# #1

# For Information

# Extracts from: Microservices: Decomposing Applications for Deployability and Scalability

Posted by [Chris Richardson](http://www.infoq.com/author/Chris-Richardson) on May 25, 2014

…

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.

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.

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. [MP: in the product of my last Client, I found a class that could respond up to 1000 method-calls…]

…

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.

…

### 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. 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.

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.

… 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.

**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.

# #2

# Analysis of PoSO with Regards to Microservices

Resource: “Updated Principles of Service Orientation”, posted by [Michael Poulin](http://www.infoq.com/author/Michael-Poulin) on Nov 03, 2014

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **№** | **Principle Title** | **Principle Content**  | **Relations to Microservices** | **Compliance with PoSO** |
| 1 | **Service Discoverability Principle** | Services are supplemented with communicative meta data by which they can be effectively discovered and interpreted. | Optional and depends on implementation | partial |
| 2 | **Standardized Content Types of Service Contracts** | Service Contracts are in compliance with the standardized contract content types and represent mutual agreements between the service provider and service consumers on what content is to be included. | 1. “The developers are free to choose whatever technologies make sense, provided that the service honours the API contract.”
2. “That's not an absolute, some changes will change service interfaces resulting in some coordination, but the aim of a good microservice architecture is to minimize these through cohesive service boundaries and evolution mechanisms in the service contracts.

…For the microservice community, overheads are particularly unattractive. That isn't to say that the community doesn't value service contracts. Quite the opposite, since there tend to be many more of them. It's just that they are looking at different ways of managing those contracts. Patterns like [Tolerant Reader](http://martinfowler.com/bliki/TolerantReader.html) and [Consumer-Driven Contracts](http://martinfowler.com/articles/consumerDrivenContracts.html) are often applied to microservices. These aid service contracts in evolving independently. Executing consumer driven contracts as part of your build increases confidence and provides fast feedback on whether your services are functioning. Indeed we know of a team in Australia who drive the build of new services with consumer driven contracts. They use simple tools that allow them to define the contract for a service” | none |
| 3 | **Service Execution Context Principle** | Service interactions and execution take place in the Execution Context. The Execution Context can affect service behaviour and lead to changes in the reachability and Real World Effect of the services. | Microservices operate within an application that is irrelevant to the EC. This is regardless that ‘internal’ services may become quite EC dependent and lead to serious mistakes when the EC is ignored. | none |
| 4 | **Service Separation of Concerns Principle** | Services own and carry out only their own functionality, independent from providers of supplemental functionality and decoupled from information sources and original information structures as much as possible. | A fine-granularity of the Microservices inside the application can be interpreted as a compliance to this principle. However, Microservices do not regulate complexity of the code behind the interfaces; they only require the ‘service’ should ne small/micro. What does this mean is not defined. | partially |
| 5 | **Service Abstraction Principle** | Service interfaces abstract the service realization or body. Not all service functionality and capabilities may be visible through the service interfaces. Service interfaces cannot be reliable resources for understanding what the service does and its RWE. | As any Web Service or REST interface abstracts the implementation behind it | Yes |
| 6 | **Service Contract Abstraction Principle** | A Service Contract abstracts the service. A Service Contract contains only information about the service that is agreed between the service provider and the service consumer(s). This information has to be sufficient for interacting with the service, utilizing agreed service functionality and reaching agreed Real World Effect in a particular Execution Context. | A contract in Microservices is the API. This is a one-way contract, i.e. it is not a contract but an enforcement of non-negotiable constraint. Because of fine-granularity of the Microservices, the API is ‘type’-specific and, thus, expose the implementation underneath | none |
| 7 | **Service Composability Principle** | Services are effective composition participants, as well as effective composition holders, regardless of the size and complexity of the composition | Since many Microservices are realised as POJO (database linked objects) they cannot participate in any number of compositions being limited by the limitations of their resources (databases).Microservices do not organise other Microservices – this is the work of the Application | none |
| 8 | **Service Relative Autonomy Principle** | Services exercise a relative independence from their Execution Contexts, control their internal resources and if necessary, interact with external entities in the SO ecosystem on a contractual basis. | Since many Microservices are realised as POJO (database linked objects) they are coupled to the resources. But may ignore other elements of the EC. No contractual basis is used for the Microservice interactions beside the API (not even an SLA for these APIs) | No conclusion |
| 9 | **Service Reusability Principle** | Services contain and express logic that can be reused in different Execution Contexts; services can be positioned as reusable intra- and inter-enterprise resources. | Since Microservices do not consider EC, their reusability is uncertain. We do not know a priory if particular API would behave the same way in another Application | No conclusion |
| 10 | **Service State Management Principle** | Services manage their own state as needed and may defer the management of their states in order to minimize consumption of service environment resources. | According to some authors, Microservices may use Messaging systems. It is unclear if the Messaging systems are a part of interaction between ‘services’ of the same application or between applications. Also, Microservices are always stateless. | partially |
| 11 | **Service Loose Coupling Principle** | Services impose low coupling requirements between their interfaces and the service body, as well as between their interfaces and the service consumers in the Execution Contexts. | Only loose-coupling between the interface and body is preserved for the Microservices.  | partially |
| 12 | **Service Precise Boundaries Principle** | The means of service implementation may not cross service boundaries, neither via invocation nor via dependency on the resources that are out of the service’s ownership or explicit control. | This principle is not preserved at all and the body of one Microservices can communicate in any possible way with the outside world even sharing the implementation/body with other Microservices.  | none |

Thus, those who claim that Microservices are SOA services in the micro form actually refer to those PoSO principles that relate to the service interfaces only.