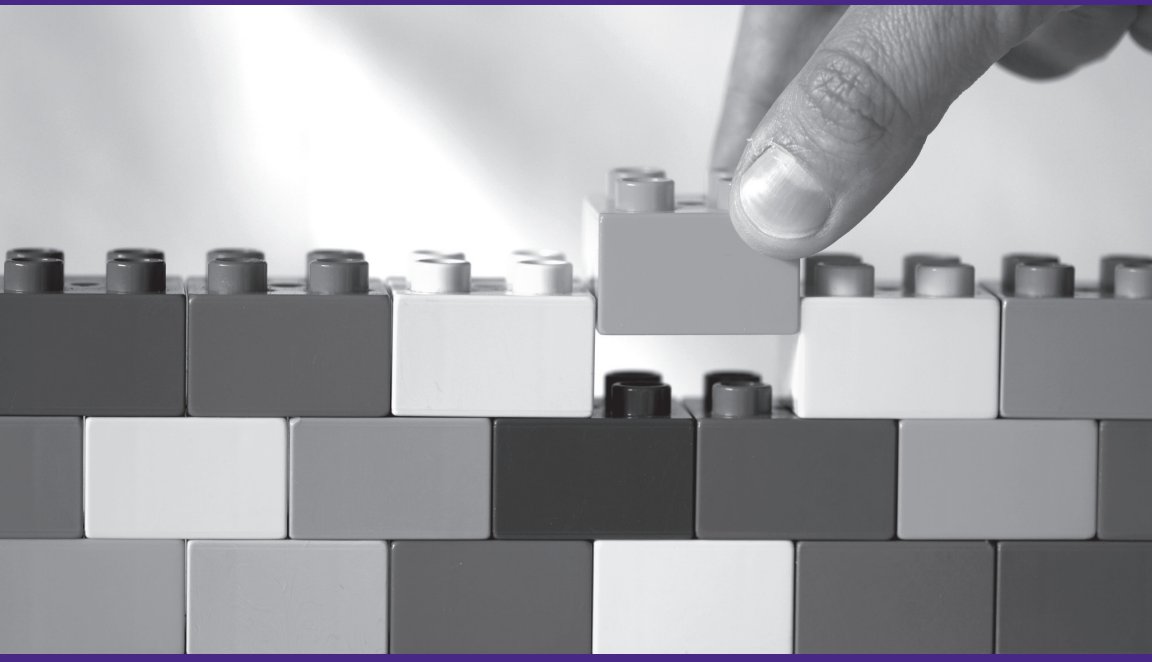
O'REILLY®

Compliments of commercetools

Microservices for Modern Commerce

Dramatically Increase Development Velocity by Applying Microservices to Commerce



Kelly Goetsch

commercetools

The end of commerce platforms as you know them

Quickly deliver new features to market

Rapidly iterate to find your next business model

Improve developer engagement

commercetools offers the industry's first API and cloud-based solution for commerce which is built for use alongside microservices. Use all of our APIs or just the ones you need. Take your business to the next level with *commercetools* and microservices.

Learn more at: **commercetools.com**

Microservices for Modern Commerce

Dramatically Increase Development Velocity by Applying Microservices to Commerce

Kelly Goetsch

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

Microservices for Modern Commerce
by Kelly Goetsch

Printed in the United States of America.

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

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

Editors: Nan Barber and Brian Foster
Production Editor: Kristen Brown
Copyeditor: Octal Publishing, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest
Technical Reviewers: Sachin Pikle, Tony Moores, Oleg Ilyenko, and Christoph Neijenhuis

October 2016: First Edition

Revision History for the First Edition
2016-10-26: First Release

978-1-491-97092-8

[LSI]

Table of Contents

Foreword

We at Rewe Group, a 90-year–old international retailer with €52 billion in revenue across 40 brands with more than 12,000 physical stores, are in the midst of an end-to-end digital transformation of our entire business. Our competitors today are technology companies—not other retailers. Innovation through technology is now at the very core of our business. Technology is what gets the right product to the right person, at the right time.

I have long believed that the role of the Chief Executive Officer and Chief Product Officer would merge, as organizations shift focus to a product-oriented mindset. Most CEOs agreed with me but have found it impossible to accomplish because of the legacy enterprise technology that powers business, particularly retail. It is not possible to run an agile business in today's world while running technology that was developed in the 1990's for a different era. Quarterly releases to production are no longer acceptable. Instead, releases to production must occur multiple times a day. It's taken 15 years for a new approach to surface; that new approach is microservices.

Microservices are central to our new approach to commerce. We now draw from an infinite pool of engineering talent across Europe to build hundreds of microservices, all in parallel. The value of microservices to us is innovation. We can quickly assemble teams. Once established, each team can then iterate on a new feature in production over the course of hours rather than the months or even years it would have taken us using the traditional approach. Today's infrastructure is all public cloud-based, which offers limitless elasticity. Teams are now owners of products, with all of the tools required to autonomously innovate.

We now have a large catalog with hundreds of completely reusable "Lego block"–like commerce APIs that can be used to build innovative experiences for our customers. We must be able to adapt quickly to changes in consumer technology. Just 10 years ago, smartphones barely existed. Now they're crucial to our everyday lives. Microservices allows us to quickly adapt to changes in consumer technology. We can have a new app running in just a few days.

Microservices has been transformational to our business in many ways and we will continue to make deep investments as we transform to be the market leader.

— Jean-Jacques van Oosten
Chief Digital Officer, Rewe Group
October 2016

CHAPTER 1

A New Commerce Landscape

Changing Consumer Demands

We are entering a new era in commerce. Consumers demand to seamlessly transact anywhere, anytime, on any client. Every sale is the culmination of potentially dozens of interactions with a consumer. Today, smartphones alone influence 84% of millennials' purchases. Digital touchpoints influence 56% of all purchases. Selling something to an end consumer is far more complicated than it used to be, even just 10 years ago. Consumers are firmly in charge and expect to make purchases on their terms.

What do today's consumers want?

A Brand Experience—Not Simply a Transaction

Those engaged in commerce are surviving and thriving in today's era of commoditized goods by creating *experiences*, often through the use of content. Consumers want a story behind the product they're buying. A product is never just a product—it's a reflection of the consumer. It's a statement. Today's brands are successful because they are able to de-commoditize the products they sell. This requires the extensive use of content—text, video, audio, and so on.

Consistency of Experience Across Channels

Consumers no longer see divisions between channels (point of sale, web, mobile, kiosk, etc.). Consumers expect to see the same inventory levels, product assortment, pricing, and other aspects, regard-

less of how they interact with a brand. Whereas tailoring an experience to a channel is acceptable, offering a fragmented experience is not.

Value-Added Features

A primary driver of online shopping is the additional functionality that that it offers beyond that of a physical store. These features include a larger product assortment, ratings/reviews, more in-depth product descriptions, additional media (enhanced product photos/videos), related products, tie-ins to social media, and so on.

Convenience

Barely a hundred years ago, physical retail stores were the only way to make a purchase. Then, catalogs came of age. In the late 1990s, the Internet began to take off, and consumers could purchase through a website. Later, smartphones came of age when the iPhone was released in 2007. In the decade since then, the number of devices on the market has exploded, from smart watches to Internet-enabled TVs. Nearly every Internet-connected consumer electronic device hitting the market today offers an interface that consumers can use for shopping. New user interfaces are hitting the market weekly and successful brands must be able to offer their unique experience on every one of these new devices.

Retailers (and Everyone Else) Are Now Powered by Software

> Technology permeates every sector of the economy, even those not formally classified as high-tech. These days every company is a tech company.
>
> —The *New York Times*

The increased demands from consumers have forced retailers to turn into software companies who happen to be in the business of selling physical or virtual products. Every aspect of a retailer runs on software—from product placement on shelves, to the robots that power warehouses, to the app that runs on the latest Apple Watch.

Software not only saves money by improving efficiency, it can drive top-line growth by enabling marketers to build successful brands. Consumers want an *experience*—not simply to buy a commoditized

product. Marketers can use technology to form life-long bonds with end consumers by matching the right content to the right consumer.

Differentiation through software is driving retailers to build software from scratch rather than buy it prepackaged from a third-party software vendor. Differentiation in the marketplace is difficult to accomplish when everyone is using the same software. If software is the core of the business, it makes sense to make substantial investments in it to provide market-leading differentiation. It's no longer an IT cost that needs to be minimized.

The Status Quo Is Too Slow

Most enterprises with $100 million per year in online revenue release code to production too slowly. Releases often occur once each quarter and require an evening of downtime. Often, the entire team must come in over a weekend.

Enterprises are still learning how to reorient themselves around software. It wasn't too long ago that commerce was seen as an IT-only expense, out on the periphery of an organization.

Let's explore a few of the varied issues.

IT Is Seen as an Expense to be Minimized

Many enterprises still see IT as an expense—not as *the* business. Work is submitted to IT, as if it were an external system integrator, rather than an enabler of the business. If work is submitted to third-party system integrators, the lowest cost bid often wins out. Software and services are often centrally procured and an entire enterprise is forced to use the same technology stack regardless of fit. This culture of cost minimization comes from the days when IT was more on the periphery of business rather than the business itself.

Organization Structure

> Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.
>
> —Mel Conway (1968)

Conway's famous observation in his seminal paper has been so crucial to microservices that microservices is often called "Hacking Conway's Law."

Most IT organizations are set up to minimize cost through specialization (see Figure 1-1). Storage administrators are in their own group, Java developers are in their own group, and so on. This allows each person to be fairly efficient, but the system as a whole is slower.

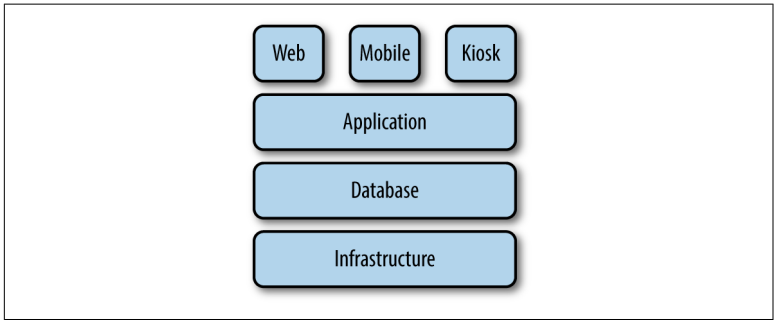


Figure 1-1. Typical horizontal-focused specialization within an enterprise

Each team has its own work location, process for receiving work (typically a ticketing system of some sort), service level agreements, process for assigning work to individuals, release cycles, and so on. This strict separation makes it difficult to make any changes that span multiple teams. For example, suppose that a Java developer receives a requirement to begin capturing a customer's shoe size during registration. In a typical enterprise, this would be incredibly difficult even though the work should take about two minutes for any competent developer to perform. Here's a non-exhaustive list of the serial steps required to perform this:

1. Java developer receives requirement
2. Java developer must use DBA's ticketing system to file a ticket
3. DBA team receives work order, prioritizes it, and assigns it
4. DBA adds column to database per work order
5. DBA updates ticket, requesting that the Java developer view that it was added correctly

6. Java developer logs in to database and finds that it was added correctly
7. Java developer updates ticket stating that the column was added correctly and that the change can be promoted
8. Java developer waits for next database build
9. Java developer updates the object relational mapping system to look for the new column in the database
10. Java developer updates the registration API to include birth date

These steps are exhausting even just to read, yet this is how even minor changes are implemented in enterprises. These steps don't even include the UI updates. This bureaucracy, due to horizontally specialized teams, is what leads to quarterly releases.

With teams so large and isolated, a corrosive culture of distrust develops. Rather than working together, teams are motivated to erect more bureaucracy (change requests, architecture review panels, change control boards, etc.) to cover themselves in the event of a problem.

Coupling

Today's enterprises are characterized by extreme coupling, both in terms of organization and architecture.

Let's begin with organization.

Enterprises cause extremely tight coupling between horizontal layers because they build teams of people who have only one focus. For example, each user interface (point of sale, web, mobile, kiosk, etc.) has its own team. Those UIs are tightly coupled to one or more applications, which are each owned by a separate team. Often, there's an integration team that glues together the different applications. Then, there's a database on which all teams are completely dependent. Infrastructure is managed by yet another team. Each team, no doubt, is good at what they do. But those barriers cause tight coupling between teams, which introduces communication overhead and causes delays.

It would be as if an auto repair shop had one person to order tires, another person to unscrew the lug nuts, another person to remove the old tire, another person to balance the new tire, another person to mount it, and one final person to screw on the lug nuts. Sure,

each of those six people are the best at what they do but the overhead of coordinating those activities across six people far outweighs any gains had by the efficiency improvement at each step. Yet this is how enterprises operate today. In the past, this was necessary because these layers all required extensive expertise. For example, networking required decades of experience and real competency. Now it's all software-based, as illustrated here:

```
$ docker network create frontend-network
```

To further complicate matters, enterprises encourage too much code sharing. Because IT is seen as a cost, and code is expensive to develop, many enterprises force development teams to reuse as much code as possible. For example, suppose that a team within an enterprise builds a new OAuth client that is forced onto the other teams within the entrprise as a form of cost savings. Every team that this library is forced on now has a firm dependency on the team that created the OAuth client. There is now a tight coupling between teams where one didn't exist before. A typical enterprise application could have hundreds of shared libraries, creating a web of dependencies. Over time, this complex labyrinth devolves into paralysis; everyone is afraid to touch anything because it could break the entire system.

Architecture introduces even more coupling. Enterprises have a handful of large, monolithic applications such as ERP, CRM, WMS, OMS, CMS, and so on. These large monolithic applications often expose many different endpoints, but those endpoints are often not independently consumable. The endpoints must be called in a specific order and fed specific data. That's why these monolithic applications are glued together by the use of *enterprise service buses*, with a lot of business logic residing in those buses. This tight coupling of large monolithic applications results in testing and releasing all monolithic applications together as an atomic unit. Changing one endpoint in one monolithic can have wide-ranging consequences across many of the other monolithic applications that might be consuming it.

Yet one more way of coupling is the practice of releasing only one version of an application to production at any time. Suppose that a company deploys version 3.2 of a monolithic commerce application to production. The website, iOS, Android, kiosk, and chatbot clients have all coded to version 3.2 of that application. What happens

when the company deploys version 4 of the commerce application? It's going to break all of the clients that have coded to version 3.2. With only one version of an application deployed, you must update your monolith and all clients at the same time, which is coupling at its most extreme.

The coupling introduced by organization structure and architecture choices has one major consequence—decreased speed.

Packaged Applications

Many of today's enterprise applications are large, monolithic packaged applications that are bought from a handful of large software vendors, deployed on-premises, and heavily customized. Many packaged commerce applications have millions of lines of customized code.

These applications are sold to thousands of customers. Those thousands of customers each write millions of lines of customized code on top of the application. As the number of customers increases, the vendors that sell the software are increasingly unable to make changes because of all the trouble it would create. The more successful the product, the slower it evolves. It gets frozen in time.

(Real) Omnichannel Is the Future

Omnichannel is the future of retail. Today's top leaders have mastered it, but the vast majority of retailers have yet to adopt it.

To end consumers, omnichannel means having a consistent experience with a brand, regardless of how they interact with it. Whether through a website, mobile device, wearable device, or in store, the experience is the same and integrated.

> The web is dead. Long live the Internet.
>
> —Chris Anderson and Michael Wolff , August 17 , 2010

This is why many in the industry have dropped "e" from "eCommerce" to reflect that it's not something different. Consumers should be able to buy online and pick up or return in-store, browse in-store and buy online, have access to the same promotions, and so on. If channels are offering different data (subset of products, different prices, etc.) it should be because the experience is being optimized for each channel or there are opportunities to price discriminate.

To technologists, omnichannel means having one backend system of record for each bit of functionality (pricing, promotions, products, inventory, etc.), with UIs being more or less disposable. Developers of UIs get a large catalog of clearly defined APIs (often HTTP plus REST) that can be composed into a single application, as demonstrated in Figure 1-2.

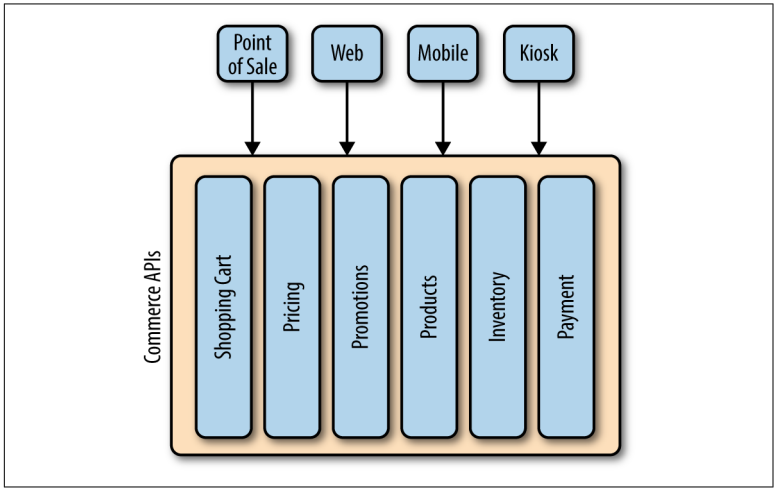


Figure 1-2. True omnichannel—single systems of record for each business function; disposable UIs

Again, the experience per channel can vary, but the variations are deliberate rather than as a result of IT fragmentation.

Most of today's enterprise commerce platforms are sidecars on top of the old in-store retail platforms. There might be additional sidecars on top of the commerce platforms for other channels, such as mobile. Each of these systems acts as its own mini system of record, with heavy integration back to the old in-store retail platform, as shown in Figure 1-3.

Each system has its own view of pricing, promotions, products, inventory, and so on, which might or might not ever be reconciled with the eventual system of record. For example, many retailers have web-only pricing, promotions, products, inventory, and so forth.

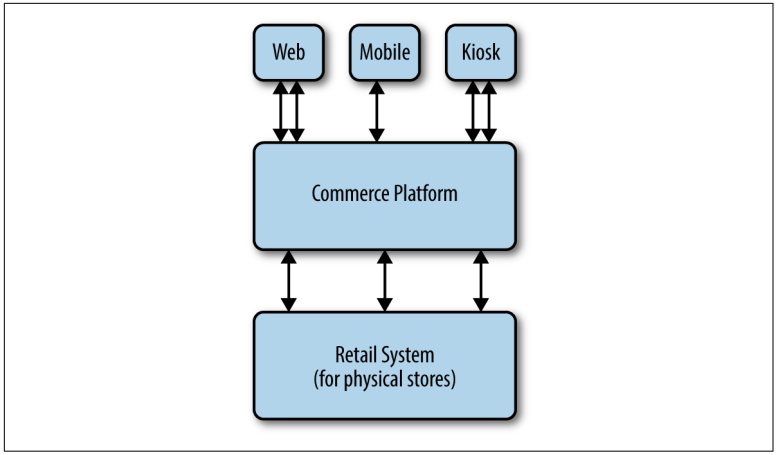


Figure 1-3. Typical commerce application—large monolithic applications tightly coupled with legacy backend systems of record (ERP, CRM, etc.)

This approach worked adequately when there were just physical stores and a website. But now, there are dozens if not hundreds of channels. The old approach just doesn't scale anymore. As a thought experiment, could you deploy Amazon.com as a single monolithic application as one large EAR file? No. It's ludicrous to think about it. But retailers trying to unseat Amazon.com regularly deploy large monolithic applications, expecting to be able to release new functionality with the same agility as those who have implemented omnichannel from the very beginning. Amazon.com has famously used microservices since 2006. Today, it has thousands of individual microservices that serve as building blocks for dozens of UIs.

Fortunately, there is a better way.…

CHAPTER 2

Introducing Microservices

Origins of Microservices

Although the term "Microservices" first rose to prominence in 2013, the concepts have been with us for decades.

> 1. Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new features.
> 2. Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.
> 3. Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.
>
> —Doug McIlroy, one of the founders of Unix and inventor of the Unix pipe, 1978

The software industry has long looked for ways to break up large monolithic applications into smaller more modular pieces with the goal of reducing complexity. From Unix pipes to dynamic-link libraries (DLLs) to object-oriented programming to service-oriented architecture (SOA), there have been many attempts.

It is only due to advances in computer science theory, organizational theory, software development methodology, and infrastructure that

microservices has emerged as a credible alternative to building applications.

So what are microservices?

Introducing Microservices

Microservices are individual pieces of business functionality that are independently developed, deployed, and managed by a small team of people from different disciplines (see Figure 2-1).

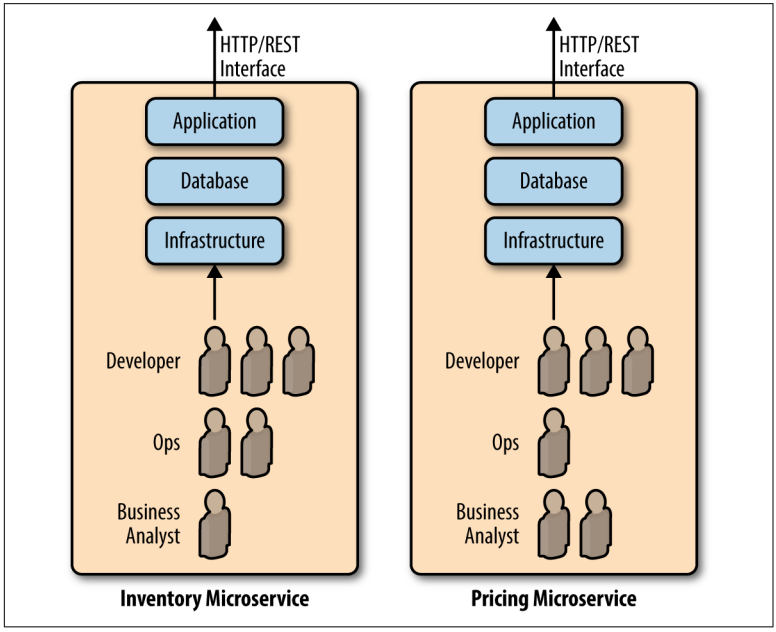


Figure 2-1. Typical microservice team composition

Inner versus Outer Complexity

Fundamentally, microservices shift complexity outward, trading external complexity for inner simplicity. "Inner" is what's in a single microservice and how it's packaged. It includes the runtime, the business logic, coding to the datastore, circuit breakers, management of application state, and so on—basically, anything for which an individual developer is responsible. "Outer" is everything outside of the individual microservice, including how instances of the application are deployed, how individual instances are discovered/routed to, load balancers, messaging, networking—basically anything for

which an ops person outside of an individual microservice team is responsible.

Monolithic applications have always been difficult to build, especially as they increase in size. But monoliths are relatively easy to deploy and manage. With microservices, each microservice is exceedingly easy to build and manage because the application is small and limited in scope. Large monolithic applications can have tens of millions of lines of code whereas a microservice may only have a few thousand. Some of the more extreme microservices practitioners say that a microservice should be so small that it can be completely rewritten over a weekend. However, although each microservice is easier to build, deploy, and manage, the outer complexity becomes more difficult.

> There is no single development, in either technology or management technique, which by itself promises even one order of magnitude [tenfold] improvement within a decade in productivity, in reliability, in simplicity.
>
> —Fred Brooks, 1986

Microservices is worth the tradeoff from inner to outer complexity, especially for commerce, because it dramatically shortens the time to market for new individual features. Because each team is isolated, a requirement can often be implemented and deployed to production within the course of an hour. This is possible because the scope of each team's work is limited to its own microservice. Each microservice stays small and simple, with each team having full control over it. Monolithic applications, on the other hand, grow in complexity with each passing year as they get larger. Over time, this dramatically slows down development due to the complexity of the monolithic applications.

As monolithic applications grow in size, the time required to implement a new feature increases, to the point where releases occur every quarter or six months. The large monolithic applications that run many banks, airlines, and retailers are sometimes deployed once a year or once every two years. Over time, the inability to deploy new features puts organizations at a severe competitive disadvantage relative to more nimble organizations that are able to release weekly, daily or even hourly, as is illustrated in Figure 2-2.

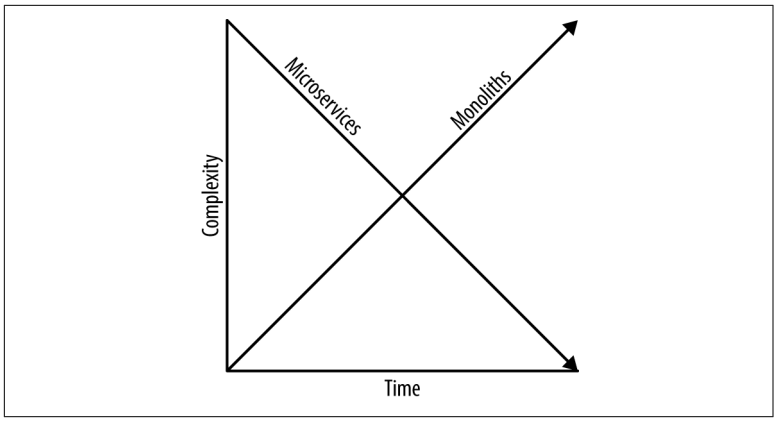


Figure 2-2. Microservices offers less complexity over time

Defining Microservices

Now that we've reviewed some of the high-level characteristics of microservices, let's look at what the defining characteristics actually are:

Single purpose
: Do one thing and do it well.

Encapsulation
: Each microservice owns its own data. Interaction with the world is through well-defined APIs (often, but not always, HTTP REST).

Ownership
: A single team of 2 to 15 (7, plus or minus 2, is the standard) people develop, deploy and manage a single microservice through its lifecycle.

Autonomy
: Each team is able to build and deploy its own microservice at any time for any reason, without having to coordinate with anyone else. Each team also has a lot of freedom in making its own implementation decisions.

Multiple versions
: Multiple versions of each microservice can exist in the same environment at the same time.

Choreography
: Actions across multiple microservices are managed in a distributed fashion, with each endpoint being intelligent enough to know its inputs and outputs. There is not a top-down workflow that manages a transaction across multiple microservice boundaries.

Eventual consistency
: Any given bit of data is, generally speaking, eventually consistent.

It's tempting to think of microservices versus monoliths, but there's a lot of gray area between the two. There are very few "true" adherents to all of the principles of microservices. What is outlined here is more of a textbook definition. Feel free to implement only what works for you and your organization. Don't get too dogmatic.

Let's explore each of these in more depth.

Single purpose

A large monolithic application can have tens of millions of lines of code and perform hundreds of individual business functions. For example, one application might contain code to handle products, inventory, prices, promotions, shopping carts, orders, profiles, and so on. Microservices, on the other hand, each perform exactly one business function. Going back to the founding principles of Unix, *Write programs that do one thing and do it well*, is a defining characteristic of microservices. Doing one thing well allows the teams to stay focused and for the complexity to stay at a minimum.

It is only natural that applications accrue more and more functionality over time. What begins as a 2,000-line microservice can evolve to one comprising more than 10,000 lines as a team builds competency and the business evolves. The size of the codebase isn't as important as the size of the team responsible for that microservice. Because many of the benefits of microservices come from working together as a small tight-knit team (2 to 15 people). If the number of people working on a microservice exceeds 15, that microservice is probably trying to do too many things and should be broken up.

In addition to team size, another quick test is to look at the name of a microservice. The name of a microservice should precisely describe what it does. A microservice should be named "Pricing" or "Inventory"—not "PricingAndInventory."

Encapsulation

Each microservice should *exclusively* own its data. Every microservice should have its own datastore, cache store, storage volume, and so forth (see Figure 2-3). No other microservice or system should *ever* bypass a microservice's API and write directly to a datastore, cache layer, file system, or anything else. A microservices' only interaction with the world should be through a clearly defined API.

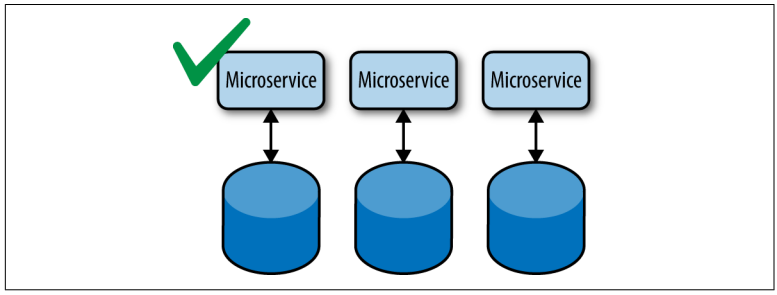


Figure 2-3. Each microservice should exclusively own its data

Again, the goal of microservices is to reduce coupling. If a microservice owns its own data, there is no coupling. If two or more microservices are reading/writing the same data (Figure 2-4), tight coupling is introduced where there was none before.

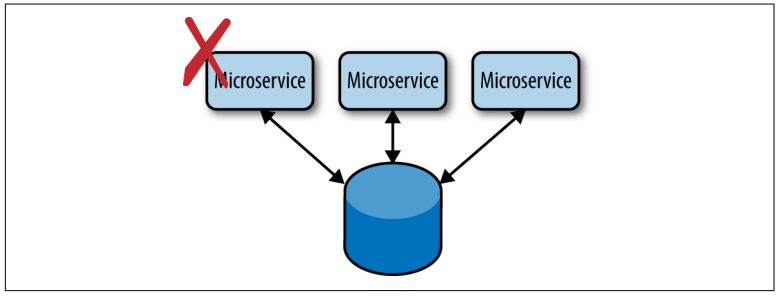


Figure 2-4. Don't share data across microservices; use application-level APIs to exchange data

Although it is preferable for each microservice to have its own datastore, cache store, storage volume, or other data storage mechanism, it is not necessary that each microservice provision its own single-tenant instance of those things. For example, it might make more sense to provision one large database, cache store, storage volume or other system to which all microservices can write. What matters is that there is a firm partition between each microservice's data: each microservice must exclusively own its own data. For example, it might make sense to have one large database with 100 schemas rather than 100 databases with one schema. In other words, feel free to share resources but not data.

The downside of sharing is coupling. The availability of your microservice is now dependent on the availability of a database that someone else manages. The team that administers the shared database might need to bring it down for scheduled maintenance.

Ownership

A typical enterprise-level commerce application has hundreds of staff who work on it. For example, it would not be uncommon to have 100 backend developers building a large monolithic application. The problem with this model is that staff members don't feel like they own anything. A single developer will be contributing just one percent of the codebase. This makes it difficult for any single developer to feel a sense of ownership.

In economic terms, lack of ownership is known as a *Tragedy of the Commons* problem. Individuals acting in their own self-interest (e.g., farmers grazing their cattle) almost inevitably end up making the common resource (e.g., public lands) less better off (by overgrazing). It's the exact same problem in a large monolithic application—hundreds of staff acting in their own self-interest end up making the monolithic application more complicated and add more technical debt. Everyone must deal with complexity and technical debt, not just the individual who created it.

Microservices works in large part due to ownership. A small team of between 2 to 15 people develop, deploy and manage a single microservice through its entire lifecycle. This team truly *owns* the microservice. Ownership brings an entirely different mentality. Owners care because they have a long-term vested interest in making their microservice succeed. The same cannot be said about large mono-

lithic applications with which hundreds of people are involved. Suppose that a team has five members—three developers, one ops person, and one business analyst. In this case, any given developer contributes 33% of the code. Every person on that team is making a substantial contribution and that contribution can be easily recognized. If a microservice is up 100 percent of the time and works perfectly, that team is able to take credit. Similarly, if a microservice is not successful, it's easy to assign responsibility.

On an individual level, microservices brings out the best in people because they can't hide in a larger team. The performance of individuals in any team take the shape of a standard bell curve. The top performers like microservices because they can have an outsized influence over a microservice. Microservices attracts high performers because it allows them to have more responsibility.

A team responsible for a microservice should be composed of 2 or more people but no more than 15. The best team size is seven, plus or minus two people. There's some interesting research and anecdotes that impact team size.

2 people
: Insufficient skills to draw from. Two people isn't quite a "team." It's more like a marriage—which can bring its own challenges.

3 people
: Unstable grouping because one person is often left out, or one person controls the other two.

4 people
: Devolves into two pairs rather than a cohesive team of four people.

5 to 9 people
: Feeling of a "team" really begins. Enough skills to draw from, large enough to avoid the instability of smaller teams, small enough that everyone is heard.

10 to 15 people
: Team becomes too large. Not everybody can be heard, smaller groups might splinter off.

Each team should have a wide mix of skills, including development, operations, datastore administration, security, project management, requirements analysis, and so on. Often, teams are composed of one

business analyst, one or two operations people, and a handful of developers. Individuals within a team often perform each other's work. For example, if the sole operations person in a team of five is out on vacation, one of the developers will assume operations responsibilities.

An important dynamic in team sizes is trust. Trust is required for real work to get done. When there's a lack of trust within a team, individuals compensate by protecting themselves. This protection often takes the form of excessive paperwork (i.e., change requests, production readiness reviews, architecture review boards) and documentation. Even though this can diffuse responsibility in the event of a problem, this behavior is counterproductive to the goals of the organization as a whole. Smaller, tight-knit, trusting teams don't have this problem.

Another way to look at team size is in terms of communication pathways. Communication pathways grow exponentially as team members are added, and can slow down progress. In his 1975 book, *The Mythical Man Month*, Fred Brooks put forth an equation that calculates the number of communication pathways in a team. Within a team, the number of communication pathways is defined as $[n^*(n-1)]/2$, where n = team size. A team of two has only one communication pathways but a team of 20 has 190 pathways, as illustrated in Figure 2-5. Fewer pathways makes for faster development and greater employee satisfaction.

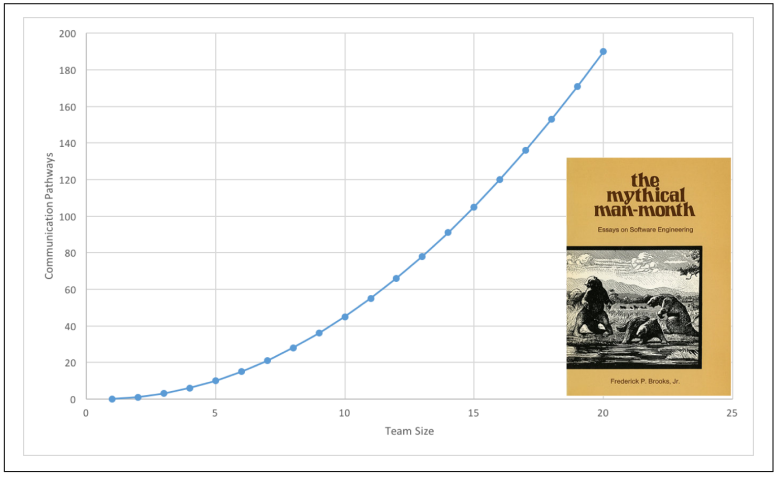


Figure 2-5. Communication pathways within a team

Autonomy

Ownership cannot exist without autonomy. Autonomy can mean many things in the context of microservices.

Each team should be able to select the technology best suited to solve a particular business problem—programming language, runtime, datastore, and other such considerations. The tech stack for a microservice that handles product image resizing is going to look very different from a tech stack for a shopping cart microservice. Each team should be able to select the tech stack that works best for its particular needs and be held accountable for that decision. Each microservice should expose only an API to the world; thus the implementation details shouldn't matter all that much. That being said, it makes sense for enterprises to require each team to select from a menu of options. For example, the programming language can be Java, Scala, or Node.js. Datastore options could be MongoDB, Cassandra or MariaDB. What matters is that each team is able to select the *type* (Relational, Document, Key/Value, etc.) of product that works best, but not necessarily the product (MongoDB, Cassandra, MariaDB) itself. Teams should be required to standardize on outer implementation details, like API protocol/format, messaging, logging, alerting, and so on. But the technology used internally should be largely up to each team.

Along with technology selection, each team should be able to make architecture and implementation decisions so long as those decisions aren't visible outside of the microservice. There should *never* be an enterprise-wide architecture review board that approves the architecture of each microservice. Code reviews should be performed by developers within the team—not by someone from the outside. For better or worse, the implementation decisions belong to the team that owns and is responsible for the microservice. If a microservice is not performing to expectation, engage a new team to build it. Remember, each microservice should be fairly small because it is solving a very granular business problem.

Each team should be able to build and run its own microservice in complete isolation, without depending on another team. One team shouldn't build a library that another team consumes. One microservice shouldn't need to call out to another microservice as it starts up or runs. Each microservice is an *independent* application that is built, deployed and managed in isolation.

Each team should also be able to publish a new version of a microservice live at any time, for any reason. If a team has version 2.4 of a microservice deployed to production, that team should be able to release version 2.5 and even 3.0.

Multiple versions

Another defining characteristic of microservices is the ability (but not obligation) to deploy more than one version of a microservice to the same environment at the same time. For example, versions 2.2, 2.3, 2.4, and 3.0 of the pricing microservice may be live in production all at the same time. All versions can be serving traffic. Clients (e.g., point of sale, web, mobile, and kiosk) and other microservices can request a specific version of a microservice when making an HTTP request. This is often done through a URL (e.g., */inventory/2/* or */inventory?version=2*). This is great for releasing minimum viable products (MVPs). Get the first release out to market as v1 and get people to use it. Later, release v2 as your more complete product. You're not "locked in" in perpetuity, as you often are with monoliths.

Monolithic applications, on the other hand, don't have this level of flexibility. Often, only one version of a monolithic application is live in an environment at any given time. Version 2.2 of a monolithic application is completely stopped and then version 2.3 is deployed and traffic is turned back on.

The ability to support multiple clients is a strength of microservices and is an enabler of real omnichannel commerce. Clients and other microservices can code to a specific major API version. Suppose that the ecosystem codes to a major version of 1. The team responsible for that microservice can then deploy versions 1.1, 1.2, 1.3, and beyond over time to fix bugs and implement new features that don't break the published API. Later, that team can publish version 2, which breaks API compatibility with version 1, as shown in Figure 2-6.

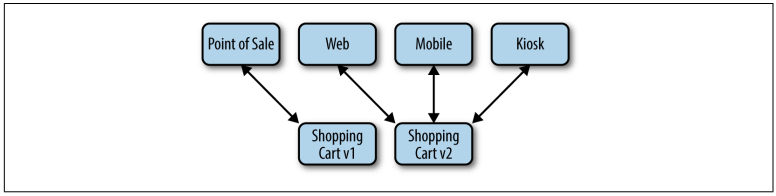


Figure 2-6. Multiple versions of the same service enable clients to evolve independently

Clients and other microservices can be notified when there's a new version available but they don't have to use it. Versions 1 and 2 coexist. A year after the publication of version 2 when nobody is using version 1 any longer, version 1 can be removed entirely.

Monolithic applications suffer from the problem of forcing all clients to use the same version of the application. Any time there's an API change that breaks compatibility, all clients must be updated. When the only client used to be a website, this was just fine. When it was mobile and web, it became more difficult. But in today's omnichannel world, there could be dozens of clients, each with its own release cycle. It is not possible to get dozens of clients to push new versions live at the same time. Each client must code to its own version, with migrations to later versions happening as each client goes through its own release cycles. This approach allows each microservice to innovate quickly, without regard to clients. This lack of coupling is part of what makes microservices so fast.

The major challenge with supporting multiple versions of a microservice concurrently is that all versions of a microservice in an environment must read/write to the same backing datastore, cache store, storage volume, or other storage schema. A microservice's data remains consistent, even though the code might change. For example, version 1.2 of the product microservice might write a new product to its backing database. A second later, version 2.2 of the same microservice might retrieve that same product, and vice versa. It can't break. Similarly, an incoming message might be serviced by any version of a microservice.

Evolvable APIs

It is possible to offer a single *evolvable* API rather than multiple versions of one API. Having one API that evolves solves the problem mentioned in the previous paragraph, but comes at the cost of reducing the level of changes you can make. For example, you can't just rewrite your pricing API, because you might have a dozen clients that depend on the original implementation. If you don't need to radically rewrite your APIs, having a single evolvable API is preferable to having multiple versions of the same API.

Support for more than one version adds a new dimension to development and deployment. That's why microservices is fundamentally

seen as a tradeoff from inner to outer complexity. But the benefit of being able to innovate quickly without being tightly coupled to clients more than outweighs the cost.

Choreography

Centralization is a defining characteristic of traditional monolithic applications. There is one centralized monolithic commerce application. Often, there is a top-down "orchestrator" system of some sort that coordinates business processes across multiple applications. For example, a product recall is issued, which triggers a product recall workflow. Figure 2-7 presents a very simple overview of what that workflow would look like.

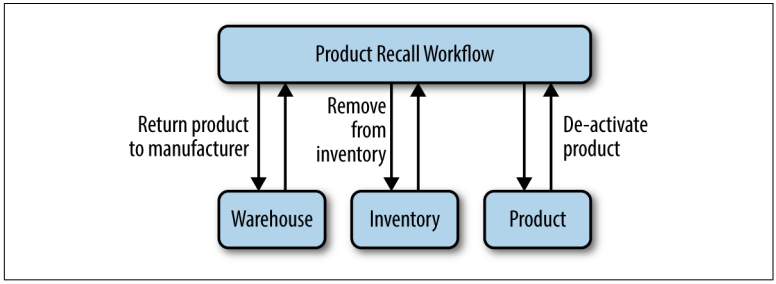


Figure 2-7. Traditional top-down, centralized, coupled workflow

This model introduces tight coupling. The workflow must call APIs across many different applications, with the output of one being fed into the next. This introduces tight coupling between applications. To change the warehouse's APIs now requires an update and retesting of the entire workflow. To properly test the workflow, all other applications called by that workflow need to be available. Even simple API changes can require retesting an enterprise's entire backend. This leads to quarterly releases to production because it's just not possible to perform this testing more frequently than that. This centralized orchestrator is also a single point of failure.

Microservices favors *choreography* rather than *orchestration*. Rather than each application being told what to do by a centralized coordinator, each application has enough intelligence to know what to do on its own.

Microservices is often referred to as having *smart endpoints and dumb pipes*. Endpoints are the individual REST APIs that are used to interface with individual microservices. Pipes are just plain HTTP. Requests from system to system are rarely if ever modified while en route.

SOA is often referred to as having *dumb endpoints and smart pipes*. Endpoints are exposed methods in large monolithic applications that cannot be called independently. Pipes are often routed through Enterprise Service Bus–type layers, which often apply vast amounts of business logic in order to glue together disparate monolithic applications.

In this model, everything (for example, a product recall) is modeled as an event and published to an event bus. Interested microservices can subscribe to receive that event. But there isn't a centralized system instructing each microservice which events to consume. The authors of the inventory microservice will independently come to the conclusion that they need to know if a product is recalled, because that would affect inventory.

Going back to our product recall example, a choreography-based approach would look like Figure 2-8.

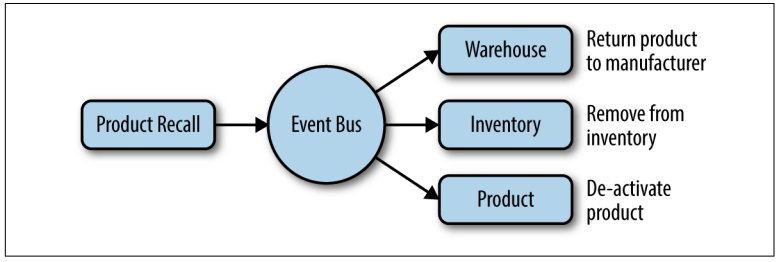


Figure 2-8. Loosely coupled, distributed, bottom-up choreography

Applications are *loosely* coupled, which is the key strength of microservices. Figure 2-9 depicts how events look from the standpoint of an individual microservice.

Each microservice subscribes to events, performs some action, and publishes more events for anyone who cares to subscribe to them. Each microservice is unaware of which application produces events,

or which application consumes the events it produces. It is a decoupled architecture meant for distributed systems.

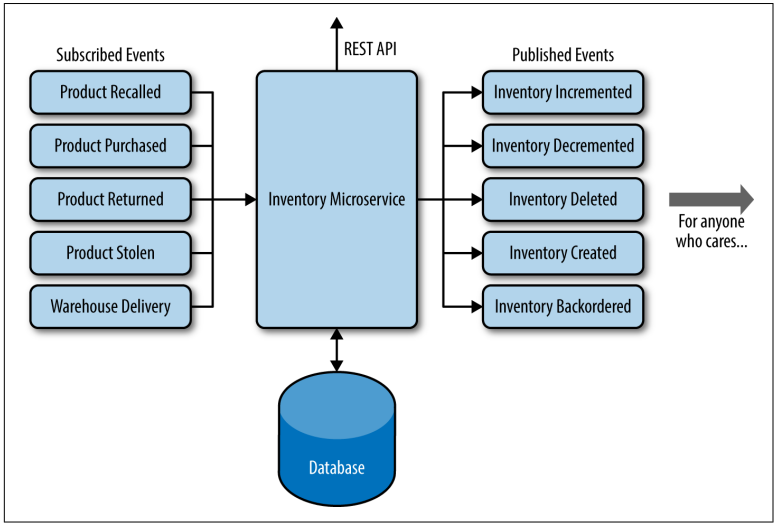


Figure 2-9. Eventing-based architecture

Eventual Consistency

Monolithic applications are strongly consistent, in that any data written to a datastore is immediately visible to the entire application because there's only one application and one datastore.

Microservices are distributed by their very nature. Rather than one large application, there might be dozens, hundreds or thousands of individual microservices, each with its own datastore. Across the entire system there might be dozens of copies of common objects, like a product. For example, the product, product catalog, and search microservices might each have a copy of a given product. In a monolithic application, there would be exactly one copy of each product. This isn't cause to panic. Instead, consider what this duplication buys you—business agility.

Adopting microservices requires embracing eventual consistency. Not all data will be completely up to date. It doesn't need to be. For example, most of the world's ATMs are not strongly consistent. Most ATMs will still dispense money if they lose connectivity back to the centralized bank. Banks favor availability over consistency for business reasons—an ATM that's available makes money. Much of the

world favors availability over consistency. That most data in commerce systems is consistent is more a byproduct of the prevailing monolithic architecture and centralized databases rather than a deliberate choice.

Suppose that an end client needs to render a category page with 10 products. In a monolithic architecture, it would look something like Figure 2-10.

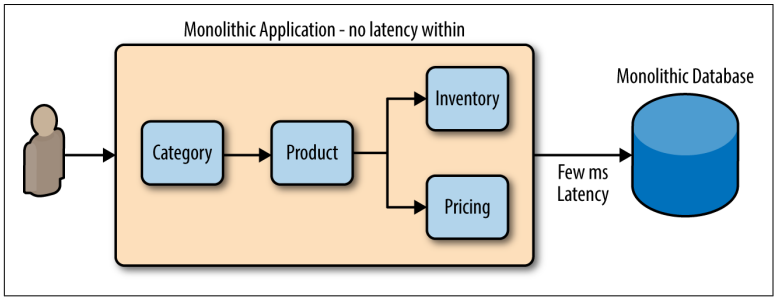


Figure 2-10. Viewing a category page with a monolith

The category method calls the product function 10 times. The product function calls the inventory and pricing functions. All of this code runs inside a single process. The application is within a few milliseconds of the database.

Now, let's take a look at Figure 2-11 to see how this would look if you directly mapped it back to a microservices-style architecture.

With tens of milliseconds separating each microservice, this clearly wouldn't work. Synchronous calls between microservices should be exceptionally rare, if entirely non-existent.

Instead, the proper way to do this with microservices is to build a product catalog microservice that is the only microservice accessed to retrieve product data. The category, product, inventory, pricing, and all other related microservices each act as the system of record for their own individual pieces of data, but they publish that data internally in the form of events. The product catalog microservice will pull down those events and update its own datastore, as demonstrated in Figure 2-12.

Yes, this results in duplication. There are now many copies of each object. But consider this: how much does storage cost today? It is minuscule compared to the benefits of faster innovation. The over-

arching goal of microservices is to eliminate coupling. Sharing data introduces tight coupling.

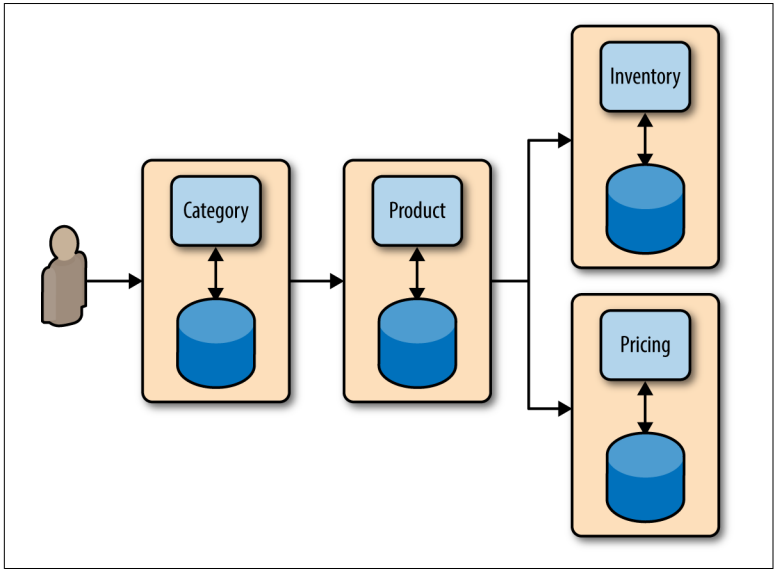


Figure 2-11. Microservices will not scale if implemented like a distributed monolith

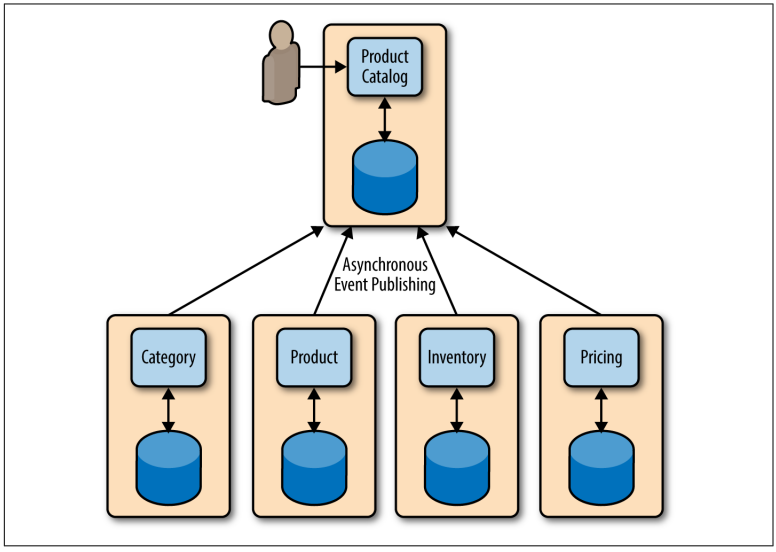


Figure 2-12. Event-based architecture for preventing synchronous calls between microservices

Sometimes data needs to be strongly consistent. For example, the shopping cart microservice should probably query the inventory microservice to ensure that there's inventory available for each product in the shopping cart. It would be a terrible experience if a customer made it to the final stage of checkout before being told a product was unavailable. There are two basic approaches to strong consistency:

- Have the clients call the system of record microservice directly. For example, the inventory microservice is the system of record for inventory. Although the product catalog can return a cached view of inventory, it might not be up-to-date enough for the shopping cart. When the shopping cart screen is shown, the client can query the product catalog microservice for product details and the inventory microservice for up-to-date inventory levels.
- Have the microservices make synchronous calls to each other. For example, */ProductCatalog?productId=123456&forceInventorySync=true* would force the product catalog microservice to synchronously query the inventory microservice for up-to-date inventory. This approach is generally frowned upon because it harms performance and availability.

There are very few instances for which data must be truly consistent. Think very hard about introducing this as a requirement, because the consequences (coupling, performance, availability) are so damaging. Don't fall into the trap of implementing microservices as a monolith.

Advantages of Microservices

Microservices offers many advantages. Let's explore some of them.

Faster Time to Market

Faster time to market of new features is the most important benefit of microservices. The overall time it takes to get to market initially might be longer due to the additional up-front complexity introduced by microservices. But after it's live, each team can independently innovate and release very quickly (Figure 2-13).

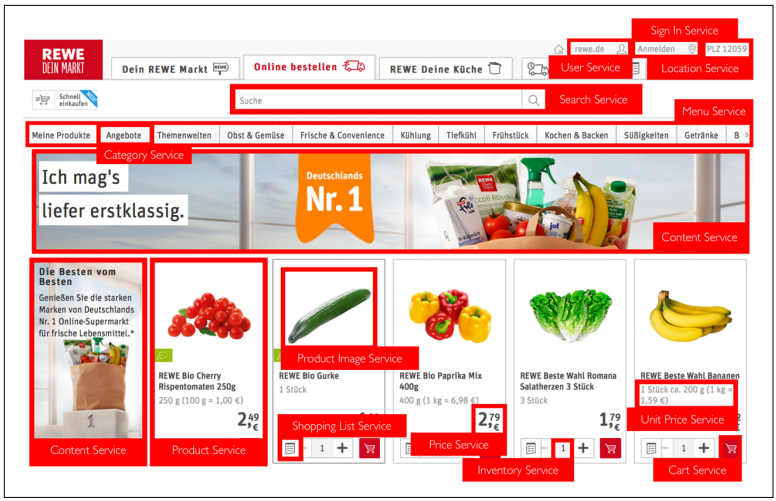


Figure 2-13. Individual teams can independently innovate

Getting a feature to market quickly can have substantial top-line benefits to your business. If releases to production occur quarterly and it takes four iterations to get a feature right, it would therefore take a year to get it right. But if releases occur daily, it takes only four days to get it right. In addition to quick iteration, microservices enables failures to occur faster. Sometimes, features just don't work out as expected. Rather than wait a quarter for the feature to be pulled, it can be pulled immediately.

Microservices are so fast because they eliminate dependencies. With microservices, you don't have to deal with the following:

- Architecture approval from centralized architecture review committee
- Release approval from management
- Horizontal dependencies—for example, no waiting on DBAs to change the shared database, no waiting on ops to provision compute
- Coordinating a release with another team (vertical dependencies)—for example, the inventory microservice team will never need to release alongside the product catalog microservice team
- Approval from QA, security, and so on—each team performs these functions

The responsibility for these centralized functions is pushed down to each small team, which has the skills required to be self-sufficient. Each team simply cares about its inputs (typically events and API calls) and its outputs (typically events and APIs). Eliminating dependencies is what dramatically increases development velocity.

True Omnichannel Commerce

Fully adopting microservices means having individual APIs that are the sole source of truth for granular business functions. The user interfaces on top of those APIs become essentially disposable. Next time Apple or Facebook introduce a new type of application, you can easily build it without having to build a whole backend. New user interfaces can be built in days.

Better and Less Complex Code

Microservices tends to produce much better code because a single microservice performs exactly one business function. Large monolithic applications can have tens of millions of lines of code, whereas a microservice might have only a few thousand. Naturally, a small codebase tends to be better than a much larger codebase.

Code is also better because a small team owns it and is responsible for it. The codebase is a reflection of that small team. Developers, like all people, want to look good to their peers and superiors.

Finally, each microservice team has a strong incentive to write quality code. With a traditional monolithic application, the team that writes the code is unlikely to ever get a call in the middle of the night if something isn't working. That's the role of ops. But with a microservice, each small team is also responsible for production support. It's easy to check in shoddy code that someone else must fix—usually at the most inconvenient of times. It's not easy to do that to the person you sit next to and go to lunch with every day.

Accountability

Each small team owns a single microservice from its inception to retirement. That team has great freedom to make architecture, implementation, and technology decisions. All of that freedom means each team and each member of that team is accountable. If a team picks a hot new open source project but it fizzles out in six

months and has to be replaced, that team is responsible for fixing it. Nobody can point fingers. Conversely, if a team's microservice is available 100 percent of the time and has a zero percent error rate, that team can take sole credit for having made good choices.

Because each microservice's interface with the world is an API, it's fairly easy to quantitatively measure its availability, error rate, and performance over time. It's also easy to measure a team's throughput in terms of story points, features, and so forth. It's one small team writing one small application, solving one business problem, exposing (usually) one API.

Microservices has the unintentional side effect of raising the quality of employees. In a large team, the worst members can simply hide. Maybe they write paperwork, approve change requests, or have some other administrative task. Maybe someone else rewrites their code. People can hide in a large team. But in a small team, poorer-performing members can't hide. If there are three developers in a team and one isn't carrying his weight, it will be very apparent to the other two developers and the rest of the members of that small team. Underperformers voluntarily or involuntarily leave. Top performers will be attracted to a culture of freedom and accountability.

Enhanced Domain Expertise

Microservices requires that business functions be granularly split into separate services. In a commerce system, you'll have microservices for promotions, pricing, product catalog, inventory, shopping carts, orders, and other components. Each small team owns a single microservice from its inception to retirement.

Each team often includes one or two business analysts, product managers, or someone else in a less-technical role whose responsibility it is to look after features and functionality. Those people are able to develop very deep domain-specific expertise because they look after just one small thing. For example, the business analyst responsible for promotions is going to be the company's expert on promotions. With monolithic applications, business analysts tend to be in a large pool and are periodically assigned to different functional areas. With microservices, it's easy to develop very deep business expertise in one area.

The technical members are also able to develop a very deep technical expertise in the technology and algorithms required to support a

business function. For example, the team building the inventory microservice are able to develop extensive expertise with distributed locking.

Easier Outsourcing

It's difficult to outsource roles such as development, operations, and QA with traditional monolithic applications because everything is so tightly coupled. For example, it might take only a few days for a third-party system integrator to build a new payment module, but it will take weeks to set up development environments, learn the process to make changes to the shared database, learn the process for deploying code, and so on.

Microservices makes it very easy to outsource. Instead of a payment module, a system integrator could be tasked with creating a payment microservice, which exposes a clean API. That system integrator could then build the payment microservice and host it themselves as a service or hand it over for hosting alongside the other microservices. By outsourcing in this manner, dozens or even hundreds of microservices could be developed in parallel by different vendors, without much interaction between the different teams.

Rather than hiring a system integrator, APIs can be bought from third-party software vendors. For example, there are vendors that offer payment APIs as a service. Or product recommendations as a service. These vendors can build highly differentiated features as a service that can simply be consumed as a service.

Security

Security is most often an afterthought with large monolithic applications. Inevitably, the application is developed and secured later, often just in the UI. One application thread, if compromised, can call any function in the entire application. Even if proper security is implemented, there's no real way to enforce that developers use it. Many developers don't implement it because the code is complicated and messy.

Microservices is very different. Each microservice exposes a small granular bit of business functionality as an API (or two). It's easy to see exactly who or what is calling each API. Rather than dealing with security at the application layer, you can instead use an API gateway or API load balancer. These products allow you to define

such things as users, roles, and organizations. Calls from one API to another pass through the API gateway or API load balancer. Each request is evaluated against a ruleset to see if it can be passed to the endpoint. For example, user Jenn in marketing is only allowed to make HTTP GET requests to the Customers microservice, but the Orders microservice is allowed to make HTTP POST and GET requests. You can implement a default deny-all policy in the API gateway or API load balancer and force all requests to pass through it.

Microservices is also better from a security standpoint because each microservice is independently deployed, often to its own private network within a public cloud. If an attacker manages to access one microservice, he can't easily get to other microservices. This is called *bulkheading*. If a monolithic application is breached, the attacker will have access to the entire datastore and all application code. Much more damage can be done.

The Disadvantages of Microservices

Although microservices certainly has advantages, it has many disadvantages, too. Remember, the goal of microservices is to speed the delivery of new features. It is not about reducing costs. Getting new features to market quickly will far more than offset the potentially higher development costs of microservices.

Outer Complexity Is More Difficult

Microservices is often seen as sacrificing inner for outer complexity. Monolithic applications are complex on the inside, whereas each microservice is simple on the inside. The outer complexity of monolithic applications is simple, whereas the outer complexity of microservices is tricky. This tradeoff allows for each microservice team to build and deploy new features very quickly. But it comes at the cost of having to manage interactions between microservices.

Examples of outer complexity introduced by microservices include the following:

Data synchronization
: Synchronously and asynchronously copying data between microservices.

Security
: Who/what can call each endpoint? What data can be retrieved? What actions can be performed?

Discovery
: What microservices are available? Which messages does a particular microservice need to consume?

Versioning
: How is a particular API or implementation version of a given API retrieved?

Data staleness
: Which microservice is the real system of record of a given piece of data?

Debugging is more difficult, too. What if there's some weird interaction between version 13.2 of the Shopping Cart microservice and version 19.1 of the Inventory microservice? Remember that each microservice can have many dozens of versions live in the same environment at any time. It is impossible to test out all of the interactions. And it would be very much against the spirit of microservices to centrally test all of the interactions between microservices.

Organizational Maturity

Organizations must have a strong structure, culture, and technical competency. Let's explore each.

As we've already discussed, an organization's structure dictates how it produces software. Centralized organizations oriented around layers produce layered monolithic applications. Even simple changes require extensive coordination across those layers, which adds time. The proper structure for an organization that wants to create microservices is to reorganize around products. Rather than a VP of operations, there should instead be a VP of product catalog, with all of the product catalog–related microservices rolling up to her. It's a fundamental change in thinking, but one that will ultimately produce better software. DevOps, which we'll discuss shortly, is one step toward tearing down the organizational boundaries between development and operations. If that transition went well, the blurring of team boundaries won't be too foreign.

From a culture standpoint, microservices requires that organizations get out of the mindset that IT is simply a cost that must be minimized. Although there are efficiencies to having centralized teams manage each layer (development, operations, infrastructure, etc.), those efficiences come at the cost of coupling, which significantly harms development velocity. The culture of the organization needs to value development velocity and reactiveness to business over all else.

Beyond structure and culture, an organization needs to have a strong understanding of Agile software development, DevOps, and the cloud. Most microservices teams use some form of Agile to manage the development of each microservice. It's difficult to go from waterfall to Agile. DevOps is a practice by which development and operations (hence, "DevOps") work collaboratively or even interchangeably to support the entire lifecycle of an application. Each microservice team is responsible for all development and operations; thus, DevOps is the operating model. Finally, cloud is a requirement for microservices. Most microservices are deployed to the cloud. But, more important, the cloud is *distributed* by its very nature and having expertise around distributed computing makes the transition to microservices easier because everything is distributed in microservices.

Duplication

A goal of microservices is not to reduce costs. Even though that can happen over time, cost savings through consolidation and standardization is not a benefit of microservice. In a traditional enterprise, the CIO buys one database, one application server, and so on, and forces everyone to use it. Many will go a step farther and build one large database, one large cluster of application servers, and so forth. This makes a lot of sense if the goal is to reduce cost. But it doesn't work with microservices, because it introduces coupling, which greatly harms speed.

Each microservice team is autonomous and should be able to pick the best product for each layer. As discussed earlier, each team should be given a menu of supported products to choose from. For example, the programming language can be Java, Scala, or Node.js. The team can purchase formal product support for all of these options. Rather than one product there are now potentially a handful. A lot of cost-focused managers would say "Just use Java," but

that could become too constrainting. Languages and runtime each have their own niche.

Besides the products themselves, the other big form of duplication is in the instances themselves. It's technically more efficient to have one enormous database that the monolithic application or microservices use. But again, that creates coupling, which harms speed. When the DBA team needs to take the database down for patching, everyone must plan around it.

So yes, microservices is not technically the most efficient. Each microservices team selects, sometimes procures, and always runs their own stack. But the point of microservices is speed.

Eventual Consistency

One of the biggest issues that enterprises have with microservices is the fact that not all data is strongly consistent. A product can exist in 20 different microservices, each having a copy of the product from a different point in time. Enterprises are used to having one single database for all commerce data, which is strongly consistent—one copy of the product that is always up to date. This is not the microservices model because it introduces coupling, which harms development velocity.

Within a commerce platform, enterprises have historically expected that data is strongly consistent. But data has never been consistent across an entire enterprise. For example, CRM, ERP, and commerce applications each have their own representation of a customer, with updates to data being propagated between applications asynchronously.

Within a microservices-based commerce system, there is always at least one microservice that has the most up-to-date data. For example, the product microservice owns all product data. But the product catalog and search microservices might have a cached copy of that data. Consider using a wiki or some other internal place to document which microservice owns which bit of data.

Testing

Microservices makes testing both easier and more difficult at the same time. Let's begin from the developers' workstation and work up to production.

Locally, a developer will need to run unit tests to ensure that code works in isolation, with as few variables as possible. This testing is much easier with microservices because the codebase is much smaller. The entire microservice can be run locally in isolation, without too much being mocked out. Because developers are real owners, they have a stronger incentive to write better code and increase unit testing coverage.

Like unit testing, component testing is fairly easy. Each microservice is fairly easy to run on its own in an environment. Often, only one API is exposed. That API can be thoroughly tested on its own. Events can also be mocked and passed into the microservice to see how it handles them.

Integration testing in a remote environment becomes tougher with microservices. It's no longer just one application that integrates with a few monolithic applications—it's dozens, hundreds, or even thousands of small microservices. Properly testing your microservice's functionality often requires writing test scripts that call other microservices to get them to produce events that your microservice consumes. For example, if you're testing the Order Status microservice, you'll need to first place an order using the Order microservice. This complexity is the essential tradeoff of microservices—trading inner complexity for outer complexity. Although integration testing is difficult, the faster development velocity gained by microservices makes this extra work worth it.

Monitoring

Like testing, monitoring becomes both easier and more difficult at the same time.

The health of each microservice is very easy to assess. It's the health of the overall system that's trickier to assess. If the Order Status microservice is unavailable and the application is able to gracefully handle the failure, there's no problem. But what happens if the Order microservice is down? Where is the line drawn between healthy and unhealthy? That line matters because at some point you need to cut off traffic and fix the problem(s). A monolithic application, on the other hand, is fairly easy to monitor. It's typically working or it's not.

Another issue that microservices introduces is end-to-end transaction tracing. Out of the dozens, hundreds or thousands of microser-

vices, which one is causing the problem? Once you identify the microservice, you then have to trace its dependencies. Perhaps a microservice upstream from it has stopped publishing events. It gets complicated quickly. If you can implement some technology and processes, it becomes manageable. But this is what most enterprises cite as the biggest issue with microservices. Monoliths are slow but the scope of problems is less.

How to Incrementally Adopt Microservices

There are essentially three different approaches to adopting microservices. Let's explore them.

Net New

If you're building a large new commerce platform from scratch, with hundreds of developers, it makes sense to start using microservices. In this model, each team is responsible for one microservice (Figure 2-14). All microservices are on the same level.

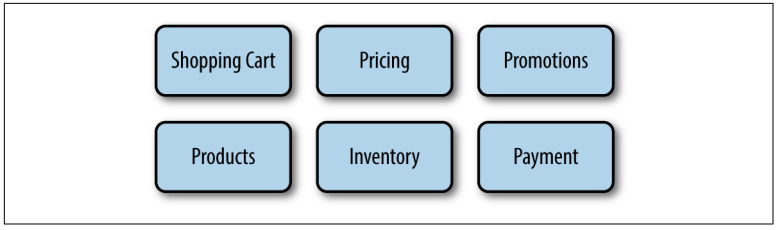


Figure 2-14. How you would structure your microservices if you were starting from scratch

It does not make sense to start with this approach if you're building a smaller application or you don't know whether the complexity will justify the initial overhead of microservices. Microservices is best for when you'll be writing millions of lines of code with hundreds or even thousands of developers. Do not prematurely optimize.

Extend the Monolith

This is the model that most enterprises will follow. It's rare to start from scratch, as you would find in the "Net New" approach. Most will start out with an existing commerce platform of some sort. In this model, you write individual microservices that the monolith then consumes (Figure 2-15).

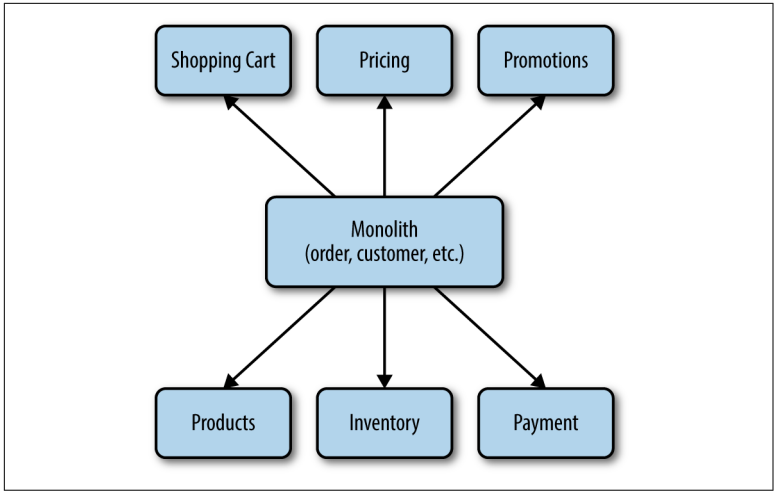


Figure 2-15. How you would structure your microservices if you were starting with an existing monolith

For example, say you need to overhaul pricing in your monolith due to new business requirements. Rather than doing it in the monolith, use this as an opportunity to rewrite pricing in a microservice. The monolith will then call back to the pricing microservice to retrieve prices.

Over time, more and more functionality is pulled out of the monolith until it eventually disappears.

This model is the standard operating procedure for most commerce platforms today. Few do payments, tax calculation, shipping, or ratings and reviews in-house. It's all outsourced to a third-party vendor. Breaking out functionality into separate microservices is the same model, except that the team behind the API you're calling happens to be part of your company.

Decompose the Monolith

This is the hardest (or potentially the easiest) approach depending on how your monolithic application was written. If your application is modular enough, the modules could be pulled out and deployed as standalone microservices (Figure 2-16).

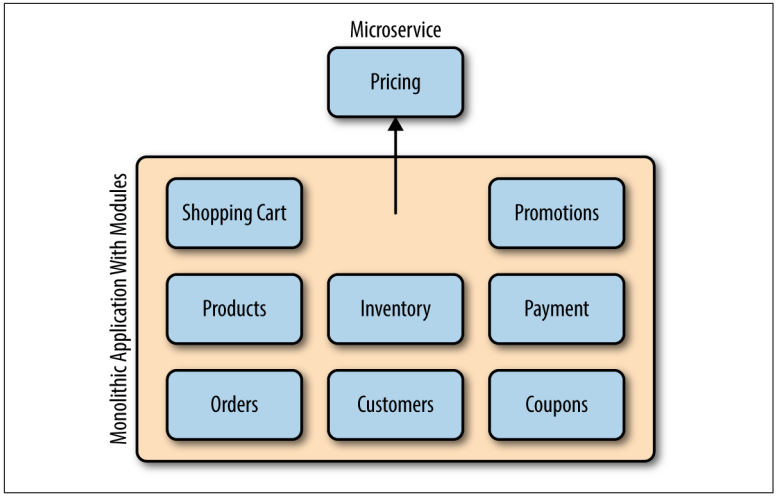


Figure 2-16. How to decompose a monolithic modular application into microservices

The hard part about doing this is that the microservices will all be written to code to the same database, since there was probably only one database in the monolithic application.

Again, this is hard to do but well worth it if you can make it work.

Summary

Speed is the currency in today's fast-paced market. The number of hours it takes to get a new feature or user interface to market matters very much. Microservices is the absolute best architecture for getting new features and user interfaces to market quickly. There are many advantages and, yes, a few disadvantages to this architecture, but overall you'll find none better for large-scale commerce.

Now that we've covered theory, let's discuss the actual technology you'll need to put microservices into practice.

CHAPTER 3
Inner Architecture

Although technology is an important enabler of microservices, it's not the defining characteristic. Microservices is primarily about building small, cross-functional teams that expose one granular piece of business functionality to the enterprise. Each team, in its own self-interest, will naturally pick the best technology to build its own microservice. This is referred to as *inner architecture* because its scope is the inner-workings of a single microservice.

Technically speaking, most of what's in the microservices technology ecosystem is not actually necessary to implement microservices. You *could* implement microservices by using any technology stack. Again, microservices is more about organization structure than any particular technology. Technology is valuable because it helps operationalize microservices at scale. A proper monitoring system will allow you to identify microservices, versions, and instances that are causing problems.

Almost all of the software used for both inner and outer architecture is open source. Microservices arose from the hacker community. The traditional software vendors have been too slow to produce relevant products in this space. Cloud vendors offer some support, but both inner and outer stacks are often composed of various open source products.

Most technology procurement in enterprises is centralized. The CIO will pick one database, programming language, runtime, or whatever, and force everyone within the organization to use it. Monolithic applications have one database, programming language,

runtime, and so on. In an organization built around minimizing costs, this strategy makes sense. The procurement department can squeeze the best prices from the vendors and you can build teams with very specialized product expertise. But this is completely contrary to the principles of microservices because it forces organizations to be horizontally tiered, with each tier focused on a specific layer. This introduces coupling between teams, which dramatically slows down development velocity.

Whereas most technology procurement today is centralized, technology procurement in microservices is distinctly decentralized. As discussed in Chapter 2, each team should have enough autonomy to select its own stack. That freedom comes with responsibility to run and maintain it, so teams are incentivized to make wise decisions. Experimenting with some new version 0.02 framework might sound like fun until it's your turn to fix a problem at 3 AM on a Sunday morning. In traditional enterprises, the people who select technology aren't the ones who actually must support it. They have no personal incentive to make conservative decisions.

APIs

Each microservice team exposes one or more APIs out to the world, so it's important that those APIs are well designed. However, this chapter focuses on inner architecture, so we're going to save the discussion on API gateways and API load balancers for Chapter 4, in which we discuss the outer architecture.

APIs are often HTTP REST, but it is not a requirement. It just happens to be a de facto standard because it's the lowest common denominator and it's easy to read.

Which format is preferred? JSON (JavaScript Object Notation) or XML?

JSON is more compact but more difficult to read, especially when data is more complex and more hierarchical. XML is best for more structured data, but it's more verbose. All modern tooling will work with JSON and XML, but you'll probably find a richer ecosystem around XML due its extensive use.

Either will work just fine. Don't get pulled into religious debates—use whichever you feel comfortable with and what works best for your organization.

Richardson Maturity Model

Leonard Richardson's book *RESTful Web APIs* (O'Reilly, 2013) outlines a four-level maturity model for designing REST APIs. Your APIs should be modeled as far up this hierarchy as possible in order to promote reuse.

Level 0: RPC

This is how most begin using REST. HTTP is simply used as a transport mechanism to shuttle data between two methods—one local, one remote. This is most often used for monolithic applications.

Let's take inventory reservation as an example to illustrate each level of maturity.

To query for inventory, you'd make an HTTP POST to the monolithic application. In this example, let's assume that it's available behind `/App`. The application itself is then responsible for parsing the XML, identifying that the requester is trying to find inventory. Already, this is problematic because you need a lot of business logic to determine where to forward the request within the application. In this example, `queryInventory` has to be mapped back to a function that will return inventory levels:

```
HTTP POST to /App

<queryInventory productId="product12345" />
```

This would be the response you get back:

```
200 OK

<inventory level="84" />
```

To reserve inventory, you'd do something like this:

```
HTTP POST to /App

<reserveInventory>
  <inventory productId="product12345" />
</reserveInventory>
```

Notice how the only HTTP verb used is POST and you're only interacting with the application. There's no notion of an individual service (`/Inventory`) or a service plus function (`/Inventory/reserveInventory`).

Level 1: resources

Rather than interact with an entire application, level 1 calls for interacting with specific *resources* (`/Inventory`) and *objects* within those resources (`product12345`). The resources map back neatly to microservices.

Here's how you would query for inventory with level 1:

```
HTTP POST to /Inventory/product12345

<queryInventory />
```

Notice how you're interacting with `/Inventory` rather than `/App`. You're also directly referencing the product (`product12345`) in the URL.

And this is what you get back:

```
200 OK

<inventory level="84" />
```

To actually reserve inventory, you'd do the following:

```
HTTP POST to /Inventory/product12345

<reserveInventory>
  <inventory qty="1" />
</reserveInventory>
```

Notice how you're only interacting with `/Inventory/product12345` regardless of whether you're querying for inventory or reserving inventory. Although this is certainly an improvement over dealing

with `/App`, it still requires a lot of business logic in your inventory microservice to parse the input and forward it to the right function within your microservice.

Level 2: HTTP verbs

Level 2 is a slight improvement over level 1, making use of HTTP verbs. To date, all HTTP requests have used the POST verb, whether retrieving or updating data. HTTP verbs are built for exactly this purpose. HTTP GET is used to retrieve, HTTP PUT/POST is used to create or update, HTTP DELETE is used to delete.

Going back to our example, you would now use GET rather than POST to *retrieve* the current inventory for a product:

```
HTTP GET to /Inventory/product12345

<inventory level="84" />
```

Reserving inventory is the same as before:

```
HTTP POST to /Inventory/product12345

<reserveInventory>
  <inventory qty="1" />
</reserveInventory>
```

HTTP PUT versus POST is beyond the scope of this discussion.

To create a new inventory record, you'd do the following:

```
HTTP POST to /Inventory

<createInventory>
  <inventory qty="1" productId="product12345" />
</createInventory>
```

Rather than a standard HTTP 200 OK response, you'd get back this:

```
201 Created
Location: /Inventory/product12345

<inventory level="84" />
```

With the `Location` of the newly created object returned in the response, the caller can now programmatically access the new object without being told its location.

Even though this is an improvement over level 1 in that it introduces HTTP verbs, you're still dealing with objects and not individual functions within those objects.

Level 3: HATEOAS (Hypertext As The Engine Of Application State)

Level 3 is the highest form of maturity when dealing with REST interfaces. Level 3 makes full use of HTTP verbs, identifies objects by URI, *and* offers guidance on how to programmatically interact with those objects. The APIs become self-documenting, allowing the caller of the API to very easily interact with the API and not know very much about it. This form of self-documentation allows the server to change URIs without breaking clients.

Level 3/HATEOAS is very difficult to actually achieve in practice. APIs and consumers of those APIs are hard to write. But the principles are valuable to follow.

Let's go back to our inventory example. To retrieve the inventory object, you'd call this:

```
GET /Inventory/product12345
```

And this is what you'd get back:

```
200 OK

<inventory level="84">
   <link rel="Inventory.reserveInventory"
       href="/Inventory/reserveInventory" />
   <link rel="Inventory.queryInventory"
       href="/Inventory/queryInventory" />
   <link rel="Inventory.deleteInventory"
       href="/Inventory/deleteInventory" />
   <link rel="Inventory.newInventory"
       href="/Inventory/newInventory" />
</inventory>
```

Notice how the response includes link tags showing how to perform all available actions against the inventory object. Callers of the APIs just need to know `Inventory.reserveInventory`. From that key, they can look up the URL (`/Inventory/reserveInventory`).

Strive to be level 3–compliant but don't worry if you fall short. This is a high bar.

REST API Markup Languages

REST API modeling should be one of the few standards with which each team should be required to comply. Many implementing

microservices adopt a REST API markup language like Swagger or RAML. These formats are rapidly becoming the standard for REST APIs, which allows for easier interoperability within a microservices implementation as well as between microservices across the industry.

By using one of these standards plus associated tooling, you can do the following:

- *Model* each API using a standard format.
- *Document* each API. Think about it like Javadocs but for REST APIs. The documentation should be autobuilt and published with the build of each microservice.
- *Edit* each API visually and using other advanced tooling.
- *Generate client or server-side stubs* to match the APIs you've modeled. This dramatically simplifies building and calling microservices.

Figure 3-1 presents an example of how you would model an API to reserve inventory using Swagger.

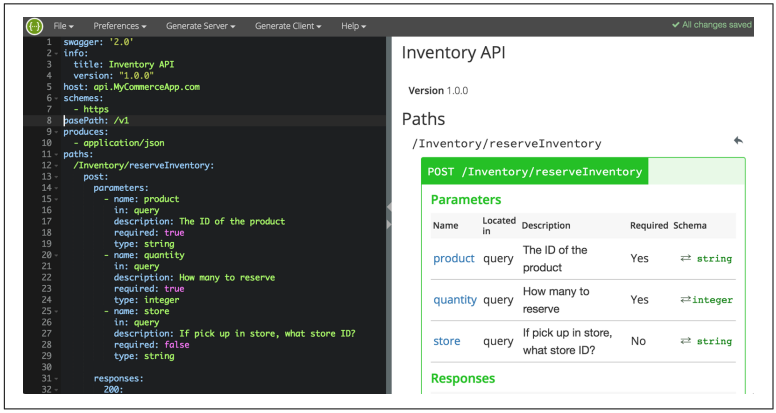


Figure 3-1. Swagger definition

From the YAML definition shown in Figure 3-2, you can generate server and client-side stubs.

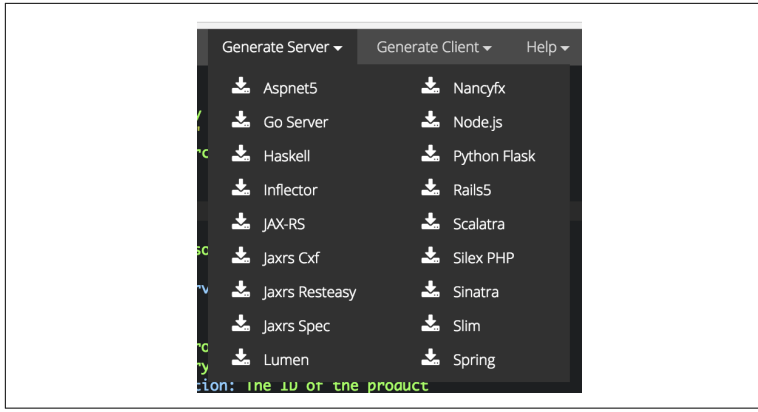


Figure 3-2. Swagger server stubs

Check these definitions in as source code. Pick the standard and associated tooling that works best for you. What matters most is that you use it consistently across all of the microservices you build.

Versioning

As we discussed in Chapter 2, a defining characteristic of microservices is supporting multiple versions of a microservice in the same environment at the same time. A microservice can have one or up to potentially dozens of versions deployed, with clients requesting the major and sometimes the minor version of an instance of the microservice they'd like to retrieve. This introduces complexities.

The first complexity is in development itself. You need to support multiple major and minor code branches, with each of those branches potentially having the standard branches you would expect when writing software. This requires full use of a feature-rich source control management system (SCM). Developers must be proficient at switching between versions and branches within the version. Even though there's more complexity, microservices makes this somewhat easier because the codebase for each microservice is so small and there's a lot less coordination within a team because team sizes are so small.

When you've built your microservice, you then need to deploy it to an environment. Your deployment mechanism should be aware of the multiple versions of the code you're running and be able to quickly pull out a version if it's not working well. Public cloud ven-

dors and many microservices platforms are beginning to offer this functionality.

After you've deployed a specific version of a microservice, you need to advertise its availability to clients. Depending on your approach, which we'll cover shortly, you'll need to allow clients to query for a specific major and sometimes minor version of an instance of the microservice that they'd like to retrieve. Monolithic applications are almost always retrieved behind the same URL, regardless of the version.

Then, you need to have version-aware autoscaling. Suppose that you have versions 1, 2, and 3 of a microservice live. Versions 2 and 3 could be getting all of the traffic. Because there's no traffic going to version 1, your autoscaling system could kill all instances of it. If a client later needs to call version 1, your autoscaling system will need to spin up new capacity in response, which could potentially take seconds or minutes depending on how long it takes to retrieve and instantiate the container image.

Finally, monitoring must be all version-aware. Only one version of a microservice might be causing problems. It's important to be able to look at health, performance, and other metrics on a per-version basis.

Alternate Formats

To this point, we've assumed that the underlying protocol is HTTP and the format is XML or JSON. The combination of HTTP and XML or JSON works well because it's human-readable, easily understood, and can be used natively by established products like proxies and API gateways.

In theory, you should rarely if ever make synchronous HTTP calls between microservices, especially if the client is waiting on the response. In practice, however, it's all too common.

For the absolute best performance, you can use binary-level transport systems. Examples include Apache Thrift, Apache Avro, and Google Protocol Buffers.

For example, Protocol Buffers from Google are 3 to 10 times smaller and 20 to 100 times faster than XML. Rather than human readability, these implementations are optimized for performance.

Containers

Containers are an important part of most microservices deployments. Containers are not required for microservices but they've both come of age at about the same time and they're extremely complementary.

This section will be about containers for a single microservice. In the next section, in which we cover outer complexity, we'll discuss how you deploy and orchestrate containers, which is perhaps their biggest value.

The value for each microservice team is that they can neatly package up their microservices into one or more containers. They can then promote those containers through environments as atomic, immutable, units of code/configuration/runtime/system libraries/operating system/start-and-stop hooks. A container deployed locally will run the exact same way in a production environment.

In addition to application packaging, containers are also extremely lightweight, often just being a few hundred megabytes in size versus virtual machine (VM) images, which often are multiple gigabytes. When a container is instantiated, a thin writable layer is created over the top of the base container image. Because of their small size and the innovative approach to instantiation, a new container can be provisioned and running in a few milliseconds as opposed to the many minutes it takes to instantiate a VM. This makes it easier for developers to run microservices locally and for individual microservices to rapidly scale up and down in near real-time response to traffic. Many containers live for just a few hours. Google launches two billion containers a week, with more than 3,000 started per second, not including long-running containers. Many containers live for just a few seconds. There's such little overhead to provision and later kill a container.

In the case of Docker, you build containers declaratively by using *Dockerfiles*. Dockerfiles are are simple YAML (*Y*AML *A*in't *M*arkup *L*anguage) scripts that define the base Docker image, along with any commands required to install/configure/run your software and its dependencies. Here's a very simple example:

```
FROM    centos:centos6

# Enable Extra Packages for Enterprise Linux (EPEL) for CentOS
RUN     yum install -y epel-release
```

```
# Install Node.js and npm
RUN     yum install -y nodejs npm

# Install app dependencies
COPY package.json /src/package.json
RUN cd /src; npm install

# Bundle app source
COPY . /src

EXPOSE  8080
CMD ["node", "/Inventory/index.js"]
```

Simply build by running `$ docker build -f /path/to/a/Dockerfile .` The output is an image that can be instantiated as a container in a few milliseconds. Containers should always be *immutable* once built. If you have to deploy a new version of your code, change your application's configuration, or set an environment variable, you should update your Dockerfile, rebuild your image, and deploy a new version. Never update a live running container. Immutable infrastructure is crucial to repeatable testing across environments. With immutable containers, you know that an individual container will run the exact same locally as in production.

Lightweight Runtimes

Remember that each microservice is fairly small and uncomplicated. The runtime required to run a 5,000 line application is very different than would be required for a 10,000,000 line application. The large application has 2,000 times more code, which could be using hundreds of advanced features like distributed database transactions. Because of this reduced complexity, you can easily use smaller, lighter, faster runtimes. Many of these runtimes are a few megabytes in size and can be started in a few milliseconds.

Circuit Breakers

Calls from one microservice to another should always be routed through a *circuit breaker* such as Hystrix from Netflix. Circuit breakers are a form of *bulkheading* in that they isolate failures.

If Microservice A synchronously calls Microservice B without going through a circuit breaker, and Microservice B fails, Microservice A is likely to fail as well. Failure is likely because Microservice A's

request-handling threads end up getting stuck waiting on a response from Microservice B. This is easily solved through the use of a circuit breaker.

A circuit breaker uses active, passive, or active plus passive monitoring to keep tabs on the health of the microservice you're calling. Active monitoring can probe the health of a remote microservice on a scheduled basis, whereas passive monitoring can monitor how requests to a remote microservice are performing. If a microservice you're calling is having trouble, the circuit breaker will stop making calls to it. Calling an endpoint that is having trouble only exacerbates its problems and ties up valuable request-handling threads.

To further protect callers from downstream issues, circuit breakers often have their own threadpool. The request-handling thread makes a request to connect to the remote microservice. Upon approval, the circuit breaker itself, using its own thread from its own pool, makes the call. If the call is unsuccessful, the circuit breaker thread ends up being blocked and the request-handling thread is able to gracefully fail. Figure 3-3 presents an overview of how a circuit breaker functions.

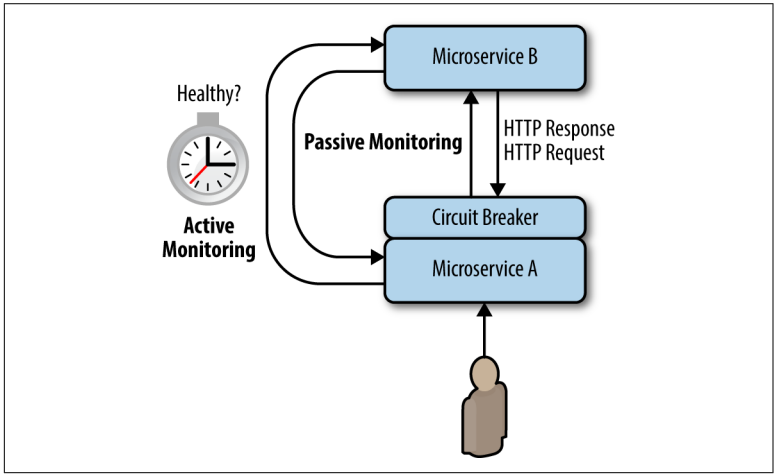


Figure 3-3. Circuit breakers are required for microservices

The use of a circuit breaker should be one of the few mandates to which all microservices teams should have to adhere.

Polyglot

Polyglot simply means that you can use multiple languages to write microservices. Each microservice is often implemented using just one language. Many see it as a defining characteristic.

As we discussed in Chapter 2, each microservice team should have some freedom in selecting the language/runtime for its own microservice. A team writing a microservice for inventory might want to use Node.js because of its ability to gracefully increment and decrement a number without locking. A team writing a product recommendations microservice might want to use R, for its ability to work with statistics. A team writing a chat microservice might want to use Erlang, for its inherent clustering abilities. Each team should have some choice in the programming language and runtime; for example, they should not all be forced to use Java or .NET.

That being said, you should not allow teams to select just any programming language and runtime they want. Instead, you should provide a list from which they can select. For example, Java and Node.js could be the general purpose language, whereas C++ and Go could be for microservices requiring high performance. Enterprises can contract with a handful of vendors for licensing and support. This compromise should offer teams the flexibility they need, while not creating too much of a headache for enterprises.

Software-Based Infrastructure

Operating in a cloud is a requirement for microservices because it allows each team to consume its own infrastructure, platform, and software, all as a service. Remember, each microservice team needs to own its *entire* stack and not be dependent on any other teams. Before the cloud, centralized IT teams were required to support each layer—compute, network, storage, and so on. Due to Conway's Law, this naturally led to large monolithic applications. What used to take months of work by a specialized team can now be performed by a single API call with less specialized individuals on a microservice team.

Each team should treat their infrastructure as essentially "disposable." There's an old "Cattle versus Pets" meme coined by Randy Bias that's illustrative of this principle. If you have 1,000 cattle in a herd and one of them breaks its leg, you euthanize it. It doesn't have a

name, it doesn't sleep in your bed, your kids don't wish it a happy birthday. It's a cow that's indistinguishable from its 999 peers in the same herd. A pet, on the other hand, is different. People will run up massive veterinarian bills to fix a pet's broken leg. It's a pet—it has a name, it spends time with your family, your kids celebrate its birthday. This isn't a perfect analogy, because cattle are worth thousands of dollars, but the larger point Bias is trying to make about the cloud is spot-on—stop treating your servers like pets. Containers deployed to the cloud have IPs and ports that are not known, and when they have problems, they are killed. They lead short anonymous lives. What matters is the aggregate throughput—not the fate of any single instance.

As part of achieving disposable infrastructure, HTTP session state (login status, cart, pages visited, etc.) should be persisted to a third-party system, like a cache grid. None of it should be persisted to a container, because a container might live for only for a few seconds. Remember that each microservice must exclusively own its data. Multiple microservices can share the same store, but each must have its own partition. One microservice cannot access the state of another microservice.

A microservice's configuration could be packed into the container or it can be externalized and pulled by the microservice as required. It's best to place the configuration inside the container so that the container itself runs exactly the same regardless of its environment. It's difficult to ensure repeatability if the configuration can be modified at runtime.

Sometimes microservices have singletons. These singletons can be used for distributing locks, synchronously processing data (e.g., orders), generating sequential numbers, and so on. In the days of static infrastructure, this was easy—you'd have a singleton that was behind a static IP address that never changed. But in the cloud, where containers might live for only for seconds, it's impossible to have named long-lived singletons that are always available. Instead, it's best to employ some form of leader election, in which singletons are named at runtime.

CHAPTER 4
Outer Architecture

The *outer architecture* is the space *between* microservices and is by far the most difficult part. As we've pointed out throughout this book, microservices is a tradeoff from inner complexity to outer complexity. Broadly speaking, the outer architecture of microservices is the space between individual microservices. Think of it as anything that is not the responsibility of an individual microservice team.

Outer architecture includes all of the infrastructure on which individual microservices are deployed, discovering and connecting to microservices, and releasing new versions of microservices, communicating between microservices, and security. It's a wide area that can become complex.

Outer architecture is best handled by a single team that is composed of your best programmers. You don't need a big team—you need a small team of people who are really smart.

> There is no number of ordinary eight-year-olds who, when organized into a team, will become smart enough to beat a grandmaster in chess.
>
> —Scott Alexander, 2015

Your goal should be to free up each microservice team to focus on just its own microservice.

Part of what complicates outer architecture is that most of the technology is brand new. The term "microservices" wasn't used widely until 2013. The technology that powers the inner architecture is all

widely established. After all, an individual microservice is just a small application.

Software-Based Infrastructure

Although each microservice team can choose its own programming language, runtime, datastore, and other upper-stack elements, all teams should be constrained to using the same cloud provider. The major cloud platforms are all "good enough" these days, both in terms of technical and commercial features.

The first major benefit is that multiple teams can access the same shared resources, like messaging, API gateways, and service discovery. Even though each team has the ability to make some choices independently, you generally want to standardize on some things across teams; for example, the messaging system. You wouldn't want 100 different teams each using its own messaging stack—it just wouldn't work. Instead, a centralized team must choose one implementation and have all of the microservices teams use it. It doesn't make sense to fragment at that level and it's difficult to implement. An added advantage of everyone using the same implementations is that latency tends to be near zero.

By standardizing on a cloud vendor, your organization has the ability to build competency with one of them. Public clouds are not standardized and specialized knowledge is required to fully use each cloud. Individuals can build competency, which is applicable regardless of the team to which they're assigned. Teams can also publish blueprints and best practices that others can use to get started.

Working across clouds adds unnecessary complexity and you should avoid doing it.

Container Orchestration

Containers are used by individual teams to build and deploy their own microservices, as part of inner architecture. *Container orchestration* is the outer architecture of microservices.

Simply put, container orchestration is the system that runs individual containers on physical or virtual hosts. Managing a handful of containers from the command line is fairly straightforward. You SSH into the server, install Docker, run your container image, and

expose your application's host/port. Simple. But that doesn't work with any more than a handful of containers. You might have dozens, hundreds or even thousands of microservices, with each microservice having multiple versions and multiple instances per version. It just doesn't scale.

These systems can also be responsible for the following:

- Releasing new versions of your code
- Deploy the code to a staging environment
- Run all integration tests
- Deploy the code to a production environment
- Service registry—making a microservice discoverable and routing the caller to the best instance
- Load balancing—both within a node and across multiple nodes
- Networking—overlay networks and dynamic firewalls
- Autoscaling—adding and subtracting containers to deal with load
- Storage—create and attach existing volumes to containers
- Security—identification, authorization and authentication

Each of these topics will be covered in greater depth in the sections that follow.

Container orchestration is essentially a new form of Platform-as-a-Service (PaaS) that many use in combination with microservices. The container itself becomes the artifact that the container orchestration system manages, which makes it extremely flexible. You can put anything you want in a container. Traditionally, a PaaS is considered "opinionated" in that it forces you do things using a very prescribed approach. Container orchestration systems are much less opinionated and are more flexible.

Besides flexibility, infrastructure utilization is a top driving factor for container orchestration adoption. Virtual Machines (VMs), even in the cloud, have fixed CPU and memory, whereas multiple containers deployed to a single host share CPU and memory. The container orchestration system is responsible for ensuring that each host (whether physical or virtual) isn't overtaxed. Utilization can be kept at 90 or 95 percent, whereas VMs are typically 10 percent uti-

lized. If utilization of a host reaches near 100 percent, containers can be killed and restarted on another, less-loaded host.

Container orchestration, like the cloud, is a system that every team should be forced to use. These systems are extremely difficult to set up, deploy, and manage. After it is set up, the marginal cost to add a new microservice is basically zero. Each microservice team is able to easily use the system.

Let's explore some of the many roles that container orchestration systems can play.

Releasing Code

Each team must continually release new versions of its code. Each team is responsible for building and deploying its own containers, but they do so using the container orchestration system. Every team should release code using the same process. The artifacts should be containers that, like microservices, do only one thing. For example, your application should be in one container and your datastore should be in another. Container orchestration systems are all built around the assumption of a container running just one thing.

To begin, you need to build a container image, inclusive of code/configuration/runtime/system libraries/operating system/start-and-stop hooks. As we previously discussed, this is best done through a Dockerfile YAML that is checked in to source control and managed like source code. The Dockerfile should be tested on a regular basis. It should be upgraded. The start/stop hook scripts should be tested, too.

After you've built your image, you need to define success/failure criteria. For example, what automated tests can you run to verify that the deployment was successful? It's best to throughly test out every API, both through the API and through any calling clients. What constitutes a failure? If an inconsequential function isn't working, should the entire release be pulled?

Then, you need to define your rollout strategy. Are you replacing an existing implementation of a microservice that's backward compatible with the existing version? Or, are you rolling out a new major version that is not backward compatible? Should the deployment proceed in one operation (often called "blue/green"), or should it be phased in gradually, at say 10 percent per hour (often called "can-

ary")? How many regions should it be deployed to? How many fault domains? How many datacenters? Which datacenters?

Following the deployment, the container orchestration system then must update load balancers with the new routes, cutover traffic, and then run the container's start/stop hooks.

As you can see, this all can become quite complicated. With a proper container orchestration system and some experience, releases can be as carried out often as every few hours. Again, it takes a small team of very specialized people to build the container orchestration system. Then, individual teams can use it.

Service Registry

When the container orchestration system places a container on a host, clients need to be able to call it. But there are a few complicating factors:

- Containers might live for only a few seconds, minutes, or hours. They are *ephemeral* by definition.
- Containers often expose nonstandard ports. For example, you might not always be able to hit HTTP over port 80.
- A microservice is likely to have many major and minor versions live at the same time, requiring the client to state a version in the request.
- There are dozens, hundreds or even thousands of different microservices.

There are two basic approaches: client-side and server-side.

The client-side approach is straightforward conceptually. The client (be it an API gateway, another microservice, or a user interface) queries a standalone service registry to ask for the path to a fully qualified endpoint. Again, the client should be able to specify the major and possibly the minor version of the microservice for which it's looking. The client will get back a fully qualified path to an instance of that microservice, which it can then call over and over again.

The main benefit of this approach is that there are no intermediaries between the client and the endpoint. Calls go directly from the client to the endpoint without traversing through a proxy. Also, clients can

be more rich in how they query for an endpoint. The query could be a formal JSON document stating version and other quality-of-service preferences, depending on the sophistication of your service registry.

The major drawback of this approach is that the client must "learn" how to query each microservice, which is a form of coupling. It's not transparent. Each microservice will have its own semantics about how it needs to be queried because each microservice implementation will differ. Another issue is that the client will need to requery for an endpoint if the one it's communicating with directly fails.

Although the client-side approach can be helpful, the server-side method is often preferable due to its simplicity and extensive use in the world today. This approach is to basically use a load balancer. When the container orchestration places a container, it registers the endpoint with the load balancer. The client can make some requests about the endpoint by specifying HTTP headers or similar.

Unlike client-side load balancing, the client doesn't need to know how to query for an endpoint. The load balancer just picks the best endpoint. It's very simple.

Load Balancing

If you use a server-side service registry, you'll need a load balancer. Every time a container is placed, the load balancer needs to be updated with the IP, port and other metadata of the newly-created endpoint.

There are two levels of load balancing within a container orchestration system: local and remote.

Local load balancing is load balancing within a single host. A host can be a physical server or it can be virtualized. A host runs one or more containers. Your container orchestration system might be capable of deploying instances of microservices that regularly communicate to the same physical host. Figure 4-1 presents an overview.

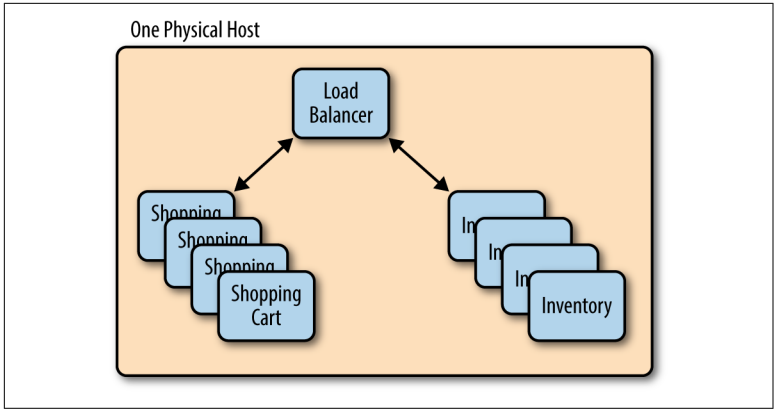


Figure 4-1. Local load balancing within a single host

By being told or by learning that certain microservices communicate, and by intelligently placing those containers together on the same host, you can minimize the vast majority of network traffic because most of it is local to a single host. Networking can also be dramatically simplified because it's all over localhost. Latency is zero, which helps improve performance.

In addition to local load balancing, remote load balancing is what you'd expect. It's a standalone load balancer that is used to route traffic across multiple hosts.

Look for products specifically marketed as "API load balancers." They're often built on traditional web servers, but they are built more specifically for APIs. They can support identification, authentication, and authorization-related security concerns. They can cache entire responses where appropriate. And finally, they have better support for versioning.

Networking

Placed on a physical host, the container needs to join a network. One of the benefits of the modern cloud is that *everything* is just software, including networks. Creating a network is now as simple as invoking it, as shown here:

```
$ docker network create pricing_microservice_network
```

With this new "pricing_microservice_network" network created, you can run a container and hook up its network to the container:

```
$ docker run -itd corp/pricing_microservice
    --network=pricing_microservice_network
    --name Pricing Microservice
```

Of course, container orchestration does this all at scale, which is part of the value. Your networking can become quite advanced depending on the container orchestration you use. What matters is that you define separate, isolated networks for each tier of each microservice, as demonstrated in Figure 4-2.

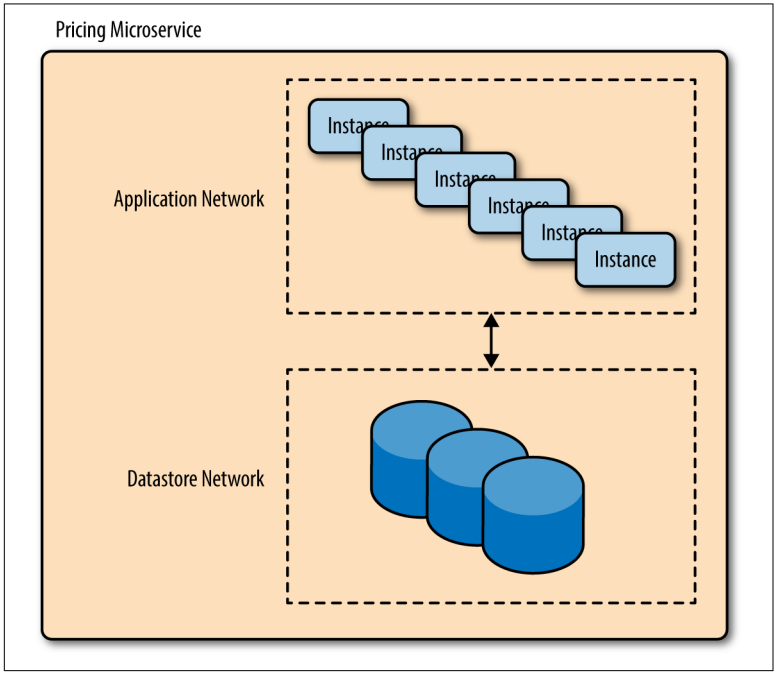


Figure 4-2. Software-based overlay networks are required for full isolation and proper bulkheading

Proper use of overlay networks is another form of *bulkheading*, which limits the damage someone can do if they break into a network. Someone can break into your Inventory microservice's application network and not have access to the Payment microservice's database network.

As part of networking, container orchestration can also deploy software-based firewalls. By default, there should be absolutely no network connectivity between microservices. But if your shopping cart microservice needs to pull up-to-date inventory before check-

out, you should configure your container orchestration system to automatically expose port 443 on your inventory microservice and permit only the Shopping Cart microservice to call it over Transport Layer Security (TLS). Exceptions should be made on a per-microservice basis. You would never set up a microservice to be exposed to accept traffic from any source.

Finally, you want to ensure that the transport layer between microservices is secured by using TLS or alternate. For example, never allow plain HTTP traffic within your network.

Autoscaling

Commerce is unique in the spikiness of traffic. Figure 4-3 shows the number of page views per second over the course of the month of November for a leading US retailer:

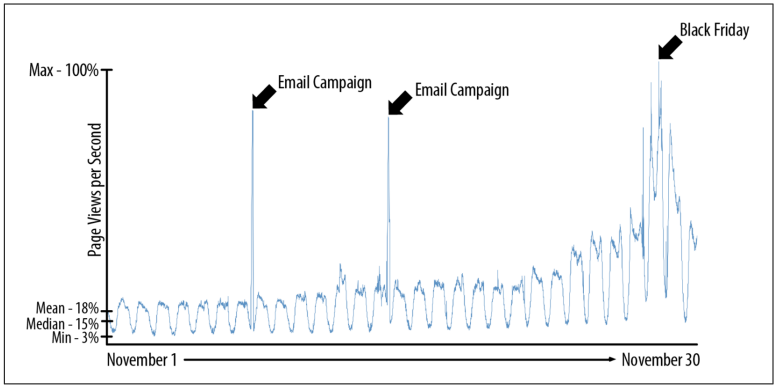


Figure 4-3. Commerce traffic is spiky

This is only web traffic. If this retailer were to be fully omnichannel, the spikes would be even more dramatic.

Before cloud and containers, this problem was handled by overprovisioning so that at steady state the system would be a few percent utilized. This practice is beyond wasteful and is no longer necessary.

The cloud and its autoscaling capabilities help but VMs take a few minutes to spin up. Spikes of traffic happen over just a few seconds, when a celebrity with 50 million followers publishes a link to your website over social media. You don't have minutes.

Containers help because they can be provisioned in just a few milliseconds. The hosts on which the containers run are preprovisioned already. The container orchestration system just needs to instantiate some containers.

Note that autoscaling needs to be version-aware. For example, versions 2.23 and 3.1 of your pricing microservice need to be individually autoscaled.

Storage

Like networking, storage is all software-defined, as well.

Containers themselves are mostly immutable. You shouldn't write any files to the local container. Anything persistent should be written to a remote volume that is redundant, highly available, backed up, and so on. Those remote volumes are often cloud-based storage services.

Defining volumes for each microservice and then attaching the right volumes to the right containers at the right place is a tricky problem.

Security

Network-level security is absolutely necessary. But you need an additional layer of security on top of that.

There are three levels: identification, authentication, and authorization. Identification forces every user to identify itself. Users can be humans, user interfaces, API gateways, or other microservices. Identification is often through a user name or public key. After a user has identified itself, the user must then be authenticated. Authentication verifies that the user is who it claims it is. Authentication often occurs through a password or a private key. After it has been identified and authenticated, a user must have authorization to perform an action.

Every caller of a microservice must be properly identified, authenticated, and authorized, even "within" the network. One microservice can be compromised. You don't want someone launching an attack from the compromised microservice.

API Gateway

A web page or screen on a mobile device might require retrieving data from dozens of different microservices. Each of those clients will need data tailored to it. For example, a web page might display 20 of a product's attributes but an Apple Watch might display only one.

You'll want an API gateway of some sort to serve as the intermediary, as depicted in Figure 4-4.

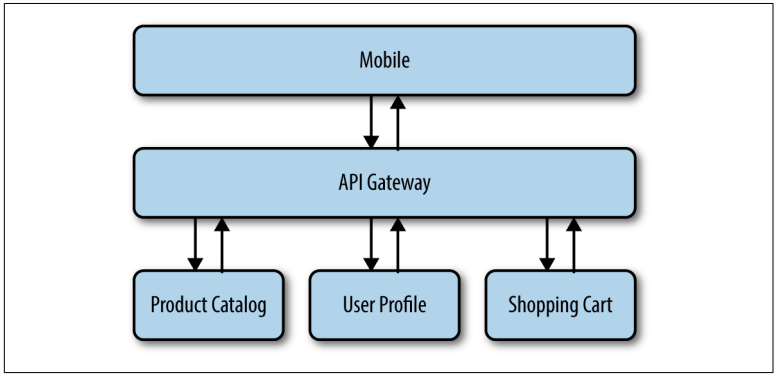


Figure 4-4. Aggregator pattern

The client makes the call to the API gateway and the API gateway makes concurrent requests to each of the microservices required to build a single response. The client gets back one tailored representation of the data. API gateways are often called "Backends for your frontend."

When you call APIs, you need to query only for what you want. A product record might have 100 properties. Some of those properties are only relevant to the warehouse. Some are only relevant to physical stores. Remember, microservices are meant to be omnichannel. When you want to display a product description on an Apple Watch, you don't want the client to retrieve all 100 properties. You don't even want the API gateway to retrieve those 100 properties from the Product microservice because of the performance hit. Instead, each layer should be making API calls (client → API gateway, API gateway → each microservice) that specify which properties to return. This too creates coupling because the layer above now needs to know more details about your service. But it's probably worth it.

The issue with API gateways is that they become tightly coupled monoliths because they need to know how to interact with every client (dozens) and every microservice (dozens, hundreds or even thousands). The very problem you sought to remedy with microservices can reappear if you're not careful.

Eventing

We've mostly discussed API calls into microservices. Clients, API gateways, and other microservices might synchronously call into a microservice and ask for the current inventory level for a product, or for a customer's order history, for example.

But behind the synchronous API calls, there's an entire ecosystem of data that's being passed around asynchronously. Every time a customer's order is updated in the Order microservice, a copy should go out as an *event*. Refunds should be thrown up as events. Eventing is far better than synchronous API calls because it can buffer messages until the microservice is able to process them. It prevents outages by reducing tight coupling.

In addition to actual data belonging to microservices, system events are also represented as microservices. Log messages are streamed out as events—the container orchestration system should send out an event every time a container is launched; every time a healthcheck fails, an event should go out. *Everything* is an event in a microservices ecosystem.

Why is it called "eventing" and not "messaging"? An event is essentially a message, but with one key difference: volume. Traditionally, messages were used exclusively to pass data. In a microservices ecosystem, everything is an event. Modern eventing systems can handle millions or tens of millions of events per second. Messaging is meant for much lower throughput. Although this is more characteristic of implementations, normal messaging tends to be durable and ordered. It's usually brokered. Eventing is often unordered and often nonbrokered.

A key challenge with Service Oriented Architecture (SOA) is that messages were routed through a centralized Enterprise Service Bus (ESB), which was too "intelligent."

> The microservice community favours an alternative approach: smart endpoints and dumb pipes.
>
> —Martin Fowler, 2014

In this model, the microservices themselves hold all of the intelligence—not the "pipes," like an ESB. In a microservices ecosystem, events shouldn't be touched between the producer and consumer. Pipes should be dumb.

Idempotency is also an important concept in microservices. It means that an event can be delivered and consumed more than once and it will not change the output.

This example is not idempotent:

```
<credit>
   <amount>100</amount>
   <forAccount>1234</account>
</credit>
```

This event is idempotent:

```
<credit>
   <amount>100</amount>
   <forAccount>1234</account>
   *<creditMemoID>4567</creditMemoId>*
</credit>
```

Summary

Microservices is revolutionizing how commerce platforms are built by allowing dozens, hundreds, or even thousands of teams to seamlessly work in parallel. New features can be released in hours rather than months.

As with any technology, there are drawbacks. Microservices does add complexity, specifically outer complexity. The enabling technology surrounding microservices is rapidly maturing and will become easier over time.

Overall, microservices is probably worth it if your application is sufficiently complex and you have the organizational maturity. Give it a try. It's a new world out there.

About the Author

Kelly Goetsch is chief product officer at commercetools, where he oversees product management, development, and delivery. He came to commercetools from Oracle, where he led product management for its microservices initiatives. Kelly held senior-level business and go-to-market responsibilities for key Oracle cloud products representing nine-plus figures of revenue for Oracle.

Prior to Oracle, he was a senior architect at ATG (acquired by Oracle), where he was instrumental to 31 large-scale ATG implementations. In his last years at ATG, he oversaw all of Walmart's implementations of ATG around the world. He holds a bachelor's degree in entrepreneurship and a master's degree in management information systems, both from the University of Illinois at Chicago. He holds three patents, including one key to distributed computing.

Kelly has expertise in commerce, microservices, and distributed computing, and speaks and publishes extensively on these topics. He is also the author of the book on the intersection of commerce and cloud, *eCommerce in the Cloud: Bringing Elasticity to eCommerce* (O'Reilly, 2014).