Mark Richards

THE CHALLENGES OF SERVICE-BASED ARCHITECTURE

Microservices is taking the IT industry by storm as the go-to style for developing highly scalable and modular applications. While Microservices offers significant advantages over monolithic architectures, there are some significant challenges to consider and overcome before jumping onto the Microservices bandwagon. In this article I will discuss some of the challenges you will face when considering a service-based architecture style and some ways of overcoming those challenges.

Introduction

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. 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.). Service-based 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 service-based 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. Service-based architectures also lend themselves toward more loosely coupled and modular architectures.

Unfortunately, very few things in life are free, and the advantages of service-based 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 article I'll describe some of theses complex issues as they relate to Service-based Architecture and also discuss some of the ways to overcome these issues.

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.

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 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 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 (github.com/realestate-com-au/pact) and Pacto (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 essentially 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 article 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.

The first contract versioning technique involves using contract version numbers in the service contract. Notice in Figure 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.

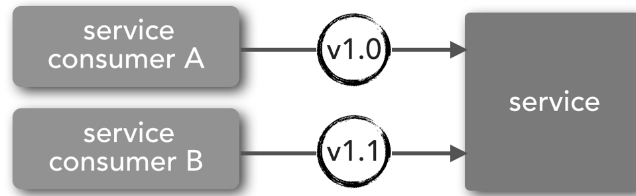

Figure 1. Using Contract Version Numbers

The second technique is to support 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 and Figure 2 in terms of the service contract shape. In Figure 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.

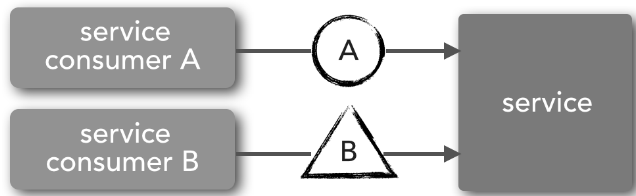

Figure 2. Using Multiple Contract Types

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.

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 3 illustrates this difference.

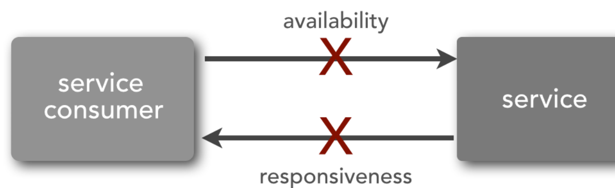

Figure 3. Availability vs. Responsiveness

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, 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 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 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.

With service-based architectures it is extremely difficult to propagate and maintain a transaction context across multiple remote services. As illustrated in Figure 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.

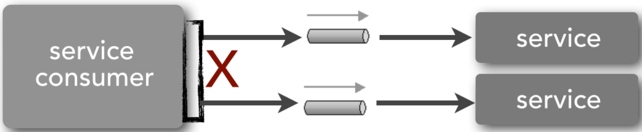

Figure 4. Service Transaction Management

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.

Rather than use ACID transactions, service-based architectures rely on something called BASE transactions. BASE stands for 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.

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.

Conclusion

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 should'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 article 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.

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 recently recorded, which can be found on Safari Online or through the O'Reilly website at 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 shop.oreilly.com/product/0636920033158.do or through Safari Online.

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* (shop.oreilly.com/product/0636920034698.do) and *Enterprise Messaging: Advanced Topics and Spring JMS* (shop.oreilly.com/product/0636920034865.do). Both of these videos are also available through Safari Online.

Finally, for a more in-depth look at these issues as well as a host of other in-depth architecture topics, you can attend a 3-day hands-on architecture training class I am conducting through NFJS. You can get more information about the architecture training courses by going to the NFJS training website at www.nofluffjuststuff.com/n/training/schedule.

About the Author

Mark Richards (www.wmrichards.com) 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* (O'Reilly video), *Enterprise Messaging* (O'Reilly video), *Java Message Service*, 2nd Edition (O'Reilly), and a contributing author of *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.