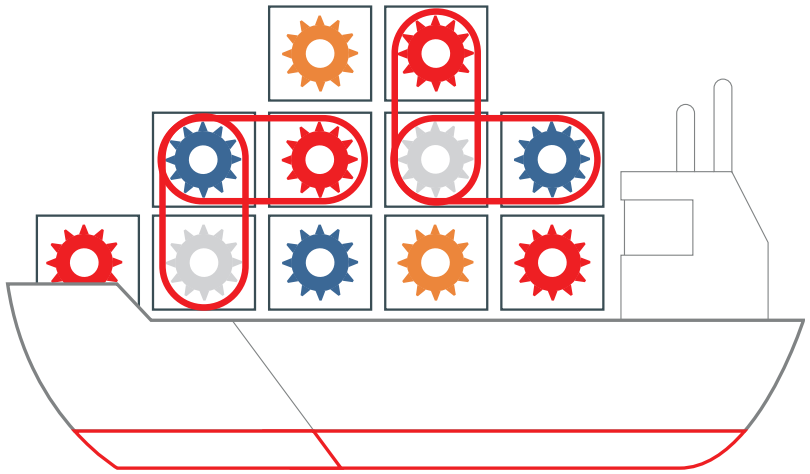
Microservices: Why Should Businesses Care?



akana™

Powering the API Economy

Abstract: Microservices offer a way to build web-scale applications by breaking a large application down into small, independent services. Microservices enable IT organizations to be more agile and reduce costs by taking advantage of the granularity and reuse of microservices. Yet, like other new architectural paradigms, they introduce challenges as well. This paper looks at how microservices work and offers some thinking on how to make the most of them, in business terms while retaining their inherent technological advantages.

Introduction

Microservices are gaining traction, making headlines and stimulating new thinking about how to organize application architecture. But, what exactly are microservices? At a high level, microservices are a new way to build applications. They break a large application down into small, independent services that are not language specific. Regardless of the language you use within your organization, you can implement a microservices architecture. Microservices offer IT organizations a great deal of potential for agility and cost reduction due to their granularity and reuse. Yet, like so many new architectural paradigms, they introduce challenges as well. This paper looks at how microservices work and offers some thinking on how to make the most of them, in business terms while retaining their inherent technological advantages.

Microservices, an Overview

The term “microservices” refers to a style of software architecture where complex applications can be composed of small, independent services. These processes, or “services” exchange data and procedural request using application programming interfaces (APIs) or events that are invariably standards-based and language-agnostic. Yet, microservices go beyond the actual architecture. They are really the product of a rapid development process, such as DevOps, service-oriented architecture (SOA) principles, and containers. When you combine fast-moving software development that leverages the principles SOA and containers, you’ve got microservices.

Unlike a monolithic application, which is usually designed as single process that encapsulates several functions related to the application, the microservices paradigm turns the monolithic architecture inside out and powers the equivalent application functionality through a set of decoupled microservices. For example, an ERP application might have an

internal process that allows a user to input a customer's contact information and create a log-in credential. Microservices might recreate that workflow with one service for the customer's name and address, another for the phone number, another for the email, and one for the log-in credentials. Also, each microservice can be written in whatever language that the developer chooses to implement it in and can be individually scaled up or down based on load. The approach enables developers to reuse the individual components to build new applications much more quickly than would be possible with conventional development tools and techniques.

Microservices are focused on providing one capability. "Micro" doesn't necessarily mean that it's small, although it often is. It's just singularly focused. It provides one piece of functionality very well. An ideal microservice also owns it data and data model, and is not dependent on any other microservice or service for it.

The Appeal of Microservices

Microservices have been in the background of IT for a long time, but they are growing in popularity today because we have new supporting technologies that makes them practical. The enthusiasm about microservices goes beyond feasibility, though. Done right, they greatly improve the entire IT agility picture, with regard to application development. The contrast is particularly relevant when comparing the development of new features in a large, monolithic application versus the microservices approach.

The Relative Inefficiency of Monolithic Architectures

An e-commerce suite provides a good example of a monolithic application and some of its inherent inefficiencies. The application might consist of a front end user interface along with services for managing a product catalog, processing orders and customer accounts. The services share a domain model consisting of entities, e.g. "Product" or "Order."

Even though the application has a logically modular design it is deployed as a monolith. With Java, it would a single WAR file running on a web container like JBOSS. This architecture has a number of benefits: They are simple to develop. Most development tools are geared to this approach. They are easy to test because they are just a single application. And, they are relatively simple to deploy.

It's a good approach for smaller applications. Unfortunately, monoliths quickly become unwieldy when applications get complex. They're hard for developers to understand and maintain. Frequent deployments are a challenge. To change one element, the team has to build and deploy the entire monolith – a complex, risky process that usually results in numerous additional test cycles.

The monolithic approach also impedes trial and adoption of new technologies. Trying a new infrastructure framework might mean rewriting the whole application. The monolithic architecture doesn't scale well in support of large, long-term applications.

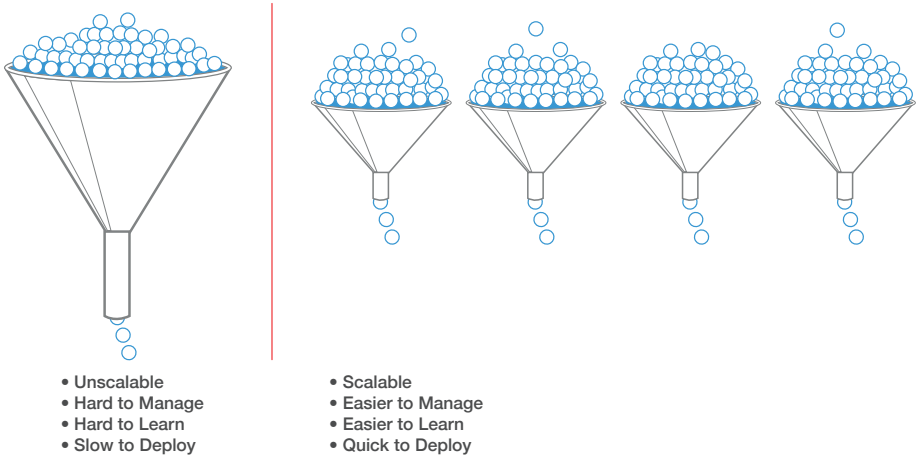


Figure 1 - Development workflow in a monolithic application vs. microservices.

How Microservices Decompose Monolithic Applications

Figure 1 visualizes the contrast using a set of funnels, which represents the throughput of feature builds in the software development process. The monolithic application is shown as the big funnel. A lot features are in the queue to get developed and deployed, but the process is slowed down by organizational and practical constraints. Within a period of time, only a certain amount of features will pass through the funnel.

Alternatively, microservices creates the equivalent of many smaller funnels. The development and deployment process scales more easily. It's easier to manage. Independent team can work on their own microservices, choose their own language and data sources.

The monolithic application is not scalable in the long term. It's not scalable from a performance perspective because it's very difficult to scale horizontally and vertically. It is also not scalable from an organizational perspective. It's very hard to manage one giant project versus a whole bunch of small projects. There is a lower cadence and slower pace of change. Agility suffers.

Microservices are also much easier for new developers to learn and then deploy. In contrast to the monolithic application, whose app servers are complex and difficult to master, container deployment is relatively simple. When an effective DevOps regimen is added to the process, it gets even easier. As other capabilities are layered into a maturing operational organization it becomes very simple to deploy and manage the microservices. However, microservices architectures are not everyone and will not be successful in organization which have not adopted a true DevOps culture along with containerization. Also security considerations are not yet that well defined for a microservices architecture and organization need to assess the risk and take proper steps to address security.

akana™
Powering the API Economy

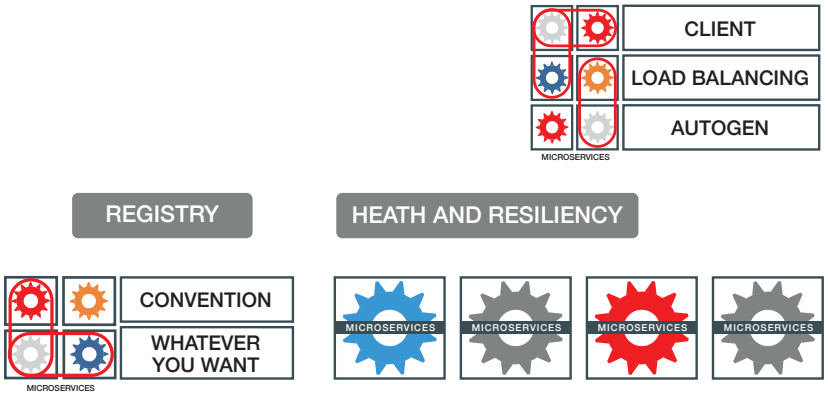


Figure 2 - Microservices architecture

The Microservices Architecture

To understand microservices, you have to see the bigger picture of how they are built and deployed. Figure 2 depicts a highly simplified microservices architecture. A registry allows you to register new end points for microservices. Microservices have re-invented the registry. The registry has become something of a runtime discovery mechanism than a design time discovery mechanism, but it facilitates both models.

The microservices themselves are deployed on containers. There should be a set of well-defined conventions on how your microservice interacts with the rest of the world. On the top right-hand side of Figure 2, a microservice uses a client for load balancing and automatically generating proxy scripts to facilitate the calls to other microservices. There are available frameworks and platform that will help you and facilitate many of these capabilities.

Containers, also known as “container-based virtualization,” automate the deployment of applications inside software containers by providing an additional layer of abstraction and automation of operating-system-level virtualization on Linux. They potentially reduce overhead associated with having every virtual machine (VM) run a completely installed operating system. Containers are essentially lightweight virtual machines. Different virtual machines can be deployed off of the same hardware. Containers are a lighter weight and smaller version of the virtual machine. A container could be limited to just the application and the supporting environment. It does not include the operating system and hypervisor framework.

Microservices vs. SOA

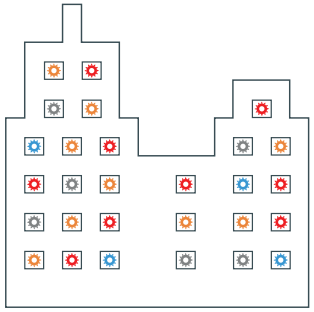
Microservices are different from Service-Oriented Architecture (SOA) though the two architectural styles share a common ancestry and a number of common traits. SOA became very vendor and standards driven, though that was never the intent. SOA was supposed to be technology agnostic, but many years ago, the vendors drove SOA down a particular path. They gave us SOAP and enterprise service buses (ESBs). This is very heavy architecture, which is contrary to today’s microservices. In some ways, microservices are not SOA. They are supposed to be lightweight and small. In other respects, while microservices reaffirm many SOA principles, they are almost a reaction against that traditional vendor-driven SOA. Microservices are decidedly anti-ESB, for instance.

akana™
Powering the API Economy

The Role of the APIs in Microservices

Understanding the role of the API in a microservices architecture requires a sense of how APIs differ from microservices. APIs are about providing connectivity. How does one connect an application to another? How does one support digital transformation and support a large eco-system of developers and partners and making it easy for them to consume your data or applications. Microservices are different. They provide agility and scale. The difference between an API and a microservice is not based on technology. It's got to do with the business case. An API is about making a service or application available for a large set of developers, and a microservice is about building flexible and agile application in and delivering them faster. These definitions are not necessarily absolutes, but they help us understand them.

APIs and microservices are complementary to each other. An API takes an existing service, productizes it and then enables it to be marketed to an end consumer. The end consumer could be internal, a partner or the general public. APIs are geared towards easier and better consumption of those services. The API is the connector that allows you to invoke microservices from an application. Essentially, you need APIs to make microservices work. As a result, the management and security of APIs are critical to a proper functioning microservices architecture. Microservices, on the other hand, are used to build the application and services themselves, while an API can frontend a single or multiple microservices in the backend.



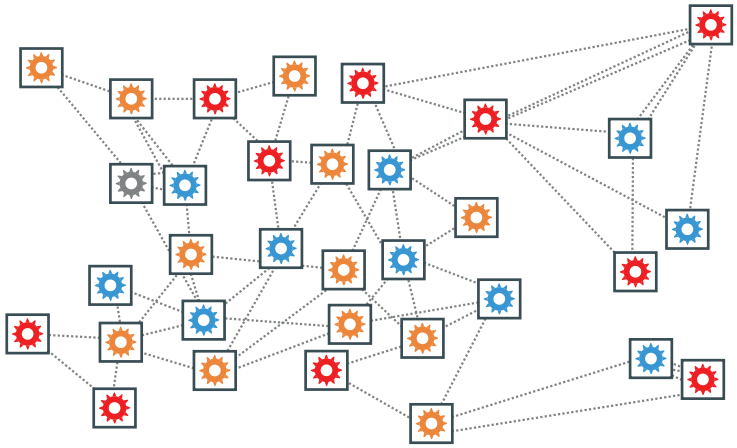
Why Should Businesses Care about Microservices?

Microservices are a potential boon to business because they can help improve agility. The combination of DevOps and microservices enables a development organization to move faster in delivering new features. As a small development effort, a microservice can be built faster than the typical large-scale software development project involving a monolithic application. It's easier and faster to manage changes. Microservices are more fluid and move at their own pace. There are no waterfalls in development process, as you have with a monolithic application. Each microservice is independent and can be developed with any programming language or constructs. You also get better variability. Small microservices can be combined into bigger solutions. Of course, this assumes that the DevOps process is working well, that the organization understands how to do microservices with DevOps and that the microservices can be managed effectively.

akana™
Powering the API Economy

Microservices failures are also less catastrophic than breakdowns in bigger systems. A failure in one part of a monolithic application is usually quite detrimental to everything else. In the aftermath, you have to diagnose the problem and perhaps release another version of the application to fix that particular error. It's quicker and easier to fix a problem in a microservices architecture. You identify it, isolate it, and compensate for it. If you have a microservices architecture, you can scale microservices to overcome a performance constraint. Or you can quickly build a new version of the faulty microservice without dealing with the operational waterfall that exists with a monolithic application.

Microservices give you better reliability and elasticity. You can scale up any one part of your application. For example, if your log-in is suffering because you are getting of a lot of new users this month, you can scale that particular part of the application better with microservices architecture. Think about how it's always been done. If you have a spike in log-ins, you would have to stand up new app servers that run the big monolithic application just because that one particular page or one particular part of the site is being used more than others. The microservices architecture provides a more elastic approach. In this case, containerization allows you to quickly deploy as many new instances of the log-in microservice as you need and then load balance across all of them.



Re-Architecting an Application for Microservices

Embracing microservices will necessarily involve some pretty big changes in thinking, processes and tooling. This will be true whether you're starting completely from scratch with a "green field" application or re-architecting existing applications into the microservices mode. It may be easier to do a green field project, but there are still a number of major issues to figure out before you start.

While a new framework and platform are required, there is a danger of investing in too heavyweight a platform. You may face pressure from various stakeholders to acquire tools that create bloat. Our recommendation is to keep it as lightweight as possible. Avoid implementing a huge platform and framework and standardizing across the entire organization just to implement microservices. The "must haves," however, are a registry, the ability to do load balancing and smart end points.

akana™
Powering the API Economy

Given the effort involved in re-architecting an application, the best practice is to select one that is strategic. (Perhaps not as your first project. For that, you may want to test the process on something small and peripheral.) However, when you need to make an investment of resources, it should be an application that can justify the time and money. It has to be strategic to the business.

Prerequisites and Sanity Checks

These are prerequisites or sanity checks prior to moving ahead and re-writing an application to leverage a microservices architecture: You will need smaller teams, for one thing. With monolithic applications, even if you divide it up functionally, ultimately you have a large team. It's the interdependencies that are tough to avoid, especially when it comes to deployment and release management. You should also try to leverage lightweight communication protocols because if you have a lot of microservices, you're going to affect your network. For example, a single page load on Amazon.com might call 150 services. You're going to have a lot of traffic as you through all of those calls. It's best to use asynchronous loading.

Breaking Up Component Parts

One approach to re-architecting an application is to conduct the process in stages, starting by breaking it up into its component parts. Each of those component parts provides a set of or one or more services. You may need to move logic out of your data tier and push data into different data stores without having a mass migration of a large volume of data that uses different architectures for different parts of the application. Moving data stores gives you more options from an application architecture perspective. You get more control over the functionality of the microservices you deploy.

Ultimately, you will have a set of services on the front end depending on a set of microservices on the back end. You have now decoupled yourself entirely from the data. Each one of these different microservices has its own data. They might be using NoSQL or something equivalent to actually persist and aggregate that data. The goal is to ensure that each of those things is independent. Potentially you have wired the microservices together.

In the end, you have migrated from a single data model and single database (Or single data store to a multi-data store) and broken up your application into individual microservices. You have split your data and application. You can scale any one of these little things in a container as much as you want, to support the needs of your front end.

New Skills and Practices

Getting into microservices means training people how to design microservices architectures. These are new skills, not widely known at this time, though the IT profession continues to become more fluent in microservices. Most notably, microservices changes development practices. The scope of development projects is different to the point where the whole requirements gathering and coding thought process has to change. This is true even for teams that are well-versed in agile methodologies. You're developing in tiny bites. Testing also changes. QA people have to understand what a microservices architecture is and what that means to the testing effort, so that things don't fall through the cracks.

Martin Fowler, an industry thought leader on enterprise software, provides an outstanding rundown on new skills and practices that IT organizations must master to succeed with

microservices.[1] In Fowler's view, the following are the concepts that people need to wrap their heads around to get an idea of what to implement the right kind of microservice. As he notes, you can take a bad application and build a bunch of bad microservices out of it.

- **Componentization via services** – Knowing how to create an interface that leverages the best technology for the job, e.g. REST.
- **Organize around business capabilities** - Microservices need to be organized around distinct business capabilities. The microservices developer creates products, in effect, not projects. Given that the service will have its own complete lifecycle, it's really a matter of product management, not software development.
- **Deliver smart end points and dumb pipes** - You will have to orchestrate or choreograph the different end points. Load balancing is built into the framework and the end points. The registry contains end points. The content has decentralized governance. The registry is built into the service itself. The data management will become decentralized, moving to a more functional domain-driven design context versus a traditional data model.
- **Automate the infrastructure and design for failure** - The architecture needs to be continually adjusting for and compensating for failure. Ideally, you build failure into the testing of the microservices infrastructure so that while you're doing testing, services fail. Containers disappear or shut down. Things start running slower. The infrastructure has to scale automatically.
- **Design** - Traditional design and the oversights that you had are no longer relevant. Everyone will use the best tools to create the microservice. You will need to control the microservice definition and the interface but you cannot control all the moving parts of the process.
- **Designing microservices the right way** - You need to facilitate design time review of available services and find a way to force people to request the service. This is why you need to have a registry that enables people to look for services. If you don't find them, you have to request a new service or a new interface. Then somebody will say, "Yes. That looks like a good interface. Let's build that as a microservice." There will probably be some approvals to go through.
- **Adopt conventions** - You will need to adopt certain conventions about the interface that you're exposing. You will need to have an end point, for example. These conventions have to be established around the design of your interfaces. Then, they have to integrate within your framework and infrastructure to enable making those lasting choices and decisions during runtime.
- **Design for robustness** - Microservices affect design and the design pattern. You might have a proxy end point, for instance, that enables you to do the routing and resiliency on the back end. Then you need to understand distributed data design and the main driven design has around data. Ultimately, data has to separated into a microservice.

[1] http://martinfowler.com/bliki/MicroservicePrerequisites.html

Conclusion

Like earlier waves of change in enterprise architecture, microservices present a bundle of opportunities and challenges. The business upside is definitely available for organizations that embrace microservices and pursue them with the right tooling and processes. Microservices represent a quite new approach to creating applications, however. They combine the concepts of SOA, containers, and DevOps. As a result, getting to a successful microservices architecture will require changes on multiple levels. The way you conceive of an application, the way you staff the development and testing teams, the way you scope out the parameters of any give microservice – these are all going to require some pretty extensive rethinking of the way things get done. Microservices require new skillsets. Migrating old applications to microservices means breaking them down into component parts and putting them back together again. None of this is easy, but it is worth it. The gain is there if you do it right. You get more agility and flexibility with your software. You can scale elastically. You can move quickly. It's time to explore microservices.

About Akana

Akana is a leading provider of API Security and Management products that help businesses plan, build, run and share APIs, through comprehensive cloud and on-premise solutions that encompass API lifecycle, security, management and developer engagement. The world's largest companies including Bank of America, Pfizer, and Verizon use Akana solutions to transform their business.

For more information, please visit www.akana.com

akana™

Powering the API Economy

Akana
12100 Wilshire Blvd, Suite 1800
Los Angeles, CA 90025

(866) 762-9876 | www.akana.com | info@akana.com