# Pattern: Microservices Architecture

## **Context**

You are developing a server-side enterprise application. It must support a variety of different clients including desktop browsers, mobile browsers and native mobile applications. The application might also expose an API for 3rd parties to consume. It might also integrate with other applications via either web services or a message broker. The application handles requests (HTTP requests and messages) by executing business logic; accessing a database; exchanging messages with other systems; and returning a HTML/JSON/XML response.

The application has either a layered or [hexagonal](http://alistair.cockburn.us/Hexagonal+architecture) architecture and consists of different types of components:

* Presentation components - responsible for handling HTTP requests and responding with either HTML or JSON/XML (for web services APIS)
* Business logic - the application’s business logic
* Database access logic - data access objects responsible for access the database
* Application integration logic - messaging layer, e.g. based on Spring integration.

There are logical components corresponding to different functional areas of the application.

## **Problem**

What's the application's deployment architecture?

## **Forces**

* There is a team of developers working on the application
* New team members must quickly become productive
* The application must be easy to understand and modify
* You want to practice continuous deployment of the application
* You must run multiple copies of the application on multiple machines in order to satisfy scalability and availability requirements
* You want to take advantage of emerging technologies (frameworks, programming languages, etc)

## **Solution**

Architect the application by applying the [Scale Cube](http://microservices.io/articles/scalecube.html) (specifically y-axis scaling) and functionally decompose the application into a set of collaborating services. Each service implements a set of narrowly, related functions. For example, an application might consist of services such as the order management service, the customer management service etc.

Service communicate using either synchronous protocols such as HTTP/REST or asynchronous protocols such as AMQP.

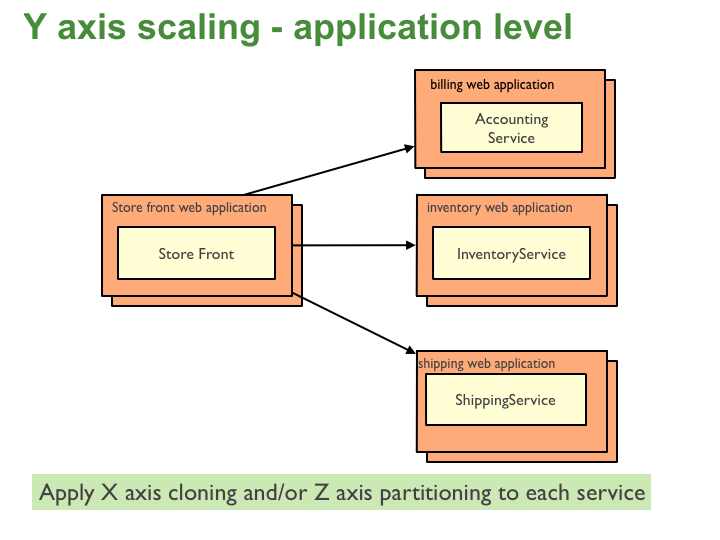
Services are developed and deployed independently of one another.

Each service has its own database in order to be decoupled from other services. When necessary, consistency is between databases is maintained using either database replication mechanisms or application-level events.

## **Example**

Let’s imagine that you are building an e-commerce application that takes orders from customers, verifies inventory and available credit, and ships them. The application consists of several components including the StoreFrontUI, which implements the user interface, along with some backend services for checking credit, maintaining inventory and shipping orders.

The application is deployed as a set of services.



## **Resulting context**

This solution has a number of benefits:

* Each microservice is relatively small
  + Easier for a developer to understand
  + The IDE is faster making developers more productive
  + The web container starts faster, which makes developers more productive, and speeds up deployments
* Each service can be deployed independently of other services - easier to deploy new versions of services frequently
* Easier to scale development. It enables you to organize the development effort around multiple teams. Each (two pizza) team is responsible a single service. Each team can develop, deploy and scale their service independently of all of the other teams.
* Improved fault isolation. For example, if there is a memory leak in one service then only that service will be affected. The other services will continue to handle requests. In comparison, one misbehaving component of a monolithic architecture can bring down the entire system.
* Each service can be developed and deployed independently
* Eliminates any long-term commitment to a technology stack

This solution has a number of drawbacks:

* Developers must deal with the additional complexity of creating a distributed system.
  + Developer tools/IDEs are oriented on building monolithic applications and don't provide explicit support for developing distributed applications.
  + Testing is more difficult
  + Developers must implement the inter-service communication mechanism.
  + Implementing use cases that span multiple services without using distributed transactions is difficult
  + Implementing use cases that span multiple services requires careful coordination between the teams
* Deployment complexity In production, there is also the operational complexity of deploying and managing a system comprised of many different service types.
* Increased memory consumption The microservices architecture replaces N monolithic application instances with NxM services instances. If each service runs in its own JVM (or equivalent), which is usually necessary to isolate the instances, then there is the overhead of M times as many JVM runtimes. Moreover, if each service runs on its own VM (e.g. EC2 instance), as is the case at Netflix, the overhead is even higher.

One challenge with using this approach is deciding when it makes sense to use it. When developing the first version of an application, you often do not have the problems that this approach solves. Moreover, using an elaborate, distributed architecture will slow down development. This can be a major problem for startups whose biggest challenge is often how to rapidly evolve the business model and accompanying application. Using Y-axis is 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 then tangled dependencies might make it difficult to decompose your monolithic application into a set of services.

Another challenge is deciding how to partition the system into microservices. This 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 you will see that the partitioned e-commerce application has a Shipping service that’s responsible for shipping complete orders. Another common example of partitioning by verb is a login service that implements the login 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. For example, later on you will see how it makes sense for the e-commerce system to have an Inventory service that keeps tracks whether products are in stock.

Ideally, each service should have only a small set of responsibilities. (Uncle) Bob Martin talks about designing classes using the [Single Responsible Principle (SRP)](http://www.objectmentor.com/resources/articles/srp.pdf). 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.

## **Related patterns**

The [Monolithic architecture](http://microservices.io/patterns/monolithic.html) is an alternative pattern.

## **Known uses**

Most large scale web sites including [Netflix](http://techblog.netflix.com/), [Amazon](http://highscalability.com/blog/2007/9/18/amazon-architecture.html) and [eBay](http://www.addsimplicity.com/downloads/eBaySDForum2006-11-29.pdf) have evolved from a monolithic architecture to a microservices architecture.

Netflix , which is a very popular video streaming service that’s responsible for up to 30% of internet traffic, has a large scale, service-oriented architecture. They handle over a billion calls per day to their video streaming API from over 800 different kinds of devices. Each API call fans out to an average of six calls to backend services.

Amazon.com originally had a two-tier architecture. In order to scale they migrated to a service-oriented architecture consisting of hundreds of backend services. Several applications call these services including the applications that implement the Amazon.com website and the web service API. The Amazon.com website application calls 100-150 services to get the data that used to build a web page.

The auction site ebay.com also evolved from a monolithic architecture to a service-oriented architecture. The application tier consists of multiple independent applications. Each application implements the business logic for a specific function area such as buying or selling. Each application uses X-axis splits and some applications such as search use Z-axis splits. Ebay.com also applies a combination of X-, Y- and Z-style scaling to the database tier.

Microservices: Decomposing Applications for Deployability and Scalability

This article describes the increasingly popular [Microservice architecture pattern](http://microservices.io/patterns/microservices.html). 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 **micro**services strongly suggests that the services should be small.

Some in the community even advocate building 10-100 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.

**Related Vendor Content**

### [JPA Repositories Chapter; Spring Data eBook Giveaway](http://www.infoq.com/infoq/url.action?i=269&t=c)

### [Tutorial: Building a RESTful Web Service with Spring](http://www.infoq.com/infoq/url.action?i=270&t=c)

### [Get Started with the Spring IO Platform](http://www.infoq.com/infoq/url.action?i=271&t=c)

### [Tutorial: Data Access with Spring](http://www.infoq.com/infoq/url.action?i=272&t=c)

### [Spring Integration 4.0 - The New Frontier](http://www.infoq.com/infoq/url.action?i=273&t=c)

**Related Sponsor**

[http://www.infoq.com/resource/sponsorship/featuredcontent/articles/microservices-intro/en/VCRBox_Pivotal_05252014.png](http://www.infoq.com/infoq/url.action?i=274&t=c)

**Spring** helps development teams everywhere build simple, portable, fast and flexible JVM-based systems and applications.

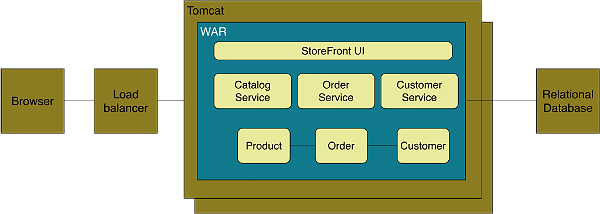
It has even been called lightweight or fine-grained SOA. And indeed, one way to think about microservice architecture is that it’s SOA without the commercialization and perceived baggage of WS\* and ESB. Despite not being an entirely novel idea, the microservice architecture is still worthy of discussion since it is different than traditional SOA and, more importantly, it solves many of the problems that many organizations currently suffer from.

In this article, you will learn about the motivations for using the microservice architecture and how it compares with the more traditional, [monolithic architecture](http://microservices.io/patterns/monolithic.html). We discuss the benefits and drawbacks of microservices. You will learn how to solve some of the key technical challenges with using the microservice architecture including inter-service communication and distributed data management.

## **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](https://engineering.groupon.com/2013/misc/i-tier-dismantling-the-monoliths/) and even[C++](http://highscalability.com/amazon-architecture).

Let’s imagine, for example, that you are building an online store that takes orders from customers, verifies inventory and available credit, and ships them. 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, which implements the user interface, along with 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 either, for example, Phusion Passenger on Apache/Nginx or JRuby on Tomcat.

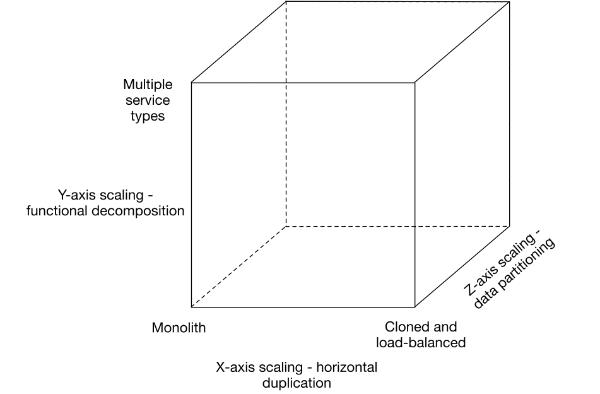
This so-called monolithic architecture has a number of benefits. Monolithic applications are simple to develop since IDEs and other development tools are oriented around developing a single application. They are easy to test since you just need to launch the one application. Monolithic applications are 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, risky, time consuming, require the coordination of many developers and result in long test cycles.

A monolithic architecture also makes it difficult to trial 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 book, [*The Art of Scalability*](http://theartofscalability.com/)*,*describes a really useful, three dimension scalability model: [the scale cube](http://akfpartners.com/techblog/2008/05/08/splitting-applications-or-services-for-scale/), which is shown in Figure 2.



**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 of improving 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 criteria is an attribute of the request such as the primary key of the entity being accessed, i.e. sharding. Another common routing criteria 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 3rd dimension to scaling is Y-axis scaling or functional decomposition. Where as 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 you will see that the partitioned online store has a Checkout UI service, which implements the UI for the checkout use case.

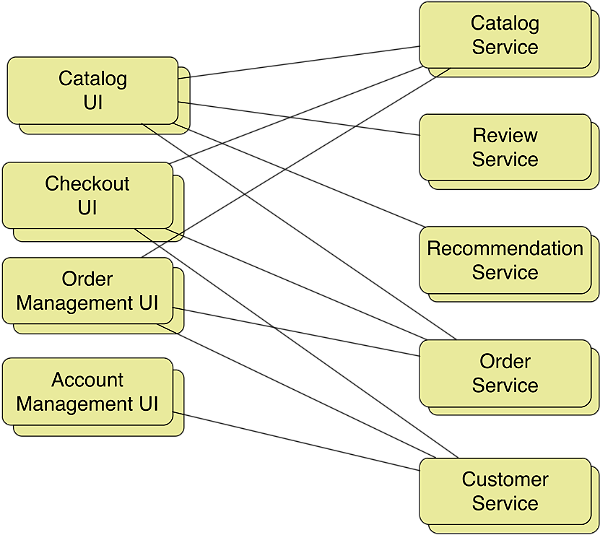
Another partitioning approach is to partition the system by nouns or resources. This kind of service is responsible for all operations that operate on entities/resources of a given type. For example, later on you will see how it makes sense for the online store to have a Catalog service, which manages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

Ideally, each service should have only a small set of responsibilities. (Uncle) Bob Martin[talks[PDF] about designing classes using the Single Responsible Principle](http://www.objectmentor.com/resources/articles/srp.pdf)(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. [10-100 LOC as some argue](http://yobriefca.se/blog/2013/04/28/micro-service-architecture/)) services simply for the sake of it. Instead, the goal is to address the problems and limitations of the monolithic architecture described above. 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.



**Figure 3 - the microservice architecture**

The decomposed application consists of various frontend services that implement different parts of the user interface and multiple backend services. The front-services include the Catalog UI, which implements product search and browsing, and Checkout UI, which implements the shopping cart and the checkout process. The backend services include the same logical services that were described at the start of this article. We have turned each of 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 of the 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](http://www.infoq.com/presentations/evolving-architecture-guardian-uk) 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](http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html). 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 are focused 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 home-grown code or a PaaS-like technology such as Netflix [Asgard](https://github.com/Netflix/asgard) and related components, or an off the shelf PaaS such as [Pivotal Cloud Foundry.](http://www.gopivotal.com/platform-as-a-service/pivotal-cf)

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, then tangled dependencies might make it difficult to decompose your monolithic application into a set of services.

Because of these issues, adopting a microservice architecture should not be undertaken lightly. However, for applications that need to scale, such as a consumer-facing web application or SaaS application, it is usually the right choice. Well known sites such as [eBay](http://www.addsimplicity.com/downloads/eBaySDForum2006-11-29.pdf)[PDF], [Amazon.com](http://highscalability.com/amazon-architecture),[Groupon](https://engineering.groupon.com/2013/misc/i-tier-dismantling-the-monoliths/), and [Gilt](http://www.slideshare.net/LappleApple/gilt-from-monolith-ruby-app-to-micro-service-scala-service-architecture) 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.

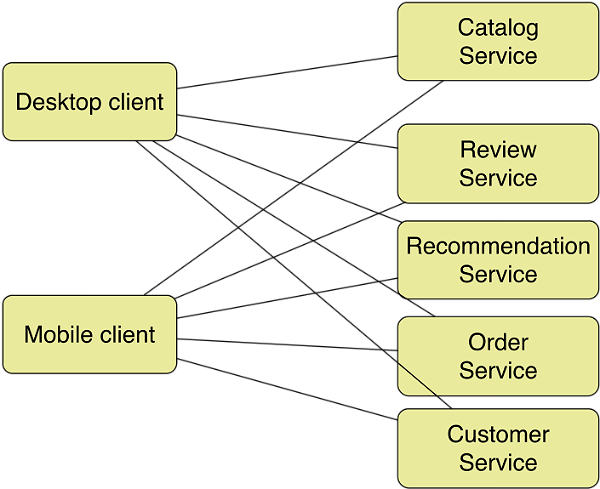
## **Communication mechanisms in a microservice architecture**

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?

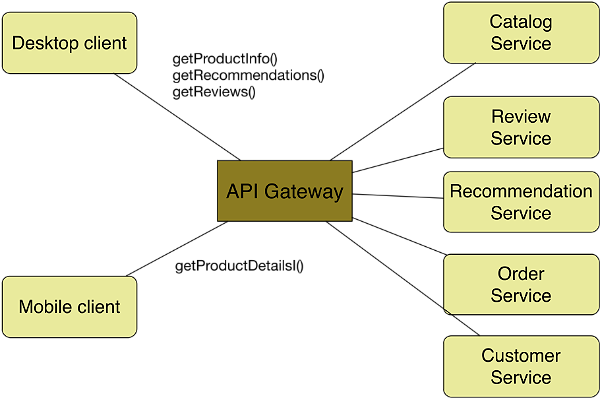
An application client, such as a native mobile application, could make RESTful HTTP requests to the individual services as shown in figure 4.



**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 web page could potentially require calls to large numbers of services. Amazon.com, for example, [describes](http://highscalability.com/amazon-architecture) how some pages require calls to 100+ services. Making that many requests, even over a high-speed internet connection, let alone a lower-bandwidth, higher-latency mobile network, would be very 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, where as 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](http://techblog.netflix.com/2013/01/optimizing-netflix-api.html) how each request fans out to on average six backend 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 examples, 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 are unaffected.

Now that we have looked at how the API gateway mediates between the application and its clients, let’s now 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 to 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 where service instances are ephemeral. Applications need to use a service discovery mechanism. Some applications use a service registry such as Apache [ZooKeeper](http://zookeeper.apache.org/) or Netflix [Eureka](https://github.com/Netflix/eureka). In other applications, services must register with a load balancer, such as an internal [ELB in an Amazon VPC](http://docs.aws.amazon.com/ElasticLoadBalancing/latest/DeveloperGuide/USVPC_creating_basic_lb.html).

### **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 needing a message broker, which is yet another moving part that adds to the complexity of the system. Another downside is that request/reply-style communication is not a natural fit.

There are pros and cons of 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 is manipulating 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**

For example, 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 their 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 where customers are managed by the CustomerService and orders are managed 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**

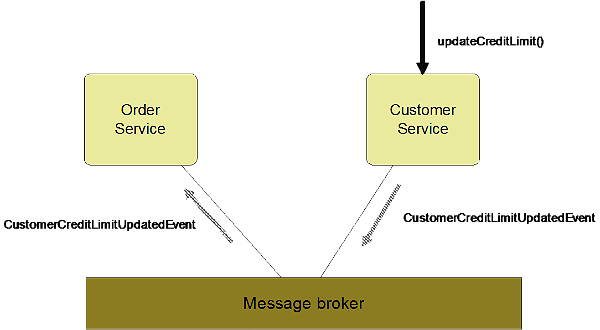
The problem of keeping the credit limit up to date in OrderService is an example of the more general problem of handling requests that update data owned by multiple services.

### **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 really 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.](http://queue.acm.org/detail.cfm?id=1394128) Services publish events announcing 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.



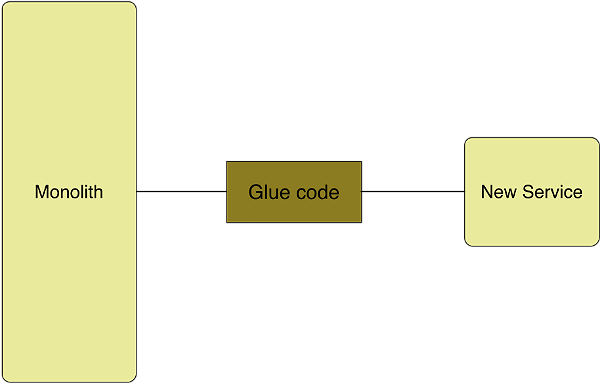
**Figure 6 - replicating the credit limit using events**

A major benefit of this approach is that producers and consumers of the events are decoupled. 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, greenfield 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 that 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 also 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](http://microservices.io/patterns/monolithic.html) 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 do and you are often permanently locked into your initial technology choices. For large applications, it makes more sense to use a [microservice architecture](http://microservices.io/patterns/microservices.html) 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](https://twitter.com/crichardson).

The Strengths and Weaknesses of Microservices

by [**Abel Avram**](http://www.infoq.com/author/Abel-Avram) on May 28, 2014 *|* [1**Discuss**](http://www.infoq.com/news/2014/05/microservices#theCommentsSection)

* Share
* **[Share](http://www.addthis.com/bookmark.php?v=250&username=infoq)**
* |
* [Share on facebook](http://www.infoq.com/news/2014/05/microservices)
* [Share on digg](http://www.addthis.com/bookmark.php?v=300&winname=addthis&pub=infoq&source=tbx-300&lng=es-419&s=digg&url=http%3A%2F%2Fwww.infoq.com%2Fnews%2F2014%2F05%2Fmicroservices&title=The%20Strengths%20and%20Weaknesses%20of%20Microservices&ate=AT-infoq/-/-/538f6a4c3fb871a4/2&frommenu=1&uid=538f6a4cbfd13f7b&ct=1&pre=http%3A%2F%2Fwww.infoq.com%2Fsearch.action%3FqueryString%3Dmicroservices%26page%3D1%26searchOrder%3D%26sst%3DyWc44dKiD2sHyfO0&tt=0&captcha_provider=nucaptcha)
* [Share on dzone](http://www.addthis.com/bookmark.php?v=300&winname=addthis&pub=infoq&source=tbx-300&lng=es-419&s=dzone&url=http%3A%2F%2Fwww.infoq.com%2Fnews%2F2014%2F05%2Fmicroservices&title=The%20Strengths%20and%20Weaknesses%20of%20Microservices&ate=AT-infoq/-/-/538f6a4c3fb871a4/3&frommenu=1&uid=538f6a4cc0904733&ct=1&pre=http%3A%2F%2Fwww.infoq.com%2Fsearch.action%3FqueryString%3Dmicroservices%26page%3D1%26searchOrder%3D%26sst%3DyWc44dKiD2sHyfO0&tt=0&captcha_provider=nucaptcha)
* [Share on twitter](http://www.infoq.com/news/2014/05/microservices)
* [Share on reddit](http://www.addthis.com/bookmark.php?v=300&winname=addthis&pub=infoq&source=tbx-300&lng=es-419&s=reddit&url=http%3A%2F%2Fwww.infoq.com%2Fnews%2F2014%2F05%2Fmicroservices&title=The%20Strengths%20and%20Weaknesses%20of%20Microservices&ate=AT-infoq/-/-/538f6a4c3fb871a4/4&frommenu=1&uid=538f6a4c1c0b07ba&ct=1&pre=http%3A%2F%2Fwww.infoq.com%2Fsearch.action%3FqueryString%3Dmicroservices%26page%3D1%26searchOrder%3D%26sst%3DyWc44dKiD2sHyfO0&tt=0&captcha_provider=nucaptcha)
* [Share on delicious](http://www.addthis.com/bookmark.php?v=300&winname=addthis&pub=infoq&source=tbx-300&lng=es-419&s=delicious&url=http%3A%2F%2Fwww.infoq.com%2Fnews%2F2014%2F05%2Fmicroservices&title=The%20Strengths%20and%20Weaknesses%20of%20Microservices&ate=AT-infoq/-/-/538f6a4c3fb871a4/5&frommenu=1&uid=538f6a4ce2dcf374&ct=1&pre=http%3A%2F%2Fwww.infoq.com%2Fsearch.action%3FqueryString%3Dmicroservices%26page%3D1%26searchOrder%3D%26sst%3DyWc44dKiD2sHyfO0&tt=0&captcha_provider=nucaptcha)
* [Share on email](http://www.infoq.com/news/2014/05/microservices)
* [**Read later**](http://www.infoq.com/news/2014/05/microservices)
* [**My Reading List**](http://www.infoq.com/showbookmarks.action)

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? Or, are microservices simpler than monolithic solutions?

Before discussing these issues, it would be a good idea to include a definition. In their article entitled [Microservices](http://martinfowler.com/articles/microservices.html), [James Lewis](https://twitter.com/boicy) and [Martin Fowler](http://www.martinfowler.com/) define the microservices architectural style as 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 some drawbacks?

[Benjamin Wootton](https://twitter.com/benjaminwootton), CTO of [Contino](http://contino.co.uk/), is currently architecting a system based on microservices and has encountered a number of difficulties detailed in an article called [Microservices - Not A Free Lunch!](http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html). Here are a digest of those.

**Major Operations Overhead**

20 services can become 40-60 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 a “daunting task”, according to Wootton:

Productizing 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” it results that “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 because

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 it's 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, observes Wootton:

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 a 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 is 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.”

**The 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 to “manage 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. Also,

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](file:///C:\Users\Abel\AppData\Local\Temp\WindowsLiveWriter1286139640\9A726B119EAA\brady.lucidgene.com), a reader commenting on Wootton’s article, added from his experience attempting to transition from a monolithic app to microservices:

I was working on a project moving towards micro-services 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, 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](file:///C:\Users\Abel\AppData\Local\Temp\WindowsLiveWriter1286139640\9A726B119EAA\www.cloudsidekick.com), 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 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. twenty 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 purest 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](http://www.dreamsongs.com/RiseOfWorseIsBetter.html) 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.

# Microservices: Usage Is More Important than Size

Using only size for defining [microservices](http://www.infoq.com/news/2014/03/microservices-soa) is a poor measure and useless for determining whether a service has the right responsibilities, [Jeppe Cramon](http://www.cramon.dk/Cramon/Front_page.html) states in a [series of blog posts](http://www.tigerteam.dk/2014/microservices-its-not-only-the-size-that-matters-its-also-how-you-use-them-part-3/) clarifying his view on microservices and the coupling problems he finds in synchronous two-way communication.

Jeppe, a Danish contractor and expert on large-scale system integration and SOA, argues that if we combine small microservices with synchronous communication we are actually sent back to the 1990s with Corba, J2EE and distributed objects, only this time with technologies such as HTTP instead of RMI or IIOP. Just because microservices tend to use HTTP, JSON and REST doesn’t make the disadvantages of remote communication, as summarized in the [8 fallacies of distributed computing](http://www.rgoarchitects.com/Files/fallacies.pdf), disappear.

Creating an alternative to this synchronous communication Jeppe builds on among others Pat Helland and his [Life beyond Distributed Transactions: An Apostate’s Opinion](http://www-db.cs.wisc.edu/cidr/cidr2007/papers/cidr07p15.pdf) where Pat argues against distributed transactions. Jeppe’s solution is three-folded:

1. Allocating data to services.
2. Identifying data.
3. Communication between services.

Jeppe solves his first two points using [Domain-Driven Design](http://en.wikipedia.org/wiki/Domain-driven_design) (DDD) concepts with data collected into entities and aggregates where each aggregate is uniquely identifiable using e.g. a [UUID](http://en.wikipedia.org/wiki/Universally_unique_identifier) and allocated to one service. These aggregates must be consistent after a transaction, the rule of thumb being: 1 use case = 1 transaction = 1 aggregate.

Communication between services is solved with asynchronous communication, specifically a true one-way communication with the sender transmitting a message using a transport channel, e.g. a message queue. The sender is not waiting for the message to be received; instead the transport channel assumes responsibility for receiving and delivering the messages to the recipient.

Jeppe’s ambition is to continue with a blog post looking at ways to divide services into smaller services and how they can communicate using asynchronous one way communication.

[Microservice Architecture](http://martinfowler.com/articles/microservices.html) has emerged during the last few years as a concept to describe a way of designing software applications as suites of autonomous and independently deployable services that cooperate to fulfill business capabilities. A [conference](https://skillsmatter.com/conferences/6312-mucon) is scheduled for late November in London.

# GOTO Berlin: Microservices as an Alternative to Monoliths

Are we building systems that are too big, bigger than they need to be, [James Lewis](http://bovon.org/index.php/archives/author/james), a principle consultant at ThoughtWorks, started his presentation at the [GOTO Berlin Conference](http://gotocon.com/berlin-2013/) when[talking](http://gotocon.com/berlin-2013/presentation/Microservices%20-%20adaptive%20architectures%20and%20organisations) about "Microservices - adaptive architectures and organisations". There are examples where up to 90% of the total cost of ownership incurs after the launch of a system. There are also lots of examples of projects where we as an industry have spent a huge amount of money on building very large, complex and ultimately unsuccessful systems.

James’ experiences at large organisations where a traditional way of building systems is to put all functionality in one big application, with one big database and the problems this creates, has led him into a path of building systems where entirely separate business capabilities are kept separate in smaller parts, as microservices, each with their own database.

His first step on this path is the idea of hexagonal business capabilities and refers to [Alistair Cockburn’s](http://alistair.cockburn.us/) [Hexagonal Architecture](http://alistair.cockburn.us/Hexagonal+architecture). One single business capability or functionality, together with its own data, forms a hexagon, a [bounded context](http://en.wikipedia.org/wiki/Domain-driven_design#Bounded_context) using [DDD](http://en.wikipedia.org/wiki/Domain-driven_design) terms. All these hexagons are then put together in a larger hexagon, eventually forming a system.

His 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 is 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](http://en.wikipedia.org/wiki/REST) for communication. Two standard application protocols James finds useful together with REST are [Atom](http://en.wikipedia.org/wiki/ATOM) and [AtomPub](http://en.wikipedia.org/wiki/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. If some parts have a high load and other a much lower load they must still run with the same capacity. With small services they can be deployed on different servers using different number of servers. 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 e.g. using programmable infrastructure. Advances in virtualization and [Infrastructure as a Service](http://en.wikipedia.org/wiki/IaaS#Infrastructure_as_a_service_.28IaaS.29), IAAS, providers during the last couples of years makes this possible.

The [GOTO Berlin Conference 2013](http://gotocon.com/berlin-2013/) is the first GOTO conference in Berlin, with over 400 attendees and about 80 speakers.

# Microservices? What about Nanoservices?

Earlier this year [we posted an article](http://www.infoq.com/news/2014/03/microservices-soa) 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) believed Microservices is really just SOA and a new term is probably not needed. After that article Steve posted [another entry](http://service-architecture.blogspot.co.uk/2014/03/microservices-is-sod-all-within-soa.html), where he discusses how Microservices is really just Service Oriented Delivery and how attempts to distance it from SOA do not make sense:

[...] Microservices lays down some nice rules for implementing certain parts of an enterprise but those are best served by an honesty that its 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](http://arnon.me/2014/03/services-microservices-nanoservices/):



He also believes that Microservices, as defined by Fowler and James, is nothing more than SOA, perhaps without some of the "bad misconception" that meant some associated SOA with a hard requirement on WS-\* and ESBs. He then asks precisely what is a Microservice and quotes James Hughes (the link he uses is not valid at the time of writing):

First things first what actually **is** a micro service? Well there really isn’t a hard and fast definition but from conversations with various people there seems to be a consensus that a micro service is a simple application that sits around the 10-100 LOC mark.

According to Arnon, James does admit later in the same article that lines of code are a very bad way to compare service implementations (something which [Jan wrote about too](http://www.infoq.com/news/2014/05/microservices-usage-size) in his article on usage being more important than size), but Arnon wants to focus on the consensus aspect 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 James mention) generate serialization/deserialization code (protobuff, thrift, avro etc.)  - this is essentially building on a smart [service host](http://arnon.me/soa-patterns/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 use 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 the smaller the service gets (towards what he calls "Nano services") the more mangement overhead you have to worry about, 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 antipattern 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 nano-services 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. So perhaps this is a term that the industry will have to accept, even if the majority understand it as another name for SOA?