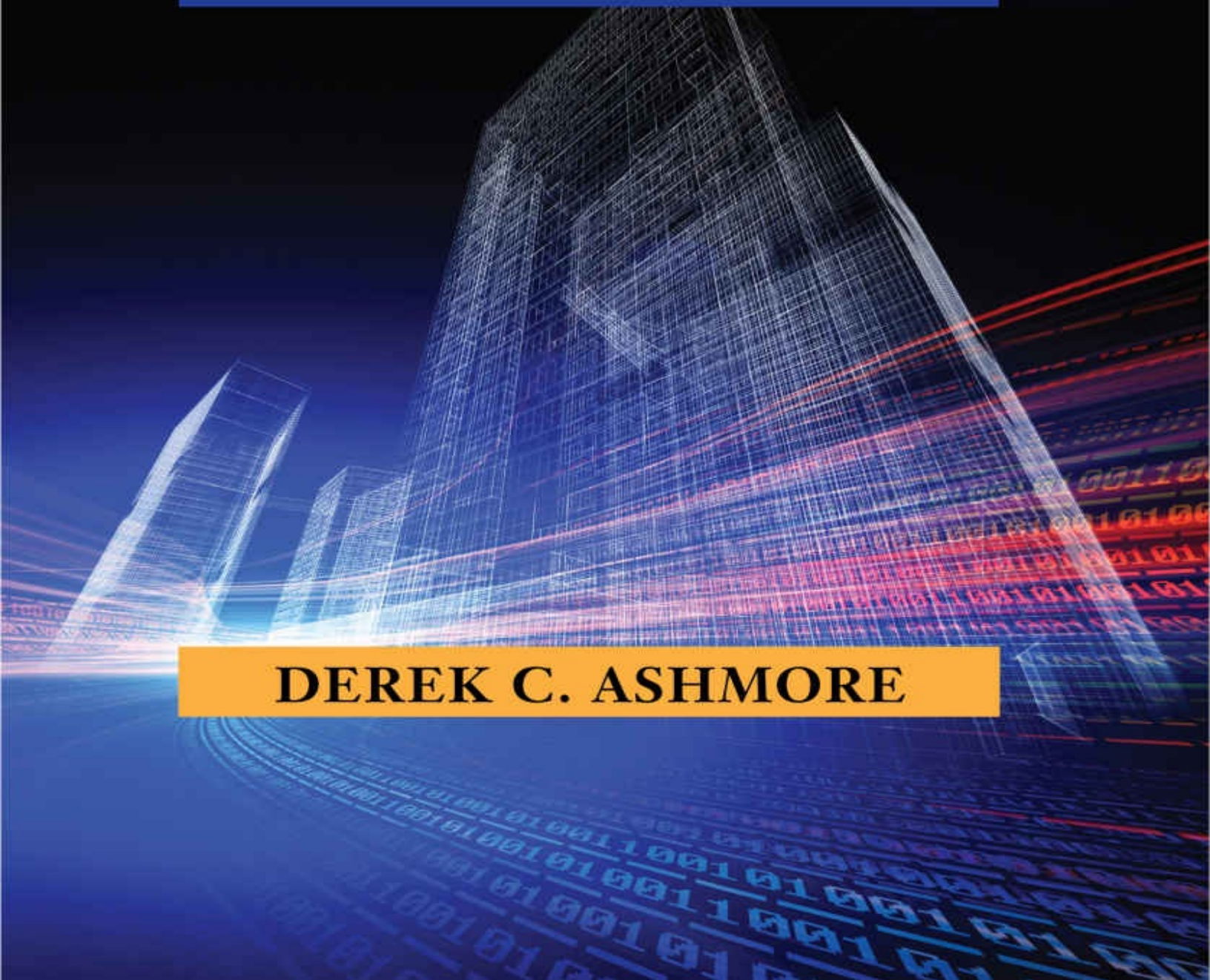
MICROSERVICES for JAVA EE Architects

Addendum for *The Java EE Architect's Handbook*



DEREK C. ASHMORE

Microservices for Java Architects: Addendum for the Java EE Architect's Handbook

By Derek C. Ashmore

Editor: Cathy Reed
Cover Design: The Roberts Group
Interior Design: The Roberts Group

Published by:
DVT Press
Bolingbrook, IL
sales@dvtpress.com
http://www.dvtpress.com

Table of Contents

Preface

Readers of the *Java EE Architect's Handbook*, Second Edition are well acquainted with traditional application architectures. Since this book was published, microservices architecture has emerged. It is an important architecture with a variety of uses that Java architects should understand and be able to use.

This book details microservices architecture and is an addendum to the *Java EE Architect's Handbook*, Second Edition. This book will define microservices architecture and provide an overview of costs and benefits. As with the Architect's Handbook, I'll go on to address architecture, design, and implementation concerns specific to microservices architecture.

This book will take the reader to the point of producing a microservice. Beyond that deployable, there's the very interesting world of DevOps that addresses how to deploy, manage, and cluster that microservice and connect it to the resources it needs to function. While this is a very interesting world and one that I encourage you to take an interest in, it is out of scope for this book.

How this book is organized

The first chapter defines microservices architecture and contrasts it with a layered web application architecture described in the *Java EE Architect's Handbook*. I'll summarize the benefits and costs of using microservices architecture. I'll also discuss similarities and differences between microservices and service oriented architectures (SOA).

The second chapter dives into design considerations for microservices. All new paradigms have coding patterns, and microservices architecture is no exception. I'll detail coding patterns to enhance performance and to increase resiliency to service failure. I'll also discuss how to make microservices easier to support.

The third chapter discusses cross-cutting concerns for microservices and various ways of incorporating them into services written in Java. I'll discuss easy ways to package and deploy microservices as well as to instrument them for health checks and performance measurement.

The fourth chapter discusses when using microservices architecture is appropriate. I also identify current marketing hype surrounding microservices architecture and the fine print required to reap the benefits from it. We're going

to find out that microservices architecture is not a silver bullet and nor should it be used for all applications.

The fifth chapter discusses various topics needed to effectively manage a large number of microservices. Much has been written about how microservices architecture changes the way we design and develop applications; but very little has been written on effectively *managing* large numbers of microservices. As application architects, we advise management and we need to help guide them through the transition to managing microservices architecture.

Description of common resources

This book often makes references to the following open-source projects that are frequently used with many Java EE applications:

- Apache HttpClient (https://hc.apache.org/)
- Apache CXF (https://cxf.apache.org/)
- Docker (https://www.docker.com/)
- Spring Boot (http://projects.spring.io/spring-boot/)
- Dropwizard (http://www.dropwizard.io/)
- Hysterix {https://github.com/Netflix/Hystrix}
- SoapUI (http://www.soapui.org/)
- Google Guava Core Libraries (https://code.google.com/p/guava-libraries/)
- Apache Commons Lang (http://commons.apache.org/lang/)
- My GitHub (https://github.com/Derek-Ashmore)

Another open-source project on which this book relies is Moneta, which is an open-source example of a microservice written in Java. Moneta illustrates several of the concepts discussed in the book.

Feedback

I'm always interested in reading comments and suggestions that will improve future editions of this book. Please send feedback directly to me at derek.ashmore@dvtconsulting.com. If your comment or suggestion is the first of its kind and is used in the next edition, I will gladly send you a copy of my next book. Additionally, reader questions are sometimes selected and answered in entries in my blog at http://www.derekashmore.com/.

About the Author

Derek Ashmore is a senior technology expert with over 25 years of experiences in a wide variety of technologies and industries. His past roles include Application Architect, Cloud Architect, Project Manager, Application Developer, and Database Administrator. He has extensive experience with custom application development as well as integrating applications with commercial products such as Great Plains, Salesforce, Microsoft Dynamics, DocuSign, and others. Derek has been designing and leading Java-related projects since 1999.

Derek is also the author of the *The Java EE Architect's Handbook, Second Edition*, which is also available from DVT Press.

If you need a project assessment for your Java-related project, Derek can be retained by contacting sales@dvtconsulting.com.

Derek can be contacted the following ways:

- Email: derek.ashmore@dvtconsulting.com
- LinkedIn: http://www.linkedin.com/in/derekashmore
- Facebook: https://www.facebook.com/JavaEEArchitectHandbook
- Twitter: https://twitter.com/Derek_Ashmore

Acknowledgments

This book could not have been written without a large amount of assistance. Several colleagues helped tremendously to refine the book concept and edit the drafts, thus keeping me from mass marketing mediocrity. They have my undying gratitude. I could not have written the book without the assistance of the following people: Budi Kurniawan, David Arthurs, David Greenfield, James Gibbings, Ken Liu, Michael Spicuzza, Peeyush Maharshi, Pradeep Sadhu, Rick Miller, Scott Kramer, Scott Wheeler, and Silvio de Morais.

Many thanks to my editor, Cathy Reed, who painstakingly corrected my numerous grammar mistakes and graciously pointed out places where my verbiage was less than clear. Many thanks to Sherry and Tony Roberts and their colleagues at the Roberts Group for fantastic cover art, typesetting, indexing, and general wizardry to make the book aesthetically pleasing.

Many thanks to my wife Beth and my children Zachary and Morgan. They

routinely put up with music coming out of the office at all hours of the night while I was working on this, and Beth routinely watched our children so that I could devote time to writing this.

I retain responsibility for any errors that remain.

Chapter 1: Microservices Defined

The microservices architecture doesn't have a concrete definition. Microservices architecture organizes applications as a suite of discrete, independent services. Usually, these web services adopt a RESTful style. Any coupling between services is restricted to the service layer, with no coupling at a code or database level. The approach does have some similarities to the Service Oriented Architecture (SOA) approach, which will be discussed later. We define it as a series of common traits that 'microservices' have.

Microservices have a single functional purpose. That is, they have a single *business* purpose. They do one thing and do it well. To level set, the term 'functional' is borrowed from the term 'functional requirement.' **Functional** requirements are typically business requirements; they support doing business directly in some way. In essence, the terms 'functional' and 'business' can be used interchangeably.

As a byproduct of having a single functional purpose, most business changes affect a small number of services (preferably only one service). In fact, we're going to see that it's a warning sign that your microservices are not well defined if single feature change requires a large number of service deployments.

As a byproduct of microservices having a single functional purpose, microservice teams tend to be smaller and more focused. This is a byproduct of microservices architecture as the code base is typically much smaller than with traditional web applications. We're going to discover that a smaller code base isn't a goal of microservices architecture directly, but rather it is a byproduct of having such a targeted business focus.

Furthermore, the single functional purpose of a microservice should be self-contained and autonomous. It represents a complete unit of work that can be executed in any context as long as its inputs are correct and its resources available. My use of the phrase 'unit of work' should not be taken to imply a single database transaction. We will learn that a functional unit of work can in fact consist of several service calls, each of which has separate database transactions. Said another way, it doesn't make ***any*** assumptions about what services have been called before it and what's to be called after it.

All microservices have non-functional cross-cutting concerns, such as providing

health checks, exception handling/logging, and performance metrics. These concerns are separate from the functional purpose of the service. Also, these items should be provided by the application architecture and re-used from service to service organization wide. I'll show you how for services written using Java.

Microservices have a standard technology-agnostic interface. Most use RESTful web services, but other standard interfaces work as long as they are technology agnostic (i.e., it doesn't matter if the service was written in Java, .Net, or anything else). SOAP services are technology-agnostic and fit this requirement.

Technology agnostic interfaces are needed to preserve the tech stack independence benefit for using microservices. It should not matter if a particular microservice is written using Java, .Net, Go, Php, Python, or anything else. We're going to learn that technology agnosticism makes it easier to migrate to newer technologies or new versions of existing technologies, and that it means much less lock-in for your technical stack.

The implications are that Java EJB or RMI services don't qualify as a microservice interface as they are Java-specific. Remember that one of the objectives of microservices architecture is freedom to switch technical stacks. Yes, that means potentially writing microservices with development platforms other than Java.

Microservices run in an independent process. In a Java world, that means they run as an independent JVM. With traditional applications, it's not uncommon to deploy several web applications into the same container; but that doesn't happen for microservices. They need to be independent so they can scale to their individual needs. For example, one service might only need a three-node cluster to satisfy consumer demand while another service might need more nodes in its cluster. One service might need more memory per node in the cluster and need to be hosted differently than other services.

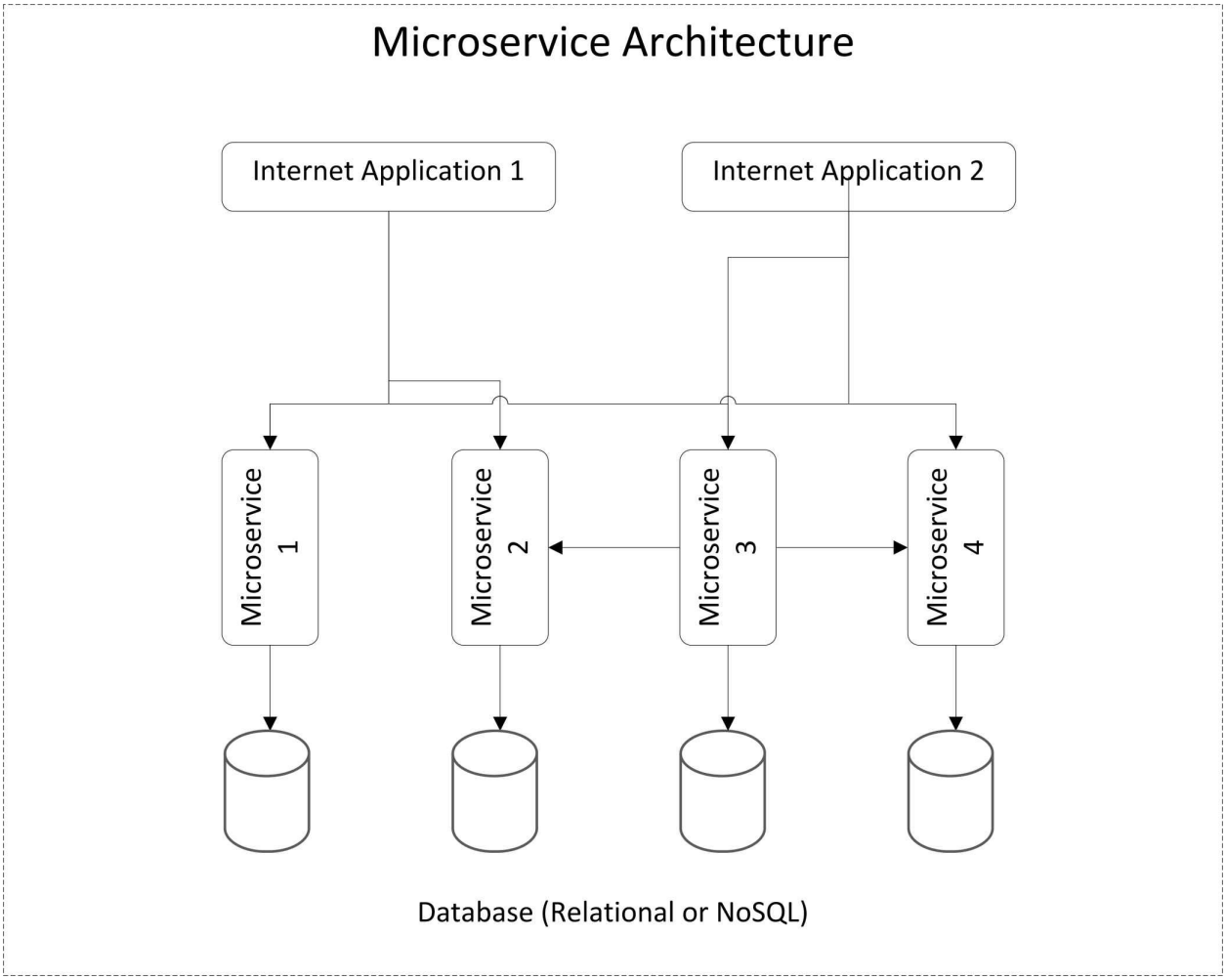
Microservices also run in an independent process so they can be independently deployed or replaced without affecting other services. Your stereo system is a good analogy, even though it's more hardware than software. When I replaced my speakers, none of the other components (e.g., receiver, monitor, Blu-ray player, etc.) were affected in any way. As long as their inputs and resources were

correct, they did their job. The same was true when I replaced my receiver. Microservices should have the same level of independence and self-sufficiency.

Another analogy that people use to describe microservices is the Linux operating system and its extremely componentized utilities. Yes, Linux uses standard input/output instead of web services, but it's still a standardized interface. A good example is the ls utility. It lists directory entries. Yes, it has many options, such as filtering or the format of its output, but it has one and only one job.

Microservices are often combined to provide full functionality needed by applications. As microservices have such a narrow focus, no single microservice can provide all the functionality needed for most applications. In fact, we're going to see that it's common for microservices to rely on other microservices to fulfill their service contract. Figure 1.1 graphically describes how microservices are leveraged to provide full application functionality. The arrows in the diagram represent call usage patterns.

Figure 1.1: Microservice Architecture



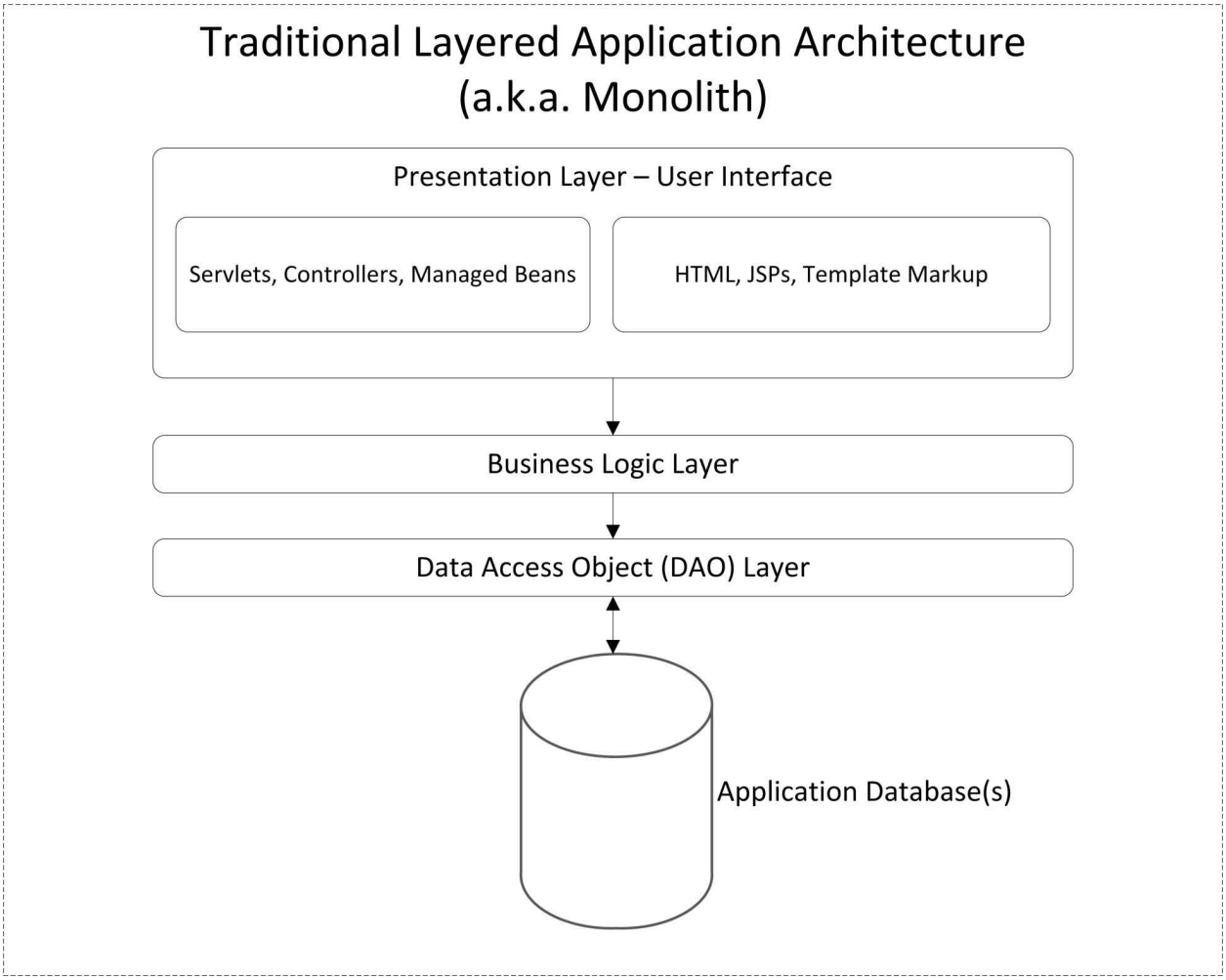
We're going to see that each microservice has its own data store or database. This eliminates coupling at the database level and ensures that database structure changes don't impact more than one service. We're going to see that the autonomous nature of microservices allow technology choices.

For an excellent article on the definition of microservices architecture, consult *Microservices* by Fowler and Lewis.

Contrast with traditional application architectures

Let's contrast this with traditional application architectures, such as what was discussed in the *Java EE Architect's Handbook*. Traditional application architecture is like a layer cake, as illustrated in Figure 1.2. Typically it has layers to provide a user interface, data access, and business logic; and depending on its feature set, it might have other additional layers. Another way to express this idea is that coupling is high.

Figure 1.2: Traditional Web Application Architecture



These layers are meant to be deployed and tested as a cohesive unit (typically as a *war* or *ear* file). While it may be possible to use portions of the code base outside of the application, that's more the exception than the rule.

The application scales as a unit. It's not unusual for applications to have features with different audiences. For example, it's not uncommon to have administrative features that only a few people use vs. more mainstream features that may be used by hundreds or thousands of users. Traditional architectures must scale the administrative function along with other more popular features as they are in the same deployable. For those not familiar with the term, a *deployable* is a runnable version of an application (e.g., a war or ear file).

As a corollary to this idea, defects tend to spill over into unrelated features. For example, if one feature has a defect that monopolizes your database connection pool and makes the container inoperative, all other unrelated features in that application are also inoperative, even though they are unrelated from a business

point of view.

Typically, the entire codebase shares database resources. This means that changes to that database potentially impact the entire application code base. Yes, we try to insulate the application from these changes by separating out a data access layer, as discussed in the *Java EE Architect's Handbook*, but that tactic isn't always 100% effective. It's not uncommon for database changes to require Entity class changes that potentially have effects on other layers.

What is a Monolith?

Typically, there are limits to how large and complex a code base can be before it starts to become unmaintainable. Applications that approach that point are called *monoliths*.

Monoliths typically have long quality assurance / testing cycles for deployment. With monoliths, the feature set is so large that executing a full regression test takes users a large amount of time. Yes, automated testing sequences shorten up the process for development staff, but user QA testing often has a large manual component.

With every change, monoliths often experience issues with unintended consequences. That is, you fix one issue and accidentally create new bugs as a result of that change. In this sense, fixing bugs in a monolith is like playing 'whack-a-mole. ' Yes, your automated unit tests and integration tests help to mitigate this issue, but that test harness is not 100% effective.

Monoliths are typically married to their technical stack. That is, it's much more difficult to upgrade or swap out a dependency in a monolith. For example, if the monolith uses the Hibernate ORM product for data access, it would be difficult to swap in a replacement ORM product (such as iBatis). Not only would partial use of iBatis make code for the application internally inconsistent; the testing effort would be fairly large for something that really doesn't directly add business value.

Burn-in time for new developers is much longer on monoliths as there is so much to learn. The code base is usually very large and interconnected. The sheer size of it introduces a learning curve for new developers, regardless of the technical libraries and frameworks involved.

As a corollary to this point, developers cannot be transferred easily from one application to another. That learning curve is specific to each monolith. This facet of monoliths creates management constraints to allocating resources and shifting priorities to meet any changes in business priorities.

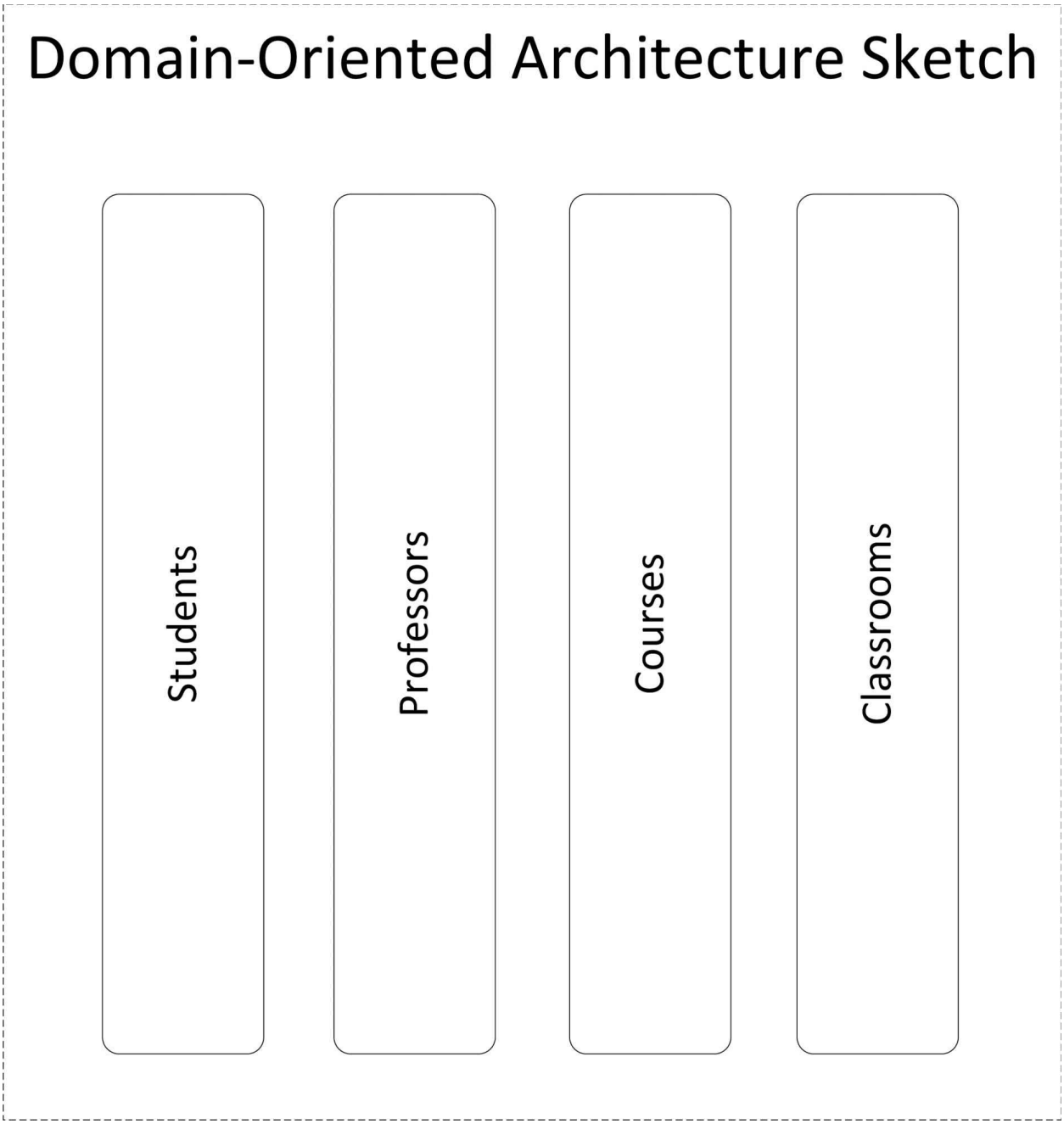
Diagnosing memory and performance issues is often more difficult on monoliths as the haystack (size of the code base) is typically much larger. Finding memory leaks can be one of the most frustrating and time-consuming activities developers go through. Furthermore, the tooling often doesn't directly reveal the code causing the leak, so manual analysis is involved. With microservices architecture, at least the size of the code base that could be the culprit is much less, leading to quicker diagnosis for memory leaks and a quicker resolution.

QA test cycles for microservices, because they have a single functional purpose, are much shorter and more manageable. Microservices are typically focused enough that components of their technical stack can easily be upgraded or swapped out without affecting consumers of the service. As the code base is typically much smaller, it is often easier to diagnose memory issues and performance issues localized to one service.

Microservice Library Organization

Microservice libraries are typically organized by business competency. That is, they organize by topics important to the business. Domain Driven Design (DDD) calls these topics **domains**. Domains are an analytical construct that essentially represents an independent model. DDD further breaks down domains into **sub-domains,** if needed, and then **bounded contexts**. Sub-domains are just a more detailed and specific domain. Bounded contexts represent physical constructs that make up a domain or sub-domain. You can think of a microservice as an example of a bounded context. For example, a domain breakdown for a college or university might look like Figure 1.3.

Figure 1.3: Domain Example



The other theoretical construct applicable to microservices architecture is Conway's Law. Conway's Law (1968) basically states that corporations build applications that follow their internal communication structure. Microservices architecture is no exception. To paraphrase this concept, microservices follow the org chart.

Microservices function by breaking large business problems into many smaller business problems. It's easier to get an understanding of what a microservices

architecture looks like by comparing it to traditional web application architecture, which you already know.

The Benefits of Using Microservices Architecture

Microservices reduce/eliminate technical stack lock-in. How many of us have seen it take an act of God to do a JDK version upgrade? As in a traditional world, it takes coordination from system administrators, and often other development teams, and it's difficult. In a microservices world, it's easy to have different services at different versions of the JDK; the service can get upgraded *in isolation*.

The technical lock-in point goes further than just version upgrades. In this world, services can choose different database platforms as they effectively 'own' them. One might use a nosql database like Cassandra, while another might stick to a relational db.

That said, organizations do get economies of scale with their available database choices. Databases need to be backed up and tuned, and sometimes recoveries need to be performed. This takes technical expertise that most organizations centralize for economic reasons. Just because one of the barriers to making different choices has been removed, doesn't mean that you should go crazy and have 50 different database platforms. From a management perspective, it's important to place boundaries or guidelines on technology choices so that what's created can be supported.

Microservices architecture speeds delivery to market. That is, overall development production rates will increase using a microservices architecture. For example, at last report, Amazon's deployment rate has been reported at 23,000 per day. Netflix' deployment rate has been reported at 5,500 per day. To be fair, most organizations implementing microservices on a large scale also implement continuous delivery and DevOps. It's reasonable to expect that DevOps and continuous delivery implementations at Amazon and Netflix also contributed to their high deployment rates.

The additional productivity depends on reduction in communication overhead for developers. That is, developers are able to be more 'heads down' in development mode and don't need to coordinate their effort as much with others.

In order to achieve that reduction in communication overhead, you must get

good at contract management. That is, you need to be good at explicitly detailing the contract that will be implemented and the behavior that those service operations will implement. Developers shouldn't have to communicate with other teams regarding the interface; they should be able to be heads down.

Each team is able to work independently without forced interaction from other teams. That is, they have their own development/deployment schedule.

As an architect, most of your time will be spent in contract management; that is, detailing the contract specification in enough detail that developers can implement. The questions you get back from developers are feedback to the architect. If the number and scope of questions are minimal, the contract was adequately specified. If developers are confused, the contract specification needs work.

You must be good at contract management. Contract management gives your development team independence and frees them from communication overhead. That contract specification can also be re-used as documentation of the service for consumers, which alleviates communication overhead for both producers and consumers. It's reduction in communication overhead that provides the development velocity increase.

Developers should only be assigned to one and only one service at a time. This reduces context switching time for developers and increases velocity.

Microservices provide resource staffing flexibility. Microservices development teams are small and focused. Developers *only* work on one service at a time. That service is supposed to have a firm contract that those developers are coding to, preferably with containerized deliverables (i.e., Docker). The point is that they're mainly doing heads-down development. They shouldn't be burdened (or have to be burdened) with communication overhead except with other developers working on that service.

They are working as an independent team, completely asynchronously from all other service teams. Combined with the fact that they have low communication overhead and defined deliverables, they can be anyplace in the world in any time zone. It doesn't matter whether they are in-sourced in some corporate office nearby or somewhere off-shore.

Given that the work of the team is confined to the focused requirements for the

service, worries about code quality, etc. are at least contained to the small service source code base. In other words, there's a limit to how much code that team can mess up and thus create maintenance issues.

> ***Note: Development for microservices can more easily be outsourced!***

This also means that you're better insulated from rogue developers. We've all managed developers who produce unmaintainable code; but with microservices, rogue developers are confined to one small and focused code base, so there's a limit to how much damage they can do. Furthermore, if that code base is small, focused, and well-defined, it's more replaceable if need be. You're not letting rogue developers loose in a 500K line monolithic application to cause havoc on a wider scale.

Microservices provide better availability and resiliency (if implemented well). Microservice dependencies force developers to build in fault tolerance and make failure a first class citizen in design and development. Faults and outages are, by definition, localized to identifiable components. We're going to learn that the ability to horizontally scale individual services allows you to decrease complete outage time.

The Disadvantages of Using Microservices Architecture

Microservices can be harder to manage if not designed well. We're going to see that enterprise architecture and contract management is critical. Without that, the benefits of adopting microservices will be eroded significantly. We're going to see that adopting a DevOps culture is a prerequisite for implementing microservices on a large scale. Often, organizations that adopt microservices also have adopted continuous delivery. Bottom line: implementing microservices is a paradigm shift for management as well as developers and architects.

Microservice implementations are more complex. Yes, each individual microservice is easier for developers to work with as it is small and focused. However, in a microservices world, there are many more moving parts. This makes delivering business functionality, which may involve multiple microservices, much more difficult to design and manage. Moreover, that complexity makes defects more difficult to diagnose and performance problems more difficult to investigate.

Microservice implementations will cause complete chaos if contracts aren't designed well. I'll present some advice on designing microservice contracts later in the book. With badly designed contracts, developers spend much more time consuming services and with support activities. This time erodes your development velocity and negatively impacts the benefits of adopting microservices to begin with. From a management perspective, consider contract management specialized labor. Some architects are better at it than others.

It's worth noting that moving to a microservice architecture requires that your organization have DevOps capabilities among other things. I provide an overview of these prerequisites in Chapter Five in the section entitled "Prerequisites for Microservice Architecture Adoption."

Microservices Architecture Compared to SOA

Many ask: "Isn't microservices architecture simply a re-branded SOA?" The answer is '"yes" for the most part. Keep in mind that SOA suffers from the same definitional malady that microservices do, in that SOA doesn't have a concrete definition. SOA means different things to different people.

They are similar in many ways except for the complexity of queues/pipes. SOA often has very complicated plumbing for moving data around in queues. Often Mule or Camel flows are used to transform, copy, or re-order data in transit and to accommodate for any formatting differences as that data is moved from service to service.

In a microservices world, process services (which we'll learn are complex services that rely on other services to function) effectively coordinate delivering complex business functionality. The role of orchestration is delegated to process services and not Mule or Camel flows.

Microservices architecture uses dumb pipes and smart endpoints. Microservices use queues too, but without any transformations you might find in a Service Oriented Architecture (SOA) world. Generally, there are no complex flows (e.g. Mule or Camel) with formatting, filtering, or sequencing changes to message data in being transmitted from service to service. Think of the electrical supply of your stereo system as an example of such a pipe. Electricity is either on or off, with nothing in between. A 'pipe' doesn't get much simpler than that, with absolutely no intelligence in the pipe itself.

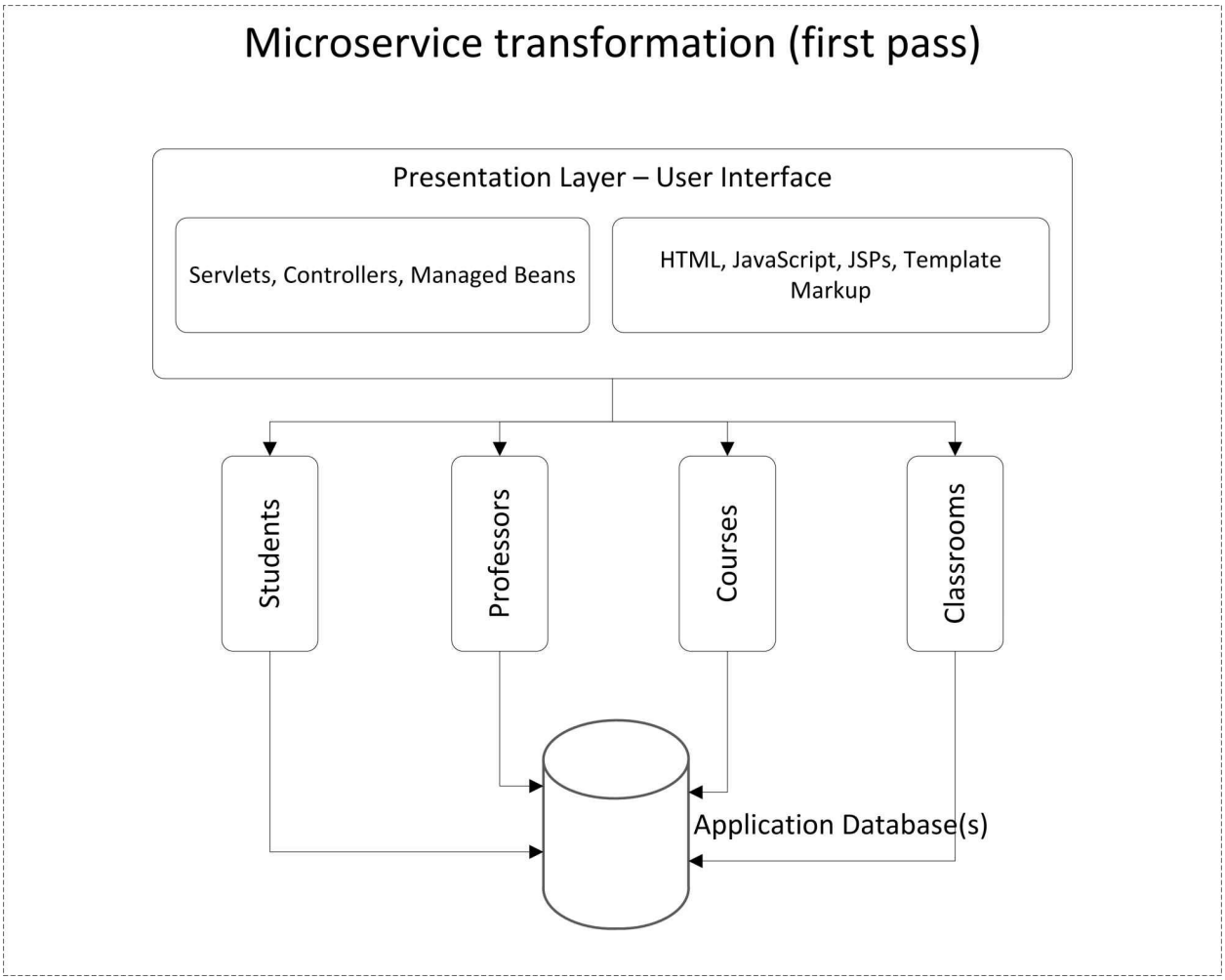
Refactoring into a Microservices Architecture

Typically, applications don't start out with a microservices architecture. In most cases, they are *refactored* into a microservices architecture. Having read numerous case studies, I haven't yet seen an example of a microservices application that didn't start as a traditional web application that got too large to effectively manage and maintain. That isn't to say it's not possible, but it's definitely not common.

Exempt the user interface from following a microservices architecture. There are significant benefits to cohesive user interfaces; they provide consistency to the user experience. Web presentation frameworks are notorious for not playing well with other frameworks. Take, for example, JSF and Spring-MVC. Both are very viable frameworks that will work for most user requirement sets, but if you try to use them together in the same user interface, you'll quickly find that you're swimming against the stream. Bottom line: consider user interfaces exempt from a microservices architecture. I'll get into more about their role and how to incorporate them later.

As it's harder to refactor *databases* (more on that later), let's leave the database alone for the first stages of microservice refactoring as well. What gets separated into individual microservices is business logic. An illustration of this can be found in Figure 1.4. Note that I'm using a simple example where identified domains are bounded contexts. That can happen, but it's more likely that the domain Students would be broken down into multiple bounded contexts and multiple microservices.

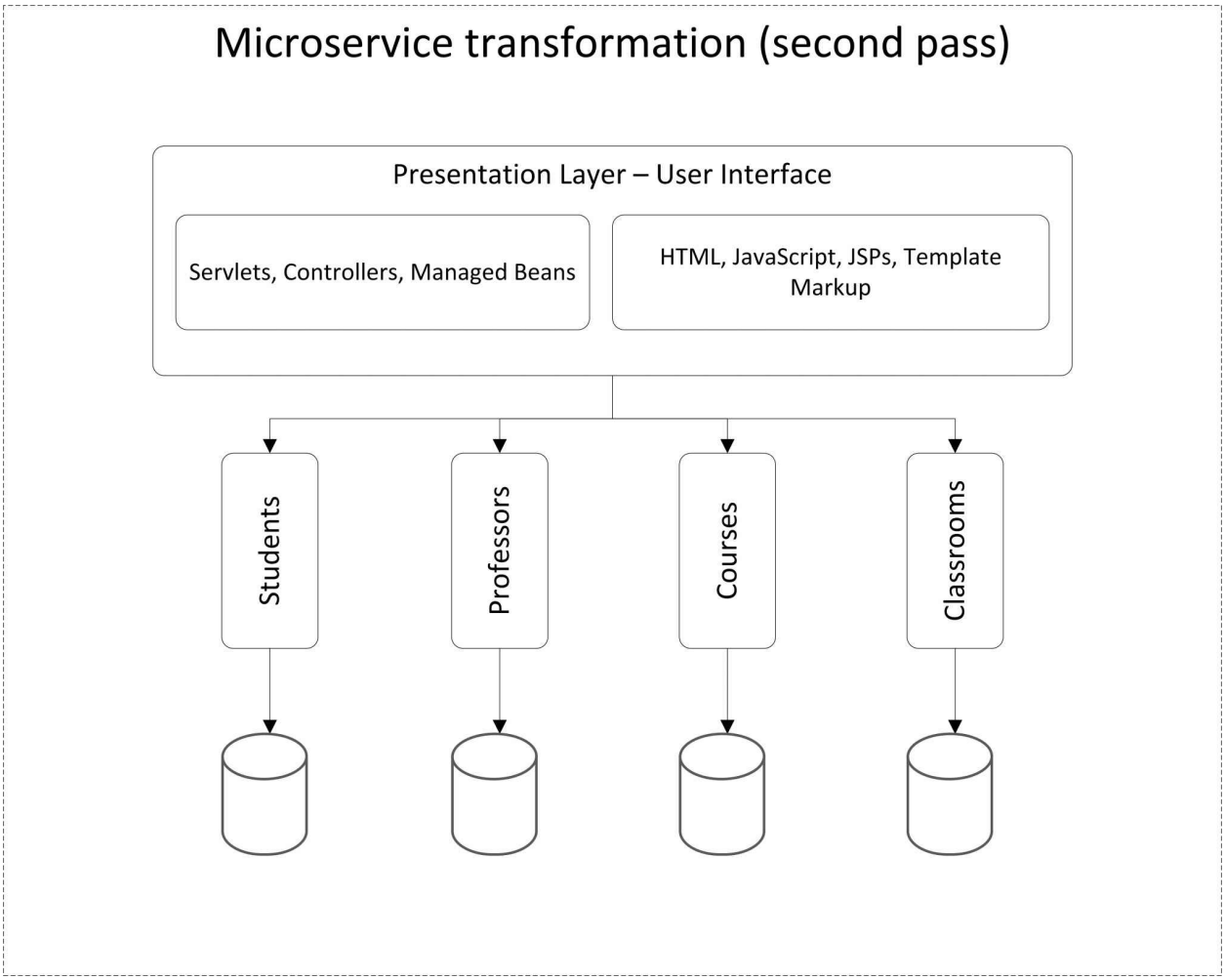
Figure 1.4: First Stage Refactoring Example



We can't leave the database out of the refactoring altogether. Having multiple services use the same database/schema is in fact 'coupling,' which a microservices architecture tries to avoid. It's not coupling in the sense we usually think of it (that is, code coupling). However, any data structure change after a first-stage refactoring will affect multiple services. For this reason, microservices architecture really incents you to separate databases.

Each microservice 'owns' its own database. Look at it in the same way that you look at instance-level fields in a Java class. An illustration of our higher education example with each service having its own database can be found in Figure 1.5

Figure 1.5: Microservice Example



This gives us loose coupling, but also a couple of problems. What do we do with common business data? All organizations have data that are common throughout the enterprise; that is, data that multiple services will need.

Consider a higher-ed example. Universities have ‘campuses.’ Indiana University, which I attended, has campuses in Bloomington, South Bend, Indianapolis, and many other locations. Multiple services with a microservices architecture will need this information. If each service has its own database, that means they are not using a central data store somewhere. The only two choices are service call or copy. That is, define a service that ‘owns’ campus information and make all other microservices call to get that information. The other option is to arrange to ‘copy’ that data to all services that need it.

Personally, I prefer using the service method. (Don’t worry; I’ll present ways to mitigate the performance impact for the additional service calls.) I prefer not to copy data from one service to another because it usually creates a maintenance

headache; there is additional coding and testing in performing the copy, and that copy can have issues. All too often, defects arise because the copies of the same data aren't in sync. I elect to avoid the problem altogether by having only one source for each item of information and making extra service calls to get it when needed.

If you opt for a copy strategy, I'd recommend publish/subscribe. That is, the service that 'owns' the business data publishes the fact that a piece of information changes on a persistent queue. Any service interested in those changes subscribes to that queue and records the necessary events in its own database.

In corporate America, we've really been into recording a fact once and only once. Consequently, it's not uncommon to find sites that have multiple applications, sometimes dozens, maintaining the same database data. For those organizations, refactoring to the point where each microservice has its own database is a tall order. There is a compromise available.

If you can't give each microservice its own database, at least refactor to the point that one and only one service updates, inserts, deletes each data item. That reduces the haystack you need to look through to solve data corruption bugs.

You do not need separate physical database servers/licenses for each microservice. The goal is to remove coupling at the database level. Using separate database schemas is enough to achieve this level of decoupling. Of course, it's important not to create any referential integrity rules (e.g., foreign key constraints) between tables used by two different microservices as this is unwanted coupling.

Chapter 2: Design Considerations for Microservices Architecture

This chapter details important design considerations for microservices architecture. As with any technology paradigm, it's possible to write overly complicated and difficult-to-support software using microservices architecture. This chapter will help you avoid that.

Most people implementing microservices do so using REST-style web services. Hence, all examples I present will assume a RESTful web service interface. It should be noted that REST isn't a requirement for implementing microservices architecture; any standardized interface will do. Most use REST for microservices architecture because its reliance on resources (a.k.a. nouns) naturally lends itself to microservice boundaries.

Service Boundaries

The most important design consideration for a microservice is defining its mission. That is, defining the single business purpose it has to fulfill. There are a couple of ways to approach this topic. One is to identify service boundaries the same way you identify REST service *resources*. Another is to identify service boundaries from a domain-driven design perspective and refine your domain model into sub-domains specific enough to fit when using them to define microservices. Between the two, I usually think in terms of REST resources. It's a natural fit anyway as most users tend to implement microservices as RESTful web services.

Defining a Microservice

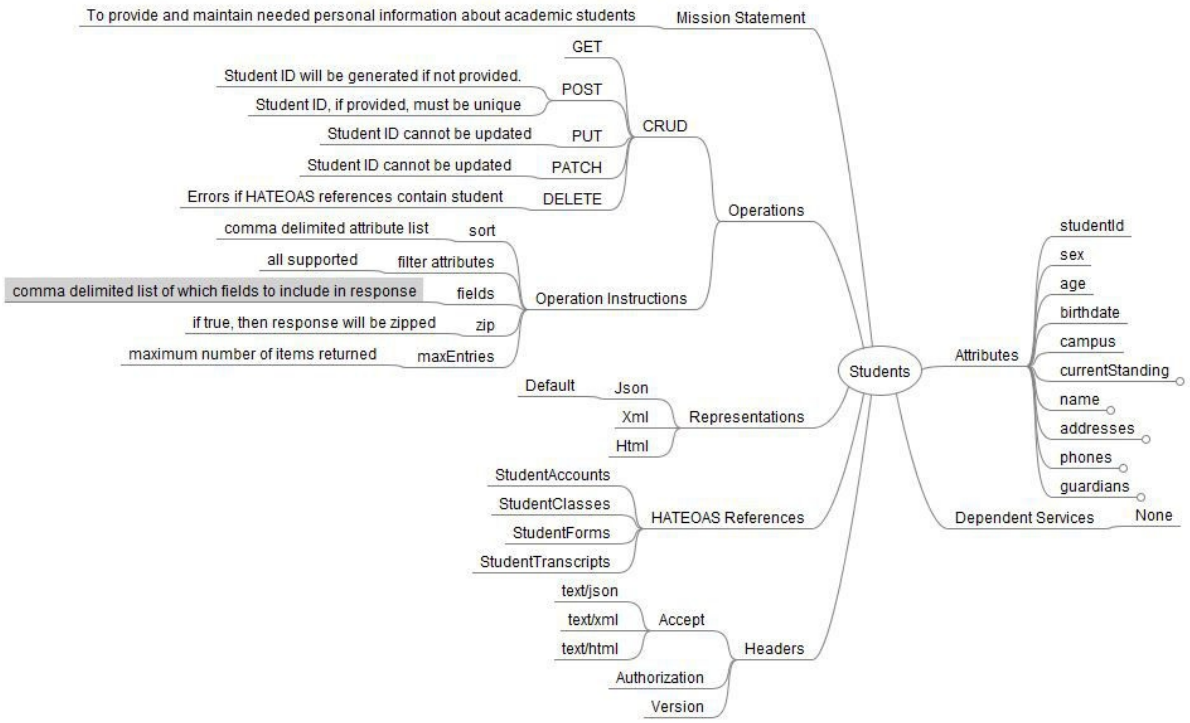
A microservice definition contains the following:

- Meaningful name
- Business function of the service written in business terms
- RESTful web service resource or resource hierarchy supporting the business function
- Operation details such as
    - request parameters
    - input post data format
    - output data format

- Attributes retained about the resource hierarchy
- Scope details
    - Explicit detail as to what business functions are and are not managed by this service.

Let’s work through an example. Take the business concept of “Student” for a college or university. Those institutions commonly have a need to record specifics about students (i.e., student name, assigned student identifier, birth date, sex, addresses, email address, phone numbers, etc.). There are more complicated business processes involving students (e.g., registering them for classes, maintaining their transcript, processing their payments, etc.), but for this example, let’s confine ourselves to merely maintaining student personal information. This example service is visually depicted in Figure 2.1.

Figure 2.1: Core Service Definition Example



The service name, Students, is in the center. To the left, there is a one-sentence description of the service in business terms. All operations are detailed in terms of the HTTP methods that support them (e.g., PUT, POST, GET, etc.). All supported qualifications are listed on GET requests, such as sort and maxEntries. All supported representation formats are listed, as well as supported HATEOAS

(short for Hypertext as the Engine of Application State) reference link. To the right, service attributes about students are listed, as well as any supporting services. An explanation of the different HTTP methods and their implications are presented in table 2.1.

Table 2.1: HTTP Method Usage

HTTP Method	Usage Notes
GET	Used for information *retrieval* only. It is assumed that GET operations do not change data in any way.
POST	Used to *create* a resource only. It is assumed that the resource doesn't already exist and is being newly created.
PUT	Used to *update* a resource. It is assumed that ***all*** information on a resource is being updated.
PATCH	Used for *partial* resource updates (e.g., a selected number of resource attributes).
DELETE	Used to *delete* a resource.

By itself, the diagram in Figure 2.1 might not be complete enough for a team to develop from it. However, at the planning stage, it certainly suffices and definitely provides developers with a starting point from which they can ask intelligent questions. As an aside, I used the open source product Freemind to produce the diagram in Figure 2.1. I usually use Freemind for the initial planning of RESTful web services like this.

Of course, there may be multiple operations to support cross-cutting concerns, such as health checks, performance metrics, runtime diagnostics, etc. Those operations are important, but they exist for *support* purposes and don't have anything to do with the service's reason for existing. I divide the world up into two categories of services: core services and process services.

Core Services

Core services store business facts about the enterprise. They are responsible for maintaining business data for a specific area of the enterprise. Core services provide mainly CRUD-like features and usually don't rely on other services. No matter how we modularize software, at some point, something needs to record business facts in some type of database. Core services fill this role in a microservices architecture. Figure 2.1 is an example of such a core service.

Core services are usually named after nouns, as that's what most business facts are. The same mental discipline you use to identify RESTful web service *resources* can be used to identify core *services*. You can usually express a core service 'mission' as being the 'service of record for a ___'. Fill in the blank on the noun. The format of the mission statement can usually be expressed in the following format:

The <service title> is the system of record for a <business term>.

As an example, consider the Student service defined in Figure 2.1:

The Student service is the system of record for personal information about students, such as their name, address, age, emergency contact and other information.

Let's delve a little deeper into the example illustrated in Figure 2.1. You might question how I decided to break out StudentAccounts, StudentClasses, StudentForms, and StudentTranscripts (which are all HATEOAS reference links) into separate services and why I didn't include them in the Students service directly. The answer is that there is business logic in each of these service references that is specific to that portion of the business. For example, regarding StudentClasses, registering a student for a class is more than just a CRUD entry in a database somewhere. Does the student meet the course prerequisites? Does this class conflict with other classes that the student has chosen? Both of these are business process questions that are best separated out and lumped together with unrelated aspects of student personal information. Whether or not a student is allowed to take a specific class has nothing to do with what their address is or who they list as their guardians.

Let's contrast this with student addresses or phone numbers, which are maintained by the Students service. In this fictional university, there is no business process surrounding recording a student address or phone number. A student provides notification of the change and we record it. No complicated business checks. This is merely a CRUD-like feature. Consequently, the motivation for breaking this out into a separate service is almost non-existent.

Any attempt to delete a student who may have been registered for classes or may have incurred charges will result in an error. With a traditional web application, we might be able to rely on relational database referential integrity rules. However, in a microservices architecture, that information is in separate data

stores and there are no referential integrity rules preventing data from other services being essentially stranded. Consequently, this needs to be enforced at an application level. Another possibility would be to cascade the delete (issue a corresponding delete with dependent services), which is inherently dangerous.

Let's note some more characteristics from the Students service in Figure 2.1.

Adhere to the strict definitions for HTTP methods. There's a brief summary of available methods here. GET requests retrieve information; GET requests never change any business data (logging the GET request and response is a non-functional, cross-cutting support concern and doesn't count as changing data). This abbreviates communication you must provide to service developers who implement the contract.

Adhere to the strict definition for HTTP method error codes. Those error codes are standardized. There is a brief summary of those error codes here. Like following HTTP method definitions, this abbreviates communication you must provide to service developers who implement the contract.

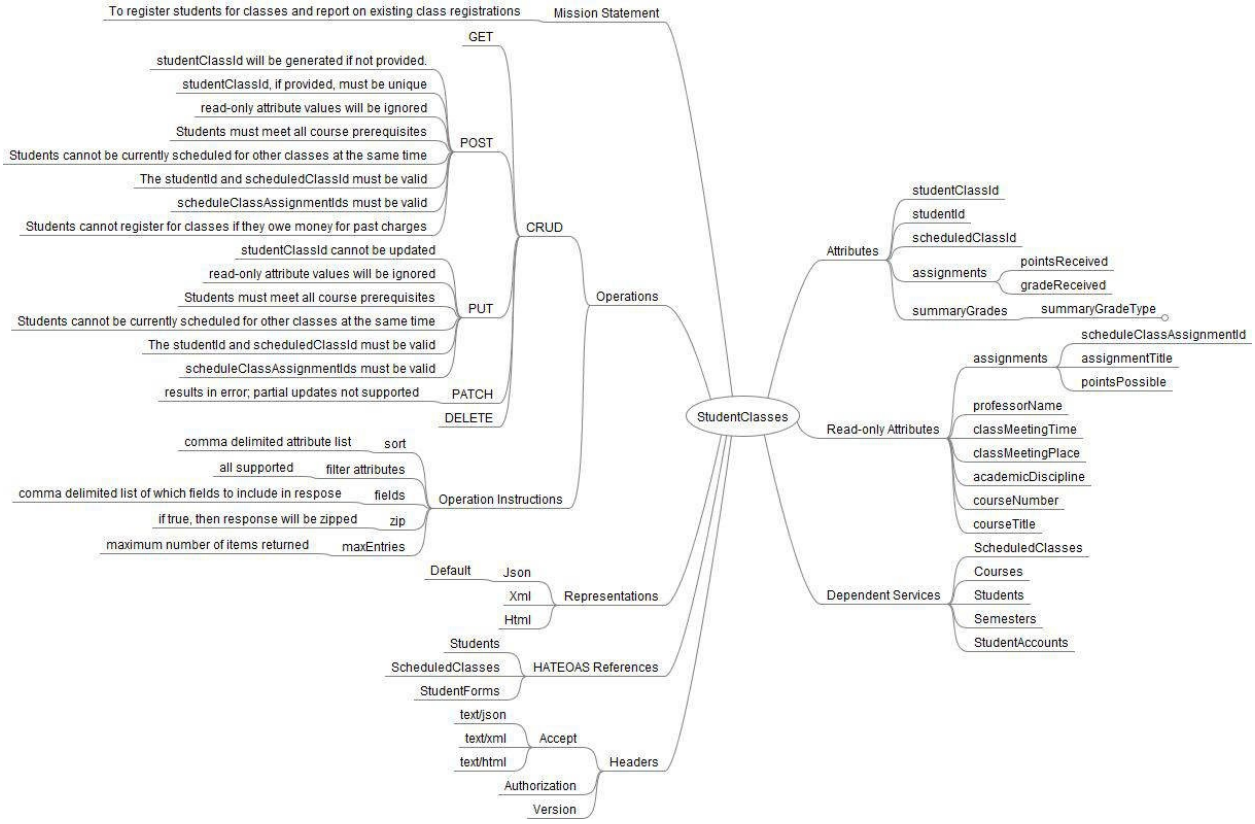
Standardize your operation instructions across all microservices. With the Students service, I've defined several commonly defined operation instructions: sort, fields, maxEntries, etc. It's common to provide a feature that allows the requester to receive requested data in an order they specify. Don't allow some microservices to use the instruction sort and others to use order. This non-standardization creates confusion for consumers and impedes the goal of increasing developer velocity.

Process Services

Process services manage complex business tasks. They usually represent a business-specific action or procedure. Normally, process services rely on other services for much of the data needed to process. Sometimes, there is persisted data needed by process services, but not always.

You can usually express a process service 'mission' as being the 'service responsible for processing _____.' As an example, the StudentClasses service is responsible for registering students for classes and reporting on existing student class registrations. Like the core service example, a similar illustration for the StudentClasses service contract can be found in Figure 2.2.

Figure 2.2: Process Service Definition Example



This definition has a similar format to the core service example in Figure 2.1. Let's look at some of the differences. For starters, the StudentClasses service enforces business rules and constraints on write operations such as the following:

- Students must meet all course prerequisites
- Students cannot be currently scheduled for other classes at the same time
- Students cannot register for classes if they owe money for past charges.

These constraints are purely business level and need to be enforced at a service level.

In addition, not all attributes presented are updatable. Figure 2.2 distinguishes between 'Attributes' that are maintained by the StudentClasses service and those 'Read-only attributes' that are presented for convenience to consumers, but are really maintained in other services. Should consumers want to change the Professor name of the scheduled class, for example, they should affect that

change via the ScheduledClasses service. Yes, presenting this information for convenience does require a service call, which can have a negative impact on performance. We'll discuss ways to mitigate the performance impact of this later in the book.

Note that the StudentClasses service has dependent services, unlike the Students service in Figure 2.1. If these services are unavailable, the StudentClasses service might not function properly. We'll discuss ways to mitigate the amount of vulnerability dependent services can cause. For now, I'm concentrating on effectively defining service boundaries and contracts.

Accept that microservices architecture increases service traffic over traditional application architectures. Process services rely on other services to complete their work. The StudentClasses service in Figure 2.2 is one of these. Process services may have a database resource, but not always.

Partitioning Is an Art

Deciding on a service boundary or determining microservice mission statements is deceptively difficult. Moreover, there is no objective formula I can give you for determining a services mission statement. If you have services do too much or define their mission too broadly, you'll end up with several baby monoliths instead of a microservice library. If you make service operations too fine-grained, you'll get excessive network chattiness and performance will suffer.

Microservices and Domain Driven Design

Evans' Domain Driven Design (DDD) concept has a part to play in determining service boundaries. DDD divides the world into "Domains" and "Sub-domains." Domains are a sphere of enterprise knowledge. They are always expressed in business terms and are a purely analytical construct. They represent an area of the enterprise that is typically too large to represent as a microservice. Sub-domains are arbitrary divisions of a domain and they too are always expressed in business terms. Sub-domains are also an analytical construct and don't necessarily correspond one-to-one to deployed services. Within sub-domains, bounded contexts are drawn.

A bounded context in DDD is an arbitrary boundary that has a unified domain model. Typically, a bounded context resides in one and only one sub-domain. The bounded context is a physical construct and very often has a service

deployment representation. Relationships between bounded contexts are explicitly defined. Usually, the intention is to reflect this context in code so that the code more closely resembles the way the enterprise actually works. Core microservices manage data for one and only one bounded context. They never span bounded contexts.

Process services manage complex processes. These can involve one or more bounded contexts. In essence, process services can describe the relationships between bounded contexts. Relationships between bounded contexts are processes managed by separate services.

Service Mission Checks

Given the artful nature of partitioning, it's hard to provide an objective algorithm for determining the mission statement for a service. I can, however, provide detailed warning signs that may indicate that a service definition needs refinement.

Boundary Mission Sanity Check

You should be able to express the business function of a service in one sentence in business terms. If you can't, it's a red-flag that the service isn't well defined.

Examples of business functions are:

- The Student service is the system of record for Professor information
- The Registration service registers students for classes
- The Discipline service suspends students
- The Student Payment service records student payments
- The Transcript service produces official transcripts

What's common about all of these examples? They are all written in ***business*** terms. Besides the term inherent in the service titles, there isn't a technical term in the lot.

Let's contrast this with examples of *poorly* defined services. Below are 'anti-pattern' examples that you should ***not*** follow, and I then discuss what's wrong with them.

Anti-Pattern Examples

1. The Customer Relationship Management (CRM) service maintains people who will be marketed to. These people include high school students, students attending other colleges, and parents of these student types.
2. The Donation service updates the donation database with people who have been, or should be, solicited for donations, as well as donations received. This includes alumni, parents of alumni, corporate benefactors and anyone else who might be solicited.

Mission statements should not contain technical terms. Use of the term CRM in the first example refers to a specific class of software product. It is a technical term and shouldn't be part of a microservice mission statement. The second example refers to a 'database,' which really micromanages how information is to be stored. Technical implementation details for microservices should be *hidden*. Storage mechanisms and specific product sets should be private and subject to change without notice to service consumers.

Microservices should not attempt to maintain information for multiple business concepts. The Donation service (second anti-pattern example) does too much inside one service. Microservices are meant do one job and do it well. This service tracks candidates who may be solicited for donations and *also* the amounts given. I would separate the business of tracking solicitation prospects from the business of recording solicitation events and donations received. Services to handle donations and record interactions to elicit donations received may very well use the service that tracks solicitation prospects in some way.

It's worth noting that while microservices have one business *purpose*, they can often be used in multiple business *contexts*. For example, the Donation service we're discussing might be used by a service to identify marketing targets or by a service implementing marketing campaigns of different types. In other words, having a single business purpose doesn't keep the service from having multiple consumers.

I would rewrite the "CRM service" mission statement as follows:

- The Marketing Prospect service is the system of record for information about people who will be marketed to.

In addition to being more than one sentence, the CRM service mission statement

contains the term ‘CRM,’ which is a technical term. CRM applications not only track customers and customer representatives for marketing campaigns, but are also used to manage sales force effectiveness and marketing campaigns of different types. It is not necessary to wrap all CRM functions with technology agnostic microservices. It is not unusual to only use a portion of the features offered by a software product. Only portions of CRM functionality that will need to be utilized by other services and applications need to be created. The mission statement seems concerned with the portion of CRM functionality that manages marketing campaign prospects – a ‘prospect’ being a potential customer or customer representative that will potentially be the subject of some type of marketing campaign or customer interaction.

I would rewrite the ‘Donation service’ mission statement and separate it into multiple services:

- The Donation prospect service is the system of record for people who may potentially be solicited for donations.
- The Solicitation service tracks attempts to solicit donations from donation prospects.
- The Donation Receipt service tracks pledges from donation prospects as well as actual donation receipts.

In addition to being more than one sentence, the reference to ‘the donation database,’ is a technical construct and not a business term. In addition, the title ‘Donation service’ is in danger of doing too much. Microservices do one job and do it well. This service tracks candidates who may be solicited for donations and also the amounts given. I would separate the business of tracking solicitation prospects from the business of recording solicitation events and donations received. Services to handle donations and record interactions to elicit donations received could use the service that tracks solicitation prospects in some way.

Context Independence Check

When other teams show up and want to call your service, are you nervous? Do you start asking questions like:

- What are you calling before us and after us?
- Why are you calling us?
- Exactly what business process are we going to be a part of, and what

do you expect us to do?

These are all red-flags that your service isn't truly context independent. It makes assumptions about the business conditions under which it's being called. A well-defined service should be agnostic as to when/how it's called. If it has its inputs and resources, and the inputs are valid, it does its job; that's it. The reaction you should have when other teams show up and want to call your service should be: "Here's the doc. Please let us know if you have questions. Bye…."

More detailed questions to ask along these lines:

- Does your service have multiple consumers? Could it without ill effects?
- Does your service care if it's executed in batch instead of indirectly initiated by a user web application?
- Do you spend time analyzing the call flow between services? The only developers doing that should be those working on process services, which often consume other services to complete their work.

The answers to these questions may very well indicate that your service is not as context independent as it should be. If you agonize over which service should perform a given activity, maybe that activity needs its own service.

Microservices are not about size

The prefix 'micro' in microservices is unfortunate. It has everybody concerned with the size of the code base rather than how well-defined the service mission is.

Microservices architecture is more about a service having *a single functional purpose*. It just so happens that with a single purpose, the code base is usually smaller. Put another way, a small code base is a byproduct of microservices architecture, not an end-goal in and of itself.

What is a line? Choose any service as an example. Write it in Java, and you'll get a certain number of lines. Write the same service with the same features in C, and you'll likely get a larger number of lines as it's a more verbose platform. Write the same service in assembly (not that you would choose to do that) and you'll get a boat-load more lines. Does the technical platform/language choice make the C or assembly versions any less of a microservice? Definitely not.

Designing to accommodate failure

With traditional architectures, most calls are local. If the application is up to make the call, resources are generally available to service it. With microservices architecture, it's common for services to rely on other services as resources. We need to design for the possibility that a dependent service is down.

Designing for failure happens in layers. The first line of defense against failure is horizontal scaling/clustering. That is, running several copies of your services on different virtual machines, thus increasing the probability that at least one copy of a service is available for consumers. The likelihood that the entire cluster is down is less if there are multiple copies of that service running. We do this with traditional applications as well.

There are legitimate reasons for a cluster to be down. It might be down for maintenance, or a resource it depends on may be down in its entirety. Bottom line: *horizontal scaling isn't enough of a defense against outages in a microservices world.*

Fail sooner than later. Have you ever surfed to an unavailable site? Typically, after 60 or 90 seconds, your browser tells you that the service isn't available. The same thing happens with web service calls, such as those that microservices make.

Why is this more of a problem than any other? In a world where service calls are nested – that is, service call services that call other services – it doesn't take much for a single failure to cascade to other services. Let's explore this a bit.

Suppose you have a microservice that's scaled to support 20 concurrent transactions in terms of memory, database resources, container worker threads, etc. Suppose further, that each service call normally takes less than 200 milliseconds (ms) to complete. Imagine what would happen if each of those calls suddenly had to wait for 30, 60, or 90 seconds to fail. You would see lots of derivative errors: out of memory conditions, not enough worker threads, not enough database connections in the services connection pools. Soon, that service that's dependent on the unavailable service would be non-operational. And the cascading of the failure would continue. Soon it would be harder to figure out what was the root cause, and a large percentage of your service library would be inoperative.

The logical conclusion is that services should fail as quickly as possible if dependent resources aren't there. The failure would be descriptive and obviously not a root cause type of issue. Administrators would be able to ferret out the root cause quickly, and work that could continue without the unavailable service would continue. This is an interesting twist on performance tuning.

Further reading on the concept of failing immediately and explicitly can be found in the article "Fail Fast" by Jim Shore.

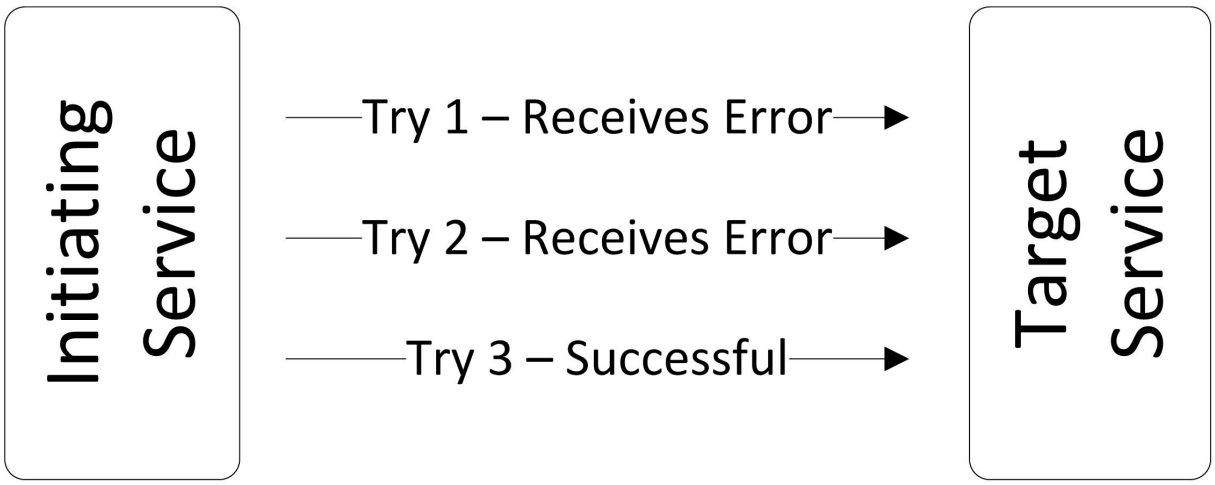
Coding Patterns for Failure

This section describes coding patterns that can be used to fail more quickly and automatically recover with no human intervention.

Retry Pattern

The retry pattern simply retries the requested work until it eventually succeeds. This reminds me of the old saying that insanity is the act of doing the same thing over and over and eventually obtaining a different result. This algorithm is illustrated in Figure 2.3.

Figure 2.3: Retry Pattern



There are usually a couple of configurable options on the retry. An example implementation of the Retry pattern can be found on my GitHub in the Insanity project, which is open source.

- The maximum number of retries before the operation is given up
- The time interval between each retry.

For people who want developer level detail, Insanity's default algorithm provides the interval and max retry configuration options. Listing 2.1 provides a usage example using a 10,000 millisecond retry interval and 2 maximum retries. Insanity requires you to embed the code to be retried in a class implementing the Callable interface from the JDK. Yes, if you want to, Insanity allows you to code your own retry algorithm that accepts different configuration options.

Listing 2.1: Retry algorithm example using Insanity

```
DefaultRetryAlgorithm algorithm = new DefaultRetryAlgorithm(2, 10L);
RetryManager<String> retryManager = new RetryManager<String>(algorithm);
MyCallable callable = new MyCallable();

String callResult = retryManager.invoke(callable);
```

The retry pattern operates on the assumption that an outage is temporary – that somebody will eventually restore service and the requested operation will complete. The retry pattern works best in cases where the work is asynchronous, so use it with care. In a microservice world, we're not supposed to care about execution context.

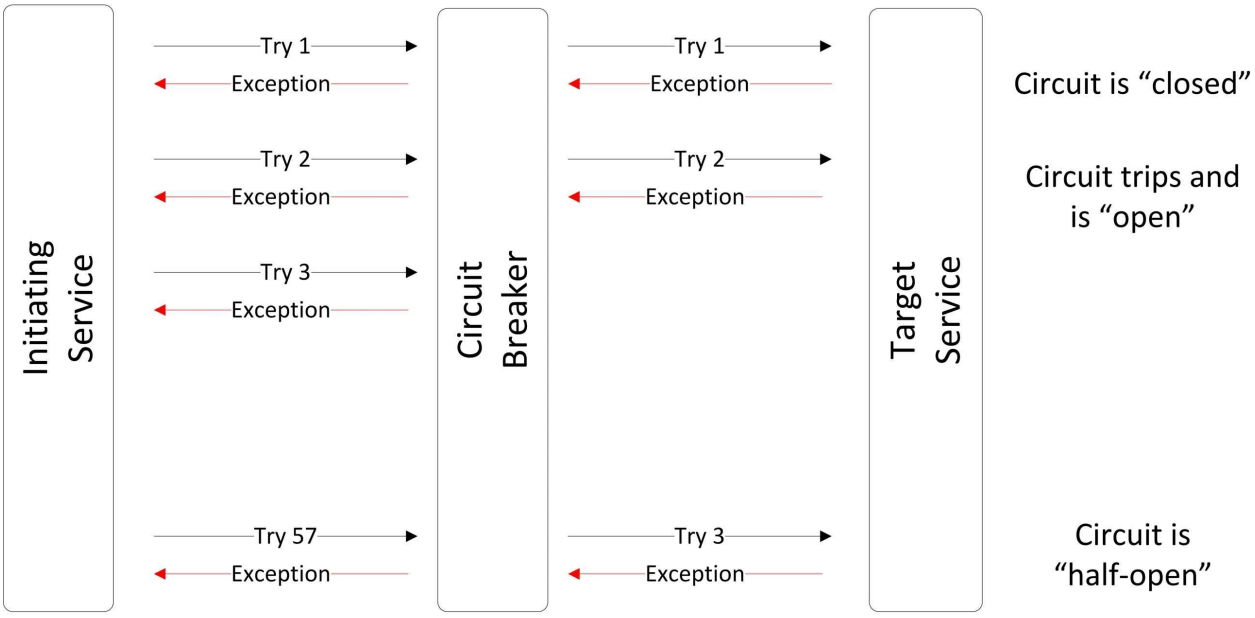
There is product support for this pattern. It is provided by the Apache Httpclient product (and thus CXF also has support because it relies on Httpclient). Spring batch provides a RetryTemplate that implements this pattern.

Circuit Breaker Pattern

This pattern is a software version of the circuit junction box in your house or apartment. Circuits are either closed or open. If they are closed, then everything on the circuit has power and can operate. When something bad happens, the circuit trips and opens. With an open circuit, nothing on that circuit has power or can operate. When you reset the circuit in the junction box, the power is restored to every device on that circuit.

The circuit breaker pattern works the same way. Upon meeting an error threshold, it trips and errors out immediately. Any subsequent calls for a configurable period of time are also errored out immediately. An example implementation of the circuit breaker pattern can be found on my GitHub project CircuitBreaker, which is open source.

Figure 2.4: Circuit Breaker Pattern



After a configurable interval, a service request operation is let through as a 'test.' If the test succeeds, then the circuit is automatically closed and all service calls are allowed through. Put another way, service is restored automatically without human intervention.

The software version of a circuit breaker usually has these configuration options:

- The number of consecutive errors required to trip the circuit and shut down service.
- The time interval between shut down and allowing a 'test' service call to proceed.

For people who want developer level detail, CircuitBreaker's default algorithm provides configuration options for the time interval and the number of consecutive errors. Listing 2.2 provides a usage example with a configuration using a 10,000 millisecond time interval and a two consecutive errors threshold. Insanity requires you to embed the code to be retried in a class implementing the Callable interface from the JDK. Yes, if you want to, CircuitBreaker allows you to code your own algorithm that accepts different configuration options.

Listing 2.2: Circuit Breaker algorithm example using CircuitBreaker

```
DefaultCircuitBreakerAlgorithm algorithm = new DefaultCircuitBreakerAlgorithm(10L, 2L);

Circuit<String> circuit = new Circuit<String>(algorithm);
MyCallable callable = new MyCallable();
```

```
String callResult = circuit.invoke(callable);
```

As the Circuit class retains state (e.g., tracks the number of consecutive errors and knows when a test execution should be allowed through), that variable declaration should actually be at an instance level at least.

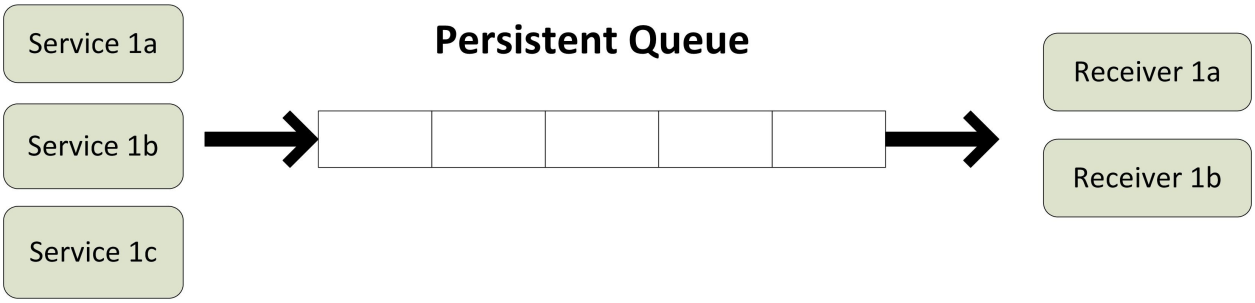
There is product support for this pattern buried in the Hysterix product suite. If you're using the Hysterix suite for other reasons, you should definitely take advantage of their implementation. If you're not, that product has a heavy footprint. In that case, I'd use a narrower implementation, like the one in my sample.

Use of the circuit breaker is best placed in *client* delegates or adapters. That way, when it trips, you avoid the network interaction of the call.

Dispatch via Messaging

We use this for traditional web applications. The idea is that you encode a service call instruction as a message and place it on a persistent queue. That is a queue that records the instruction and guarantees delivery at some point. The service reads its instructions off the queue and executes them. If the target service is down, the work backs up in the queue. When service is restored, the instructions in the queue are processed as they should be.

Figure 2.5: Dispatch Via Messaging Pattern



Note that this is also a performance pattern. While it doesn't really do work faster, it provides users the 'illusion' of a speed improvement by making work asynchronous. That is, users perceive a speed improvement because they aren't waiting for the work to complete.

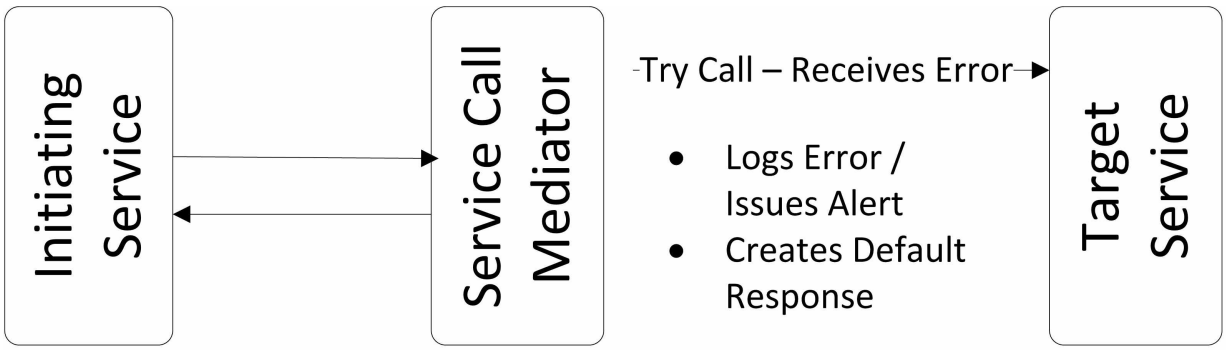
I do recommend AMQP for messaging as opposed to native JMS. JMS is platform-specific (java only), which defeats the technical stack independence benefit for microservices architecture. For tooling support, most consider the

JMS api easy enough to use directly. Spring batch provides a JMSTemplate and RabbitTemplate to assist with message production.

Service Call Mediator

Services should be resilient enough to operate partially when outages occur. The root idea is that providing degraded service is preferred to providing no service. In other words, we're designing to provide partial availability in the event of an outage. One example is a business that sells WiFi service on airplanes. Customers pay a small fee by credit card and receive service. If the credit card payment processing functionality is unavailable, it might make business sense to provide the WiFi service to that customer anyway rather than irritate them by denying service. The mediator pattern is one of the 'gang-of-four' behavior patterns defined in the book Design Patterns. An example implementation of the service call mediator pattern can be found in my GitHub project TaskMediator, which is open source.

Figure 2.6: Service Call Mediator Pattern



Since the initiating service receives a default/normal response, it processes as usual. The amount of code in the initiating service that is needed in order to provide partial functionality is minimized. From the perspective of the initiating service, no error occurred. However, the service call mediator logged the error and initiated any support alerts needed.

For those wanting a more code-level example, Listing 2.3 provides an example of the service call mediator pattern using TaskMediator.

Listing 2.3: Service Call Mediator algorithm example using TaskMediator

```
TaskMediator<String> taskMediator = new TaskMediator<String>(DEFAULT_RESPONSE);
MyCallable callable = new MyCallable();

String callResult = taskMediator.invoke(callable);
```

Note that it is possible to combine the service mediator pattern with the circuit breaker pattern. In this way, you get the benefits of failing earlier and with fewer resources, and also the benefit of providing partial functionality. An example of how to accomplish this using CircuitBreaker can be found in Listing 2.4.

Listing 2.4: Example combining Service Call Mediator with CircuitBreaker

```
DefaultCircuitBreakerAlgorithm algorithm = new DefaultCircuitBreakerAlgorithm(10L, 2L);

Circuit<String> circuit = new Circuit<String>(algorithm);
MyCallable myCallable = new MyCallable();
CallableCircuit<String> circuitCallable = new CallableCircuit<String>(circuit,
myCallable);

TaskMediator<String> taskMediator = new TaskMediator<String>(DEFAULT_RESPONSE);
String callResult = taskMediator.invoke(circuitCallable);
```

Designing for Performance

In a microservices architecture, there's a lot more network traffic. This increased

network traffic can cause architects to be concerned regarding performance. After all, service calls are dreadfully expensive when compared to a local method call. Also, since architects may 'assume' that microservices architecture uses services that are fine-grained, this can also contribute to their concern. But this assumption is false. All that you learned with SOA about services being coarse-grained still applies; but even with coarse-grained service calls, network traffic increases with microservices architecture. Fortunately there are design tactics we can use to mitigate the performance impact of increased service call traffic.

Coding Patterns for Performance

There are several coding patterns that help to mitigate the performance impact of service call traffic. I'll take you through them.

One of these patterns, dispatch via messaging, is also a failure design pattern discussed in a previous section and won't be described again here. It's also a performance pattern in that by taking work offline, it provides users with a perceived performance gain as the portion of the work they see is shorter.

Back-ends to Front-ends (API Gateway)

The back-ends to front-ends pattern improves performance by ensuring that microservice traffic is on your local network and not on the open internet. This pattern uses gateway services that consolidate microservice calls for web applications / user interface products. The idea is that the browser portion of the application will make one or two calls to its application service provider. That provider will make the numerous microservice calls (either directly or indirectly) that are needed to accomplish the work requested by a user in the web application.

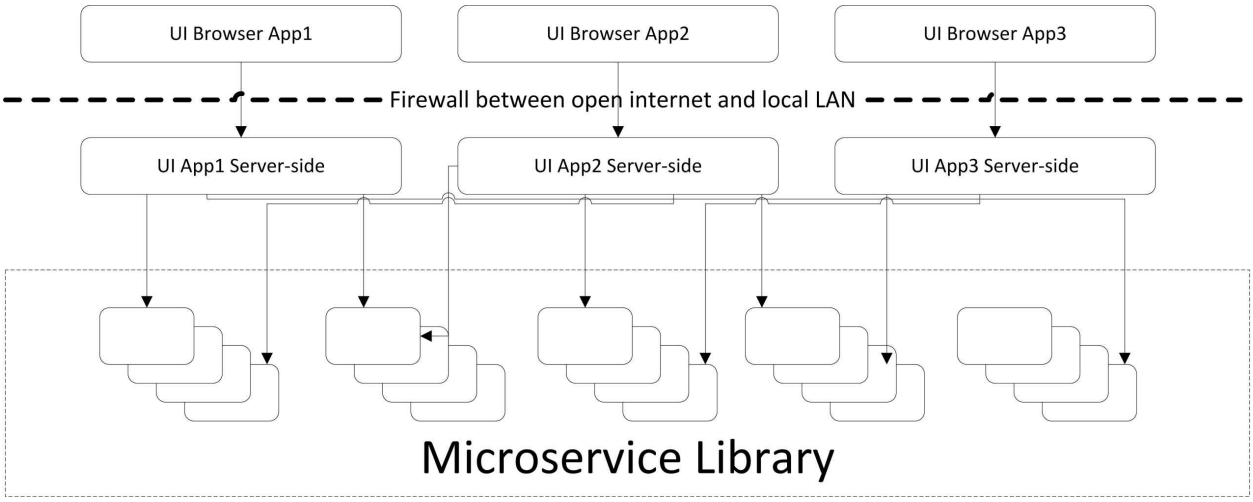
I'm using the term 'application service provider' to describe that gateway service that consolidates user interface calls and responses.

Another name for this pattern is 'API Gateway.' However, I do not use that term as there is generally confusion between the API Gateway *pattern* and API Gateway *products*, such as WSO2. API gateway products are about routing and governance and not about any kind of call consolidation activity. I'm talking exclusively about the *pattern*.

Application service providers consolidate service calls. That is, a call to them

might mean calls to multiple services in your microservice library. Any data from those microservice calls are consolidated and returned to the caller. All the chatty microservice traffic is on your local network, and that can be properly tuned by your network engineers. That local network can also be properly scaled to provide service for the extra network traffic that microservices architecture requires.

Figure 2.7: Back-ends to Front-ends Pattern



Yes, this does mean that the consolidating call will likely return a larger amount of data. Let's work that through a bit. The cost of the network portion of a service call can be thought of in two parts: the cost of establishing a connection with the service host, and then the data transfer of the request/response. By consolidating service calls, back-ends for front-ends drastically reduces the connection establishment portion of the traffic.

Transfer cost might also be reduced in that the gateway service sorts through all microservice output and streamlines the result to include only what that specific web application needs to function. For example, it is not unusual for core services to provide information about a business topic that the caller doesn't need. That 'extra' information is weeded out by the application gateway manager and not sent to the browser. In this way, the data transfer portion of the cost is optimized.

Do not expose microservice traffic directly to the browser. This is a bad idea and anti-pattern. Use the back-ends to front-ends pattern instead of directly exposing your microservice library to the browser. You don't control the network between the browser and your service library for external web sites. Users will get varying performance experiences depending upon how they are connected to the internet. You'll not be in any position to really help them. For this reason and this alone, all browser service calls should be consolidated.

Expiring Cache

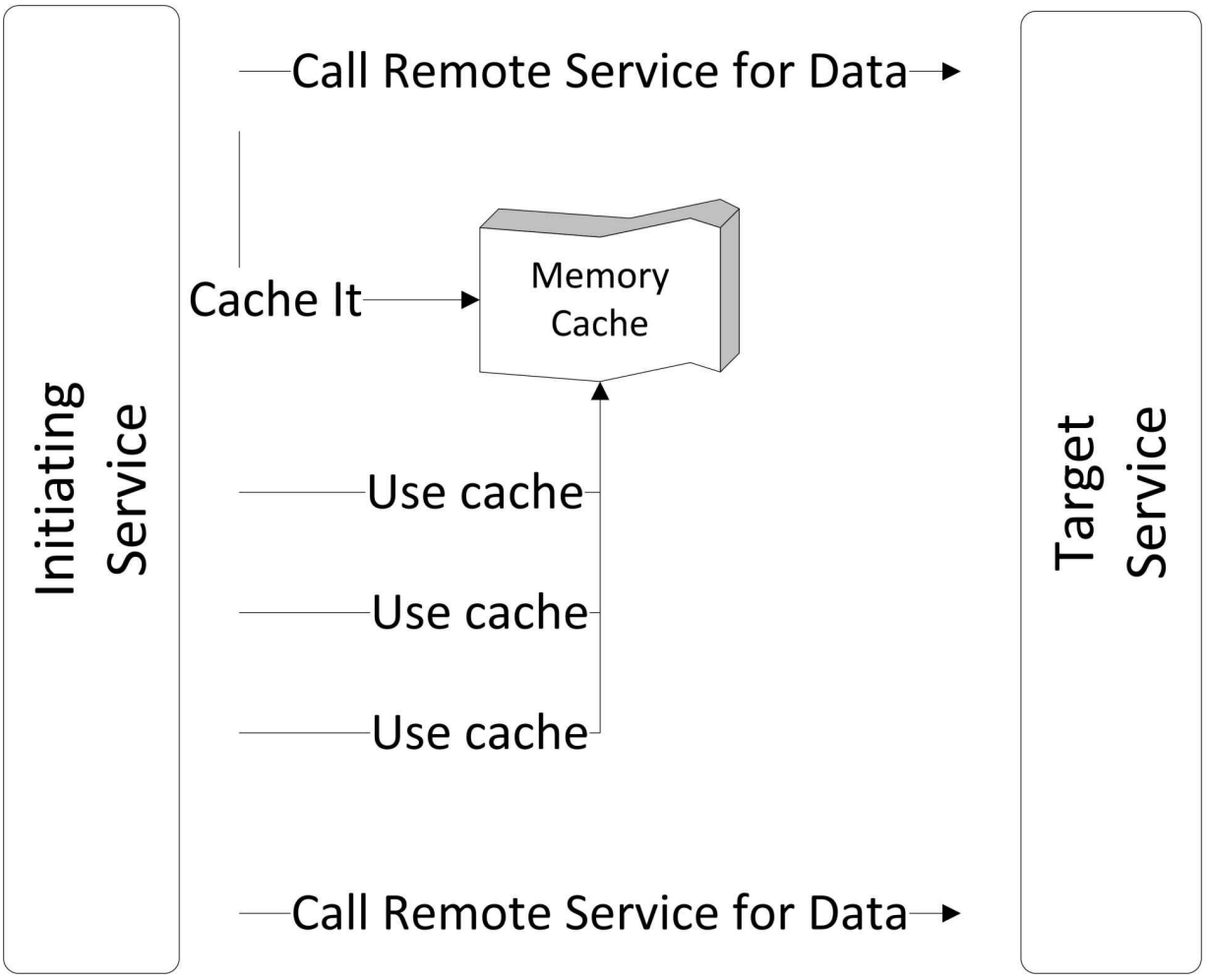
The expiring cache pattern, sometimes known as 'Cache Aside,' exploits the fact that some data in every organization is static and doesn't require a fresh read every time it's needed. The concept is that static data is read and then cached for

a configurable period of time. During that period of time, the cached version of that data is used and not freshly read from a database.

Expiring cache pattern trades memory for speed. The pattern consumes memory for the cache, but average retrieval time is extremely quick.

Let’s take an example. Indiana University, my alma mater, has several campuses: Bloomington, Indianapolis, South Bend, and others. The specifics of each of those campuses are relatively static; that is, very seldom does the university establish a new campus or decommission an old one. There is no reason to re-read campus information every time it’s needed and therefore it is a logical candidate for the expiring cache pattern; especially in a microservices architecture where that ‘read’ would likely be a service call to a core service.

Figure 2.8: Expiring Cache Pattern



Typically, expiring caches have the configurable expiration timeframe. It might be safe to cache some information for long periods of time, even a day, while other types of information require refreshes in shorter intervals.

Utilize the expiring cache service in consuming services and don't centralize that logic in the target service for several reasons. Performance enhancement comes not only from avoiding the database read, but also from the service call to a core service. Secondly, the consumer knows more about context and understands how current the cached information needs to be. It might be that some services need data more current than others and will want a shorter expiration time.

I use the Google Guava product for expiring cache. A reference on how Guava caching works can be found here. That said, EHCache and other products provide this functionality as well. At this point, there's really no need to roll your own. Implementing an expiring cache using Guava has two parts: extending

CacheLoader and defining the cache itself.

The CacheLoader will look up a value if given a key. Most classes that extend CacheLoader only need to implement the load(key) method that will perform a lookup and return the value for a given key. The second part is defining the cache itself. An example can be found in listing 2.5. While, by default, cached values are expired after 100 milliseconds, you can choose any time interval you deem appropriate.

Listing 2.5: Guava Cache Definition Example

```
LoadingCache<Integer, String> sampleCache = CacheBuilder.newBuilder()
    .expireAfterWrite(100, TimeUnit.MILLISECONDS)
    .build(loader);
```

It is possible that different nodes of a cluster might have different answers because they'll be on different caching schedules. This is usually okay, because to use microservices architecture, you need to accept the idea of 'eventual consistency' anyway. We'll get more into this point later. Should you need to force a reload of a Guava cache, executing invalidateAll() on the cache will effectively force a reload.

Designing for Integrity

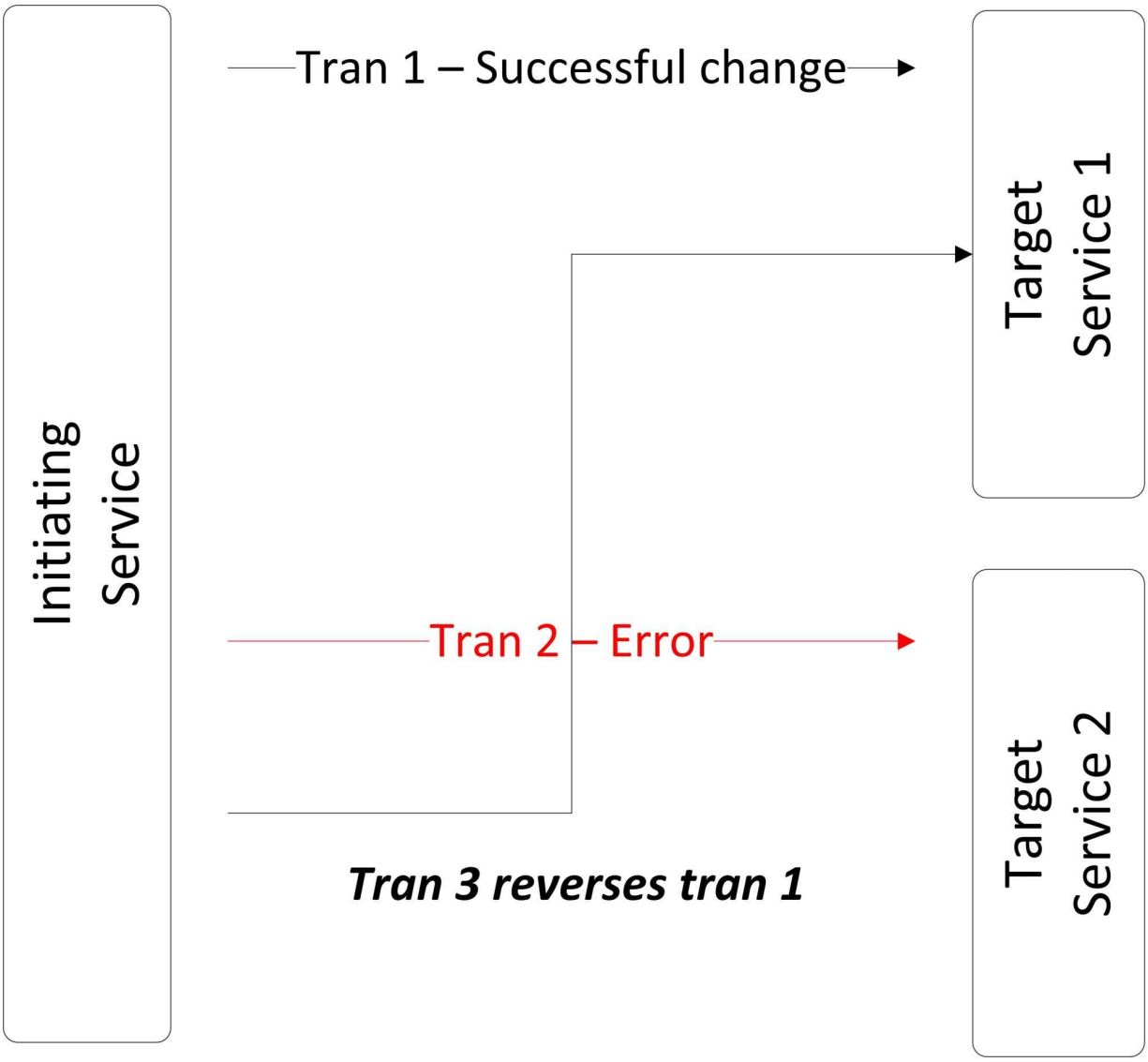
In traditional web applications, we have the safety of the database transaction. If an error occurs, it's common to execute a database rollback, which leaves the database in a consistent state. In a microservice world where a user action may utilize multiple services, this safety net doesn't exist because those transactions are taking place in several databases initiated by different services. Yes, core services typically have a database transaction safety net, as their focus is largely CRUD related, but process services, which utilize other services, do not.

Custom Rollback

Basically, this pattern consists of issuing reversing transactions in your exception handling logic. This pattern is also known by the title of 'Compensating Transaction.' Essentially, when an error occurs, reversing transactions are issued for any changes that might have succeeded. For example: say a process service call results in three service calls to other microservices. Say the first transaction succeeds, but the second fails. In the exception handling logic for the process service, it issues a reversing service call for the transaction that succeeded,

leaving data in a consistent state.

Figure 2.9: Custom Rollback Pattern



This strategy isn't perfect. It's possible that the reversing transaction will error out too or that the process service will tank before the reversing transaction can be issued. Either way, stored data will be left in an inconsistent state, thus requiring some type of fix. Usually, this fix is a manual one.

Writing custom rollback transactions takes labor, so you don't want to use this pattern on everything. Basically, you have two choices: to code custom rollbacks to reduce the need for manual intervention, or to manually make data consistent when errors happen. Usually, organizations start with manual intervention and then use code custom rollbacks to address areas with the highest frequency of failure.

This pattern is incredibly hard to generalize and provide product support for. The specific logic needed for a reversing transaction needs to be custom coded.

Managing Service Versions

Managing versions of services is hard enough with traditional web applications. In a microservice world, the problem exponentiates as the number of services exponentiates. There are two popular versioning strategies in use. The first literally places a version number in the URL for the service. The second allows the client to request a service version in the header. Should they not request a version, they default to the most current version. I'll go through both versioning methods. But first, let's discuss different types of changes and your civic duty as a producer or consumer of a service.

Breaking vs. non-breaking changes

A breaking change is one that causes existing service calls to error out. Breaking changes force consumers to change code to avoid errors. Yes, you can mitigate the effect of this by adopting a deprecation strategy (which I'll discuss later), but still, you force change. By definition, that forced change is going to get prioritized above most other tasks for the consuming application team and will likely usurp other feature enhancements that team is supposed to work on. From the consuming team's perspective, that change comes out of the blue and isn't as easily manageable as scheduling feature enhancements.

Not all changes are breaking changes. If a service call requesting information about a topic suddenly returns a few additional items of information, this should be a non-breaking change. It's non-breaking because consumers shouldn't pay any attention to the additional information provided since they don't currently have the logic to process it. Should they make a decision to process those additional information items, they can enhance their service on their own schedule. In fact, they might not need the additional items of information to enhance their service at all.

As a service publisher, your civic duty is to introduce a breaking change only as a last resort. Only introduce breaking changes if absolutely nothing else is possible. This can happen despite the best of intentions. It could be that a dependent service introduced a breaking change, thus forcing a reaction. It could be that a feature that was supported can no longer be supported because the required functionality from dependent services is no longer there. It could be that

the organization has decided to deprecate a business process, and as a result, you need to introduce a breaking change to accommodate that business decision. Note that this probably isn't a complete list of what could force a breaking change.

As a consumer, your civic duty is to ignore any piece of information returned in a service call that you don't need. The reason is that this makes your consuming service more resilient and less impacted by changes in services you call. You won't be affected by additions to your result set or the removal of attributes you don't really need. This decreases the chance that changes to the services you call will affect the consuming service.

As a service provider, you need to establish a deprecation strategy. That is, you can't just introduce a breaking change and make consumers scramble to adjust. For a period of time, you need to support the old **and** the new service. You also need a communication mechanism that allows you to announce version changes and also to announce the point where support for specific versions will no longer be provided.

Versioning is *only* needed for breaking changes. Non-breaking changes can be published under the same version; providers shouldn't have to maintain separate versions if changes are non-breaking. This incents consumers to ignore returned data they don't use as it minimizes the impact of change they experience as the services they call are enhanced.

That leaves several questions. If clients, in effect, manage their own migration schedule from *version a* to *version b* of a service, there needs to be a way for clients to specify which version of the service they would like. On the provider side, architects need a sensible strategy for supporting both versions of a service for a period of time. I say 'both' because ideally, at any one time, you should have one version in deprecation and one current version.

Versioning by URL

One popular way to version is to version by URL. That is, encode the version number somewhere in the URL and make consumers change their request URL when they move from version *a* to version *b*. Examples of URL versioning can be found in Listing 2.6.

Listing 2.6: URL Versioning Examples

```
https://university.edu/services/Students-1.0/12345
https://university.edu/services/Courses-2.1/PY101
```

Versioning by URL is initially appealing as it is easily supported by web server software. You can have multiple versions of a service deployed and web server software can route clients to the correct version. For instance, web server software can easily distinguish between Students-1.0 and Students-1.1 and route clients automatically to the version they request. When a version is deprecated, clients attempting to use it can either receive a 426 (HTTP status code for 'upgrade required') or be automatically forwarded to a new version.

This becomes problematic if you provide references via HATEOAS because it requires services providing HATEOAS link content to be aware of version changes in those referred services. For example, if the Students-1.0 services provide a HATEOAS reference to StudentClasses-1.1, that service needs to change when new versions of the StudentClasses service are introduced. This makes providing HATEOAS references more complicated and problematic.

Additionally, you have no control over when clients who have been provided with those urls will actually use them. Encoding versions in the urls provided can be problematic in that the versions encoded in them could actually be deprecated between the time they are provided and when they are used. This is another reason I prefer versioning by header (described below).

Versioning by header

One way to version is to allow the client to request a specific version in the header of the service request. There are no standard header names that address this. Some have suggested using the ACCEPT header to specify both the version and the data format expected. To me, using the ACCEPT header is a leakage of concerns issue. ACCEPT is really only about the format expected. Version specification implies possible behavior differences in operations besides GET. Listings 2.7a and 2.7b are examples of versioning by header.

Listing 2.7a: Header Versioning Example

```
URL:  https://university.edu/services/Students/12345
ACCEPT-VERSION: 1.0
ACCEPT: application/json
```

Listing 2.7b: Header Versioning Example using ACCEPT header for

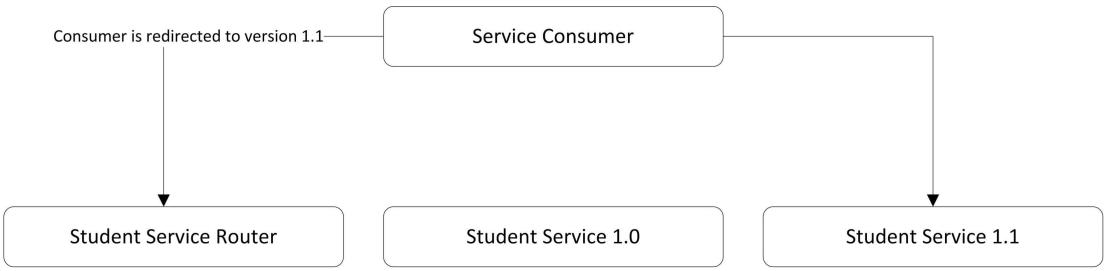
versioning

```
URL:   https://university.edu/services/Students/12345
ACCEPT: application/vnd.students.v1.0+json
```

I prefer to introduce a custom header, as with listing 2.7a, since none of the standard header names really apply. This also leaves ACCEPT free to specify format in case your service supports multiple presentations. For example, a client might specify an ACCEPT-VERSION=1.0 request header in a call to the Students service. If that version is honored, then it is provided by the service publisher. If that version has been deprecated, then the oldest supported version attempts to satisfy the request. Maybe it works, maybe not. After all, if a consumer isn't going to listen to your deprecation announcements, they get what they get. If a consumer doesn't specify a version at all, they get the most current version available. Similarly, the return data format can default. If you separate out the version specification into a separate header, then it's easier for consumers to let the version default and specify the format or vice versa.

Versioning by header does require some type of router that interrogates the header value and redirects requests to the appropriate version. I'm not aware of product support for header versioning, but it would be easy enough to genericize it and provide it.

Figure 2.10: Header Versioning Example



Always provide a response header that indicates the version of the service being provided (e.g., VERSION=1.1). As there isn't a standard response header for versioning, one would need to be introduced. This information could potentially be useful to consumers diagnosing support issues.

Supporting Multiple Version Deployments

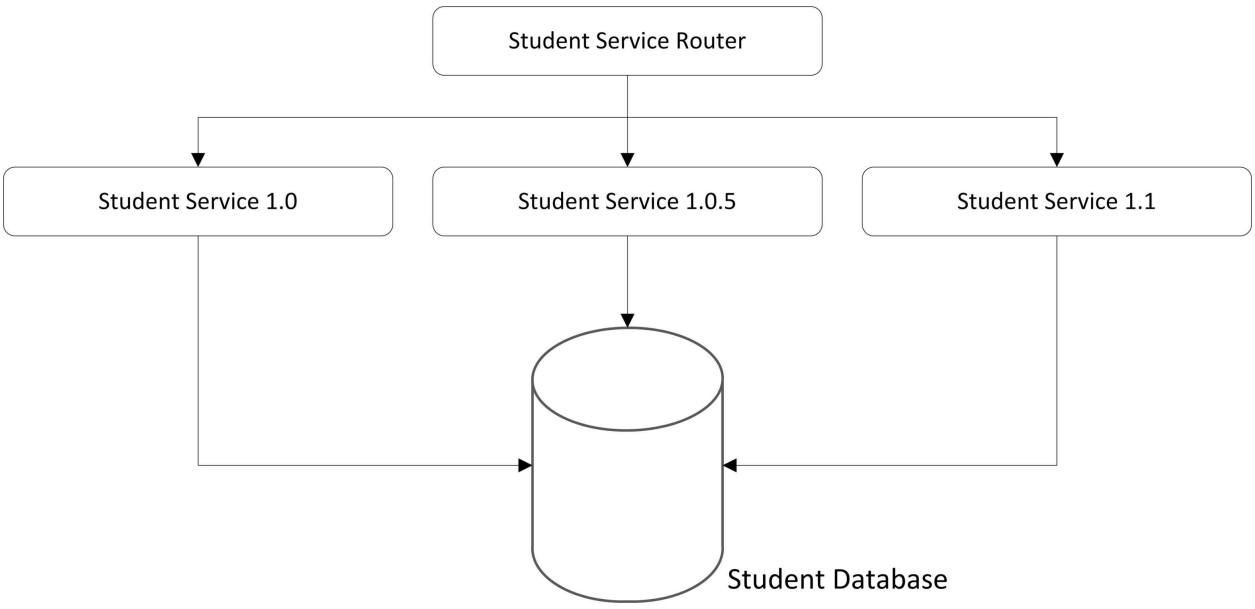
Deploy all versions as separate applications. Do not attempt to commingle multiple versions in the same service. There's too great a risk that unintended changes inadvertently get made to previous versions. In addition, it makes the service code much more complex.

Horizontally scale versions separately. It is conceivable that consumer load isn't identical across versions. For example, it's possible that load for version 1.0 of your service is less than version 1.1 or 1.2. Consequently, version clusters might be configured differently. If your clustering is supported by DNS round-robin, use subdomains to distinguish different versions (e.g., students_1_1.university.edu).

All versions of the service should use the same database. It's possible that feature enhancements introduced in newer versions require a database refactoring that can make support of older versions of a service problematic. The book *Refactoring Databases* provides numerous patterns that can assist with resolving these problems.

An illustration of these concepts can be found in Figure 2.11.

Figure 2.11: Version Deployment Example



Adapter Services for Third Party Products

With traditional web application deployments, it is not uncommon to use adapters for interfacing third party products (SalesForce, Microsoft Dynamics, Great Plains, etc.). These adapters translate between your service's internal domain model and that used by the third party product.

In a microservices world, we do the same thing. The only difference is that the adapter is an entire service, not just a class. If we left the adapter as a set of classes, changes to that third party service, such as product switches or upgrades, would affect multiple services. That's not what's wanted in this world.

It is imperative that the adapter service be product-generic. Any part of the product domain model used by your service contract represents an unwanted coupling. The benefit to the adapter is that product switches or upgrades should **only** affect the adapter; nothing else.

Log all requests and responses to third party products/services. This is needed for support reasons. Should you need to call a provider for support, you don't want to confuse the issue by talking about your code internally. Figure out what you need to correct with your service request text and work backwards to what you need to change in your code.

Design adapters to handle failure for third party products or services. You don't directly control the availability or performance of third party products or services. Because of this, it's not uncommon for adapter services to implement

circuit breakers or call mediators to more gracefully operate during outages of third parties.

Common Enterprise Code for Microservices

A question I'm frequently asked is if there can be common code between services. The answer is a qualified 'yes.' The bottom line is that you version common code and treat it the same way as you treat third-party products like Hibernate or Apache Commons Lang. Individual services decide when/if to upgrade.

Changes or enhancements to common code cannot force deployments of multiple services. This would create a high degree of coupling between services and thus defeat the benefits of using microservices architecture. Consequently, it's rare that business logic is in common code. We don't want changes to the business to affect multiple services.

Common code should be versioned just like any third-party library you use. Individual services utilizing the common code base should be completely free to decide when they upgrade to newer versions. Version upgrades for common code products should never be forced upon them.Z

Chapter 3: Cross-cutting Concerns for Microservices

This chapter deals with numerous cross-cutting non-functional concerns that all microservices must address, regardless of which technical stack they employ. These cross-cutting concerns include being able to effectively monitor services, detect failure, and diagnose and fix reported defects. Satisfying non-functional requirements make it possible for operations staff to support microservices architecture once it is deployed to production. Some of these requirements exist for traditional web applications, but become more imperative with microservices as the number of deployments increases exponentially.

There are non-functional requirements that all microservices should meet, regardless of which technical platform or language they employ. Typically, these non-functional requirements are also cross-cutting concerns. That is, they are not specific to the business functionality being provided; they exist to make deployed services more supportable. As architects, we should establish re-usable and easily implementable ways of satisfying non-functional requirements. This chapter will detail what those requirements are and provide options for easily satisfying them. Microservices should have the following to make them easier to support:

- health checks (typically by URL)
- performance metrics and alerts
- resource consumption alerts
- logging and error handling
- deployment formats (e.g., Docker image).

All of these non-functional requirements should be a part of the application architecture. Individual service teams should not have to ‘develop’ tooling to satisfy any of these requirements. Install and configure, yes. Develop, no.

One of the advantages of writing microservices using Java, or at least one of the interpreted languages running on the Java Virtual Machine, is the ease with which functionality to address non-functional requirements needed by all microservices can be written and merely consumed by those services. This architecture is all about allowing developers to concentrate on the functionality they are providing instead of getting bogged down by technology and infrastructure support concerns.

As application architects, ensuring that development teams have ready-made solutions to satisfy non-functional requirements is where we add value. This streamlines the productivity of development teams, which in turn increases the development velocity of those teams. It also streamlines the support effort and cost for that development once it is released into production. This "streamlining" is imperative.

Microservices architecture greatly increases the number of deployments, sometimes exponentially. Consequently, support for these deployments ***must*** be much less manual and time-consuming than it typically is today. Put another way, the business processes we use to provide support for less than a dozen deployables won't work for the hundreds of deployables that microservices architecture grows into.

Ways to address all common non-functional requirements should be already developed and present in the application architecture. This architecture should be utilized by all microservices written in Java. It should be versioned so that individual microservices can, in most cases, upgrade on their own schedule. This concept was discussed in much more detail in the section on common code.

Consider making the maintenance of this application architecture the goal of a separate team. It helps to rotate team members into other teams creating and supporting microservices. This utilizes the 'eat your own dog food' type mentality. They'll get feedback as to how well they are providing effective application architecture support.

Health Checks are imperative. Our methodology and options for doing health checks become much more important as the number of deployments increase, as they do with microservices architecture. It becomes important to standardize these health checks to the point where they are merely included and configured with microservices; development time spent on these should be nominal. More detailed treatment of this topic is coming up later in the chapter.

Logging Consolidation is imperative. Logging consolidation tools such as Splunk and LogStash have become common for good reason. With a larger number of deployments, logging consolidation isn't an option; it's a necessity when implementing microservices. With large numbers of deployments, reviewing or searching logs individually to diagnose defects can take large amounts of time. Log consolidation toolsets greatly reduce that time and make

support teams more productive. As implementing microservices exponentially increases the number of deployables, it also greatly increases the number of logs and the amount of log data.

Develop and document non-functional requirements for microservices and enforce them enterprise-wide. Regardless of the languages and tooling chosen for a given microservice, the enterprise as a whole needs to be able to utilize the service. That means being able to deploy and monitor its health and resource consumption.

Microservice teams are responsible for ensuring that their services meet all non-functional requirements. Effectively, this means that choosing a technical stack 'new' to the organization is inherently 'taxed' as it also needs to supply the capability for meeting non-functional requirements. For new technical stacks, strategies to address non-functional requirements may not exist within the enterprise and need to be identified and possibly developed. Ideally, the microservice team adopting a technology new to the enterprise will develop this capability in a way that other teams can reuse it for other services. For example, if an enterprise adopts the JRuby framework and decides to write microservices using it, capabilities to incorporate health checks and performance metrics, and to address other non-functional requirements, may need to be identified and established.

Health Checks

A health check is typically a URL that provides some type of indication that the service is responsive and able to fulfill its function. That URL is invoked by some type of monitoring tool (e.g., Nagios) that is capable of alerting an administrator if the health check fails. The simplest alert is that the microservice JVM is unresponsive. The question then becomes: what is your definition of 'responsive'? Often just able to respond with an HTTP 200 isn't sufficient.

In addition to verifying that the service itself is able to respond to a URL without error, I like to check access to external resources and dependencies. For example, I like to check database connection pool health. Yes, databases and other external resources should also be monitored with separate health checks, but checking the health of the connection pool used for database access also verifies that the pool is sized properly for the transaction volume that the service is currently handling. This kind of check can only be performed within the

microservice process itself and cannot effectively be monitored from the outside. That kind of condition can also happen with traditional web applications and isn't really new with microservices.

Health checks, regardless of the technical stack used for the microservices you are writing, should be part of the application architecture provided by the enterprise. There is no reason why each service producer should need to build their own. I usually utilize the Dropwizard metrics library to code health checks.

Those looking for a code-level example of such a health check can find one here. This example checks the connection pool to validate that the number of transactions waiting for a database connection is less than a configurable number (it defaults to three). That is, if too many transactions are waiting for database connections, then the service isn't healthy because consumers are suffering performance issues. In this case, the connection pool might need to be resized for the load that the service is handling. In addition, I validate a connection from the pool to make sure not only that the database is available, but also that network connectivity between the service and database is available.

Performance Metrics and Alerts

Performance metrics aid support developers in investigating performance problems. I track performance metrics, such as performance for each page in the application, for traditional web applications as well. In receiving vague reports of slow performance, it's useful to know exactly where performance currently is when compared to performance history, if you have it. More importantly, it also speeds up performance investigations.

With microservices architecture, user actions can involve multiple services. Localizing a performance problem (i.e., determining which service(s) are contributing to that performance problem) becomes a time-consuming issue. Having that performance data for all services in hand greatly saves the support developer's time.

Traditional web application performance is easier to measure. Typically, performance metrics are gathered at a URL / web page level. Tools to gather performance metrics for individual pages/URLs for a traditional web application are plentiful. I use Admin4J, but there are other product options.

With RESTful web services, performance measurement is a bit more difficult

because URLs contain identifiers, and behavior differs by operation (GET, POST, DELETE, etc.). To elaborate, measuring by strict URL doesn't make sense for RESTful web services as URLs contain data identifiers. For example, it's not meaningful to measure GET operations for URLs /students/1 and students/2 separately. They are both identical lookups, just on different students. It would be better to have consolidated measurement for the entire mask /students/*. This gets even more difficult if your services have options for returning related data items. For example, if the service supports /students/1/classes, the behavior for a *class* search result would be different than for the *student* search result and it might be useful to differentiate. Admin4J will consolidate performance metrics for RESTful web service calls; at the time of this writing I haven't found any other product that will.

At the very least, measure the average call time and the total number of calls for each service. With the large number of deployables that most microservices architecture projects have, knowing the number of calls allows you to concentrate effort on tuning the most frequently used services. Having the average call time helps you localize performance issues where user actions can involve multiple services. While I'm used to also having the minimum, maximum, and standard deviations of call times, along with the average and total number, that information is less useful and can be looked at as optional.

Place automatic alerts for service call times beyond a configurable threshold. This allows support developers to start proactively diagnosing and fixing performance problems before end users report them. Determining a sensible threshold for each service will involve some amount of trial and error. Having a threshold that's too low will produce false alerts, or alerts that turn out not to be indications of material performance problems.

Resource Consumption Alerts and Responses

There are some resource consumption issues within a JVM process that are effectively service health issues, should they occur.

Monitor memory usage for deployed services. When you look at the way that the JVM allocates memory, however, it's not the aggregate memory usage at the process level that's most relevant, because that measurement includes native memory allocated by the JVM whether or not it's actually used by Java code running within. What's most relevant is consumed and available heap memory

within the JVM. Available heap memory is available to Java classes and can actually be consumed. I use Admin4J to monitor memory usage and issue alerts for shortages, but there are other tools available.

In diagnosing memory shortages, the first question is whether the memory usage spike detected was caused by a memory leak of some type or not. If the cause was a memory leak, then allocating additional memory will not help you. If the cause was normal usage, it's possible that there's a capacity planning issue, and that either more memory needs to be allocated or additional occurrences of the service need to be defined within the cluster.

It is possible, with the Oracle JVM, to trigger a memory dump when an OutOfMemoryError is generated through the HeapDumpOnOutOfMemoryError startup option. This is rather limiting as this event can only happen once. Should the JVM recover from the memory shortage, additional occurrences will not generate heap dumps.

Monitor concurrent load for deployed services. This information is needed for capacity planning. Increases in concurrent load will increase required memory and often increase average transaction time. In years past, I've monitored concurrent load from within the JVM using Admin4J. These days, with cloud vendors providing scaling capabilities, it might make more sense to monitor load from outside the JVM.

Alerts for resource consumption issues are usually handled via log management software. That is, those 'alerts' are recorded in the log at an ERROR level, and log management software initiates some type of alert (e.g., initiate email or work ticket) to an administrator.

Logging and Error Handling

With microservices architecture, the ability to easily search all logs from all services is essential. In that world, the number of deployables exponentially increases, and it's not practical for support personnel to manually search individual log files from those deployables. With microservices architecture, using some type of log management system transitions from being a 'nice to have' to being an absolute requirement.

Log management software, such as LogStash or Splunk, gathers individual log entries and centrally stores them. Furthermore, these tools provide a central way

for support personnel to search them and configure automatic alerts. It's not uncommon for errors detected by your log management software to either generate some type of support ticket to be investigated or initiate an email to a central support email account.

Bundle needed information with exceptions that you throw. One of the main reasons for logging is to provide information needed to resolve a bug resulting in an exception. This additional logging code adds additional risk since exception code isn't typically tested as thoroughly. Hence, there's usually a larger chance of excepting (e.g., from a NullPointerException) during exception message formatting than with non-exception code.

Apache Commons Lang, starting with V3.0, provides a contexted exception ability that reduces this risk by making it easy and less risky to include additional information with the exception itself (see classes ContextedException and ContexedRuntimeException from Commons Lang). This additional information will be present when this exception is logged. Consider the example in Listing 3.1.

Listing 3.1: Sample Contexted Exception

```
throw new ContextedRuntimeException("Error creating application", exception)
  .addContextValue("application id", applicationId)
  .addContextValue("customer id", customerId);
```

When the exception in Listing 3.1 is logged, the context labels and values will automatically appear in the log, along with the error message and stack trace. Note that application code doesn't increase the risk of a derivative exception by adding conditional logging logic or formatting logic. If it weren't for the contexted exception ability, developers would have to add logging code to supply the information needed to resolve the bug.

Log transaction exceptions only once. Logging exceptions multiple times will greatly inflate your logs and make a specific exception pertaining to a reported defect much harder to find. You can avoid the problem of repeating log messages by logging exceptions *at the entry points only*. For example, it's easy to implement a servlet filter that will log exceptions for any web transactions. This ensures that exceptions are logged only once. This advice assumes that you've bundled any additional information needed to resolve the error with the

exception itself, as described earlier in this chapter. Listing 3.2 illustrates a servlet filter of this type.

Listing 3.2: Logging Servlet Filter example

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws
          IOException, ServletException {
      try {chain.doFilter(request, response);}
      catch (Throwable t) {
          logger.error("Web Transaction Error", t);
          ServletUtils.reThrowServletFilterException(t);
      }
}
```

Source: Class net.admin4j.ui.filters.ErrorLoggingFilter from Admin4J (http://www.admin4J.net)

This technique reduces application code and increases the chance that generated exceptions are logged. Furthermore, if you leverage the Admin4J product, which has an error handling filter, you'll get the request URL and a dump of any request and session attributes at the time of the exception. In other words, you can leverage an open source product to accomplish this, rather than creating your own servlet filter for this purpose.

Incidentally, you'll need to provide similar architectural support for batch job classes, EJB classes, and Message-Driven beans. Leaving exception logging to application code increases the probability that an exception doesn't get properly logged.

Do not commingle transaction logs with normal application logging. Transaction logs (e.g., user Fred changed the address of customer Acme) have a different purpose. Often, transaction logs are needed by normal business processes and really aren't directed at product support. Most logging products will allow you to direct transaction logs to a separate output for this purpose.

Note that log retention and storage concerns are a completely separate topic. Most logging products support storing logs in a variety of formats, including file or database formats. Application code should only be concerned with issuing log messages, not with the specifics of how they are stored or for how long they are retained.

Provide a default logging configuration that can be overridden at deployment. This is purely a convenience for developers that may need to run a build but

might not have taken the trouble to configure their local deployment. Furthermore, it's not possible for developers to anticipate logging needs for specific deployments. Consequently, it's important that administrators be able to easily configure logging at deployment time.

Posting Change Events

Given that each microservice has its own database, there are two options for accommodating common business data that is needed by several services. (I highlighted this issue earlier.) The first option, which I prefer, is not to duplicate that common data but rather to call the service responsible for maintaining that data when it's needed. The second option is to 'copy' common business data to all services that need it. For those that opt to 'copy' common business data between services instead of nesting service calls, copies are generally accomplished using publish/subscribe messaging.

The microservice responsible for maintaining common business data would publish any change (e.g., add, change, or delete) under an established topic in a consistent format. For example, a service responsible for maintaining campus information (e.g., a Campus microservice) at a university might publish the addition of a new campus or a change in status for an existing campus under the topic CampusEvent.

Any interested services would listen on that topic and process any published changes. Typically, services listening for those changes would need to publish a listener to monitor that topic. For example, a Course microservice might listen to the CampusEvent topic and process any changes received. The Course microservice would record any campus changes received in its own database.

Develop a way to refresh all copied data from its source. Copy processes can have defects too. When that happens, a 'copy' of data will become out of sync. In our example, the Course service copy of campus information won't match what the Campus service has if an event is somehow missed or the recording of a change errors out in the Course service. As all code has defects, this issue will happen if you copy data between services. For that reason, support developers need a way to refresh a data copy if it gets corrupted.

Packaging Options for Microservices

In years past, with traditional web application deployments, it was common in

some organizations to deploy multiple web applications per container. Often this was a result of licensing costs for commercial containers which incent reducing the number of containers as much as possible. That combination strategy doesn't happen in a microservice world. Microservices are deployed as independent processes, and that means separate JVMs.

There are multiple products that facilitate deploying java services as a single process. The two most popular products at the time of this writing appear to be Spring Boot and DropWizard. Yes, there are others and they work just fine, but I'm concentrating on the two products that are most popular at the time of this writing.

In addition to those two products, I'll also discuss a virtualization product useful for microservices; it's called Docker. Docker standardizes processes and provides supporting tools to support mass deployments of services. We're going to see that Docker goes a long way toward decoupling developers from operations.

Spring Boot

Spring Boot packages java web applications into a single executable jar file that can be executed via a single command: java –jar myApp.jar

That deployable jar contains an embedded container (either Tomcat or Jetty) and contains every dependency class needed to run your application.

Besides your service, Spring Boot requires you to create an Application class implementing main() so that it can be run from a command prompt. That application class handles most configuration items (connection pools, number of worker threads, etc.). In addition, you can add configuration properties specific to your application. If you do so, make sure you steer clear of name conflicts for property names that Spring Boot already honors (list here). To make sure, I've adopted the habit of using my application name as the first node in the property (e.g., moneta.server.max.threads). An example of application.properties can be found here.

Spring Boot does automatically consume an application.properties file in the class path. These properties are mainly application-specific. This mechanism allows you to configure at least application specifics from outside. Everything else is coded in the application class. Let's consider an example of a spring boot

application class that I've open-sourced and published here. You can find an example servlet definition with listing 3.3 and a filter definition example in listing 3.4.

Listing 3.3: Spring Boot Servlet Configuration example

```
@Bean
public ServletRegistrationBean monetaServlet() {
     ServletRegistrationBean registration =
         new ServletRegistrationBean(new MonetaServlet(),
               "/moneta/topic/*");
     registration.addInitParameter(
          MonetaServlet.CONFIG_IGNORED_CONTEXT_PATH_NODES, "moneta,topic");
      return registration;
}
```

Source: Class org.moneta.config.springboot.MonetaSpringBootApplication from Moneta (https://github.com/Derek-Ashmore/moneta)

Listing 3.4: Spring Boot Filter Configuration example

```
@Bean
public FilterRegistrationBean monetaPerformanceFilter() {
    FilterRegistrationBean registration =
         new FilterRegistrationBean(new MonetaPerformanceFilter(),
            monetaServlet(), monetaTopicListServlet());
     return registration;
}
```

Source: Class org.moneta.config.springboot.MonetaSpringBootApplication from Moneta(https://github.com/Derek-Ashmore/moneta)

As you might expect, this class has a main that loads an application configuration, installs health checks, and activates performance metrics collection and JMX. In addition, this class defines several servlets and servlet filters. With traditional web applications, these are defined in the web.xml configuration file; but Spring Boot configuration is different.

Spring Boot uses Logback logging by default. At this level, a logging configuration is likely for developer convenience. When the service is deployed, a production logging configuration will likely override what's placed in the spring boot deployable. Look at the logging configuration at this level as a 'default,' but nothing more.

If you use a logging abstraction like Slf4j, then providing a default logging configuration is as easy as including a logback.xml file in the Spring Boot jar. Spring Boot will already list Logback as a dependency as it's the default logging API used. As the logging configuration is really a default at this level, I send all output to the console. I don't want to make any file or disk assumptions about where this might be run. Those decisions are logically made at deployment time.

Spring Boot has a way to configure database connection pools using the application.properties file. The reference has a complete list, but I've included a sample in listing 3.5.

Listing 3.5: Spring Boot Connection Pool application.properties example

```
# Connection pool definition
spring.datasource.driver-class-name=org.hsqldb.jdbc.JDBCDriver
spring.datasource.name=testdb
spring.datasource.url=jdbc:hsqldb:mem:my-sample
spring.datasource.username=SA
spring.datasource.password=
spring.datasource.continue-on-error=true
# Pooling properties
spring.datasource.max-active=5
spring.datasource.min-idle=1
spring.datasource.test-on-borrow=true
spring.datasource.test-on-return=false
spring.datasource.test-while-idle=false
spring.datasource.time-between-eviction-runs-millis= 1
spring.datasource.validation-query=SELECT CURRENT_DATE AS today, CURRENT_TIME AS now
        FROM (VALUES(0))
```

In high-volume deployments, it's often necessary to configure lower level container settings, such as minimum, maximum, and idle timeout settings for container worker threads. There are other settings that you need to be concerned with in high-volume deployments, but here I want to illustrate how you might do that using Spring Boot. This can be accomplished by overriding the servletContainer method in the application class. An example of how to do this can be found in my sample and in listing 3.6.

Listing 3.6: Spring Boot Container Configuration example

```
@Bean
public EmbeddedServletContainerFactory servletContainer() { JettyEmbeddedServletContainerFactory
        factory = new JettyEmbeddedServletContainerFactory();
```

```
    factory.addServerCustomizers(new JettyServerCustomizer() {
        public void customize(final Server server) {
          // Tweak the connection pool used by Jetty to handle incoming
          // HTTP connections
          Integer localServerMaxThreads = deriveValue(serverMaxThreads,
              DEFAULT_SERVER_MAX_THREADS);
          Integer localServerMinThreads = deriveValue(serverMinThreads,
              DEFAULT_SERVER_MIN_THREADS);
          Integer localServerIdleTimeout = deriveValue(serverIdleTimeout,
              DEFAULT_SERVER_IDLE_TIMEOUT);

          logger.info("Container Max Threads={}", localServerMaxThreads);
          logger.info("Container Min Threads={}", localServerMinThreads);
          logger.info("Container Idle Timeout={}", localServerIdleTimeout);

          final QueuedThreadPool threadPool = server.getBean(QueuedThreadPool.class);
          threadPool.setMaxThreads(Integer.valueOf(localServerMaxThreads));
          threadPool.setMinThreads(Integer.valueOf(localServerMinThreads));
          threadPool.setIdleTimeout(Integer.valueOf(localServerIdleTimeout));
        }

    });

    return factory;
}
```

Source: Class org.moneta.config.springboot.MonetaSpringBootApplication from Moneta(https://github.com/Derek-Ashmore/moneta)

Spring Boot has an add-on called Actuator. This add-on provides preconfigured health checks, performance metrics, and diagnostics. If you're using Spring Boot, you should use Actuator. Note that if your application has custom health checks, it's relatively easy to write an adapter so that these custom health checks will work with Actuator. All that's required is to implement the HealthIndicator interface, which requires one method. You can find an example here.

Dropwizard

Dropwizard packages Java web applications into a single executable jar file that can be executed via a single command:

java –jar myApp.jar server myAppConfig.yaml

That deployable jar contains an embedded Jetty container and contains every dependency class needed to run your application. Note that Dropwizard also requires additional arguments. The server argument specifies that the application is to be run as an HTTP server. The yaml file input is your configuration. While not strictly required, it's where container configuration items are listed.

Like Spring Boot, Dropwizard requires an application configuration class with a main()that loads an application configuration, installs health checks, and activates performance metrics collection and JMX. You can find an example Dropwizard application class here.

The application class also defines any servlets or filters needed by the application. In this world, servlets and filters are not defined in the web.xml configuration file. An example servlet configuration can be found in listing 3.7. An example filter configuration can be found in listing 3.8.

Listing 3.7: Dropwizard Servlet Configuration example

```
ServletHolder topicHolder = new ServletHolder(Source.EMBEDDED);
topicHolder.setHeldClass(MonetaServlet.class);
topicHolder.setInitOrder(0);
topicHolder.setInitParameter(
        MonetaServlet.CONFIG_IGNORED_CONTEXT_PATH_NODES, "moneta,topic");
environment.getApplicationContext()
.getServletHandler()
.addServletWithMapping(topicHolder, "/moneta/topic/*");

Source: Class org.moneta.config.dropwizard.MonetaDropwizardApplication from
Moneta(https://github.com/Derek-Ashmore/moneta)
```

Listing 3.8: Dropwizard Filter Configuration example

```
FilterHolder perfFilterHolder = new FilterHolder(Holder.Source.EMBEDDED);
perfFilterHolder.setHeldClass(MonetaPerformanceFilter.class);
perfFilterHolder.setInitParameter(
        MonetaPerformanceFilter.PARM_MAX_TRNASACTION_TIME_THRESHOLD_IN_MILLIS,
        "3000");
environment.getApplicationContext()
.addFilter(perfFilterHolder, "/moneta/*", null);

Source: Class org.moneta.config.dropwizard.MonetaDropwizardApplication from
Moneta(https://github.com/Derek-Ashmore/moneta)
```

Like Spring Boot, Dropwizard uses Logback logging by default. At this level, a logging configuration is likely for developer convenience. When the service is deployed, a production logging configuration will likely override what's placed in the spring boot deployable. Look at the logging configuration at this level as a 'default,' nothing more.

Logging configuration for Dropwizard is configured in the yaml file you supply

when the container starts (example here). I prefer to supply a default configuration, including logging, for developer convenience. Dropwizard, by default, accepts only a file reference for configuration with no option to load it from the classpath. Fortunately, that is easy to change. The application configuration is controlled by a class that implements the ConfigurationSourceProvider interface. I've developed an enhancement that will look for the configuration on the classpath as well (here). You must also add the new configuration provider to the application bootstrap in the application class. To accomplish this, you just need to override the application initialization process. An example can be found in listing 3.9.

Listing 3.9: Example for Enhancing Application Configuration

```
@Override
public void initialize(Bootstrap<MonetaDropwizardConfiguration> bootstrap) {
  bootstrap.setConfigurationSourceProvider(new MonetaConfigurationSourceProvider());
  super.initialize(bootstrap);
}
```

Source: Class org.moneta.config.dropwizard.MonetaDropwizardApplication from Moneta(https://github.com/Derek-Ashmore/moneta)

Database connection pooling can be configured with Dropwizard. I wish the documentation for it were more centrally located, however. All options are in the database category and described in the product's javadoc documentation here. All the features we commonly use, such as configuring maximum and minimum pool size, validation behavior, and idle timeout behavior, are all here. An example of this section of the configuration can be found in listing 3.10.

Listing 3.10: Example Dropwizard Connection Pool Settings

```
# Connection Pool settings
database:
  driverClass: org.hsqldb.jdbc.JDBCDriver
  url: jdbc:hsqldb:mem:my-sample
  user: SA
  password:
  autoCommentsEnabled: false
# Sizing options
  initialSize: 1
  minSize: 1
  maxSize: 10
  maxWaitForConnection: 30 seconds
```

```
  evictionInterval: 5 seconds
# Validation behavior
  validationQuery: SELECT CURRENT_DATE AS today, CURRENT_TIME AS now FROM
        (VALUES(0))
  checkConnectionWhileIdle: true
  checkConnectionOnBorrow: false
  checkConnectionOnConnect: false
  checkConnectionOnReturn: false
  validationInterval: 30 seconds
```

As an aside, it's interesting how Dropwizard configuration permits specifying the unit of time in question instead of assuming the often inconvenient "milliseconds." In addition to milliseconds, seconds, minutes, hours, days, and others are supported. This information was gleaned from Dropwizard source (class Duration from dropwizard-util); it's not documented anywhere that I could find.

When it's necessary to configure lower level container settings, Dropwizard provides a way to accomplish this using the input yaml configuration file. A list of the settings offered can be found here. An example of how this can be accomplished can be found in my sample and in listing 3.11.

Listing 3.11: Example Dropwizard Container Settings

```
# Container settings
server:
  type: default
  maxThreads: 150
  minThreads: 20
  maxQueuedRequests: 20
```

Docker

Products such as Spring Boot and DropWizard are reliant on an operating system and java installation to run. In most organizations, that java installation is shared across multiple applications. Any upgrade to that java installation runs a risk of adversely affecting multiple applications and having unintended consequences. In other words, this installation is a type of coupling that's not really wanted in a microservices world. Docker eliminates this coupling as well as any coupling for any dependent native code on which that java application relies.

A Docker deployment can be thought of as a mini-virtual machine. It comes complete with operating system (a linux kernel) and all the native software a

product needs to run. Docker deployments can contain java JDK installs that service that deployment, and that java JDK can be upgraded without fear of adversely affecting multiple services. In essence, it provides upgrade insurance. Those of us who have worked in large corporate environments have seen that upgrading the version of Java is a very large endeavor due to the testing effort inherent in such an upgrade. Bundling the java installation with your deployment alleviates these concerns to a large extent.

Docker has fast become a favorite deployment format. For those of you who haven't used Docker, it is 'package once, run anywhere.' Docker does for software what shipping containers do for tangible goods. It has standard connection mechanisms for disk and networking. Additionally, it's easy to link Docker 'containers' together (e.g., link database Docker container with your application).

Not only is it a win for system administrators as it makes deployment management easier and more easily automated; it's also a win for Java developers and architects. As the applications environment is portable and is packaged with your application, environment-specific problems and defects are greatly reduced. What runs in production is more like what you test with. In addition, since your application runs with its own version of Java and the operating system, you can upgrade your JDK and any native code you rely on with less fear of impacting other applications or even involving system administrators to make the native software change.

Getting past the hype, there are costs and disadvantages to using Docker. Running Docker and its associated tools is much easier on Linux; Windows has limited support. This fact creates inconveniences in most corporate environments where development typically occurs on Windows. Platform differences also present issues when adding maven functionality to produce Docker images; maven builds are supposed to be platform independent. Additionally, Docker does have a learning curve.

Some may fear performance issues with an additional virtualization layer. However, IBM did a good study showing the performance impact to be negligible. All in all, the advantages of using Docker outweigh the costs.

For Maven users, there are a few Maven plugins that will produce Docker images as artifacts. My favorite among them at the time of this writing is the

Spotify docker-maven-plugin. It's very easy to use and install. A usage example can be found in this pom with the Moneta microservice example on my GitHub.

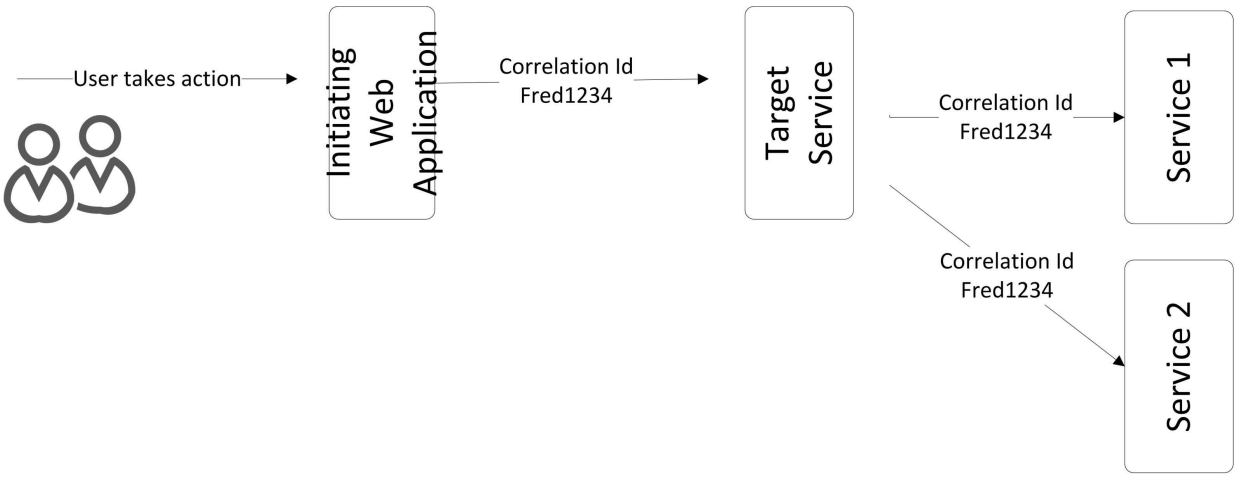
Transaction Tracking

With a traditional web application, it's easier to track transactions all the way through a given business process. Stack traces from exceptions are more meaningful as the code base for a given transaction is often complete.

In a microservices world, this isn't the case. In a world where a business transaction could utilize multiple services, tracking a history for all transactions becomes a necessary item. A common way to track transactions across service calls is through correlation id's.

At the beginning of a transaction, a unique transaction id (or correlation id) is assigned. That correlation id is then logged with the transaction for all log entries. Any resulting service calls for that transaction should contain that correlation id in the header so that it can be logged with any activity, including calls to dependent services.

Figure 3.1: Correlation Id Example



The idea is to ensure that the correlation id is present with any log messages for the transaction, so logs can be combined to provide a complete picture of what happened for a particular user action. Logging products call this a mapped diagnostic context or MDC. Think of MDC as a thread local context for a transaction. You programmatically record the correlation id with the MDC when the transaction starts. This is commonly achieved using a servlet filter or Spring interceptor. You also specify that MDC information as part of the pattern layout in your logging configuration so that the id is reported. An example of a servlet filter that ensures that a correlation id is set, and records that id with the MDC using the Slf4j product, can be found here. Example code ensuring a correlation id exists for a transaction, and recording it with the MDC using the Slf4j logging product, can be found in listing 3.12.

Listing 3.12: Servlet Filter Correlation Id MDC Example

```
public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain chain) throws IOException, ServletException {
    HttpServletRequest httpServletRequest = (HttpServletRequest) request;

    // Ensure that a correlation id is assigned.
    String correlationId = httpServletRequest.getHeader(correlationHeaderName);
    if (StringUtils.isEmpty(correlationId)) {
        correlationId = UUID.randomUUID().toString();
    }

    // Record the correlation id with the MDC
    MDC.put(loggerMdcName, correlationId);

    try {chain.doFilter(httpServletRequest, response);}
    finally {
        // Remove the correlation id from the MDC
        MDC.remove(loggerMdcName);
    }
```

```
}
```

As an example, let's say the correlation id is recorded with the MDC under the label requestId. That requestId can be specified in the pattern layout in your logging configuration. An example logback configuration that allows the correlation id to be output with any log events can be found in listing 3.13.

Listing 3.13: Logback MDC Pattern Example

```
<appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <!-- encoders are assigned the type ch.qos.logback.classic.encoder.PatternLayoutEncoder
        by default -->
    <encoder>
        <pattern>%d{HH:mm:ss.SSS} [${HOSTNAME}] [%thread] %-5level
            %logger{36} REQUEST_ID=%X{requestId} - %msg%n</pattern>
    </encoder>
</appender>
```

Information on the Slf4j implementation for MDC can be found here. Logback and Log4j support MDC functionality.

From the Java perspective, there are at least three parts to this algorithm. Typically, there will be a servlet filter to ensure that each transaction has a correlation id. If it doesn't, one is generated and assigned. Also, that filter registers the correlation id with the logging framework used so that the information is included in log entries automatically.

Secondly, the log format needs to include the correlation id with all log entries. All the major frameworks support this concept. There's an example of a servlet filter such as this that utilizes Slf4J as a logging framework here.

Thirdly, all outgoing service calls need to have the correlation id in the header. Unfortunately, all implementations for this are product specific. It depends on the web service framework used. For the Apache HttpClient product, you should add the correlation id to the header.

The intent for this technique is to cover REST as well as SOAP. You shouldn't use the SOAP headers; use the http headers.

Exception Handling for Remote Calls

With microservices architecture, localizing problems is often where large

amounts of support time is spent. There are some easy steps you can take to reduce effort spent localizing problems.

Utilize Contexted Exceptions

Contexted exceptions from Apache Commons Lang V3 and later have ContextedException and ContextedRuntimeException. These exceptions make it very easy to include information with the exception that makes it easier for support developers to diagnose the root cause. An example of how to use the ContextedRuntimeException is provided in listing 3.14a. Example output when the exception is reported in the log is presented in listing 3.14b. As you can see, it's easy to provide additional information in the log without incurring the risk (e.g., guarding against null pointer exceptions) or labor for formatting the exception message.

Listing 3.14a: ContextedRuntimeException Usage Example

```
throw new ContextedRuntimeException("Error adding customer", rEx)
   .addContextValue("accountId", accountId)
   .addContextValue("accountType", accountType)
   .addContextValue("customerName", customerName);
```

Listing 3.14b: ContextedRuntimeException Log Output Example

```
org.apache.commons.lang3.exception.ContextedRuntimeException: Error adding customer
Exception Context:
      [1:accountId=12345]
      [2:accountType=PERSONAL]
      [3:customerName=null]
---------------------------------
      at examples.commons.lang.context.ContextedExceptionTest.test(ContextedExceptionTest.java:18)
      at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
      at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
      at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
      at java.lang.reflect.Method.invoke(Method.java:606)
      at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:50)
------------ Output omitted for brevity
```

Record request and response for failed remote calls. Include the request and response associated with the call so that if communication with the group supporting the called service becomes necessary, you'll already have all the information that group needs to diagnose the issue. To provide some context on how useful this can be: with a past client, we implemented this practice for all

remote service calls, and as a result we could localize the problem and identify which service had the issue 100% of the time. Additionally, developers saved time because they had a ready-made test case they could execute locally and debug. An example of how to accomplish this using the Apache HttpClient product can be found in listing 3.15.

Listing 3.15: Example HttpClient Exception

```
public static CloseableHttpResponse executeGet(String requestUrl, Map<String,String> headerMap) {
    Validate.notEmpty(requestUrl, "Null or blank requestUrl not allowed.");
    HttpUriRequest uriRequest = new HttpGet(requestUrl);
    for (String header : headerMap.keySet()) {
        uriRequest.addHeader(header, headerMap.get(header));
    }
    CloseableHttpClient httpclient = HttpClients.createDefault();
    CloseableHttpResponse response = null;
    try {
        response = httpclient.execute(uriRequest);
        logger.info("Sevice Request Execution. requestUrl={} \nheaders={} \nresponse={}",
            uriRequest, headerMap, response);
    } catch (Exception e) {
        throw new ContextedRuntimeException(
                "error making RESTful web service call", e)
            .addContextValue("requestUrl", requestUrl)
            .addContextValue("headers", headerMap);
    }
    return response;
}
```

Some people would go further and recommend capturing all requests and responses for a configurable length of time. This facilitates debugging problems that don’t result in an exception, i.e., problems that result in incorrect behavior, but don’t except. Having a library of searchable requests/responses will save developers time when investigating support issues.

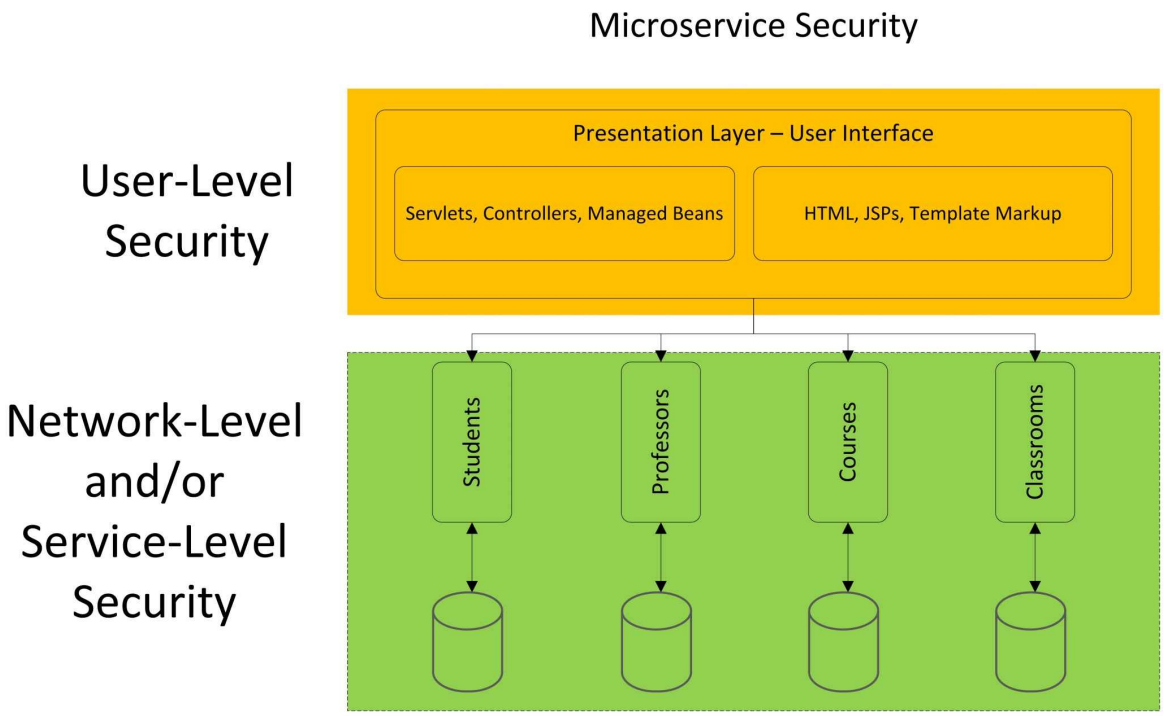
Service Security

Security is implemented in layers. Each layer represents a ‘hurdle’ for an attacker. The more sensitive the secured resource is, the more layers of security are placed around it; making unauthorized access to the secured resource more difficult. This is a basic security concept that we utilize everywhere. As a simple example of a security layer, organizations use firewalls to place a security level around internal resources; and you must have at least internal access per the

firewall to access internal web applications and resources. Organizations don't stop with this layer; they normally add another layer using access groups (e.g., LDAP groups) to differentiate which users can access which resources – but you get the idea. Each layer presents an obstacle for attackers, thus making the resource harder to breach. Microservice security is no exception and fits into this paradigm rather well.

Restrict network access to your production microservice library. The first line of defense is networking. It's not uncommon to put your entire server-side network architecture, including your microservice library, behind a firewall and restrict virtual machines (VMs) that have access to it. That is, the firewall allows consumers for your microservice library in, but nothing else. If you implement the API Gateway pattern, your web applications (both internal and external) are the only consumers of your microservice library and are VMs that truly need access. Support personnel can be provided remote access via some support environment that has network access. Most people in the organization do not need direct access to the microservice library and can easily be restricted from using it via firewall. A diagram of this concept can be found in Figure 3.2. Developers will need access to a non-production implementation of your microservice library for development, as well as non-production versions of any required databases and other resources.

Figure 3.2: Network Access for Microservice Libraries



The advantages to using network access security for your microservice library are several: It's easy and comprehensive. There's absolutely no impact on the microservices themselves or the packaging of them when you implement the security layer; and this layer can be enhanced and changed without affecting individually deployed microservices in the slightest. Network access security takes no planning on the developer side; however, there are some services that are more sensitive than others. Examples I can think of are human resources (e.g., salaries and sensitive personal information about employees) or medical records. Some services and data are going to need an additional level of security.

Establish service level accounts for services that need additional security. This is very similar to how database data is usually secured. End user security credentials aren't used for database connections; service accounts are established so we can utilize connection pooling. Services are no different. Most secure microservices use OAUTH as they typically provide a RESTful web service interface. Furthermore, make the service accounts and passwords configurable as we do with database credentials. The mechanics of securing RESTful web services using OAUTH are no different for microservices than for other types of RESTful web services and won't be directly addressed in this book.

Don’t provision microservice access directly to end users. End user security for your web applications shouldn’t be any different now that they are relying on a microservice library. End user credentials are often needed to allow/restrict individual users from specific features of the web application. Using a microservices architecture doesn’t alter this need in the slightest. However, do not pass end user credentials onto your service library. That would add an extreme level of complexity to managing security and wouldn’t provide any additional benefits.

Contract Testing Procedures and Requirements

Scheduled and automated contract testing procedures for microservice libraries are critical for several reasons. This section will discuss these concerns.

Continuous integration for microservices is essential. Microservices architecture facilitates higher development throughput. That means a higher frequency of production deployments and less opportunity for manual integration testing. The implication here is that automated contract testing for microservice libraries is essential. Without the safeguard of continuous integration and the automated contract testing that comes with it, you’ll find defects migrated to production. Continuous integration is the main safeguard against deploying defects to production. Unit testing for code within microservices, while useful, isn’t as effective as a safeguard in a microservices architecture. Some even advocate abandoning unit testing altogether in favor of contract testing, but I’m not quite willing to go that far.

Continuous integration compensates for developer myopia. In a world where developers work on one service at a time and increasingly rely on service calls, they don’t have the complete context for any change. They don’t understand exactly what might break from a business perspective should they deploy a defect. You might see this as a negative, but it’s purposely part of the paradigm to make developers more focused and productive. My point is that developer myopia needs mitigation, and continuous integration and automated testing is that mitigation.

Continuous integration allows individual microservices to be truly disposable. In a world where services approach being ‘disposable’ – that is, it’s relatively easy to scrap one version of a service and completely rewrite it – a new version of a service needs to do exactly what the old one did. With a healthy automated

test library, developers can rewrite and refactor services more safely and with less fear of causing production issues.

Continuous integration prevents developer mistakes from proceeding to production. From a management perspective, contract testing is your bulkhead against rogue developers. We've all had to manage developers that don't seem to take direction and want to do their own thing, sometimes not for the good of the organization. Microservices architecture gives you an opportunity to put a container around rogue developers, particularly if you have rigorous contract testing in place.

Consumer-based Contract Testing

There's a comparatively recent development with contract testing that's worth considering. Most of us are used to writing contract tests for services we publish. In a world where services are more reliant on other services, some organizations are having consumers test the functionality of services they depend on. That is, you consume a service, and you contribute automated tests for features in that service that you rely on. That service can't deploy until it passes all of its consumer-provided tests as well as those provided by the service publisher.

For process services – that is, those that rely on other services – you can argue that this is merely a semantic change. Contract tests for process services, by definition, test services they rely on. In this way, everyone implementing microservices will, in effect, be implementing consumer-based testing at some level. However, with most process service tests, there won't be a direct correlation between that test failure and the effective failure of one of the services it relies on. That is, without some type of investigation, it won't be obvious that a service contract change is the cause of a process service test failure.

While this creates more work for service publishers, it adds safety. You could also look at it as optimizing support resources. Is it more efficient to react when a defect goes into production or is it easier to prevent it? Many organizations have come to the conclusion that it's cheaper and easier to prevent it; hence consumer-based testing.

Consumer-based testing guards against unannounced contract changes in the services you consume. If a service can't proceed to production without passing all consumer-based contract tests, developers for that service will be much less

likely to introduce contract-breaking changes. It will (properly) be put as a last resort for them.

Tooling Support for Contract Testing

Fortunately, as RESTful web services and SOA architectures have been around for a long time, the tooling needed to support contract testing for microservices is plentiful. Some products to consider are the following:

- Apache HttpClient for RESTful web service contract testing
- SoapUI for either Soap or RESTful web service contract testing
- ActiveMQ for JMS or AQMP testing. They have an effective embedded broker that works for testing
- Jenkins for automated and scheduled builds.

Chapter 4: Microservice Usage Guidelines

This chapter provides advice on when to use microservices architecture. Hint: it's not all the time. When new paradigms arrive, they tend to be over-used. Microservice usage is no exception. This chapter will provide guidance on when microservices architecture can provide benefit, and it will attempt to separate the wheat from the chaff. That is, it will tell you where the marketing fluff is – hint: there is some!

When to Use Microservices Architecture

Microservices architecture isn't appropriate for all applications. Any distributed application paradigm, such as microservices, is by definition more complex and takes more resources to maintain. This section will detail when microservices architecture should be considered.

Evolve into a microservices architecture. As a general rule, applications should be initially created as traditional applications and then evolve into hybrid or microservices architecture as their complexity grows. Traditional web applications are easier to initially build and support. YAGNI (You Aren't Going to Need It) applies. Many applications won't grow to be complex enough to warrant using microservices architecture. Don't go to the trouble and expense before it's warranted.

Microservices architecture isn't all or nothing. It's possible for a traditional application to rely on a handful of microservices; it's not necessary for the entire application to evolve into a microservices architecture. In fact, as a traditional web application evolves into a microservices architecture, the transition usually involves deployments where a traditional web application relies on a small microservice library for some portion of the work that it does.

Initial attempts developing microservices should be small and targeted. As with most paradigms, a measured migration to microservices architecture is warranted. You need the experience utilizing microservices, along with DevOps procedures needed to support a greatly increased number of deployables, before you will be effective at maintaining a large microservices library. Good initial proof of concept microservices should be important enough to be noticed, but contained enough so that business risk can be managed.

Multiple applications need the same functionality. With traditional web

applications, you can share a common library of some sort, or publish services in one application that are consumed by another. This becomes complex when external resources, such as database resources, are needed by this shared functionality. This shared functionality is a prime candidate for separation into a separate, independent deployable with its own database.

Parts of an application have different scaling requirements. Separating out functionality into microservices allows that functionality to scale differently. For example, one microservice might only need three nodes in a cluster while another might need dozens due to its larger transaction volume.

Parts of an application may have higher availability requirements. Separating out functionality into microservices not only allows increased transaction throughput but makes the service less likely to be completely down. In other words, the increased scale also lessens the chance that all nodes of a service cluster would be down, making the service unavailable.

Warning signs that your traditional application is too large:

Given that most applications *evolved* into using microservices architectures and didn't start there, how do you know when your application(s) has reached a point where microservices architecture should be considered? There's no objective rule to use, but I can provide some guidelines.

Changes come with unintended consequences. When applications get large and complex, deployments contain more risk. It becomes harder and harder to make isolated and targeted changes without unintended consequences. With these applications, when you fix one problem, other problems appear unintentionally. It's like playing the whack-a-mole game where you hit one mole only to see another appear. This is a sign that an application has become too complex to manage as a single unit and can benefit from breaking the application into smaller pieces with paradigms such as microservices architecture.

Release testing / QA cycles become unreasonably large. This is a sign that an application has become too large and complex to be effectively managed as a unit. In these cases, breaking up the large problem into more manageable smaller problems can reduce support costs. Microservices architecture can be useful in situations like this.

Development teams for a single application become too large to manage

effectively. I go with the Jeff Bezos quote: ‘If you can’t feed the team on two large pizzas, then the team is too large.’ Microservices architecture can make it easier to manage large development efforts as work units are broken into independent, more manageable teams.

Where’s the Marketing Fluff?

With any new paradigm, there’s a certain amount of hype and marketing that comes along with it. Microservices is no exception. This section will attempt to separate the wheat from the chaff and let you know where those marketing fluff items are.

Are Microservices easier to manage?

One of the selling points is that yes, they are. But you’ll need to observe the requirements in fine print before microservices architecture will be easier to manage.

You must be very good at contract management. Your management efficiency comes from reduction in the need for communication. That reduction comes from organizing teams into much smaller, more focused work groups. With microservices architecture, much of that communication comes from interaction with other teams to design features in your service contract. How effective you are at service contract management directly impacts how much communication is needed between teams.

You must be very good at specifying the mission and features of each service. A close corollary to this is documenting that service for your consumers. To see how this affects your ability to manage microservices architecture, look at what happens when services aren’t effectively documented. To consume a poorly documented service, you need to enlist the assistance of the providing team to learn how to effectively use it. This greatly increases communication overhead and requires coordinating multiple people. It also leads to a larger number of technical questions and issues during development to correct code consuming the service. In other words, providing good documentation along with usage examples is a win-win. Service providers don’t spend as much on support, and consuming teams are able to develop much more rapidly.

Services must be truly context independent and self-sufficient. Services should not place restrictions on the context in which they are invoked. For instance,

services should be as easily usable in a user interface as in a batch process. All services and service calls should be stateless. For instance, there should be no context (e.g., session variable content) that travels from call to call. If the service uses a database, all service calls are complete units of work. No open transactions are left when a service call completes.

Are Microservices easier for developers to understand?

One of the selling points for microservices architecture is that it makes problems small enough to fit into the developers' heads, but in reality, the answer is 'no.' It is easier for a developer to understand one particular service in isolation. No argument with this. It's a smaller code base. Smaller feature set. Just smaller and more focused.

Where this selling point is misleading is that, in a microservices world, it can take dozens of services working in concert to provide a given piece of business functionality. Yes, if you've drawn boundaries effectively, developers can look at services as 'black boxes' and not delve into the underlying mechanics of exactly how that service does what it does. But the large number of them means that the entire service set is like a complex API that developers need to understand to write process services effectively. Essentially, the complexity here is no different than understanding a large number of packages in a traditional application.

There is a reason that I didn't list this as a benefit before now; I don't agree with the assertion that microservice development is less complex. While it is easier to understand one individual part of the entire application, it is not easier to understand the application as a whole.

Do Microservices increase development throughput?

That is, does developer velocity increase for efforts using microservices architecture? The answer is 'maybe.'

You must be good at contract management. As explained earlier, your ability here directly impacts communication overhead. Communication overhead drastically reduces your velocity and development throughput.

The goal is really misstated. The goal should be 'increasing the speed at which you deliver functionality to end-users and to business units.' While it's easier to measure developer velocity for individual teams, it's functionality delivered to the business that is really important. It's common for new business level features

to require multiple new services or feature enhancements to several microservices. This means that effective contract management really means coordinating contract enhancements across multiple service teams. Without that coordination, developer velocity might increase, but velocity at which business functionality is delivered might not. This coordination is required in order to be effective.

Are Microservices truly disposable?

Microservices are certainly *more* disposable than traditional web applications. That said, this benefit is a bit over-stated. Let's think this through.

A replacement for a service better do *exactly* what its predecessor did, or you'll have defects and unintended consequences. Furthermore, the replacement service needs to address *all* cross-cutting concerns as its predecessor did. In order to effectively replace a service without introducing defects, you need rigorous contract testing.

Replace a microservice exactly before making changes; never combine changes with the replacement. This simplifies testing. You can run your test suite against the old and the new – and you should always get the same answer. If there are intended changes in the new version, then it's much harder to do this. This begs the question: do you replicate any bugs you discover in the process of rewriting a service? I say 'yes'; it keeps testing simple. It's possible that one of your consumers discovered your bug and is currently coded to work around it; fixing that bug would break such a consumer.

You must be good at contract testing. Without rigorous contract testing capabilities, deploying a rewritten version of a service is a risk. Run your test suite against both service versions and it should pass. The implication here is that the test suite is specific to the contract being tested, but not the underlying service specifically.

You must be good at providing robust application architecture support. You don't want developers of new services spending more than configuration / set-up time on addressing cross-cutting concerns. If you do, that increases the cost of establishing new services. It also makes it more difficult to replace services as they need to address these cross-cutting concerns as well.

Best Practices and Common Mistakes

As with all new paradigms, best practices and common mistakes in usage of that paradigm emerge as it matures. We're starting to see that now with microservices architecture. This section will detail some of them.

Best Practices

Record Microservice Requests and Responses

This helps support developers diagnose problems that don't result in an exception. Often, this log is a rolling log that only keeps transactions for a configurable period of time for space reasons. This tremendously saves support developer time as often you don't need to take time to build/replicate a transaction that produced unexpected/unwanted results; that transaction already exists in your history and it is often quicker to find it rather than re-create it.

Correlate service requests with the response. I've stated that a log management software implementation, such as Splunk or Logstash, is essential for microservices architecture. Recording requests and responses can be as simple as logging them and making those logs available to developers. It's imperative that developers be able to match the request and response for each service call. If you utilize the technique of associating each log entry with a unique transaction or correlation id, as described in a preceding chapter, all you need to do is log all the requests and responses, and developers will have this information.

Common Mistakes

As with any technology paradigm, microservices architecture can be misused.

Inappropriate service boundaries

Services should be truly loosely coupled and self-sufficient. If you find that one change requires deployment of multiple services, it most likely means that you have coupling you shouldn't have or are intentionally introducing a contract-breaking change. While the latter is sometimes unavoidable, the former usually indicates a contract boundary design flaw.

Services shouldn't care about execution context. For instance, they should be just as usable with a user interface and batch process. If you experience unintended consequences when you acquire new consumers, it most likely means that the service isn't truly context independent; that it has some contextual requirement that your new consumer somehow isn't meeting. This is

most likely some type of contract boundary design flaw as well.

Exposing microservices to the browser

Microservices are not meant for public consumption. Much of the reasoning for this was presented with the ‘back ends for front-ends’ pattern for deploying user interfaces. Organizations do not control performance on the public internet. Combine that with the fact that user actions often involve multiple service calls and it becomes obvious that some call consolidation is needed for user requests.

Ineffective transaction logging

The amount of time spent localizing problems (figuring out which service contains the root cause of a problem) is much greater with microservices architecture. Effective logging of microservice requests and responses can reduce this time tremendously. With the increased number of deployables inherent with microservices architecture, localizing the services in which defects occur will be a time-consuming problem without access to service requests and responses.

Ineffective exception handling

A high percentage of exception reports should contain enough information in the exception to at least localize the root cause of the problem, if not make the root cause visible. Without that information, support developers are left with time-consuming trial-and-error to attempt to replicate the issue in a test environment. This burns time.

Ineffective request validation

Catching errors earlier rather than later will always save time in the long run. Letting invalid requests through and eventually degenerating into some type of derivative exception will always cost support developers more time than having the exception reported earlier and closer to the scene of the crime.

Ineffective response validation

Have you ever received null pointer exceptions or other types of derivative exceptions because a service you called didn’t return what was expected? If you rely on the results, it’s better to validate those values on return so that support developers get a clearer error message. The reasoning here is very similar to why it’s important to validate request data.

Why Now?

SOA and modularizing functionality via services has been around for a long time, so microservices architecture isn't an entirely new idea. That begs the question: why now?

Virtualization and Cloud Advances make microservices architecture feasible. When SOA was popular, finding and installing hardware for a plethora of new services was a real issue. Now, most organizations have adopted hybrid clouds and an administrator can easily establish any new virtual machines that are needed to host new services.

Dev Ops Automation makes it possible to support larger numbers of deployables. When SOA was popular, supporting an exponentially larger number of deployables was a real issue for operations staff. Back in those days, set-ups were manual and often re-invented. In a dev ops world, new environments can be configured and deployment happens in an automated fashion, thanks to tools like Docker, Chef, and Puppet; so not nearly as much manual effort is needed.

Hardware and network infrastructure is cheaper than it's ever been. In the old days, tuning resource consumption was more imperative, as equipment was extremely expensive. Now the extra memory and process overhead required by microservices architecture isn't nearly as much of a concern.

Future Directions for Modularity

Microservices architecture isn't the only paradigm for modularizing Java EE applications. This section will briefly describe other attempts to modularize Java EE and contrast them with microservices architecture.

Java 9 with Jigsaw

Jigsaw formally introduces modularity to the Java platform. An early release is available, although a production release isn't expected until 2016. Essentially, this feature comes with a new deployable (the module library or jmod) that does work in-process but is guaranteed its own mini-runtime environment. More detail about Jigsaw can be found here.

There are several differences when comparing Jigsaw to microservices architecture.

Jigsaw has a couple of advantages:

- Lower memory consumption as it's in process
- Compiler-enforcement of contract (as opposed to runtime).

However, microservices architecture retains several advantages over Jigsaw:

- Services are independently scalable.
- Services are technology stack independent; not restricted to java.
- Service deployments are independently manageable; not compiled into the runtime artifact.

As we've discussed with the lower cost of hardware, optimizing memory isn't as important as it once was. Overall, microservices architecture offers the enterprise more freedom and flexibility than Jigsaw.

OSGi

OSGi is a modularity solution for the Java platform, invented before modularity was formally supported by the platform. It has the exact same list of advantages and disadvantages – compared to microservices architecture – that Jigsaw does. In fact, you can look at Jigsaw as filling the same goals as OSGi.

There are several differences when comparing OSGi to microservices architecture. OSGi has a couple of advantages:

- Lower memory consumption as it's in process
- Compiler-enforcement of contract (as opposed to runtime).

Microservices architecture retains several advantages over OSGi:

- Services are independently scalable.
- Services are technology stack independent; not restricted to java.
- Service deployments are independently manageable; not compiled into the runtime artifact.

By the way, OSGi has had very low adoption rates. If you look at statistics from indeed.com or Google Trends, jobs for OSGi rose steadily until 2011 and have oscillated at the same low level ever since.

Open Source Adoption

I'm seeing an ever larger growth of open source products meant to be used as

microservices being released on GitHub and other open source forges. I expect this trend to continue. As an example, searching GitHub for the term ‘microservice’ in June 2015 resulted in slightly over 1,100 repositories. In January 2016, that number was slightly over 3,200. While it’s certainly true that not all of those repositories contain products designed to be deployed as microservices, some percentage of them will be, and it’s fair to suspect that the number has grown over the past few months.

Chapter 5: Managing Microservice Development

The implications of microservices architecture for developers and architects are large. The implications for management are just as large. Most authors and presenters for microservices architecture topics concentrate on the more technical aspects of microservices. As architects, we're supposed to deliver business value. Part of that value is advising management about what the ramifications of adopting microservices architecture are, and that they have a role to play.

Prerequisites for Microservices Architecture Adoption

Microservices architecture has infrastructure requirements that need to be in place and working before adoption of microservices is feasible on a large scale. This section will try to provide guidance to architects advising management and to managers themselves.

Changes in Governance

You no doubt have processes and procedures that govern the lifecycle for moving development through testing and user acceptance to production. Perhaps there are forms that need to be filled out and approvals that need to be acquired before production migrations happen. Perhaps there are prerequisites that developers need to meet before those approvals are granted and production migrations happen. The intent of these procedures is often to mitigate deployment risk and prevent defects from being migrated to production. I'm sure that those processes work quite well for the volume of deployments you currently deal with, without having microservices architecture in place.

The goal is to provide governance and infrastructure support for microservices architecture, but to do so in a way that doesn't slow everything down. Remember that one of the benefits is supposed to be higher development velocity. Governance and velocity are *diametrically opposed*.

Most governance procedures won't scale exponentially. Microservices architecture will exponentially increase the number of deployments you manage. Most processes that organizations put in place aren't designed to grow to handle 10 or 20 times your current volume. It's likely that your bureaucratic process for moving development to production won't survive that type of growth either. It's more likely that your governance procedures will need to change and adapt to be

able to handle the exponential growth that microservices architecture will present.

Automate the production deployment process, including testing and approvals. One of the benefits of microservices architecture is that microservices have 'virtual' users. As such, there are no user interfaces for them that need visual inspection to verify aesthetics, etc. It's possible to entirely automate tests for microservices. Web application user interfaces would still need to be manually inspected to make sure that the more subjective qualities that they have are correct, but microservices don't need that. Yes, it takes investment to automate the entire process, but if you invest heavily in microservices with the intention of reaping the benefits this paradigm provides, the automation will pay for itself at some point.

Automate defect detection and backout initiation. Automated testing will provide some level of protection against defective deployments. An additional level of protection to consider is putting a mechanism in place to detect changes in failure rates *after* deployment and automatically initiate a rollback if the increase in failure rate exceeds a configurable threshold. Anyone who is not comfortable with relying on an entirely automated deployment process might increase their comfort level by adding an automatic rollback, should a deployment produce errors.

Investment in Operations Automation

Embrace a DevOps culture. The number of deployables with microservices architecture grows exponentially, sometimes as many as 20 times what you have with traditional web applications. It is not possible to have completely manual processes for establishing new environments and managing deployments. Embracing a DevOps culture where environments are created and managed programmatically, and reducing the amount of manual labor per environment, is absolutely essential for managing a large number of deployables.

The ability to create and secure new environments quickly is essential. The number of virtual machines you use will increase as new services are identified and new resources requested. The growth in virtual machines can be reduced by utilizing the Apache Mesos product or some other type of resource management software. That in itself requires work and additional investment. Cloud integration isn't technically required to adopt microservices architecture, but

many users do choose to integrate so they have the flexibility to exponentially grow the number of virtual machines they support in a short period of time.

Demands on Database Administration Staff will increase. Most people think of additional demands on operations staff. As a microservices architecture also increases the number of databases as well as the number of deployables, demands for new schema/database creations and enhancements will *also* increase. This means that demands on database administrators will *also* increase. Some organizations migrate that onto the microservice teams themselves, but the work doesn't go away no matter how it's organized. It's not often discussed, but database administrators will also need to adopt *a DevOps culture* and automate more of what they do.

Service Contract Management

Not a lot has been written about how microservice teams get their feature enhancement assignments. Microservices, by their very nature, provide the guts of the capabilities users request, but not in a context that those users are likely to understand. It's more likely that users will make a feature request of the team providing them a web application or user interface. Upon realizing that they don't have the capability to deliver on the feature request, the team will need to pass the non-GUI portions of the request onto one or more microservice teams to implement. For instance, suppose a UI team receives a request to store a professor's syllabus for a specific class and there's no service currently available that has the ability to remember that information. Somebody needs to shepherd that request, figure out which service it belongs in, ensure that the feature request contains enough information to be actionable, and route it appropriately. There's a body of work here that needs to be accomplished and accommodated somehow. Most organizations utilize some type of task management software (e.g., Jira) to manage and track feature requests such as this.

Service portfolio management will be essential. Yes, you can let developers design and publish service contracts as a grassroots effort. In other words, let them do what they want and pay no attention. Initially, this is the easiest way to manage. However, as the number of deployed services grows, the developers won't be able to see the forest for the trees. You'll get lots of duplication, and you'll get lots of inconsistency in the services that are created and deployed. Eventually, these defects will grow as the list of services increases and will erode some of the benefits you get from microservices architecture.

I would never suggest that you impose a formal process to review and bless all service contracts before they are deployed and used. That would hinder developer velocity, which is one of the intended benefits of microservices architecture. However, somebody should trail behind the development teams and catalog published services in a standard format available to everyone, including management. They would then be in a position to document service inconsistencies, send change recommendations to service teams, and help facilitate developers adopting existing services – as opposed to inventing new versions of existing services. They would also be able to point out potential risk areas to management, should there be any.

Managing contract change will be essential. In a microservices world, the quality of the contracts will greatly impact the amount of benefit you receive from adopting microservices architecture. Poorly defined contracts will lead to confusion and chaos and negatively impact development velocity for new features. Some organizations are careful to limit the amount of change at any one time to keep quality high and guard against negatively impacting developer velocity.

Feature enhancement assignments need to be unambiguous and clear to microservice team members. Remember that higher developer velocity is coming from the developer's ability to be focused, and free from communication overhead. Giving them unclear feature assignments means that they will need to go and gather clarity from the users requesting them. Some process for grooming and tracking feature enhancement requests needs to be in place.

Service Testing and Deployment Automation

Investment in automated testing is essential. With higher development velocity comes an exponentially higher production deployment rate. Without testing automation, defects will quickly wreak havoc on your production environment. Automated contract testing is essential for mitigating these effects, as in this world there often isn't time for manual testing. This automation isn't free; it requires labor to set up as well as care and feeding. It also requires enforcement.

Automated deployments are essential. With increased deployment velocity, manual intervention with each deployment isn't as possible as it used to be. An argument can also be made that manual steps in deployments actually add risk because they add a chance of error. Yes, this statement assumes a well-written,

well-tested deployment process that typically doesn't have a failure history. It goes without saying that only products with green test suites should be allowed into production.

Database changes should be automated too. Often database changes accompany service deployments of new features. These days, it's possible for those changes to also be automated. Products like FlyWayDb are good for managing changes like this. Yes, back outs or failed deployments will still require a certain amount of manual intervention, but hopefully, failed deployments will be a small percentage of the total.

Consider providing a process for consumer-based testing. Consumer-based tests need to be automated and to be part of the test suite run against services. You can treat this as a test addition that must be green in order for a production deployment to proceed. The only difference here is that the consuming team is providing and maintaining them. I would treat this as an 'option' for service teams consuming other services rather than a requirement. In the past, I've had to rely on services that proved to be less than reliable, resulting in many failing tests. For unreliable services, it's nice to know at the *outset* that dependent services are down and the test failures you're getting are expected. It saves investigation time.

Team Management

Manage the contract deliverables for teams, not the internal management of the teams themselves. One of the benefits of microservices architecture is the elimination of communication overhead. Managing by deliverable, and not micromanaging teams, helps eliminate that overhead. As long as you're getting feature enhancements at an acceptable rate and quality, the way the team chooses to organize can be left to them.

Developers should be working on one service at a time. This helps guard against making assumptions about business processes using services in a specific order or context. I say this because you won't have a developer working on a service and its consuming service at the same time. This also helps them focus without distractions and keeps their velocity high. In this world, anything that drags down developer velocity sacrifices the benefit you get from microservices architecture.

Developers who publish a service should be made to support it. This incents

intelligent development product decisions and writing high-quality code that is supportable. In other words, developers ‘eat their own dog food.’ Most developers want a good family life; this paradigm supports that. As long as they produce high-quality services that have a low defect rate, then the time they spend on support is minimized.

Service Health Monitoring

An automated response to defects is essential. With higher development velocity comes an exponentially higher production deployment rate. Health monitoring is essential to discovering outages and defects quickly. Some of my clients use Splunk or LogStash to automatically create a defect ticket in a ticketing system when errors are encountered in production. Deciding which service team should be notified when an error is encountered deserves some thought and planning. For instance, a service call that errors on erroneous input should go to the consuming team, not the team producing the service. Certainly, any type of unplanned runtime exception should go to the team producing the service.

Dealing with error repeats also deserves consideration. One defect could get thousands of reports, depending on volume. Getting thousands of tickets for the one defect is hardly helpful. Suppressing the repeats is dangerous as detecting what’s a repeat of a previous issue and what truly has a different cause can be problematic.

Management Benefits for Microservices Architecture Adoption

Much of the literature concentrates on describing the technical benefits that microservices architecture provides. There are business/management benefits too. This section will detail some of those.

Resource Staffing Flexibility and Insulation

Microservice development teams are small and focused. Developers **only** work on one service at a time. That service is supposed to have a firm contract that those developers are coding to, preferably with containerized deliverables (i.e., Docker). The point is that they’re all about heads-down development. They shouldn’t be burdened (or have to be burdened) with communication overhead; only with other developers working on that service.

They are working as an independent team, completely asynchronously from all

other service teams. Combined with the fact that they have low communication overhead and defined deliverables, they can be anyplace in the world in any time zone. It doesn't matter whether they are in-sourced in a corporate office somewhere or off-shore.

Given that the work of the team is confined to the focused requirements for the service, worries about code quality, etc. are at least contained to the small service source code base. In other words, there's a limit to how much code that team can mess up and thus create maintenance issues.

This means that development for microservices services can be more easily outsourced.

This also means that you're better insulated from rogue developers. We've all managed developers who produce unmaintainable code. As rogue developers are confined to one small and focused code base, there's a limit to how much damage they can do to that code base. Furthermore, that code base is small, focused, and well-defined; it's more replaceable if need be. You're not letting rogue developers loose in a 500K line monolithic application to cause havoc on a wider scale.

Increased Development Velocity

Assuming that contracts for needed services are fully defined, along with the dependencies between those services, it's easier to bring larger number of development teams on for ongoing efforts. The concern about bringing on additional developers is that this typically means more communication overhead. As a result of that increased overhead, developer productivity has diminishing returns. You get less and less productivity out of each additional developer.

While individual developer productivity does increase in microservice teams because they don't have as much communication overhead, there is a limit to how much productivity increase you'll get with any individual developer. Your velocity increase really comes from the ease with which new teams can be added. Developers never see or are impacted by the size of the overall effort as they are doing heads-down development in a small, focused team. Consequently, it's easier for management to scale development to larger numbers of developers. Note that in addition to having all service contracts fully defined, you also have to have identified any dependencies between contracts; some services might require that other services be completed before work on a particular service can

finish. Note that any development team can stub dependent services and start work before dependent services are finished.

What really happens here is that the constraint to scaling development shifts to management and enterprise architects responsible for identifying individual services and specifying the service contracts. With larger efforts will come more coordination points. As some services call other services, those dependent services need to be built first. The order in which work flows down to development teams needs to be coordinated.

Technical Stack Freedom

In a microservices world, services are containerized. That is, the deployment artifact has an operating system and other software that it needs (e.g., Java JVM, PHP, Python, GoLang, etc.). Each service can easily operate at different levels of these technologies. The corollary is that new technologies can be more easily introduced or upgraded. That makes it easier for you to capitalize on newer technologies much more quickly than you can with traditional monolithic web applications.

With traditional web applications, the code base is much larger. Therefore, the impact of upgrades will be much larger and require more extensive regression testing. Moreover, switching development languages is much more difficult with the larger code base.

This is a technical benefit, but I believe it to be a management benefit as well. Managers are always looking for new ways to deliver software faster. One of the avenues for increased developer productivity is technology advances with the available toolsets. Your ability to utilize newly developed toolsets and bring about productivity increases that the business wants is better with microservices architecture than with traditional web deployments.

Further Reading

If you're interested in microservices architecture, there are several additional resources I can point you to.

My interest in microservices began when I viewed this presentation by Fred George. I ran into it quite by accident, but I was fascinated by the ramifications of microservices architecture that he articulates. In essence, he instigated this book, although he probably has no knowledge of this. Specifically, I was intrigued by the notion of services being 'disposable.'

The definitive guide on what microservices architecture is, and on defining the paradigm, is *Microservices* by Fowler and Lewis. You may recall that I defined microservices as a list of characteristics that most microservices have. I wish I could claim that as original thought, but I can't. The Fowler/Lewis article is the definitive standard for a list of traits that microservices have. I chose what I deemed to be the most important traits to highlight.

Fowler has also done valuable work on prerequisites for microservices architecture (here); that is, capabilities you should have in place before attempting microservices architecture on a large scale.

Another thought leader on microservices is Adrian Cockroft. A sample of his work can be found on this video. I follow his twitter feed.

I closely follow two open source projects in this space. The first is the Netflix product suite. They have to support an incredible amount of throughput and have been generous enough to open source the tooling they built to support it. The second is Apache Mesos, which essentially makes computing resources ubiquitous. That is, it allows you to look at computing resources generically and not be concerned with where those resources are hosted. With the large number of deployables that microservices architecture produces, this is very promising for reducing deployment management issues.

Thank you for taking the time to read this book. Feel free to contact me at derek.ashmore@dvtconsulting.com or via LinkedIn if you have comments or questions. I'm always interested in feedback.