith to turn into a cohesive, standalone service. Good candidates for extraction include components that are constantly changing or components that have conflicting resource requirements, such as large in-memory caches or CPU-intensive operations.

The presentation tier is another good candidate. You then turn the component into a service and write glue code to integrate with the rest of the application. Once again, this will probably be painful but it enables you to incrementally migrate to a microservice architecture.

Summary

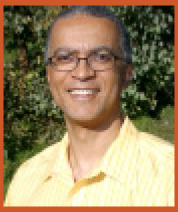
The monolithic architecture pattern is a commonly used pattern for building enterprise applications. It works reasonable well for small applications: developing, testing, and deploying small monolithic applications is relatively simple. However, for large, complex applications, the monolithic architecture becomes an obstacle to development and deployment. Continuous delivery is difficult to achieve and you are often permanently locked into your initial technology choices. For large applications, it makes more sense to use a microservice architecture that decomposes the application into a set of services.

The microservice architecture has a number of advantages. For example, individual services are easier to understand and can be developed and deployed independently of other services. It is also a lot easier to use new languages and frameworks because you can try out new technologies one service at a time.

A microservice architecture also has some significant drawbacks. In particular, applications are much more complex and have many more moving parts. You need a high level of automation, such as a PaaS, to use microservices effectively. You also need to deal with some complex distributed-data management issues when developing microservices. Despite the drawbacks, a microservice architecture makes sense for large, complex applications that are evolving rapidly, especially for SaaS-style applications.

There are various strategies for incrementally evolving an existing monolithic application to a microservice architecture. Developers should implement new functionality as a standalone service and write glue code to integrate the service with the monolith. It also makes sense to iteratively identify components to extract from the monolith and turn into services. While the evolution is not easy, it's better than trying to develop and maintain an unwieldy monolithic application.

ABOUT THE AUTHOR



Chris Richardson is a developer and architect. He is a Java Champion, a JavaOne rock star, and the author of POJOs in Action, which describes how to build enterprise Java applications with POJOs and frameworks such as Spring and Hibernate. Chris is also the founder of the original Cloud Foundry, an early Java PaaS for Amazon EC2. He consults with organizations to improve how they develop and deploy applications using technologies such as cloud computing, microservices, and NoSQL. Twitter @crichardson.

READ THIS ARTICLE ONLINE ON InfoQ

Microservices and SOA

by *Mark Little*

Over the last few years, the term "microservice" has grown up to describe an architecture composed of a suite of smaller services. At QCon San Francisco 2012, James Lewis of ThoughtWorks presented on the concept and co-authoed an article on the same topic with Martin Fowler.

Recently, Steve Jones has argued that microservices aren't new but is in fact another term for SOA. In order to do this, he compared and contrasted the current definition of microservice with the OASIS Reference Model for Service-Oriented Architecture (OASIS SOA RM). Steve followed the breakdown that James and Martin used in their article.

Componentization via services

Steve quoted the OASIS SOA RM: "Service-oriented architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains."

He pointed out:

> *OASIS then goes nicely into WTF actually is a service:*
> - *The capability to perform work for another*
> - *The specification of the work offered for another*
> - *The offer to perform work for another*

He added, from the OASIS SOA RM, "in SOA, services are the mechanism by which needs and capabilities are brought together."

Steve paraphrased what Martin and James said on the subject, which is actually as follows:

> *Services are out-of-process components who communicate with a mechanism such as a Web service request or remote procedure call.… Another consequence of using services as components is a more explicit component interface.*

Organized around business capabilities

This is where Steve believes Martin and James get it wrong and that microservices aren't new. The OASIS SOA RM defines a capability as "a real-world effect that a service provider is able to provide to a service consumer". Steve pointed out that it is important to differentiate between the capability (that which does the work) and the service (the organising construct).

> *What we found when doing SOA in the wild for over a decade, and all the people on the OASIS SOA RM had lots of experience doing this, was that the organising framework was separate from the actions themselves. The reason this is crucially important is that people started often making services where service = capability so you ended up*

with lots and lots of services (ummm, if I was being insulting I'd call them microservices).

He believes that whilst Martin's text on the subject is okay, it lacks important references to the past.

> *New? Hell, even I wrote a book which talked about how to model, manage, and set up teams around this approach. The SOA Manifesto (2009) talks about key principles behind SOA (I still prefer the RM, though) from a big group of practitioners. The point here is that there are two problems: first, the confusion of service and capabilities and secondly, the lack of recognition of hierarchies importance in governance.*

Products not projects

Steve said that the article from Martin and James went beyond the SOA RM on this topic, but again did not say anything new.

> *A quick hit on Google just shows how many pieces there are in this space, some better than others and some products better than others, but really this is not new. I used to use the phrase "Programs not projects" and always talked about assigning architects for the full lifecycle "to make them accountable". Again, it.s not that this statement is in anyway wrong, its just that it's in no way new. We've known that this has helped for years but it has a significant issue: cost.*

Smart endpoints and dumb pipes

Steve wholeheartedly agreed with this principle but again does not believe it is anything new. He would also phrase it slightly differently.

> *In the OASIS SOA RM, there is the concept of an "execution context", which is the bit that lets a service be called and its capability invoked. Clearly the endpoint is "smart" as it's what does the work, hence the phrase "mechanism" used above. The "pipes" may or may not be dumb (the "pipes" talking to those rovers on Mars are pretty smart, I think) but what they are is without value. This was a crucial finding in SOA and is well documented in the SOA RM. The execution context is where all of the internal plumbing is done but its the service that provides the access to the capability and it's the capability which delivers the value.*

Decentralized governance

This is again something on which Steve agreed with Martin and James, but he added:

> *I think it's not bad advice. It's just that SOA gives so much more than microservices in terms of governance. SOA as described in the OASIS SOA RM allows these principles to be applied to all IT assets, not just those implemented with a specific implementation style, and indeed to just IT assets, meaning it's an approach that business schools are teaching.*

Microservices and SOA

Initially, Steve felt the article from James and Martin did not reference SOA (enough or at all), but apparently as a result of recent interactions with Steve, the co-authors added a sidebar on the topic to the article. However, Steve thinks that not only was it insufficient, it's a red herring:

> *...It's not a true definition of SOA, instead rolling out the old "big" ESB and WS-* trope that was so loved by the RESTafarians when explaining why their way was better. The claim is that this article "crisply" describes the microservices style and thus is valid in comparison with SOA, as "SOA means so many things". This I fundamentally disagree with, firstly because microservices would be better served as an implementation approach if it could explain how it fits with non-microservices approaches, something that SOA does a great job of doing, and second, that it can't even say what makes a service "micro" with services ranging from decent-sized (12-person) teams down to individuals.*

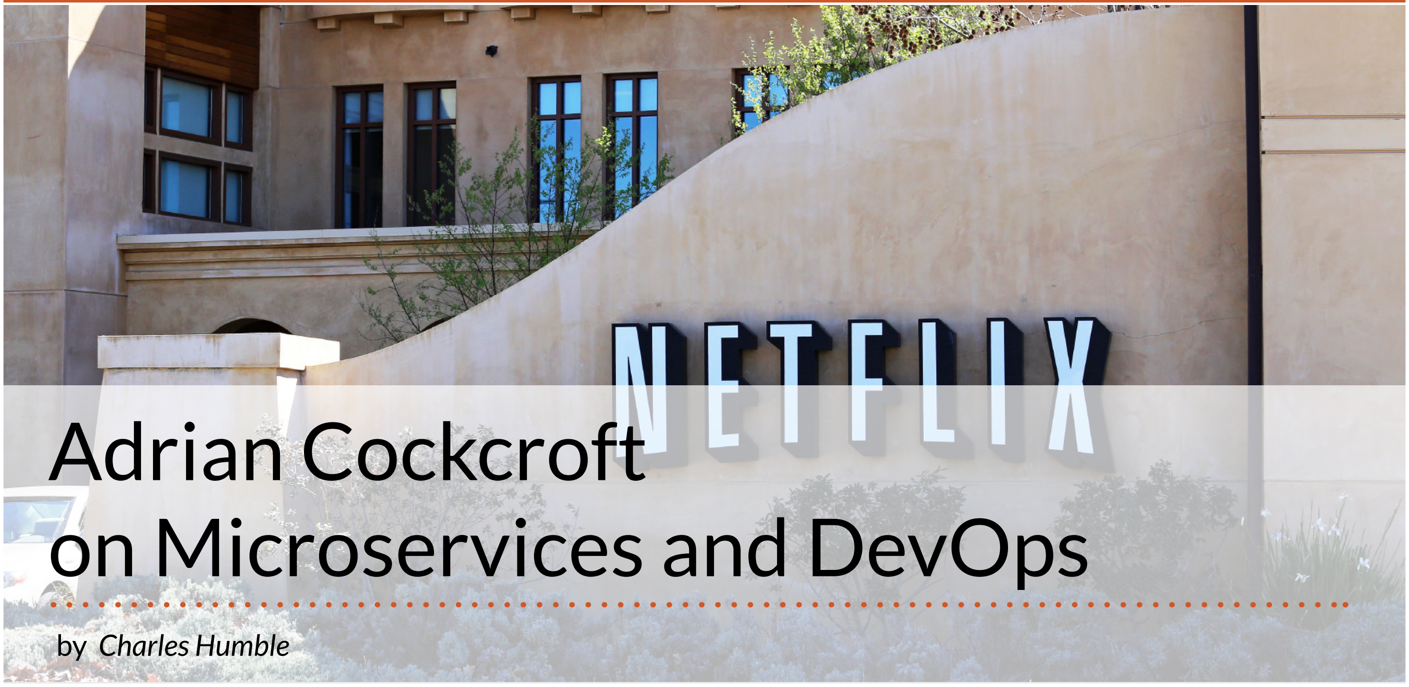
In conclusion, Steve re-iterated that in his opinion microservices is nothing new and is in fact just a service-oriented-delivery approach. In his view, an architect/developer would be better off using SOA, which is much better defined and with much more experience behind it. Instead of talking about microservices as something new, Steve thinks, we should be talking about modern SOA. He mentions:

> *Additionally, the fact that one of the references that is used is that of Netflix, who actually use the term "fine-grained SOA" as recognised in the footnotes, sort of underlines the fact and the fact that another (Amazon.com) also says it's SOA.*

ABOUT THE AUTHOR

Dr Mark Little is Vice President at Red Hat, where he leads JBoss technical direction, research and development. Prior to this he was SOA technical development manager, and director of standards. He was chief architect and co-founder at Arjuna Technologies, and Distinguished Engineer at Hewlett Packard when Arjuna was spun off. He has worked in the area of reliable distributed systems since the mid-80s. His PhD was on fault-tolerant distributed systems, replication and transactions. He is currently also a professor at Newcastle University.

READ THIS ARTICLE ONLINE ON InfoQ



Adrian Cockcroft on Microservices and DevOps

by *Charles Humble*

At QCon New York 2014, Adrian Cockcroft, formerly cloud architect at Netflix, and InfoQ head of editorial Charles Humble sat down to discuss how Netflix's culture evolved to embrace microservices and what the organization has learned from their adoption. This article summarizes the content of the interview, which may be found in its entirety on InfoQ.com.

A culture of developer responsibility: DevOps

Seven years ago, Netflix restructured its development effort to focus on agility and speed in order to outrun its larger competitors. This shift precipitated a change in the way Netflix developed and delivered services, moving from a monolithic deployment strategy, replete with its trappings of QA and ops handoffs, to a culture of DevOps where the freedom afforded to developers was matched with the responsibility of managing the entirety of their software's lifecycle.

Previous to this shift, Adrian noted, the Netflix development model reflected very common patterns in the industry at the time.

Every two weeks, the code would be given to QA for a few days and then Operations would try to make it work, and eventually they would. Every two weeks, we would do that again, go through that cycle.

As their online business grew, Adrian's team faced more challenges than just delivering functionality; their need for availability and more granular change to their code base began to grow. They still needed to be agile and found the best way to retain their speed was to decouple all their services into discrete, bounded contexts, each deployed separately.

That lead to a move to split everything up, which ended up with what's now called a microservices architecture, but at the time it was just "We have to just separate these concerns," or bounded contexts if you like, so that different teams could update their own things independently. That's how it all basically ended up.

The new strategy resulted in more granular deployments, but these deployments were congruent with the bounded contexts within their organization. Developers could manage the entire lifecycle of their services and act independently from the rest of the company.

Basically, you build your own service. You build a bounded context around the thing that your team, your two or three engineers, is building and you build a service or a group of services that interface with all the other things that your company is doing as if they were separate companies. It's a different bounded context. So you talk to them but you are not tightly coupled.

The way that the microservices stuff is different is, I think, the lack of coupling and that you are trying to build a single-verb kind of model where you've got... – the bounded context idea from domain-driven design is quite a good way of thinking about it. If you make the bounds too small then you've got too much coupling; there is not enough information within the context and you end up with very high rates of coupling outside. If you make it too big, you've got too many concepts sort of juggled, sort of a bag of things.

But this new strategy put the developer on the line for the entire lifecycle of their software. At Netflix, the finish line was no longer "I gave it to QA," or even "I gave it to Ops," or even "It's in production." The finish line became "It's no longer running in production; I've removed it." This model required developers to learn operations and manage the services they wrote and deployed.

It turns out that given that feedback loop, people write really, really resilient code, and they figure out how to make it scale. And the ones that don't will generally get woken enough times that they leave or you push them out gradually because they haven't figured it out. So you find out who's able to deal with that and it's not that hard. Another way of saying it is it's easier to teach developers to operate their code than is to teach the operators to develop code, maybe, or to look after the developers' code; it's actually a simpler problem.

This new model of developer freedom and responsibility gave developers incentive to build resilient code: to avoid wakeup calls. This fate, usually reserved for operations, was now a reality for developers and as they adapted to this new responsibility, they learned to rely on their team.

It also encourages pair programming because you don't want to be the only person that knows how the service works, because then you're always going to be on pager duty and you are always going to be called. So you want to code-review it with somebody else, typically another member of your team or several other people, and then you take turns being on call so you can go on vacation, sleep at night, and things like that.

Building a microservice

Netflix uses a multilayer strategy when deploying its microservice APIs. In order to provide flexibility and support for multiple languages, Netflix developers typically build a client layer that is responsible for calling the underlying service using a specialized and stable protocol. Multiple client libraries can be built based upon the need for a particular language's support; the service team is responsible for developing all of the necessary clients as well.

Think of having a client library that does nothing but talk to one service. It knows how to serialize and deserialize requests to that service and it knows how to handle errors from that service. You are not tied into the type system of the entire infrastructure. The idea of that is it should be a very simple function thing and that interface library should be built by the people who built the service.

So, you build a new service, you define the protocol, and you build a client library for that protocol for the different languages that you want to support – a JAR will support all the JVM languages, but you might need a Python library or you might need whatever, Go or something, Ruby, whatever you might need. And you publish those libraries in Artifactory or whatever. And so, if somebody wants to consume your service, they go to the repository and they pull out that library and build against it and call against it, and that's the most basic way to call a service.

Consumers of these client libraries can then build their own layers that are distinct to their own bounded contexts. In a similar vein to domain-driven design's anti-corruption layer, these libraries are essentially orchestration layers that allow the client team to interface with a service or a set of services in their own terms, while simultaneously shielding themselves from any upstream changes their dependencies may introduce.

Your object model is optimized for a specific use case and there might be a group of services that use your service in a certain way, and they share a higher-level library that talks to your low-level library. But that library is built by the team that consumes it, with their object model, so the binding is upwards. And then there is this interface between the two that doesn't change very often, and in which you can patch things up, because all the code is written to this higher-level library and the protocol is written to the lower-level library.

You can fudge things. You can route traffic; you can basically say, "This is only implemented in the old version of the service" and you can route it to the old library. You can have old and new and things side by side like that. So it gets maybe a little more complicated to think about, because the user behavior depends on the routing of traffic to all these different microservices. But once you figure out the power of being able to route traffic, to control behavior, and do versioning and things like that, it's quite a powerful technique.

While the layered-service practices enable flexibility and wide consumption, the protocol between the client and service layer provides the speed.

The serialization matters. Most of the latency is not on the wire because the calling between machines within the data center is sub-millisecond; that's not the problem. The problem is serializing and deserializing the request.

Through benchmarking and trial and error with many libraries such as Google's protobufs, Thrift, and Avro, Netflix established ways of communicating efficiently between services, greatly reducing the cost of serialization between their layers.

If you are doing something complicated between services that's fairly high-speed, then use one of those (libraries mentioned above). And there is a mixture of all three of those in use at Netflix - Thrift, protobufs, and Avro. Avro is the one we picked as the main recommendation because it turns out it to have good compression in it so the objects are very small. These objects tend to get stored in memcached as well - you tend to build memcached tiers between layers in the system and stuff gets collected in there. You want to store it in a very compact way because if you use half the space, you get twice as much stuff in your memcached. And itys certainly preferable compared to storing raw JSON kind of things in memcache; that would use probably five to 10 times the space. So it's worth using reasonably well-optimized protocols.

A system of microservices

Having a system composed of microservices introduces management challenges at scale. It is important to not only be able to visualize your system. If your system is significantly large and everything in your system seems to reference every other, simply diagramming your systems dependencies may seem like a good place to start. But creating a web of service calls on your whiteboard may yield more confusion than understanding.

I have a diagram in my deck that I call the Death Star Diagram for which you render all your microservices icons in a circle. The icons all overlap each other, and then you draw the connections, all the calls, between them. You get lines going from everywhere to everywhere and you just get this round blob of undecipherable mess. That's if you do it really without any hierarchy. And I've found diagrams like Gilt and Groupon and Twitter and Netflix, all have a diagram that looks very similar: it's just a blob in a circle with stuff between them.

The way that you really need to visualize this is with a lot of hierarchy. You need to group related services into a sub thing and then sort of go out so that you can see at the high level what's going on and drill in to see more and more detail. So, again, you've got layers of bounded contexts.

Netflix also introduced ways of monitoring the pattern of communications between its services. By injecting a GUID into the request at each service boundary, it has been able to effectively track all of the communication occurring within its system. It can then leverage this data to help track the health and scaling needs of particular services.

They throw a header field that's got a GUID in it, and that GUID tags the transaction and that flows down; when you do the next call you send it on down. But if you fork and make an additional one, there is a sub-transaction in there, too. So you can actually track the parent transaction and the child, and you build the tree of these GUIDS. So, an incoming request hits the website and it explodes into this tree of requests that probably hits 50 to 100 different services to gather everything that it needs to just render the homepage for a single Web request from a browser – 50 to 100 services quite commonly."

Netflix's final layer of concern for microservices is the cloud environment in which they are deployed. Due to the chaotic nature of the cloud, machines come and go and you cannot depend on the existence of any services from call to call. It's important to know what's going on in your system and monitoring is the best way to peer into the deployment.

A lot of monitoring tools assume that adding new machines is an infrequent operation and it isn't in this model, that is if you build a really dynamic system. The other problem is just scale: Netflix ended up with 10, 20, 30,000 machines, all running at once, and you are trying to monitor and make sense of that and the tools generally die in horrible ways. The monitoring system is collecting billions of metrics and it's processing huge numbers of data. It's a monstrous system just for keeping track of everything.

Getting started with microservices

Due to all of the ways for a service or data center to die, Netflix takes great care in not only monitoring the health of their system but ensuring that it is resilient enough to handle both partition and service failures. The company's patterns of development may not be unique, many coming from popular works such as Michael Nygard's book Release It!, but its code and the way it has open-sourced many of the core features of its system is a key characteristic of Netflix.

Understanding how Netflix approaches the problems of scale and microservices can help an organization to get a head start on developing a microservice architecture of its own.

ABOUT THE INTERVIEWEE

Adrian Cockcroft has had a long career working at the leading edge of technology. He's currently working for VC firm Battery, advising the firm and its portfolio companies about technology issues. Before joining Battery, Adrian helped lead Netflix's migration to a large scale, highly available public-cloud architecture and the open sourcing of the cloud-native NetflixOSS platform.

READ THIS INTERVIEW ONLINE ON InfoQ

Microservices? What About Nanoservices?

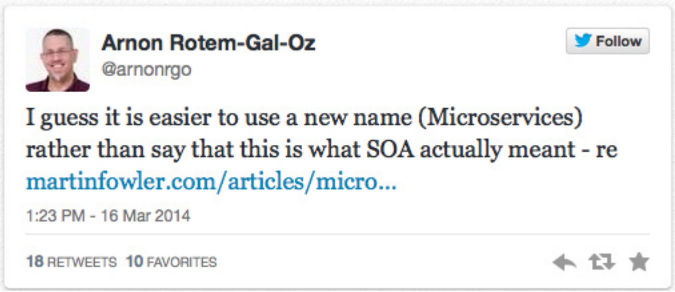
by Mark Little

Earlier this year, we posted an article about the recent rise in discussions around the term "microservices" and whether this represents a new approach to architecture or, as some like Steve Jones suggest, it is simply SOA by another name. Given the comments in the article, it seems that the majority of readers (or the majority of those who commented) believe microservices really is just SOA and a new term is probably not needed.

After that article, Steve posted another entry in which he discussed how microservices is really just service-oriented delivery and how attempts to distance it from SOA do not make sense. He wrote:

> *[...] Microservices lays down some nice rules for implementing certain parts of an enterprise but those are best served by an honesty that it's an implementation choice within a broader service-oriented architecture. That doesn't devalue it in any way, just places it in the right context.*

Arnon Rotem-Gal-Oz has also entered the debate:



Arnon Rotem-Gal-Oz
@arnonrgo
Follow

I guess it is easier to use a new name (Microservices) rather than say that this is what SOA actually meant - re martinfowler.com/articles/micro...

1:23 PM - 16 Mar 2014

18 RETWEETS 10 FAVORITES

Arnon also believes that microservices, as defined by Martin Fowler and James Lewis, is nothing more than SOA, perhaps without some of the bad misconception that meant some people associated SOA with a hard requirement on WS-* and ESBs. He asked precisely what is a microservice and quotes James Hughes (the link he uses is not valid at the time of writing) as saying the following.

> *First things first: what actually is a microservice? Well, there really isn't a hard and fast definition but from conversations with various people there seems to be a consensus that a microservice is a simple application that sits around the 10 to100-LOC mark.*

According to Arnon, James does admit later in the same article that counting lines of code is a bad way to compare service implementations (something that Jan Stenberg wrote about in his article on usage being more important than size), but Arnon wants to focus on the consensus to which James refers.

> *So how can you have 100-LOC services? You can get there if you rely on frameworks (like Finagle or Sinatra, as James mentions) to generate serialization/deserialization code (protobuf, Thrift, Avro, etc.) – this is essentially building on*

a smart service host. Another example for this would be developing in Erlang with its supervisor hierarchies, which also brings us to another way to reduce LOC by using languages that are less verbose (like the aforementioned Erlang, Python, or Scala vs. say, Java).

Arnon's point is that services with 10 to 100 lines are likely to be exposing functions rather than being a "real service". He also believes that the smaller the service gets (towards what he calls "nanoservices") the more you have to worry about management overhead, serialization/deserialization costs, security, etc. Essentially, the smaller these services become, the more glue you need to pull them together into a useful whole. As Arnon says:

Nanoservice is an anti-pattern where a service is too fine-grained. A nanoservice is a service whose overhead (communications, maintenance, and so on) outweighs its utility.

Like Steve and others, Arnon concludes that microservices is just another name for SOA. He does believe that years ago, early in the hype of SOA, perhaps another name might not have been such a bad thing, but today with the concepts behind SOA fairly well established and understood a rename is not helpful.

Furthermore, if we do want to name proper SOA by a new name, I think microservices is a poor term as it leads toward the slippery slope into nanoservices and 10 lines of code which are just your old Web-service method executed by a fancy, chic host using a hot serialization format.

Despite the fact that many people seem to agree that the term "microservice" is neither new nor needed, we are seeing a rise in frameworks that are being sold on their ability to support microservice. Perhaps this is a term that the industry will have to accept, even if the majority understands it as another name for SOA.

ABOUT THE AUTHOR

Dr Mark Little is Vice President at Red Hat, where he leads JBoss technical direction, research and development. Prior to this he was SOA technical development manager, and director of standards. He was chief architect and co-founder at Arjuna Technologies, and Distinguished Engineer at Hewlett Packard when Arjuna was spun off. He has worked in the area of reliable distributed systems since the mid-80s. His PhD was on fault-tolerant distributed systems, replication and transactions. He is currently also a professor at Newcastle University.

READ THIS ARTICLE ONLINE ON InfoQ

Building Products at SoundCloud

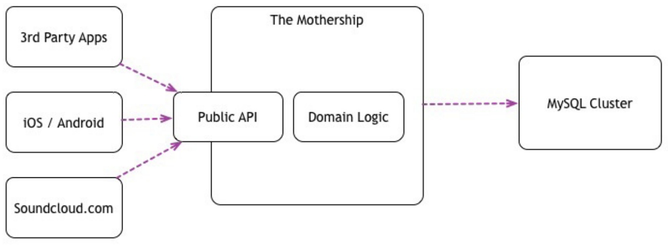
by *Phil Calçado*

Dealing with the Monolith

Most of SoundCloud's products are written in Scala, Clojure, or JRuby. This wasn't always the case. Like many other start-ups, SoundCloud was created as a single, monolithic Ruby on Rails application running on the MRI, Ruby's official interpreter, and backed by memcached and MySQL.

We affectionately call this system Mothership. Its architecture was a good solution for a new product used by several hundreds of thousands of artists to share their work, collaborate on tracks, and be discovered by the industry.

The Rails codebase contained both our Public API, used by thousands of third-party applications, and the user-facing web application. With the launch of the Next SoundCloud in 2012, our interface to the world became mostly the Public API — we built all of our client applications on top of the same API partners and developers used.



These days, we have about 12 hours of music and sound uploaded every minute, and hundreds of millions of people use the platform every day. SoundCloud combines the challenges of scaling both a very large social network with a media distribution powerhouse.

To scale our Rails application to this level, we developed, contributed to, and published several components and tools to help run database migrations at scale, be smarter about how Rails accesses databases, process a huge number of messages, and more. In the end we have decided to fundamentally change the way we build products, as we felt we were always patching the system and not resolving the fundamental scalability problem.

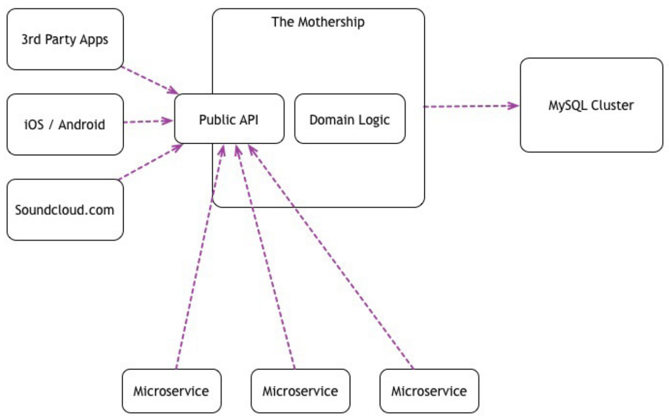
The first change was in our architecture. We decided to move towards what is now known as a microservices architecture. In this style, engineers separate domain logic into very small components. These components expose a well-defined API, and implement a Bounded Context —including its persistence layer and any other infrastructure needs.

Big-bang refactoring has bitten us in the past, so the team decided that the best approach to deal with the architecture changes would not be to split the Mothership immediately, but rather to not add anything new to it. All of our new features were built as microservices, and whenever a larger refactoring of a feature in the Mothership was required, we extract the code as part of this effort.

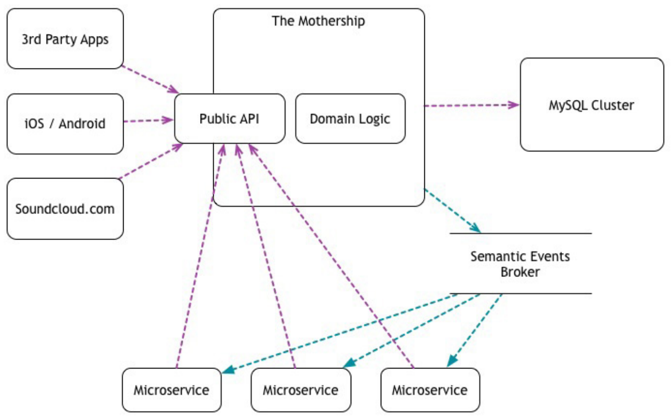
This started out very well, but soon enough we detected a problem. Because so much of our logic was still in the Rails monolith, pretty much all of our microservices had to talk to it somehow.

One option to solve this problem was to have the microservices accessing the Mothership database directly. This is a very common approach in some corporate settings, but because this database is a Public, but not Published Interface, it usually leads to many problems when we need to change the structure of shared tables.

Instead, we went for the only Published Interface we had, which was the Public API. Our internal microservices would behave exactly like the applications developed by third-party organizations that integrate with the SoundCloud platform.



Soon enough, we realized that there was a big problem with this model as our microservices needed to react to user activity. The push-notifications system, for example, needed to know whenever a track had received a new comment so that it could inform the artist about it. At our scale, polling was not an option. We needed to create a better model.



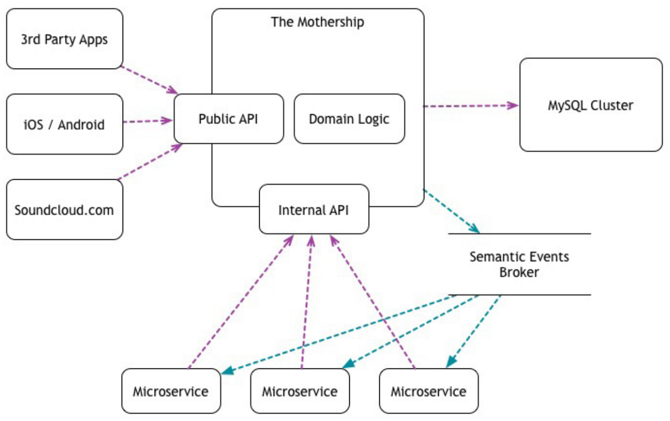
We were already using AMQP in general - the RabbitMQ implementation to be specific since in a Rails application you often need a way to dispatch slow jobs to a worker process to avoid hogging the concurrency-weak Ruby interpreter. Sebastian Ohm and Tomás Senart presented the details of how we use AMQP, but over several iterations we developed a model called Semantic Events, where changes in the domain objects result in a message being dispatched to a broker and consumed by whichever microservice finds the message interesting.

This architecture enabled Event Sourcing, which is how many of our microservices deal with shared data, but it did not remove the need to query the Public API —for example, you might need all fans of an artist and their email addresses to notify them about a new track.

While most of the data was available through the Public API, we were constrained by the same rules we enforced on third-party applications. It was not possible, for example, for a microservice to notify users about activity on private tracks as users could only access public information.

We explored several possible solutions to the problem. One of the most popular alternatives was to extract all of the ActiveRecord models from the Mothership into a Ruby gem, effectively making the Rails model classes a Published Interface and a shared component. There were several important issues with this approach, including the overhead of versioning the component across so many microservices, and it became clear that the microservices would be implemented in languages other than Ruby. Therefore, we had to think about a different solution.

In the end, the team decided to use Rails' features of engines (or plugins, depending on the framework's version) to create an Internal API that is available only within our private network. To control what could be accessed internally, we used Oauth 2.0 when an application is acting on behalf of a user, with different authorisation scopes depending on which microservice needs the data.



Although we are constantly removing features from the Mothership, having both a push and pull interface to the old system makes sure that we do not couple our new microservices to the old architecture. The microservice architecture has proven itself crucial to developing production-ready features with much shorter feedback cycles. External-facing examples are the visual sounds, and the new stats system.

https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith

Breaking the Monolith

After the architecture changes were made, our teams were free to build their new features and enhancements in a much more flexible environment. An important question remained, though: how do we extract the features from the monolithic Rails application called Mothership?

Splitting a legacy application is never easy, but luckily there are plenty of industry and academic publications to help you out.

The first step in any activity like this is to identify and apply the criteria used to define the units to be extracted. At SoundCloud, we have decided to use the work of Eric Evans and Martin Fowler in what is called a Bounded Context. An obvious example of Bounded Context in our domain was user-to-user messages. This was a well-contained feature set, highly cohesive, and not too coupled with the rest of the domain, as it just needs to hold references to users.

After we identified the Bounded Context, the next task was to find a way to extract it. Unfortunately, Rails' ActiveRecord framework often leads to a very coupled design. The code dealing with such messages was as follows:

```
   def index
    if (InboxItem === item)
      respond mailbox_items_in_collection.
  index.paginate(:page => params[:page])
    else
      respond mailbox_items_in_collection.
  paginate(
      :joins => «INNER JOIN messages ON
#{safe_collection}_items.message_id =
messages.id»,
      :page  => params[:page],
      :order => 'messages.created_at
DESC')
  end
end
```

Because we wanted to extract the messages' Bounded Context into a separate microservice, we needed the code above to be more flexible. The first step we took was to refactor this code into what Michael Feathers describes as a seam:

A seam is a place where you can alter behavior in your program without editing in that place.

So we changed our code a little bit:

```
def index
  conversations = cursor_for do |cursor|
    conversations_service.conversations_
for(
    current_user,
    cursor[:offset],
    cursor[:limit])
  end

  respond collection_for(conversations,
:conversations)
end
```

The first version of the conversations_service#conversations_for method was not that different from the previous code; it performed the exact same ActiveRecord calls.

We were ready to extract this logic into a microservice without having to refactor lots of controllers and other Presentation Layer code. We first replaced the implementation of conversations_service#conversations_for with a call to the service:

```
def conversations_for(user, offset = 0,
limit = 50)
  response = @http_client.
do_get(service_path(user),
pagination(offset, limit))
  parse_response(user, response)
end
```

We avoided big-bang refactorings as much as we could, and this required us to have the microservices working together with the old Mothership code for as long as it took to completely extract the logic into the new microservice.

As described before, we did not want to use the Mothership's database as the integration point for microservices. That database is an Application Database, and making it an Integration Database would cause problems because we would have to synchronize any change in the database across many different microservices that would now be coupled to it.

Although using the database as the integration point between systems was not planned, we had the new microservices accessing the Mothership's database during the transition period.

This brought up two important issues. During the whole transition period, the new microservices could not change the relational model in MySQL—or, even worse, use a different storage engine. For extreme cases, like user-to-user messages where a threaded-based model was replaced by a chat-like one, we had cronjobs keep different databases synchronized.

The other issue was related to the Semantic Events system described in Part I. The way our architecture and infrastructure was designed requires events to be emitted when a state change has happened, and this ought to be a single system. Because we could not have both the Mothership and the new microservice emitting events, we had to implement only the read-path endpoints until we were ready to make the full switch from the Mothership to the new microservice. This was less problematic than what we first thought, but nevertheless it did impact product prioritization because features to be delivered were constrained by this strategy.

By applying these principles we were able to extract most services from the Mothership. Currently we have only the most coupled part of our domain there, and products like the new user-to-user messaging system were built completely decoupled from the monolith.`

Microservices in Scala and Finagle

In the first two parts of this article, we talked about how SoundCloud started breaking away from a monolithic Ruby on Rails application into a microservices architecture. In this part we will talk a bit more about the platforms and languages in which we tend to write these microservices.

At the same time that we started the process of building systems outside the Mothership (our Rails monolith) we started breaking our large team of engineers into smaller teams that focused on one specific area of our platform.

It was a phase of high experimentation, and instead of defining which languages or runtimes these teams should use, we had the rule of thumb to write it in whatever you feel confident enough putting in production and being on-call for.

This led to a Cambrian Explosion of languages, runtimes and skills. We had systems being developed in everything from Perl to Julia, including Haskell, Erlang, and node.js.

While this process proved quite productive in creating new systems, we started having problems when maintaining them. The bus factor for several of our systems was very low, and we eventually decided to consolidate our tools.

Based on the expertise and preferences across teams, and an assessment of the industry and our peers, we decided to stick to the JVM and select JRuby, Clojure, and Scala as our company-wide supported languages for product development. For infrastructure and tooling, we also support Go and Ruby.

It turns out that selecting the runtime and language is just one step in building products in a microservices architecture. Another important aspect an organization has to think about is what stack to use for things like RPC, resilience, and concurrency.

After some research and prototyping, we ended up with three alternatives: a pure Netty implementation, the Netflix stack, and the Finagle stack.

Using pure Netty was tempting at first. The framework is well documented and maintained, and the support for HTTP, our main protocol for RPC, is good. After a while though, we found ourselves implementing abstractions on top of it to do basic things for the concurrency and resilience requirements of our systems. If such abstractions were to be required, we would rather use something that exists than re-invent the wheel.

We tried the Netflix stack, and a while back Joseph Wilk wrote about our experience with Hystrix and Clojure. Hystrix does very well in the resilience and concurrency requirements, but its API based on the Command pattern was a turnoff. In our experience, Hystrix commands do not compose very well unless you also use RxJava, and although we use this library for several back-end systems and our Android application, we decided that the reactive approach was not the best for all of our use cases.

We then started trying out Finagle, a protocol-agnostic RPC system developed by Twitter and used by many companies our size. Finagle does very well in our three requirements, and its design is based on a familiar and extensible Pipes-and-Filters meets Futures model.

The first issue we found with Finagle is that, as opposed to the other alternatives, it is written in Scala, therefore the language runtime jar file is required even for a Clojure or JRuby application. We decided that this wasn't too important, though it adds about 5MB to the transitive dependencies' footprint, the language runtime is very stable and does not change often.

The other big issue was to adapt the framework to our conventions. Twitter uses mostly Thrift for RPC; we use HTTP. They use ZooKeeper for Service Discovery; we use DNS. They use a Java properties-based configuration system; we use environment variables. They have their own telemetry system; we have our own telemetry system (we're not ready to show it just yet, but stay tuned for some exciting news there). Fortunately, Finagle has some very nice abstractions for these areas, and most of the issues were solved with very minimal changes and there was no need to patch the framework itself.

We then had to deal with the very messy state of Futures in Scala. Heather Miller, from the Scala core team, explained the history and changes introduced by newer versions of the language in a great presentation. But in summary, what we have across the Scala ecosystem are several different implementations of Futures and Promises, with Finagle coupled to Twitter›s Futures. Although Scala allows for compatibility between these implementations, we decided to use Twitter's everywhere, and invest time in helping the Finagle community move closer to the most recent versions of Scala rather than debug weird issues that this interoperability might spawn.

With these issues addressed, we focused on how best to develop applications using Finagle. Luckly, Finagle's design philosophy is nicely described by Marius Eriksen, one of its core contributors, in his paper Your Server as a Function. You don't need to follow these principles in your userland code, but in our experience everything integrates much better if you do. Using a Functional programming language like Scala makes following these principles quite easy, as they map very well to pure functions and combinators.

We have used Finagle for HTTP, Thrift, memcached, Redis, and MySQL. Every request to the SoundCloud platform is very likely hitting at least one of our Finagle-powered microservices, and the performance we have from these is quite amazing.

This article was originally published as a series of blog posts on the Soundcloud developers' blog. You can read them here:

https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith

https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-2-breaking-the-monolith

https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-3-microservices-in-scala-and-finagle

The Strengths and Weaknesses of Microservices

by *Abel Avram*

There has been significant buzz around microservices lately, enough to generate some hype. After implementing heavy and cumbersome SOA solutions for more than a decade, are microservices the solution the industry has been waiting for? Are microservices simpler than monolithic solutions?

In their article "Microservices", James Lewis and Martin Fowler define the microservices architectural style.

> *In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.*

Some of the benefits of microservices are pretty obvious:

- Each microservice is quite simple, being focused on one business capability
- Microservices can be developed independently by different teams
- Microservices are loosely coupled
- Microservices can be developed using different programming languages and tools

Those benefits make it look like microservices are the perfect solution, but aren't there drawbacks?

Benjamin Wootton, CTO of Contino, is currently architecting a system based on microservices and has encountered a number of difficulties he detailed in an article called "Microservices - Not a Free Lunch!" Here is a digest of those.

Major operations overhead

Twenty services can become 40 to 60 processes after failover and resilience are introduced into the equation. The number of processes grows when load balancing and messaging middleware are added. Operating and orchestrating all these services can be challenging, according to Wootton.

> *Productionising all of this needs high-quality monitoring and operations infrastructure. Keeping an application server running can be a full-time job, but we now have to ensure that tens or even hundreds of processes stay up, don't run out of disk space, don't deadlock, stay performant. It's a daunting task.*

The operations processes need to be automated, but because "there is not much in terms of frameworks and open-source tooling to support this..., a team rolling out microservices will need to a make significant investment in custom scripting or development to manage these processes before they write a line of code that delivers business value."

DevOps is a must

Wootton believes that an organization implementing microservices needs DevOps.

> *You simply can't throw applications built in this style over the wall to an operations team. The development team need to be very operationally focused and production aware, as a microservices-based application is very tightly integrated into its environmental context.*

Since many of the services will probably need their own data stores, the developers will also need to have a "good understanding of how to deploy, run, optimize, and support a handful of NoSQL products."

Interfaces mismatch

Services rely on the interfaces between them to communicate. Changing one service's interface implies changing other services, Wootton observed.

> *Change syntax or semantics on one side of the contract and all other services need to understand that change. In a microservices environment, this might mean that simple cross-cutting changes end up requiring changes to many different components, all needing to be released in co-ordinated ways.*
>
> *Sure, we can avoid some of these changes with backwards-compatibility approaches, but you often find that business-driven requirements prohibit staged releases anyway....*
>
> *If we let collaborating services move ahead and become out of sync, perhaps in a canary-releasing style, the effects of changing message formats can become very hard to visualize.*

Code duplication

To avoid introducing "synchronous coupling into the system", Wootton believes that sometimes it's useful to add certain code to different services, leading to code duplication which is a "bad idea as every instance of the code will need to be tested and maintained going forward." A solution would be to use a shared library between services but "it won't always work in a polyglot environment and introduces coupling which may mean that services have to be released in parallel to maintain the implicit interface between them."

Complexity of a distributed system

As a distributed system, microservices introduce a level of complexity and several issues to take care of, such as "network latency, fault tolerance, message serialization, unreliable networks, asynchronicity, versioning, varying loads within our application tiers, etc."

Asynchronicity

Wootton considers that microservices usually make use of asynchronous programming, messaging, and parallelism, which can be complex when "things have to happen synchronously or transactionally", requiring management of "correlation IDs and distributed transactions to tie various actions together."

Testing

Testing is another issue to consider when doing a microservices architecture since it may be "difficult to recreate environments in a consistent way for either manual or automated testing," wrote Wootton.

> *When we add in asynchronicity and dynamic message loads, it becomes much harder to test systems built in this style and gain confidence in the set of services that we are about to release into production.*

We can test the individual service, but in this dynamic environment, very subtle behaviors can emerge from the interactions of the services which are hard to visualize and speculate on, let alone comprehensively test for.

Brady, a reader who commented on Wootton's article, added his experience in attempting to transition from a monolithic app to microservices.

> *I was working on a project moving towards microservices from a monolithic application and we ran into a lot of the hurdles mentioned in your post. We ended up having a lot of code duplication (since the services were built on different languages and frameworks) - lots of little "implicit contracts", for example, mapping user data from one service to another (one service not necessarily needing all the same data as another). While there are some clear benefits of the approach, (it's) probably not*

something you want to jump into without some careful planning. Our approach in the end was to modularize the monolithic application (so we can share code repository, deployments, and code between modules - but still have nice, loosely coupled components) and pull out the modules into their own independent micro-services that can be deployed/managed independently only if really necessary.

Dennis Ehle, another reader, shared his experience with microservices, concluding that indeed microservices come with a cost.

> *We're currently implementing CD pipeline-automation framework for a client that has over 450 developers working across 50 services (or microservices). To me, one the most fascinating aspects of this architecture is none of those 450 developers will ever write a single line of code to support a customer-facing user interface. All customer-facing UX work is performed by a different group entirely.*
>
> *While the level (of) overall flexibility, risk reduction, and cost savings this client currently enjoys is significant and a direct result of moving away from a monolithic architecture, there is no doubt a very real "microservice tax" paid due to many of the factors you very articulately outline in your post!*

On the other hand, Steve Willcox, yet another reader, sees opportunities in challenges introduced by microservices.

> *Being one of the tech leads on transforming a monolithic Java application to a SOA implementation, I've come across everyone of the issues you raise but instead of seeing those as problems I see them as opportunities to build software better.…*
>
> *You say "Substantial DevOps Skills Required". I see that as a good thing. It gives the people writing the code the responsibility of how it runs in production. Going to a SOA implementation almost forces you to a de-centralized DevOps where the service team developers do the DevOps as compared to the old school "throwing it over the wall" to the centralized operations team. It's a big positive to have the dev team be responsible for the operations of their code.…*
>
> *Yes, there are more services compared to a monolithic application to build, test, and deploy but in today's world those things should all be automated. Having two vs. 20 that follow the same automated patterns should not be that much more work.*

Regarding code duplication, Willcox said that it may not be that bad.

> *I used to be a (purist) in this area in that any and all code duplication is bad but then realized this purity sometimes resulted in much more costly and complex solutions and no one won but idealism. I'm now more practical in this area and simplicity has to be a part of the equation as well. I really like what Richard Gabriel wrote in The Rise of Worse Is Better about this.*

In conclusion, we could say that the microservices architecture has a number of appealing benefits, but one must be aware of its challenges before embarking on such a journey.

READ THIS ARTICLE ONLINE ON InfoQ

GOTO Berlin: Microservices As an Alternative to Monoliths

by *Jan Stenberg*

James Lewis, a principle consultant at ThoughtWorks, considered whether we are building systems that are too big, bigger than they need to be, in his presentation at the GOTO Berlin Conference titled "Microservices – adaptive architectures and organisations".

James stated that up to 90% of the total cost of ownership incurs after the launch of a system. He offered lots of examples of projects where we as an industry have spent a huge amount of money on building large, complex, and ultimately unsuccessful systems.

James's experiences at large organisations that build systems in the traditional way, putting all functionality in one big application with one big database, and the problems this creates, has led him toward building systems where entirely separate business capabilities are kept separate in smaller parts, as microservices, each with its own database.

His first step on this path was the idea of hexagonal business capabilities, which refers to Alistair Cockburn's hexagonal architecture. Each single business capability or functionality, together with its own data, forms a hexagon – a bounded context using DDD terms. All these hexagons are then put together in a larger hexagon, eventually forming a system.

James's next step concerns a uniform interface for all services. A common integration technology when building system in isolation has been integration with direct database access. Problems with this are that it tightly couples different parts of a system, logic and data gets easily scattered all over the system, and it becomes difficult to predict effect of changes. James prefers to use the ways of integration that the Web so successfully uses, based on HTTP, HTML, and hypermedia, and therefore uses REST for communication. Two standard application protocols he finds useful together with REST are Atom and AtomPub.

James believes that all these small services should obey the single-responsibility principle and that the principle should apply at every level of abstraction, from objects to subsystems' business capabilities and all the way up to the system.

In his final step, James looked into scalability. Building a single large system with all functionality makes it hard or impossible to scale different parts of the system. Even though some parts have a high load and others a much lower load, they must still run with the same capacity. With small services, they can be deployed on different servers and use more or fewer servers as required. Another advantage is that different services can be implemented on different platforms. A vital factor when using many small services is to automate monitoring and deployment, for example with programmable infrastructure. Advances in virtualization and infrastructure-as-a-service (IaaS) providers during the last couple of years make this possible.

The GOTO Berlin Conference 2013 was the first GOTO conference in Berlin, with over 400 attendees and about 80 speakers.

READ THIS ARTICLE ONLINE ON InfoQ