Microservices, SOA, and APIs: Friends or enemies?

A comparison of key integration and application architecture concepts for an evolving enterprise

Kim J. Clark
Integration Specialist
IBM

21 January 2016

Comparing a microservices architecture and service-oriented architecture (SOA) is a sensitive topic and often cause for a swift source of disagreement. This article examines where these controversies stem from and considers how best to resolve them. It then looks forward to see how these concepts are combining with those of API management to enable more agile, decentralized, and resilient enterprise architectures.

Introduction

When comparing a microservices architecture and a service-oriented architecture (SOA), it is nearly impossible to gain agreement on how they are related to one another. Adding application programming interfaces (APIs) into the mix makes it even more challenging to understand the differences. Some might say that these concepts are distinct, solve their own set of problems, and have a unique scope. Others might be more generous and say that they achieve similar goals and work from the same principles. They might also say that a microservices architecture is a "fine-grained SOA" or that it is "SOA done right."

This article defines each of these concepts, explains where the varying opinions come from, and tries to find a middle ground. It also examines how these three concepts might be combined going forward.

An over-simplified view

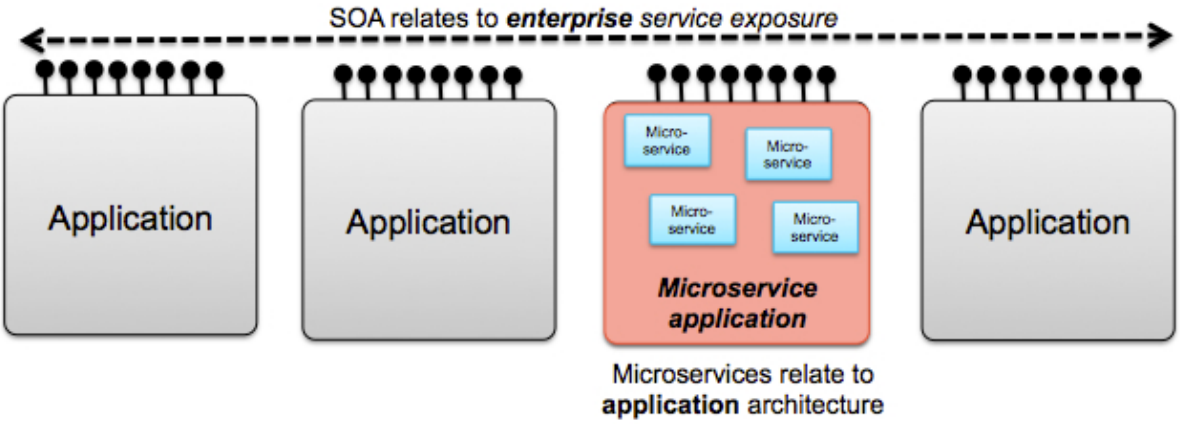
The reason that it is difficult to compare SOA and microservices is that their definitions have a lot of room for interpretation. If you have only a surface-level knowledge of the two concepts, they can sound similar. Key aspects, such as componentization, decoupling, and standardized communication protocols, describe most software initiatives in the last few decades, so we need to dig deeper.

Consider the following simple definitions:

- **Microservices architecture** is an alternative approach to structuring applications. An application is broken into smaller, completely independent components, enabling them to have greater agility, scalability, and availability.
- **SOA** exposes the functions of applications as more readily accessible service interfaces, making it easier to use their data and logic in the next generation of applications.

Figure 1 illustrates these definitions. An SOA appears to have an *enterprise scope*, where applications communicate with one another. An SOA exposes services through standardized interfaces between applications. The microservices architecture appears to have an *application scope*, with a focus on the structure and components within an application.

Figure 1. Differences between a microservices architecture and SOA



These definitions of SOA and microservices are too simplistic. In fact, the relationship between them is much more complex.

Dichotomy of SOA initiatives

When you look at an SOA in more detail, you can see that its original intent was broader than exposing interfaces as SOAP web services. SOA is based on two views that address two different needs.

Integration-led technical element

The first view encompasses the need to integrate deep into existing systems over their complex and often proprietary data formats, protocols, and transports. Then, the need is to expose them by using standardized mechanisms (such as SOAP/HTTP or more recently JSON/HTTP) to make them easier to re-use in new applications. This view is shown on the left side of Figure 2. Some or

all of this view is often referred to as an *Enterprise Service Bus (ESB) pattern*. However, this term is used indiscriminately to the point of making it meaningless.

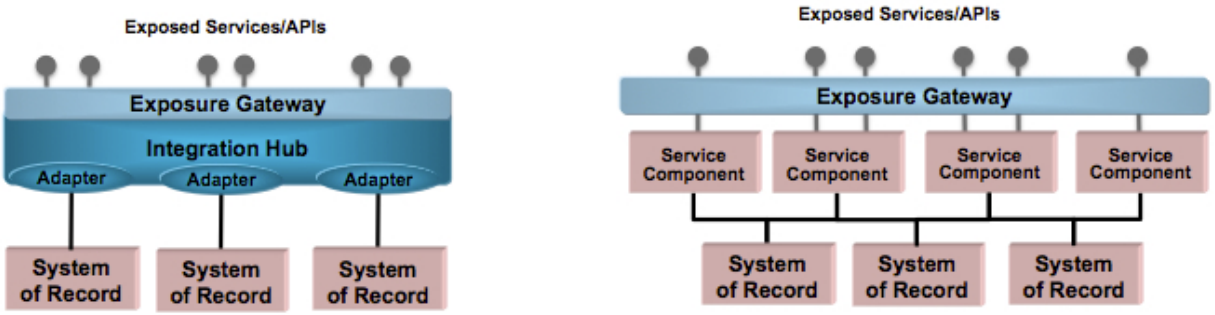
The need to do deep integration (integration hub and adapters) and to expose these integrations as services or APIs in a standardized way (exposure gateway) is essential. This aspect has everything to do with integration challenges and little to do with application design. Consequently, therefore, it appears to have little relationship with the microservices application architecture.

Business-led functional element

The second view is from a business perspective. The concern is that the interfaces on the current systems are largely meaningless. They don't make sense to the business, and they don't provide what's needed for the next generation of applications. They might be too granular, exposing too much of the complex data models within the systems. The data that is required might be spread across multiple systems. The data models might not resemble the terminology that is used by the business.

The need entails functional refactoring to expose something that the business can tangibly build into future solutions. This refactoring requires the creation of new applications to bind together requests across the existing systems of record. In the SOA reference architecture, these applications were often referred to as *service components* (right side of Figure 2). This view shows a relationship to application design (and, therefore, the microservices architecture) and the functional decomposition of capabilities into separate components.

Figure 2. Technical and functional views of SOA



The challenge of mixing the views

Organizations vary in which of the two views is the greater challenge. For some organizations, their greatest challenge is the diversity and complexity of the integration. For others, refactoring and relandscaping to achieve the right business functionality is the primary challenge. Figure 2

shows how different the problem can look, depending on which of these two challenges is forefront in your mind.

For many, the challenge is a painful mixture of both views. It is painful because it's difficult to merge the two views into a single course of action. Integration tools are not the right place to perform business logic. Conversely, you do not want your business applications to be cluttered with technical integration concerns.

The goal of most SOA programs is to achieve the functional aspects. They want easily accessible business-relevant services that can be used to build new applications more effectively. However, many run out of momentum, or more commonly out of budget, while still solving technical integration challenges. In large enterprises, SOAs are often perceived to have failed. This thought might be true in that they failed to deliver the final business value, although huge strides were made to improve the accessibility of systems of record. However, in smaller companies (or more contained environments within larger companies), SOAs often claim true business successes because they can quickly overcome integration issues and move on to functional benefits.

These two views of SOA make comparisons with microservices challenging.

How APIs compare to SOA exposed services

APIs used to mean low-level programming code interfaces. In recent years, the term has been reappropriated to mean simple interfaces provided over HTTP. Typically, it equates to *REST interfaces*, which provide data by using the JSON data format (sometimes XML), and the HTTP verbs PUT, GET, POST, and DELETE to depict create, read, update, and delete actions. These protocols and data formats are simpler to use than the web services standards based on SOAP that were more prevalent in early SOA. Also, they are more suited to such languages as JavaScript that are commonly used when making API requests.

However, the difference between SOA web services and APIs isn't defined by protocols and data formats because they are not used consistently between the two. The difference lies in the intent behind APIs and SOA services. One key difference is in their economics.

The reusable SOA

In SOA programs, service exposure was about exposing each business function so that it could be reused as much as possible. This way, each new project didn't have to go through the pain of performing integration to the back-end system again. The typical consumers were internal applications that attempted to put fresh user interfaces onto older systems of record. At the time, integration was difficult and took a significant portion of an IT project's budget. If you could make all of the core functions of the company available over reusable interfaces, you could significantly cut project costs. SOA was about cost saving, not generating new revenue.

APIs had a different starting point, with the assumption that integration was already simplified. This simplification occurred either through an earlier SOA initiative or by upgrading back-end systems to provide more ready-to-use modern interfaces. The new challenge is to craft an appealing interface to potential consumers. APIs are designed for the context in which they are likely to be

used. For example, they are ideally suited to provide the data that is required by a particular type of mobile application.

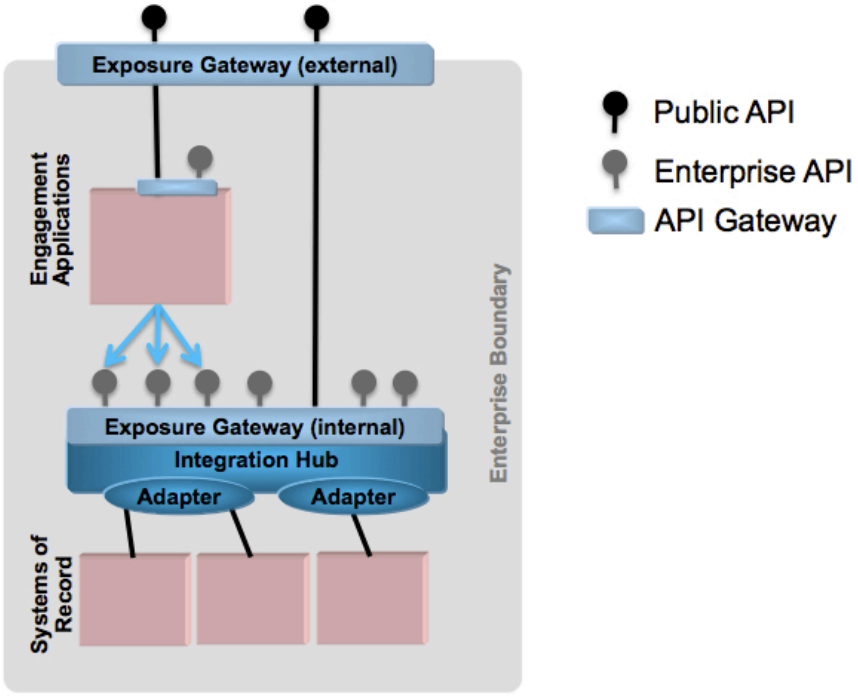
The dawn of API management

API popularity skyrocketed with the rise in smartphone use. Smartphones run rich client-side applications, creating a disruptive new business channel. As a result, application developers needed simple access to back-end functions and data; they needed APIs. APIs became a saleable product, with API providers competing against each other for the attention of developers. The focus of APIs isn't on reuse and cost saving as it was in SOA. Rather, the focus is on consumability and competing in the API economy. APIs are a saleable product.

This change in dynamic altered the technical requirements for APIs compared to SOA services. APIs needed sophisticated portals so that developers could discover and experiment with the APIs. They also needed mechanisms for developers to register to use and pay for the API. API providers needed the ability to set up payment plans to accommodate the various API usage rates. Because APIs are exposed publically, the exposing gateway needed strong security capabilities. All of these features needed to be self-service, and above all, simple. This change introduced a whole new type of IT capability now known as *API management*.

To this point, the focus has been on APIs as something to expose publically to external consumers; the dividing line between APIs and internal SOA services has been clear. With the maturing of API management technologies, APIs have brought about such benefits as ease of use and self-administration. As a result, many companies now want to also use API technologies and protocols to expose services inside the company as shown in Figure 3. The lines between SOA web services and API are now blurred and almost irrelevant. They have differences in their origin, to whom they are exposed to, and the data models they use, but many SOA "services" could also be potentially described as internal APIs.

Figure 3. Exposing APIs internally and externally



Today, the term *APIs* is commonly used to refer to any interface that is exposed over REST (HTTP/JSON) or a web service (SOAP/HTTP). The APIs are typically categorized by their scope, such as a public API or enterprise API. Enterprises that sustained SOA initiatives sometimes retain the term "service" for internal, enterprise-wide APIs. For more information about the differences between SOA and API, see *Integration architecture: Comparing web APIs with service-oriented architecture and enterprise application integration*.

The term *API* represents an evolution in the "service exposure" aspect of SOA. It uses simpler protocols and provides more sophistication around the exposure itself, including developer portals, policy controls, and self-administration.

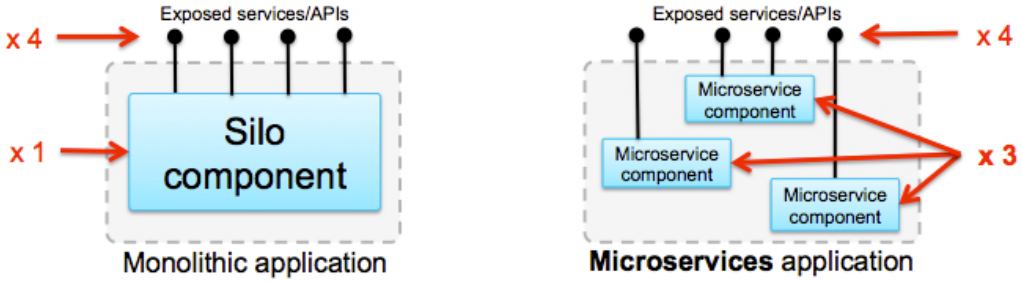
Microservices: An alternative architecture

Before you look at a comparison of microservices and SOA, you need to understand what a microservices architecture means. From a fundamental point, microservices are an alternative architecture for building *applications*. They offer a better way to decouple components *within* an application boundary. In fact, if microservices were called "micro components," their true nature would be clearer.

The boundaries of the application remain the same. As shown in Figure 4, despite being broken into separate microservice components on the inside, the application might still look the same from the outside. The number and granularity of APIs that a microservice-based application exposes should not be any different than if the API was built as a siloed application. The prefix "micro" in

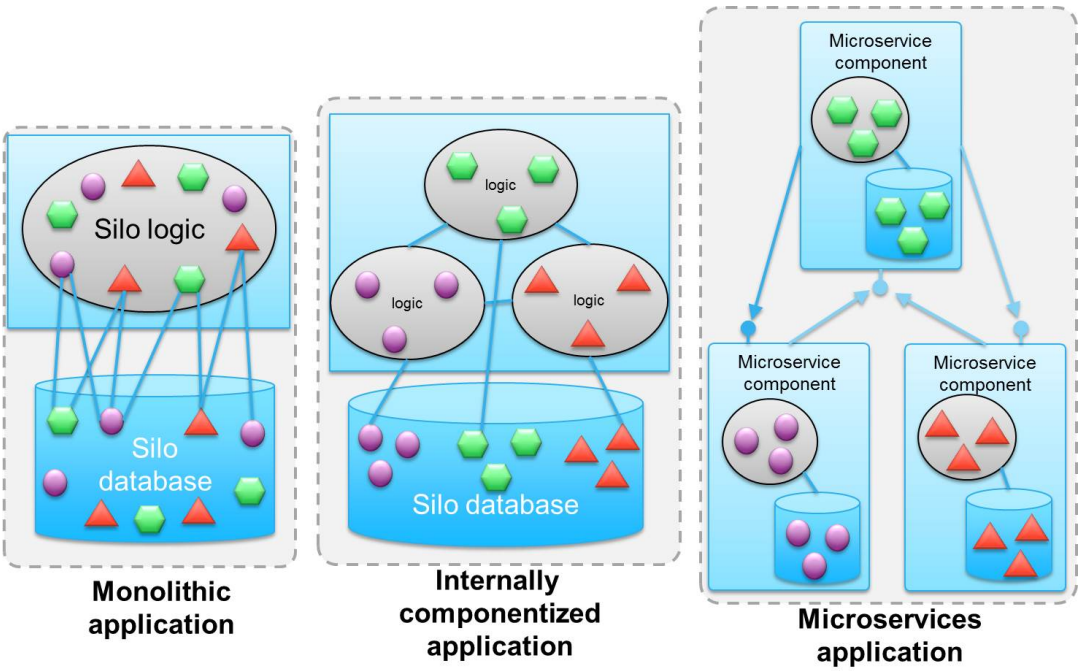
microservice refers to the granularity of the internal components, not the granularity of the exposed interfaces.

Figure 4. Microservices application exposing the same interfaces on the application boundary as the silo



Logically separating components within an application is not new. A host of different technologies has been developed over the years to enable clean separation of the parts of an overall application. Application servers can run multiple application components within them for a long time as shown by the middle image in Figure 5. Microservices go one step further by making the isolation between those application components absolute. They become separately running processes on the network as shown on the right side in Figure 5. To achieve decoupling, you should also partition your data model to align with the microservices.

Figure 5. From monolithic applications to microservices



The benefits of microservices

Fully independent microservice components enable completely autonomous ownership, resulting in the following benefits:

- **Agility and productivity**. The team that is developing the microservice can completely understand the codebase. They can build, deploy, and test it independently of other components in much faster iteration cycles. Because the microservice component is simply another component on the network, you can write it in the best suited language or framework for the required functionality and the most appropriate persistence mechanism.
  This approach can significantly reduce the amount of code to write and make it dramatically simpler to maintain. It ensures that teams can take on new technologies or versions of existing technology as needed rather than waiting for the rest of the application domain to catch up. For some definitions of microservice granularity, a microservice component should be simple enough that it can be rewritten in its next iteration if it makes sense to do so.
- **Scalability**: The microservices development team can scale the component at run time independently of other microservice components, enabling efficient use of resources and rapid reaction to changes in workload. In theory, the workload of a component can be moved to the most appropriate infrastructure for the task. It can also be relocated independently of the rest of the components to take advantage of network positioning. Well-written microservices offer extraordinary on-demand scalability, which was demonstrated by early innovators and adopters in this space. These microservices are also best placed to take

advantage of the elastic capabilities of cloud-native environments that have cost-effective access to enormous resources.
- **Resilience**: The separate run time immediately provides resilience that is independent of failures in other components. With a carefully decoupled design, such as avoiding synchronous dependencies and using circuit breaker patterns, each microservice component can be written to satisfy its own availability requirements without imposing those requirements across the application domain. Technologies, such as containers, and lightweight run times have enabled microservice components to fail quickly and independently, instead of taking down whole areas of unrelated functionality. Equally they are written in a highly stateless fashion so that they can immediately redistribute workloads and almost instantaneously bring up new run times.

These examples of benefits capture some of the most common reasons that organizations are turning to microservices.

Key factors to consider when choosing microservices

Before you decide whether to write applications as microservices, you must understand the following factors to ensure that your organization is prepared to handle them:

- **New patterns of technology**. Microservices are a radically different approach to building applications. Because they are on the network, they require a whole new set of components on the network alongside them. The enabling technologies exist, including service discovery, workload orchestration, container management, and logging frameworks. However, you must pull them into one coherent set, which takes significant experimentation, skill, and learning. You must determine what constitutes the perfect setup for microservices for your requirements, which might be different than those of other enterprises.
- **Application suitability**. Microservices aren't right for every application. One paradox in the microservices community at the moment is that you do not gain any advantage by wading in the murky waters of microservices for a new, relatively straightforward application, with a highly cohesive data model. Also, it is a huge undertaking to refactor a complex existing application into a microservices architecture.
  If not on old or new applications, when would you use microservices? One recommendation is not to use microservices until the evolution of a traditionally written application starts to reach an inflexion point of complexity. However, for this approach to work, you need to write a suitably structured application from the beginning and choose the right moment to make the transition.
- **Different design paradigms**. The microservices application architecture requires different approaches to design. To get the best results from the microservices approach, you might need to:
  - Accept eventual consistency models, rather than the transactional interactions that you are used to.
  - Understand how to work with event-sourced applications with no central operational data store.

  You also need to:

    - Ensure that your application logic is state free if it needs to take advantage of the significant rapid scalability benefits.
    - Become familiar with the subtle potential side effects of asynchronous communication if you decouple yourself from downstream components.
    - Understand the logic implications of implementing circuit breaker patterns.
    - Recognize the error handling limitations of HTTP/JSON communication compared to in-process communication.
    - Consider network latency in chained interactions.
- **DevOps maturity**. Microservices require a mature delivery capability. Continuous integration, deployment, and fully automated tests are a must. The developers who write code must be responsible for it in production. Build and deployment chains need significant changes to provide the right separation of concerns for a microservices environment.

If you're comfortable with these factors, you might be in a position to realize big benefits from a microservices application architecture.

How microservices fit within the SOA picture and the integration challenge

If our mental model of SOA is focused on the integration aspects, microservices are completely separate. It is an alternative way of writing the applications to which the integration architecture is trying to connect as shown in Figure 1.

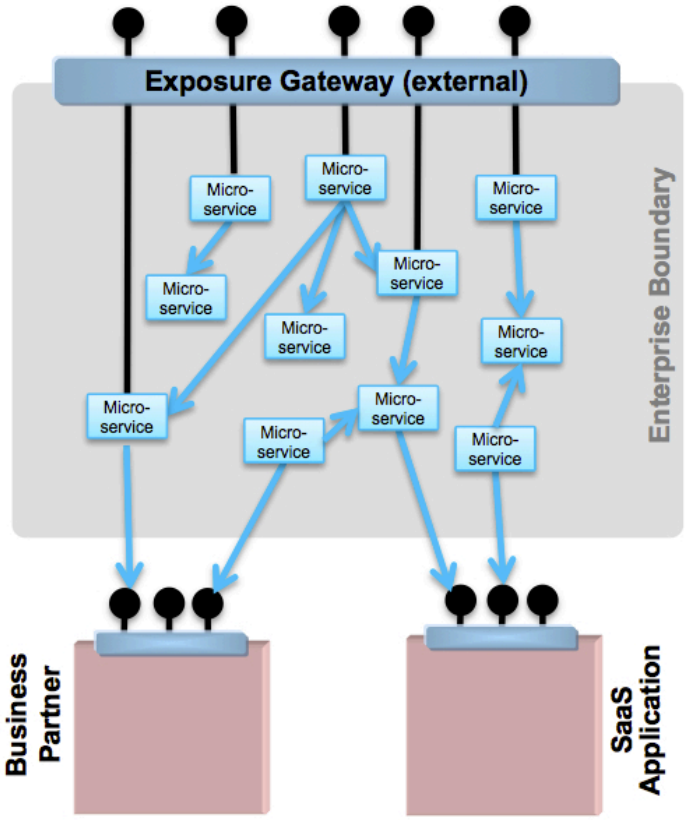
However, if our mental model of SOA is focused on relandscaping the applications into more business meaningful "service components," the service components that are shown on the right side in Figure 2 can start to look more like microservice components. A microservices architecture can now be seen as an evolution of SOA. To illustrate this point, let's compare two ends of the spectrum.

First, consider a fresh start-up company with a new idea for an entirely online product, such as social media or trading. Because it is starting with no existing architecture to work around, the company must create a suite of new applications to fulfil the unique aspects of the business. It might then choose to outsource the parts of the business that are not its core of adding value and use software-as-a-service (SaaS) applications, for example, for customer relationship management functions.

The company's landscape might be largely created from scratch. The primary focus might be on its ability to rapidly add new functions, with minimal downtime in a constantly available environment (the idea of a green field). The company might want to scale elastically (that is, scale both up and down) in line with unpredictable customer demand. It might want to provide a round-the-clock, resilient, highly available online presence.

The microservices architecture is a logical choice for much of the company's landscape as shown in Figure 6.

Figure 6. Microservices architecture within a green field site



The new applications can live within a single microservices framework that provides nonfunctional capabilities, such as scalability, availability, and resource management. You can expect low-level integration concerns to be minimal because all microservice components and SaaS applications that are involved use common protocols, such as HTTP/JSON APIs for communication. One key objective of an SOA exposing valuable functionality so that it can be combined and used simply across the enterprise is largely present. In this example, the lines between a well-implemented SOA and microservices architecture are blurred. If you imagine a perfect implementation of an SOA, it might look something like this example, but only a new company can create such a homogeneous architecture.

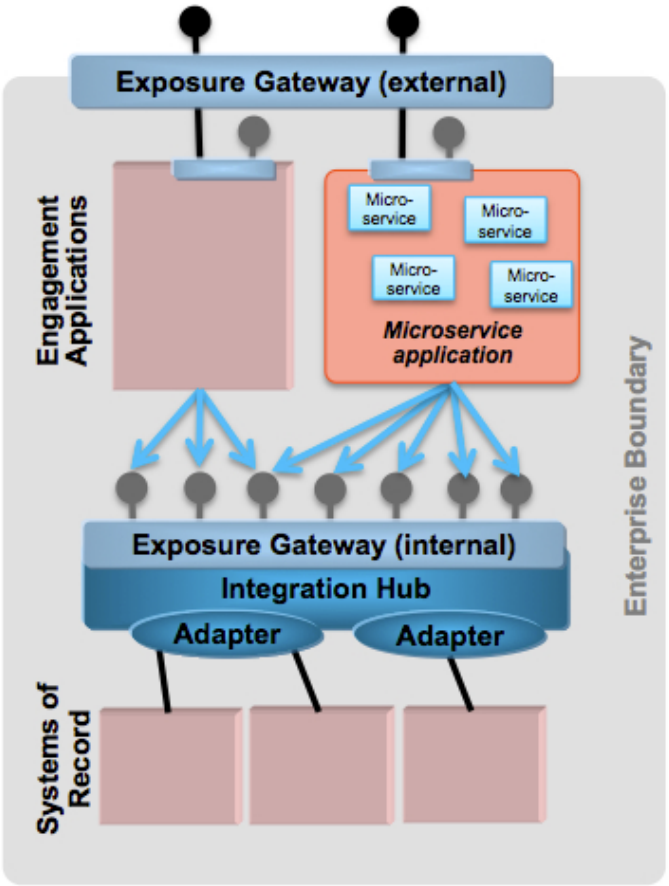
This article does not address whether an SOA "service component" is equivalent in size to a microservice component. The granularity of microservice components and how they are grouped is another debate entirely.

Now consider an opposite example of a large enterprise that has been growing and acquiring its IT landscape over many decades. This enterprise, which might be a traditional bank or insurance company, can have hundreds or even thousands of significant applications that are built from technologies that stretch back over decades. The enterprise might have strong divisions within the company, such as between healthcare, pensions, and general insurance, or between retail and investment banking. Each business unit might have independent applications that are dedicated to their core business. The divisions might also have a suite of applications, such as for human resources, that might be shared wherever possible.

The company has likely grown by acquisition or merging with competitors. Within the landscape, you find much duplication of data across applications. Customer accounts might spread across many systems, depending on which original company they were with. Correlation of the same customer in multiple systems might not be straightforward. These back-end applications are typically difficult to change internally. In this environment, SOA has an enormous task to reimagine the back-end systems into something more useful for future business requirements.

The integration challenge is also complex. It might call for integration tools, as shown in Figure 7, to enable data and functions from back-end applications to be accessed despite the challenges with protocols, transport, and data formats. For largely historical reasons, this integration exercise is often labeled "SOA," although it is focused on only half of the SOA challenge. It is labeled SOA because integration is the first area that most SOA initiatives tackle. In many cases, this is all that they achieve with the available funding.

Figure 7. Large enterprise with microservices as part of the landscape



However, a further aspect that companies need to achieve with SOA is reshaping the data and functions into more business-centric capabilities. They need to determine how to satisfy new channels, such as mobile, that require a radically different granularity of service to traditional applications. To achieve these aspects, companies need responsiveness, availability, and scalability that might not be available in current systems. Applications must be written to satisfy

these new channels in a style that enables rapid agile change, provides for extreme scalability, and offers superior availability.

The attraction of using a microservices architecture for these new applications is easy to see. As shown in Figure 7, the initial usage of microservices in large enterprises is focused on new systems-of-engagement applications. The SOA concept might be tainted by early integration-centric efforts. Therefore, microservices are often viewed as being separate from SOA, providing greater agility, scalability, and responsiveness, but in many cases, relying on the ground work of the integration phase of SOA.

Combining microservices, SOA, and APIs in the future

From an architectural point of view, SOA has three key elements:

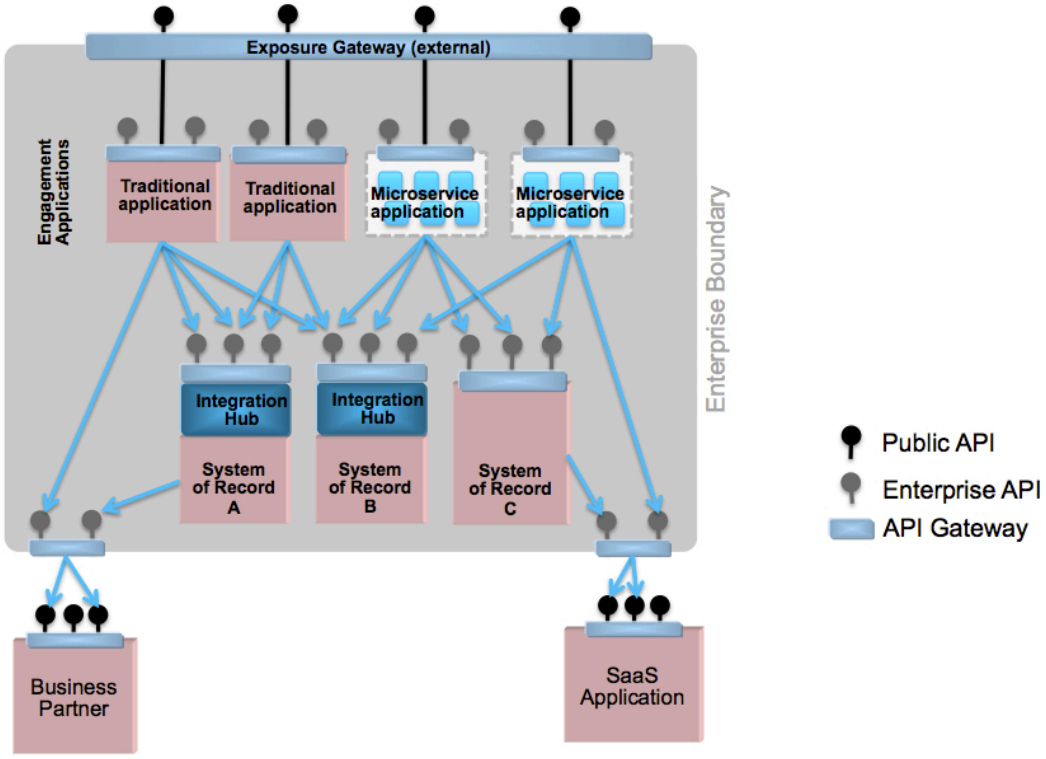
- **Deep integration** to enable aging systems to expose their data and functions to be surfaced by using an interface
- **Service exposure** to standardize and simplify the way that those interfaces are made available to broader audiences
- **Services components** to further compose interfaces into more valuable business usable assets

These three elements are still present in future architectures, but they are necessarily distributed across the landscape as shown in in Figure 8.

Deep integration

Some systems still require the deep integration capabilities that are provided by integration hubs to expose their underlying functions and data as APIs. Other systems might be able to provide APIs directly when they are upgraded to newer versions. The key difference starts with where SOA tended to draw deep integration capabilities into a centralized function. The more advanced tools and techniques should enable integration to be federated more often to application owners as shown by the placement of the integration hubs in Figure 8.

Figure 8. Microservices, SOA, and APIs combined



Service exposure

Going forward, all systems need to provide APIs if they are to remain relevant. Application-level APIs need a lightweight layer of control as illustrated by the API gateways in Figure 8. This layer of control is an evolution of the service exposure concept from SOA. It has morphed into the much broader and decentralized API exposure.

The API gateway and management capability might be a common resource across the enterprise. It is "decentralized" in the sense that application teams can self-publish APIs, and equally self-subscribe to the APIs they need, without needing an extra team. You can gain the benefits of standardized mechanisms for traffic management and monitoring, logging, auditing, and security in a standardized way throughout the enterprise, while retaining the agility required by the business. These same API gateways might also be used to help govern interaction with business partners and external SaaS capabilities.

Service components

Traditional, more siloed applications are still appropriate for some implementations. However, microservices provide an alternative means for building some classes of applications, providing agility, scalability, and resilience that traditional applications cannot. Microservices applications are most common in the engagement layer, where their specific characteristics are most in need, enabling the creation of new channel-specific capabilities and Internet-facing APIs.

Conclusion

At least two different perspectives exist on what SOA was intended to achieve. A direct comparison between SOA and microservices architecture is likely to be fraught with difficulties. The concepts of an SOA are present in modern architectures but have evolved in several ways. Integration tools, patterns, and standards have evolved so that functions and data are more easily brought to the surface. Service exposure has evolved into APIs, simplifying exposure, consumption, management, and, in some cases, monetizing business functions. New application architectures, including the microservices architecture, enable developers to focus more closely on business logic, continuously pushing infrastructural detail to the environment in which they run. The combination of these developments enables solutions to be built in more agile styles and applications to benefit from new levels of elastic scalability and fault tolerance.

Acknowledgements

Thank you to the following people for their input and review of the material in this article: Andy Garratt, Andy Gibbs, Carlo Marcoli, and Brian Petrini.

Resources

- A seminal article on microservices and its relationship to SOA by James Lewis and Martin Fowler: *Microservices: A definition of this new architectural term*
- A related presentation from MuCon London 2015: *Where is integration in a microservices world*.
- A comparison of SOA and APIs in relation to traditional integration by Kim J. Clark: *Integration architecture: Comparing web APIs with service-oriented architecture and enterprise application integration*
- Information about microservices, APIs, and SOA from developerWorks:
  - Microservices
  - IBM API Management tutorials
  - The API explorer
  - API Management developer community
  - Service-oriented architecture
- News and information about IBM WebSphere and related products: developerWorks WebSphere zone.

About the author

Kim J. Clark

Kim J. Clark is a strategist on IBM's integration portfolio providing guidance to the offering management team on current trends and challenges. He has spent the last two decades working in the field, implementing solutions that are focused on SOA, BPM, and APIs.