Microservices verses
Service-Oriented
Architecture

Mark Richards
Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>v</td>
</tr>
<tr>
<td>1. The World of Service-Based Architectures</td>
<td>7</td>
</tr>
<tr>
<td>Service Contracts</td>
<td>8</td>
</tr>
<tr>
<td>Service Availability</td>
<td>11</td>
</tr>
<tr>
<td>Security</td>
<td>13</td>
</tr>
<tr>
<td>Transactions</td>
<td>14</td>
</tr>
<tr>
<td>Too Much Complexity?</td>
<td>16</td>
</tr>
<tr>
<td>2. Comparing Service Characteristics</td>
<td>17</td>
</tr>
<tr>
<td>Service Taxonomy</td>
<td>17</td>
</tr>
<tr>
<td>Service Ownership and Coordination</td>
<td>21</td>
</tr>
<tr>
<td>Service Granularity</td>
<td>23</td>
</tr>
<tr>
<td>Granularity and Pattern Selection</td>
<td>25</td>
</tr>
<tr>
<td>3. Comparing Architecture Characteristics</td>
<td>27</td>
</tr>
<tr>
<td>Component Sharing</td>
<td>28</td>
</tr>
<tr>
<td>Service Orchestration and Choreography</td>
<td>30</td>
</tr>
<tr>
<td>Middleware vs. API Layer</td>
<td>36</td>
</tr>
<tr>
<td>Accessing Remote Services</td>
<td>39</td>
</tr>
<tr>
<td>4. Comparing Architecture Capabilities</td>
<td>41</td>
</tr>
<tr>
<td>Application Scope</td>
<td>41</td>
</tr>
<tr>
<td>Heterogeneous Interoperability</td>
<td>43</td>
</tr>
<tr>
<td>Contract Decoupling</td>
<td>45</td>
</tr>
<tr>
<td>5. Summary</td>
<td>49</td>
</tr>
</tbody>
</table>
A. About the Author. .................................................. 51
Preface

In the mid-2000’s Service-Oriented Architecture (SOA) took the IT industry by storm. Numerous companies adopted this new architecture style as a way to bring better business functionality reuse into the organization and to provide a foundation that allowed the IT organization and the business to better communicate and collaborate with each other. Dozens of best practices for implementing SOA emerged during this time, as well as a plethora of third-party products to help companies implement Service-Oriented Architecture. Unfortunately, companies learned the hard way that SOA was a big, expensive, complicated architecture style that took too long to design and implement, resulting in failed projects that drove SOA out of favor in the industry.

Today Microservices is taking the IT industry by storm as the go-to architecture style used to develop highly scalable and modular applications. Microservices holds the promise of being able to address some of the problems associated with large, complex SOA architectures as well as the problems found with big, over-bloated monolithic applications. But how much different is Microservices from Service-Oriented Architecture? Is the industry destined to repeat the same experience with Microservices as with SOA?

In this report I will walk you through a detailed comparison of the Microservices and SOA architecture patterns. You will learn the basics of each of these architectures and core differences between them in terms of the architecture style, architecture characteristics, service characteristics, and the capability differences between them. By using the information in this report you will gain the knowledge necessary to know how these two architecture styles differ from
each other and which architecture style is best suited for your par-
ticular situation.
Both Microservices Architecture and Service-Oriented Architecture (SOA) are considered service-based architectures, meaning they are architecture patterns that place a heavy emphasis on services as the primary architecture component used to implement and perform business and non-business functionality. Although Microservices and SOA are very different architecture styles, they share many common characteristics.

One thing all service-based architectures have in common is that they are generally distributed architectures, meaning service components are accessed remotely through some sort of remote access protocol (e.g., REST, SOAP, AMQP, JMS, MSMQ, RMI, .NET Remoting, etc.). Distributed architectures offer significant advantages over monolithic and layered-based architectures, including better scalability, better decoupling, and better control over development, testing and deployment. Components within a distributed architecture tend to be more self-contained, allowing for better change control and easier maintenance, which in turn leads to more robust and more responsive applications. Distributed architectures also lend themselves toward more loosely coupled and modular architectures.

In the context of service-based architecture, modularity is the practice of encapsulating portions of your application into self-contained services that can be individually designed, developed, tested, and
deployed with little or no dependency on other components or services in the application. Modular architectures also support the notion of favoring rewrite over maintenance, allowing architectures to be refactored or replaced in smaller pieces over time as the business grows as opposed to replacing or refactoring an entire application using a big-bang approach.

Unfortunately, very few things in life are free, and the advantages of distributed architectures is no exception. The tradeoffs associated with those advantages are primarily increased complexity and cost. Maintaining service contracts, choosing the right remote access protocol, dealing with non-responsive or unavailable services, securing remote services, and managing distributed transactions are just a few of the many complex issues you have to address when creating service-based architectures. In this chapter I’ll describe some of these complex issues as they relate to Service-Based Architecture.

**Service Contracts**

A *service contract* is an agreement between a service (usually remote) and a service consumer (client) that specifies the inbound and outbound data along with the contract format (e.g., XML, JSON, Java Object, etc.). Creating and maintaining service contracts is a difficult task, one that should not be taken lightly or treated as an afterthought. As such, the topic of service contracts deserves some special attention in the scope of service-based architecture.

In service-based architecture there are two basic types of service contract models you can use - service-based contracts and consumer-driven contracts. The real difference between these contract models is that of collaboration. With service-based contracts, the service is the sole owner of the contract, and is generally free to evolve and change the contract without considering the needs of the service consumers. This model forces all service consumers to adopt new service contract changes, regardless whether the service consumers need or want the new service functionality.

Consumer-driven contracts, on the other hand, are based on a closer relationship between the service and the service consumers. With this model there is strong collaboration between the service owner and the service consumers so that needs of the service consumers are taken into account in terms of the contracts between them. This type of model generally requires the service to know who it's con-
sumers are and how the service is used by each service consumer. Service consumers are free to suggest changes to the service contract, which the service may either adopt or reject depending on how it affects other service consumers. In a perfect scenario service consumers deliver tests to the service owner so that if one consumer suggests a change, tests can be executed to see if the change breaks another service consumer. Open source tools such as Pact (https://github.com/realestate-com-au/pact) and Pacto (http://thoughtworks.github.io/pacto) can help with maintaining and testing consumer-driven contracts.

Another critical topic within the context of service contracts is that of contract versioning. Let's face it - at some point the contracts binding your services and service consumers are bound to change. The degree and magnitude of this change is largely dependent on how those changes affect each service consumer and the backward compatibility supported by the service with respect to the contract changes.

Contract versioning allows you to roll out new service features that involve contract changes while at the same time providing backwards compatibility for service consumers that are still using prior contracts. Perhaps one of the most important words of advice in this chapter is to plan for contract versioning from the very start of your development effort, even if you don't think you'll need it - because eventually you will. While there are several open source and commercial frameworks available to help you manage and implement contract versioning strategies, there are two basic techniques you can use to implement your own custom contract versioning strategy - homogeneous versioning and heterogeneous versioning.

Homogeneous versioning involves using contract version numbers in the same service contract. Notice in Figure 1-1 that the contract used by service consumer A and service consumer B are both the same circle shape (signifying the same contract) but contain different version numbers. A simple example of this might be an XML-based contract that represents an order for some goods with a contract version number 1.0. Let's say a newer version (version 1.1) is released containing an additional field used to provide delivery instructions in the event the person is not at home when the order is delivered. In this case the original contract (version 1.0) can remain backwards compatible by making the new delivery instructions field optional.
Heterogeneous versioning involves supporting multiple types of contracts. This technique is closer to the concept of consumer-driver contracts described earlier in this section. With this technique, as new features are introduced, new contracts are introduced as well that support that new functionality. Notice the difference between Figure 1-1 and Figure 1-2 in terms of the service contract shape. In Figure 1-2, service consumer A communicates using a contract represented by a circle, whereas service consumer B uses an entirely different contract represented by the triangle. In this case backwards compatibility is supplied by different contracts rather than versions of the same contract. This is a common practice in many JMS-based messaging systems, particularly those leveraging the ObjectMessage message type. For instance, a Java-based receiver can interrogate the payload object sent through the message using the instanceof keyword and take appropriate action based on the object type. Alternatively, XML payload can be sent through a JMS TextMessage that contains entirely different XML schema for each contract, with a message property indicating the corresponding XML schema associated with the XML payload.
Providing backwards compatibility is the real goal of contract versioning. Maintaining a mindset that services must support multiple versions of a contract (or multiple contracts) will allow your development teams to quickly deploy new features and other changes without fear of breaking the existing contracts with other service consumers. Keep in mind that it is also possible to combine these two techniques by supporting multiple version numbers for different contract types.

One last thing about service contracts with respect to contract changes - be sure to have a solid service consumer communication strategy in place from the start so that service consumers know when a contract changes or a particular version or contract type is no longer supported. In many circumstances this may not be feasible due to a large number of internal and/or external service consumers. In this situation an integration hub (i.e., messaging middleware) can help by providing an abstraction layer to transform service contracts between services and service consumers. I’ll be talking more about this capability later in this report under the Contract Decoupling section in Chapter 3.

**Service Availability**

Service availability and service responsiveness are two other considerations common to all service-based architectures. While both of these topics relate to the ability of the service consumer to communicate with a remote service, they have slightly different meanings and are addressed by service consumers in different ways.

_Service availability_ refers to the ability of a remote service to accept requests in a timely manner (e.g., establishing a connection to the remote service). _Service responsiveness_, on the other hand, refers to the ability of the service consumer to receive a timely response from the service. The diagram in figure 1-3 illustrates this difference.
Although the end result of these error conditions is the same (the service request cannot be processed), they are handled in different ways. Since service availability is related to service connectivity, there is not much a service consumer can do except to retry the connection for a set number of times or queue the request for later processing if possible.

Service responsiveness, on the other hand, is much more difficult to address. Once you successfully send a request to a service, how long should you wait for a response? Is the service just slow, or did something happen in the service preventing the response from being sent?

Addressing timeout conditions is perhaps one of the most challenging aspects of remote service connectivity. A common way of determining reasonable timeout values is to first establish benchmarks under load to get the maximum response time, and then add extra time to account for variable load conditions. For example, let's say you run some benchmarks and find that the maximum response time for a particular service request is 2000 milliseconds. In this case you might double that value to account for high load conditions, resulting in a timeout value of 4000 milliseconds.

While this may seem like a reasonable solution to calculate a service response timeout, it is riddled with problems. First of all, if the service really is down and not running, every request must wait 4 seconds before determining that the service is not responding. This is inefficient and annoying to the end user of the service request. The other problem is that your benchmarks may not have been accurate, and under heavy load the service response is actually averaging 5 seconds rather than the 4 seconds you calculated. In this case the
service is in fact responding, but the service consumer will reject every request due to the timeout value being set too low.

A popular technique to address this issue is to use the circuit breaker pattern. If the service is not responding in a timely manner (or not at all), a software circuit breaker will be thrown so that service consumers don’t have to waste time waiting for timeout values to occur. The cool thing is that unlike a physical circuit breaker, this pattern can be implemented to reset itself when the service starts responding or becomes available. There are numerous open-source implementations of the circuit breaker pattern, including Ribbon from Netflix. You can read more about the circuit breaker pattern in Michael Nygard’s book *Release It!*

When dealing with timeout values try to avoid the use of global timeout values for every request. Instead, consider using context-based timeout values and always make these externally configurable so that you can respond quickly for varying load conditions without having to rebuild or redeploy the application. Another option is to create “smart timeout values” embedded in your code that can adjust themselves based on varying load conditions. For example, the application could automatically increase the timeout value in response to heavy load or network issues. As load decreases and response times become faster, the application could then calculate the average response time for a particular request and lower the timeout value accordingly.

**Security**

Because services are generally accessed remotely in service-based architectures, it is important to make sure the service consumer is allowed to access a particular service. Depending on your situation, service consumers may need to be both authenticated and authorized. Authentication refers to whether the service consumer can connect to the service, usually through sign-on credentials using a username and password. In some cases authentication is not enough - just because a service consumer can connect to a service doesn’t necessarily mean they can access all of the functionality in that service. Authorization refers to whether or not a service consumer is allowed to access specific business functionality within a service.

Security was a major issue with early Service-Oriented Architecture implementations. Functionality that used to be located in a secure

---

"Security | 13"
silo-based application was suddenly available globally to the entire enterprise. This issue created a major shift in how we think about services and how to protect them from consumers who should not have access to them.

With Microservices, security becomes a challenge primarily due to the lack of a middleware component to handle security-based functionality. Instead, each service must handle security on its own, or in some cases the API layer can be made more intelligent to handle the security aspects of the application. One security design I have seen implemented in Microservices that works well is to delegate authentication to a separate service and place the responsibility for authorization in the service itself. While this design could be modified to delegate both authentication and authorization to a separate security service, I prefer encapsulating the authorization in the service itself to avoid chattiness with a remote security service and to create a stronger bounded context with fewer external dependencies.

**Transactions**

Transaction management is a big challenge in service-based architectures. Most of the time when we talk about transactions we are referring to the ACID transactions (atomicity, consistency, isolation, and durability) found in most business applications. ACID transactions are used to maintain database consistency by coordinating multiple database updates within a single request so that if an error occurs during processing, all database updates are rolled back for that request.

Given that service-based architectures are generally distributed architectures, it is extremely difficult to propagate and maintain a transaction context across multiple remote services. As illustrated in Figure 1-4, a single service request (represented by the box next to the red X) may need to call multiple remote services to complete the request. The red X in the diagram indicates that it is not feasible to use an ACID transaction in this scenario.
Transaction issues are much more prevalent in Service-Oriented Architecture because, unlike Microservices architecture, multiple services are typically used to perform a single business request. I discuss this in more detail in the Service Orchestration section of Chapter 3.

Rather than use ACID transactions, service-based architectures rely on something called BASE transactions. BASE is a family of styles that include basic availability, soft state, and eventual consistency. Distributed applications relying on BASE transactions strive for eventual consistency in the database rather than consistency at every transaction. A classic example of BASE transactions is making a deposit into an ATM machine. When you deposit cash into your account through an ATM machine, it may take anywhere from several minutes to several hours for your money to appear in your account. In other words, there is a soft transition state where the money has left your hands but has not reached your bank account. We are tolerant of this time lag and rely on soft state and eventual consistency, knowing and trusting that the money will reach our account at some point soon. Batch jobs also sometimes rely on eventual consistency when seen from a holistic system view.

Switching to the world of service-based architectures requires us to change our way of thinking about transactions and consistency. In situations where you simply cannot rely on eventual consistency and soft state and require transactional consistency, you can make your services more coarse-grained to encapsulate the business logic into a single service, allowing the use of ACID transactions to achieve consistency at the transaction-level. You can also leverage event-driven techniques to push notifications to consumers when the state of a request has become consistent. This technique adds a significant amount of complexity to an application, but helps in managing transactional state when using BASE transactions.
Too Much Complexity?

Although service-based architectures are a significant improvement over monolithic applications, as you can see there are a lot of things to consider, including service contracts, availability, security, and transactions (to name a few). Unfortunately, very few things in life are free, including the tradeoffs associated with moving to a service-based architecture approach such as Microservices and SOA. For this reason, you shouldn't embark on a service-based architecture solution unless you are ready and willing to address the many issues facing distributed computing.

While the issues identified in this chapter are complex, they certainly aren't showstoppers. Most teams using service-based architectures are able to successfully address and overcome these challenges through a combination of open source tools, commercial tools, and custom solutions.

Are service-based architectures complex? Absolutely. However, with added complexity comes additional characteristics and capabilities that will make your development teams more productive, produce more reliable and robust applications, reduce overall costs, and improve overall time-to-market. In the next three chapters I will walk you through those capabilities by comparing Microservices and SOA so you can better decide which architecture pattern is right for you.
The OASIS Reference Model for Service Oriented Architecture (https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm) defines a service as “a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description”. In other words, a service has some business capabilities that has a well-defined interface and well-defined contract to access that service. What this definition does not specify, however, is how services are further defined based classification, organizational ownership, and granularity (i.e., service size). Understanding these service characteristics helps define the context of a service within a particular architecture pattern.

Although Microservices and SOA both rely on services as the main architecture component, they vary greatly in terms of service characteristics described above. In this chapter I will compare Microservices and SOA by focusing on how the services are classified within each pattern (i.e., service taxonomy), how services are coordinated based on the service owner, and finally the difference in service granularity between Microservices and SOA.

Service Taxonomy

The term service taxonomy refers to how services are classified within an architecture. There are two basic types of service classifications - service type and business area. Service type classifica-
tion refer to the type of role the service plays in the overall architecture. For example, some services might implement business functionality, whereas other services might implement some sort of non-business functionality such as logging, auditing, and security. Business area classification refers to the role a business service plays with regard to a specific business functional area such as reporting, trade processing, order shipping, etc.

Service type classification is generally defined at the architecture pattern level, whereas business area classification is defined at the architecture implementation level. While architecture patterns provide a good base and starting point for defining the service types, as an architect you are free to modify these and come up with your own classifications. In this section we will be focusing on architecture patterns and therefore the types of services that you would generally find in Microservices and SOA.

Microservices architectures have a limited service taxonomy when it comes to service type classification, mostly consisting of only two service types as illustrated in Figure 2-1. Functional services are those services that support specific business operations or functions, whereas infrastructure services support non-functional tasks such as authentication, authorization, auditing, logging, and monitoring. The reason this is an important distinction within a Microservices architecture is because infrastructure services are not exposed to the outside world, but rather treated as private shared services only available internally to other services. Functional services, on the other hand, are accessed externally and are generally not shared with any other service.
The service taxonomy within Service-Oriented Architecture varies significantly from Microservices. In SOA there is a very distinct and formal service taxonomy in terms of the type of service and the role that service plays in the overall architecture. While there can be any number of service types within SOA, the architecture pattern defines 4 basic types of services as illustrated in Figure 2-2.

*Business services* are abstract, high-level coarse-grained services that define the core business operations that are performed at the enterprise-level. Being abstract, they are void of any implementation or protocol, and usually only include the name of the service, the expected input, and the expected output. Optionally these services types also can include processing steps or special orchestration rules associated with the service. Business services are typically repre-
sent through either XML, WSDL (Web Services Definition Lan-
guage) or BPEL (Business Process Execution Language). A good
litmus test for determining whether a service should be considered a
business service is to add the words “Are we in the business of” in
front of the context of the service name. For example, consider the
services ProcessTrade and InsertCustomer. Saying “Are we in the
business of processing trades” makes it clear that ProcessTrade is a
good business service candidate, whereas “Are we in the business of
inserting customers” is a clear indication that the InsertCustomer
service is not a good abstract business service candidate, but rather a
concrete service that is invoked as a response to a business service.

Enterprise services are concrete enterprise-level coarse-grained serv-
ces that implement the functionality defined by the business serv-
ces described above. As illustrated in Figure 2-2, it is usually the
middleware component that bridges the abstract business services
and the corresponding concrete enterprise services implementa-
tions. Enterprise services can have a one-to-one or one-to-many
relationship with a business service. They can be custom written
using any programming language and platform, or they can be
implemented using a third-party COTS (Commercial Off The Shelf)
product. One unique thing about enterprise services is that they are
generally shared across the organization. For example, a Retrieve-
Customer enterprise service may be used by different parts of the
organization to provide a common way to retrieve customer infor-
mation. CheckTradeCompliance, CreateCustomer, ValidateOrder, and
GetInventory are all good examples of enterprise services. Enterprise
services typically rely on application services and infrastructure serv-
ces to fulfill a particular business request, but in some cases all of
the business functionality needed for a particular request may be
self-contained within that enterprise service.

Application services are fine-grained, application-specific services
that are bound to a specific application context. Application services
provide specific business functionality not found at the enterprise-
level. For example, an auto quoting application as part of a large
insurance company might expose services to calculate auto insur-
ance rates - something that is specific to that application and not to
the enterprise. Application services may be called directly through a
dedicated user interface, or through an enterprise service. Some
examples of an application service might be AddDriver, AddVehicle,
and CalculateAutoQuote.
The final basic type of service found in Service-Oriented Architecture is *Infrastructure services*. Like Microservices, these are services that implement non-functional tasks such as auditing, security, and logging. In SOA, infrastructure services can be called from either application services or enterprise services.

Remember, as an architect you can choose to use the standard service types that are part of these architecture patterns, or completely discard them and create your own classification scheme. Regardless of what you do, the important thing is to make sure you have a well defined and well documented service taxonomy for your architecture.

**Service Ownership and Coordination**

A *service owner* is defined as the type of group within the organization that is responsible for creating and maintaining a service. Because Microservices has a limited service taxonomy (functional services and infrastructure services), it is typical for application development teams to own both the infrastructure and functional services. While there may be dozens of application development teams writing services, the important thing here to note is that it is all the same type of group (e.g., application development teams). Figure 2-3 illustrates the typical service ownership model for Microservices.

![Figure 2-3. Microservices Service Ownership Model](image)
With Service-Oriented Architecture, there are usually different service owners for each type of service. Business services are typically owned by business users, whereas enterprise services are typically owned by shared services teams or architects. Application services are usually owned by application development teams, and infrastructure services are owned by either application development teams or infrastructure services teams. Although not formally a service, the middleware components usually found in SOA are typically owned by integration architects or middleware teams. Figure 2-4 illustrates the typical service ownership model for SOA.

![Figure 2-4. SOA Service Ownership Model](image)

The significance of the service owner is that of overall service coordination. In SOA, you must coordinate with multiple groups to create or maintain a single business request; business users must be consulted about the abstract business services, shared services teams must be consulted about the enterprise services created to implement the business services, application development teams must be coordinated so that enterprise services can invoke lower-level functionality, and infrastructure teams must be coordinated to ensure non-functional requirements are met through the infrastructure services. Finally, all of that needs to be coordinated through the middleware teams or integration architects managing the messaging middleware.

With Microservices, there is little or no coordination between services to fulfill a single business request. If coordination is needed between service owners it is done quickly and efficiently through small application development teams.
This difference in service ownership and overall coordination between Microservices and SOA directly relates to the effort and time involved in developing, testing, deploying, and maintaining services in each of these architecture patterns. This is an area where Microservices stands out from the crowd. Due to the small service size and minimal coordination needed with other groups, services can be quickly developed, tested, and deployed through an effective deployment pipeline. This translates to faster time to market, less development and maintenance costs, and more robust applications.

**Service Granularity**

Perhaps one of the biggest differences from a services perspective between Microservices and SOA is service granularity. As the name suggests, Microservices is all about small, fine-grained services. More specifically, service components within a Microservices architecture are generally single-purpose services that do one thing really, really well. With Service-Oriented Architecture, service components can range in size anywhere from small application services to very large enterprise services. In fact, it is common to have a service component within SOA represented by a large product or even a subsystem.

Interestingly enough, one of the biggest challenges originally facing Service-Oriented Architecture was that of service granularity. Not understanding the impact of service granularity, architects frequently designed services that were too fine-grained, resulting in chatty and poor performing applications. Architects and component designers quickly learned that large, coarse-grained services with views into the data were the way to go. For example, rather than fine-grained getter and setter services like `GetCustomerAddress`, `GetCustomerName`, `UpdateCustomerName`, and so on, architects and shared services teams adopted an approach of having an enterprise `Customer` service that handled more coarse-grained update and retrieval data views, delegating the lower-level getters and setters to application-level services that were not exposed remotely to the enterprise. In this manner operations such as `GetCustomerDemographics` or `GetCustomerInformation` would return a bulk of customer data related to that context rather than each individual field.
This difference in granularity naturally relates to differences in service component scope and functionality. With Microservices, the service component functionality (what the service actually does) tends to be very small, sometimes implemented through only one or two modules. With SOA, services tend to encompass much more business functionality, sometimes implemented as complete subsystems (e.g., claims processing engines or warehousing systems). However, more typically SOA relies on multiple services to complete a single business request, whereas Microservices generally does not. I discuss this topic in more detail in the Service Orchestration section of the next chapter.

Regardless of whether you are using a Microservices architecture or SOA, designing services with the right level of granularity is not an easy task. Service granularity affects both performance and transaction management. Services that are too fine-grained will require inter-service communication between them to fulfill a single business request, resulting in numerous remote service calls that take up valuable time. For example, let’s say it takes 4 services to process a particular user request. Let’s also say that the time spent just on the remote access protocol to communicate to or from the service is 100 milliseconds. The diagram in Figure 2-5 shows that in this example 600 milliseconds would be spent just on transport time. Consolidating these services into a single service would reduce that transport time to just 200 milliseconds, shaving off close to half a second of processing time.

![Figure 2-5. Service granularity impact on performance](image)

Transaction management is also impacted by service granularity. What I am referring to here are traditional ACID transactions, not the BASE transactions I discussed in the previous chapter. If your remote services are too fine-grained, you will not be able to coordinate the services using a single transactional unit of work as shown on the top diagram of Figure 2-6. However, by combining these services into one larger remote service as shown by the bottom diagram in Figure 2-6, you can now use a transaction to coordinate the
services, thereby ensuring that database updates are all contained within a single transactional unit of work.

![Figure 2-6. Service granularity impact on transactions](image)

When dealing with service granularity I usually find it easier to start out with more coarse-grained services that you might otherwise create, and then break them apart as you learn how they are used. As Sam Newman states in his excellent book *Building Microservices* from O'Reilly, "Start with a small number of larger services first. " Just watch out for transaction issues and too much inter-service communication, particularly with Microservices - these are good indicators that your services might be too fine-grained.

**Granularity and Pattern Selection**

Out of the three service characteristics described in this chapter, service granularity perhaps has the most impact on your choice of which architecture pattern is best suited for your situation. The very small fine-grained service concept within Microservices allows this architecture pattern to improve all aspects of the software development lifecycle, including development, testing, deployment, and maintenance. While moving to more coarse-grained services cer-
tainly resolves performance and transactional issues, it also adversely affects development, testing, deployment, and maintenance. If you find your services range in size from small to large, you will likely need to look towards more of a Service-Oriented Architecture pattern than the more simple Microservices architecture pattern. However, if you are able to break down the business functionality of your application into very small, independent parts, then Microservices is a likely candidate for your architecture.

There are many other aspects to consider besides service characteristics when comparing Microservices to SOA. In the next chapter I will take more of a global view and compare the architectural aspects between Microservices and SOA, including the level of sharing between components, service orchestration and choreography, the use of middleware vs. a simple API layer, and finally differences in how remote services are accessed in each of these patterns.
A component is defined as a unit of software that has a well-defined interface and a well-defined set of roles and responsibilities. Components form the building blocks of the architecture. For Service-based Architectures those building blocks are usually referred to as services (or service components). Regardless of the label you put on a component, when creating an architecture you will need to determine how components are shared, how they communicate with each other, how they are combined to fulfill a particular business request, and how they are accessed from remote service consumers.

Determining all of this is not always an easy task. This is where architecture patterns come in. Each architecture pattern has a unique topology which defines the shape and general characteristics of the architecture, including how components relate, communicate, and act together to fulfill business requests. By analyzing the topology of the architecture pattern you can better determine whether the pattern is the right choice for you.

In this chapter I will explore the differences between Microservices and SOA in terms of the overall architecture topology and the defining characteristics of the architecture pattern. Specifically, I will focus on the level of service component sharing between Microservices and SOA, the level of service component communication between the two patterns, and how remote service components are typically accessed within each pattern. I will also dive into the differ-
ences between the messaging middleware found in the SOA architecture pattern and the optional API Layer found in the Microservices architecture pattern.

**Component Sharing**

Microservices and SOA are inherently different when it comes to sharing components. Service-Oriented Architecture is built on the concept of a *share-as-much-as-possible* architecture style, whereas Microservices is build on the concept of a *share-as-little-as-possible* architecture style. In this section we will explore the differences between these two concepts as they relate to Microservices and SOA.

Component sharing is one of the core tenants of SOA. As a matter of fact, component sharing is what enterprise services are all about. For example, consider a large retail company as illustrated in Figure 3-1 that has many applications associated with the processing of an order, such as a customer management system, warehouse management system, and order fulfillment system. All of these systems have their own version of an *Order* service. In this example, let’s assume that the process to update an order requires special business logic. This means that the special processing logic needs to be replicated across several applications in the enterprise, requiring additional verification and coordination between these applications. The other thing to notice in Figure 3-1 is that each system in this example has its own database, meaning each system might have a slightly different representation of an order.

![Figure 3-1. Silo-based Processing](image)

---

28 | Chapter 3: Comparing Architecture Characteristics
Service-Oriented Architecture tries to address this problem through enterprise-level shared services (enterprise services). Continuing with our retail example, by creating a centrally shared Order enterprise service as shown in Figure 3-2, every application can share the same processing logic associated with updating an order.

Notice in Figure 3-2 that although the Order service is now shared, it still accesses three different databases (each one representing the respective system it is associated with). This is a critical concept in SOA when using the share-as-much-as-possible architecture style. The Order service is smart enough to know which database to go to to retrieve and update order data for each system, while at the same time synchronizing the data between all three systems. In other words, the order is not represented by one database, but by a combination of three databases.

While the concept of a share-as-much-as-possible architecture solves issues associated with the duplication of business functionality, it also tends to lead towards tightly coupled components and increases the overall risk associated with change. For example, suppose you make a change to the Order service as shown in Figure 3-2. Since the Order service is an enterprise service and available globally across the company, it is very difficult to test all possible uses of this global service to make sure the change isn’t affecting other areas of the enterprise.
Microservices, being an architecture built on the concept of share-as-little-as-possible, leverages a concept from domain driven design called a bounded context. Architecturally, a bounded context refers to the coupling of a component (or in this case, a service) and its associated data as a single closed unit with minimal dependencies. A service component designed this way is essentially self-contained and only exposes a well-defined interface and a well-defined contract.

Realistically there will always be some services that are shared, even in a Microservices Architecture (for example, infrastructure services). However, whereas SOA tries to maximize on component sharing, Microservices tries to minimize on sharing through the concept of a bounded context. One way to achieve a bounded context and minimize dependencies in extreme cases is to violate the DRY principle (Don’t Repeat Yourself) and replicate common functionality across services to achieve total independence. Another way is to take relatively static modules and compile them into shared libraries that service components can use in either a compile-time or runtime binding. My friend and colleague Neal Ford takes a slightly different view of this by saying that Microservices is a share nothing architecture with the exception of two things - how services integrate with one another, and the infrastructure plumbing to ensure engineering consistency.

There are numerous advantages of leveraging the bounded context concept. Maintaining services becomes much easier due to the lack of dependent modules, allowing services to change and evolve independent of other services. Deployment becomes much easier as well because there is less code to deploy and also less risk that a change made to one module or service will affect other parts of the application. This in turn creates more robust applications that have fewer side effects based on service changes.

### Service Orchestration and Choreography

The difference between service orchestration and service choreography is unfortunately not always clear. In this section I will describe the differences between orchestration and choreography and how these service communication concepts are used in both Microservices and SOA.
The term *Service orchestration* refers to the coordination of multiple services through a centralized mediator such as a service consumer or an integration hub (e.g., Mule, Camel, Spring Integration, etc.). The diagram in Figure 3-3 illustrates the concept of service orchestration.

![Figure 3-3. Service Orchestration](image)

The easy way to think about service orchestration is to think about an orchestra. There are a number of musicians playing different instruments at different times, but they are all coordinated through a central person - the conductor. In the same way, the mediator component in service orchestration acts as an orchestra conductor, coordinating all of the service calls necessary to complete the business transaction.

*Service choreography*, on the other hand, refers to the coordination of multiple service calls without a central mediator. The term *inter-service communication* is sometimes used in conjunction with service choreography. With service choreography one service calls another service, which may call another service and so one, performing what is also referred to as *service chaining*. This concept is illustrated in Figure 3-4.

![Figure 3-4. Service Choreography](image)
One way to think about service choreography is to think about a dance company performing on stage. All of the dancers move in synchronization with each other, but there is no one conducting or directing the dancers. Dances are choreographed through the individual dancers working in conjunction with one another, whereas concerts are orchestrated by a single conductor.

Microservices favors service choreography over service orchestration, primarily due the lack of a centralized middleware component in the architecture topology. If you look at the diagram in Figure 3-5 you will see that the overall architecture topology only consists of two major components - service components and optionally a non-intelligent API layer (I will be discussing the API layer and its role in the next section). From an implementation standpoint you may have other components such as a service registration and discovery component, a service monitoring component, and a service deployment manager, but architecturally those components would be considered infrastructure services as part of the service taxonomy of the Microservices architecture pattern.

Because Microservices is a share-as-little-as-possible architecture, you should try to minimize the amount of service choreography in your architecture, restricting interactions to only those between functional services and infrastructure services. As mentioned in the prior chapter, if you find you need a lot of service choreography between your functional services, chances are your services are too fine-grained.
Too much service choreography in a Microservices architecture can lead to high efferent coupling, which is the degree to which one component is dependent on other components to complete a single business request. Consider the example illustrated in Figure 3-6 that shows three services that are required to process an order request - validate order, place order, and notify customer. Architecturally, this business request has a high degree of efferent coupling, something architects strive to minimize in most Microservices architectures.

This type of service coupling within service choreography can lead to poor performance and less robust applications. As I discussed in the prior chapter, since services are generally remote in a Microservices architecture, each service call made while coordinating services using service choreography adds response time to the request due to the remote access protocol communication and transport time. Furthermore, coordinating multiple services for a single business request increases the probability that a particular service in the call chain might not be available or might not be responding, resulting in a less reliable and robust application.

One solution to the issue of service choreography between functional services within a Microservices architecture is to combine fine-grained services into a more coarse-grained service. If a fine-grained service happens to be shared between multiple services, you can either keep this as a separate service, or depending on the size and nature of the functionality, violate the DRY (Don’t Repeat Yourself) principle and add that common functionality to each coarse-grained service.

Figure 3-7 shows how moving from three fine-grained services to one coarse-grained service eliminates the need for service choreography, thereby addressing three issues associated with service choreography. First, it increases overall performance due to fewer remote calls. Second, it increases overall robustness due to fewer service availability issues. Finally, it simplifies overall development and maintenance by eliminating the need for remote service contracts.
Service-Oriented Architecture, being a *share-as-much-as-possible* architecture, relies on both service orchestration and service choreography to process business requests. As illustrated in Figure 3-8, the messaging middleware component of SOA manages service orchestration by calling multiple enterprise services based on a single business service request. Once in the enterprise service, service choreography may be used to call application services or infrastructure services to help fulfill the particular business request.
Figure 3-8 also illustrates the variations that can occur within SOA with regards to service choreography. For example, an enterprise service may need to call an application service, and that application service may in turn need to call an infrastructure service to complete its business processing. Alternatively, the enterprise service may only need to call an application service or an infrastructure service directly, or the business logic may be self-contained within the enterprise service, thereby not requiring any service choreography.

The differences between Microservices and SOA with regards to service orchestration and service choreography bring out many architectural characteristics between these patterns, including performance, development, testing, and deployment. Because SOA typically relies on multiple services (and service types) to complete a single business request, architectures build on SOA tend to be slower than Microservices and require more time and effort to develop, test, deploy, and maintain. In fact, these factors were some of the drivers that led architects away from SOA and more towards the simple and streamline Microservices architecture pattern.
Middleware vs. API Layer

If you compare Figures 3-5 and 3-8 from the previous section you will notice that both architecture patterns appear to have a middleware component that handles mediation. However, this is not the case. The Microservices architecture pattern typically has what is known as an API Layer, whereas SOA has a messaging middleware component. In this section we will compare these two components in terms of the role they play and the capabilities they provide.

Microservices does not support the concept of messaging middleware (e.g., integration hub or enterprise service bus). Rather, it supports the notion of an API layer in front of the services that acts as a service access facade. Placing an API layer between your service consumers and the services is generally a good idea because it forms an abstraction layer so that service consumers don't need to know the actual location of the service end points. It also allows you to change the granularity level of your services without impacting the service consumers. Abstracting service granularity does require a bit of intelligence and some level of orchestration within the API layer, but this can be refactored over time, allowing services to evolve without constant changes to the corresponding service consumers.

For example, let's say you have a service that performs some business functionality related to product ordering. You decide it is too coarse-grained and you want to split the service into two smaller fine-grained services to increase scalability and ease deployment. Without an API layer abstracting the actual service endpoints, each service consumer using the service would have to be modified to call two services rather than just one. However, by using an API layer the service consumers don't know (or care) that the single request is now going to two separate services.

Service-Oriented Architecture, on the other hand, relies on its messaging middleware to coordinate service calls. Using messaging middleware (what I like to refer to as an integration hub) provides a host of additional architectural capabilities not found in the Microservices architecture style, including mediation and routing, message enhancement, message transformation, and protocol transformation.
Mediation and routing describes the capability of the architecture to locate and invoke a service (or services) based on a specific business or user request. This capability is illustrated in Figure 3-9 below. Notice in the diagram the use of a service registry or service discovery component as well as the use of service orchestration. Both Microservices and SOA share this capability, particularly with regards to a service registry or service discovery component. However, with Microservices service orchestration is typically minimized or not used at all, whereas with SOA it is frequently used.

Message enhancement describes the capability of the architecture to modify, remove, or augment the data portion of a request before it reaches the service. Examples of message enhancement include things like changing a date format, adding additional derived or calculated values to the request, and performing a database lookup to transform one value into another (like a CUSIP into a stock symbol and visa-versa). Microservices does not support this capability, primarily due to the lack of a middleware component to implement this functionality. SOA, on the other hand, fully supports this capability through its messaging middleware. Figure 3-10 illustrates this capability. Notice in the diagram that the service consumer is sending a CUSIP (a standard trading instrument identifier) and a date in mm/dd/yy format, whereas the service is expecting a SEDOL (another type of trading instrument identifier), the date in yyyy.mm.dd format, and the stock symbol (in the event of an equity trade). In this case the messaging middleware can perform these enhancements to convert the CUSIP for Apple, Inc. (037833100)
into the SEDOL for Apple, Inc. (2046251), lookup and add the symbol (AAPL), and convert the date from 04/23/15 to 2015.04.23.

**Figure 3-10. Message Enhancement Capability**

*Message transformation* describes the capability of the architecture to modify the format of the data from one type to other. For example, as illustrated in Figure 3-11, the service consumer is calling a service and sending the data in JSON (Javascript Object Notation) format, whereas the service requires a Java object. Notice that message enhancement is not concerned about the data of the request, but rather only about the format of the wrapper containing the data. Again, Microservices does not support this capability, but SOA does through the use of the messaging middleware.

**Figure 3-11. Message Transformation Capability**

Finally, *protocol transformation* describes the capability of the architecture to have a service consumer call a service with a different protocol that what the service is expecting. Figure 3-12 illustrates this
capability. Notice in the diagram that the service consumer is communicating through REST, but the services invoked that are responsible for processing the request require an RMI/IIOP connection (e.g., EJB3 bean) and an AMQP connection (Advanced Message Queueing Protocol). While Microservices can support multiple protocol types, the service consumer and service must be the same protocol. In SOA, you can mix and match them as much as you want.

![Diagram of Protocol Transformation Capability]

Figure 3-12. Protocol Transformation Capability

I will be discussing these capabilities in more detail in the next chapter as they relate to the comparison of architecture capabilities between Microservices and SOA.

**Accessing Remote Services**

Since services are usually accessed remotely in Microservices and SOA, these architecture patterns need to provide a way for service consumers to access the remote services. One of the fundamental differences between Microservices and SOA with regards to remote access is that Microservices architectures tend to rely on REST as their primary remote access protocol, whereas SOA has no such restrictions. As a matter of fact, having the ability to handle dozens of different kinds of remote access protocols is one of the main things that sets SOA apart from Microservices.

One of the fundamental principles within Microservices that contributes to the simplicity of the architecture pattern is that the number of technology and architecture choices is generally limited.
to a few options. For example, most Microservices architectures usually rely on only two different remote access protocols to access services - REST and simple messaging (e.g., JMS, MSMQ, AMQP, etc.). That’s not to say you couldn’t leverage other remote access protocols such as SOAP or .NET Remoting, but the point is that the remote access protocol found in Microservices is usually homogeneous. In other words, services are either REST-based, Messaging-based, or some other access protocol, but rarely mixed within the same application or system. One exception to this is the case where services that rely on publish-and-subscribe broadcast capabilities might be message-based, whereas other non-broadcast services might be REST-based.

Service-Oriented Architecture, on the other hand, has no pre-described limits as to what remote access protocols can be used as part of the architecture pattern. As you will see in the next chapter, it is the messaging middleware component of the architecture that provides support for any number of remote access protocols, allowing for transformation from one protocol to another. That being said, most SOA architectures typically rely on messaging (e.g., JMS, AMQP, MSMQ) and SOAP as the primary service remote access protocols. Depending on the scope and size of the SOA architecture, it’s not uncommon to use upwards to a half a dozen different remote access protocols between heterogeneous services.
In the last chapter I showed you how architecture patterns can help define the basic architectural characteristics of the architecture. In this chapter I take a similar approach, but instead of architecture characteristics I will focus on the architecture capabilities that are described through the patterns. By looking at an architecture pattern you can tell whether applications will likely be scalable, maintainable, and extensible, and whether they will be relatively easy to develop, test, and deploy.

In this chapter I will compare Microservices and SOA by focusing on three major architectural capabilities - the size of the application each architecture pattern supports, the type of systems and components that can be integrated using each architecture pattern, and finally the ability of the architecture pattern to support contract decoupling.

**Application Scope**

Application scope refers to the overall size of the application an architecture pattern can support. For example, architectures patterns such as the Microkernel or Pipeline architecture are better suited for smaller applications or subsystems, whereas other patterns such as event-driven architecture are well suited for larger, more complex applications. So where do Microservices and Service-Oriented Architecture fit in this spectrum?
Service-Oriented Architecture is well-suited for large, complex, enterprise-wide systems that require integration with many heterogeneous applications and services. It is also well-suited for applications that have many shared components, particularly components that are shared across the enterprise. As such, SOA tends to be a good fit for large insurance companies due to the heterogeneous systems environment and the sharing of common services such as Customer, Claim, Policy, etc. across multiple applications and systems.

However, workflow-based applications that have a well-defined processing flow and not many shared components (such as securities trading) are difficult to implement using the SOA architecture pattern. Small web-based applications are also not a good fit for SOA due to the lack of need for an extensive service taxonomy, abstraction layers, and messaging middleware components.

Microservices, on the other hand, is better suited for smaller, well partitioned web-based systems rather than large-scale enterprise-wide systems. The lack of a mediator (messaging middleware) is one of the factors that makes it ill-suited for large-scale complex business application environments. Other examples of applications that are well-suited for the Microservices Architecture pattern are ones that have few shared components and ones that can be broken down into very small discreet operations.

In some cases you might find that Microservices was a good initial architecture choice in the early stages of your business, but as the business grows and matures, you begin to need capabilities such as complex request transformation, complex orchestration, and heterogeneous systems integration. In these situations you will likely turn to the Service-Oriented Architecture pattern to replace your initial Microservices architecture. Of course, the opposite is true as well - you may have started out with a large, complex SOA architecture, only to find that you didn't need all of those powerful capabilities it supports after all. In this case you will likely find yourself in the common position of moving from an SOA architecture to Microservices to simplify the architecture.
Heterogeneous Interoperability

Heterogeneous interoperability refers to the ability to integrate with systems implemented in different programming languages and platforms. For example, you might have a situation where a single complex business request requires the coordination of a Java-based application, a .NET application, and a CICS COBOL program to process the single request. Other examples include a trading application implemented in the .NET platform that needs to access an AS400 to perform trade compliance checks on a stock trade, and a Java-based retail shop that needs to integrate with a large third-party .NET warehousing system.

These examples are found everywhere in most large companies. Many banking and insurance systems still have a majority of the backend core processing in COBOL mainframe applications that must be accessed by modern web-based platforms. The ability to integrate with multiple heterogeneous systems and services is one of the few areas where Microservices takes a backseat to SOA.

The Microservices architecture style attempts to simplify the architecture pattern and corresponding implementations by reducing the number of choices for services integration. REST and simple messaging are two of the most common remote access protocols used. Service-Oriented Architecture, on the other hand, has no upper limit and promotes the proliferation of multiple heterogeneous protocols through its messaging middleware component.

Microservices supports what is known as protocol-aware heterogeneous interoperability. With protocol-aware heterogeneous interoperability, the architecture can support multiple types of remote access protocols, but the communication between a particular service consumer and the corresponding service its invoking must be the same (e.g., REST). As illustrated in Figure 4-1, just because the remote access protocol is known between the service consumer and service does not necessarily mean the implementation of either is known or has to be the same. With REST, for example, the service consumer could easily be implemented in C# with .NET, whereas the service could be implemented in Java. However, with Microservices, the protocol between the service consumer and service must be the same because there is no central middleware component to transform the protocol.
Service-Oriented Architecture also supports protocol-aware heterogeneous interoperability, but it takes this concept one step further by supporting what is known as protocol-agnostic heterogeneous interoperability. With protocol-agnostic heterogeneous interoperability, the service consumer is not only ignorant of the implementation of the service, but also the protocol the service is listening on. For example, as illustrated in Figure 4-2, a particular service consumer written in C# on the .NET platform may invoke a corresponding service using REST, but the service (in this case an EJB3 bean) is only able to communicate using RMI. Being able to translate the consumer protocol to the service protocol is known as protocol transformation, and is supported through the use of a messaging middleware component. Again, since Microservices does not have any concept of a messaging middleware component, it does not support the concept of protocol-agnostic heterogeneous interoperability.

Figure 4-1. Protocol-Aware Heterogeneous Interoperability
If you find yourself in a heterogeneous environment where you need to integrate several different types of systems or services using different protocols, that chances are you will need to look towards Service-Oriented Architecture rather than Microservices. However, if all of your services can be exposed and accessed through the same remote access protocol (e.g., REST), then Microservices may be the right choice. In either case, this is one area where you need to know your interoperability requirements prior to selecting an architecture pattern.

**Contract Decoupling**

Contract decoupling is the holy grail of abstraction. Imagine being able to communicate with a service using different data in a different message format than what the service is expecting - that is the very essence of contract decoupling.
Contract decoupling is a very powerful capability that provides the highest degree of decoupling between service consumers and services. This capability allows services and service consumers to evolve independently from one another while still maintaining a contract between them. It also helps give your service consumers the ability to drive contract changes using consumer-driven contracts, thereby creating a more collaborative relationship between the service and the service consumer.

There are two primary forms of contract decoupling - message transformation and message enhancement. Message transformation is concerned only about the format of the message, not the actual request data. For example, a service might require XML as it's input format, but a service consumer decides to send JSON payload instead. This is a straightforward transformation task that is handled very nicely by most of the open source integration hubs, including Apache Camel, Mule, and Spring Integration.

Things tend to get a bit more complicated when the data sent by a service consumer differs from the data expected by the corresponding service. This impedance mismatch in the actual contract data is addressed through a capability known as message enhancement. Whereas message transformation is concerned about the format of the request, message enhancement is concerned about the request data. This capability allows a component (usually a middleware component) to add or change request data so that the data sent by the service consumer matches the data expected by the service (and visa-versa).

Consider the scenario where a service consumer is sending some data as a JSON object for a simple stock trade. In this example, the service consumer is invoking a service by sending a customer id, a CUSIP identifying the stock to be traded, the number of shares to be traded, and finally the trade date in MM/DD/YY format as follows:

```json
{ "trade": {  
    "cust_id": "12345",
    "cusip": "037833100",
    "shares": "1000",
    "trade_dt": "10/12/15"
 }}
```

The service, on the other hand, is expecting data in XML format consisting of an account number, a stock symbol (assuming an
equity trade), the shared to be traded, and the trade date in YYYY-MM-DD format as follows:

```xml
<trade>
  <acct>321109</acct>
  <symbol>AAPL</symbol>
  <shares>1000</shares>
  <date>2015-10-12</date>
</trade>
```

When differences occur in the format of the contract between the service consumer and the service, it is usually the messaging middleware component or custom client adapter that performs the necessary data transformation and data lookup functionality to make the different contracts work together. The diagram in Figure 4-3 illustrates this example. Database or cache lookups are performed to get the account number based on the customer id and the symbol based on the CUSIP. The date is also converted to a different format, and the shares copied over to the new format since that field does not require any translation. This allows the service consumer to have a different contract than the service so that when contract changes are made, they can be abstracted through the messaging middleware.

![Diagram of service consumer, enhance message, and service with XML交易示例](image)

**Figure 4-3. Contract Decoupling**

There are obviously some practical limitations to contract decoupling. If the data required by a service cannot be derived from another source or calculated using the data provided by the service consumer, the service call will fail because the service contract is not satisfied. Fortunately, lookup capabilities and basic transformations (such as date, time, and number fields) can usually fix most contract variances between service consumers and services.

An ongoing struggle in the IT industry is knowing how to prevent technology (the IT department) from driving the business. Whether
you are performing a major software version upgrade of a large subsystem or replacing your accounting or customer management system, abstracting the interfaces and contracts through contract decoupling allows the IT department to make technology changes without impact to the business applications across the enterprise. The stock trading scenario described earlier is a good example of this; swapping out a trading platform that uses CUSIPs to one that requires SEDOLs should not require all the business applications throughout the enterprise to change to SEDOLs.

Unfortunately, Microservices must once again take a back seat to SOA with respect to this architecture capability. Microservices does not support contract decoupling, whereas contract decoupling is one of the primary capabilities offered within a Service-Oriented Architecture. If this level of abstraction is something you require in your architecture, you will need to look towards a Service-Oriented Architecture solution rather than a Microservices one for your application or system.
The Microservices architecture pattern is a rising star in the IT industry. While Microservices has certainly addressed the many issues commonly found in large monolithic applications and complex SOA architectures, it does lack some of the core capabilities provided by a SOA, including contract decoupling and protocol agnostic heterogeneous interoperability.

One of the fundamental concepts to remember is that Microservices is a share-as-little-as-possible architecture pattern that places a heavy emphasis on the concept of a bounded context, whereas SOA is a share-as-much-as-possible architecture pattern that places heavy emphasis on abstraction and business functionality reuse. By understanding this fundamental concept, as well as the other characteristics, capabilities, and shortcomings of both Microservices and SOA I discussed in this report, you can make a more informed decision about which architecture pattern is right for your situation.

For more information about Microservices, Service-Oriented Architecture, and distributed architecture in general, you can view the O'Reilly video Software Architecture Fundamentals: Service-based Architectures that Neal Ford and I recorded, which can be found on Safari Online or through the O'Reilly website at http://shop.oreilly.com/product/0636920042655.do.

For an excellent in-depth look at Microservices, I would highly recommend Sam Newman’s book Building Microservices which you can get through the O'Reilly website at http://shop.oreilly.com/product/0636920033158.do or through Safari Online.
Finally, for more information about messaging as it relates to service-based architectures for both Microservices and SOA, you can view my O’Reilly videos Enterprise Messaging: JMS 1.1 and JMS 2.0 Fundamentals (http://shop.oreilly.com/product/0636920034698.do) and Enterprise Messaging: Advanced Topics and Spring JMS (http://shop.oreilly.com/product/0636920034865.do). Both of these videos are also available through Safari Online.
Mark Richards is an experienced, hands-on software architect involved in the architecture, design, and implementation of microservices architectures, service-oriented architectures, and distributed systems in J2EE and other technologies. He has been in the software industry since 1983 and has significant experience and expertise in application, integration, and enterprise architecture. Mark served as the president of the New England Java Users Group from 1999 through 2003. He is the author of numerous technical books and videos, including Software Architecture Fundamentals (http://bit.ly/sa-fundamentals) (O’Reilly video), Enterprise Messaging (http://bit.ly/enterprise-messaging) (O’Reilly video), Java Message Service, 2nd Edition (http://bit.ly/java-message-service) (O’Reilly), and a contributing author to 97 Things Every Software Architect Should Know (http://bit.ly/97-things-software) (O’Reilly). Mark has a master’s degree in computer science and numerous architect and developer certifications from IBM, Sun, The Open Group, and BEA. He is a regular conference speaker at the No Fluff Just Stuff (NFJS) Symposium Series and has spoken at more than 100 conferences and user groups around the world on a variety of enterprise-related technical topics. When he is not working, Mark can usually be found hiking in the White Mountains of New Hampshire and along the Appalachian Trail.