O'REILLY®

Microservices vs. Service-Oriented Architecture

Mark Richards

O'REILLY®

Additional Resources

4 Easy Ways to Learn More and Stay Current

Programming Newsletter
Get programming related news and content delivered weekly to your inbox.
oreilly.com/programming/newsletter

Free Webcast Series
Learn about popular programming topics from experts live, online.
webcasts.oreilly.com

O'Reilly Radar
Read more insight and analysis about emerging technologies.
radar.oreilly.com

Conferences
Immerse yourself in learning at an upcoming O'Reilly conference.
conferences.oreilly.com

Microservices vs. Service-Oriented Architecture

Mark Richards

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Microservices vs. Service-Oriented Architecture
by Mark Richards

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Nan Barber, Rachel Roumeliotis
Production Editor: Nicholas Adams
Copyeditor: Eileen Cohen
Proofreader: Nicholas Adams

Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

November 2015: First Edition

Revision History for the First Edition
2015-11-17: First Release
2015-04-22: Second Release

978-1-491-94161-4

[LSI]

Table of Contents

Foreword

One of the fascinating aspects of software engineering is how great concepts endure, but their execution and application are regularly reinvented using the tools and practices of the day. The rise of microservices patterns and practices is a great example of this process.

Skeptics may dismiss microservices as little more than the service-oriented architecture (SOA) practices of the 2000s, reheated. The reality is that microservices are an example of convergent evolution, emerging from modern development, testing, and deployment techniques and shaped by the democratizing influence of open source. The resulting incarnation of SOA is transforming our industry.

What's the nature of this transformation? Early adopters report that when used with an agile and autonomous approach to engineering structures and a DevOps approach to delivery, the microservices approach enables a much quicker cadence of application development. In turn, software developers welcome this, having learned their trade in the fast-moving, app-driven world of services and applications. The net result is faster innovation, and a potential competitive advantage for Internet-facing organizations that embrace these practices.

The power of microservices comes from their non-prescriptive nature. There is no formal, slow-moving, industry-driven specification; rather, the microservices approach has emerged as a pattern of development that has been practiced and refined by pioneers. Born in the modern Web, microservices are interconnected using a thin layer of simple APIs and the lingua franca of HTTP. At NGINX, we are incredibly proud of our role in accelerating, securing, scaling, and delivering microservices-based applications. As the HTTP "connective tissue" of modern applications, NGINX serves as both the stable entry point to a microservices-based application, and the traffic manager for communication between the microservices themselves.

The lack of a prescriptive industry specification is sure to raise many, many questions—especially if you have experience with previous SOA practices. Microservices have at times been billed as "SOA done right," but is that really the case? And what exactly did

earlier practices do wrong that microservices are getting right? In this report, Mark Richards provides a thoughtful breakdown of the two approaches and discusses where one approach might be preferred over the other. By the end of this report, you'll have not only a greater understanding of these two architectural patterns, but also of application development as a whole. We hope you enjoy this report.

—Owen Garrett,
Head of Products, NGINX, Inc.

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 enable the IT organization and the business to better communicate and collaborate with each other. Dozens of best practices for implementing SOA emerged at that time, as well as a plethora of third-party products to help companies implement SOA. 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 architecture is taking the IT industry by storm as the go-to style for developing highly scalable and modular applications. Microservices architecture holds the promise of being able to address some of the problems associated with large, complex SOAs as well as the problems found with big, bloated monolithic applications. But how much do microservices and SOA differ? Is the industry destined to repeat the same experience with microservices as with SOA?

In this report I 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 capabilities. By using the information in this report, you will know how these two architecture styles differ from each other and which of the two is best suited for your particular situation.

CHAPTER 1

The World of Service-Based Architectures

Both microservices architecture and SOA are considered *service-based architectures*, meaning that they are architecture patterns that place a heavy emphasis on *services* as the primary architecture component used to implement and perform business and nonbusiness functionality. Although microservices and SOA are very different architecture styles, they share many characteristics.

One thing all service-based architectures have in common is that they are generally *distributed architectures*, meaning that service components are accessed remotely through some sort of remote-access protocol—for example, Representational State Transfer (REST), Simple Object Access Protocol (SOAP), Advanced Message Queuing Protocol (AMQP), Java Message Service (JMS), Microsoft Message Queuing (MSMQ), Remote Method Invocation (RMI), or .NET Remoting. 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 applications that are more robust and more responsive. Distributed architectures also lend themselves to more loosely coupled and modular applications.

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 are no exception. The trade-offs associated with those advantages are, primarily, increased complexity and cost. Maintaining service contracts, choosing the right remote-access protocol, dealing with unresponsive 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 serviced-based architecture.

Service Contracts

A *service contract* is an agreement between a (usually remote) service and a service consumer (client) that specifies the inbound and outbound data along with the contract format (XML, JavaScript Object Notation [JSON], Java object, etc.). Creating and maintaining service contracts is a difficult task 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 you can use two basic types of service contract models: service-based contracts and consumer-driven contracts. The real difference between these contract models is the degree 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, whether or not 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 with respect to the contracts that bind them. This type of model generally requires the service to know who its consumers 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 can 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 and Pacto can help with maintaining and testing consumer-driven contracts.

Another critical topic within the context of service contracts is *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 are 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 and at the same time provide backward compatibility for service consumers that are still using prior contracts. Perhaps one of the most important pieces 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 several open source and commercial frameworks are available to help you manage and implement contract-versioning strategies, you can use two basic techniques 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 recipient is not at home when the order is delivered. In this case the original contract (version 1.0) can

remain backward compatible by making the new delivery-instructions field optional.

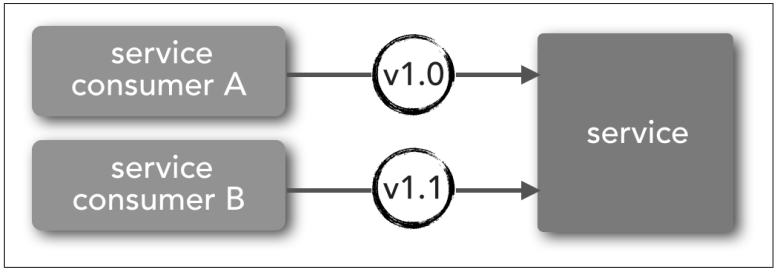

Figure 1-1. Contract version numbers

Heterogeneous versioning involves supporting multiple types of contracts. This technique is closer to the concept of consumer-driven 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, backward 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.

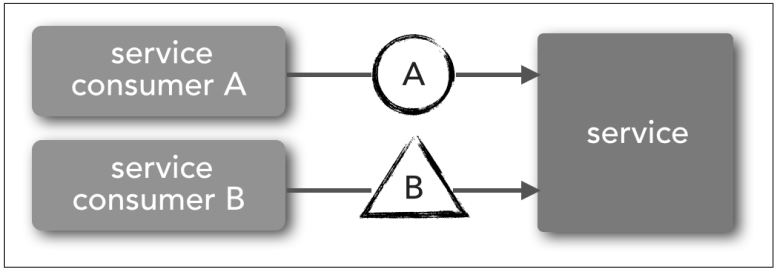

Figure 1-2. Multiple contracts

Providing backward 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 because the number of internal and/or external service consumers is large. 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 in the "Contract Decoupling" section in Chapter 4.

Service Availability

Service availability and service responsiveness are two other considerations common to all service-based architectures. Although 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* 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.

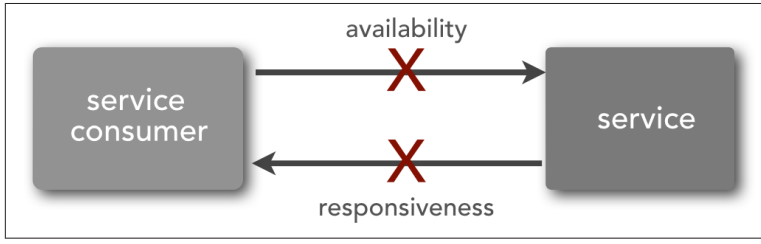

Figure 1-3. Service availability and responsiveness

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 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 to prevent the response from being sent?

Addressing timeout conditions can be one of the more challenging aspects of remote service connectivity. A common way to determine 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 2,000 milliseconds. In this case you might double that value to account for high load conditions, resulting in a timeout value of 4,000 milliseconds.

Although this may seem like a reasonable solution for calculating 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 four seconds before determining that the service is not responding. This is inefficient and annoying to the end user of the service request. Another problem is that your benchmarks may not have been accurate, and under heavy load the service response is actually averaging five seconds rather than the four seconds you calculated. In this case the service is in fact responding, but the service consumer will reject every request because the timeout value is 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!* (Pragmatic Bookshelf).

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: the fact that service consumers can connect to a service doesn't necessarily mean that 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 SOA implementations. Functionality that used to be located in a secure 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 because no middleware component handles 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 authori-

zation in the service itself. Although 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 (atomicity, consistency, isolation, and durability) transactions found in most business applications. ACID transaction 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.

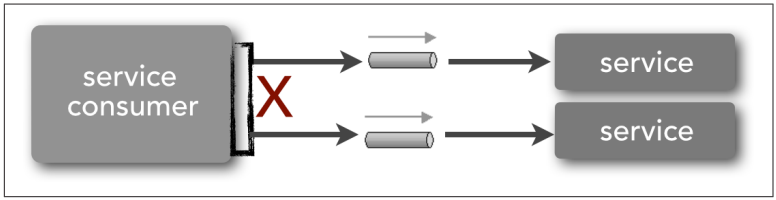

Figure 1-4. Service transaction management

Transaction issues are much more prevalent in SOA because, unlike in 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 *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. When you deposit cash into your account through an ATM, it can take from several minutes to several hours for the deposit to appear in your account. In other words, there is a soft transition state in which 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 in which 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 BASE transactions are used.

Too Much Complexity?

Service-based architectures are a significant improvement over monolithic applications, but as you can see they involve many considerations—including service contracts, availability, security, and transactions (to name a few). Unfortunately, moving to a service-based architecture approach such as microservices or SOA involves trade-offs. 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.

The issues identified in this chapter are complex, but 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 come 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 walk you through those capabilities by comparing microservices and SOA to help you decide which architecture pattern is right for you.

CHAPTER 2

Comparing Service Characteristics

The OASIS Reference Model for Service Oriented Architecture 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 and 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 on 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. In this chapter I 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 classification refers 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 nonbusiness 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, or order shipping.

Service type classification is generally defined at the architecture pattern level, whereas business area classification is defined at the architecture implementation level. Although 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 I focus 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 services that support specific business operations or functions, whereas *infrastructure services* support nonfunctional tasks such as authentication, authorization, auditing, logging, and monitoring. This is an important distinction within a microservices architecture, because infrastructure services are not exposed to the outside world but rather are treated as private shared services only available internally to other services. Functional services are accessed externally and are generally not shared with any other service.

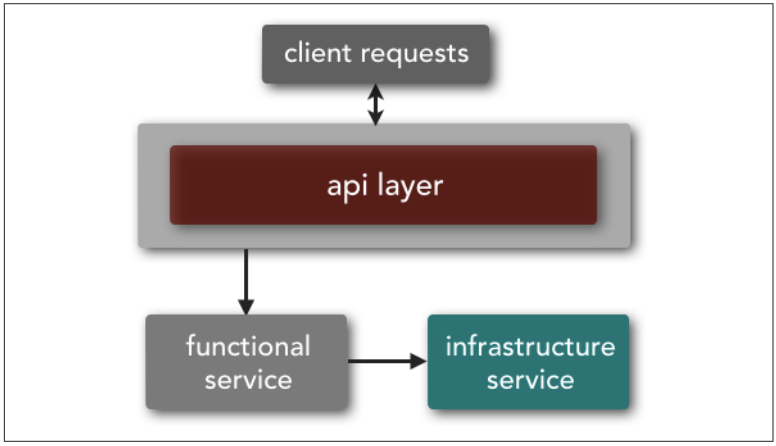

Figure 2-1. Microservice service taxonomy

The service taxonomy within SOA varies significantly from microservices taxonomy. In SOA there is a very distinct and formal service taxonomy in terms of the type of service and the role of that service in the overall architecture. While there can be any number of service types within SOA, the architecture pattern defines four basic types, as illustrated in Figure 2-2.

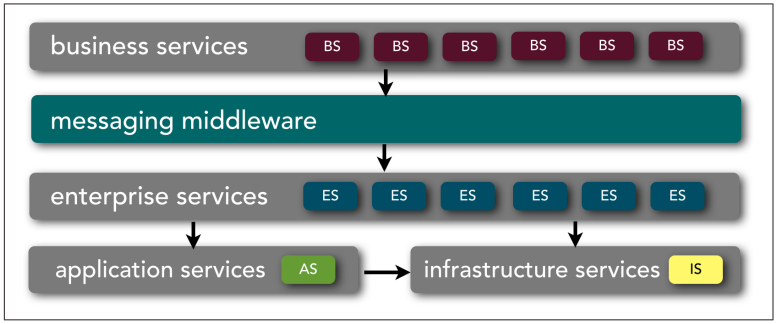

Figure 2-2. SOA taxonomy

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 devoid of any implementation or protocol, and they 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 represented through either XML, Web Services Definition Language (WSDL), or Business Process Execution Language (BPEL). A good litmus test for determining if 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 *ProcessTrade* and *InsertCustomer* services. 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 services that implement the functionality defined by business services. As illustrated in Figure 2-2, it is usually the middleware component that bridges the abstract business services and the corresponding concrete enterprise services implementations. Enterprise serv-

ices 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 commercial off-the-shelf (COTS) product. One unique thing about enterprise services is that they are generally shared across the organization. For example, a *RetrieveCustomer* enterprise service may be used by different parts of the organization to provide a common way to retrieve customer information. *CheckTradeCompliance*, *CreateCustomer*, *ValidateOrder*, and *GetInventory* are all good examples of enterprise services. Enterprise services typically rely on *application services* and *infrastructure services* 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 insurance 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 SOA is *infrastructure services*. As in microservices architecture, these are services that implement nonfunctional 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 which 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 the type of group within the organization that is responsible for creating and maintaining a service. Because micro-

services architecture 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. Even though dozens of application development teams might be writing services, the important thing here to note is that they all belong to the same type of group (i.e., application development teams). Figure 2-3 illustrates the typical service ownership model for microservices.

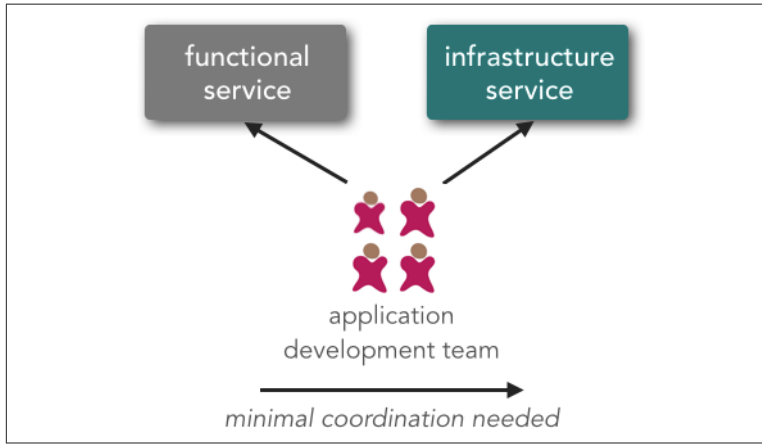

Figure 2-3. Microservices service ownership model

With SOA, 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.

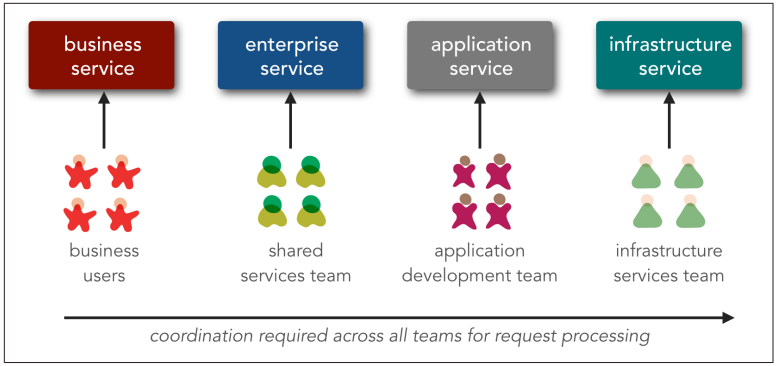

Figure 2-4. SOA service ownership model

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 nonfunctional 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 among services to fulfill a single business request. If coordination is needed among service owners, it is done quickly and efficiently through small application development teams.

This difference between microservices and SOA in service ownership and overall coordination 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 in which the microservices pattern stands out from the crowd. Thanks 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, lower development and maintenance costs, and more-robust applications.

Service Granularity

One of the bigger differences from a services perspective between microservices and SOA is service granularity. As the name suggests, microservices are 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 SOA, 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 SOA was service granularity. Not understanding the impact of service granularity, architects frequently designed services that were too fine-grained, resulting in chatty and poorly 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 architecture generally does not. I discuss this topic in more detail in the "Service Orchestration" section of the next chapter.

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 interservice communication to fulfill a single business request, resulting in numerous remote service calls that take up valuable time. For example, let's say it takes four 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 spend *just on transport time*. Consolidating these services into a single service would reduce that transport time to 200 milliseconds, shaving off close to half a second of processing time.

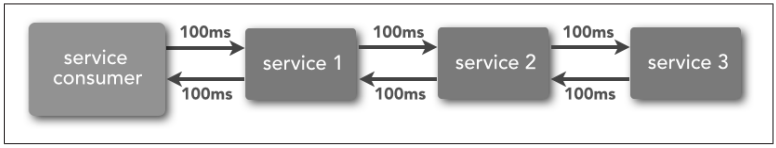

Figure 2-5. Service granularity impact on performance

Transaction management is also impacted by service granularity. I am referring here to 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 by the top diagram in 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.

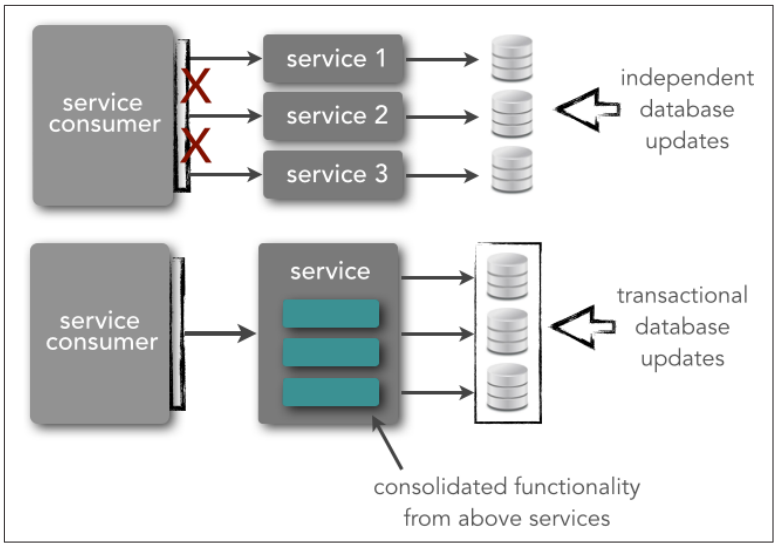

Figure 2-6. Service granularity impact on transactions

When dealing with service granularity I usually find it easier to start out with services that are more coarse-grained than 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* (O'Reilly), "*Start with a small number of larger services first.*" Just watch out for transaction issues and too much interservice 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 has the most potential 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. Although moving to services that are more coarse-grained certainly resolves performance and transactional issues, it also adversely affects development, testing, deployment, and maintenance. If you find that your services range in size from small to large, you will likely need to look toward more of a SOA 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 the microservices pattern 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 take more of a global view and compare their architectural aspects, including the level of sharing among 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 pattern.

CHAPTER 3

Comparing Architecture Characteristics

A *component* is 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, 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 that 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 if the pattern is the right choice for you.

In this chapter I explore the differences between microservices and SOA in terms of the overall architecture topology and the defining characteristics of the architecture pattern. Specifically, I focus on the differences between the two patterns with respect to the level of service-component sharing, the level of service-component communication, and how remote service components are typically accessed. I also dive into the differences 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. SOA is built on the concept of a *share-as-much-as-possible* architecture style, whereas microservices architecture is built on the concept of a *share-as-little-as-possible* architecture style. In this section I explore the differences between these two concepts as they relate to microservices and SOA.

Component sharing is one of the core tenets 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 among these applications. The other thing to notice in Figure 3-1 is that each system in this example has its own database, so each system might have a sightly different representation of an order.

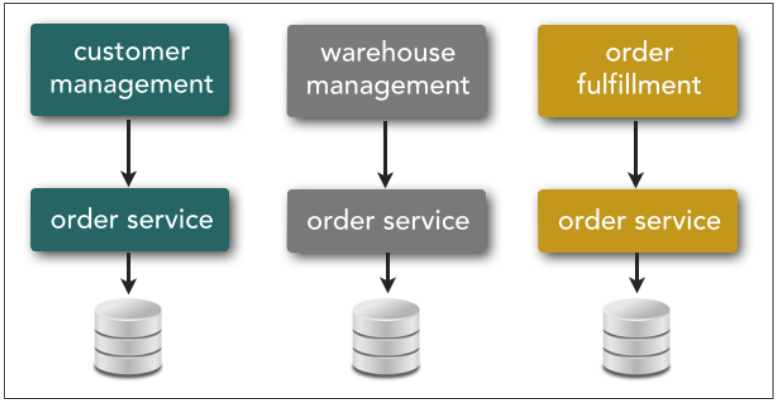

Figure 3-1. Silo-based processing

SOA tries to address this problem through enterprise-level shared services (enterprise services). Continuing with our retail example, if a centrally shared *Order* enterprise service is created, as shown in Figure 3-2, every application can share the same processing logic associated with updating an order.

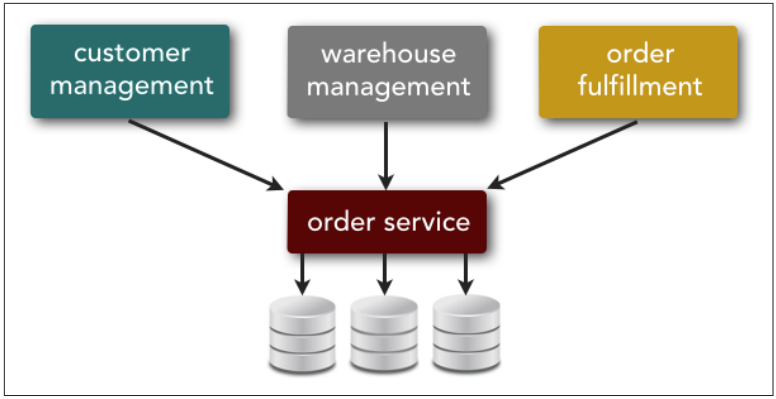

Figure 3-2. Service component sharing

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 the *share-as-much-as-possible* architecture style is used. The *Order* service is smart enough to know which database to go to to retrieve and update order data for each system, at the same time synchronizing the data among all three systems. In other words, the order is represented not by one database, but by a combination of three databases.

Alhtough the concept of a *share-as-much-as-possible* architecture solves issues associated with the duplication of business functionality, it also tends to lead to tightly coupled components and increases the overall risk associated with change. For example, suppose you make a change to the *Order* service 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 architecture, being 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 component sharing, microservices architecture 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 Don't Repeat Yourself (DRY) principle and replicate common functionality across services to achieve total independence. Another way is to compile relatively static modules 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 architecture 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 to leveraging the bounded context concept. Maintaining services becomes much easier because of 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 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 (Mule, Camel, Spring Integration, etc.). The diagram in Figure 3-3 illustrates the concept of service orchestration.

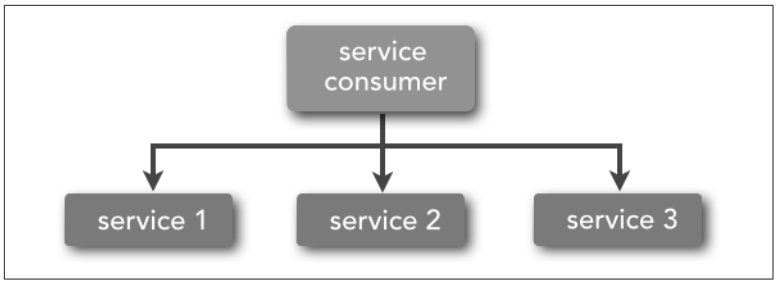

Figure 3-3. Service orchestration

The easy way to think about service orchestration is to think about an orchestra. A number of musicians are 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 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 on, performing what is also referred to as *service chaining*. This concept is illustrated in Figure 3-4.

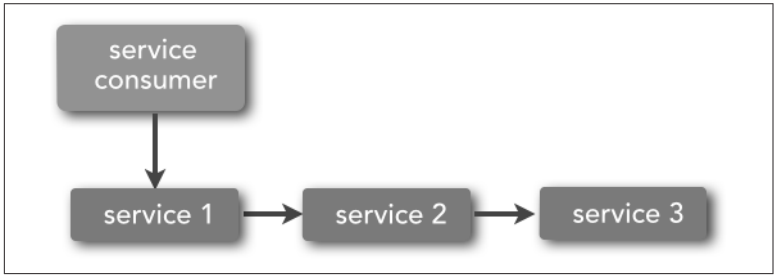

Figure 3-4. Service choreography

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 one another, but no one is 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 architecture favors service choreography over service orchestration, primarily because the architecture topology lacks a centralized middleware component. The diagram in Figure 3-5 shows that the overall architecture topology consists of only two major components—service components and, optionally, an unintelligent API layer. (I discuss 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.

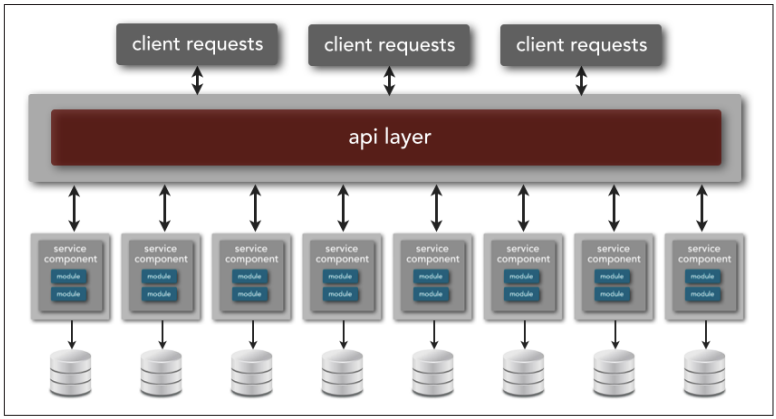

Figure 3-5. Microservices architecture topology

Because microservices architecture is a *share-as-little-as-possible* architecture, you should try to minimize the amount of service choreography in your architecture, restricting interactions to those between functional services and infrastructure services. As I mentioned in the prior chapter, if you find that 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, which 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.

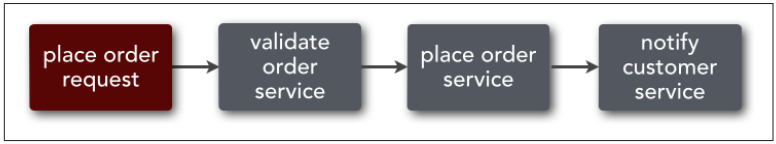

Figure 3-6. Order processing service choreography example

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 be unavailable or unresponsive, resulting in a less reliable and robust application.

One solution to the issue of service choreography among 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 among multiple services, you can either keep this as a separate service, or—depending on the size and nature of the functionality—violate the DRY 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 because fewer remote calls are needed. Second, it increases overall robustness because fewer service availability issues occur. Finally, it simplifies overall development and maintenance by eliminating the need for remote service contracts.

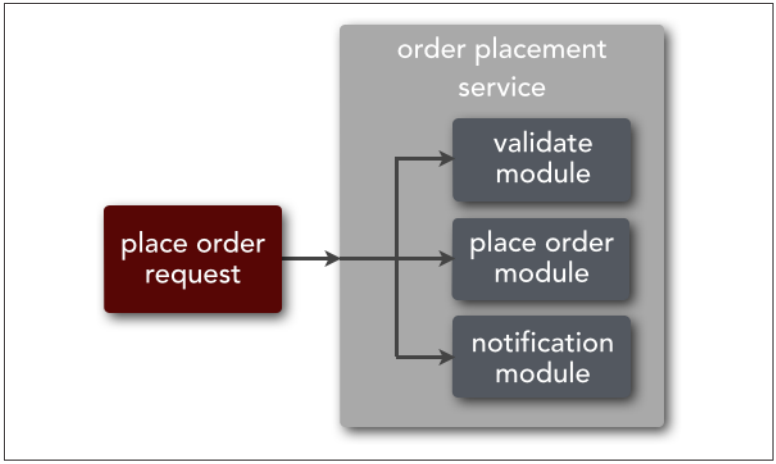

Figure 3-7. Order processing consolidated service example

SOA, 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 it is in the enterprise service, service choreography can be used to call application services or infrastructure services to help fulfill the particular business request.

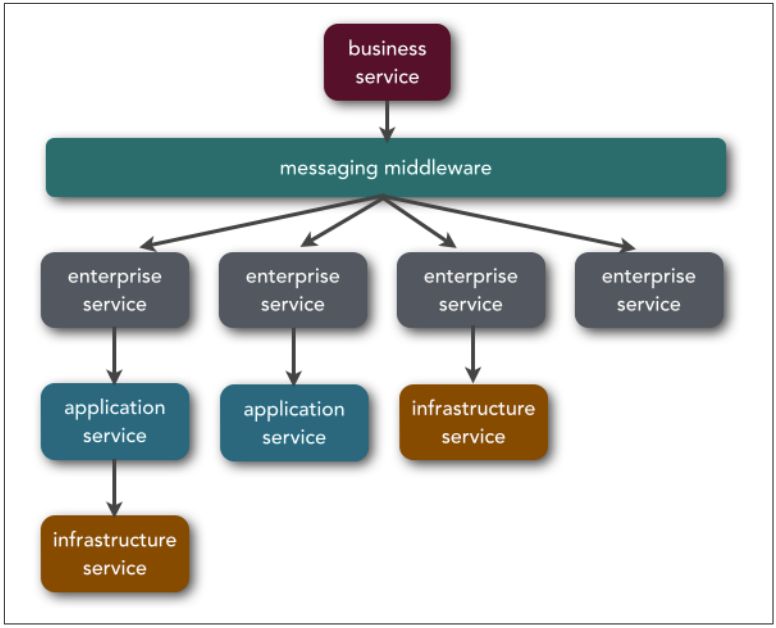

Figure 3-8. SOA topology

Figure 3-8 also illustrates the variations that can occur within SOA with regard 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 regard to service orchestration and service choreography underscore many differences between the two patterns in architectural characteristics, including performance, development, testing, and deployment. Because SOA typically relies on multiple services (and service types) to complete a single business request, systems built 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 toward the simple and streamlined microservices architecture pattern.

Middleware vs. API Layer

If you compare Figure 3-5 and Figure 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 I compare these two components in terms of the roles they play and the capabilities they provide.

The microservices pattern 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 endpoints. 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 that 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. If you use an API layer, the service consumers don't know (or care) that the single request is now going to two separate services.

SOA 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. 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 regard 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.

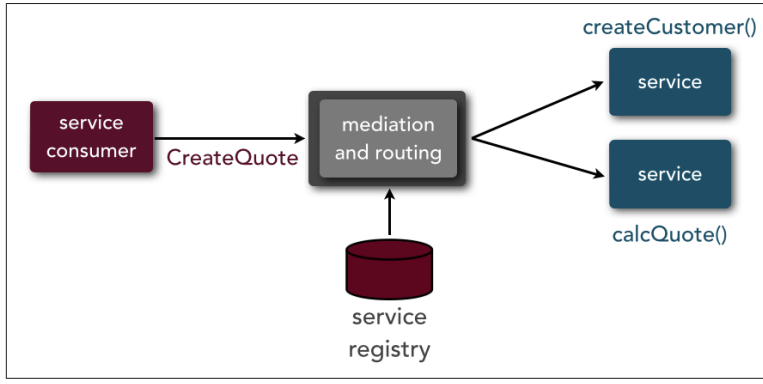

Figure 3-9. Mediation and routing capability

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 (such as a Committee on Uniform Security Identification Procedures [CUSIP] number into a stock symbol, and vice versa). The microservices pattern does not support this capability, primarily because it doesn't include a middleware component to implement this functionality. SOA 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 number (a standard trading-instrument identifier) and a date in MM/DD/YY format, whereas the service is expecting a Stock Exchange Daily Official List (SEDOL) number (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 number for Apple, Inc. (037833100) into the SEDOL

number for Apple, Inc. (2046251), look up and add the symbol (AAPL), and convert the date from 04/23/15 to 2015.04.23.

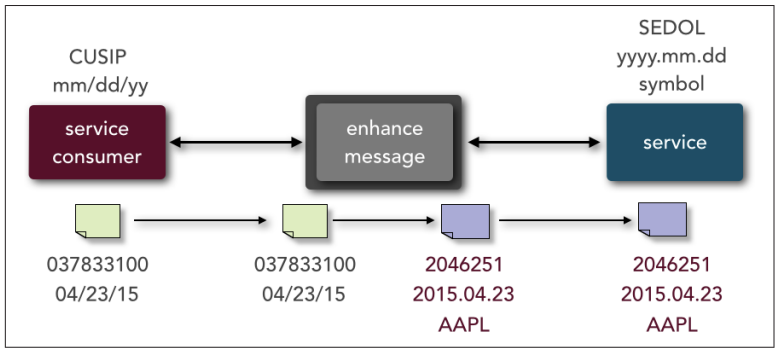

Figure 3-10. Message enhancement capability

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 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 architecture does not support this capability, but SOA does through the use of the messaging middleware.

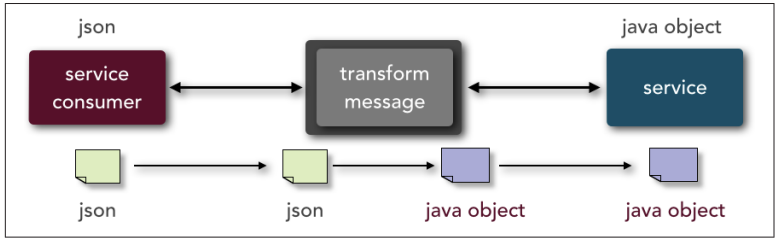

Figure 3-11. Message transformation capability

Finally, *protocol transformation* describes the capability of the architecture to have a service consumer call a service with a protocol that differs from 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., Enterprise JavaBeans 3 [EJB3] bean) and an AMQP connection. Microservices can support multiple protocol types, but the ser-

vice consumer and service must use the same protocol. In SOA, you can mix and match them as much as you want.

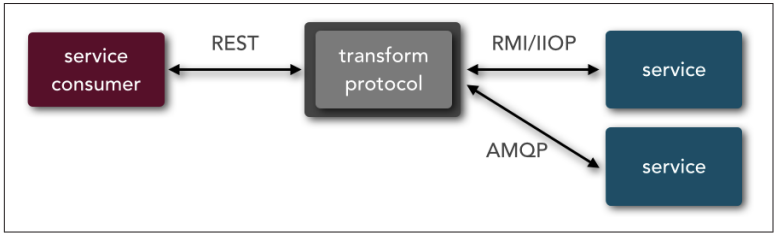

Figure 3-12. Protocol transformation capability

I discuss 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 regard 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 (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 based on some other access protocol, but the access protocols rarely mixed within the same application or system. One exception to this is the case in which services that rely on publish-and-subscribe broadcast

capabilities might be message-based, whereas other nonbroadcast services might be REST-based.

SOA has no pre-described limits as to which 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 of a half a dozen different remote-access protocols among heterogeneous services.

CHAPTER 4

Comparing Architecture Capabilities

In the last chapter I showed you how architecture patterns can help define basic architectural characteristics. In this chapter I take a similar approach, but instead of architecture characteristics I focus on the architecture capabilities that are described through the patterns. By looking at an architecture pattern you can tell if applications will likely be scalable, maintainable, and extensible, and if they will be relatively easy to develop, test, and deploy.

In this chapter I 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 that an architecture pattern can support. For example, architecture 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 the microservices and SOA patterms fit along this spectrum?

SOA 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—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 because they don't need an extensive service taxonomy, abstraction layers, and messaging middleware components.

The microservices pattern 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 discrete operations.

In some cases you might find that the microservices pattern is 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 SOA 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 that 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 in which a single

complex business request requires the coordination of a Java-based application, a .NET application, and a Customer Information Control System (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 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 architecture takes a back seat 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. SOA, has no upper limit and promotes the proliferation of multiple heterogeneous protocols through its messaging middleware component.

The microservices architecture style supports *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 that it's invoking must be the same (e.g., REST). As illustrated in Figure 4-1, the fact that the remote-access protocol between the service consumer and service is known does not necessarily mean that 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.

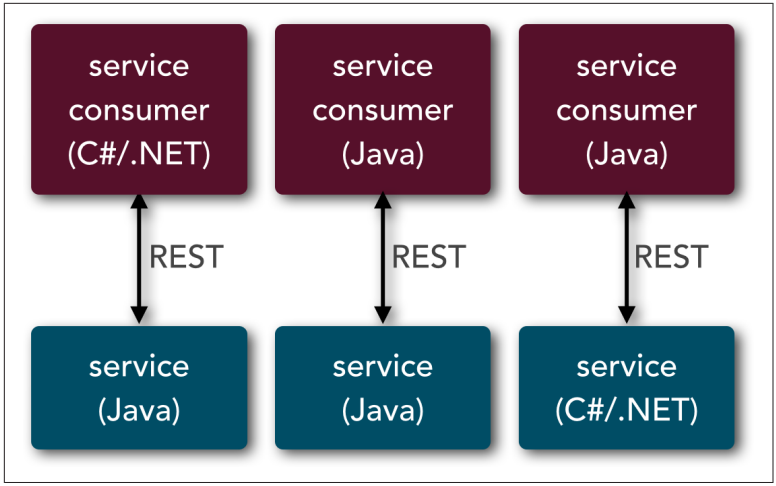

Figure 4-1. Protocol-aware heterogeneous interoperability

SOA also supports protocol-aware heterogeneous interoperability, but it takes this concept one step further by supporting *protocol-agnostic heterogeneous interoperability*. With protocol-agnostic heterogeneous interoperability, the service consumer is ignorant not only of the implementation of the service, but also of 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 known as *protocol transformation* is supported through the use of a messaging middleware component. Again, since a microservices architecture has no concept of a messaging middleware component, it does not support the concept of protocol-agnostic heterogeneous interoperability.

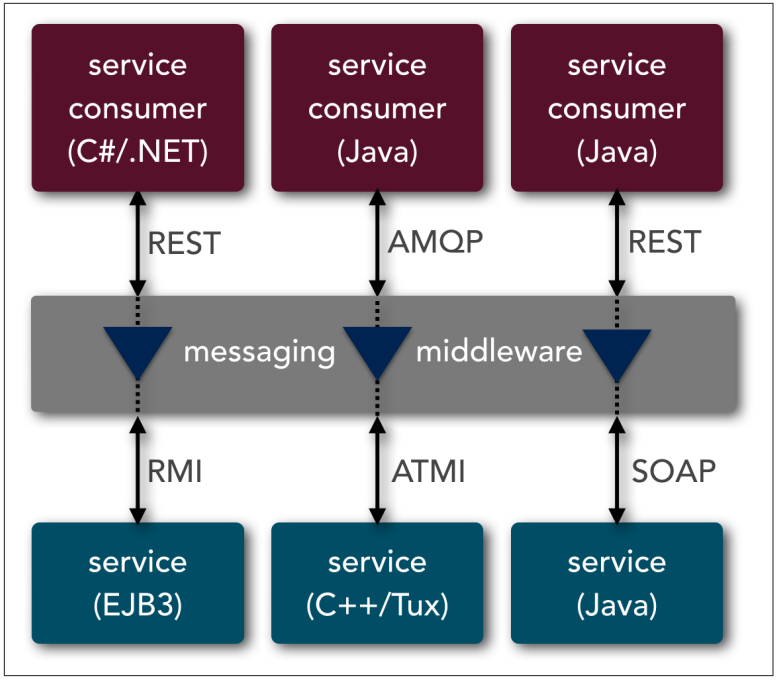

Figure 4-2. Protocol-agnostic heterogeneous interoperability

If you find yourself in a heterogeneous environment where you need to integrate several different types of systems or services using different protocols, chances are that you will need to look toward SOA 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 can 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 message format that differs from 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 each other 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. M*essage transformation* is concerned only about the *format* of the message, not the actual request data. For example, a service might require XML as its 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 message enhancement capability. 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 vice versa).

Consider the scenario in which 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 number identifying the stock to be traded, the number of shares to be traded, and finally the trade date in MM/DD/YY format:

```
{"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:

```
<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 number. The date is also converted to a different format, and the shares are copied over to the new format since that field does not require any translation. This allows the service consumer to have a different contract from the service, so that when contract changes are made, they can be abstracted through the messaging middleware.

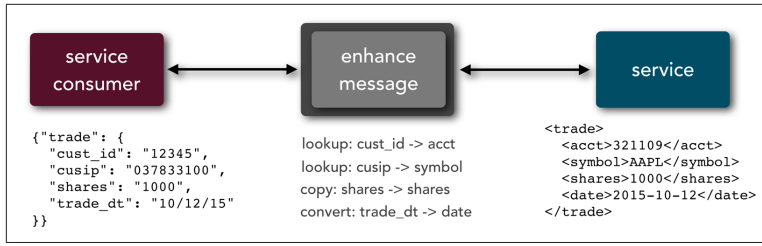

Figure 4-3. Contract decoupling

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

with no impact on 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 CUSIP numbers to one that requires SEDOL numbers should not require all the business applications throughout the enterprise to change to SEDOL numbers.

Unfortunately, microservices must once again take a back seat to SOA with respect to this architecture capability. Microservices architecture does not support contract decoupling, whereas contract decoupling is one of the primary capabilities offered within a SOA. If you require this level of abstraction in your architecture, you will need to look toward a SOA solution rather than a microservices one for your application or system.

CHAPTER 5
Summary

The microservices architecture pattern is a rising star in the IT industry. Although the microservices pattern 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 architecture 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 that 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, SOA, and distributed architecture in general, you can view *Service-Based Architectures: Structure, Engineering Practices, and Migration* (O'Reilly video) by Neal Ford and Mark Richards.

For an excellent in-depth look at microservices, I highly recommend Sam Newman's book *Building Microservices* (O'Reilly).

Finally, for more information about messaging as it relates to service-based architectures for both microservices and SOA, you

can view *Enterprise Messaging: JMS 1.1 and JMS 2.0 Fundamentals* (O'Reilly video) and *Enterprise Messaging: Advanced Topics and Spring JMS* (O'Reilly video).

About the Author

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 Understanding the Basics* (O'Reilly video), *Enterprise Messaging* (O'Reilly video), *Java Message Service, 2nd Edition* (O'Reilly), and a contributing author to *97 Things Every Software Architect Should Know* (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.