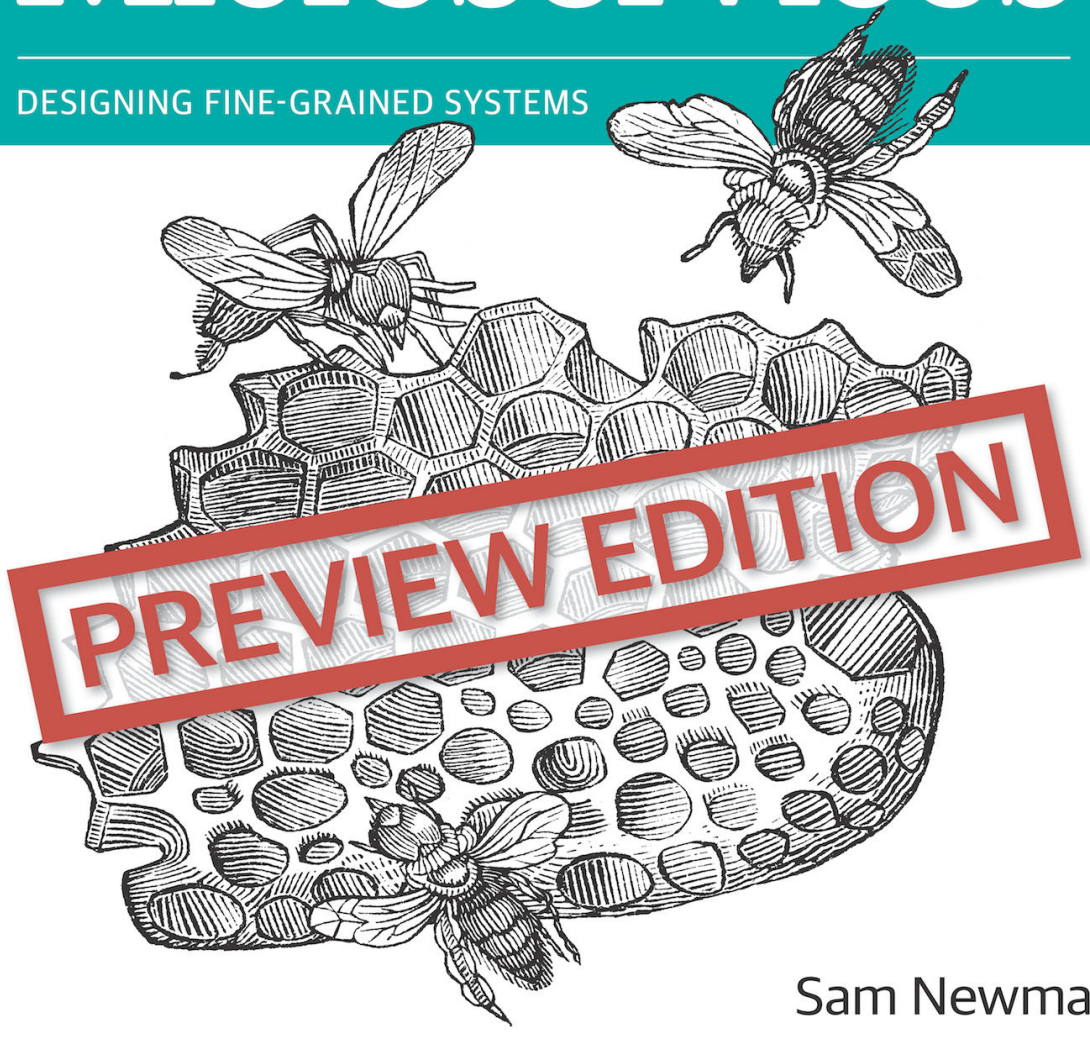
O'REILLY®

Compliments of NGINX

Building Microservices

DESIGNING FINE-GRAINED SYSTEMS



PREVIEW EDITION

Sam Newman

This Preview Edition of *Building Microservices*, *Chapters 1, 4, and 11*, is a work in progress. The final book is currently scheduled for release in February 2015 and will be available at *oreilly.com* and other retailers once it is published.

Building Microservices

Sam Newman

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Building Microservices
by Sam Newman

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: Mike Loukides and Brian MacDonald
Production Editor: Kristen Brown
Copyeditor: Rachel Monaghan
Proofreader: Jasmine Kwityn
Indexer: Judith McConville
Interior Designer: David Futato
Cover Designer: Ellie Volckhausen
Illustrator: Rebecca Demarest

February 2015: First Edition

Revision History for the First Edition
2014-01-30: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491950357* for release details.

978-1-491-95035-7

[LSI]

Table of Contents

CHAPTER 1
Microservices

For many years now, we have been finding better ways to build systems. We have been learning from what has come before, adopting new technologies, and observing how a new wave of technology companies operate in different ways to create IT systems that help make both their customers and their own developers happier.

Eric Evans's book *Domain-Driven Design* (Addison-Wesley) helped us understand the importance of representing the real world in our code, and showed us better ways to model our systems. The concept of continuous delivery showed how we can more effectively and efficiently get our software into production, instilling in us the idea that we should treat every check-in as a release candidate. Our understanding of how the Web works has led us to develop better ways of having machines talk to other machines. Alistair Cockburn's concept of hexagonal architecture guided us away from layered architectures where business logic could hide. Virtualization platforms allowed us to provision and resize our machines at will, with infrastructure automation giving us a way to handle these machines at scale. Some large, successful organizations like Amazon and Google espoused the view of small teams owning the full lifecycle of their services. And, more recently, Netflix has shared with us ways of building antifragile systems at a scale that would have been hard to comprehend just 10 years ago.

Domain-driven design. Continuous delivery. On-demand virtualization. Infrastructure automation. Small autonomous teams. Systems at scale. Microservices have emerged from this world. They weren't invented or described before the fact; they emerged as a trend, or a pattern, from real-world use. But they exist only because of all that has gone before. Throughout this book, I will pull strands out of this prior work to help paint a picture of how to build, manage, and evolve microservices.

Many organizations have found that by embracing fine-grained, microservice architectures, they can deliver software faster and embrace newer technologies. Microser-

vices give us significantly more freedom to react and make different decisions, allowing us to respond faster to the inevitable change that impacts all of us.

What Are Microservices?

Microservices are small, autonomous services that work together. Let's break that definition down a bit and consider the characteristics that make microservices different.

Small, and Focused on Doing One Thing Well

Codebases grow as we write code to add new features. Over time, it can be difficult to know where a change needs to be made because the codebase is so large. Despite a drive for clear, modular monolithic codebases, all too often these arbitrary in-process boundaries break down. Code related to similar functions starts to become spread all over, making fixing bugs or implementations more difficult.

Within a monolithic system, we fight against these forces by trying to ensure our code is more cohesive, often by creating abstractions or modules. Cohesion—the drive to have related code grouped together—is an important concept when we think about microservices. This is reinforced by Robert C. Martin's definition of the *Single Responsibility Principle*, which states "Gather together those things that change for the same reason, and separate those things that change for different reasons."

Microservices take this same approach to independent services. We focus our service boundaries on business boundaries, making it obvious where code lives for a given piece of functionality. And by keeping this service focused on an explicit boundary, we avoid the temptation for it to grow too large, with all the associated difficulties that this can introduce.

The question I am often asked is *how small is small?* Giving a number for lines of code is problematic, as some languages are more expressive than others and can therefore do more in fewer lines of code. We must also consider the fact that we could be pulling in multiple dependencies, which themselves contain many lines of code. In addition, some part of your domain may be legitimately complex, requiring more code. Jon Eaves at RealEstate.com.au in Australia characterizes a microservice as something that could be rewritten in two weeks, a rule of thumb that makes sense for his particular context.

Another somewhat trite answer I can give is *small enough and no smaller*. When speaking at conferences, I nearly always ask the question *who has a system that is too big and that you'd like to break down*? Nearly everyone raises their hands. We seem to have a very good sense of what is too big, and so it could be argued that once a piece of code no longer *feels* too big, it's probably small enough.

A strong factor in helping us answer *how small?* is how well the service aligns to team structures. If the codebase is too big to be managed by a small team, looking to break it down is very sensible. We'll talk more about organizational alignment later on.

When it comes to how small is small enough, I like to think in these terms: the smaller the service, the more you maximize the benefits and downsides of microservice architecture. As you get smaller, the benefits around interdependence increase. But so too does some of the complexity that emerges from having more and more moving parts, something that we will explore throughout this book. As you get better at handling this complexity, you can strive for smaller and smaller services.

Autonomous

Our microservice is a separate entity. It might be deployed as an isolated service on a platform as a service (PAAS), or it might be its own operating system process. We try to avoid packing multiple services onto the same machine, although the definition of *machine* in today's world is pretty hazy! As we'll discuss later, although this isolation can add some overhead, the resulting simplicity makes our distributed system much easier to reason about, and newer technologies are able to mitigate many of the challenges associated with this form of deployment.

All communication between the services themselves are via network calls, to enforce separation between the services and avoid the perils of tight coupling.

These services need to be able to change independently of each other, and be deployed by themselves without requiring consumers to change. We need to think about what our services should expose, and what they should allow to be hidden. If there is too much sharing, our consuming services become coupled to our internal representations. This decreases our autonomy, as it requires additional coordination with consumers when making changes.

Our service exposes an application programming interface (API), and collaborating services communicate with us via those APIs. We also need to think about what technology is appropriate to ensure that this itself doesn't couple consumers. This may mean picking technology-agnostic APIs to ensure that we don't constrain technology choices. We'll come back time and again to the importance of good, decoupled APIs throughout this book.

Without decoupling, everything breaks down for us. The golden rule: can you make a change to a service and deploy it by itself without changing anything else? If the answer is no, then many of the advantages we discuss throughout this book will be hard for you to achieve.

To do decoupling well, you'll need to model your services right and get the APIs right. I'll be talking about that a lot.

Key Benefits

The benefits of microservices are many and varied. Many of these benefits can be laid at the door of any distributed system. Microservices, however, tend to achieve these benefits to a greater degree primarily due to how far they take the concepts behind distributed systems and service-oriented architecture.

Technology Heterogeneity

With a system composed of multiple, collaborating services, we can decide to use different technologies inside each one. This allows us to pick the right tool for each job, rather than having to select a more standardized, one-size-fits-all approach that often ends up being the lowest common denominator.

If one part of our system needs to improve its performance, we might decide to use a different technology stack that is better able to achieve the performance levels required. We may also decide that how we store our data needs to change for different parts of our system. For example, for a social network, we might store our users' interactions in a graph-oriented database to reflect the highly interconnected nature of a social graph, but perhaps the posts the users make could be stored in a document-oriented data store, giving rise to a heterogeneous architecture like the one shown in Figure 1-1.

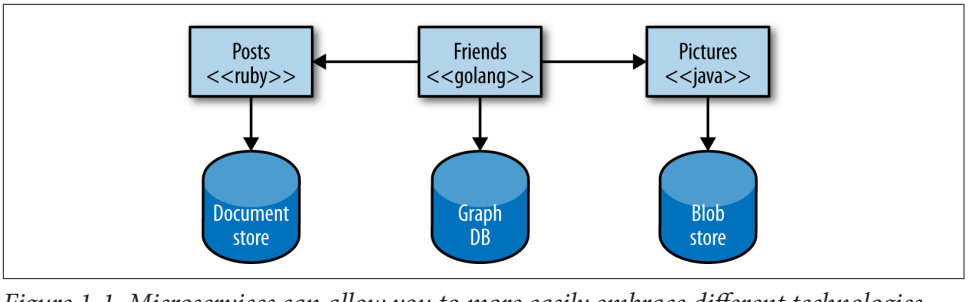


Figure 1-1. Microservices can allow you to more easily embrace different technologies

With microservices, we are also able to adopt technology more quickly, and understand how new advancements may help us. One of the biggest barriers to trying out and adopting new technology is the risks associated with it. With a monolithic application, if I want to try a new programming language, database, or framework, any change will impact a large amount of my system. With a system consisting of multiple services, I have multiple new places in which to try out a new piece of technology. I can pick a service that is perhaps lowest risk and use the technology there, knowing that I can limit any potential negative impact. Many organizations find this ability to more quickly absorb new technologies to be a real advantage for them.

Embracing multiple technologies doesn't come without an overhead, of course. Some organizations choose to place some constraints on language choices. Netflix and Twitter, for example, mostly use the Java Virtual Machine (JVM) as a platform, as they have a very good understanding of the reliability and performance of that system. They also develop libraries and tooling for the JVM that make operating at scale much easier, but make it more difficult for non-Java-based services or clients. But neither Twitter nor Netflix use only one technology stack for all jobs, either. Another counterpoint to concerns about mixing in different technologies is the size. If I really can rewrite my microservice in two weeks, you may well mitigate the risks of embracing new technology.

As you'll find throughout this book, just like many things concerning microservices, it's all about finding the right balance. We'll discuss how to make technology choices in (cross-ref to come), which focuses on evolutionary architecture; and in Chapter 2, which deals with integration, you'll learn how to ensure that your services can evolve their technology independently of each other without undue coupling.

Resilience

A key concept in resilience engineering is the bulkhead. If one component of a system fails, but that failure doesn't cascade, you can isolate the problem and the rest of the system can carry on working. Service boundaries become your obvious bulkheads. In a monolithic service, if the service fails, everything stops working. With a monolithic system, we can run on multiple machines to reduce our chance of failure, but with microservices, we can build systems that handle the total failure of services and degrade functionality accordingly.

We do need to be careful, however. To ensure our microservice systems can properly embrace this improved resilience, we need to understand the new sources of failure that distributed systems have to deal with. Networks can and will fail, as will machines. We need to know how to handle this, and what impact (if any) it should have on the end user of our software.

We'll talk more about better handling resilience, and how to handle failure modes, in Chapter 3.

Scaling

With a large, monolithic service, we have to scale everything together. One small part of our overall system is constrained in performance, but if that behavior is locked up in a giant monolithic application, we have to handle scaling everything as a piece. With smaller services, we can just scale those services that need scaling, allowing us to run other parts of the system on smaller, less powerful hardware, like in Figure 1-2.

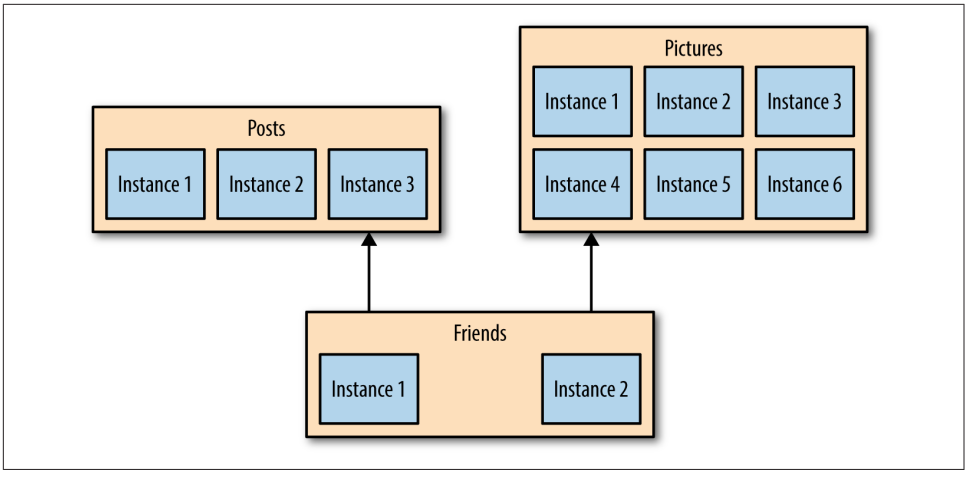


Figure 1-2. You can target scaling at just those microservices that need it

Gilt, an online fashion retailer, adopted microservices for this exact reason. Starting in 2007 with a monolithic Rails application, by 2009 Gilt's system was unable to cope with the load being placed on it. By splitting out core parts of its system, Gilt was better able to deal with its traffic spikes, and today has over 450 microservices, each one running on multiple separate machines.

When embracing on-demand provisioning systems like those provided by Amazon Web Services, we can even apply this scaling on demand for those pieces that need it. This allows us to control our costs more effectively. It's not often that an architectural approach can be so closely correlated to an almost immediate cost savings.

Ease of Deployment

A one-line change to a million-line-long monolithic application requires the whole application to be deployed in order to release the change. That could be a large-impact, high-risk deployment. In practice, large-impact, high-risk deployments end up happening infrequently due to understandable fear. Unfortunately, this means that our changes build up and build up between releases, until the new version of our application hitting production has masses of changes. And the bigger the delta between releases, the higher the risk that we'll get something wrong!

With microservices, we can make a change to a single service and deploy it independently of the rest of the system. This allows us to get our code deployed faster. If a problem does occur, it can be isolated quickly to an individual service, making fast rollback easy to achieve. It also means we can get our new functionality out to customers faster. This is one of the main reasons why organizations like Amazon and Netflix use these architectures—to ensure they remove as many impediments as possible to getting software out the door.

The technology in this space has changed greatly in the last couple of years, and we'll be looking more deeply into the topic of deployment in a microservice world in (cross-ref to come).

Organizational Alignment

Many of us have experienced the problems associated with large teams and large codebases. These problems can be exacerbated when the team is distributed. We also know that smaller teams working on smaller codebases tend to be more productive.

Microservices allow us to better align our architecture to our organization, helping us minimize the number of people working on any one codebase to hit the sweet spot of team size and productivity. We can also shift ownership of services between teams to try to keep people working on one service colocated. We will go into much more detail on this topic when we discuss Conway's law in (cross-ref to come).

Composability

One of the key promises of distributed systems and service-oriented architectures is that we open up opportunities for reuse of functionality. With microservices, we allow for our functionality to be consumed in different ways for different purposes. This can be especially important when we think about how our consumers use our software. Gone is the time when we could think narrowly about either our desktop website or mobile application. Now we need to think of the myriad ways that we might want to weave together capabilities for the Web, native application, mobile web, tablet app, or wearable device. As organizations move away from thinking in terms of narrow channels to more holistic concepts of customer engagement, we need architectures that can keep up.

With microservices, think of us opening up seams in our system that are addressable by outside parties. As circumstances change, we can build things in different ways. With a monolithic application, I often have one coarse-grained seam that can be used from the outside. If I want to break that up to get something more useful, I'll need a hammer! In (cross-ref to come), I'll discuss ways for you to break apart existing monolithic systems, and hopefully change them into some reusable, re-composable microservices.

Optimizing for Replaceability

If you work at a medium-size or bigger organization, chances are you are aware of some big, nasty legacy system sitting in the corner. The one no one wants to touch. The one that is vital to how your company runs, but that happens to be written in some odd Fortran variant and runs only on hardware that reached end of life 25 years ago. Why hasn't it been replaced? You know why: it's too big and risky a job.

With our individual services being small in size, the cost to replace them with a better implementation, or even delete them altogether, is much easier to manage. How often have you deleted more than a hundred lines of code in a single day and not worried too much about it? With microservices often being of similar size, the barriers to rewriting or removing services entirely are very low.

Teams using microservice approaches are comfortable with completely rewriting services when required, and just killing a service when it is no longer needed. When a codebase is just a few hundred lines long, it is difficult for people to become emotionally attached to it, and the cost of replacing it is pretty small.

What About Service-Oriented Architecture?

Service-oriented architecture (SOA) is a design approach where multiple services collaborate to provide some end set of capabilities. A service here typically means a completely separate operating system process. Communication between these services occurs via calls across a network rather than method calls within a process boundary.

SOA emerged as an approach to combat the challenges of the large monolithic applications. It is an approach that aims to promote the reusability of software; two or more end-user applications, for example, could both use the same services. It aims to make it easier to maintain or rewrite software, as theoretically we can replace one service with another without anyone knowing, as long as the semantics of the service don't change too much.

SOA at its heart is a very sensible idea. However, despite many efforts, there is a lack of good consensus on how to do SOA *well*. In my opinion, much of the industry has failed to look holistically enough at the problem and present a compelling alternative to the narrative set out by various vendors in this space.

Many of the problems laid at the door of SOA are actually problems with things like communication protocols (e.g., SOAP), vendor middleware, a lack of guidance about service granularity, or the wrong guidance on picking places to split your system. We'll tackle each of these in turn throughout the rest of the book. A cynic might suggest that vendors co-opted (and in some cases drove) the SOA movement as a way to sell more products, and those selfsame products in the end undermined the goal of SOA.

Much of the conventional wisdom around SOA doesn't help you understand how to split something big into something small. It doesn't talk about how big is too big. It doesn't talk enough about real-world, practical ways to ensure that services do not become overly coupled. The number of things that go unsaid is where many of the pitfalls associated with SOA originate.

The microservice approach has emerged from real-world use, taking our better understanding of systems and architecture to do SOA well. So you should instead think of microservices as a specific approach for SOA in the same way that XP or Scrum are specific approaches for Agile software development.

Other Decompositional Techniques

When you get down to it, many of the advantages of a microservice-based architecture come from its granular nature and the fact that it gives you many more choices as to how to solve problems. But could similar decompositional techniques achieve the same benefits?

Shared Libraries

A very standard decompositional technique that is built into virtually any language is breaking down a codebase into multiple libraries. These libraries may be provided by third parties, or created in your own organization.

Libraries give you a way to share functionality between teams and services. I might create a set of useful collection utilities, for example, or perhaps a statistics library that can be reused.

Teams can organize themselves around these libraries, and the libraries themselves can be reused. But there are some drawbacks.

First, you lose true technology heterogeneity. The library typically has to be in the same language, or at the very least run on the same platform. Second, the ease with which you can scale parts of your system independently from each other is curtailed. Next, unless you're using dynamically linked libraries, you cannot deploy a new library without redeploying the entire process, so your ability to deploy changes in isolation is reduced. And perhaps the kicker is that you lack the obvious seams around which to erect architectural safety measures to ensure system resiliency.

Shared libraries do have their place. You'll find yourself creating code for common tasks that aren't specific to your business domain that you want to reuse across the organization, which is an obvious candidate for becoming a reusable library. You do need to be careful, though. Shared code used to communicate between services can become a point of coupling, something we'll discuss in Chapter 2.

Services can and should make heavy use of third-party libraries to reuse common code. But they don't get us all the way there.

Modules

Some languages provide their own modular decomposition techniques that go beyond simple libraries. They allow some lifecycle management of the modules, such that they can be deployed into a running process, allowing you to make changes without taking the whole process down.

The Open Source Gateway Initiative (OSGI) is worth calling out as one technology-specific approach to modular decomposition. Java itself doesn't have a true concept of modules, and we'll have to wait at least until Java 9 to see this added to the language. OSGI, which emerged as a framework to allow plug-ins to be installed in the Eclipse Java IDE, is now used as a way to retrofit a module concept in Java via a library.

The problem with OSGI is that it is trying to enforce things like module lifecycle management without enough support in the language itself. This results in more work having to be done by module authors to deliver on proper module isolation. Within a process boundary, it is also much easier to fall into the trap of making modules overly coupled to each other, causing all sorts of problems. My own experience with OSGI, which is matched by that of colleagues in the industry, is that even with good teams it is easy for OSGI to become a much bigger source of complexity than its benefits warrant.

Erlang follows a different approach, in which modules are baked into the language runtime. Thus, Erlang is a very mature approach to modular decomposition. Erlang modules can be stopped, restarted, and upgraded without issue. Erlang even supports running more than one version of the module at a given time, allowing for more graceful module upgrading.

The capabilities of Erlang's modules are impressive indeed, but even if we are lucky enough to use a platform with these capabilities, we still have the same shortcomings as we do with normal shared libraries. We are strictly limited in our ability to use new technologies, limited in how we can scale independently, can drift toward integration techniques that are overly coupling, and lack seams for architectural safety measures.

There is one final observation worth sharing. Technically, it should be possible to create well-factored, independent modules within a single monolithic process. And yet we rarely see this happen. The modules themselves soon become tightly coupled with the rest of the code, surrendering one of their key benefits. Having a process boundary separation does enforce clean hygiene in this respect (or at least makes it harder to do the wrong thing!). I wouldn't suggest that this should be the main driver for process separation, of course, but it is interesting that the promises of modular separation within process boundaries rarely deliver in the real world.

So while modular decomposition within a process boundary may be something you want to do as well as decomposing your system into services, by itself it won't help

solve everything. If you are a pure Erlang shop, the quality of Erlang's module implementation may get you a very long way, but I suspect many of you are not in that situation. For the rest of us, we should see modules as offering the same sorts of benefits as shared libraries.

No Silver Bullet

Before we finish, I should call out that microservices are no free lunch or silver bullet, and make for a bad choice as a golden hammer. They have all the associated complexities of distributed systems, and while we have learned a lot about how to manage distributed systems well (which we'll discuss throughout the book) it is still hard. If you're coming from a monolithic system point of view, you'll have to get much better at handling deployment, testing, and monitoring to unlock the benefits we've covered so far. You'll also need to think differently about how you scale your systems and ensure that they are resilient. Don't also be surprised if things like distributed transactions or CAP theorem start giving you headaches, either!

Every company, organization, and system is different. A number of factors will play into whether or not microservices are right for you, and how aggressive you can be in adopting them. Throughout each chapter in this book I'll attempt to give you guidance highlighting the potential pitfalls, which should help you chart a steady path.

Summary

Hopefully by now you know what a microservice is, what makes it different from other compositional techniques, and what some of the key advantages are. In each of the following chapters we will go into more detail on how to achieve these benefits and how to avoid some of the common pitfalls.

There are a number of topics to cover, but we need to start somewhere. One of the main challenges that microservices introduce is a shift in the role of those who often guide the evolution of our systems: the architects. We'll look next at some different approaches to this role that can ensure we get the most out of this new architecture.

CHAPTER 2

Integration

Getting integration right is the single most important aspect of the technology associated with microservices in my opinion. Do it well, and your microservices retain their autonomy, allowing you to change and release them independent of the whole. Get it wrong, and disaster awaits. Hopefully once you've read this chapter you'll learn how to avoid some of the biggest pitfalls that have plagued other attempts at SOA and could yet await you in your journey to microservices.

Looking for the Ideal Integration Technology

There is a bewildering array of options out there for how one microservice can talk to another. But which is the right one: SOAP? XML-RPC? REST? Protocol buffers? We'll dive into those in a moment, but before we do, let's think about what we want out of whatever technology we pick.

Avoid Breaking Changes

Every now and then, we may make a change that requires our consumers to also change. We'll discuss how to handle this later, but we want to pick technology that ensures this happens as rarely as possible. For example, if a microservice adds new fields to a piece of data it sends out, existing consumers shouldn't be impacted.

Keep Your APIs Technology-Agnostic

If you have been in the IT industry for more than 15 minutes, you don't need me to tell you that we work in a space that is changing rapidly. The one certainty *is* change. New tools, frameworks, and languages are coming out all the time, implementing new ideas that can help us work faster and more effectively. Right now, you might be a .NET shop. But what about in a year from now, or five years from now? What if you

want to experiment with an alternative technology stack that might make you more productive?

I am a big fan of keeping my options open, which is why I am such a fan of microservices. It is also why I think it is very important to ensure that you keep the APIs used for communication between microservices technology-agnostic. This means avoiding integration technology that dictates what technology stacks we can use to implement our microservices.

Make Your Service Simple for Consumers

We want to make it easy for consumers to use our service. Having a beautifully factored microservice doesn't count for much if the cost of using it as a consumer is sky high! So let's think about what makes it easy for consumers to use our wonderful new service. Ideally, we'd like to allow our clients full freedom in their technology choice, but on the other hand, providing a client library can ease adoption. Often, however, such libraries are incompatible with other things we want to achieve. For example, we might use client libraries to make it easy for consumers, but this can come at the cost of increased coupling.

Hide Internal Implementation Detail

We don't want our consumers to be bound to our internal implementation. This leads to increased coupling. This means that if we want to change something inside our microservice, we can break our consumers by requiring them to also change. That increases the cost of change—the exact result we are trying to avoid. It also means we are less likely to want to make a change for fear of having to upgrade our consumers, which can lead to increased technical debt within the service. So any technology that pushes us to expose internal representation detail should be avoided.

Interfacing with Customers

Now that we've got a few guidelines that can help us select a good technology to use for integration between services, let's look at some of the most common options out there and try to work out which one works best for us. To help us think this through, let's pick a real-world example from MusicCorp.

Customer creation at first glance could be considered a simple set of CRUD operations, but for most systems it is more complex than that. Enrolling a new customer may need to kick off additional processes, like setting up financial payments or sending out welcome emails. And when we change or delete a customer, other business processes might get triggered as well.

So with that in mind, we should look at some different ways in which we might want to work with customers in our MusicCorp system.

The Shared Database

By far the most common form of integration that I or any of my colleagues see in the industry is database (DB) integration. In this world, if other services want information from a service, they reach into the database. And if they want to change it, they reach into the database! This is really simple when you first think about it, and is probably the fastest form of integration to start with—which probably explains its popularity.

Figure 2-1 shows our registration UI, which creates customers by performing SQL operations directly on the database. It also shows our call center application that views and edits customer data by running SQL on the database. And the warehouse updates information about customer orders by querying the database. This is a common enough pattern, but it's one fraught with difficulties.

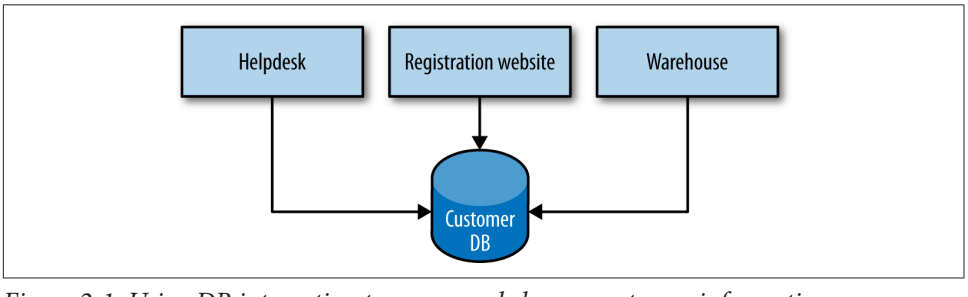


Figure 2-1. Using DB integration to access and change customer information

First, we are allowing external parties to view and bind to internal implementation details. The data structures I store in the DB are fair game to all; they are shared in their entirety with all other parties with access to the database. If I decide to change my schema to better represent my data, or make my system easier to maintain, I can break my consumers. The DB is effectively a very large, shared API that is also quite brittle. If I want to change the logic associated with, say, how the helpdesk manages customers and this requires a change to the database, I have to be extremely careful that I don't break parts of the schema used by other services. This situation normally results in requiring a large amount of regression testing.

Second, my consumers are tied to a specific technology choice. Perhaps right now it makes sense to store customers in a relational database, so my consumers use an appropriate (potentially DB-specific) driver to talk to it. What if over time we realize we would be better off storing data in a nonrelational database? Can it make that decision? So consumers are intimately tied to the implementation of the customer service. As we discussed earlier, we really want to ensure that implementation detail is hidden from consumers to allow our service a level of autonomy in terms of how it changes its internals over time. Goodbye, loose coupling.

Finally, let's think about behavior for a moment. There is going to be logic associated with how a customer is changed. Where is that logic? If consumers are directly manipulating the DB, then they have to own the associated logic. The logic to perform the same sorts of manipulation to a customer may now be spread among multiple consumers. If the warehouse, registration UI, and call center UI all need to edit customer information, I need to fix a bug or change the behavior in three different places, and deploy those changes too. Goodbye, cohesion.

Remember when we talked about the core principles behind good microservices? Strong cohesion and loose coupling—with database integration, we lose both things. Database integration makes it easy for services to share data, but does nothing about *sharing behavior*. Our internal representation is exposed over the wire to our consumers, and it can be very difficult to avoid making breaking changes, which inevitably leads to a fear of any change at all. Avoid at (nearly) all costs.

For the rest of the chapter, we'll explore different styles of integration that involve collaborating services, which themselves hide their own internal representations.

Synchronous Versus Asynchronous

Before we start diving into the specifics of different technology choices, we should discuss one of the most important decisions we can make in terms of how services collaborate. Should communication be synchronous or asynchronous? This fundamental choice inevitably guides us toward certain implementation detail.

With synchronous communication, a call is made to a remote server, which blocks until the operation completes. With asynchronous communication, the caller doesn't wait for the operation to complete before returning, and may not even care whether or not the operation completes at all.

Synchronous communication can be easier to reason about. We know when things have completed successfully or not. Asynchronous communication can be very useful for long-running jobs, where keeping a connection open for a long period of time between the client and server is impractical. It also works very well when you need low latency, where blocking a call while waiting for the result can slow things down. Due to the nature of mobile networks and devices, firing off requests and assuming things have worked (unless told otherwise) can ensure that the UI remains responsive even if the network is highly laggy. On the flipside, the technology to handle asynchronous communication can be a bit more involved, as we'll discuss shortly.

These two different modes of communication can enable two different idiomatic styles of collaboration: *request/response* or *event-based*. With request/response, a client initiates a request and waits for the response. This model clearly aligns well to synchronous communication, but can work for asynchronous communication too. I

might kick off an operation and register a callback, asking the server to let me know when my operation has completed.

With an event-based collaboration, we invert things. Instead of a client initiating requests asking for things to be done, it instead says *this thing happened* and expects other parties to know what to do. We never tell anyone else what to do. Event-based systems by their nature are asynchronous. The smarts are more evenly distributed—that is, the business logic is not centralized into core brains, but instead pushed out more evenly to the various collaborators. Event-based collaboration is also highly decoupled. The client that emits an event doesn't have any way of knowing who or what will react to it, which also means that you can add new subscribers to these events without the client ever needing to know.

So are there any other drivers that might push us to pick one style over another? One important factor to consider is how well these styles are suited for solving an often-complex problem: how do we handle processes that span service boundaries and may be long running?

Orchestration Versus Choreography

As we start to model more and more complex logic, we have to deal with the problem of managing business processes that stretch across the boundary of individual services. And with microservices, we'll hit this limit sooner than usual. Let's take an example from MusicCorp, and look at what happens when we create a customer:

1. A new record is created in the loyalty points bank for the customer.
2. Our postal system sends out a welcome pack.
3. We send a welcome email to the customer.

This is very easy to model conceptually as a flowchart, as we do in Figure 2-2.

When it comes to actually implementing this flow, there are two styles of architecture we could follow. With orchestration, we rely on a central brain to guide and drive the process, much like the conductor in an orchestra. With choreography, we inform each part of the system of its job, and let it work out the details, like dancers all finding their way and reacting to others around them in a ballet.

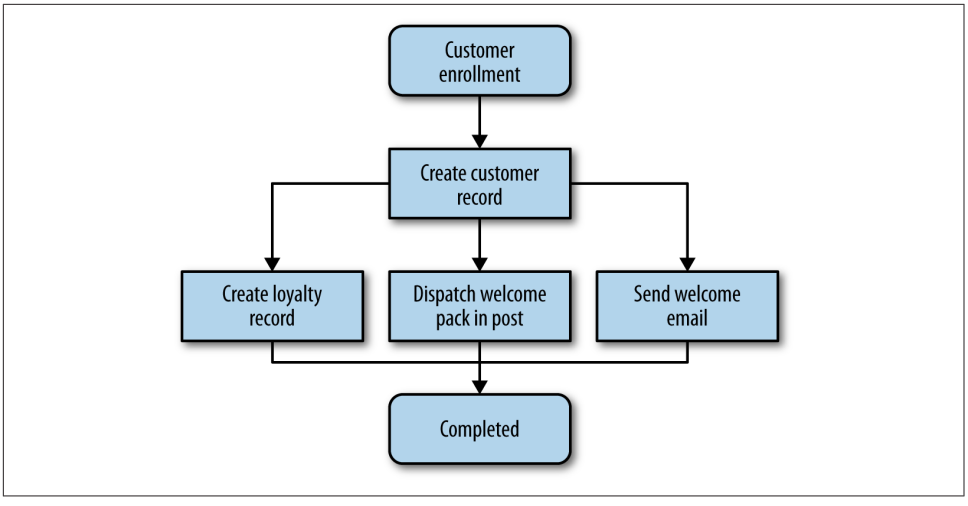


Figure 2-2. The process for creating a new customer

Let's think about what an orchestration solution would look like for this flow. Here, probably the simplest thing to do would be to have our customer service act as the central brain. On creation, it talks to the loyalty points bank, email service, and postal service as we see in Figure 2-3, through a series of request/response calls. The customer service itself can then track where a customer is in this process. It can check to see if the customer's account has been set up, or the email sent, or the post delivered. We get to take the flowchart in Figure 2-2 and model it directly into code. We could even use tooling that implements this for us, perhaps using an appropriate rules engine. Commercial tools exist for this very purpose in the form of business process modeling software. Assuming we use synchronous request/response, we could even know if each stage has worked.

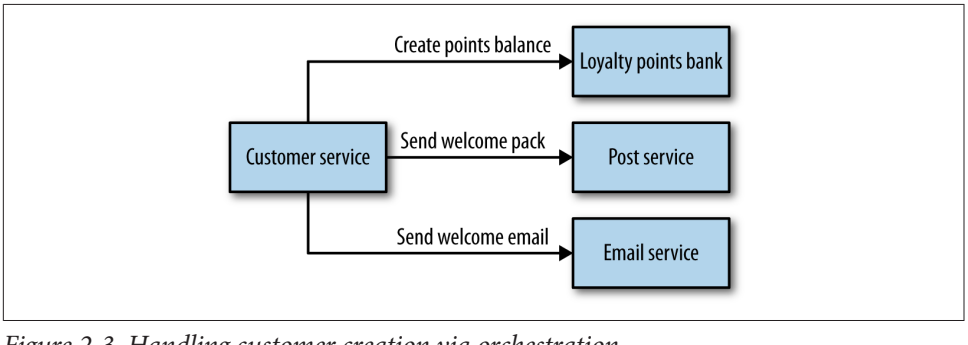


Figure 2-3. Handling customer creation via orchestration

The downside to this orchestration approach is that the customer service can become too much of a central governing authority. It can become the hub in the middle of a

web, and a central point where logic starts to live. I have seen this approach result in a small number of smart *god* services telling anemic CRUD-based services what to do.

With a choreographed approach, we could instead just have the customer service emit an event in an asyychronous manner, saying *Customer created*. The email service, postal service, and loyalty points bank then just subscribe to these events and react accordingly, as in Figure 2-4. This approach is significantly more decoupled. If some other service needed to reach to the creation of a customer, it just needs to subscribe to the events and do its job when needed. The downside is that the explicit view of the business process we see in Figure 2-2 is now only implicitly reflected in our system.

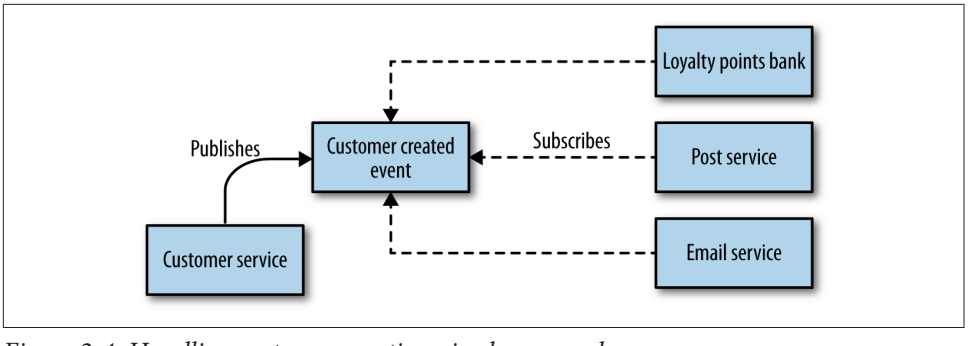


Figure 2-4. Handling customer creation via choreography

This means additional work is needed to ensure that you can monitor and track that the right things have happened. For example, would you know if the loyalty points bank had a bug and for some reason didn't set up the correct account? One approach I like for dealing with this is to build a monitoring system that explicitly matches the view of the business process in Figure 2-2, but then tracks what each of the services does as independent entities, letting you see odd exceptions mapped onto the more explicit process flow. The flowchart we saw earlier isn't the driving force, but just one lens through which we can see how the system is behaving.

In general, I have found that systems that tend more toward the choreographed approach are more loosely coupled, and are more flexible and amenable to change. You do need to do extra work to monitor and track the processes across system boundaries, however. I have found most heavily orchestrated implementations to be extremely brittle, with a higher cost of change. With that in mind, I strongly prefer aiming for a choreographed system, where each service is smart enough to understand its role in the whole dance.

There are quite a few factors to unpack here. Synchronous calls are simpler, and we get to know if things worked straightaway. If we like the semantics of request/response but are dealing with longer-lived processes, we could just initiate asynchro-

nous requests and wait for callbacks. On the other hand, asynchronous event collaboration helps us adopt a choreographed approach, which can yield significantly more decoupled services—something we want to strive for to ensure our services are independently releasable.

We are, of course, free to mix and match. Some technologies will fit more naturally into one style or another. We do, however, need to appreciate some of the different technical implementation details that will further help us make the right call.

To start with, let's look at two technologies that fit well when we are considering request/response: remote procedure call (RPC) and REpresentational State Transfer (REST).

Remote Procedure Calls

Remote procedure call refers to the technique of making a local call and having it execute on a remote service somewhere. There are a number of different types of RPC technology out there. Some of this technology relies on having an interface definition (SOAP, Thrift, protocol buffers). The use of a separate interface definition can make it easier to generate client and server stubs for different technology stacks, so, for example, I could have a Java server exposing a SOAP interface, and a .NET client generated from the Web Service Definition Language (WSDL) definition of the interface. Other technology, like Java RMI, calls for a tighter coupling between the client and server, requiring that both use the same underlying technology but avoid the need for a shared interface definition. All these technologies, however, have the same, core characteristic in that they make a local call look like a remote call.

Many of these technologies are binary in nature, like Java RMI, Thrift, or protocol buffers, while SOAP uses XML for its message formats. Some implementations are tied to a specific networking protocol (like SOAP, which makes nominal use of HTTP), whereas others might allow you to use different types of networking protocols, which themselves can provide additional features. For example, TCP offers guarantees about delivery, whereas UDP doesn't but has a much lower overhead. This can allow you to use different networking technology for different use cases.

Those RPC implementations that allow you to generate client and server stubs help you get started very, very fast. I can be sending content over a network boundary in no time at all. This is often one of the main selling points of RPC: its ease of use. The fact that I can just make a normal method call and theoretically ignore the rest is a huge boon.

Some RPC implementations, though, do come with some downsides that can cause issues. These issues aren't always apparent initially, but nonetheless they can be severe enough to outweigh the benefits of being so easy to get up and running quickly.

Technology Coupling

Some RPC mechanisms, like Java RMI, are heavily tied to a specific platform, which can limit which technology can be used in the client and server. Thrift and protocol buffers have an impressive amount of support for alternative languages, which can reduce this downside somewhat, but be aware that sometimes RPC technology comes with restrictions on interoperability.

In a way, this technology coupling can be a form of exposing internal technical implementation details. For example, the use of RMI ties not only the client to the JVM, but the server too.

Local Calls Are Not Like Remote Calls

The core idea of RPC is to hide the complexity of a remote call. Many implementations of RPC, though, hide too much. The drive in some forms of RPC to make remote method calls look like local method calls hides the fact that these two things are very different. I can make large numbers of local, in-process calls without worrying overly about the performance. With RPC, though, the cost of marshalling and unmarshalling payloads can be significant, not to mention the time taken to send things over the network. This means you need to think differently about API design for remote interfaces versus local interfaces. Just taking a local API and trying to make it a service boundary without any more thought is likely to get you in trouble. In some of the worst examples, developers may be using remote calls without knowing it if the abstraction is overly opaque.

You need to think about the network itself. Famously, the first of the fallacies of distributed computing is "The network is reliable". Networks aren't reliable. They can and will fail, even if your client and the server you are speaking to are fine. They can fail fast, they can fail slow, and they can even malform your packets. You should assume that your networks are plagued with malevolent entities ready to unleash their ire on a whim. Therefore, the failure modes you can expect are different. A failure could be caused by the remote server returning an error, or by you making a bad call. Can you tell the difference, and if so, can you do anything about it? And what do you do when the remote server just starts responding slowly? We'll cover this topic when we talk about resiliency in Chapter 3.

Brittleness

Some of the most popular implementations of RPC can lead to some nasty forms of brittleness, Java's RMI being a very good example. Let's consider a very simple Java interface that we have decided to make a remote API for our customer service. Example 2-1 declares the methods we are going to expose remotely. Java RMI then generates the client and server stubs for our method.

Example 2-1. Defining a service endpoint using Java RMI

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface CustomerRemote extends Remote {
  public Customer findCustomer(String id) throws RemoteException;

  public Customer createCustomer(String firstname, String surname, String emailAddress)
      throws RemoteException;
}
```

In this interface, `findCustomer` takes the first name, surname, and email address. What happens if we decide to allow the `Customer` object to also be created with just an email address? We could add a new method at this point pretty easily, like so:

```
...
public Customer createCustomer(String emailAddress) throws RemoteException;
...
```

The problem is that now we need to regenerate the client stubs too. Clients that want to consume the new method need the new stubs, and depending on the nature of the changes to the specification, consumers that don't need the new method may also need to have their stubs upgraded too. This is manageable, of course, but to a point. The reality is that changes like this are fairly common. RPC endpoints often end up having a large number of methods for different ways of creating or interacting with objects. This is due in part to the fact that we are still thinking of these remote calls as local ones.

There is another sort of brittleness, though. Let's take a look at what our `Customer` object looks like:

```
public class Customer implements Serializable {
  private String firstName;
  private String surname;
  private String emailAddress;
  private String age;
}
```

Now, what if it turns out that although we expose the `age` field in our `Customer` objects, none of our consumers ever use it? We decide we want to remove this field. But if the server implementation removes `age` from its definition of this type, and we don't do the same to all the consumers, then even though they never used the field, the code associated with deserializing the `Customer` object on the consumer side will break. To roll out this change, I would have to deploy both a new server and clients at the same time. This is a key challenge with any RPC mechanism that promotes the use of binary stub generation: you don't get to separate client and server deployments. If you use this technology, lock-step releases may be in your future.

Similar problems occur if I want to restructure the `Customer` object even if I didn't remove fields—for example, if I wanted to encapsulate `firstName` and `surname` into a new `naming` type to make it easier to manage. I could, of course, fix this by passing around dictionary types as the parameters of my calls, but at that point, I lose many of the benefits of the generated stubs because I'll still have to manually match and extract the fields I want.

In practice, objects used as part of binary serialization across the wire can be thought of as *expand-only* types. This brittleness results in the types being exposed over the wire and becoming a mass of fields, some of which are no longer used but can't be safely removed.

Is RPC Terrible?

Despite its shortcomings, I wouldn't go so far as to call RPC terrible. Some of the common implementations that I have encountered can lead to the sorts of problems I have outlined here. Due to the challenges of using RMI, I would certainly give that technology a wide berth. Many operations fall quite nicely into the RPC-based model, and more modern mechanisms like protocol buffers or Thrift mitigate some of sins of the past by avoiding the need for lock-step releases of client and server code.

Just be aware of some of the potential pitfalls associated with RPC if you're going to pick this model. Don't abstract your remote calls to the point where the network is completely hidden, and ensure that you can evolve the server interface without having to insist on lock-step upgrades for clients. Finding the right balance for your client code is important, for example. Make sure your clients aren't oblivious to the fact that a network call is going to be made. Client libraries are often used in the context of RPC, and if not structured right they can be problematic. We'll talk more about them shortly.

Compared to database integration, RPC is certainly an improvement when we think about options for request/response collaboration. But there's another option to consider.

REST

REpresentational State Transfer (REST) is an architectural style inspired by the Web. There are many principles and constraints behind the REST style, but we are going to focus on those that really help us when we face integration challenges in a microservices world, and when we're looking for an alternative style to RPC for our service interfaces.

Most important is the concept of resources. You can think of a resource as a thing that the service itself knows about, like a `Customer`. The server creates different representations of this `Customer` on request. How a resource is shown externally is com-

pletely decoupled from how it is stored internally. A client might ask for a JSON representation of a `Customer`, for example, even if it is stored in a completely different format. Once a client has a representation of this `Customer`, it can then make requests to change it, and the server may or may not comply with them.

There are many different styles of REST, and I touch only briefly on them here. I strongly recommend you take a look at the Richardson Maturity Model, where the different styles of REST are compared.

REST itself doesn't really talk about underlying protocols, although it is most commonly used over HTTP. I have seen implementations of REST using very different protocols before, such as serial or USB, although this can require a lot of work. Some of the features that HTTP gives us as part of the specification, such as verbs, make implementing REST over HTTP easier, whereas with other protocols you'll have to handle these features yourself.

REST and HTTP

HTTP itself defines some useful capabilities that play very well with the REST style. For example, the HTTP verbs (e.g., GET, POST, and PUT) already have well-understood meanings in the HTTP specification as to how they should work with resources. The REST architectural style actually tells us that methods should behave the same way on all resources, and the HTTP specification happens to define a bunch of methods we can use. GET retrieves a resource in an idempotent way, and POST creates a new resource. This means we can avoid lots of different `createCustomer` or `editCustomer` methods. Instead, we can simply POST a customer representation to request that the server create a new resource, and initiate a GET request to retrieve a representation of a resource. Conceptually, there is one *endpoint* in the form of a `Customer` resource in these cases, and the operations we can carry out upon it are baked into the HTTP protocol.

HTTP also brings a large ecosystem of supporting tools and technology. We get to use HTTP caching proxies like Varnish and load balancers like mod_proxy, and many monitoring tools already have lots of support for HTTP out of the box. These building blocks allow us to handle large volumes of HTTP traffic and route them smartly, in a fairly transparent way. We also get to use all the available security controls with HTTP to secure our communications. From basic auth to client certs, the HTTP ecosystem gives us lots of tools to make the security process easier, and we'll explore that topic more in (cross-ref to come). That said, to get these benefits, you have to use HTTP well. Use it badly, and it can be as insecure and hard to scale as any other technology out there. Use it right, though, and you get a lot of help.

Note that HTTP can be used to implement RPC too. SOAP, for example, gets routed over HTTP, but unfortunately uses very little of the specification. Verbs are ignored,

as are simple things like HTTP error codes. All too often, it seems, the existing, well-understood standards and technology are ignored in favor of new standards that can only be implemented using brand-new technology—conveniently provided by the same companies that help design the new standards in the first place!

Hypermedia As the Engine of Application State

Another principle introduced in REST that can help us avoid the coupling between client and server is the concept of *hypermedia as the engine of application state* (often abbreviated as HATEOAS, and boy, did it need an abbreviation). This is fairly dense wording and a fairly interesting concept, so let's break it down a bit.

Hypermedia is a concept whereby a piece of content contains links to various other pieces of content in a variety of formats (e.g., text, images, sounds). This should be pretty familiar to you, as it's what the average web page does: you follow links, which are a form of hypermedia controls, to see related content. The idea behind HATEOAS is that clients should perform interactions (potentially leading to state transitions) with the server via these links to other resources. It doesn't need to know where exactly customers live on the server by knowing which URI to hit; instead, the client looks for and navigates links to find what it needs.

This is a bit of an odd concept, so let's first step back and consider how people interact with a web page, which we've already established is rich with hypermedia controls.

Think of the Amazon.com shopping site. The location of the shopping cart has changed over time. The graphic has changed. The link has changed. But as humans we are smart enough to still see a shopping cart, know what it is, and interact with it. We have an understanding of what a shopping cart means, even if the exact form and underlying control used to represent it has changed. We know that if we want to view the cart, this is the control we want to interact with. This is why web pages can change incrementally over time. As long as these implicit contracts between the customer and the website are still met, changes don't need to be breaking changes.

With hypermedia controls, we are trying to achieve the same level of *smarts* for our electronic consumers. Let's look at a hypermedia control that we might have for MusicCorp. We've accessed a resource representing a catalog entry for a given album in Example 2-2. Along with information about the album, we see a number of hypermedia controls.

Example 2-2. Hypermedia controls used on an album listing

```
<album>
  <name>Give Blood</name>
  <link rel="/artist" href="/artist/theBrakes" /> ❶
  <description>
    Awesome, short, brutish, funny and loud. Must buy!
```

```
  </description>
  <link rel="/instantpurchase" href="/instantPurchase/1234" /> ❷
</album>
```

❶ This hypermedia control shows us where to find information about the artist.

❷ And if we want to purchase the album, we now know where to go.

In this document, we have two hypermedia controls. The client reading such a document needs to know that a control with a relation of `artist` is where it needs to navigate to get information about the artist, and that `instantpurchase` is part of the protocol used to purchase the album. The client has to understand the semantics of the API in much the same way as a human being needs to understand that on a shopping website the cart is where the items to be purchased will be.

As a client, I don't need to know which URI scheme to access to *buy* the album, I just need to access the resource, find the buy control, and navigate to that. The buy control could change location, the URI could change, or the site could even send me to another service altogether, and as a client I wouldn't care. This gives us a huge amount of decoupling between the client and server.

We are greatly abstracted from the underlying detail here. We could completely change the implementation of how the control is presented as long as the client can still find a control that matches its understanding of the protocol, in the same way that a shopping cart control might go from being a simple link to a more complex JavaScript control. We are also free to add new controls to the document, perhaps representing new state transitions that we can perform on the resource in question. We would end up breaking our consumers only if we fundamentally changed the semantics of one of the controls so it behaved very differently, or if we removed a control altogether.

Using these controls to decouple the client and server yields significant benefits over time that greatly offset the small increase in the time it takes to get these protocols up and running. By following the links, the client gets to progressively discover the API, which can be a really handy capability when we are implementing new clients.

One of the downsides is that this navigation of controls can be quite chatty, as the client needs to follow links to find the operation it wants to perform. Ultimately, this is a trade-off. I would suggest you start with having your clients navigate these controls first, then optimize later if necessary. Remember that we have a large amount of help out of the box by using HTTP, which we discussed earlier. The evils of premature optimization have been well documented before, so I don't need to expand upon them here. Also note that a lot of these approaches were developed to create distributed hypertext systems, and not all of them fit! Sometimes you'll find yourself just wanting good old-fashioned RPC.

Personally, I am a fan of using links to allow consumers to navigate API endpoints. The benefits of progressive discovery of the API and reduced coupling can be significant. However, it is clear that not everyone is sold, as I don't see it being used anywhere near as much as I would like. I think a large part of this is that there is some initial upfront work required, but the rewards often come later.

JSON, XML, or Something Else?

The use of standard textual formats gives clients a lot of flexibility as to how they consume resources, and REST over HTTP lets us use a variety of formats. The examples I have given so far used XML, but at this stage, JSON is a much more popular content type for services that work over HTTP.

The fact that JSON is a much simpler format means that consumption is also easier. Some proponents also cite its relative compactness when compared to XML as another winning factor, although this isn't often a real-world issue.

JSON does have some downsides, though. XML defines the `link` control we used earlier as a hypermedia control. The JSON standard doesn't define anything similar, so in-house styles are frequently used to shoe-horn this concept in. The Hypertext Application Language (HAL) attempts to fix this by defining some common standards for hyperlinking for JSON (and XML too, although arguably XML needs less help). If you follow the HAL standard, you can use tools like the web-based HAL browser for exploring hypermedia controls, which can make the task of creating a client much easier.

We aren't limited to these two formats, of course. We can send pretty much anything over HTTP if we want, even binary. I am seeing more and more people just using HTML as a format instead of XML. For some interfaces, the HTML can do double duty as a UI and an API, although there are pitfalls to be avoided here, as the interactions of a human and a computer are quite different! But it is certainly an attractive idea. There are lots of HTML parsers out there, after all.

Personally, though, I am still a fan of XML. Some of the tool support is better. For example, if I want to extract only certain parts of the payload (a technique we'll discuss more in "Versioning" on page 36) I can use XPATH, which is a well-understood standard with lots of tool support, or even CSS selectors, which many find even easier. With JSON, I have JSONPATH, but this is not widely supported. I find it odd that people pick JSON because it is nice and lightweight, then try and push concepts into it like hypermedia controls that already exist in XML. I accept, though, that I am probably in the minority here and that JSON is the format of choice for most people!

Beware Too Much Convenience

As REST has become more popular, so too have the frameworks that help us create RESTFul web services. However, some of these tools trade off too much in terms of short-term gain for long-term pain; in trying to get you going fast, they can encourage some bad behaviors. For example, some frameworks actually make it very easy to simply take database representations of objects, deserialize them into in-process objects, and then directly expose these externally. I remember at a conference seeing this demonstrated using Spring Boot and cited as a major advantage. The inherent coupling that this setup promotes will in most cases cause far more pain than the effort required to properly decouple these concepts.

There is a more general problem at play here. How we decide to store our data, and how we expose it to our consumers, can easily dominate our thinking. One pattern I saw used effectively by one of our teams was to delay the implementation of proper persistence for the microservice, until the interface had stabilized enough. For an interim period, entities were just persisted in a file on local disk, which is obviously not a suitable long-term solution. This ensured that how the consumers wanted to use the service drove the design and implementation decisions. The rationale given, which was borne out in the results, was that it is too easy for the way we store domain entities in a backing store to overtly influence the models we send over the wire to collaborators. One of the downsides with this approach is that we are deferring the work required to wire up our data store. I think for new service boundaries, however, this is an acceptable trade-off.

Downsides to REST Over HTTP

In terms of ease of consumption, you cannot easily generate a client stub for your REST over HTTP application protocol like you can with RPC. Sure, the fact that HTTP is being used means that you get to take advantage of all the excellent HTTP client libraries out there, but if you want to implement and use hypermedia controls as a client you are pretty much on your own. Personally, I think client libraries could do much better at this than they do, and they are certainly better now than in the past, but I have seen this apparent increased complexity result in people backsliding into smuggling RPC over HTTP or building shared client libraries. Shared code between client and server can be very dangerous, as we'll discuss in "DRY and the Perils of Code Reuse in a Microservice World" on page 33.

A more minor point is that some web server frameworks don't actually support all the HTTP verbs well. That means that it might be easy for you to create a handler for GET or POST requests, but you may have to jump through hoops to get PUT or DELETE requests to work. Proper REST frameworks like Jersey don't have this problem, and you can normally work around this, but if you are locked into certain framework choices this might limit what style of REST you can use.

Performance may also be an issue. REST over HTTP payloads can actually be more compact than SOAP because it supports alternative formats like JSON or even binary, but it will still be nowhere near as lean a binary protocol as Thrift might be. The overhead of HTTP for each request may also be a concern for low-latency requirements.

HTTP, while it can be suited well to large volumes of traffic, isn't great for low-latency communications when compared to alternative protocols that are built on top of Transmission Control Protocol (TCP) or other networking technology. Despite the name, WebSockets, for example, has very little to do with the Web. After the initial HTTP handshake, it's just a TCP connection between client and server, but it can be a much more efficient way for you to stream data for a browser. If this is something you're interested in, note that you aren't really using much of HTTP, let alone anything to do with REST.

For server-to-server communications, if extremely low latency or small message size is important, HTTP communications in general may not be a good idea. You may need to pick different underlying protocols, like User Datagram Protocol (UDP), to achieve the performance you want, and many RPC frameworks will quite happily run on top of networking protocols other than TCP.

Consumption of the payloads themselves requires more work than is provided by some RPC implementations that support advanced serialization and deserialization mechanisms. These can become a coupling point in their own right between client and server, as implementing tolerant readers is a nontrivial activity (we'll discuss this shortly), but from the point of view of getting up and running, they can be very attractive.

Despite these disadvantages, REST over HTTP is a sensible default choice for service-to-service interactions. If you want to know more, I recommend *REST in Practice* (O'Reilly), which covers the topic of REST over HTTP in depth.

Implementing Asynchronous Event-Based Collaboration

We've talked for a bit about some technologoies that can help us implement request/response patterns. What about event-based, asynchronous communication?

Technology Choices

There are two main parts we need to consider here: a way for our microservices to emit events, and a way for our consumers to find out those events have happened.

Traditionally, message brokers like RabbitMQ try to handle both problems. Producers use an API to publish an event to the broker. The broker handles subscriptions, allowing consumers to be informed when an event arrives. These brokers can even handle the state of consumers, for example by helping keep track of what messages

they have seen before. These systems are normally designed to be scalable and resilient, but that doesn't come for free. It can add complexity to the development process, because it is another system you may need to run to develop and test your services. Additional machines and expertise may also be required to keep this infrastructure up and running. But once it does, it can be an incredibly effective way to implement loosely coupled, event-driven architectures. In general, I'm a fan.

Do be wary, though, about the world of middleware, of which the message broker is just a small part. Queues in and of themselves are perfectly sensible, useful things. However, vendors tend to want to package lots of software with them, which can lead to more and more smarts being pushed into the middleware, as evidenced by things like the Enterprise Service Bus. Make sure you know what you're getting: keep your middleware dumb, and keep the smarts in the endpoints.

Another approach is to try to use HTTP as a way of propagating events. ATOM is a REST-compliant specification that defines semantics (among other things) for publishing feeds of resources. Many client libraries exist that allow us to create and consume these feeds. So our customer service could just publish an event to such a feed when our customer service changes. Our consumers just poll the feed, looking for changes. On one hand, the fact that we can reuse the existing ATOM specification and any associated libraries is useful, and we know that HTTP handles scale very well. However, HTTP is not good at low latency (where some message brokers excel), and we still need to deal with the fact that the consumers need to keep track of what messages they have seen and manage their own polling schedule.

I have seen people spend an age implementing more and more of the behaviors that you get out of the box with an appropriate message broker to make ATOM work for some use cases. For example, the Competing Consumer pattern describes a method whereby you bring up multiple worker instances to compete for messages, which works well for scaling up the number of workers to handle a list of independent jobs. However, we want to avoid the case where two or more workers see the same message, as we'll end up doing the same task more than we need to. With a message broker, a standard queue will handle this. With ATOM, we now need to manage our own shared state among all the workers to try to reduce the chances of reproducing effort.

If you already have a good, resilient message broker available to you, consider using it to handle publishing and subscribing to events. But if you don't already have one, give ATOM a look, but be aware of the sunk-cost fallacy. If you find yourself wanting more and more of the support that a message broker gives you, at a certain point you might want to change your approach.

In terms of what we actually send over these asynchronous protocols, the same considerations apply as with synchronous communication. If you are currently happy with encoding requests and responses using JSON, stick with it.

Complexities of Asynchronous Architectures

Some of this asynchronous stuff seems fun, right? Event-driven architectures seem to lead to significantly more decoupled, scalable systems. And they can. But these programming styles do lead to an increase in complexity. This isn't just the complexity required to manage publishing and subscribing to messages as we just discussed, but also in the other problems we might face. For example, when considering long-running async request/response, we have to think about what to do when the response comes back. Does it come back to the same node that initiated the request? If so, what if that node is down? If not, do I need to store information somewhere so I can react accordingly? Short-lived async can be easier to manage if you've got the right APIs, but even so, it is a different way of thinking for programmers who are accustomed to intra-process synchronous message calls.

Time for a cautionary tale. Back in 2006, I was working on building a pricing system for a bank. We would look at market events, and work out which items in a portfolio needed to be repriced. Once we determined the list of things to work through, we put these all onto a message queue. We were making use of a grid to create a pool of pricing workers, allowing us to scale up and down the pricing farm on request. These workers used the competing consumer pattern, each one gobbling messages as fast as possible until there was nothing left to process.

The system was up and running, and we were feeling rather smug. One day, though, just after we pushed a release out, we hit a nasty problem. Our workers kept dying. And dying. And dying.

Eventually, we tracked down the problem. A bug had crept in whereby a certain type of pricing request would cause a worker to crash. We were using a transacted queue: as the worker died, its lock on the request timed out, and the pricing request was put back on the queue—only for another worker to pick it up and die. This was a classic example of what Martin Fowler calls a catastrophic failover.

Aside from the bug itself, we'd failed to specify a maximum retry limit for the job on the queue. We fixed the bug itself, and also configured a maximum retry. But we also realized we needed a way to view, and potentially replay, these bad messages. We ended up having to implement a message hospital (or dead letter queue), where messages got sent if they failed. We also created a UI to view those messages and retry them if needed. These sorts of problems aren't immediately obvious if you are only familiar with synchronous point-to-point communication.

The associated complexity with event-driven architectures and asynchronous programming in general leads me to believe that you should be cautious in how eagerly you start adopting these ideas. Ensure you have good monitoring in place, and strongly consider the use of correlation IDs, which allow you to trace requests across process boundaries, as we'll cover in depth in (cross-ref to come).

I also strongly recommend *Enterprise Integration Patterns* (Addison-Wesley), which contains a lot more detail on the different programming patterns that you may need to consider in this space.

Services as State Machines

Whether you choose to become a REST ninja, or stick with an RPC-based mechanism like SOAP, the core concept of the service as a state machine is powerful. We've spoken before (probably ad nauseum by this point) about our services being fashioned around bounded contexts. Our customer microservice *owns* all logic associated with behavior in this context.

When a consumer wants to change a customer, it sends an appropriate request to the customer service. The customer service, based on its logic, gets to decide if it accepts that request or not. Our customer service controls all lifecycle events associated with the customer itself. We want to avoid dumb, anemic services that are little more than CRUD wrappers. If the decision about what changes are allowed to be made to a customer leak out of the customer service itself, we are losing cohesion.

Having the lifecycle of key domain concepts explicitly modeled like this is pretty powerful. Not only do we have one place to deal with collisions of state (e.g., someone trying to update a customer that has already been removed), but we also have a place to attach behavior based on those state changes.

I still think that REST over HTTP makes for a much more sensible integration technology than many others, but whatever you pick, keep this idea in mind.

Reactive Extensions

Reactive extensions, often shortened to Rx, are a mechanism to compose the results of multiple calls together and run operations on them. The calls themselves could be blocking or nonblocking calls. At its heart, Rx inverts traditional flows. Rather than asking for some data, then performing operations on it, you observe the outcome of an operation (or set of operations) and react when something changes. Some implementations of Rx allow you to perform functions on these observables, such as RxJava, which allows traditional functions like `map` or `filter` to be used.

The various Rx implementations have found a very happy home in distributed systems. They allow us to abstract out the details of how calls are made, and reason about things more easily. I observe the result of a call to a downstream service. I don't care if it was a blocking or nonblocking call, I just wait for the response and react. The beauty is that I can compose multiple calls together, making handling concurrent calls to downstream services much easier.

As you find yourself making more service calls, especailly when making multiple calls to perform a single operation, take a look at the reactive extensions for your chosen technology stack. You may be surprised how much simpler your life can become.

DRY and the Perils of Code Reuse in a Microservice World

One of the acronyms we developers hear a lot is DRY: don't repeat yourself. Though its definition is sometimes simplified as trying to avoid duplicating code, DRY more accurately means that we want to avoid duplicating our system *behavior and knowledge*. This is very sensible advice in general. Having lots of lines of code that do the same thing makes your codebase larger than needed, and therefore harder to reason about. When you want to change behavior, and that behavior is duplicated in many parts of your system, it is easy to forget everywhere you need to make a change, which can lead to bugs. So using DRY as a mantra, in general, makes sense.

DRY is what leads us to create code that can be reused. We pull duplicated code into abstractions that we can then call from multiple places. Perhaps we go as far as making a shared library that we can use everywhere! This approach, however, can be deceptively dangerous in a microservice architecture.

One of the things we want to avoid at all costs is overly coupling a microservice and consumers such that any small change to the microservice itself can cause unnecessary changes to the consumer. Sometimes, however, the use of shared code can create this very coupling. For example, at one client we had a library of common domain objects that represented the core entities in use in our system. This library was used by all the services we had. But when a change was made to one of them, all services had to be updated. Our system communicated via message queues, which also had to be drained of their now *invalid* contents, and woe betide you if you forgot.

If your use of shared code ever leaks outside your service boundary, you have introduced a potential form of coupling. Using common code like logging libraries is fine, as they are internal concepts that are invisible to the outside world. RealEstate.com.au makes use of a tailored service template to help bootstrap new service creation. Rather than make this code shared, the company copies it for every new service to ensure that coupling doesn't leak in.

My general rule of thumb: don't violate DRY within a microservice, but be relaxed about violating DRY across all services. The evils of too much coupling between services are far worse than the problems caused by code duplication. There is one specific use case worth exploring further, though.

Client Libraries

I've spoken to more than one team who has insisted that creating client libraries for your services is an essential part of creating services in the first place. The argument

is that this makes it easy to use your service, and avoids the duplication of code required to consume the service itself.

The problem, of course, is that if the same people create both the server API and the client API, there is the danger that logic that should exist on the server starts leaking into the client. I should know: I've done this myself. The more logic that creeps into the client library, the more cohesion starts to break down, and you find yourself having to change multiple clients to roll out fixes to your server. You also limit technology choices, especially if you mandate that the client library has to be used.

A model for client libraries I like is the one for Amazon Web Services (AWS). The underlying SOAP or REST web service calls can be made directly, but everyone ends up using just one of the various software development kits (SDKs) that exist, which provide abstractions over the underlying API. These SDKs, though, are written by the community or AWS people other than those who work on the API itself. This degree of separation seems to work, and avoids some of the pitfalls of client libraries. Part of the reason this works so well is that the client is in charge of when the upgrade happens. If you go down the path of client libraries yourself, make sure this is the case.

Netflix in particular places special emphasis on the client library, but I worry that people view that purely through the lens of avoiding code duplication. In fact, the client libraries used by Netflix are as much (if not more) about ensuring reliability and scalability of their systems. The Netflix client libraries handle service discovery, failure modes, logging, and other aspects that aren't actually about the nature of the service itself. Without these shared clients, it would be hard to ensure that each piece of client/server communications behaved well at the massive scale at which Netflix operates. Their use at Netflix has certainly made it easy to get up and running and increased productivity while also ensuring the system behaves well. However, according to at least one person at Netflix, over time this has led to a degree of coupling between client and server that has been problematic.

If the client library approach is something you're thinking about, it can be important to separate out client code to handle the underlying transport protocol, which can deal with things like service discovery and failure, from things related to the destination service itself. Decide whether or not you are going to insist on the client library being used, or if you'll allow people using different technology stacks to make calls to the underlying API. And finally, make sure that the clients are in charge of when to upgrade their client libraries: we need to ensure we maintain the ability to release our services independently of each other!

Access by Reference

One consideration I want to touch on is how we pass around information about our domain entities. We need to embrace the idea that a microservice will encompass the

lifecycle of our core domain entities, like the `Customer`. We've already talked about the importance of the logic associated with changing this `Customer` being held in the customer service, and that if we want to change it we have to issue a request to the customer service. But it also follows that we should consider the customer service as being the source of truth for `Customer`s.

When we retrieve a given `Customer` resource from the customer service, we get to see what that resource looked like when we made the request. It is possible that after we requested that `Customer` resource, something else has changed it. What we have in effect is a memory of what the `Customer` resource once looked like. The longer we hold on to this memory, the higher the chance that this memory will be false. Of course, if we avoid requesting data more than we need to, our systems can become much more efficient.

Sometimes this memory is good enough. Other times you need to know if it has changed. So whether you decide to pass around a memory of what an entity once looked like, make sure you also include a reference to the original resource so that the new state can be retrieved.

Let's consider the example where we ask the email service to send an email when an order has been shipped. Now we could send in the request to the email service with the customer's email address, name, and order details. However, if the email service is actually queuing up these requests, or pulling them from a queue, things could change in the meantime. It might make more sense to just send a URI for the pass: +Customer+ and `Order` resources, and let the email server go look them up when it is time to send the email.

A great counterpoint to this emerges when we consider event-based collaboration. With events, we're saying *this happened*, but we need to know *what* happened. If we're receiving updates due to a `Customer` resource changing, for example, it could be valuable to us to know what the `Customer` looked like when the event occurred. As long as we also get a reference to the entity itself so we can look up its current state, then we can get the best of both worlds.

There are other trade-offs to be made here, of course, when we're accessing by reference. If we always go to the customer service to look at the information associated with a given `Customer`, the load on the customer service can be too great. If we provide additional information when the resource is retrieved, letting us know at what time the resource was in the given state and perhaps how long we can consider this information to be *fresh*, then we can do a lot with caching to reduce load. HTTP gives us much of this support out of the box with a wide variety of cache controls, some of which we'll discuss in more detail in Chapter 3.

Another problem is that some of our services might not need to know about the whole `Customer` resource, and by insisting that they go look it up we are potentially

increasing coupling. It could be argued, for example, that our email service should be more dumb, and that we should just send it the email address and name of the customer. There isn't a hard-and-fast rule here, but be very wary of passing around data in requests when you don't know its freshness.

Versioning

In every single talk I have ever done about microservices, I get asked *how do you do versioning?* People have the legitimate concern that eventually they will have to make a change to the interface of a service, and they want to understand how to manage that. Let's break down the problem a bit and look at the various steps we can take to handle it.

Defer It for as Long as Possible

The best way to reduce the impact of making breaking changes is to avoid making them in the first place. You can achieve much of this by picking the right integration technology, as we've discussed throughout this chapter. Database integration is a great example of technology that can make it very hard to avoid breaking changes. REST, on the other hand, helps because changes to internal implementation detail are less likely to result in a change to the service interface.

Another key to deferring a breaking change is to encourage good behavior in your clients, and avoid them binding too tightly to your services in the first place. Let's consider our email service, whose job it is to send out emails to our customers from time to time. It gets asked to send an order shipped email to customer with the ID 1234. It goes off and retrieves the customer with that ID, and gets back something like the response shown in Example 2-3.

Example 2-3. Sample response from the customer service

```
<customer>
  <firstname>Sam</firstname>
  <lastname>Newman</lastname>
  <email>sam@magpiebrain.com</email>
  <telephoneNumber>555-1234-5678</telephoneNumber>
</customer>
```

Now to send the email, we need only the `firstname`, `lastname`, and `email` fields. We don't need to know the `telephoneNumber`. We want to simply pull out those fields we care about, and ignore the rest. Some binding technology, especially that used by strongly typed languages, can attempt to bind *all* fields whether the consumer wants them or not. What happens if we realize that no one is using the `telephoneNumber` and we decide to remove it? This could cause consumers to break needlessly.

Likewise, what if we wanted to restructure our `Customer` object to support more details, perhaps adding some further structure as in Example 2-4? The data our email service wants is still there, and still with the same name, but if our code makes very explicit assumptions as to where the `firstname` and `lastname` fields will be stored, then it could break again. In this instance, we could instead use XPath to pull out the fields we care about, allowing us to be ambivalent about where the fields are, as long as we can find them. This pattern—of implementing a reader able to ignore changes we don't care about—is what Martin Fowler calls a Tolerant Reader.

Example 2-4. A restructured Customer resource: the data is all still there, but can our consumers find it?

```
<customer>
  <naming>
    <firstname>Sam</firstname>
    <lastname>Newman</lastname>
    <nickname>Magpiebrain</nickname>
    <fullname>Sam "Magpiebrain" Newman</fullname>
  </naming>
  <email>sam@magpiebrain.com</email>
</customer>
```

The example of a client trying to be as flexible as possible in consuming a service demonstrates Postel's Law (otherwise known as the *robustness principle*), which states: "Be conservative in what you do, be liberal in what you accept from others." The original context for this piece of wisdom was the interaction of devices over networks, where you should expect all sorts of odd things to happen. In the context of our request/response interaction, it can lead us to try our best to allow the service being consumed to change without requiring us to change.

Catch Breaking Changes Early

It's crucial to make sure we pick up changes that will break consumers as soon as possible, because even if we choose the best possible technology, breaks can still happen. I am strongly in favor of using consumer-driven contracts, which we'll cover in (cross-ref to come), to help spot these problems early on. If you're supporting multiple different client libraries, running tests using each library you support against the latest service is another technique that can help. Once you realize you are going to break a consumer, you have the choice to either try to avoid the break altogether or else embrace it and start having the right conversations with the people looking after the consuming services.

Use Semantic Versioning

Wouldn't it be great if as a client you could look just at the version number of a service and know if you can integrate with it? *Semantic versioning* is a specification that allows just that. With semantic versioning, each version number is in the form `MAJOR.MINOR.PATCH`. When the `MAJOR` number increments, it means that backward incompatible changes have been made. When `MINOR` increments, new functionality has been added that should be backward compatible. Finally, a change to `PATCH` states that bug fixes have been made to existing functionality.

To see how useful semantic versioning can be, let's look at a simple use case. Our helpdesk application is built to work against version 1.2.0 of the customer service. If a new feature is added, causing the customer service to change to 1.3.0, our helpdesk application should see no change in behavior and shouldn't be expected to make any changes. We couldn't guarantee that we could work against version 1.1.0 of the customer service, though, as we may rely on functionality added in the 1.2.0 release. We could also expect to have to make changes to our application if a new 2.0.0 release of the customer service comes out.

You may decide to have a semantic version for the service, or even for an individual endpoint on a service if you are coexisting them as detailed in the next section.

This versioning scheme allows us to pack a lot of information and expectations into just three fields. The full specification outlines in very simple terms the expectations clients can have of changes to these numbers, and can simplify the process of communicating about whether changes should impact consumers. Unfortunately, I haven't see this approach used enough in the context of distributed systems.

Coexist Different Endpoints

If we've done all we can to avoid introducing a breaking interface change, our next job is to limit the impact. The thing we want to avoid is forcing consumers to upgrade in lock-step with us, as we always want to maintain the ability to release microservices independently of each other. One approach I have used successfully to handle this is to coexist both the old and new interfaces in the same running service. So if we want to release a breaking change, we deploy a new version of the service that exposes both the old and new versions of the endpoint.

This allows us to get the new microservice out as soon as possible, along with the new interface, but give time for consumers to move over. Once all of the consumers are no longer using the old endpoint, you can remove it along with any associated code, as shown in Figure 2-5.

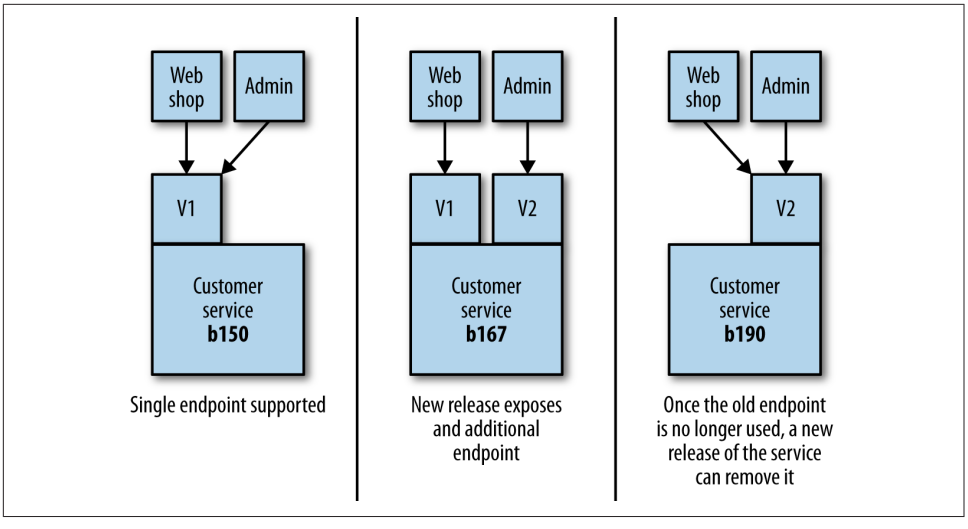


Figure 2-5. Coexisting different endpoint versions allows consumers to migrate gradually

When I last used this approach, we had gotten ourselves into a bit of a mess with the number of consumers we had and the number of breaking changes we had made. This meant that we were actually coexisting three different versions of the endpoint. This is not something I'd recommend! Keeping all the code around and the associated testing required to ensure they all worked was absolutely an additional burden. To make this more manageable, we internally transformed all requests to the V1 endpoint to a V2 request, and then V2 requests to the V3 endpoint. This meant we could clearly delineate what code was going to be retired when the old endpoint(s) died.

This is in effect an example of the expand and contract pattern, which allows us to phase breaking changes in. We *expand* the capabilities we offer, supporting both old and new ways of doing something. Once the old consumers do things in the new way, we *contract* our API, removing the old functionality.

If you are going to coexist endpoints, you need a way for callers to route their requests accordingly. For systems making use of HTTP, I have seen this done with both version numbers in request headers and also in the URI itself—for example, */v1/customer/* or */v2/customer/*. I'm torn as to which approach makes the most sense. On the one hand, I like URIs being opaque to discourage clients from hardcoding URI templates, but on the other hand, this approach does make things very obvious and can simplify request routing.

For RPC, things can be a little trickier. I have handled this with protocol buffers by putting my methods in different namespaces—for example, `v1.createCustomer` and `v2.createCustomer`—but when you are trying to support different versions of the same types being sent over the network, this can become really painful.

Use Multiple Concurrent Service Versions

Another versioning solution often cited is to have different versions of the service live at once, and for older consumers to route their traffic to the older version, with newer versions seeing the new one, as shown in Figure 2-6. This is the approach used sparingly by Netflix in situations where the cost of changing older consumers is too high, especially in rare cases where legacy devices are still tied to older versions of the API. Personally, I am not a fan of this idea, and understand why Netflix uses it rarely. First, if I need to fix an internal bug in my service, I now have to fix and deploy two different sets of services. This would probably mean I have to branch the codebase for my service, and this is always problematic. Second, it means I need smarts to handle directing consumers to the right microservice. This behavior inevitably ends up sitting in middleware somewhere or a bunch of `nginx` scripts, making it harder to reason about the behavior of the system. Finally, consider any persistent state our service might manage. Customers created by either version of the service need to be stored and made visible to all services, no matter which version was used to create the data in the first place. This can be an additional source of complexity.

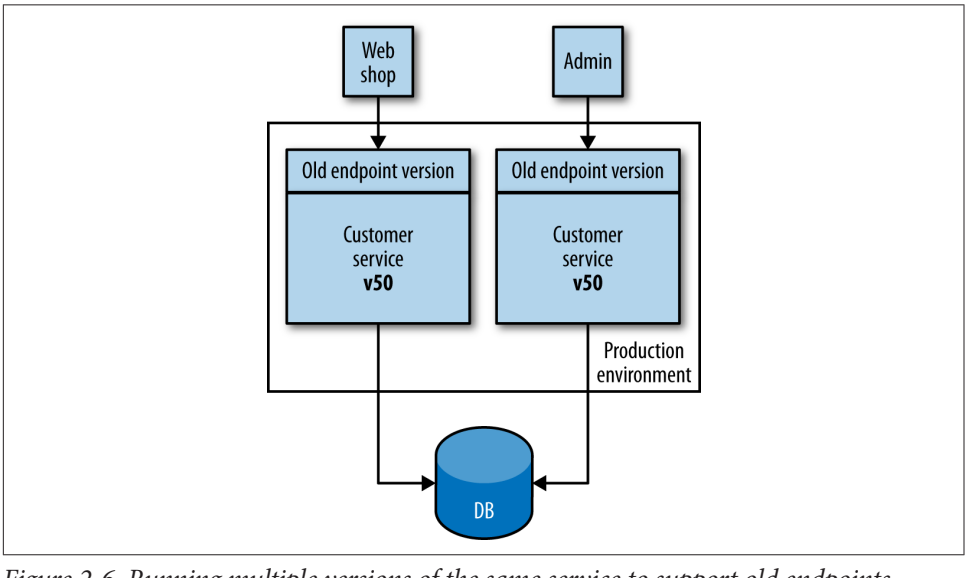


Figure 2-6. Running multiple versions of the same service to support old endpoints

Coexisting concurrent service versions for a short period of time can make perfect sense, especially when you're doing things like blue/green deployments or canary releases (we'll be discussing these patterns more in (cross-ref to come)). In these situations, we may be coexisting versions only for a few minutes or perhaps hours, and normally will have only two different versions of the service present at the same time. The longer it takes for you to get consumers upgraded to the newer version and

released, the more you should look to coexist different endpoints in the same microservice rather than coexist entirely different versions. I remain unconvinced that for the average project this work is worthwhile.

User Interfaces

So far, we haven't really touched on the world of the user interface. A few of you out there might just be providing a cold, hard, clinical API to your customers, but many of us find ourselves wanting to create beautiful, functional user interfaces that will delight our customers. But we really do need to think about them in the context of integration. The user interface, after all, is where we'll be pulling all these microservices together into something that makes sense to our customers.

In the past, when I first started computing, we were mostly talking about big, fat clients that ran on our desktops. I spent many hours with Motif and then Swing trying to make my software as nice to use as possible. Often these systems were just for the creation and manipulation of local files, but many of them had a server-side component. My first job at ThoughtWorks involved creating a Swing-based electronic point-of-sale system that was just part of a large number of moving parts, most of which were on the server.

Then came the Web. We started thinking of our UIs as being *thin* instead, with more logic on the server side. In the beginning, our server-side programs rendered the entire page and sent it to the client browser, which did very little. Any interactions were handled on the server side, via GETs and POSTs triggered by the user clicking on links or filling in forms. Over time, JavaScript became a more popular option to add dynamic behavior to the browser-based UI, and some applications could now be argued to be as *fat* as the old desktop clients.

Toward Digital

Over the last couple of years, organizations have started to move away from thinking that web or mobile should be treated differently; they are instead thinking about digital more holistically. What is the best way for our customers to use the services we offer? And what does that do to our system architecture? The understanding that we cannot predict exactly how a customer might end up interacting with our company has driven adoption of more granular APIs, like those delivered by microservices. By combining the capabilities our services expose in different ways, we can curate different experiences for our customers on their desktop application, mobile device, wearable device, or even in physical form if they visit our brick-and-mortar store.

So think of user interfaces as compositional layers—places where we weave together the various strands of the capabilities we offer. So with that in mind, how do we pull all these strands together?

Constraints

Constraints are the different forms in which our users interact with our system. On a desktop web application, for example, we consider constraints such as what browser visitors are using, or their resolution. But mobile has brought a whole host of new constraints. The way our mobile applications communicate with the server can have an impact. It isn't just about pure bandwidth concerns, where the limitations of mobile networks can play a part. Different sorts of interactions can drain battery life, leading to some cross customers.

The nature of interactions changes, too. I can't easily right-click on a tablet. On a mobile phone, I may want to design my interface to be used mostly one-handed, with most operations being controlled by a thumb. Elsewhere, I might allow people to interact with services via SMS in places where bandwidth is at a premium—the use of SMS as an interface is huge in the global south, for example.

So, although our core services—our core offering—might be the same, we need a way to adapt them for the different constraints that exist for each type of interface. When we look at different styles of user interface composition, we need to ensure that they address this challenge. Let's look at a few models of user interfaces to see how this might be achieved.

API Composition

Assuming that our services already speak XML or JSON to each other via HTTP, an obvious option available to us is to have our user interface interact directly with these APIs, as in Figure 2-7. A web-based UI could use JavaScript GET requests to retrieve data, or POST requests to change it. Even for native mobile applications, initiating HTTP communications is fairly straightforward. The UI would then need to create the various components that make up the interface, handling synchronization of state and the like with the server. If we were using a binary protocol for service-to-service communication, this would be more difficult for web-based clients, but could be fine for native mobile devices.

There are a couple of downsides with this approach. First, we have little ability to tailor the responses for different sorts of devices. For example, when I retrieve a customer record, do I need to pull back all the same data for a mobile shop as I do for a helpdesk application? One solution to this approach is to allow consumers to specify what fields to pull back when they make a request, but this assumes that each service supports this form of interaction.

Another key question: who creates the user interface? The people who look after the services are removed from how their services are surfaced to the users—for example, if another team is creating the UI, we could be drifting back into the bad old days of

layered architecture where making even small changes requires change requests to multiple teams.

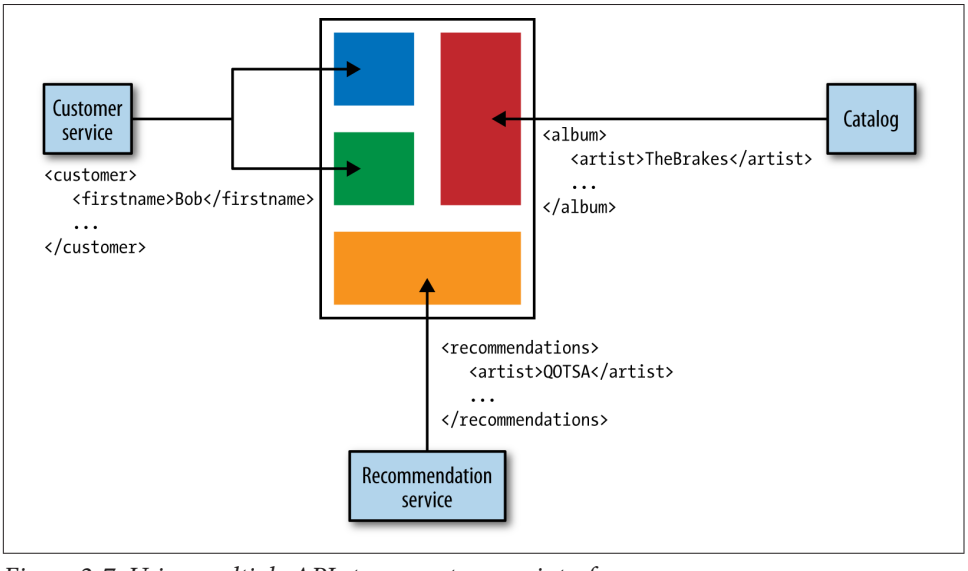


Figure 2-7. Using multiple APIs to present a user interface

This communication could also be fairly chatty. Opening lots of calls directly to services can be quite intensive for mobile devices, and could be a very inefficient use of a customer's mobile plan! Having an API gateway can help here, as you could expose calls that aggregate multiple underlying calls, although that itself can have some downsides that we'll explore shortly.

UI Fragment Composition

Rather than having our UI make API calls and map everything back to UI controls, we could have our services provide parts of the UI directly, and then just pull these fragments in to create a UI, as in Figure 2-8. Imagine, for example, that the recommendation service provides a recommendation widget that is combined with other controls or UI fragments to create an overall UI. It might get rendered as a box on a web page along with other content.

A variation of this approach that can work well is to assemble a series of coarser-grained parts of a UI. So rather than creating small widgets, you are assembling entire panes of a thick client application, or perhaps a set of pages for a website.

These coarser-grained fragments are served up from server-side apps that are in turn making the appropriate API calls. This model works best when the fragments align well to team ownership. For example, perhaps the team that looks after order management in the music shop serves up all the pages associated with order management.

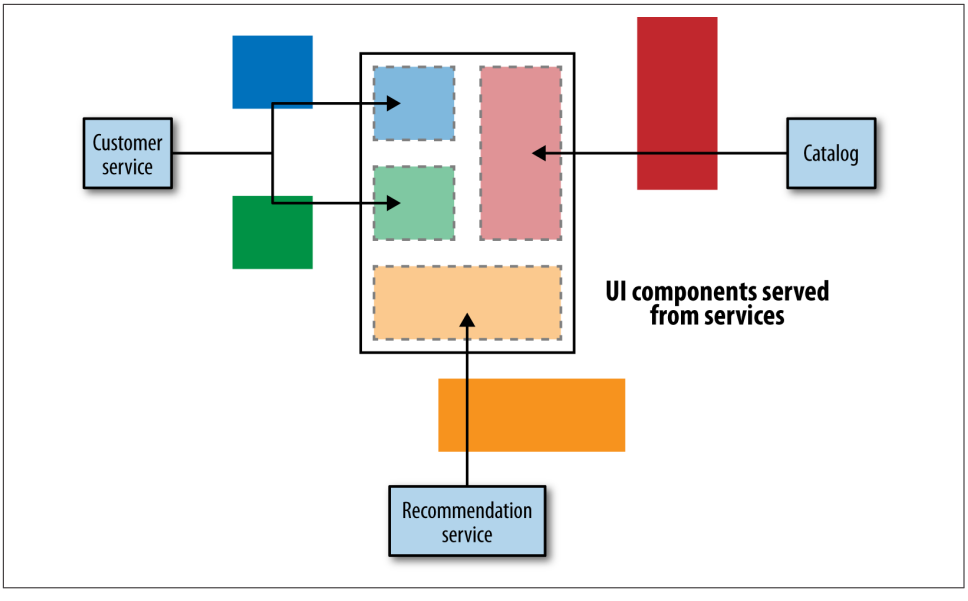


Figure 2-8. Services directly serving up UI components for assembly

You still need some sort of assembly layer to pull these parts together. This could be as simple as some server-side templating, or, where each set of pages comes from a different app, perhaps you'll need some smart URI routing.

One of the key advantages of this approach is that the same team that makes changes to the services can also be in charge of making changes to those parts of the UI. It allows us to get changes out faster. But there are some problems with this approach.

First, ensuring consistency of the user experience is something we need to address. Users want to have a seamless experience, not to feel that different parts of the interface work in different ways, or present a different design language. There are techniques to avoid this problem, though, such as living style guides, where assets like HTML components, CSS, and images can be shared to help give some level of consistency.

Another probem is harder to deal with. What happens with native applications or thick clients? We can't serve up UI components. We could use a hybrid approach and use native applications to serve up HTML components, but this approach has been shown time and again to have downsides. So if you need a native experience, we will have to fall back to an approach where the frontend application makes API calls and handles the UI itself. But even if we consider web-only UIs, we still may want very different treatments for different types of devices. Building responsive components can help, of course.

There is one key problem with this approach that I'm not sure can be solved. Sometimes the capabilities offered by a service do not fit neatly into a widget or a page. Sure, I might want to surface recommendations in a box on a page on our website, but what if I want to weave in dynamic recommendations elsewhere? When I search, I want the type ahead to automatically trigger fresh recommendations, for example. The more cross-cutting a form of interaction is, the less likely this model will fit and the more likely it is that we'll fall back to just making API calls.

Backends for Frontends

A common solution to the problem of chatty interfaces with backend services, or the need to vary content for different types of devices, is to have a server-side aggregation endpoint, or API gateway. This can marshal multiple backend calls, vary and aggregate content if needed for different devices, and serve it up, as we see in Figure 2-9. I've seen this approach lead to disaster when these server-side endpoints become thick layers with too much behavior. They end up getting managed by separate teams, and being another place where logic has to change whenever some functionality changes.

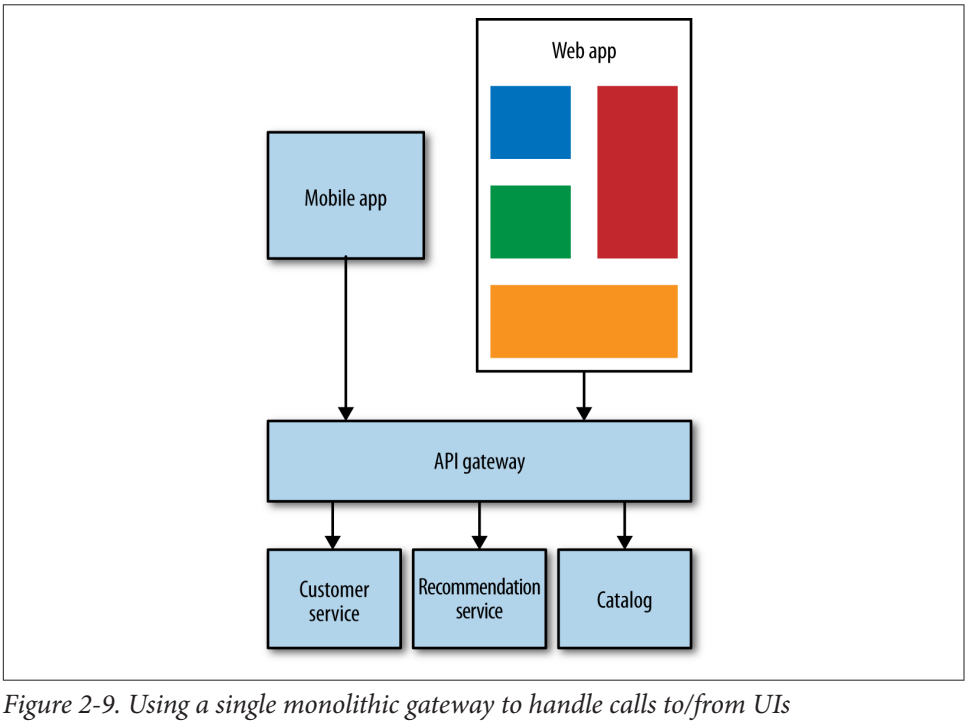


Figure 2-9. Using a single monolithic gateway to handle calls to/from UIs

The problem that can occur is that normally we'll have one giant layer for all our services. This leads to everything being thrown in together, and suddenly we start to lose isolation of our various user interfaces, limiting our ability to release them independently. A model I prefer and that I've seen work well is to restrict the use of these backends to one specific user interface or application, as we see in Figure 2-10.

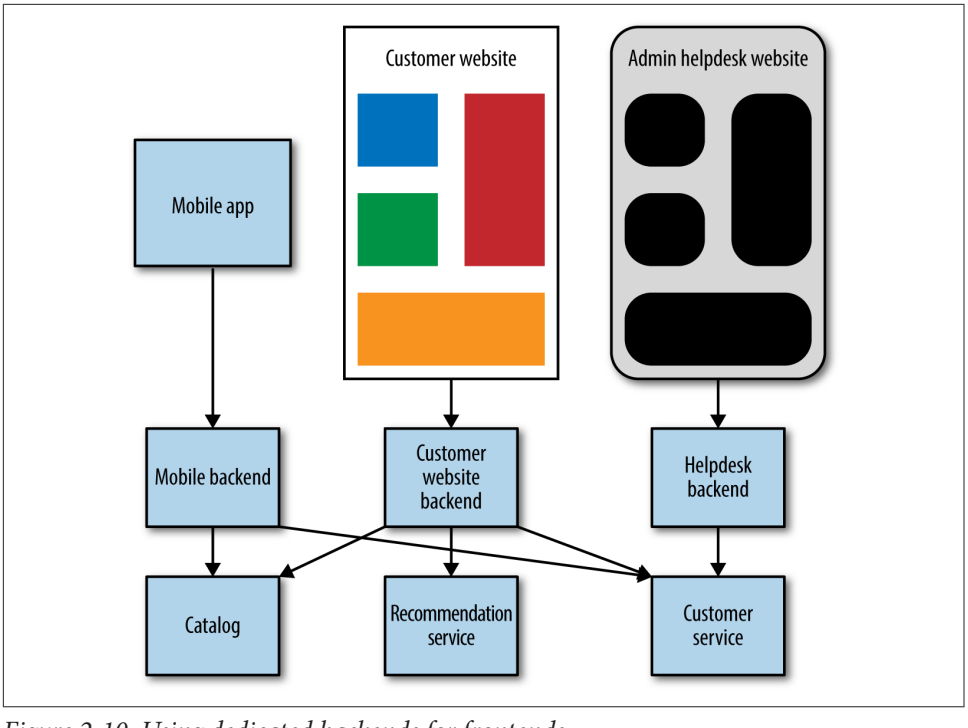


Figure 2-10. Using dedicated backends for frontends

This pattern is sometimes referred to as *backends for frontends (BFFs)*. It allows the team focusing on any given UI to also handle its own server-side components. You can see these backends as parts of the user interface that happen to be embedded in the server. Some types of UI may need a minimal server-side footprint, while others may need a lot more. If you need an API authentication and authorization layer, this can sit between our BFFs and our UIs. We'll explore this more in (cross-ref to come).

The danger with this approach is the same as with any aggregating layer; it can take on logic it shouldn't. The business logic for the various capabilities these backends use should stay in the services themselves. These BFFs should only contain behavior specific to delivering a particular user experience.

A Hybrid Approach

Many of the aforementioned options don't need to be one-size-fits-all. I could see an organization adopting the approach of fragment-based assembly to create a website, but using a backends-for-frontends approach when it comes to its mobile application. The key point is that we need to retain cohesion of the underlying capabilities that we offer our users. We need to ensure that logic associated with ordering music or changing customer details lives inside those services that handle those operations, and doesn't get smeared all over our system. Avoiding the trap of putting too much behavior into any intermediate layers is a tricky balancing act.

Integrating with Third-Party Software

We've looked at approaches to breaking apart existing systems that are under our control, but what about when we can't change the things we talk to? For many valid reasons, the organizations we work for buy custom off-the-shelf software (COTS) or make use of software as a service (SaaS) offerings over which we have little control. So how do we integrate with them sensibly?

If you're reading this book, you probably work at an organization that writes code. You might write software for your own internal purposes or for an external client, or both. Nonetheless, even if you are an organization with the ability to create a significant amount of custom software, you'll still use software products provided by external parties, be they commercial or open source. Why is this?

First, your organization almost certainly has a greater demand for software than can be satisfied internally. Think of all the products you use, from office productivity tools like Excel to operating systems to payroll systems. Creating all of those for your own use would be a mammoth undertaking. Second, and most important, it wouldn't be cost effective! The cost for you to build your own email system, for example, is likely to dwarf the cost of using an existing combination of mail server and client, even if you go for commercial options.

My clients often struggle with the question "Should I build, or should I buy?" In general, the advice I and my colleagues give when having this conversation with the average enterprise organization boils down to "Build if it is unique to what you do, and can be considered a strategic asset; buy if your use of the tool isn't that special."

For example, the average organization would not consider its payroll system to be a strategic asset. People on the whole get paid the same the world over. Likewise, most organizations tend to buy content management systems (CMSes) off the shelf, as their use of such a tool isn't considered something that is key to their business. On the other hand, I was involved early on in rebuilding the *Guardian's* website, and there the decision was made to build a bespoke content management system, as it was core to the newspaper's business.

So the notion that we will occasionally encounter commercial, third-party software is sensible, and to be welcomed. However, many of us end up cursing some of these systems. Why is that?

Lack of Control

One challenge associated with integrating with and extending the capabilities of COTS products like CMS or SaaS tools is that typically many of the technical decisions have been made for you. How do you integrate with the tool? That's a vendor decision. Which programming language can you use to extend the tool? Up to the vendor. Can you store the configuration for the tool in version control, and rebuild from scratch, so as to enable continuous integration of customizations? It depends on choices the vendor makes.

If you are lucky, how easy—or hard—it is to work with the tool from a development point of view has been considered as part of the tool selection process. But even then, you are effectively ceding some level of control to an outside party. The trick is to bring the integration and customization work back on to your terms.

Customization

Many tools that enterprise organizations purchase sell themselves on their ability to be heavily customized *just for you*. Beware! Often, due to the nature of the tool chain you have access to, the cost of customization can be more expensive than building something bespoke from scratch! If you've decided to buy a product but the particular capabilities it provides aren't that special to you, it might make more sense to change how your organization works rather than embark on complex customization.

Content management systems are a great example of this danger. I have worked with multiple CMSes that by design do not support continuous integration, that have terrible APIs, and for which even a minor-point upgrade in the underlying tool can break any customizations you have made.

Salesforce is especially troublesome in this regard. For many years it has pushed its Force.com platform, which requires the use of a programming language, Apex, that exists only within the Force.com ecosystem!

Integration Spaghetti

Another challenge is how you integrate with the tool. As we discussed earlier, thinking carefully about how you integrate between services is important, and ideally you want to standardize on a small number of types of integration. But if one product decides to use a proprietary binary protocol, another some flavor of SOAP, and another XML-RPC, what are you left with? Even worse are the tools that allow you to

reach right inside their underlying data stores, leading to all the same coupling issues we discussed earlier.

On Your Own Terms

COTS and SAAS products absolutely have their place, and it isn't feasible (or sensible) for most of us to build everything from scratch. So how do we resolve these challenges? The key is to move things back on to your own terms.

The core idea here is to do any customizations on a platform you control, and to limit the number of different consumers of the tool itself. To explore this idea in detail, let's look at a couple of examples.

Example: CMS as a service

In my experience, the CMS is one of the most commonly used product that needs to be customized or integrated with. The reason for this is that unless you want a basic static site, the average enterprise organization wants to enrich the functionality of its website with dynamic content like customer records or the latest product offerings. The source of this dynamic content is typically other services inside the organization, which you may have actually built yourself.

The temptation—and often the selling point of the CMS—is that you can customize the CMS to pull in all this special content and display it to the outside world. However, the development environment for the average CMS is *terrible*.

Let's look at what the average CMS specializes in, and what we probably bought it for: content creation and content management. Most CMSes are pretty bad even at doing page layout, typically providing drag-and-drop tools that don't cut the mustard. And even then, you end up needing to have someone who understands HTML and CSS to fine-tune the CMS templates. They tend to be terrible platforms on which to build custom code.

The answer? Front the CMS with your own service that provides the website to the outside world, as shown in Figure 2-11. Treat the CMS as a service whose role is to allow for the creation and retrieval of content. In your own service, you write the code and integrate with services how you want. You have control over scaling the website (many commercial CMSes provide their own proprietary add-ons to handle load), and you can pick the templating system that makes sense.

Most CMSes also provide APIs to allow for content creation, so you also have the ability to front that with your own service façade. For some situations, we've even used such a façade to abstract out the APIs for retrieving content.

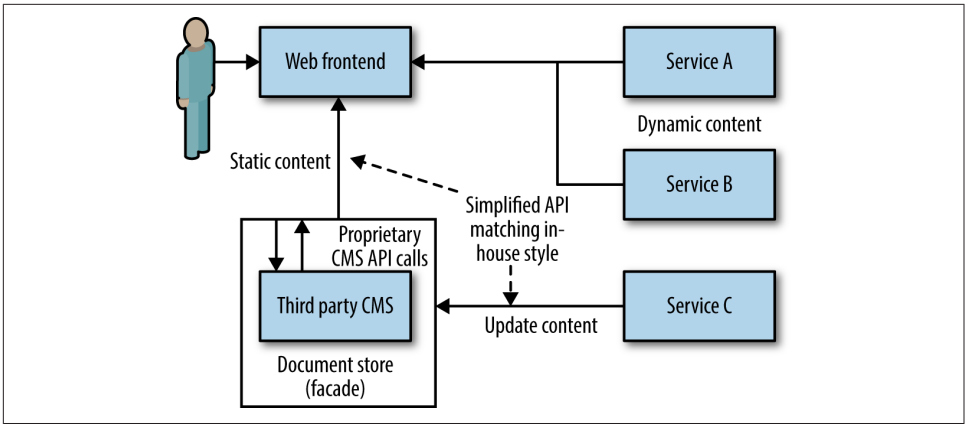


Figure 2-11. Hiding a CMS using your own service

We've used this pattern multiple times across ThoughtWorks in the last few years, and I've done this more than once myself. One notable example was a client that was looking to push out a new website for its products. Initially, it wanted to build the entire solution on the CMS, but it had yet to pick one. We instead suggested this approach, and started development of the fronting website. While waiting for the CMS tool to be selected, we *faked* it by having a web service that just surfaced static content. We ended up going live with the site well before the CMS was selected by using our fake content service in production to surface content to the live site. Later on, we were able to just drop in the eventually selected tool without any change to the fronting application.

Using this approach, we keep the scope of what the CMS does down to a minimum and move customizations onto our own technology stack.

Example: The multirole CRM system

The CRM—or Customer Relationship Management—tool is an often-encountered beast that can instill fear in the heart of even the hardiest architect. This sector, as typified by vendors like Salesforce or SAP, is rife with examples of tools that try to do everything for you. This can lead to the tool itself becoming a single point of failure, and a tangled knot of dependencies. Many implementations of CRM tools I have seen are among the best examples of *adhesive* (as opposed to cohesive) services.

The scope of such a tool typically starts small, but over time it becomes an increasingly important part of how your organization works. The problem is that the direction and choices made around this now-vital system are often made by the tool vendor itself, not by you.

I was involved recently in an exercise to try to wrest some control back. The organization I was working with realized that although it was using the CRM tool for a lot

of things, it wasn't getting the value of the increasing costs associated with the platform. At the same time, multiple internal systems were using the less-than-ideal CRM APIs for integration. We wanted to move the system architecture toward a place where we had services that modeled our businesses domain, and also lay the groundwork for a potential migration.

The first thing we did was identify the core concepts to our domain that the CRM system currently owned. One of these was the concept of a *project*—that is, something to which a member of staff could be assigned. Multiple other systems needed project information. What we did was instead create a project service. This service exposed projects as RESTful resources, and the external systems could move their integration points over to the new, easier-to-work-with service. Internally, the project service was just a façade, hiding the detail of the underlying integration. You can see this in Figure 2-12.

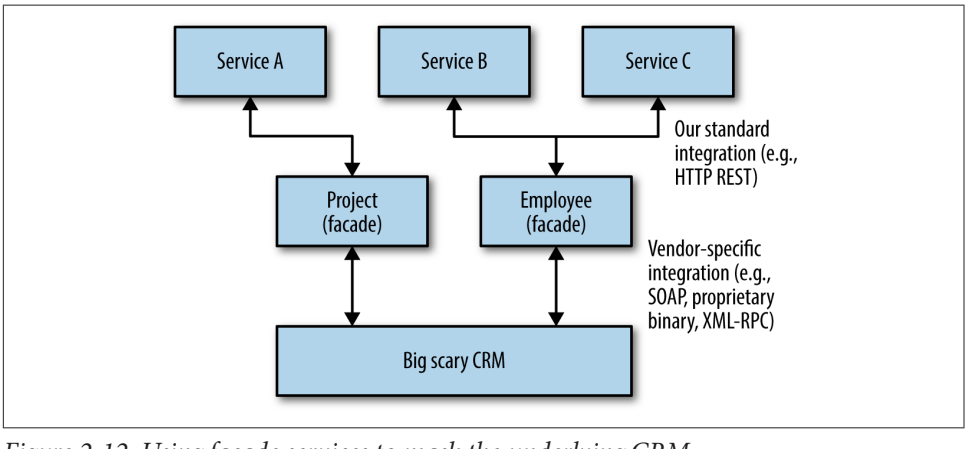


Figure 2-12. Using façade services to mask the underlying CRM

The work, which at the time of this writing was still under way, was to identify other domain concepts that the CRM was handling, and create more façades for them. When the time comes for migration away from the underlying CRM, we could then look at each façade in turn to decide if an internal software solution or something off the shelf could fit the bill.

The Strangler Pattern

When it comes to legacy or even COTS platforms that aren't totally under our control, we also have to deal with what happens when we want to remove them or at least move away from them. A useful pattern here is the Strangler Application Pattern. Much like with our example of fronting the CMS system with our own code, with a strangler you capture and intercept calls to the old system. This allows you to decide if you route these calls to existing, legacy code, or direct them to new code you may

have written. This allows you to replace functionality over time without requiring a big bang rewrite.

When it comes to microservices, rather than having a single monolithic application intercepting all calls to the existing legacy system, you may instead use a series of microservices to perform this interception. Capturing and redirecting the original calls can become more complex in this situation, and you may require the use of a proxy to do this for you.

Summary

We've looked at a number of different options around integration, and I've shared my thoughts on what choices are most likely to ensure our microservices remain as decoupled as possible from their other collaborators:

- Avoid database integration at all costs.
- Understand the trade-offs between REST and RPC, but strongly consider REST as a good starting point for request/response integration.
- Prefer choreography over orchestration.
- Avoid breaking changes and the need to version by understanding Postel's Law and using tolerant readers.
- Think of user interfaces as compositional layers.

We covered quite a lot here, and weren't able to go into depth on all of these topics. Nonetheless, this should be a good foundation to get you going and point you in the right direction if you want to learn more.

We also spent some time looking at how to work with systems that aren't completely under our control in the form of COTS products. It turns out that this description can just as easily apply to software we wrote!

Some of the approaches outlined here apply equally well to *legacy* software, but what if we want to tackle the often-monumental task of bringing these older systems to heel and decomposing them into more usable parts? We'll discuss that in detail in the next chapter.

CHAPTER 3

Microservices at Scale

When you're dealing with nice, small, book-sized examples, everything seems simple. But the real world is a more complex space. What happens when our microservice architectures grow from simpler, more humble beginnings to something more complex? What happens when we have to handle failure of multiple separate services or manage hundreds of services? What are some of the coping patterns when you have more microservices than people? Let's find out.

Failure Is Everywhere

We understand that things can go wrong. Hard disks can fail. Our software can crash. And as anyone who has read the fallacies of distributed computing can tell you, we know that the network is unreliable. We can do our best to try to limit the causes of failure, but at a certain scale, failure becomes inevitable. Hard drives, for example, are more reliable now than ever before, but they'll break eventually. The more hard drives you have, the higher the likelihood of failure for an individual unit; failure becomes a statistical certainty at scale.

Even for those of us not thinking at extreme scale, if we can embrace the possibility of failure we will be better off. For example, if we can handle the failure of a service gracefully, then it follows that we can also do in-place upgrades of a service, as a planned outage is much easier to deal with than an unplanned one.

We can also spend a bit less of our time trying to stop the inevitable, and a bit more of our time dealing with it gracefully. I'm amazed at how many organizations put processes and controls in place to try to stop failure from occurring, but put little to no thought into actually making it easier to recover from failure in the first place.

Baking in the assumption that everything can and will fail leads you to think differently about how you solve problems.

I saw one example of this thinking while spending some time on the Google campus many years ago. In the reception area of one of the buildings in Mountain View was an old rack of machines, there as a sort of exhibit. I noticed a couple of things. First, these servers weren't in server enclosures, they were just bare motherboards slotted into the rack. The main thing I noticed, though, was that the hard drives were attached by velcro. I asked one of the Googlers why that was. "Oh," he said, "the hard drives fail so much we don't want them screwed in. We just rip them out, throw them in the bin, and velcro in a new one."

So let me repeat: at scale, even if you buy the best kit, the most expensive hardware, you cannot avoid the fact that things can and will fail. Therefore, you need to assume failure can happen. If you build this thinking into everything you do, and plan for failure, you can make different trade-offs. If you know your system can handle the fact that a server can and will fail, why bother spending much on it at all? Why not use a bare motherboard with cheaper components (and some velcro) like Google did, rather than worrying too much about the resiliency of a single node?

How Much Is Too Much?

We touched on the topic of cross-functional requirements in (cross-ref to come). Understanding cross-functional requirements is all about considering aspects like durability of data, availability of services, throughput, and acceptable latency of services. Many of the techniques covered in this chapter and elsewhere talk about approaches to implement these requirements, but only you know exactly what the requirements themselves might be.

Having an autoscaling system capable of reacting to increased load or failure of individual nodes might be fantastic, but could be overkill for a reporting system that only needs to run twice a month, where being down for a day or two isn't that big of a deal. Likewise, figuring out how to do blue/green deployments to eliminate downtime of a service might make sense for your online ecommerce system, but for your corporate intranet knowledge base it's probably a step too far.

Knowing how much failure you can tolerate, or how fast your system needs to be, is driven by the users of your system. That in turn will help you understand which techniques will make the most sense for you. That said, your users won't always be able to articulate what the exact requirements are. So you need to ask questions to help extract the right information, and help them understand the relative costs of providing different levels of service.

As I mentioned previously, these cross-functional requirements can vary from service to service, but I would suggest defining some general cross-functionals and then overriding them for particular use cases. When it comes to considering if and how to

scale out your system to better handle load or failure, start by trying to understand the following requirements:

Response time/latency
: How long should various operations take? It can be useful here to measure this with different numbers of users to understand how increasing load will impact the response time. Given the nature of networks, you'll always have outliers, so setting targets for a given percentile of the responses monitored can be useful. The target should also include the number of concurrent connections/users you will expect your software to handle. So you might say, "We expect the website to have a 90th-percentile response time of 2 seconds when handling 200 concurrent connections per second."

Availability
: Can you expect a service to be down? Is this considered a 24/7 service? Some people like to look at periods of acceptable downtime when measuring availability, but how useful is this to someone calling your service? I should either be able to rely on your service responding or not. Measuring periods of downtime is really more useful from a historical reporting angle.

Durability of data
: How much data loss is acceptable? How long should data be kept for? This is highly likely to change on a case-by-case basis. For example, you might choose to keep user session logs for a year or less to save space, but your financial transaction records might need to be kept for many years.

Once you have these requirements in place, you'll want a way to systematically measure them on an ongoing basis. You may decide to make use of performance tests, for example, to ensure your system meets acceptable performance targets, but you'll want to make sure you are monitoring these stats in production as well!

Degrading Functionality

An essential part of building a resilient system, especially when your functionality is spread over a number of different microservices that may be up or down, is the ability to safely degrade functionality. Let's imagine a standard web page on our ecommerce site. To pull together the various parts of that website, we might need several microservices to play a part. One microservice might display the details about the album being offered for sale. Another might show the price and stock level. And we'll probably be showing shopping cart contents too, which may be yet another microservice. Now if one of those services is down, and that results in the whole web page being unavailable, then we have arguably made a system that is less resilient than one that requires only one service to be available.

What we need to do is understand the impact of each outage, and work out how to properly degrade functionality. If the shopping cart service is unavailable, we're probably in a lot of trouble, but we could still show the web page with the listing. Perhaps we just hide the shopping cart or replace it with an icon saying "Be Back Soon!"

With a single, monolithic application, we don't have many decisions to make. System health is binary. But with a microservice architecture, we need to consider a much more nuanced situation. The right thing to do in any situation is often not a technical decision. We might know what is technically possible when the shopping cart is down, but unless we understand the business context we won't understand what action we should be taking. For example, perhaps we close the entire site, still allow people to browse the catalog of items, or replace the part of the UI containing the cart control with a phone number for placing an order. But for every customer-facing interface that uses multiple microservices, or every microservice that depends on multiple downstream collaborators, you need to ask yourself, "What happens if this is down?" and know what to do.

By thinking about the criticality of each of our capabilities in terms of our cross-functional requirements, we'll be much better positioned to know what we can do. Now let's consider some things we can do from a technical point of view to make sure that when failure occurs we can handle it gracefully.

Architectural Safety Measures

There are a few patterns, which collectively I refer to as *architectural safety measures*, that we can make use of to ensure that if something does go wrong, it doesn't cause nasty ripple-out effects. These are points it is essential you understand, and should strongly consider standardizing in your system to ensure that one bad citizen doesn't bring the whole world crashing down around your ears. In a moment, we'll take a look at a few key safety measures you should consider, but before we do, I'd like to share a brief story to outline the sort of thing that can go wrong.

I was a technical lead on a project where we were building an online classified ads website. The website itself handled fairly high volumes, and generated a good deal of income for the business. Our core application handled some display of classified ads itself, and also proxied calls to other services that provided different types of products, as shown in Figure 3-1. This is actually an example of a *strangler application*, where a new system intercepts calls made to legacy applications and gradually replaces them altogether. As part of this project, we were partway through retiring the older applications. We had just moved over the highest volume and biggest earning product, but much of the rest of the ads were still being served by a number of older applications. In terms of both the number of searches and the money made by these applications, there was a very long tail.

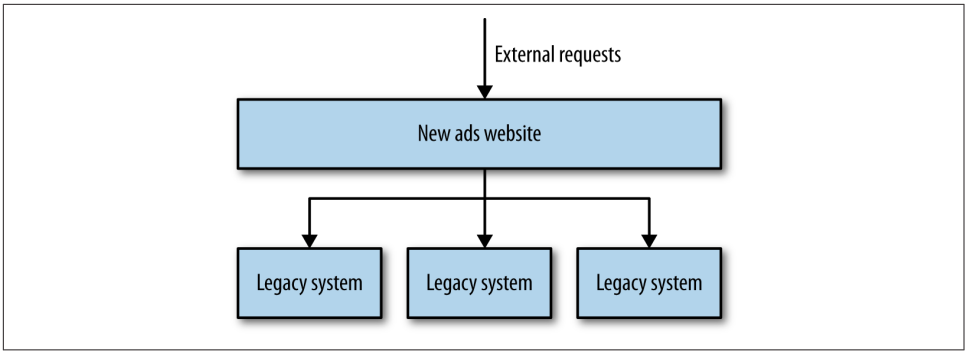


Figure 3-1. A classified ads website strangling older legacy applications

Our system had been live for a while and was behaving very well, handling a not insignificant load. At that time we must have been handling around 6,000–7,000 requests per second during peak, and although most of that was very heavily cached by reverse proxies sitting in front of our application servers, the searches for products (the most important aspect of the site) were mostly uncached and required a full server round-trip.

One morning, just before we hit our daily lunchtime peak, the system started behaving slowly, then gradually started failing. We had some level of monitoring on our new core application, enough to tell us that each of our application nodes was hitting a 100% CPU spike, well above the normal levels even at peak. In a short period of time, the entire site went down.

We managed to track down the culprit and bring the site back up. It turned out one of the downstream ad systems, one of the oldest and least actively maintained, had started responding very slowly. Responding very slowly is one of the worst failure modes you can experience. If a system is just not there, you find out pretty quickly. When it's just *slow*, you end up waiting around for a while before giving up. But whatever the cause of the failure, we had created a system that was vulnerable to a cascading failure. A downstream service, over which we had little control, was able to take down our whole system.

While one team looked at the problems with the downstream system, the rest of us started looking at what had gone wrong in our application. We found a few problems. We were using an HTTP connection pool to handle our downstream connections. The threads in the pool itself had timeouts configured for how long they would wait when making the downstream HTTP call, which is good. The problem was that the workers were all taking a while to time out due to the slow downstream system. While they were waiting, more requests went to the pool asking for worker threads. With no workers available, these requests themselves hung. It turned out the connection pool library we were using did have a timeout for waiting for workers, but this

was *disabled by default*! This led to a huge build-up of blocked threads. Our application normally had 40 concurrent connections at any given time. In the space of five minutes, this situation caused us to peak at around 800 connections, bringing the system down.

What was worse was that the downstream service we were talking to represented functionality that less than 5% of our customer base used, and generated even less revenue than that. When you get down to it, we discovered the hard way that systems that just act slow are *much* harder to deal with than systems that just fail fast. In a distributed system, latency kills.

Even if we'd had the timeouts on the pool set correctly, we were also sharing a single HTTP connection pool for all outbound requests. This meant that one slow service could exhaust the number of available workers all by itself, even if everything else was healthy. Lastly, it was clear that the downstream service in question wasn't healthy, but we kept sending traffic its way. In our situation, this meant we were actually making a bad situation worse, as the downstream service had no chance to recover. We ended up implementing three fixes to avoid this happening again: getting our *timeouts* right, implementing *bulkheads* to separate out different connection pools, and implementing a *circuit breaker* to avoid sending calls to an unhealthy system in the first place.

The Antifragile Organization

In his book *Antifragile* (Random House), Nassim Taleb talks about things that actually benefit from failure and disorder. Ariel Tseitlin used this concept to coin the concept of the antifragile organization in regards to how Netflix operates.

The scale at which Netflix operates is well known, as is the fact that Netflix is based entirely on the AWS infrastructure. These two factors mean that it has to embrace failure well. Netflix goes beyond that by actually *inciting* failure to ensure that its systems are tolerant of it.

Some organizations would be happy with *game days*, where failure is simulated by systems being switched off and having the various teams react. During my time at Google, this was a fairly common occurrence for various systems, and I certainly think that many organizations could benefit from having these sorts of exercises regularly. Google goes beyond simple tests to mimic server failure, and as part of its annual DiRT (Disaster Recovery Test) exercises it has simulated large-scale disasters such as earthquakes. Netflix also takes a more aggressive approach, by writing programs that cause failure and running them in production on a daily basis.

The most famous of these programs is the Chaos Monkey, which during certain hours of the day will turn off random machines. Knowing that this can and will happen in production means that the developers who create the systems really have to be prepared for it. The Chaos Monkey is just one part of Netflix's Simian Army of failure bots. The Chaos Gorilla is used to take out an entire availability center (the AWS equivalent of a data center), whereas the Latency Monkey simulates slow network connectivity between machines. Netflix has made these tools available under an open source license. For many, the ultimate test of whether your system really is robust might be unleashing your very own Simian Army on your production infrastructure.

Embracing and inciting failure through software, and building systems that can handle it, is only part of what Netflix does. It also understands the importance of learning from the failure when it occurs, and adopting a blameless culture when mistakes do happen. Developers are further empowered to be part of this learning and evolving process, as each developer is also responsible for managing his or her production services.

By causing failure to happen, and building for it, Netflix has ensured that the systems it has scale better, and better support the needs of its customers.

Not everyone needs to go to the sorts of extremes that Google or Netflix do, but it is important to understand the mindset shift that is required with distributed systems. Things will fail. The fact that your system is now spread across multiple machines (which can and will fail) across a network (which will be unreliable) can actually make your system more vulnerable, not less. So regardless of whether you're trying to provide a service at the scale of Google or Netflix, preparing yourself for the sorts of failure that happen with more distributed architectures is pretty important. So what do we need to do to handle failure in our systems?

Timeouts

Timeouts are something it is easy to overlook, but in a downstream system they are important to get right. How long can I wait before I can consider a downstream system to actually be down?

Wait too long to decide that a call has failed, and you can slow the whole system down. Time out too quickly, and you'll consider a call that might have worked as failed. Have no timeouts at all, and a downstream system being down could hang your whole system.

Put timeouts on all out-of-process calls, and pick a default timeout for everything. Log when timeouts occur, look at what happens, and change them accordingly.

Circuit Breakers

In your own home, circuit breakers exist to protect your electrical devices from spikes in the power. If a spike occurs, the circuit breaker gets blown, protecting your expensive home appliances. You can also manually disable a circuit breaker to cut the power to part of your home, allowing you to work safely on the electrics. Michael Nygard's book *Release It!* (Pragmatic Programmers) shows how the same idea can work wonders as a protection mechanism for our software.

Consider the story I shared just a moment ago. The downstream legacy ad application was responding very slowly, before eventually returning an error. Even if we'd got the timeouts right, we'd be waiting a long time before we got the error. And then we'd try it again the next time a request came in, and wait. It's bad enough that the downstream service is malfunctioning, but it's making us go slow too.

With a circuit breaker, after a certain number of requests to the downstream resource have failed, the circuit breaker is blown. All further requests fail fast while the circuit breaker is in its blown state. After a certain period of time, the client sends a few requests through to see if the downstream service has recovered, and if it gets enough healthy responses it resets the circuit breaker. You can see an overview of this process in Figure 3-2.

How you implement a circuit breaker depends on what a *failed* request means, but when I've implemented them for HTTP connections I've taken failure to mean either a timeout or a 5XX HTTP return code. In this way, when a downstream resource is down, or timing out, or returning errors, after a certain threshold is reached we automatically stop sending traffic and start failing fast. And we can automatically start again when things are healthy.

Getting the settings right can be a little tricky. You don't want to blow the circuit breaker too readily, nor do you want to take too long to blow it. Likewise, you really want to make sure that the downstream service is healthy again before sending traffic. As with timeouts, I'd pick some sensible defaults and stick with them everywhere, then change them for specific cases.

While the circuit breaker is blown, you have some options. One is to queue up the requests and retry them later on. For some use cases, this might be appropriate, especially if you're carrying out some work as part of a asynchronous job. If this call is being made as part of a synchronous call chain, however, it is probably better to fail fast. This could mean propagating an error up the call chain, or a more subtle degrading of functionality.

If we have this mechanism in place (as with the circuit breakers in our home), we could use them manually to make it safer to do our work. For example, if we wanted to take a microservice down as part of routine maintenance, we could manually blow

all the circuit breakers of the dependent systems so they fail fast while the microservice is offline. Once it's back, we can reset the circuit breakers and everything should go back to normal.

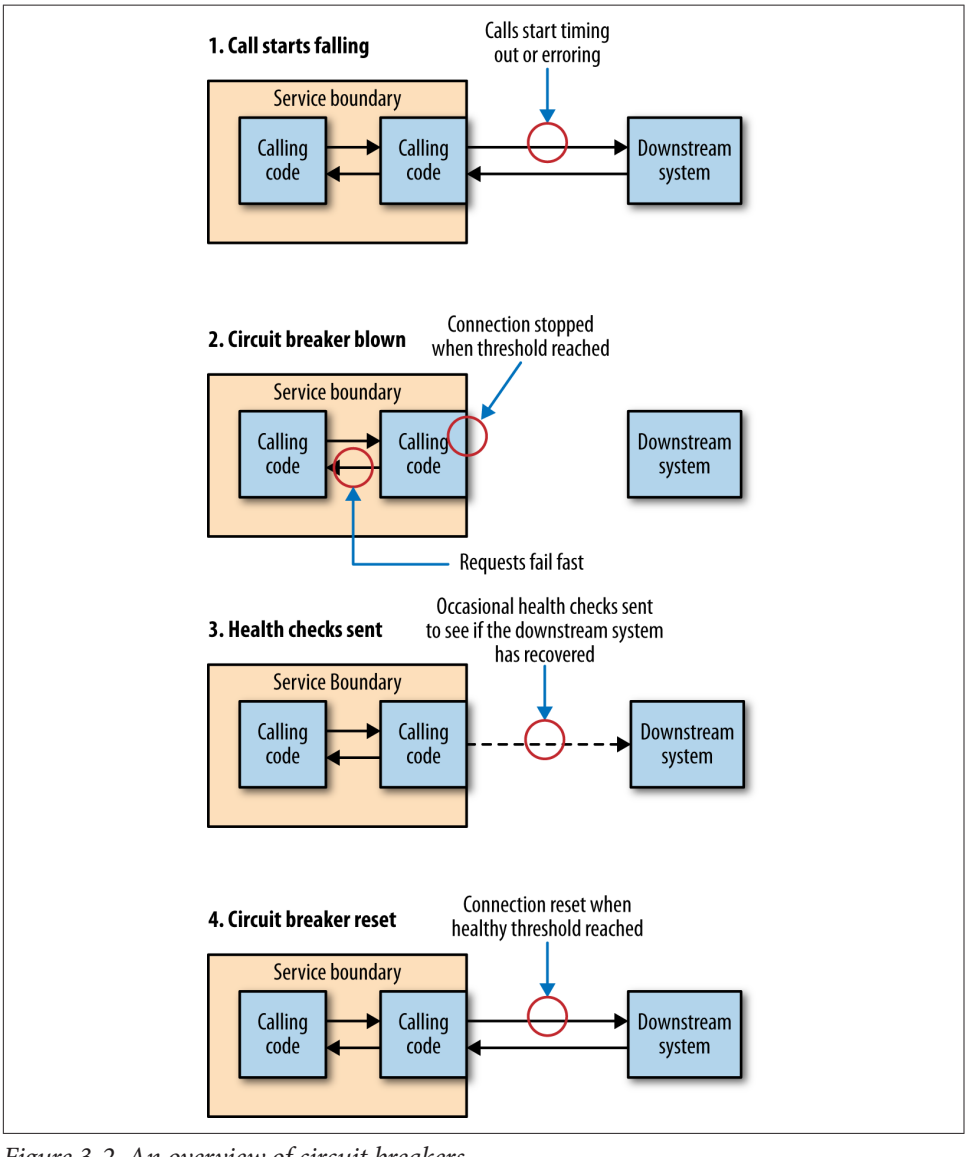


Figure 3-2. An overview of circuit breakers

Bulkheads

In another pattern from *Release It!*, Nygard introduces the concept of a *bulkhead* as a way to isolate yourself from failure. In shipping, a bulkhead is a part of the ship that can be sealed off to protect the rest of the ship. So if the ship springs a leak, you can close the bulkhead doors. You lose part of the ship, but the rest of it remains intact.

In software architecture terms, there are lots of different bulkheads we can consider. Returning to my own experience, we actually missed the chance to implement a bulkhead. We should have used different connection pools for each downstream connection. That way, if one connection pool gets exhausted, the other connections aren't impacted, as we see in Figure 3-3. This would ensure that if a downstream service started behaving slowly in the future, only that one connection pool would be impacted, allowing other calls to proceed as normal.

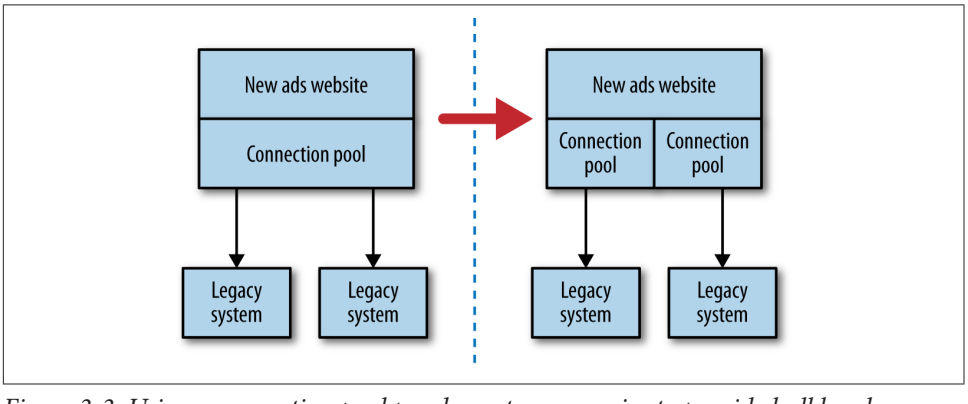


Figure 3-3. Using a connection pool per downstream service to provide bulkheads

Separation of concerns can also be a way to implement bulkheads. By teasing apart functionality into separate microservices, we reduce the chance of an outage in one area affecting another.

Look at all the aspects of your system that can go wrong, both inside your microservices and between them. Do you have bulkheads in place? I'd suggest starting with separate connection pools for each downstream connection at the very least. You may want to go further, however, and consider using circuit breakers too.

We can think of our circuit breakers as an automatic mechanism to seal a bulkhead, to not only protect the consumer from the downstream problem, but also to potentially protect the downstream service from more calls that may be having an adverse impact. Given the perils of cascading failure, I'd recommend mandating circuit breakers for all your synchronous downstream calls. You don't have to write your own, either. Netflix's Hystrix library is a JVM circuit breaker abstraction that comes with

some powerful monitoring, but other implementations exist for different technology stacks, such as Polly for .NET, or the circuit_breaker mixin for Ruby.

In many ways, bulkheads are the most important of these three patterns. Timeouts and circuit breakers help you free up resources when they are becoming constrained, but bulkheads can ensure they don't become constrained in the first place. Hystrix allows you, for example, to implement bulkheads that actually reject requests in certain conditions to ensure that resources don't become even more saturated; this is known as *load shedding*. Sometimes rejecting a request is the best way to stop an important system from becoming overwhelmed and being a bottleneck for multiple upstream services.

Isolation

The more one service depends on another being up, the more the health of one impacts the ability of the other to do its job. If we can use integration techniques that allow a downstream server to be offline, upstream services are less likely to be affected by outages, planned or unplanned.

There is another benefit to increasing isolation between services. When services are isolated from each other, much less coordination is needed between service owners. The less coordination needed between teams, the more autonomy those teams have, as they are able to operate and evolve their services more freely.

Idempotency

In *idempotent* operations, the outcome doesn't change after the first application, even if the operation is subsequently applied multiple times. If operations are idempotent, we can repeat the call multiple times without adverse impact. This is very useful when we want to replay messages that we aren't sure have been processed, a common way of recovering from error.

Let's consider a simple call to add some points as a result of one of our customers placing an order. We might make a call with the sort of payload shown in Example 3-1.

Example 3-1. Crediting points to an account

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

If this call is received multiple times, we would add 100 points multiple times. As it stands, therefore, this call is not idempotent. With a bit more information, though, we allow the points bank to make this call idempotent, as shown in Example 3-2.

Example 3-2. Adding more information to the points credit to make it idempotent

```
<credit>
  <amount>100</amount>
  <forAccount>1234</account>
  <reason>
    <forPurchase>4567</forPurchase>
  </reason>
</credit>
```

Now we know that this credit relates to a specific order, `4567`. Assuming that we could receive only one credit for a given order, we could apply this credit again without increasing the overall number of points.

This mechanism works just as well with event-based collaboration, and can be especially useful if you have multiple instances of the same type of service subscribing to events. Even if we store which events have been processed, with some forms of asynchronous message delivery there may be small windows where two workers can see the same message. By processing the events in an idempotent manner, we ensure this won't cause us any issues.

Some people get quite caught up with this concept, and assume it means that subsequent calls with the same parameters can't have *any* impact, which then leaves us in an interesting position. We really would still like to record the fact that a call was received in our logs, for example. We want to record the response time of the call and collect this data for monitoring. The key point here is that it is the underlying business operation that we are considering idempotent, not the entire state of the system.

Some of the HTTP verbs, such as GET and PUT, are defined in the HTTP specification to be idempotent, but for that to be the case, they rely on your service handling these calls in an idempotent manner. If you start making these verbs nonidempotent, but callers think they can safely execute them repeatedly, you may get yourself into a mess. Remember, just because you're using HTTP as an underlying protocol doesn't mean you get everything for free!

Scaling

We scale our systems in general for one of two reasons. First, to help deal with failure: if we're worried that something will fail, then having more of it will help, right? Second, we scale for performance, either in terms of handling more load, reducing latency, or both. Let's look at some common scaling techniques we can use and think about how they apply to microservice architectures.

Go Bigger

Some operations can just benefit from more grunt. Getting a bigger box with faster CPU and better I/O can often improve latency and throughput, allowing you to process more work in less time. However, this form of scaling, often called *vertical scaling*, can be expensive—sometimes one big server can cost more than two smaller servers with the same combined raw power, especially when you start getting to really big machines. Sometimes our software itself cannot do much with the extra resources available to it. Larger machines often just give us more CPU cores, but not enough of our software is written to take advantage of them. The other problem is that this form of scaling may not do much to improve our server's resiliency if we only have one of them! Nonetheless, this can be a good quick win, especially if you're using a virtualization provider that lets you resize machines easily.

Splitting Workloads

As outlined in (cross-ref to come), having a single microservice per host is certainly preferable to a multiservice-per-host model. Initially, however, many people decide to coexist multiple microservices on one box to keep costs down or to simplify host management (although that is an arguable reason). As the microservices are independent processes that communicate over the network, it should be an easy task to then move them onto their own hosts to improve throughput and scaling. This can also increase the resiliency of the system, as a single host outage will impact a reduced number of microservices.

Of course, we could also use the need for increased scale to split an existing microservice into parts to better handle the load. As a simplistic example, let's imagine that our accounts service provides the ability to create and manage individual customers' financial accounts, but also exposes an API for running queries to generate reports. This query capability places a significant load on the system. The query capacity is considered noncritical, as it isn't needed to keep orders flowing in during the day. The ability to manage the financial records for our customers *is* critical, though, and we can't afford for it to be down. By splitting these two capabilities into separate services, we reduce the load on the critical accounts service, and introduce a new accounts reporting service that is designed not only with querying in mind (perhaps using some of the techniques we outlined in Chapter 2, but also as a noncritical system doesn't need to be deployed in as resilient a way as the core accounts service.

Spreading Your Risk

One way to scale for resilience is to ensure that you don't put all your eggs in one basket. A simplistic example of this is making sure that you don't have multiple services on one host, where an outage would impact multiple services. But let's consider what *host* means. In most situations nowadays, a *host* is actually a virtual concept. So

what if I have all of my services on different hosts, but all those hosts are actually virtual hosts, running on the same physical box? If that box goes down, I could lose multiple services. Some virtualization platforms enable you to ensure that your hosts are distributed across multiple different physical boxes to reduce this chance.

For internal virtualization platforms, it is a common practice to have the virtual machine's root partition mapped to a single SAN (storage area network). If that SAN goes down, it can take down all connected VMs. SANs are big, expensive, and designed not to fail. That said, I have had big expensive SANs fail on me at least twice in the last 10 years, and each time the results were fairly serious.

Another common form of separation to reduce failure is to ensure that not all your services are running in a single rack in the data center, or that your services are distributed across more than one data center. If you're using an underlying service provider, it is important to know if a service-level agreement (SLA) is offered and plan accordingly. If you need to ensure your services are down for no more than four hours every quarter, but your hosting provider can only guarantee a downtime of eight hours per quarter, you have to either change the SLA, or come up with an alternative solution.

AWS, for example, is split into regions, which you can think of as distinct clouds. Each region is in turn split into two or more availability zones (AZs). AZs are AWS's equivalent of a data center. It is essential to have services distributed across multiple availability zones, as AWS does not offer any guarantees about the availability of a single node, or even an entire availability zone. For its compute service, it offers only a 99.95% uptime over a given monthly period of the region as a whole, so you'll want to distribute your workloads across multiple availability zones inside a single region. For some people, this isn't good enough, and instead they run their services across multiple regions too.

It should be noted, of course, that because providers give you an SLA *guarantee*, they will tend to limit their liability! If them missing their targets costs you customers and a large amount of money, you might find yourself searching through contracts to see if you can claw anything back from them. Therefore, I would strongly suggest you understand the impact of a supplier failing in its obligations to you, and work out if you need to have a plan B (or C) in your pocket. More than one client I've worked with has had a disaster recovery hosting platform with a different supplier, for example, to ensure they weren't too vulnerable to the mistakes of one company.

Load Balancing

When you need your service to be resilient, you want to avoid single points of failure. For a typical microservice that exposes a synchronous HTTP endpoint, the easiest way to achieve this is to have multiple hosts running your microservice instance, sitting behind a load balancer, as shown in Figure 3-4. To consumers of the microservice, you don't know if you are talking to one microservice instance or a hundred.

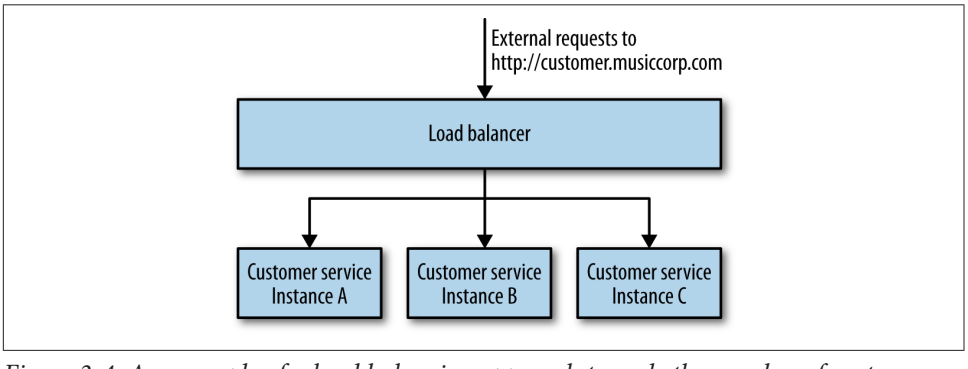


Figure 3-4. An example of a load balancing approach to scale the number of customer service instances

Load balancers come in all shapes and sizes, from big and expensive hardware appliances to software-based load balancers like mod_proxy. They all share some key capabilities. They distribute calls sent to them to one or more instances based on some algorithm, remove instances when they are no longer healthy, and hopefully add them back in when they are.

Some load balancers provide useful features. A common one is *SSL termination*, where inbound HTTPS connections to the load balancer are transformed to HTTP connections once they hit the instance itself. Historically, the overhead of managing SSL was significant enough that having a load balancer handle this process for you was fairly useful. Nowadays, this is as much about simplifying the set-up of the individual hosts running the instance. The point of using HTTPS, though, is to ensure that the requests aren't vulnerable to a man-in-the-middle attack, as we discussed in (cross-ref to come), so if we use SSL termination, we are potentially exposing ourselves somewhat. One mitigation is to have all the instances of the microservice inside a single VLAN, as we see in Figure 3-5. A VLAN is a virtual local area network, that is isolated in such a way that requests from outside it can come only via a router, and in this case our router is also our SSL-terminating load balancer. The only communication to the microservice from outside the VLAN comes over HTTPS, but internally everything is HTTP.

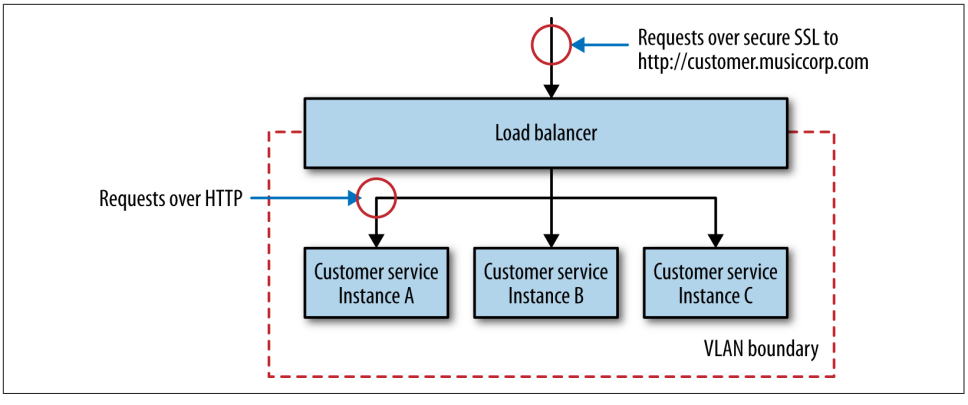


Figure 3-5. Using HTTPS termination at the load balancer with a VLAN for improved security

AWS provides HTTPS-terminating load balancers in the form of ELBs (elastic load balancers) and you can use its security groups or virtual private clouds (VPCs) to implement the VLAN. Otherwise, software like mod_proxy can play a similar role as a software load balancer. Many organizations have hardware load balancers, which can be difficult to automate. In such circumstances I have found myself advocating for software load balancers sitting *behind* the hardware load balancers to allow teams the freedom to reconfigure these as required. You do want to watch for the fact that all too often the hardware load balancers themselves are single points of failure! Whatever approach you take, when considering the configuration of a load balancer, treat it as you treat the configuration of your service: make sure it is stored in version control and can be applied automatically.

Load balancers allow us to add more instances of our microservice in a way that is transparent to any service consumers. This gives us an increased ability to handle load, and also reduce the impact of a single host failing. However, many, if not most, of your microservices will have some sort of persistent data store, probably a database sitting on a different machine. If we have multiple microservice instances on different machines, but only a single host running the database instance, our database is still a single source of failure. We'll talk about patterns to handle this shortly.

Worker-Based Systems

Load balancing isn't the only way to have multiple instances of your service share load and reduce fragility. Depending on the nature of the operations, a worker-based system could be just as effective. Here, a collection of instances all work on some shared backlog of work. This could be a number of Hadoop processes, or perhaps a number of listeners to a shared queue of work. These types of operations are well

suited to batch work or asynchronous jobs. Think of tasks like image thumbnail processing, sending email, or generating reports.

The model also works well for *peaky* load, where you can spin up additional instances on demand to match the load coming in. As long as the work queue itself is resilient, this model can be used to scale both for improved throughput of work, but also for improved resiliency—the impact of a worker failing (or not being there) is easy to deal with. Work will take longer, but nothing gets lost.

I've seen this work well in organizations where there is lots of unused compute capacity at certain times of day. For example, overnight you might not need as many machines to run your ecommerce system, so you can temporarily use them to run workers for a reporting job instead.

With worker-based systems, although the workers themselves don't need to be that reliable, the system that contains the work to be done does. You could handle this by running a persistent message broker, for example, or perhaps a system like Zookeeper. The benefit here is that if we use existing software for this purpose, someone has done much of the hard work for us. However, we still need to know how to set up and maintain these systems in a resilient fashion.

Starting Again

The architecture that gets you started may not be the architecture that keeps you going when your system has to handle very different volumes of load. As Jeff Dean said in his presentation "Challenges in Building Large-Scale Information Retrieval Systems" (WSDM 2009 conference), you should "design for ~10× growth, but plan to rewrite before ~100×." At certain points, you need to do something pretty radical to support the next level of growth.

Recall the story of Gilt, which we touched on in (cross-ref to come). A simple monolithic Rails application did well for Gilt for two years. Its business became increasingly successful, which meant more customers and more load. At a certain tipping point, the company had to redesign the application to handle the load it was seeing.

A redesign may mean splitting apart an existing monolith, as it did for Gilt. Or it might mean picking new data stores that can handle the load better, which we'll look at in a moment. It could also mean adopting new techniques, such as moving from synchronous request/response to event-based systems, adopting new deployment platforms, changing whole technology stacks, or everything in between.

There is a danger that people will see the need to rearchitect when certain scaling thresholds are reached as a reason to build for massive scale from the beginning. This can be disastrous. At the start of a new project, we often don't know exactly what we want to build, nor do we know if it will be successful. We need to be able to rapidly experiment, and understand what capabilities we need to build. If we tried building

for massive scale up front, we'd end up front-loading a huge amount of work to prepare for load that may never come, while diverting effort away from more important activities, like understanding if anyone will want to actually use our product. Eric Ries tells the story of spending six months building a product that no one ever downloaded. He reflected that he could have put up a link on a web page that 404'd when people clicked on it to see if there was any demand, spent six months on the beach instead, and learned just as much!

The need to change our systems to deal with scale isn't a sign of failure. It is a sign of success.

Scaling Databases

Scaling stateless microservices is fairly straightforward. But what if we are storing data in a database? We'll need to know how to scale that too. Different types of databases provide different forms of scaling, and understanding what form suits your use case best will ensure you select the right database technology from the beginning.

Availability of Service Versus Durability of Data

Straight off, it is important to separate the concept of availability of the service from the durability of the data itself. You need to understand that these are two different things, and as such they will have different solutions.

For example, I could store a copy of all data written to my database in a resilient filesystem. If the database goes down, my data isn't lost, as I have a copy, but the database itself isn't available, which may make my microservice unavailable too. A more common model would be using a standby. All data written to the primary database gets copied to the standby replica database. If the primary goes down, my data is safe, but without a mechanism to either bring it back up or promote the replica to the primary, we don't have an available database, even though our data is safe.

Scaling for Reads

Many services are read-mostly. Think of a catalog service that stores information for the items we have for sale. We add records for new items on a fairly irregular basis, and it wouldn't at all be surprising if we get more than 100 reads of our catalog's data for every write. Happily, scaling for reads is much easier than scaling for writes. Caching of data can play a large part here, and we'll discuss that in more depth shortly. Another model is to make use of *read replicas*.

In a relational database management system (RDBMS) like MySQL or Postgres, data can be copied from a primary node to one or more replicas. This is often done to ensure that a copy of our data is kept safe, but we can also use it to distribute our reads. A service could direct all writes to the single primary node, but distribute reads

to one or more read replicas, as we see in Figure 3-6. The replication from the primary database to the replicas happens at some point after the write. This means that with this technique reads may sometimes see *stale* data until the replication has completed. Eventually the reads will see the consistent data. Such a setup is called *eventually consistent*, and if you can handle the temporary inconsistency it is a fairly easy and common way to help scale systems. We'll look into this in more depth shortly when we look at the CAP theorem.

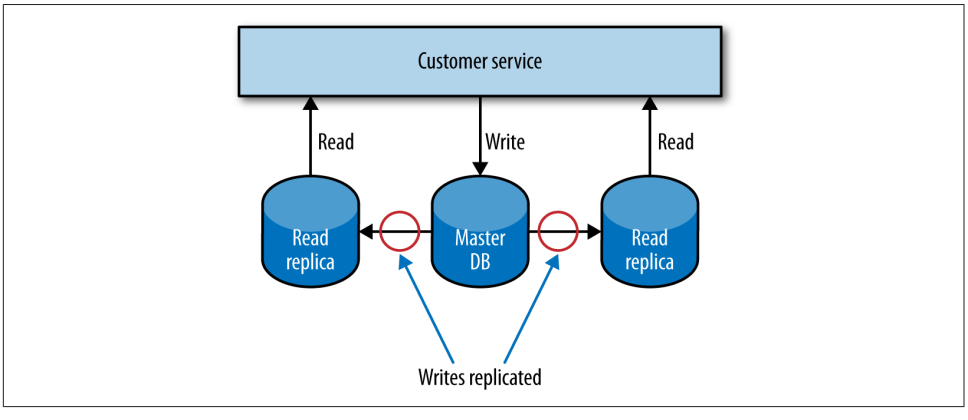


Figure 3-6. Using read replicas to scale reads

Years ago, using read replicas to scale was all the rage, although nowadays I would suggest you look to caching first, as it can deliver much more significant improvements in performance, often with less work.

Scaling for Writes

Reads are comparatively easy to scale. What about writes? One approach is to use *sharding*. With sharding, you have multiple database nodes. You take a piece of data to be written, apply some hashing function to the key of the data, and based on the result of the function learn where to send the data. To pick a very simplistic (and actually bad) example, imagine that customer records A–M go to one database instance, and N–Z another. You can manage this yourself in your application, but some databases, like Mongo, handle much of it for you.

The complexity with sharding for writes comes from handling queries. Looking up an individual record is easy, as I can just apply the hashing function to find which instance the data should be on, and then retrieve it from the correct shard. But what about queries that span the data in multiple nodes—for example, finding all the customers who are over 18? If you want to query all shards, you either need to query each individual shard and join in memory, or have an alternative read store where both data sets are available. Often querying across shards is handled by an asynchro-

nous mechanism, using cached results. Mongo uses map/reduce jobs, for example, to perform these queries.

One of the questions that emerges with sharded systems is, what happens if I want to add an extra database node? In the past, this would often require significant downtime—especially for large clusters—as you might have to take the entire database down and rebalance the data. More recently, more systems support adding extra shards to a live system, where the rebalancing of data happens in the background; Cassandra, for example, handles this very well. Adding shards to an existing cluster isn't for the faint of heart, though, so make sure you test this thoroughly.

Sharding for writes may scale for write volume, but may not improve resiliency. If customer records A–M always go to Instance X, and Instance X is unavailable, access to records A–M can be lost. Cassandra offers additional capabilities here, where we can ensure that data is replicated to multiple nodes in a *ring* (Cassandra's term for a collection of Cassandra nodes).

As you may have inferred from this brief overview, scaling databases for writes are where things get very tricky, and where the capabilities of the various databases really start to become differentiated. I often see people changing database technology when they start hitting limits on how easily they can scale their existing write volume. If this happens to you, buying a bigger box is often the quickest way to solve the problem, but in the background you might want to look at systems like Cassandra, Mongo, or Riak to see if their alternative scaling models might offer you a better long-term solution.

Shared Database Infrastructure

Some types of databases, such as the traditional RDBMS, separate the concept of the database itself and the schema. This means one running database could host multiple, independent schemas, one for each microservice. This can be very useful in terms of reducing the number of machines we need to run our system, but we are introducing a significant single point of failure. If this database infrastructure goes down, it can impact multiple microservices at once, potentially resulting in a catastrophic outage. If you are running this sort of setup, make sure you consider the risks. And be very sure that the database itself is as resilient as it can be.

CQRS

The Command-Query Responsibility Segregation (CQRS) pattern refers to an alternate model for storing and querying information. With normal databases, we use one system for performing modifications to data and querying the data. With CQRS, part of the system deals with commands, which capture requests to modify state, while another part of the system deals with queries.

Commands come in requesting changes in state. These commands are validated, and if they work, they will be applied to the model. Commands should contain information about their intent. They can be processed synchronously or asynchronously, allowing for different models to handle scaling; we could, for example, just queue up inbound requests and process them later.

The key takeaway here is that the internal models used to handle commands and queries are themselves completely separate. For example, I might choose to handle and process commands as events, perhaps just storing the list of commands in a data store (a process known as *event sourcing*). My query model could query an event store and create projections from stored events to assemble the state of domain objects, or could just pick up a feed from the command part of the system to update a different type of store. In many ways, we get the same benefits of read replicas that we discussed earlier, without the requirement that the backing store for the replicas be the same as the data store used to handle data modifications.

This form of separation allows for different types of scaling. The command and query parts of our system could live in different services, or on different hardware, and could make use of radically different types of data store. This can unlock a large number of ways to handle scale. You could even support different types of read format by having multiple implementations of the query piece, perhaps supporting a graph-based representation of your data, or a key/value-based form of your data.

Be warned, however: this sort of pattern is quite a shift away from a model where a single data store handles all our CRUD operations. I've seen more than one experienced development team struggle to get this pattern right!

Caching

Caching is a commonly used performance optimization whereby the previous result of some operation is stored, so that subsequent requests can use this stored value rather than spending time and resources recalculating the value. More often than not, caching is about eliminating needless round-trips to databases or other services to serve results faster. Used well, it can yield huge performance benefits. The reason that HTTP scales so well in handling large numbers of requests is that the concept of caching is built in.

Even with a simple monolithic web application, there are quite a few choices as to where and how to cache. With a microservice architecture, where each service is its own source of data and behavior, we have many more choices to make about where and how to cache. With a distributed system, we typically think of caching either on the client side or on the server side. But which is best?

Client-Side, Proxy, and Server-Side Caching

In client-side caching, the client stores the cached result. The client gets to decide when (and if) it goes and retrieves a fresh copy. Ideally, the downstream service will provide hints to help the client understand what to do with the response, so it knows when and if to make a new request. With proxy caching, a proxy is placed between the client and the server. A great example of this is using a reverse proxy or content delivery network (CDN). With server-side caching, the server handles caching responsibility, perhaps making use of a system like Redis or Memcache, or even a simple in-memory cache.

Which one makes the most sense depends on what you are trying to optimize. Client-side caching can help reduce network calls drastically, and can be one of the fastest ways of reducing load on a downstream service. In this case, the client is in charge of the caching behavior, and if you want to make changes to how caching is done, rolling out changes to a number of consumers could be difficult. Invalidation of stale data can also be trickier, although we'll discuss some coping mechanisms for this in a moment.

With proxy caching, everything is opaque to both the client and server. This is often a very simple way to add caching to an existing system. If the proxy is designed to cache generic traffic, it can also cache more than one service; a common example is a reverse proxy like Squid or Varnish, which can cache any HTTP traffic. Having a proxy between the client and server does introduce additional network hops, although in my experience it is very rare that this causes problems, as the performance optimizations resulting from the caching itself outweigh any additional network costs.

With server-side caching, everything is opaque to the clients; they don't need to worry about anything. With a cache near or inside a service boundary, it can be easier to reason about things like invalidation of data, or track and optimize cache hits. In a situation where you have multiple types of clients, a server-side cache could be the fastest way to improve performance.

For every public-facing website I've worked on, we've ended up doing a mix of all three approaches. But for more than one distributed system, I've gotten away with no caching at all. But it all comes down to knowing what load you need to handle, how fresh your data needs to be, and what your system can do right now. Knowing that you have a number of different tools at your disposal is just the beginning.

Caching in HTTP

HTTP provides some really useful controls to help us cache either on the client side or server side, which are worth understanding even if you aren't using HTTP itself.

First, with HTTP, we can use `cache-control` directives in our responses to clients. These tell clients if they should cache the resource at all, and if so how long they should cache it for in seconds. We also have the option of setting an `Expires` header, where instead of saying how long a piece of content can be cached for, we specify a time and date at which a resource should be considered stale and fetched again. The nature of the resources you are sharing determines which one is most likely to fit. Standard static website content like CSS or images often fit well with a simple `cache-control` time to live (TTL). On the other hand, if you know in advance when a new version of a resource will be updated, setting an `Expires` header will make more sense. All of this is very useful in stopping a client from even needing to make a request to the server in the first place.

Aside from `cache-control` and `Expires`, we have another option in our arsenal of HTTP goodies: Entity Tags, or ETags. An ETag is used to determine if the value of a resource has changed. If I update a customer record, the URI to the resource is the same, but the value is different, so I would expect the ETag to change. This becomes powerful when we're using what is called a *conditional GET*. When making a GET request, we can specify additional headers, telling the service to send us the resource only if some criteria are met.

For example, let's imagine we fetch a customer record, and its ETag comes back as `o5t6fkd2sa`. Later on, perhaps because a `cache-control` directive has told us the resource should be considered stale, we want to make sure we get the latest version. When issuing the subsequent GET request, we can pass in a `If-None-Match: o5t6fkd2sa`. This tells the server that we want the resource at the specified URI, unless it already matches this ETag value. If we already have the up-to-date version, the service sends us a `304 Not Modified` response, telling us we have the latest version. If there is a newer version available, we get a `200 OK` with the changed resource, and a new ETag for the resource.

The fact that these controls are built into such a widely used specification means we get to take advantage of a lot of preexisting software that handles the caching for us. Reverse proxies like Squid or Varnish can sit transparently on the network between client and server, storing and expiring cached content as required. These systems are geared toward serving huge numbers of concurrent requests very fast, and are a standard way of scaling public-facing websites. CDNs like AWS's CloudFront or Akamai can ensure that requests are routed to caches near the calling client, making sure that traffic doesn't go halfway round the world when it needs to. And more prosaically, HTTP client libraries and client caches can handle a lot of this work for us.

ETags, `Expires`, and `cache-control` can overlap a bit, and if you aren't careful you can end up giving conflicting information if you decide to use all of them! For a more in-depth discussion of the various merits, take a look at the book *REST In Practice*

(O'Reilly) or read section 13 of the HTTP 1.1 specification, which describes how both clients and servers are supposed to implement these various controls.

Whether you decide to use HTTP as an interservice protocol, caching at the client and reducing the need for round-trips to the client is well worth it. If you decide to pick a different protocol, understand when and how you can provide hints to the client to help it understand how long it can cache for.

Caching for Writes

Although you'll find yourself using caching for reads more often, there are some use cases where caching for writes make sense. For example, if you make use of a write-behind cache, you can write to a local cache, and at some later point the data will be flushed to a downstream source, probably the canonical source of data. This can be useful when you have bursts of writes, or when there is a good chance that the same data will be written multiple times. When used to buffer and potentially batch writes, write-behind caches can be a useful further performance optimization.

With a write-behind cache, if the buffered writes are suitably persistent, even if the downstream service is unavailable we could queue up the writes and send them through when it is available again.

Caching for Resilience

Caching can be used to implement resiliency in case of failure. With client-side caching, if the downstream service is unavailable, the client could decide to simply use cached but potentially stale data. We could also use something like a reverse proxy to serve up stale data. For some systems, being available even with stale data is better than not returning a result at all, but that is a judgment call you'll have to make. Obviously, if we don't have the requested data in the cache, then we can't do much to help, but there are ways to mitigate this.

A technique I saw used at the *Guardian*, and subsequently elsewhere, was to crawl the existing *live* site periodically to generate a static version of the website that could be served in the event of an outage. Although this crawled version wasn't as fresh as the cached content served from the live system, in a pinch it could ensure that a version of the site would get displayed.

Hiding the Origin

With a normal cache, if a request results in a cache miss, the request goes on to the origin to fetch the fresh data with the caller blocking, waiting on the result. In the normal course of things, this is to be expected. But if we suffer a massive cache miss, perhaps because an entire machine (or group of machines) that provide our cache fail, a large number of requests will hit the origin.

For those services that serve up highly cachable data, it is common for the origin itself to be scaled to handle only a fraction of the total traffic, as most requests get served out of memory by the caches that sit in front of the origin. If we suddenly get a thundering herd due to an entire cache region vanishing, our origin could be pummelled out of existence.

One way to protect the origin in such a situation is never to allow requests to go to the origin in the first place. Instead, the origin itself populates the cache asynchronously when needed, as shown in Figure 3-7. If a cache miss is caused, this triggers an event that the origin can pick up on, alerting it that it needs to repopulate the cache. So if an entire shard has vanished, we can rebuild the cache in the background. We could decide to block the original request waiting for the region to be repopulated, but this could cause contention on the cache itself, leading to further problems. It's more likely if we are prioritizing keeping the system stable that we would fail the original request, but it would fail fast.

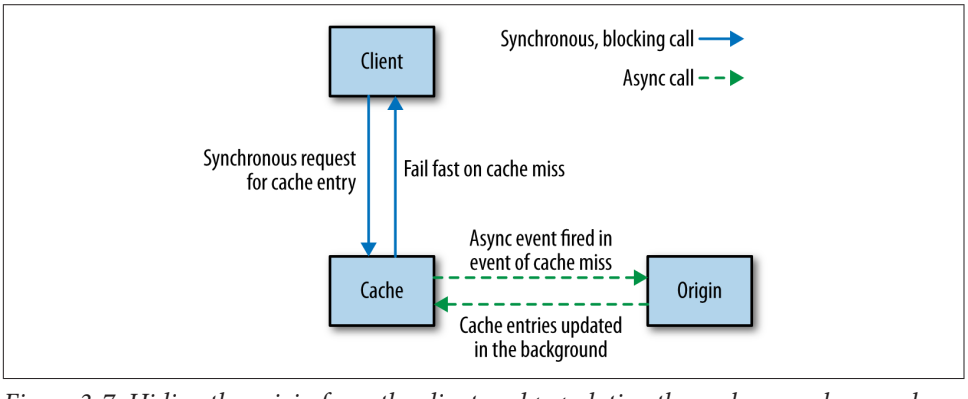


Figure 3-7. Hiding the origin from the client and populating the cache asynchronously

This sort of approach may not make sense for some situations, but it can be a way to ensure the system remains up when parts of it fail. By failing requests fast, and ensuring we don't take up resources or increase latency, we avoid a failure in our cache from cascading downstream and give ourselves a chance to recover.

Keep It Simple

Be careful about caching in too many places! The more caches between you and the source of fresh data, the more stale the data can be, and the harder it can be to determine the freshness of the data that a client eventually sees. This can be especially problematic with a microservice architecture where you have multiple services involved in a call chain. Again, the more caching you have, the harder it will be to assess the freshness of any piece of data. So if you think a cache is a good idea, keep it simple, stick to one, and think carefully before adding more!

Cache Poisoning: A Cautionary Tale

With caching we often think that if we get it wrong the worst thing that can happen is we serve stale data for a bit. But what happens if you end up serving stale data forever? Earlier I mentioned the project I worked on where we were using a strangler application to help intercept calls to multiple legacy systems with a view of incrementally retiring them. Our application operated effectively as a proxy. Traffic to our application was routed through to the legacy application. On the way back, we did a few housekeeping things; for example, we made sure that the results from the legacy application had proper HTTP cache headers applied.

One day, shortly after a normal routine release, something odd started happening. A bug had been introduced whereby a small subset of pages were falling through a logic condition in our cache header insertion code, resulting in us not changing the header at all. Unfortunately, this downstream application had also been changed sometime previously to include an `Expires: Never` HTTP header. This hadn't had any effect earlier, as we were overriding this header. Now we weren't.

Our application made heavy use of Squid to cache HTTP traffic, and we noticed the problem quite quickly as we were seeing more requests bypassing Squid itself to hit our application servers. We fixed the cache header code and pushed out a release, and also manually cleared the relevant region of the Squid cache. However, that wasn't enough.

As I mentioned earlier, you can cache in multiple places. When it comes to serving up content to users of a public-facing web application, you could have multiple caches between you and your customer. Not only might you be fronting your website with something like a CDN, but some ISPs make use of caching. Can you control those caches? And even if you could, there is one cache that you have little control over: the cache in a user's browser.

Those pages with `Expires: Never` stuck in the caches of many of our users, and would never be invalidated until the cache became full or the user cleaned them out manually. Clearly we couldn't make either thing happen; our only option was to change the URLs of these pages so they were refetched.

Caching can be very powerful indeed, but you need to understand the full path of data that is cached from source to destination to really appreciate its complexities and what can go wrong.

Autoscaling

If you are lucky enough to have fully automatable provisioning of virtual hosts, and can fully automate the deployment of your microservice instances, then you have the building blocks to allow you to automatically scale your microservices.

For example, you could also have the scaling triggered by well-known trends. You might know that your system's peak load is between 9 a.m. and 5 p.m., so you bring up additional instances at 8:45 a.m., and turn them off at 5:15 p.m.. If you're using something like AWS (which has very good support for autoscaling built in), turning off instances you don't need any longer will help save money. You'll need data to understand how your load changes over time, from day to day, week to week. Some businesses have obvious seasonal cycles too, so you may need data going back a fair way to make proper judgment calls.

On the other hand, you could be reactive, bringing up additional instances when you see an increase in load or an instance failure, and remove instances when you no longer needed them. Knowing how fast you can scale up once you spot an upward trend is key. If you know you'll only get a couple of minutes' notice about an increase in load, but scaling up will take you at least 10 minutes, you know you'll need to keep extra capacity around to bridge this gap. Having a good suite of load tests is almost essential here. You can use them to test your autoscaling rules. If you don't have tests that can reproduce different loads that will trigger scaling, then you're only going to find out in production if you got the rules wrong. And the consequences of failure aren't great!

A news site is a great example of a type of business where you may want a mix of predictive and reactive scaling. On the last news site I worked on, we saw very clear daily trends, with views climbing from the morning to lunchtime and then starting to decline. This pattern was repeated day in, day out, with traffic less pronounced at the weekend. That gave you a fairly clear trend that could drive proactive scaling up (and down) of resources. On the other hand, a big news story would cause an unexpected spike, requiring more capacity at often short notice.

I actually see autoscaling used much more for handling failure of instances than for reacting to load conditions. AWS lets you specify rules like "There should be at least 5 instances in this group," so if one goes down a new one is automatically launched. I've seen this approach lead to a fun game of whack-a-mole when someone forgets to turn off the rule and then tries to take down the instances for maintenance, only to see them keep spinning up!

Both reactive and predictive scaling are very useful, and can help you be much more cost effective if you're using a platform that allows you to pay only for the computing resources you use. But they also require careful observation of the data available to you. I'd suggest using autoscaling for failure conditions first while you collect the data. Once you want to start scaling for load, make sure you are very cautious about scaling down too quickly. In most situations, having more computing power at your hands than you need is much better than not having enough!

CAP Theorem

We'd like to have it all, but unfortunately we know we can't. And when it comes to distributed systems like those we build using microservice architectures, we even have a mathematical proof that tells us we can't. You may well have heard about the CAP theorem, especially in discussions about the merits of various different types of data stores. At its heart it tells us that in a distributed system, we have three things we can trade off against each other: *consistency*, *availability*, and *partition tolerance*. Specifically, the theorem tells us that we get to keep two in a failure mode.

Consistency is the system characteristic by which I will get the same answer if I go to multiple nodes. Availability means that every request receives a response. Partition tolerance is the system's ability to handle the fact that communication between its parts is sometimes impossible.

Since Eric Brewer published his original conjecture, the idea has gained a mathematical proof. I'm not going to dive into the math of the proof itself, as not only is this not that sort of book, but I can also guarantee that I would get it wrong. Instead, let's use some worked examples that will help us understand that under it all, the CAP theorem is a distillation of a very logical set of reasoning.

We've already talked about some simple database scaling techniques. Let's use one of these to probe the ideas behind the CAP theorem. Let's imagine that our inventory service is deployed across two separate data centers, as shown in Figure 3-8. Backing our service instance in each data center is a database, and these two databases talk to each other to try to synchronize data between them. Reads and writes are done via the local database node, and replication is used to synchronize the data between the nodes.

Now let's think about what happens when something fails. Imagine that something as simple as the network link between the two data centers stops working. The synchronization at this point fails. Writes made to the primary database in DC1 will not propagate to DC2, and vice versa. Most databases that support these setups also support some sort of queuing technique to ensure that we can recover from this afterward, but what happens in the meantime?

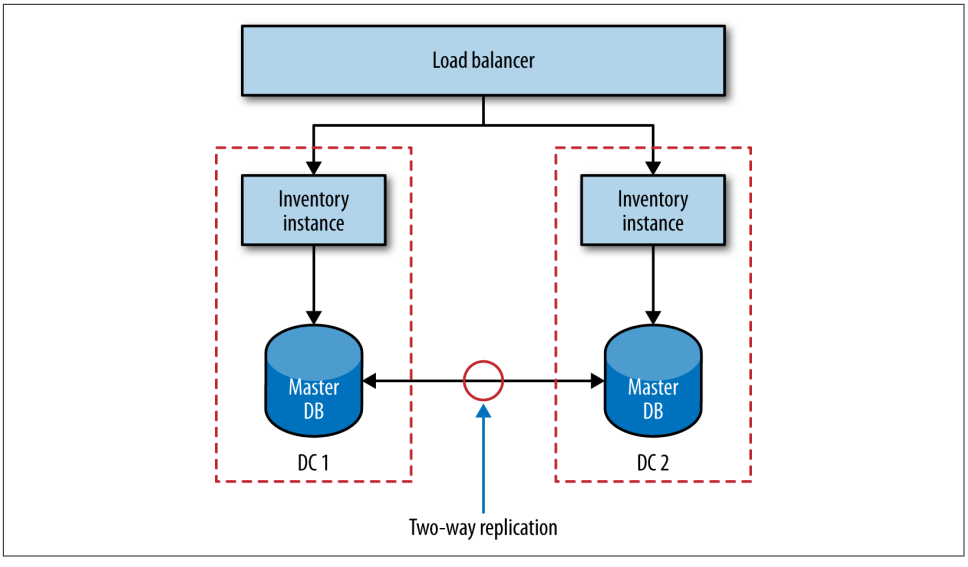


Figure 3-8. Using multiprimary replication to share data between two database nodes

Sacrificing Consistency

Let's assume that we don't shut the inventory service down entirely. If I make a change now to the data in DC1, the database in DC2 doesn't see it. This means any requests made to our inventory node in DC2 see potentially stale data. In other words, our system is still *available* in that both nodes are able to serve requests, and we have kept the system running despite the *partition*, but we have lost *consistency*. This is often called a *AP* system. We don't get to keep all three.

During this partition, if we keep accepting writes then we accept the fact that at some point in the future they have to be resynchronized. The longer the partition lasts, the more difficult this resynchronization can become.

The reality is that even if we don't have a network failure between our database nodes, replication of data is not instantaneous. As touched on earlier, systems that are happy to cede consistency to keep partition tolerance and availability are said to be *eventually consistent*; that is, we expect at some point in the future that all nodes will see the updated data, but it won't happen at once so we have to live with the possibility that users see old data.

Sacrificing Availability

What happens if we need to keep consistency and want to drop something else instead? Well, to keep consistency, each database node needs to know the copy of the data it has is the same as the other database node. Now in the partition, if the data-

base nodes can't talk to each other, they cannot coordinate to ensure consistency. We are unable to guarantee consistency, so our only option is to refuse to respond to the request. In other words, we have sacrificed availability. Our system is consistent and partition tolerant, or CP. In this mode our service would have to work out how to degrade functionality until the partition is healed and the database nodes can be resynchronized.

Consistency across multiple nodes is really hard. There are few things (perhaps nothing) harder in distributed systems. Think about it for a moment. Imagine I want to read a record from the local database node. How do I know it is up to date? I have to go and ask the other node. But I also have to ask that database node to not allow it to be updated while the read completes; in other words, I need to initiate a transactional read across multiple database nodes to ensure consistency. But in general people don't do transactional reads, do they? Because transactional reads are slow. They require locks. A read can block an entire system up. All consistent systems require some level of locking to do their job.

As we've already discussed, distributed systems have to expect failure. Consider our transactional read across a set of consistent nodes. I ask a remote node to lock a given record while the read is initiated. I complete the read, and ask the remote node to release its lock, but now I can't talk to it. What happens now? Locks are really hard to get right even in a single process system, and are significantly more difficult to implement well in a distributed system.

Remember when we talked about distributed transactions in (cross-ref to come)? The core reason they are challenging is because of this problem with ensuring consistency across multiple nodes.

Getting multinode consistency right is so hard that I would strongly, *strongly* suggest that if you need it, don't try to invent it yourself. Instead, pick a data store or lock service that offers these characteristics. Consul, for example, which we'll discuss shortly, implements a strongly consistent key/value store designed to share configuration between multiple nodes. Along with "Friends don't let friends write their own crypto" should go "Friends don't let friends write their own distributed consistent data store." If you think you need to write your own CP data store, read all the papers on the subject first, then get a PhD, and then look forward to spending a few years getting it wrong. Meanwhile, I'll be using something off the shelf that does it for me, or more likely trying *really hard* to build eventually consistent AP systems instead.

Sacrificing Partition Tolerance?

We get to pick two, right? So we've got our eventually consistent AP system. We have our consistent, but hard to build and scale, CP system. Why not a CA system? Well, how can we sacrifice partition tolerance? If our system has no partition tolerance, it

can't run over a network. In other words, it needs to be a single process operating locally. CA systems don't exist in distributed systems.

AP or CP?

Which is right, AP or CP? Well, the reality is *it depends*. As the people building the system, we know the trade-off exists. We know that AP systems scale more easily and are simpler to build, and we know that a CP system will require more work due to the challenges in supporting distributed consistency. But we may not understand the business impact of this trade-off. For our inventory system, if a record is out of date by five minutes, is that OK? If the answer is yes, an AP system might be the answer. But what about the balance held for a customer in a bank? Can that be out of date? Without knowing the context in which the operation is being used, we can't know the right thing to do. Knowing about the CAP theorem just helps you understand that this trade-off exists and what questions to ask.

It's Not All or Nothing

Our system as a whole doesn't need to be either AP or CP. Our catalog could be AP, as we don't mind too much about a stale record. But we might decide that our inventory service needs to be CP, as we don't want to sell a customer something we don't have and then have to apologize later.

But individual services don't even need to be CP or AP.

Let's think about our points balance service, where we store records of how many loyalty points our customers have built up. We could decide that we don't care if the balance we show for a customer is stale, but that when it comes to updating a balance we need it to be consistent to ensure that customers don't use more points than they have available. Is this microservice CP, or AP, or is it both? Really, what we have done is push the trade-offs around the CAP theorem down to individual service capabilities.

Another complexity is that neither consistency nor availability is all or nothing. Many systems allow us a far more nuanced trade-off. For example, with Cassandra I can make different trade-offs for individual calls. So if I need strict consistency, I can perform a read that blocks until all replicas have responded confirming the value is consistent, or until a specific quorum of replicas have responded, or even just a single node. Obviously, if I block waiting for all replicas to report back and one of them is unavailable, I'll be blocking for a long time. But if I am satisfied with just a simple quorum of nodes reporting back, I can accept some lack of consistency to be less vulnerable to a single replica being unavailable.

You'll often see posts about people *beating* the CAP theorem. They haven't. What they have done is create a system where some capabilities are CP, and some are AP. The

mathematical proof behind the CAP theorem holds. Despite many attempts at school, I've learned that you don't beat math.

And the Real World

Much of what we've talked about is the electronic world—bits and bytes stored in memory. We talk about consistency in an almost child-like fashion; we imagine that within the scope of the system we have created, we can stop the world and have it all make sense. And yet so much of what we build is just a reflection of the real world, and we don't get to control that, do we?

Let's revisit our inventory system. This maps to real-world, physical items. We keep a count in our system of how many albums we have. At the start of the day we had 100 copies of *Give Blood* by The Brakes. We sold one. Now we have 99 copies. Easy, right? By what happens if when the order was being sent out, someone knocks a copy of the album onto the floor and it gets stepped on and broken? What happens now? Our systems say 99, but there are 98 copies on the shelf.

What if we made our inventory system AP instead, and occasionally had to contact a user later on and tell him that one of his items is actually out of stock? Would that be the worst thing in the world? It would certainly be much easier to build, scale, and ensure it is correct.

We have to recognize that no matter how consistent our systems might be in and of themselves, they cannot know everything that happens, especially when we're keeping records of the real world. This is one of the main reasons why AP systems end up being the right call in many situations. Aside from the complexity of building CP systems, they can't fix all our problems anyway.

Service Discovery

Once you have more than a few microservices lying around, your attention inevitably turns to knowing where on earth everything is. Perhaps you want to know what is running in a given environment so you know what you should be monitoring. Maybe it's as simple as knowing where your accounts service is so that those microservices that use it know where to find it. Or perhaps you just want to make it easy for developers in your organization to know what APIs are available so they don't reinvent the wheel. Broadly speaking, all of these use cases fall under the banner of *service discovery*. And as always with microservices, we have quite a few different options at our disposal for dealing with it.

All of the solutions we'll look at handle things in two parts. First, they provide some mechanism for an instance to register itself and say, "I'm here!" Second, they provide a way to find the service once it's registered. Service discovery gets more complicated, though, when we are considering an environment where we are constantly destroying

and deploying new instances of services. Ideally, we'd want whatever solution we pick to cope with this.

Let's look at some of the most common solutions to service delivery and consider our options.

DNS

It's nice to start simple. DNS lets us associate a name with the IP address of one or more machines. We could decide, for example, that our accounts service is always found at *accounts.musiccorp.com*. We would then have that entry point to the IP address of the host running that service, or perhaps have it resolve to a load balancer that is distributing load across a number of instances. This means we'd have to handle updating these entries as part of deploying our service.

When dealing with instances of a service in different environments, I have seen a convention-based domain template work well. For example, we might have a template defined as *<servicename>-<environment>.musiccorp.com*, giving us entries like *accounts-uat.musiccorp.com* or *accounts-dev.musiccorp.com*.

A more advanced way of handling different environments is to have different domain name servers for different environments. So I could assume that *accounts.musiccorp.com* is where I always find the accounts service, but it could resolve to different hosts depending on where I do the lookup. If you already have your environments sitting in different network segments and are comfortable with managing your own DNS servers and entries, this could be quite a neat solution, but it is a lot of work if you aren't getting other benefits from this setup.

DNS has a host of advantages, the main one being it is such a well-understood and well-used standard that almost any technology stack will support it. Unfortunately, while a number of services exist for managing DNS inside an organization, few of them seem designed for an environment where we are dealing with highly disposable hosts, making updating DNS entries somewhat painful. Amazon's Route53 service does a pretty good job of this, but I haven't seen a self-hosted option that is as good yet, although (as we'll discuss shortly) Consul may help us here. Aside from the problems in updating DNS entries, the DNS specification itself can cause us some issues.

DNS entries for domain names have a *time to live* (TTL). This is how long a client can consider the entry fresh. When we want to change the host to which the domain name refers, we update that entry, but we have to assume that clients will be holding on to the old IP for *at least* as long as the TTL states. DNS entries can get cached in multiple places (even the JVM will cache DNS entries unless you tell it not to), and the more places they are cached in, the more stale the entry can be.

One way to work around this problem is to have the domain name entry for your service point to a load balancer, which in turn points to the instances of your service,

as shown in Figure 3-9. When you deploy a new instance, you can take the old one out of the load-balancer entry and add the new one. Some people use DNS round-robining, where the DNS entries themselves refer to a group of machines. This technique is extremely problematic, as the client is hidden from the underlying host, and therefore cannot easily stop routing traffic to one of the hosts should it become sick.

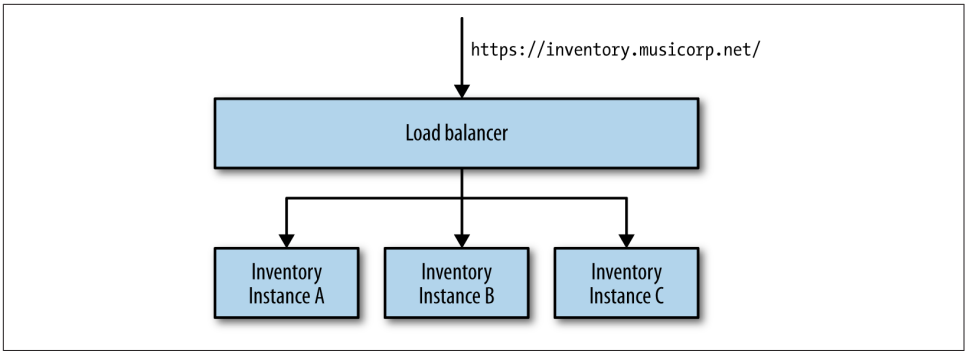


Figure 3-9. Using DNS to resolve to a load balancer to avoid stale DNS entries

As mentioned, DNS is well understood and widely supported. But it does have one or two downsides. I would suggest investigating whether it is a good fit for you before picking something more complex. For a situation where you have only single nodes, having DNS refer directly to hosts is probably fine. But for those situations where you need more than one instance of a host, have DNS entries resolve to load balancers that can handle putting individual hosts into and out of service as appropriate.

Dynamic Service Registries

The downsides of DNS as a way of finding nodes in a highly dynamic environment have led to a number of alternative systems, most of which involve the service registering itself with some central registry, which in turn offers the ability to look up these services later on. Often, these systems do more than just providing service registration and discovery, which may or may not be a good thing. This is a crowded field, so we'll just look at a few options to give you a sense of what is available.

Zookeeper

Zookeeper was originally developed as part of the Hadoop project. It is used for an almost bewildering array of use cases, including configuration management, synchronizing data between services, leader election, message queues, and (usefully for us) as a naming service.

Like many similar types of systems, Zookeeper relies on running a number of nodes in a cluster to provide various guarantees. This means you should expect to be run-

ning at least three Zookeeper nodes. Most of the smarts in Zookeeper are around ensuring that data is replicated safely between these nodes, and that things remain consistent when nodes fail.

At its heart, Zookeeper provides a hierarchical namespace for storing information. Clients can insert new nodes in this hierarchy, change them, or query them. Furthermore, they can add watches to nodes to be told when they change. This means we could store the information about where our services are located in this structure, and as a client be told when they change. Zookeeper is often used as a general configuration store, so you could also store service-specific configuration in it, allowing you to do tasks like dynamically changing log levels or turning off features of a running system. Personally, I tend to shy away from the use of systems like Zookeeper as a configuration source, as I think it can make it harder to reason about the behavior of a given service.

Zookeeper itself is fairly generic in what it offers, which is why it is used for so many use cases. You can think of it just as a replicated tree of information that you can be alerted about when it changes. This means that you'll typically build things on top of it to suit your particular use case. Luckily, client libraries exist for most languages out there.

In the grand scheme of things, Zookeeper could be considered *old* by now, and doesn't provide us that much functionality out of the box to help with service discovery compared to some of the newer alternatives. That said, it is certainly tried and tested, and widely used. The underlying algorithms Zookeeper implements are quite hard to get right. I know one database vendor, for example, that was using Zookeeper just for leader election in order to ensure that a primary node got properly promoted during failure conditions. The client felt that Zookeeper was too heavyweight and spent a long time ironing out bugs in its own implementation of the PAXOS algorithm to replace what Zookeeper did. People often say you shouldn't write your own cryptography libraries. I'd extend that by saying you shouldn't write your own distributed coordination systems either. There is a lot to be said for using existing stuff that just works.

Consul

Like Zookeeper, Consul supports both configuration management and service discovery. But it goes further than Zookeeper in providing more support for these key use cases. For example, it exposes an HTTP interface for service discovery, and one of Consul's killer features is that it actually provides a DNS server out of the box; specifically, it can serve SRV records, which give you both an IP and port for a given name. This means if part of your system uses DNS already and can support SRV records, you can just drop in Consul and start using it without any changes to your existing system.

Consul also builds in other capabilities that you might find useful, such as the ability to perform health checks on nodes. This means that Consul could well overlap the capabilities provided by other dedicated monitoring tools, although you would more likely use Consul as a source of this information and then pull it into a more comprehensive dashboard or alerting system. Consul's highly fault-tolerant design and focus on handling systems that make heavy use of ephemeral nodes does make me wonder, though, if it may end up replacing systems like Nagios and Sensu for some use cases.

Consul uses a RESTful HTTP interface for everything from registering a service, querying the key/value store, or inserting health checks. This makes integration with different technology stacks very straightforward. One of the other things I really like about Consul is that the team behind it has split out the underlying cluster management piece. Serf, which Consul sits on top of, handles detection of nodes in a cluster, failure management, and alerting. Consul then adds service discovery and configuration management. This separation of concerns appeals to me, which should be no surprise to you given the themes that run through this book!

Consul is very new, and given the complexity of the algorithms it uses, this would normally make me hesitant in recommending it for such an important job. That said, Hashicorp, the team behind it, certainly has a great track record in creating very useful open source technology (in the form of both Packer and Vagrant), the project is being actively developed, and I've spoken to a few people who are happily using it in production. Given that, I think it is well worth a look.

Eureka

Netflix's open source Eureka system bucks the trend of systems like Consul and Zookeeper in that it doesn't also try to be a general-purpose configuration store. It is actually very targeted in its use case.

Eureka also provides basic load-balancing capabilities in that it can support basic round-robin lookup of service instances. It provides a REST-based endpoint so you can write your own clients, or you can use its own Java client. The Java client provides additional capabilities, such as health checking of instances. Obviously if you bypass Eureka's own client and go directly to the REST endpoint, you're on your own there.

By having the clients deal with service discovery directly, we avoid the need for a separate process. However, you do require that every client implement service discovery. Netflix, which standardizes on the JVM, achieves this by having all clients use Eureka. If you're in a more polyglot environment, this may be more of a challenge.

Rolling Your Own

One approach I have used myself and seen done elsewhere is to roll your own system. On one project we were making heavy use of AWS, which offers the ability to add tags to instances. When launching service instances, I would apply tags to help define what the instance was and what it was used for. These allowed for some rich metadata to be associated with a given host, for example:

- service = accounts
- environment = production
- version = 154

I could then use the AWS APIs to query all the instances associated with a given AWS account to find machines I cared about. Here, AWS itself is handling the storing of the metadata associated with each instance, and providing us with the ability to query it. I then built command-line tools for interacting with these instances, and making dashboards for status monitoring becomes fairly easy, especially if you adopt the idea of having each service instance exposing health check details.

The last time I did this we didn't go as far as having services use the AWS APIs to find their service dependencies, but there is no reason why you couldn't. Obviously, if you want upstream services to be alerted when the location of a downstream service changes, you're on your own.

Don't Forget the Humans!

The systems we've looked at so far make it easy for a service instance to register itself and look up other services it needs to talk to. But as humans we sometimes want this information too. Whatever system you pick, make sure you have tools available that let you build reports and dashboards on top of these registries to create displays for humans, not just for computers.

Documenting Services

By decomposing our systems into finer-grained microservices, we're hoping to expose lots of seams in the form of APIs that people can use to do many, hopefully wonderful, things. If you get your discovery right, we know where things are. But how do we know what those things do, or how to use them? One option is obviously to have documentation about the APIs. Of course, documentation can often be out of date. Ideally, we'd ensure that our documentation is always up to date with the microservice API, and make it easy to see this documentation when we know where a service end-

point is. Two different pieces of technology, Swagger and HAL, try to make this a reality, and both are worth looking at.

Swagger

Swagger lets you describe your API in order to generate a very nice web UI that allows you to view the documentation and interact with the API via a web browser. The ability to execute requests is very nice: you can define POST templates, for example, making it clear what sort of content the server expects.

To do all of this, Swagger needs the service to expose a sidecar file matching the Swagger format. Swagger has a number of libraries for different languages that does this for you. For example, for Java you can annotate methods that match your API calls, and the file gets generated for you.

I like the end-user experience that Swagger gives you, but it does little for the incremental exploration concept at the heart of hypermedia. Still, it's a pretty nice way to expose documentation about your services.

HAL and the HAL Browser

By itself, the Hypertext Application Language (HAL) is a standard that describes standards for hypermedia controls that we expose. As we covered in Chapter 2, hypermedia controls are the means by which we allow clients to progressively explore our APIs to use our service's capabilities in a less coupled fashion than other integration techniques. If you decide to adopt HAL's hypermedia standard, then not only can you make use of a wide number of client libraries for consuming the API (at the time of writing, the HAL wiki listed 50 supporting libraries for a number of different languages), but you can also make use of the HAL browser, which gives you a way to explore the API via a web browser.

Like Swagger, this UI can be used not only to act as living documentation, but also to execute calls against the service itself. Executing calls isn't quite as slick, though. Whereas with Swagger you can define templates to do things like issue a POST request, with HAL you're more on your own. The flipside to this is that the inherent power of hypermedia controls lets you much more effectively explore the API exposed by the service, as you can follow links around very easily. It turns out that web browsers are pretty good at that sort of thing!

Unlike with Swagger, all the information needed to drive this documentation and sandbox is embedded in the hypermedia controls. This is a double-edged sword. If you are already using hypermedia controls, it takes little effort to expose a HAL browser and have clients explore your API. However, if you aren't using hypermedia, you either can't use HAL or have to retrofit your API to use hypermedia, which is likely to be an exercise that breaks existing consumers.

The fact that HAL also describes a hypermedia standard with some supporting client libraries is an added bonus, and I suspect is a big reason why I've seen more uptake of HAL as a way of documenting APIs than Swagger for those people already using hypermedia controls. If you're using hypermedia, my recommendation is to go with HAL over Swagger. But if you're using hypermedia and can't justify the switch, I'd definitely suggest giving Swagger a go.

The Self-Describing System

During the early evolution of SOA, standards like Universal Description, Discovery, and Integration (UDDI) emerged to help people make sense of what services were running. These approaches were fairly heavyweight, which led to alternative techniques to try to make sense of our systems. Martin Fowler discussed the concept of the humane registry, where a much more lightweight approach is simply to have a place where humans can record information about the services in the organization in something as basic as a wiki.

Getting a picture of our system and how it is behaving is important, especially when we're at scale. We've covered a number of different techniques that will help us gain understanding directly from our system. By tracking the health of our downstream services together with correlation IDs to help us see call chains, we can get real data in terms of how our services interrelate. Using service discovery systems like Consul, we can see where our microservices are running. HAL lets us see what capabilities are being hosted on any given endpoint, while our health-check pages and monitoring systems let us know the health of both the overall system and individual services.

All of this information is available programatically. All of this data allows us to make our humane registry more powerful than a simple wiki page that will no doubt get out of date. Instead, we should use it to harness and display all the information our system will be emitting. By creating custom dashboards, we can pull together the vast array of information that is available to help us make sense of our ecosystem.

By all means, start with something as simple as a static web page or wiki that perhaps scrapes in a bit of data from the live system. But look to pull in more and more information over time. Making this information readily available is a key tool to managing the emerging complexity that will come from running these systems at scale.

Summary

As a design approach, microservices are still fairly young, so although we have some notable experiences to draw upon, I'm sure the next few years will yield more useful patterns in handling them at scale. Nonetheless, I hope this chapter has outlined some steps you can take on your journey to microservices at scale that will hold you in good stead.

In addition to what I have covered here, I recommend Michael Nygard's excellent book *Release It!*. In it he shares a collection of stories about system failure and some patterns to help deal with it well. The book is well worth a read (in fact, I would go so far as to say it should be considered essential reading for anyone building systems at scale).

We've covered quite a lot of ground, and we're nearing the end. In our next and final chapter, we will look to pull everything back together and summarize what we have learned in the book overall.