O'REILLY®

Microservices AntiPatterns and Pitfalls

Mark Richards

Microservices AntiPatterns and Pitfalls

Mark Richards

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Microservices Antipatterns and Pitfalls
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*.

Editor: Brian Foster
Production Editor: Melanie Yarbrough
Copyeditor: Christina Edwards
Proofreader: Amanda Kersey
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

July 2016: First Edition

Revision History for the First Edition
2016-07-06: First Release

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-96331-9

[LSI]

Table of Contents

Preface

In late 2006 service-oriented architecture (SOA) was all the craze. Companies were jumping on the bandwagon and embracing SOA before fully understanding the advantages and disadvantages of this very complex architecture style. Those companies that embarked on SOA projects often found constant struggles with service granularity, performance, data migrations, and in particular the organizational change that comes about with SOA. As a result, many companies either abandoned their SOA efforts or built hybrid architectures that did not fulfill all of the promises of SOA.

Today we are poised to repeat this same experience with a relatively new architecture style known as microservices. Microservices is a current trend in the industry right now, and like SOA back in the mid 2000s, is all the craze. As a result, many companies are looking toward this architecture style to leverage the benefits provided by microservices such as ease of testing, fast and easy deployments, fine-grained scalability, modularity, and overall agility. However, like SOA, companies developing microservices are finding themselves struggling with things like service granularity, data migration, organizational change, and distributed processing challenges.

As with any new technology, architecture style, or practice, antipatterns, and pitfalls usually emerge as you learn more about it and experience the many "lessons learned" during the process. While antipatterns and pitfalls may seem like the same thing, there is a subtle difference between them. Andrew Koenig defines an antipattern as something that seems like a good idea when you begin, but leads you into trouble, whereas my friend Neal Ford defines a pitfall as something that was never a good idea, even from the start. This is an important distinction because you may not experience the negative results from an antipattern until you are well into the development lifecycle or even well into production, whereas with a pitfall you usually find out you are headed down the wrong path relatively quickly.

This report will introduce several of the more common antipatterns and pitfalls that emerge when using microservices. My goal with this report is to help you avoid costly mistakes by not only helping you understand when an antipattern or pitfall is occurring, but more importantly helping you understand the techniques and practices for avoiding these antipatterns and pitfalls.

While I don't have time in this report to cover the details of all of the various antipatterns and pitfalls you might encounter with microservices, I do cover some of the more common ones. These include antipatterns and pitfalls related to service granularity (Chapter 5, *Grains of Sand Pitfall*), data migration (Chapter 1, *Data-Driven Migration AntiPattern*), remote access latency (Chapter 9, *Are We There Yet Pitfall*), reporting (Chapter 4, *Reach-in Reporting AntiPattern*), contract versioning (Chapter 8, *The Static Contract Pitfall*), service responsiveness (Chapter 2, *The Timeout AntiPattern*), and many others.

I recently recorded a video for O'Reilly called *Microservices AntiPatterns and Pitfalls: Learning to Avoid Costly Mistakes* that contains the complete set of antipatterns and pitfalls you may encounter when using microservices, as well as a more in-depth look into each one. Included in the video is a self-assessment workbook containing analysis tasks and goals oriented around analyzing your current application. You can use this assessment workbook to determine whether you are experiencing any of the antipatterns and pitfalls introduced in the video and ways to avoid them.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

I would like to acknowledge several people who helped make this report a success. First, I would like to thank Matt Stine for the technical review of the report. His technical knowledge and expertise in microservices helped to not only validate the various technical aspects of the report, but also to enhance certain chapters with additional insight and material. Next, I would like to thank my friend and partner-in-crime Neal Ford for helping me understand the differences between pitfalls and antipatterns and for also helping me properly classify each of the antipatterns and pitfalls I wrote about in this report. I would also like to thank the editorial staff at O’Reilly for their help in making the authoring process as easy and painless as possible. Finally, I would like to thank my wife and kids for putting up with me on yet another project (albeit small) that takes me away from what I like doing most—being with my family.

CHAPTER 1

Data-Driven Migration AntiPattern

Microservices is about creating lots of small, distributed single-purpose services, with each service owning its own data. This service and data coupling supports the notion of a bounded context and a share-nothing architecture, where each service and its corresponding data are compartmentalized and completely independent from all other services, exposing only a well-defined interface (the contract). This bounded context is what allows for quick and easy development, testing, and deployment with minimal dependencies.

The data-driven migration antipattern occurs mostly when you are migrating from a monolithic application to a microservices architecture. The reason this is an antipattern is that it seems like a good idea at the start to migrate both the service functionality and the corresponding data together when creating microservices, but as you will learn in this chapter, this will lead you down a bad path that can result in high risk, excess cost, and additional migration effort.

There are two primary goals during any microservices conversion effort. The first goal is to split the functionality of the monolithic application into small, single-purpose services. The second goal is to then migrate the monolithic data into small databases (or separate schemas) owned by each service. Figure 1-1 shows what a typical migration might look like when both the service code and the corresponding data are migrated at the same time.

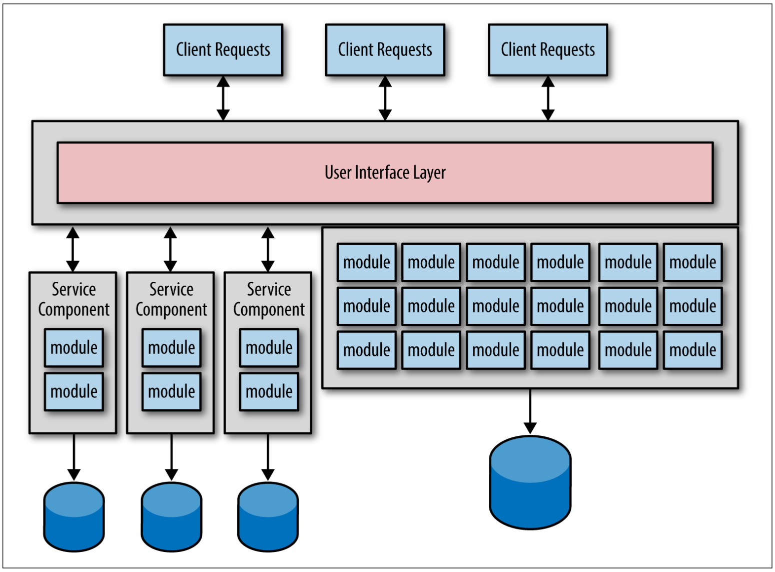

Figure 1-1. Service and data migration

Notice there are three services created from the monolithic application along with three separate databases. This is a natural migration process because you are creating that critical bounded context between each service and its corresponding data. However, problems start to arise with this common practice, thus leading you into the data-driven migration antipattern.

Too Many Data Migrations

The main problem with this type of migration path is that you will rarely get the granularity of each service right the first time. Knowing it is always a good idea to start with a more coarse-grained service and split it up further if needed when you learn more about the service, you may be frequently adjusting the granularity of your services. Consider the migration illustrated in Figure 1-1, focusing on the leftmost service. Let's say after learning more about the service you discover it's too coarse-grained and needs to be split up into two smaller services. Alternatively, you may find that the two leftmost services are too fine-grained and need to be consolidated. In either case you are faced with two migration efforts—one for the service functionality and another for the database. This scenario is illustrated in Figure 1-2.

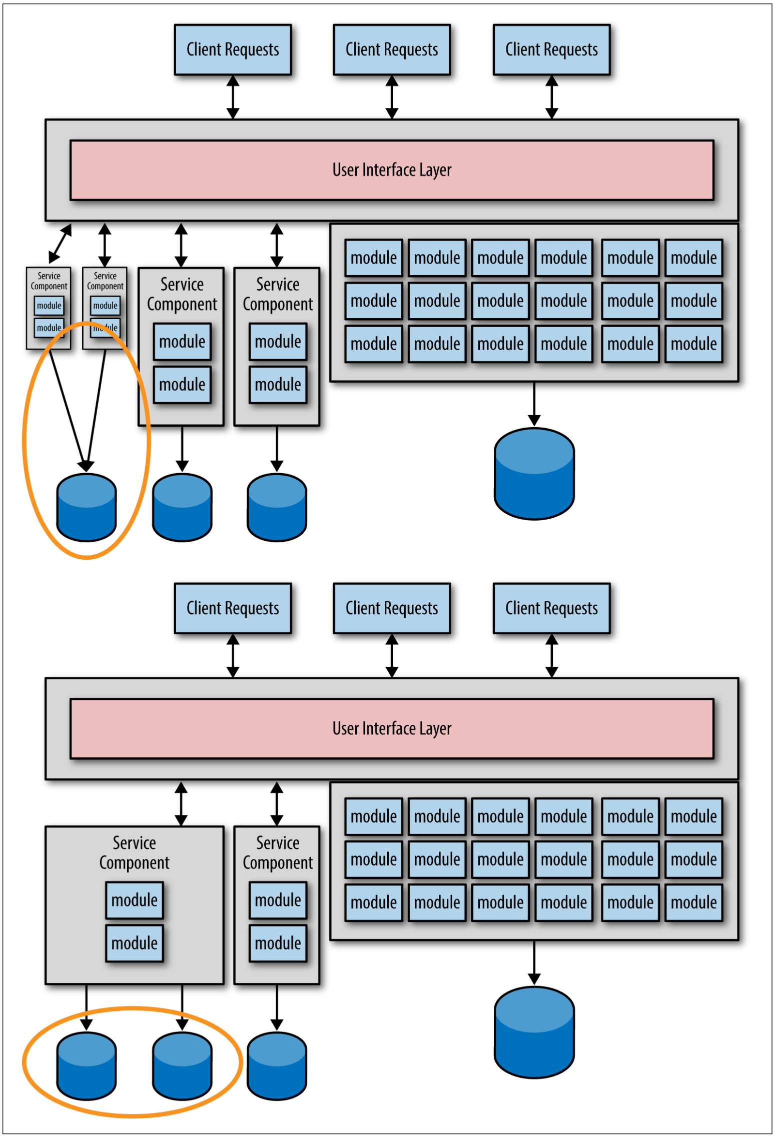

Figure 1-2. Extra data migration after service granularity adjustment

My good friend and fellow O'Reilly author Alan Beaulieu (*Learning SQL*) once told me "Data is a corporate asset, not an application asset." Given Alan's statement, you can gain an appreciation for the risk involved and the concerns raised with continually migrating data. Data migrations are complex and error-prone—much more so than source code migrations. Optimally you want to migrate the data for each service only once. Understanding the risks involved with data migration and the importance of "data over functionality" is the first step in avoiding this antipattern.

Functionality First, Data Last

The primary avoidance technique for this antipattern is to migrate the functionality of the service first, and worry about the bounded context between the service and the data later. Once you learn more about the service you will likely find the need to adjust the level of granularity through service consolidation or service splitting. After you are satisfied that you have the level of granularity correct, then migrate the data, thereby creating the much-needed bounded context between the service and the data.

This technique is illustrated in Figure 1-3. Notice how all three services have been migrated, but are still connecting to the monolithic data. This is perfectly fine for an interim solution, because now you can learn more about how the service is used and what type of requests will be handled by each service.

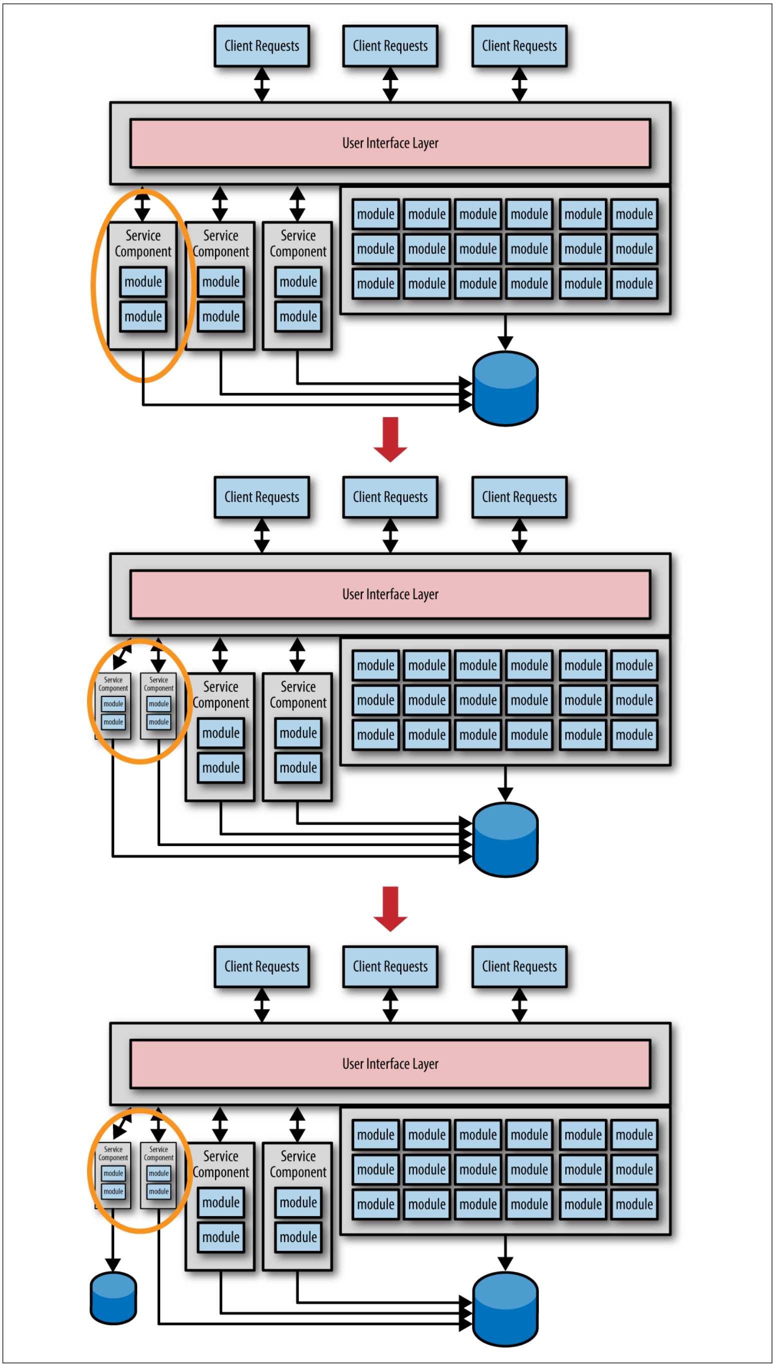

Figure 1-3. Migrate service functionality first, then data portion later

In Figure 1-3, notice how the service was found to be too coarse-grained and was consequently split into two smaller services. Now that the granularity is correct, the data can be migrated to create the bounded context between the service and the corresponding data. This technique avoids costly and repeated data migrations and makes it easier to adjust the service granularity when needed. While it is impossible to say how long to wait before migrating the data, it is important to understand the consequences of this avoidance technique—a poor bounded context. The time between when the service is created and the data is finally migrated creates a data coupling between services. This means that when the database schema is changed, all services using that schema must be coordinated from a change control and release standpoint, something you want to avoid with the microservices architecture. However, this tradeoff is well worth the reduced risk involved with avoiding multiple costly database migrations.

CHAPTER 2

The Timeout AntiPattern

Microservices is a distributed architecture, meaning all of the components (i.e., services) are deployed as separate applications and are accessed remotely through some sort of remote access protocol. One of the challenges of any distributed architecture is managing remote process availability and responsiveness. Although service availability and service responsiveness are both related to service communication, they are two very different things. Service availability is the ability of the service consumer to connect with the service and be able to send it a request, as shown in Figure 2-1. Service responsiveness, on the other hand, is the time it takes for the service to respond to a given request once you've communicated with it.

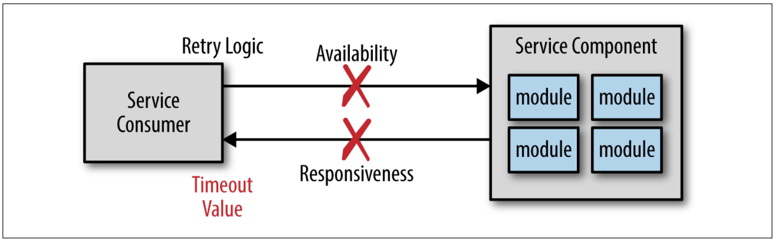

Figure 2-1. Service availability vs. responsiveness

If the service consumer cannot connect with or talk to the service (i.e., availability), the service consumer is usually immediately notified within milliseconds, as Figure 2-1 shows. The service consumer may choose to pass this error onto the client or retry the connection several times before giving up and throwing some sort of connection failure. However, assuming the service was reached and a request was made, what happens if the service doesn't respond? In this case the service consumer can choose to wait indefinitely or leverage some sort of timeout value.

Using a timeout value for service responsiveness seems like a good idea, but can lead you down a bad path known as the timeout anti-pattern.

Using Timeout Values

You might be a bit confused at this point. After all, isn't setting a timeout value a good thing? Maybe, but in most cases it can lead you down a bad path. Consider the example where you are making a service request to buy 1000 shares of Apple stock (AAPL). The very last thing you want to do as the service consumer is time out the request right when the service has successfully placed the trade and is about to give you a confirmation number. You can try to resubmit the trade, but you have to add significant complexity into your service to determine if this is a new trade or a duplicate trade. Furthermore, since you don't have a confirmation number from the first trade it is very difficult to know whether the trade was actually successful or not.

So, given that you don't want to time out the request too early, what should the timeout value be? There are several techniques to address this problem. The first is to calculate the database timeout within the service and use that as a base for determining what the service timeout should be. The second solution, which is by far the most popular technique, is to calculate the maximum time under load and double it, thereby giving you that extra buffer in the event it sometimes takes longer.

Figure 2-2 illustrates this technique. Notice that on average the service responds within 2 seconds to place a trade. However, under load the maximum time observed is 5 seconds. Therefore, using the doubling technique, the timeout value for the service consumer would be 10 seconds. Again, the intention with this technique is to avoid timing out the request when in fact it was successful and was in the process of sending you back the confirmation number.

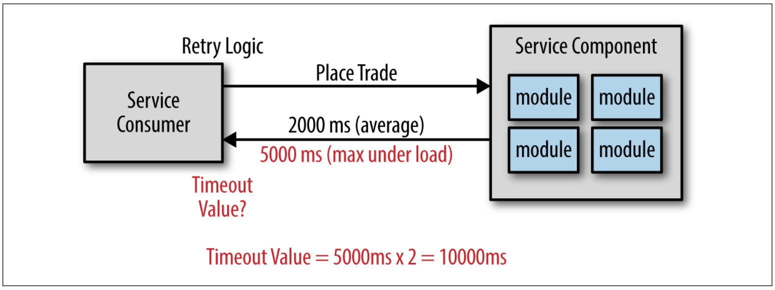

Figure 2-2. Calculating a timeout value

It should be clear now why this approach is an antipattern. While this seems like a perfectly logical solution to the timeout problem, it causes *every request* from service consumers to have to wait 10 seconds just to find out the service is not responsive. Ten seconds is a long time to wait for an error. In most cases users won't wait more than 2 to 3 seconds before hitting the submit button again or giving up and closing the screen. There must be a better way to deal with server responsiveness.

Using the Circuit Breaker Pattern

Rather than relying on timeout values for your remote service calls, a better approach is to use something called the *circuit breaker pattern*. This software pattern works just like a circuit breaker in your house. When it is closed, electricity flows through it, but once it is open, no electricity can pass until the breaker is closed. Similarly, if a software circuit breaker detects that a service is not responding, it will open, rejecting requests to that service. Once the service becomes responsive, the breaker will close, allowing requests through.

Figure 2-3 illustrates how the circuit breaker pattern works. The circuit breaker continually monitors the remote service, ensuring that it is alive and responsive (more on that part later). While the service remains responsive the breaker will be closed, allowing requests through. If the remote service suddenly becomes unresponsive, the circuit breaker opens, thus preventing requests from going through until the service once again becomes responsive. However, unlike the circuit breaker in your house, a software circuit breaker can continue monitoring the service and close itself once the remote service becomes responsive again.

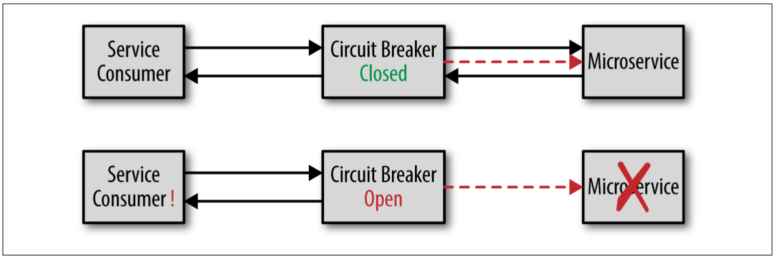

Figure 2-3. Circuit breaker pattern

Depending on the implementation, the service consumer will always check with the circuit breaker first to see if it is open or closed. This can also be done through an interceptor pattern so the service consumer doesn't need to know the circuit breaker is in the request path. In either case, the significant advantage of the circuit breaker pattern over timeout values is that the service consumer knows right away that the service has become unresponsive rather than having to wait for the timeout value. In the prior example, if a circuit breaker was used instead of the timeout value, the service consumer would know within milliseconds that the trade-placement service was not responsive rather than having to wait 10 seconds (10,000 milliseconds) to get the same information.

Circuit breakers can monitor the remote service in several ways. The simplest way is to do a simple heartbeat check on the remote service (e.g., ping). While this is relatively easy and inexpensive, all it does is tell the circuit breaker that the remote service is alive, but says nothing as to the responsiveness of the actual service request. To get better information about the responsiveness of the request you can use synthetic transactions. A synthetic transaction is another monitoring technique circuit breakers can use where a fake transaction is periodically sent to the service (e.g., once every 10 seconds). The fake transaction performs all of the functionality required within that service, allowing the circuit breaker to gain an accurate measure of responsiveness. Synthetic transactions can be very tricky and difficult to implement in that all parts of the application or system need to know about the synthetic transaction. A third type of monitoring is real-time user monitoring, where actual production transactions are monitored for responsiveness. Once a threshold is reached, the breaker moves into what is called a half-open state, where only a certain number of transactions are let through (say 1 out of 10).

Once the service responsiveness goes back to normal, the breaker is then closed, allowing all transactions through.

There are several open source implementations of the circuit breaker pattern, including Hystrix from Netflix and a plethora of GitHub implementations. The Akka framework includes a circuit breaker implementation as part of the framework implemented through the Akka `CircuitBreaker` class.

You can get more information about the circuit breaker pattern through the following resources:

- Michael Nygard's excellent book *Release It!*
- Martin Fowler's circuit breaker blog post
- Microsoft MSDN library

CHAPTER 3
The “I Was Taught to Share” AntiPattern

Microservices is known as a “share-nothing” architecture. Pragmatically, I prefer to think of it as a “share-as-little-as-possible” architecture because there will always be some level of code that is shared between microservices. For example, rather than having a security service that is responsible for authentication and authorization, you might have the source code and security functionality wrapped in a JAR file named *security.jar* that all services use. Assuming security is handled at the services level, this is generally a good practice because it eliminates the need to make a remote call to a security service for every request, thereby increasing both performance and reliability.

However, taken too far, you end up with a dependency nightmare as illustrated in Figure 3-1, where every service is dependent on multiple custom shared libraries.

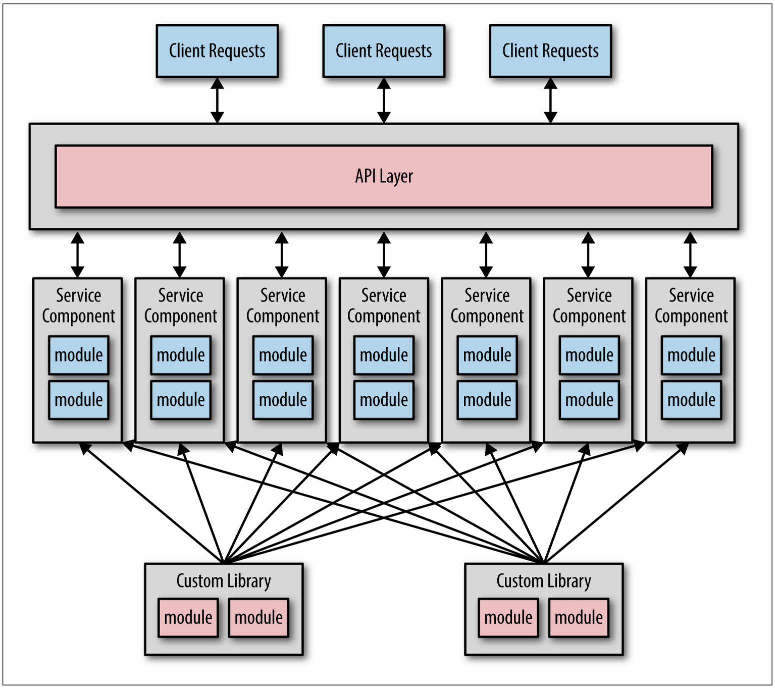

Figure 3-1. Sharing multiple custom libraries

This level of sharing not only breaks down the bounded context of each service, but also introduces several issues, including overall reliability, change control, testability, and deployment.

Too Many Dependencies

If you consider how most object-oriented software applications are developed, it's not hard to see the issues with sharing, particularly when migrating from a monolithic layered architecture to a microservices one. One of the things to strive for in most monolithic applications is code reuse and sharing. Figure 3-2 illustrates the two main artifacts (abstract classes and shared utilities) that end up being shared in most monolithic layered architectures.

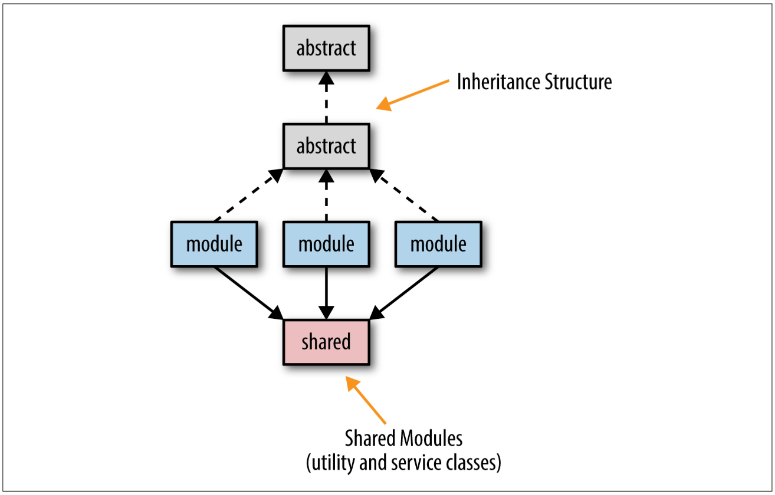

Figure 3-2. Sharing inheritance structures and utility classes

While creating abstract classes and interfaces is a common practice with most object-oriented programming languages, they get in the way when trying to migrate modules to a microservices architecture. The same goes with custom shared classes and utilities such as common date or string utilities and calculation utilities. What do you do with the code that needs to be shared by potentially hundreds of services?

One of the primary goals of the microservices architecture style is to share as little as possible. This helps preserve the bounded context of each service, which is what gives you the ability to do quick testing and deployment. With microservices it all boils down to change control and dependencies. The more dependencies you have between services, the harder it is to isolate service changes, making it difficult to separately test and deploy individual services. Sharing too much creates too many dependencies between services, resulting in brittle systems that are very difficult to test and deploy.

Techniques for Sharing Code

It's easy to say the best way to avoid this antipattern is simply not to share code between services. But, as I stated at the start of this chapter, pragmatically there will always be some code that needs to be shared. Where should that shared code go?

Figure 3-3 illustrates the four basic techniques for addressing the problem of code sharing: shared projects, shared libraries, replication, and service consolidation.

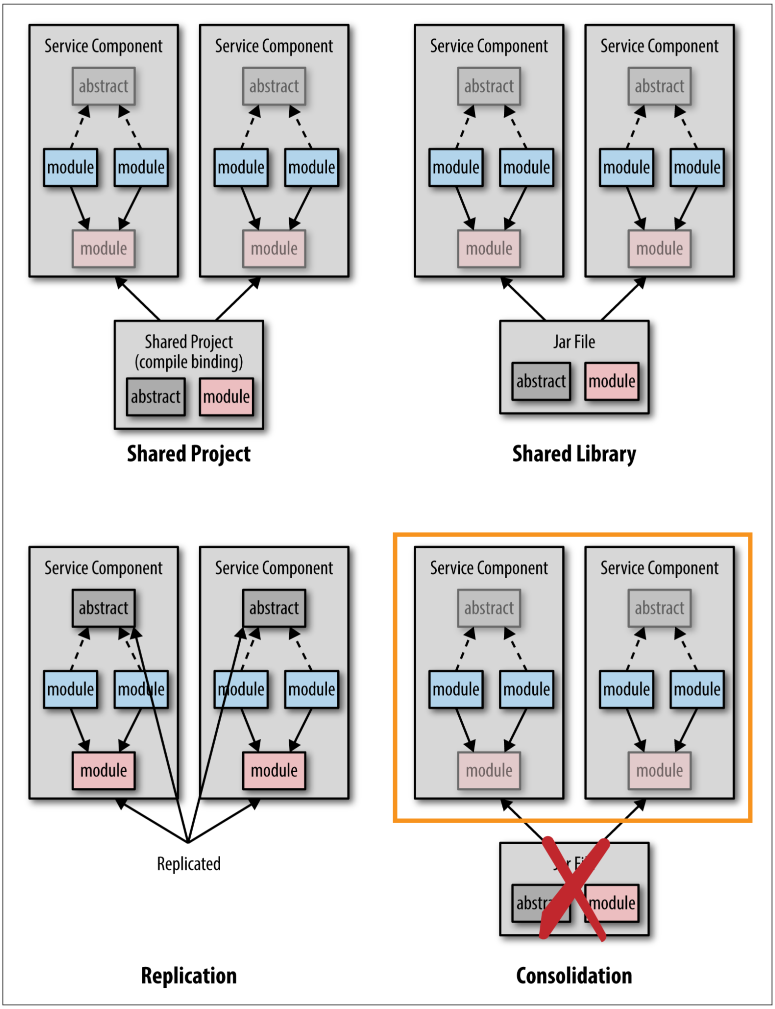

Figure 3-3. Module-sharing techniques

Using a shared project forms a compile-time binding between common source code that is located in a shared project and each service project. While this makes it easy to change and develop software, it is my least favorite sharing technique because it causes potential issues and surprises during runtime, making applications less robust. The main issue with the shared project technique is that of communication and control—it is difficult to know what shared modules changed and why, and also hard to control whether you want that particular change or not. Imagine being ready to release your microservice just to find out someone made a breaking change to a shared module, requiring you to change and retest your code prior to deployment.

A better approach if you have to share code is to use a shared library (e.g., .NET assembly or JAR file). This approach makes development more difficult because for each change made to a module in a shared library, the developer must first create the library, then restart the service, and then retest. However, the advantage of the shared library technique is that libraries can be versioned, providing better control over the deployment and runtime behavior of a service. If a change is made to a shared library and versioned, the service owner can make decisions about when to incorporate that change.

A third technique that is common in a microservices architecture is to violate the don't-repeat-yourself (DRY) principle and replicate the shared module across all services needing that particular functionality. While the replication technique may seem risky, it avoids dependency sharing and preserves the bounded context of a service. Problems arise with this technique when the replicated module needs to be changed, particularly for a defect. In this case all services need to change. Therefore, this technique is only really useful for very stable shared modules that have little or no change.

A fourth technique that is sometimes possible is to use service consolidation. Let's say two or three services are all sharing some common code, and those common modules frequently change. Since all of the services must be tested and deployed with the common module change anyway, you might as well just consolidate the functionality into a single service, thereby removing the dependent library.

One word of advice regarding shared libraries—avoid combining all of your shared code into a single shared library like *common.jar*. Using a common library makes it difficult to know whether you need to incorporate the shared code and when. A better technique is to separate your shared libraries into ones that have context. For example, create context-based libraries like *security.jar*, *persistence.jar*, *dateutils.jar*, and so on. This separates code that doesn't change often from code that changes frequently, making it easier to determine whether or not to incorporate the change right away and what the context of the change was.

CHAPTER 4

Reach-in Reporting AntiPattern

With the microservices architecture style, services and the corresponding data are contained within a single bounded context, meaning that the data is typically migrated to separate databases (or schemas). While this works well for services, it plays havoc with respect to reporting within a microservices architecture.

There are four main techniques for handling reporting in a microservices architecture: the database pull model, HTTP pull model, batch pull model, and finally the event-based push model. The first three techniques pull data from each of the service databases, hence the antipattern name "reach-in reporting." Since the first three models represent the problem associated with this antipattern, let's take a look at those techniques first to see why they lead you into trouble.

Issues with Microservices Reporting

The problem with reporting is two-fold: how do you obtain reporting data in a timely manner and still maintain the bounded context between the service and its data? Remember, the bounded context within microservices includes the service and its corresponding data, and it is critical to maintain it.

One of the ways reporting is typically handled in a microservices architecture is to use what is known as the *database pull model*, where a reporting service (or reporting requests) pulls the data directly from the service databases. This technique is illustrated in Figure 4-1.

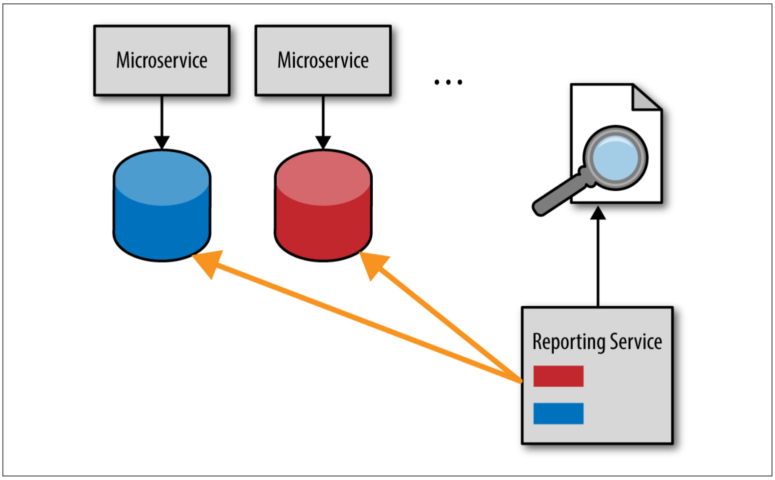

Figure 4-1. Database pull-reporting model

Logically, the fastest and easiest way to get timely data is to access it directly. While this may seem like a good idea at the time, it leads to significant interdependencies between services and the reporting service. This is a typical implementation of the shared database integration style, which couples applications together through a shared database. This means that the services no longer own their data. Any service database schema change or database refactoring must include reporting service modifications as well, breaking that important bounded context between the service and the data.

The way to avoid the issue of data coupling is to use another technique called the *HTTP pull model*. With this model, rather than accessing each service database directly, the reporting service makes a restful HTTP call to each service, asking for its data. This model is illustrated in Figure 4-2.

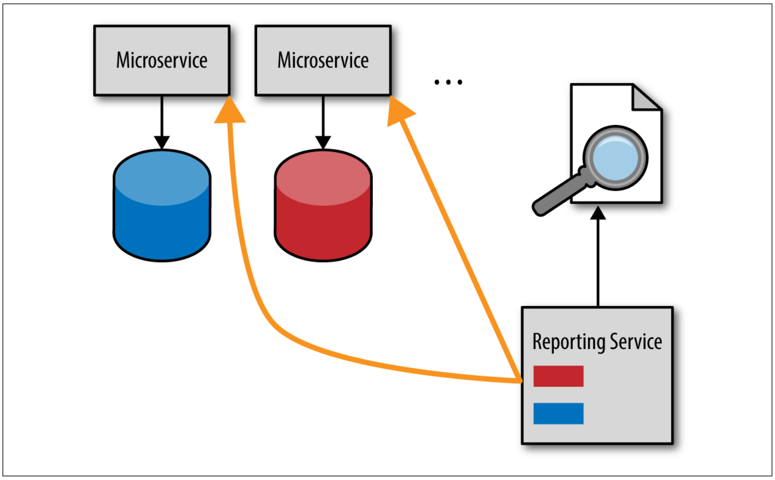

Figure 4-2. HTTP pull-reporting model

While this model preserves the bounded context of each service, it is unfortunately too slow, particularly for complex reporting requests. Furthermore, depending on the report being requested, the data volume might be too large of a payload for a simple HTTP call.

A third option in response to the issues associated with the HTTP pull model is to use the batch pull model illustrated in Figure 4-3. Notice that this model uses a separate reporting database or data warehouse that contains the aggregated and reduced reporting data. The reporting database is usually populated through a batch job that runs in the evening to extract all reporting data that has changed, aggregate and reduce that data, and insert it into the reporting database or data warehouse.

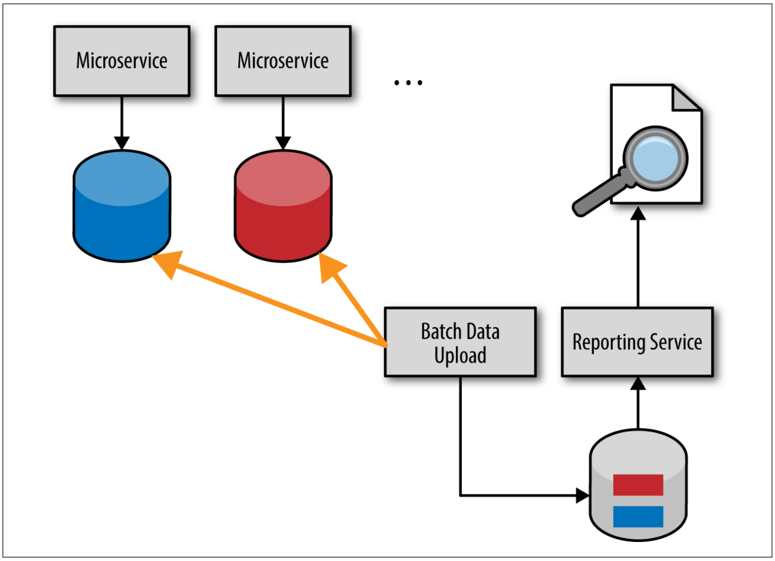

Figure 4-3. Batch pull-reporting model

The batch pull model shares the same issue with the HTTP pull model—they both implement the shared database integration style—therefore breaking the bounded context of each service. If the service database schema changes, so must the batch data upload process.

Asynchronous Event Pushing

The solution for avoiding the reach-in reporting antipattern is to use what is called an *event-based push model*. Sam Newman, in his book *Building Microservices*, refers to this technique as a data pump. This model, which is illustrated in Figure 4-4, relies on asynchronous event processing to make sure the reporting database has the right information as soon as possible.

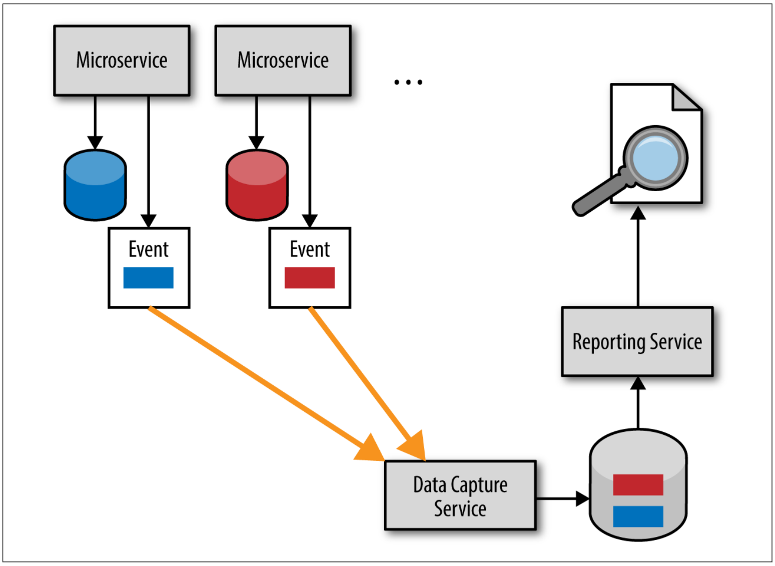

Figure 4-4. Event-based push-reporting model

While it is true that the event-based push model is relatively complex to implement, it does preserve the bounded context of each service while at the same time ensuring a reasonable timeliness of data. Like the batch pull model, this model also has a separate reporting database owned by the reporting service. However, rather than a batch process pulling data, each microservice asynchronously sends its notable data updates (e.g., the data the reporting service needs) as a separate event to a data-capture service, which then reduces the data and updates the reporting database.

The event-based push model requires a contract between each microservice and the data capture service for the data it is asynchronously sending, but that contract is separate from the database schema owned by the service. However, the services are somewhat coupled in that each service must know when to send what information for reporting purposes.

In the chart in Figure 4-5, you can see that the database pull model maximizes on timeliness of data, but breaks the bounded context. The HTTP pull model preserves the bounded context, but has issues associated with timeouts and data volume. The batch pull model turns out to be the least-desirable model out of the four options because optimizes neither the bounded context nor the timeliness of data. Only the event-based push model maximizes both the bounded context of each service and the timeliness of reporting data.

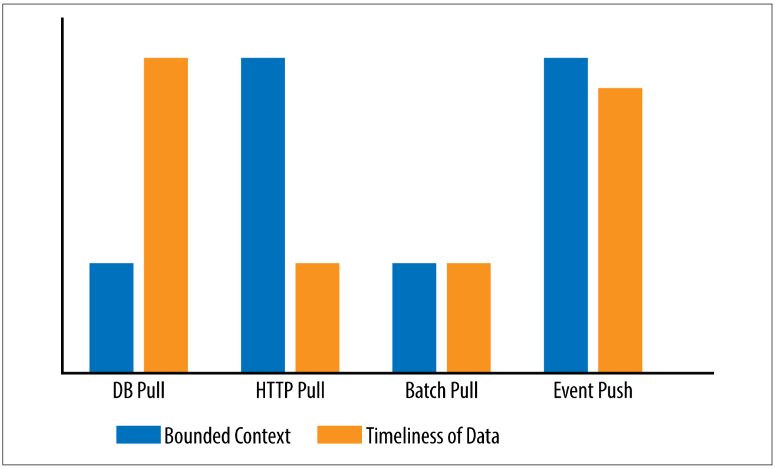

Figure 4-5. Comparing reporting models

CHAPTER 5
Grains of Sand Pitfall

Perhaps one of the biggest challenges architects and developers face when creating applications using a microservices architecture is service granularity. How big should a service be? How small should it be? Choosing the right level of granularity for your services is critical to the success of any microservices effort. Service granularity can impact performance, robustness, reliability, change control, testability, and even deployment.

The *grains of sand pitfall* occurs when architects and developers create services that are too fined-grained. Wait—isn't that why it's called *micro*services in the first place? The word "micro" implies that a service should be very small, but how small is "small"?

One of the primary reasons this pitfall occurs is because developers often confuse a *service* with a *class*. Too many times I've seen development teams create services by thinking that the implementation class they're writing is actually the service. Nothing could be further from the truth.

A service should always be thought of as a *service component*. A service component is a component of the architecture that performs a specific function in the system. The service component should have a clear and concise roles and responsibility statement and have a well-defined set of operations. It is up to the developer to decide *how* the service component should be implemented and how many implementation classes are needed for the service.

As Figure 5-1 shows, a service component is implemented through one or more modules (e.g., Java classes). Implementing a service component using a one-to-one relationship between a module and a service component not only lends itself toward components that are too fine-grained, it also leads to poor programming practices as well. Services implemented through a single class tend to have classes that are too big and carry too much responsibility, making them hard to maintain and test.

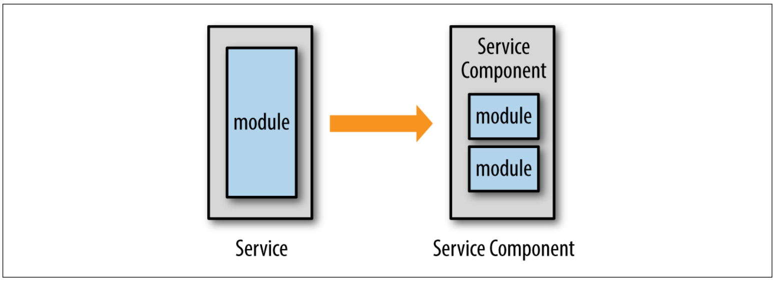

Figure 5-1. Relationship between modules and a service

The number of implementation classes should not be a defining characteristic for determining the granularity of a service. Some services may only need a single class file to implement all of the business functionality, whereas others may need six or more classes.

If the number of implementation classes has no impact on the granularity of a service, then what does? Fortunately, there are three basic tests you can use to determine the right level of granularity for your services: the service scope and functionality, the need for database transactions, and finally the level of service choreography.

Analyzing Service Scope and Function

The first way to determine whether your services have the right level of granularity is to analyze the scope and function of the service. What does the service do? What are its operations? Documenting or verbally stating the service scope and function is a great way to determine if the service is doing too much. Using words like "and" and "in addition" is usually a good indicator that the service is probably doing too much.

Cohesion also plays a role with regards to the service scope and function. Cohesion is defined as the degree and manner to which the operations of the service are interrelated. You want to strive for strong cohesion within your services. For example, let's say you have a customer service with the following operations:

- add_customer
- update_customer
- get_customer
- notify_customer
- record_customer_comments
- get_customer_comments

In this example the first three operations are interrelated as they all pertain to maintaining and retrieving customer information. However, the last three (notify_customer, record_customer_comments, and get_customer_comments) do not relate to basic CRUD operations on basic customer data. In analyzing the level of cohesion of the operations in this service, it becomes clear that the original service should perhaps be split into three separate services (customer maintenance, customer notification, and customer comments).

Figure 5-2 illustrates the point that, in general, when analyzing the service scope and function you will likely find that your services are too coarse-grained and you will move toward services that are more fine-grained.

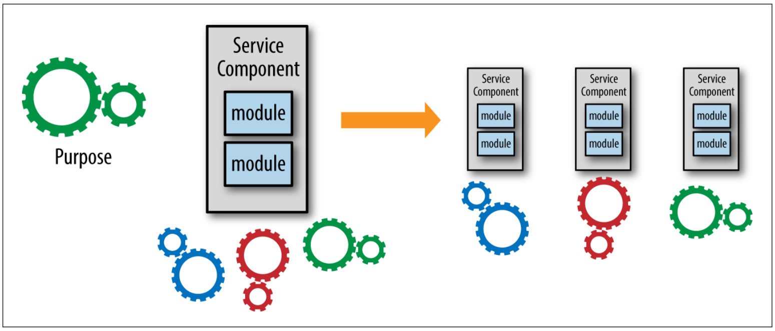

Figure 5-2. Impact of analyzing service functionality and scope

Sam Newman offers some good solid advice in this area—start out more coarse-grained and move to fine-grained as you learn more about the service. Following this advice will help you get started in defining your service components without having to worry so much about the granularity right away.

While analyzing the service scope and functionality is a good start, you don't want to stop there. After looking at the service scope, you need to then analyze your database transaction needs.

Analyzing Database Transactions

Another test for validating the level of service granularity is the need for database transactions for certain operations. Database transactions are more formally referred to as ACID transactions (atomicity, consistency, isolation, and durability). ACID transactions coordinate multiple database updates into a single unit of work. The database updates are either committed as a whole unit or rolled back if an error condition occurs.

Because services in a microservices architecture are distributed and deployed as separate applications, it is extremely difficult to maintain an ACID transaction between two or more remote services. For this reason, microservices architectures generally rely on a technique known as BASE transactions (basic availability, soft state, and eventual consistency). Regardless, there will usually be times where you do require an ACID transaction for certain business operations. If you find you are constantly battling issues surrounding ACID vs. BASE transactions and you need to coordinate multiple updates, chances are you have made your services too fine-grained.

When analyzing your transaction needs and find that you can't live with eventual consistency you will generally move from fine-grained services to more coarse-grained ones, thereby keeping multiple updates coordinated within a single service context, as illustrated in Figure 5-3.

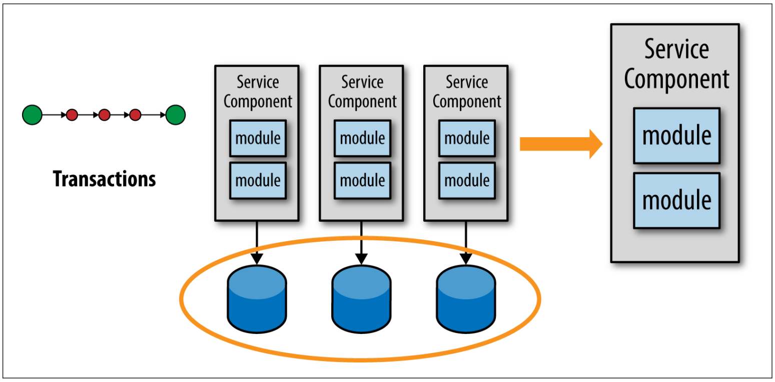

Figure 5-3. Impact of analyzing database transactions

Notice that from an ACID transaction standpoint it doesn't matter whether you consolidate the separate databases or keep them as individual ones. Generally you will want to consolidate the databases as well, but this is not a requirement to maintain an ACID transaction (assuming the databases and the transaction manager you are using support XA—e.g., two-phase commit—transactions).

Once you have analyzed your transaction needs, it's time to move on to the third test, service choreography.

Analyzing Service Choreography

A third test you can use to validate the level of service granularity is *service choreography*. Service choreography refers to the communication between services, also commonly referred to as interservice communication. Service choreography is generally something you want to be careful of within in a microservices architecture. First of all, it decreases the overall performance of your application since each call to another service is a remote call. For example, assuming it takes 100 milliseconds to make a restful call to another service, making five remote service calls is a half a second spent *just in remote access time*.

The other issue with too much service choreography is that it can impact the overall reliability and robustness of your system. The more remote calls you make for a single business request, the better the chances are that one of those remote calls will fail or time out.

If you find you are having to communicate with too many services to complete single business requests, then you've probably made your services too fine-grained. When analyzing the level of service choreography, you will generally move from fine-grained services to ones that are more coarse-grained, as illustrated in Figure 5-4.

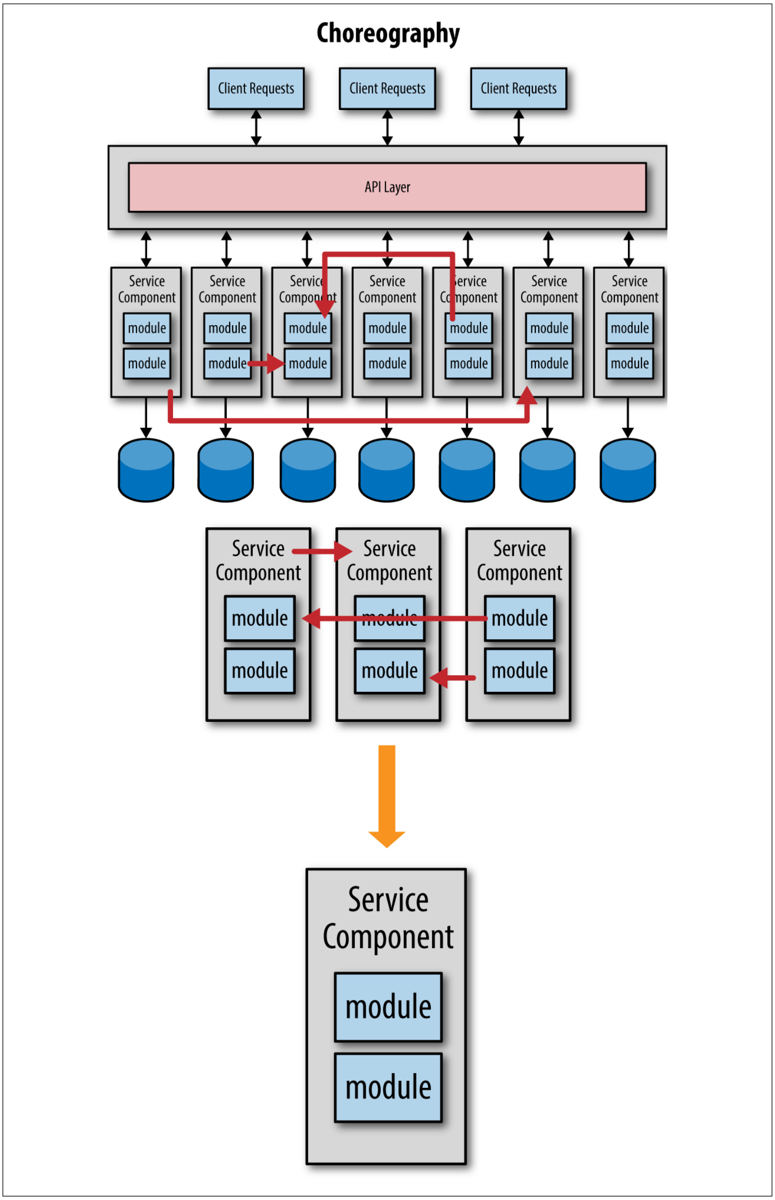

Figure 5-4. Impact of analyzing service choreography

By consolidating services and moving to more coarse-grained services you can improve performance and increase the overall reliability and robustness of your applications. You also remove dependencies between services, allowing for better change control, testing, and deployment.

The other approach when dealing with service choreography to help overcome the performance and reliability issues is to leverage asynchronous parallel processing combined with reactive architecture techniques for error handling. Executing multiple requests at the same time increases overall responsiveness, allowing you to coordinate multiple services in a single business request in a timely fashion. The key point here is to understand and analyze the trade-offs associated with service choreography to ensure both sufficient responsiveness to the user and sufficient overall reliability of your system.

CHAPTER 6

Developer Without a Cause Pitfall

I first saw James Dean in the movie *Rebel Without a Cause* when I was just a young lad, but I still remember everything about the movie. When thinking about a name for this antipattern I immediately thought of James Dean—a troubled young man who made decisions for the wrong reasons. Perfect.

I have observed more times that I can count architects and developers making decisions about various aspects of microservices, particularly with regards to service granularity and devops tools, for all the wrong reasons. It all boils down to tradeoffs. Rich Hickey says "Programmers know the benefits of everything and the tradeoffs of nothing." My friend Neal Ford likes to follow up on Rich's quote by saying "Architects must understand both." I maintain that developers should know both as well.

Making the Wrong Decisions

Figure 6-1 illustrates one common scenario where services are discovered to be too fine-grained, therefore impacting performance and overall reliability due to the amount of interservice communication between them. In this scenario, the developer or architect makes the decision that these services should be consolidated into a single, more coarse-grained service to address the performance and reliability issues.

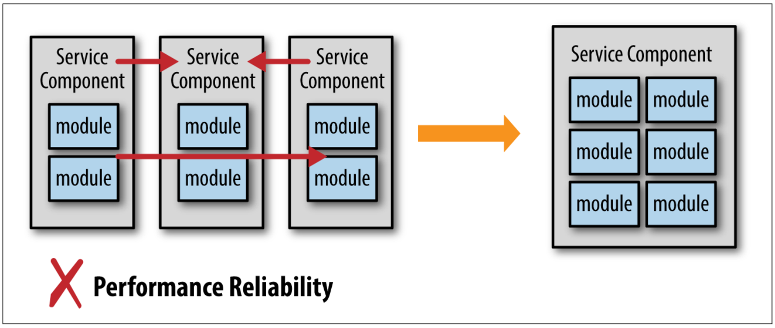

Figure 6-1. Moving from fine-grained to coarse-grained

While this seems like a reasonable decision, the tradeoff of doing this is ignored. Deployment, change control, and testing are all impacted by moving to a single coarse-grained service. The question is, what is most important?

Consider the example illustrated in Figure 6-2 where the reverse situation occurs. In this scenario services are too coarse-grained, therefore impacting the overall testing effort and coordination for deployment. In this case the architect or developer makes the decision that the service should be split up into smaller services to reduce the scope of each service, therefore making them easier to test and deploy.

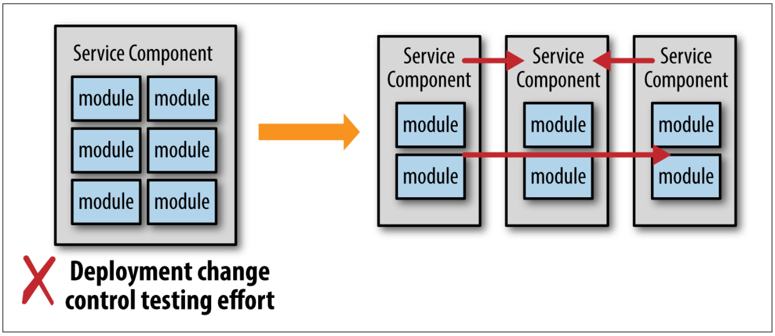

Figure 6-2. Moving from coarse-grained to fine-grained

While we might applaud the architect or developer for making this decision, the tradeoffs are once again forgotten. While services are certainly easier to test and deploy with this change, the application suddenly experiences issues with performance and reliability due to an increase in service choreography. Which is more important?

Understanding Business Drivers

Understanding the business drivers behind choosing microservices is the key to avoiding this pitfall. Every architect and developer on the team should know the answer to each of the following questions:

- Why are you doing microservices?
- What are the primary business drivers?
- What architecture characteristics are most important?

Using deployability, performance, robustness, and scalability as the primary architecture characteristics, consider the following scenarios where the business driver is known. Notice how the business drivers are what drive the decision regarding service consolidation or service splitting, not the characteristics themselves.

Scenario 1: The reason for moving to microservices is to achieve better time to market via an effective deployment pipeline.

In this scenario the deployability of each service outweighs performance, reliability, and scalability, so with this business driver you will tend to create more finer-grained services, trading off a potential increase in service choreography (and consequently impacts on performance and reliability). Referring back to Figure 6-1, given this driver the developer would have actually made the wrong decision to consolidate services.

Scenario 2: The reason for moving to microservices is to increase the overall reliability and robustness of the application.

This scenario is a common reason for companies moving from monolithic applications to a microservices architecture, primarily due to issues with monolithic architectures surrounding tight coupling and hence brittle applications. In this scenario the business driver clearly states the need for reliability and robustness, meaning that you would likely trade off ease of testing and deployment for better reliability and robustness, therefore favoring more coarse-grained services rather than finer-grained ones.

One technique I frequently use is to write the business drivers in big red letters on the top of the common team whiteboard as illustrated in Figure 6-3. Then, anytime there is a decision on service granularity or tool selection, the team can always look up, refer to the whiteboard, and say "oh, yeah, that's right. Okay, let's keep the services fine-grained and figure out another way to address the performance and reliability issues."

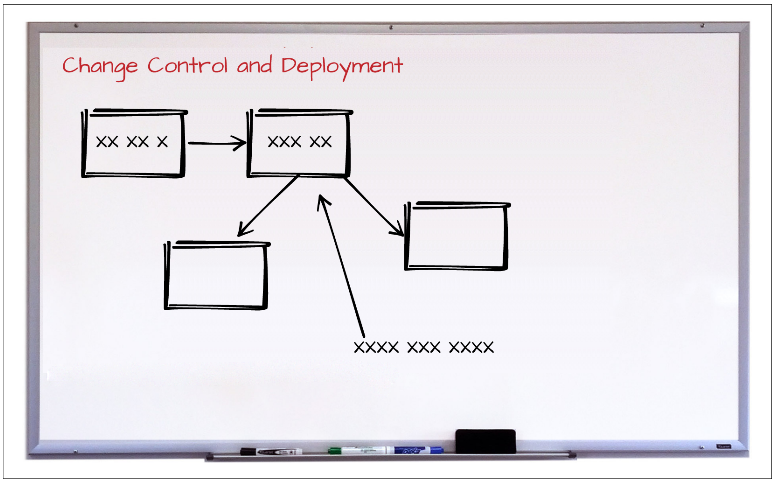

Figure 6-3. Put business drivers on the whiteboard

CHAPTER 7

Jump on the Bandwagon Pitfall

You *must* embrace microservices. It's undeniably the latest trend in the industry, everyone else is doing it, and besides, it's great to have on your resume.

The *jump on the bandwagon pitfall* is all about embracing microservices before analyzing your business needs, business drivers, and overall organizational structure and technology environment. While the microservices architecture is a very powerful and popular architecture style, it's not suited for every application or environment.

You can easily avoid this pitfall by first understanding the advantages and disadvantages of microservices. Then, once you've gained a full understanding of what microservices is all about, you can match your business needs and goals to the architectural characteristics to determine if microservices is a fit for your situation and organization. You can also avoid this pitfall by learning more about other architecture patterns that may be a better fit for your situation.

Advantages and Disadvantages

The first step in avoiding this pitfall is to understand the advantages and disadvantages of the microservices architecture style. The following are some of the more important advantages you should know about:

Deployment
: Ease of deployment is one of the primary drivers for moving to a microservices architecture. Microservices are small, single-

purpose services deployed as separate applications. It is significantly easier and far less risk to deploy a single service than an entire monolithic application. As a matter of fact, the whole notion of continuous delivery is in part what prompted the creation of the microservices architecture style.

Testability

Ease of testing is another big advantage of the microservices architecture. The small scope of a service coupled with the lack of shared dependencies with other services makes them relatively easy to test. However, perhaps one of the more significant aspects of this characteristic is that with microservices you have the ability to do more complete regression testing than with bigger monolithic applications.

Change control

With microservices it is easier to control what gets changed when adding new functionality. This is again due to the limited service scope and the bounded context maintained by each service. Having small, independent services with few interdependencies means less coordination for developing, testing, and releasing changes.

Modularity

Microservices is a highly modular architecture style, which in turn leads to highly agile applications. Agility is best defined as the ability to respond quickly to change. The more modular an architecture, the faster the ability to develop, test, and release changes. The microservices architecture style is perhaps the most modular architecture out of all the architecture patterns due to the fine level of service granularity.

Scalability

Because microservices are fine-grained single-purpose services that are separately deployed, this architecture style boasts the highest level of scalability out of all the architecture patterns. It is relatively easy to scale out a particular piece of functionality with the microservices architecture style, in part due to the containerized nature of the service topology and sophisticated monitoring tools that allow you to start and stop services dynamically through automation.

While these advantages might convince you that microservices is the best solution for your situation, consider the following list of disadvantages.

Organizational change

Microservices requires organizational change at many levels. Development teams must be restructured and reorganized into more cross-functional teams so that small teams can own the end-to-end technical aspects of the services they are responsible for, including the user interface, backend processing, rules processing, and database processing and modeling. The traditional corporate development team model of user interface teams, backend development teams, and database engineers/administrators simply doesn't work with a microservices architecture. In addition, the organizational structures involved with releasing software must also change. With microservices it is not feasible to use the traditional software development lifecycle procedures that exist with monolithic, layered architectures. Rather, you must embrace automation and leverage devops tools and practices to develop an effective deployment pipeline for releasing microservices.

Performance

Because every microservice is a separately deployed application, communication to and from services, as well as communication between services, is remote. Performance can be significantly impacted depending on your environment and the amount of service choreography you have in your microservices application. It is important to understand your remote access latency (see "Are We There Yet Pitfall") and also how much service communication you will need (see *Grains of Sand Pitfall*) to fully understand the performance impacts of using microservices.

Reliability

For the same reasons that performance can be impacted by using microservices, the same is true with overall reliability. Because every request is a remote access call, you run the risk that one of the services you need to communicate with to complete a single business request is not available or fails to respond.

DevOps

With the microservices architecture you can have anywhere from hundreds to even thousands of microservices. Due to the large number of services you might have, it is simply not feasible to manually manage hundreds of concurrent release cycles and deployments. Automation and continuous collaboration between developers, testers, and release engineers is vital to the success of any microservices endeavor. For this reason you need to embrace various operations-related tools and practices, which can be a very complicated task. There are about 12 different categories of operations-related tools and frameworks used within a microservices architecture, and each of those categories contains several dozen tool and product choices. For example, there are monitoring tools, service registry and discovery tools, deployment tools, and so on. Which ones are best for your environment and situation? The answer to this question requires several months of research, proof-of-concept efforts, and tradeoff analysis to determine the best combination of tools and frameworks for your application and environment.

Matching Business Needs

After understanding the advantages and disadvantages of the microservices architecture style, you must then analyze your business needs and goals to determine if microservices is the right approach for the problem you are trying to solve. When determining whether microservices is a fit, ask yourself the following questions:

- What are my business and technical goals?
- What am I trying to accomplish with microservices?
- What are my current and foreseeable pain points?
- What are the primary driving architecture characteristics for this application (e.g., performance, scalability, maintainability, etc.)?

Answering these questions can help you match up your business needs and goals with the advantages and disadvantages of microservices to determine if it is truly the right fit for your situation.

Other Architecture Patterns

The microservices architecture style is a very powerful one that carries with it many advantages, but it isn't the only architecture style out there. Another thing you can do to avoid this pitfall is to understand and analyze other architecture patterns to determine if one of those might be a better fit for your situation.

Besides microservices there are seven other common architecture patterns you might want to consider for your application or system:

- Service-Based Architecture
- Service-Oriented Architecture
- Layered Architecture
- Microkernel Architecture
- Space-Based Architecture
- Event-Driven Architecture
- Pipeline Architecture

Of course, you don't have to select one single architecture pattern for your application. You can certainly combine patterns to create an effective solution. Some examples are event-driven microservices, event-based microkernel, layered space-based architecture, and pipeline microkernel.

Use the following resources to learn more about other architecture patterns:

- *Software Architecture Fundamentals: Understanding the Basics*
- *Software Architecture Fundamentals: Beyond the Basics*
- *Software Architecture Fundamentals: Service-Based Architecture*
- *Software Architecture Patterns*)
- *Microservices vs. Service-Oriented Architecture*

CHAPTER 8

The Static Contract Pitfall

All microservices have contracts between the service consumers and the microservice. A contract usually contains a schema specifying the expected input and output data, and sometimes the name of the operation (depending on how you are implementing your service). Contracts are usually owned by the service, and can be represented through formats like XML, JSON, or even a Java or C# object. And of course, those contracts never change, right? Wrong.

The static contract pitfall occurs when you fail to version your service contracts from the very start, or even not at all. Contract versioning is absolutely critical for not only avoiding breaking changes (changing a contract and breaking all consumers using that contract), but also to maintain agility by supporting backward compatibility.

Here's an example that illustrates how you can get into trouble by not versioning your contracts. Assume you have a microservice that is accessed by three different clients (client 1, client 2, and client 3). Client 1 would like to make a change to the service contract right away. You check with client 2 and client 3 to see if they can accommodate the change, and both clients inform you that it will take weeks to implement that change due to other things going on with those clients. Now you must inform client 1 that it will take weeks to make that change because you need to coordinate the update with clients 2 and 3. However, client 1 cannot wait weeks.

By providing versioning in your contracts, and hence providing backward compatibility, you can now be more agile in terms of client 1's request. Agility is defined as how fast you can respond to change. If you properly versioned your contracts from the very start, you could immediately respond to client 1's request for the contract change by simply creating a new version of the contract, say version 1.1. Clients 2 and 3 are both using version 1.0 of the contract, so now you can implement the change right away without having to wait for client 2 or client 3 to respond. In addition, you can make the change without making what is called a "breaking change."

There are two basic techniques for contract versioning: versioning at the header level and versioning in the contract schema itself. In this chapter I will cover each of these techniques in detail, but first let's look at an example.

Changing a Contract

To illustrate the problem with not versioning a contract I will use an example of buying a certain number of shares of Apple common stock (AAPL). The schema for this request might look something like this:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "properties": {
      "acct": {"type": "number"},
      "cusip": {"type": "string"},
      "shares": {"type": "number", "minimum": 100}
   },
    "required": ["acct", "cusip", "shares"]
}
```

In this case to buy stock you must specify the brokerage account (*acct*), the stock you wish to purchase in CUSIP (Committee on Uniform Security Identification Procedures) format (*cusip*), and finally the number of shares (*shares*), which must be greater than 100. All three fields are required.

The code to make a request to purchase 1000 shares of Apple stock (CUSIP 037833100) for brokerage account 12345 using REST would look like this:

```
POST /trade/buy
Accept: application/json
{ "acct": "12345",
  "cusip": "037833100",
  "shares": "1000" }
```

Now let's say that the service changes its contract to accept a SEDOL (Stock Exchange Daily Official List) rather than a CUSIP, which is another industry standard way of identifying a particular instrument to be traded. Now the contract looks like this:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "properties": {
      "acct": {"type": "number"},
      "sedol": {"type": "string"},
      "shares": {"type": "number", "minimum": 100}
   },
    "required": ["acct", "sedol", "shares"]
}
```

This would be considered a breaking change in that the prior client code will now fail because it is still using a CUSIP. What you need to do is use versioning so that version 1 uses a CUSIP and version 2 uses a SEDOL to identify the stock being traded.

Header Versioning

The first technique for contract versioning is to put the contract version number in the header of the remote access protocol as illustrated in Figure 8-1. I like to refer to this as *protocol-aware contract versioning* because the information about the version of the contract you are using is contained within the header of the remote access protocol (e.g., REST, SOAP, AMQP, JMS, MSMQ, etc.).

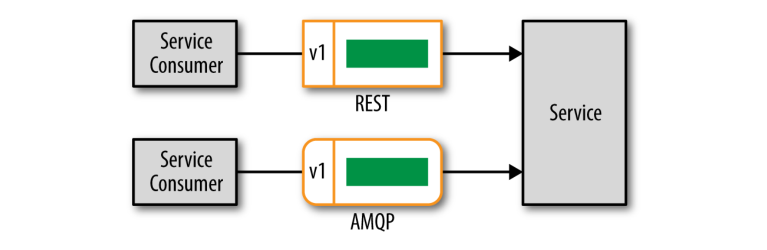

Figure 8-1. Header contract versioning

When using REST you can use what is called a *vendor mime type* to specify the version of the contract you wish to use in the accept header of the request:

```
POST /trade/buy
Accept: application/vnd.svc.trade.v2+json
```

By using the vendor mime type (vnd) in the accept header of the URI you can specify the version number of the contract, thereby directing the service to perform processing based on that contract version number. Correspondingly, the service will need to parse the accept header to determine the version number. One example of this would be to use a regular expression to find the version as illustrated below:

```
def version
   request.headers
   ["Accept"][/^application/vnd.svc.trade.v(d)/, 1].to_i
end
```

Unfortunately that is the easy part; the hard part is coding all of the cyclomatic complexity into the service to provide conditional processing based on the contract version (e.g., if version 1 then... else if version 2 then...). For this reason, we need some sort of version-deprecation policy to control the level of cyclomatic complexity you introduce into each service.

Using messaging you will need to supply the version number in the property section of the message header. For JMS 2.0 that would look something like this:

```
String msg = createJSON(
  "acct","12345",
  "sedol","2046251",
  "shares","1000")};

jmsContext.createProducer()
.setProperty("version", 2)
.send(queue, msg);
```

Each messaging standard will have its own way of setting this header. The important thing to remember here is that regardless of the messaging standard, the version property is a string value that needs to match exactly with what the service is expecting, including being case-sensitive. For this reason it's generally not a good idea to supply a default version if the version number cannot be found in the header.

Schema Versioning

Another contract-versioning technique is adding the version number to the actual schema itself. This technique is illustrated in Figure 8-2. I usually refer to this technique as *protocol-agnostic con-*

tract versioning because the version identification is completely independent of the remote access protocol. Nothing needs to be specified in the headers of the remote access protocol in order to use versioning.

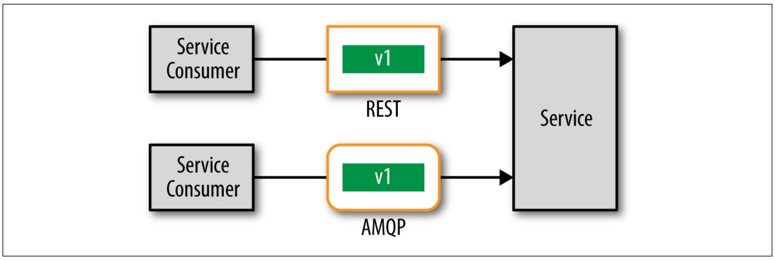

Figure 8-2. Schema-based contract versioning

By using schema-based versioning, the schema used in the previous example would look like this:

```
{
    "$schema": "http://json-schema.org/draft-04/schema#",
    "properties": {
      "version": {"type": "integer"},
      "acct": {"type": "number"},
      "cusip": {"type": "string"},
      "sedol": {"type": "string"},
      "shares": {"type": "number", "minimum": 100}
   },
    "required": ["version", "acct", "shares"]
}
```

Notice that the the schema actually contains the version number field (*version*) as an integer value. Because you only have one schema now, you will need to add all of the combinations of possibilities to the schema. In the example above both the CUSIP and SEDOL are added to the schema because that is what varies between the versions.

The big advantage of this technique is that the schema (including the version) is independent of the remote access protocol. This means that the same exact schema can be used by multiple protocols. For example, the same schema can be used by REST and JMS 2.0 without any modifications to the remote access protocol headers:

```
POST /trade/buy
Accept: application/json
{ "version": "2",
  "acct": "12345",
  "sedol": "2046251",
  "shares": "1000" }


String msg = createJSON(
  "version","2",
  "acct","12345",
  "sedol","2046251",
  "shares","1000")};
jmsContext.createProducer().send(queue, msg);
```

Unfortunately this technique has a lot of disadvantages associated with it. First, you must parse the actual payload of the message to extract the version number. This precludes using things like XML appliances (e.g., DataPower) to do routing, and also might present issues when trying to parse the schema (particularly with XML). Secondly, the schemas can get quite complex, making it difficult to do automated conversions of the schema (e.g., JSON to Java object). Finally, custom validations may be required in the service to validate the schema. In the example above, the service would have to validate that either the CUSIP or SEDOL is filled in based on the version number.

CHAPTER 9

Are We There Yet Pitfall

With the microservices architecture every service is deployed as a separate application, meaning all of the communication to a microservice from the client or API layer, as well as communication between services, requires a remote call.

This pitfall occurs when you don't know how long the remote access call takes. You might assume the latency it around 50 milliseconds, but have you ever measured it? Do you know what the average latency is for your particular environment? Do you know what the "long tail" latency is (e.g., 95, 99, 99.5 percentiles) for your environment? Measuring both of these metrics is important, because even with good average latency, bad long-tail latency can destroy you.

Measuring Latency

Measuring the remote access latency under load in your production environment (or production-like environment) is critical for understanding the performance profile of your application. For example, let's say a particular business request requires the coordination of four microservices. Assuming that your remote access latency is 100 milliseconds, that particular business request would consume 500 milliseconds just in remote access latency alone (the initial request plus four remote calls between services). That is a half a second of request time without one single line of source code being executed for the actual business request processing. Most applications simply cannot absorb that sort of latency.

You might think the way to avoid this pitfall is to simply measure and determine your average latency for your chosen remote access protocol (e.g., REST). However, that only provides you with one piece of information—the average latency of the particular remote access protocol you are using. The other task is to investigate the comparative latency using other remote access protocols such as Java Message Service (JMS), Advanced Message Queuing Protocol (AMQP), and Microsoft Message Queue (MSMQ).

Comparing Protocols

The comparative latency will vary greatly based on both your environment and the nature of the business request, so it is important to establish these benchmarks on a variety of business requests with different load profiles.

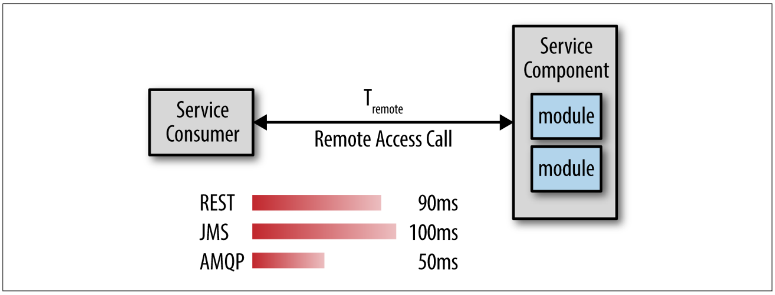

Figure 9-1. Comparing remote access latency

In looking at the hypothetical example in Figure 9-1 you notice that AMQP is in fact almost twice as fast as REST. You can now leverage this information to make intelligent choices as to which requests should use which remote access protocol. For example, you may choose to use REST for all communications from client requests to your microservices and AMQP for all interservice communication in order to increase performance within your application.

Performance is not the only consideration when selecting your remote access protocol. As you will see in Chapter 10, you may want to leverage messaging to provide additional capabilities to your application as well.

CHAPTER 10
Give It a Rest Pitfall

Using REST is by far the most popular choice for accessing microservices and communicating between services. It's so common a choice that most of the popular template frameworks (e.g., DropWizard, Spring Boot, etc.) have REST access already built into the service templates. If REST is such a popular choice, then why is it a pitfall? The give it a rest pitfall is about using REST as the only communication protocol and ignoring the power of messaging to enhance your microservices architecture. For example, in a RESTful microservices architecture, how would you handle asynchronous communications? What about the need for broadcast capabilities? What do you do if you need to manage multiple remote RESTful calls within a transactional unit of work?

There are two types of messaging standards you should be aware of when considering using messaging for your microservices architecture—platform-specific standards and platform-independent standards. Platform-specific standards include the JMS for the Java platform and MSMQ for the .NET platform. Both describe a standard API used within the platform, independent of the messaging provider (vendor) you are using. For example, in the Java platform you can swap out brokers (e.g., ActiveMQ, HornetQ, etc.) with no API changes. While the API is standard and remains the same, it's the underlying proprietary protocol between these brokers that is different (which is why you need to have the same client JAR and server JAR for the same vendor). With platform-standard messaging protocols you are more concerned about portability via a common

API rather than the actual vendor product you are using or the wire-level protocols used.

The current platform-independent standard is AMQP. AMQP, which standardizes on the wire-level protocol, not the API. This allows heterogeneous platforms to communicate with one another, regardless of the vendor product you are using. A client using RabbitMQ, for example, can easily communicate with a StormMQ server (assuming they are using the same protocol version). AMQP using RabbitMQ is currently the most popular choice for messaging within a microservices architecture, mostly because of its platform-independent nature.

Asynchronous Requests

The first consideration for using messaging within your microservices architecture is asynchronous communication. With asynchronous requests the service caller does not need to wait for a response from the service when making a request, as illustrated in Figure 10-1. This is sometimes referred to as "fire-and-forget" processing.

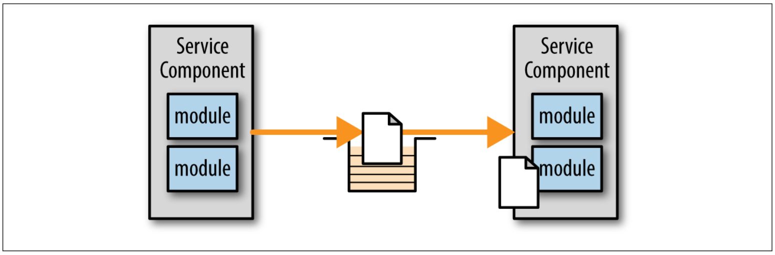

Figure 10-1. Asynchronous communications using messaging

Not only does asynchronous processing increase overall performance, but it also adds an element of reliability to your system. Performance is increased because callers don't have to wait for a response if none is needed. Through guaranteed delivery, the message broker ensures that the service will eventually receive the message. Reliability is increased because the caller doesn't need to worry about setting timeout values or using the circuit breaker pattern when communicating with a service (see Chapter 2).

Broadcast Capabilities

Another very powerful feature of messaging that is not available within REST is the capability to broadcast a message to multiple services. This is known in messaging as "publish-and-subscribe" messaging, and usually involves topics and subscribers (depending on the messaging standard you are using). Figure 10-2 illustrates the basic behavior of broadcast messaging.

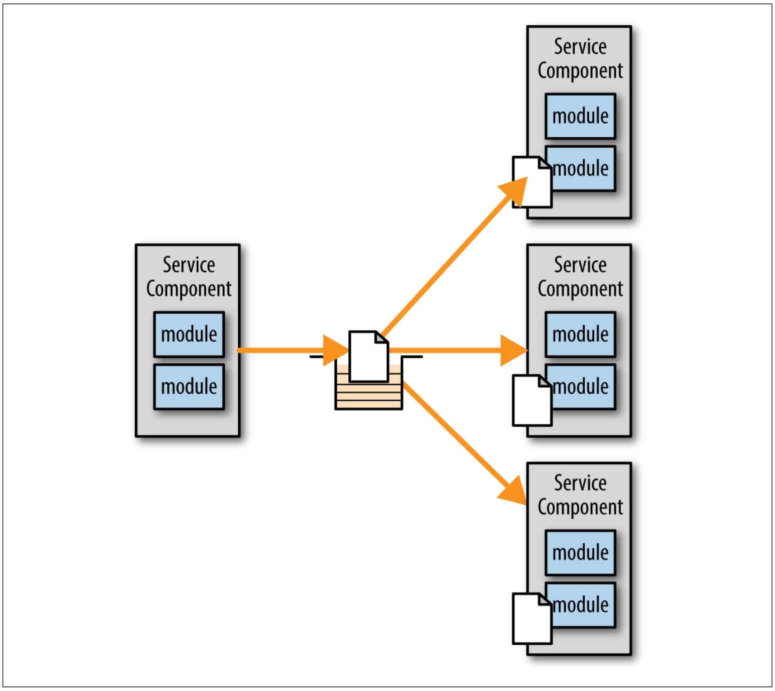

Figure 10-2. Broadcast capabilities using messaging

Broadcast messaging involves a message producer sending out the same message to multiple message receivers (i.e., services). The message producer generally doesn't know who is accepting the message or what they are going to do with it. For example, a message producer may broadcast a message informing consumers about a stock split for Apple stock (AAPL). The message producer only has the responsibility of publishing a message to a topic (JMS), a fanout or topic exchange (AMQP), or a multicast queue (MSMQ). The stock split message may be picked up by any number of consumers, or no consumers at all.

Transacted Requests

Messaging systems support the notion of transacted messages, meaning that if messages are sent to multiple queues or topics within the context of a transaction, the messages are not actually received by the services until the sender does a commit on that transaction. The service consumer sends a message to the first service and then sends another message to the second service, as illustrated in Figure 10-3. Until the service consumer performs a commit, those messages are held in the queues. Once the service consumer performs a commit, both messages are then released.

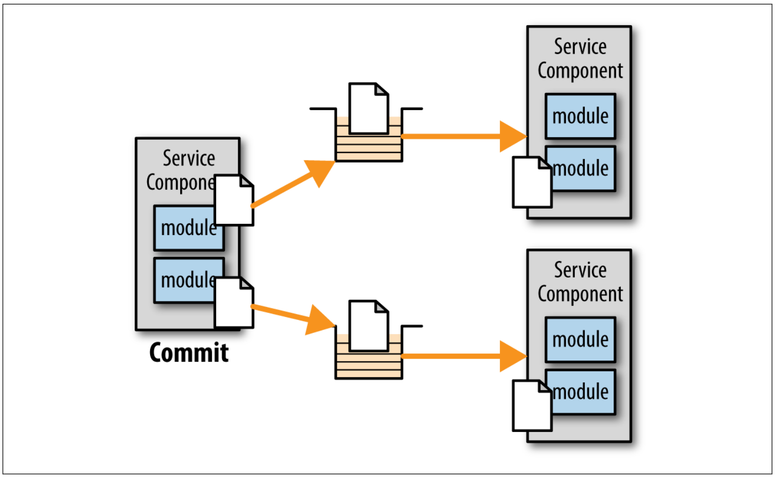

Figure 10-3. Transaction capabilities of messaging

If the service consumer in Figure 10-3 sends a message to the first queue, but then experiences some sort of error, the service consumer can perform a rollback on the messaging transaction, which would effectively remove the message from the first queue.

Implementing this sort of transaction capability using REST would be very difficult, essentially requiring the service consumer to issue compensating requests to reverse the updates made by each request. Therefore, it is a good idea to consider using transacted messaging any time a service consumer needs to orchestrate multiple remote requests.

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 the *Software Architecture Fundamentals Video Series* (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 teaching architecture fundamentals classes and hiking in the White Mountains of New Hampshire and along the Appalachian Trail.